ПРИВАТНЕ АКЦІОНЕРНЕ ТОВАРИСТВО
ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«МІЖРЕГІОНАЛЬНА АКАДЕМІЯ УПРАВЛІННЯ ПЕРСОНАЛОМ»
ІНСТИТУТ КОМП'ЮТЕРНО-ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

БАКАЛАВРСЬКА ДИПЛОМНА РОБОТА

на тему:

«Розробка веборієнтованої системи контролю строків виконання заявок та автоматизованих нагадувань»

Студент: Малєєв Артем Юрійович
Група: ІК-12-21-Б1ІПЗ(4.6з)
Спеціальність: Інженерія програмного забезпечення
Кваліфікаційний рівень: Бакалавр
Київ – 2026

ЗАВДАННЯ НА БАКАЛАВРСЬКУ РОБОТУ

Тема роботи: «Розробка веборієнтованої системи контролю строків виконання заявок та автоматизованих нагадувань».

Метою роботи є створення серверного вебзастосунку, який дозволяє реєструвати заявки, призначати відповідальних виконавців, контролювати строки виконання та автоматично формувати нагадування про наближення або порушення дедлайнів.

У процесі виконання роботи необхідно проаналізувати предметну область, визначити вимоги до системи, спроєктувати архітектуру, інформаційну модель і API, реалізувати серверну частину засобами ASP.NET Core Web API, організувати збереження даних у PostgreSQL та перевірити працездатність застосунку через Swagger UI.

Результатом роботи має бути програмний продукт, який запускається локально після клонування з GitHub-репозиторію та не потребує обов’язкового розгортання в хмарній інфраструктурі.

АНОТАЦІЯ

У бакалаврській дипломній роботі розглянуто процес розробки веборієнтованої системи контролю строків виконання заявок та автоматизованих нагадувань. Робота присвячена практичній задачі, яка виникає у багатьох організаціях: потрібно не лише створити заявку, а й не втратити її в роботі, своєчасно відстежити наближення дедлайну та зафіксувати факт прострочення.

У межах роботи проаналізовано предметну область, визначено основні проблеми ручного контролю строків, сформульовано функціональні й нефункціональні вимоги до системи. Для реалізації обрано ASP.NET Core Web API, Entity Framework Core, PostgreSQL і Swagger UI. Механізм автоматичних нагадувань реалізується за допомогою BackgroundService, що працює в межах серверного застосунку.

Практичним результатом є серверний застосунок з документованим API, який можна запустити локально. Такий варіант реалізації зменшує залежність від зовнішньої інфраструктури та дозволяє зосередитися на основній інженерній частині: моделі даних, бізнес-логіці, фоновій перевірці строків і тестуванні сценаріїв роботи із заявками.

Ключові слова: заявка, дедлайн, нагадування, ASP.NET Core, Web API, REST, PostgreSQL, Swagger, BackgroundService, Entity Framework Core, ООП, SOLID, dependency injection.

ЗМІСТ

ЗМІСТ (продовження)

ВСТУП

Актуальність теми. У роботі розглядається не абстрактний механізм таймерів, а більш практична задача: як організувати контроль строків для заявок, які створюються в межах внутрішніх процесів організації. На перший погляд така задача здається простою. Достатньо записати дедлайн у таблицю або календар, а потім час від часу перевіряти його. Проблема з'являється тоді, коли заявок стає багато, вони мають різних виконавців, змінюють статуси, а частина строків переноситься або порушується.

У реальних робочих процесах заявка часто проходить кілька етапів: її створюють, уточнюють, призначають виконавцю, виконують, перевіряють і лише після цього закривають. Якщо строк виконання контролюється вручну, частина інформації неминуче розпорошується між листуванням, нотатками, електронними таблицями або усними домовленостями. Через це складно швидко відповісти на прості питання: які заявки вже прострочені, які наближаються до дедлайну, хто зараз відповідає за виконання і коли саме змінювався статус.

Веборієнтована система дозволяє зібрати ці дані в одному місці. У межах цієї дипломної роботи система не позиціонується як великий корпоративний продукт. Це навчальний, але працездатний серверний застосунок, який демонструє основні принципи обробки заявок: збереження даних у реляційній базі, доступ через API, фіксацію історії змін і автоматичну перевірку строків. Для розробки обрано ASP.NET Core Web API, оскільки ця платформа підходить для створення серверних вебзастосунків і добре інтегрується з механізмами dependency injection, конфігурації та middleware [1; 16].

Окремо важливо зазначити, що у проєкті не передбачається обов'язкове розгортання в хмарі. Застосунок має запускатися локально після клонування з GitHub-репозиторію. Такий підхід є практичним для бакалаврської роботи: він прибирає залежність від платних сервісів і дозволяє сконцентруватися на архітектурі, моделі даних, серверній логіці та тестуванні. Для перевірки API

використовується Swagger UI, який надає інтерактивну документацію та дозволяє виконувати HTTP-запити без окремого frontend-застосунку [6; 17].

Метою роботи є розробка веборієнтованої системи контролю строків виконання заявок та автоматизованих нагадувань, яка забезпечує створення, збереження, обробку й моніторинг заявок, контроль строків їх виконання, формування нагадувань і перегляд актуального стану виконання.

Об’єктом дослідження є процес контролю строків виконання заявок у веборієнтованих інформаційних системах.

Предметом дослідження є методи, моделі, програмні засоби та архітектурні підходи до розробки веборієнтованої системи контролю строків виконання заявок і автоматизованих нагадувань.

Для досягнення мети потрібно виконати кілька послідовних завдань: проаналізувати предметну область, визначити типові проблеми ручного контролю строків, сформулювати вимоги до системи, обґрунтувати вибір технологічного стеку, спроєктувати архітектуру й модель даних, реалізувати API для роботи із заявками, додати фонову перевірку дедлайнів, а також перевірити коректність роботи основних сценаріїв.

У роботі використано методи аналізу й узагальнення для опису предметної області, системний аналіз для визначення компонентів застосунку, об’єктно-орієнтоване моделювання для виділення сутностей, проєктування баз даних для побудови інформаційної моделі, а також методи програмної інженерії та тестування програмного забезпечення.

Практичне значення роботи полягає в тому, що її результатом є не лише опис алгоритму, а працездатний серверний застосунок. Його можна використати як основу для невеликої системи обліку заявок або як демонстраційний проєкт, який показує зв’язок між API, реляційною базою даних, фоновою обробкою та журналюванням подій.

Структурно робота складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків. У першому розділі розглянуто предметну область і сформульовано постановку задачі. У другому розділі описано проєктування системи, її архітектуру, модель даних, API та логіку нагадувань. У третьому розділі подано опис реалізації, локального запуску, перевірки через Swagger і тестування основних сценаріїв.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Особливості процесу обробки заявок у сучасних організаціях

У межах цієї роботи заявкою вважається запис про звернення, доручення або внутрішній запит, який потребує фіксації в системі та подальшого контролю. Для такої заявки важливі не тільки назва й опис, а й відповідальний виконавець, пріоритет, строк виконання, поточний статус і історія змін. Без цих атрибутів заявка швидко перетворюється на звичайну нотатку, яку складно відстежувати в часі.

Заявки можуть виникати в різних процесах: технічна підтримка користувачів, обробка адміністративних звернень, внутрішні доручення між підрозділами, сервісне обслуговування або контроль виконання невеликих робочих завдань. Незалежно від конкретного домену, логіка роботи з ними зазвичай схожа. Спочатку заявку створюють, потім уточнюють її зміст, призначають виконавця, визначають строк виконання, виконують потрібні дії та закривають після перевірки результату.

На практиці найбільша складність полягає не в самому створенні заявки. Значно важче підтримувати її актуальний стан. Наприклад, виконавець може змінитися, дедлайн може бути перенесений, статус може перейти з «Нова» до «У роботі», а потім до «Очікує уточнення». Якщо такі зміни не фіксуються, пізніше майже неможливо встановити, чому заявка була виконана із затримкою або хто відповідав за неї на певному етапі.

Саме тому в розроблюваній системі заявка розглядається як об’єкт із життєвим циклом. Це означає, що система повинна не лише зберігати поточні поля, а й дозволяти відстежувати події, які відбувалися із заявкою. До таких подій належать створення, зміна статусу, призначення виконавця, зміна дедлайну, створення нагадування та закриття.

Окремий акцент у роботі зроблено на контролі строків. Строк виконання є тим параметром, який змінює звичайний облік заявок на процес управління виконанням. Якщо система бачить, що дедлайн наближається, вона повинна створити нагадування. Якщо строк уже порушено, потрібно зафіксувати прострочення та зробити його видимим через API. Така поведінка зменшує ризик того, що відповідальний користувач просто не помітить критичний момент.

Для демонстрації роботи системи не створюється окремий вебінтерфейс. Замість цього використовується Swagger UI. Таке рішення є свідомим: головною частиною дипломного проєкту є серверна логіка, модель даних і механізм автоматизованих нагадувань. Swagger дозволяє перевірити всі основні сценарії через браузер і водночас слугує документацією до API [6; 17].

Рис. 1.1 – Узагальнений життєвий цикл заявки

На рис. 1.1 показано спрощений життєвий цикл заявки. У реальній системі частина етапів може повторюватися: заявка може кілька разів повертатися на уточнення, дедлайн може переноситися, а виконавець може змінюватися. Для цієї дипломної роботи важливо, що кожна така дія повинна бути відображена в даних системи.

Таблиця 1.1 – Основні атрибути заявки у межах розроблюваної системи

Атрибут	Зміст	Практичне призначення
Назва	Коротке формулювання суті заявки	Дозволяє швидко ідентифікувати запис у списку
Опис	Детальніше пояснення проблеми або запиту	Дає виконавцю контекст для роботи
Статус	Поточний стан заявки	Показує, на якому етапі перебуває виконання
Пріоритет	Оцінка важливості або терміновості	Допомагає визначати порядок обробки
Дедлайн	Дата і час, до яких заявку потрібно виконати	Використовується для контролю строків і нагадувань
Виконавець	Користувач, відповідальний за виконання	Закріплює відповідальність за результат

Наведені атрибути не є надмірними для навчального проєкту. Навпаки, вони задають мінімальний набір даних, без якого неможливо коректно перевірити логіку контролю строків. Якщо прибрати дедлайн, система втратить основну функцію; якщо не зберігати статуси, буде незрозуміло, які заявки потрібно перевіряти; якщо не фіксувати виконавця, нагадування не матиме адресата навіть у внутрішній моделі.

1.2. Проблеми ручного контролю строків виконання

Ручний контроль строків часто починається як просте й зручне рішення. Для кількох заявок достатньо електронної таблиці або короткого списку задач. Але такий підхід погано масштабується. Щойно з’являються різні виконавці, кілька пріоритетів і десятки активних записів, таблиця перестає бути надійним інструментом контролю.

Перша типова проблема — розпорошеність інформації. Частина даних може зберігатися в таблиці, частина в електронній пошті, частина в чатах, а рішення про перенесення строку може взагалі залишитися в усній домовленості. У результаті

немає єдиного джерела правди. Коли потрібно швидко перевірити стан заявки, доводиться шукати інформацію в кількох місцях.

Друга проблема пов’язана з людським фактором. Навіть якщо працівники відповідально ставляться до задач, вони можуть пропустити дедлайн через завантаженість, велику кількість паралельних звернень або відсутність централізованого нагадування. Система не повинна повністю замінювати відповідальність виконавця, але вона може зменшити ймовірність випадкового пропуску.

Третя проблема — слабка історичність. Якщо статуси змінюються вручну без журналу, складно зрозуміти, як заявка рухалася в процесі. Наприклад, заявка могла бути прострочена не через виконавця, а через те, що довго очікувала додаткову інформацію від ініціатора. Без історії подій такі нюанси втрачаються.

Четверта проблема — відсутність швидкої аналітики. Керівнику або відповідальній особі може бути потрібно побачити, скільки заявок зараз у роботі, скільки прострочено, які пріоритети переважають і які виконавці мають найбільше відкритих задач. У ручному режимі така інформація або не збирається взагалі, або потребує додаткової підготовки.

У межах цієї дипломної роботи зазначені проблеми не просто описуються, а перетворюються на вимоги до системи. Якщо інформація розпорошена, потрібна єдина база даних. Якщо дедлайни пропускаються, потрібен фоновий механізм перевірки. Якщо немає історії, треба зберігати події. Якщо складно перевірити API, потрібна інтерактивна документація через Swagger.

Таблиця 1.2 – Проблеми ручного контролю та відповідь системи

Проблема	**Наслідок**	**Рішення у межах проєкту**
Розпорошеність даних	Складно знайти актуальний стан заявки	Зберігання заявок і пов’язаних подій у PostgreSQL
Пропуск дедлайнів	Заявки виконуються із запізненням	Періодична перевірка строків через BackgroundService

Проблема	Наслідок	Рішення у межах проєкту
Відсутність історії	Незрозуміло, хто і коли змінив заявку	Окрема сутність історії змін
Неактуальні статуси	Користувачі бачать неточну картину роботи	Чіткий набір статусів і API для зміни стану
Складність перевірки	Потрібен окремий інтерфейс або ручні запити	Swagger UI для демонстрації endpoint-ів

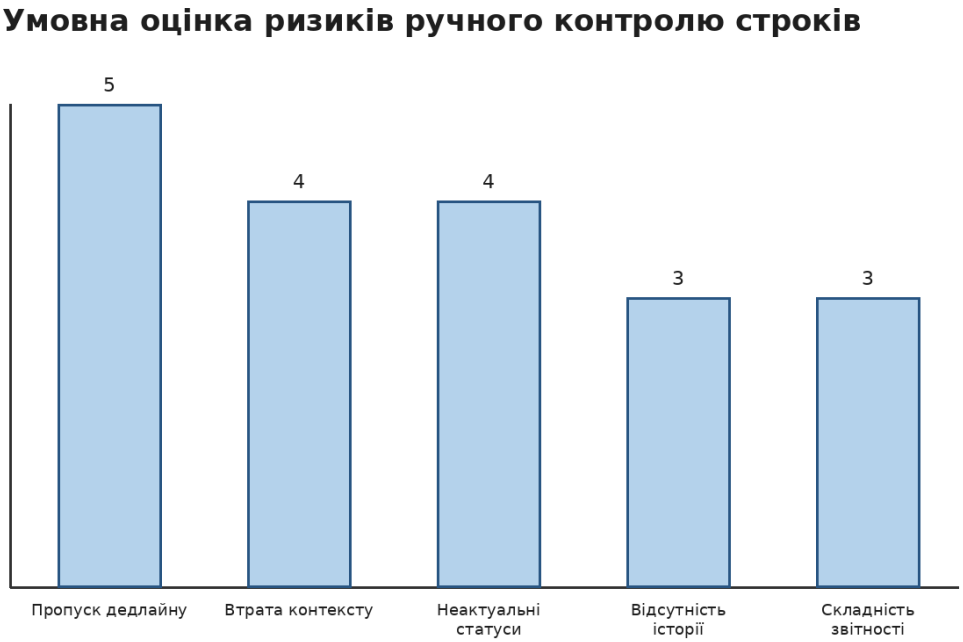

Рис. 1.2 – Умовна оцінка ризиків ручного контролю строків

Рисунок 1.2 має ілюстративний характер. Він не подається як результат статистичного дослідження, а використовується для пояснення того, які ризики є найбільш важливими саме для цієї роботи. Найвищий пріоритет має ризик пропуску дедлайну, оскільки він безпосередньо пов’язаний з темою дипломної роботи.

1.3. Аналіз існуючих підходів та програмних рішень

Для роботи із заявками й задачами існує багато готових інструментів. До них можна віднести системи керування проєктами, help desk-рішення, CRM-системи та прості task management-застосунки. Вони відрізняються масштабом, набором

функцій і складністю впровадження, але більшість із них так чи інакше працює з поняттями задачі, відповідального, статусу і строку.

Наприклад, системи на зразок Jira або YouTrack підходять для командної роботи над програмними продуктами. Вони мають розвинені workflow, дошки, ролі, фільтри, звіти та інтеграції. Але для бакалаврського проєкту повторювати таку систему немає сенсу. Завдання роботи інше: виділити невелике ядро функціональності та показати, як його можна реалізувати власними засобами.

Help desk-системи більше орієнтовані на обробку звернень користувачів. Вони корисні там, де є потік заявок від клієнтів або працівників, а також потрібне SLA, пріоритети й контроль часу реакції. У цій роботі ідея запозичується частково: заявка має життєвий цикл і дедлайн, але система не претендує на повноцінний help desk із каналами комунікації, поштовими інтеграціями або складними ролями.

Task management-застосунки на кшталт Trello або Asana зручні для візуального планування. Вони добре показують задачі на дошках і дозволяють переміщати їх між колонками. Однак їхня логіка зазвичай більше орієнтована на взаємодію через інтерфейс користувача. У цій роботі акцент зроблено на серверній частині: API, базі даних, фоновій перевірці та тестуванні.

Окремо варто згадати, що готові системи часто є надлишковими для невеликого сценарію. Якщо організації потрібен тільки облік заявок із дедлайнами й нагадуваннями, впровадження великої платформи може бути невиправданим. У такому випадку легший власний сервіс або модуль може бути достатнім. Саме такий підхід використано у дипломному проєкті.

Під час аналізу готових рішень важливо не копіювати їхню структуру, а визначити базові функції, без яких система контролю заявок не буде корисною. До таких функцій належать створення заявки, призначення виконавця, встановлення дедлайну, зміна статусу, перегляд прострочених заявок і формування нагадувань.

Таблиця 1.3 – Порівняння підходів до контролю заявок

Підхід	Переваги	Обмеження для цієї роботи
Електронна таблиця	Швидкий старт, не потребує розробки	Немає автоматичних нагадувань, слабка історія змін
Календар	Зручно бачити дати	Не зберігає повний життєвий цикл заявки
Task management-система	Зручний інтерфейс і дошки	Може бути надмірною для демонстрації backend-логіки
Help desk-система	Підтримує звернення, SLA, ролі	Складна для повного відтворення у бакалаврській роботі
Власний API-сервіс	Контрольована архітектура, зрозумілий обсяг MVP	Потребує самостійного проєктування й реалізації

Після такого порівняння стає зрозуміло, чому для роботи обрано саме власний серверний застосунок. Він дозволяє сфокусуватися на ключовій інженерній задачі — зв’язати облік заявок, контроль строків і автоматичні нагадування в одну узгоджену модель.

1.4. Формування вимог до веборієнтованої системи

Вимоги до системи сформовано з урахуванням того, що проєкт має бути достатньо реалістичним, але не надмірним за обсягом. Він повинен демонструвати повний шлях роботи із заявкою, але не перетворюватися на велику корпоративну платформу. Тому в межах MVP залишено ті функції, які безпосередньо пов’язані з темою роботи.

Функціональні вимоги описують, що саме система повинна робити. До них належать створення заявки, перегляд списку заявок, отримання деталей конкретної заявки, зміна статусу, призначення виконавця, робота з дедлайнами, створення нагадувань, перегляд історії змін і перевірка прострочених записів. Розгорнута матриця вимог винесена в додатки, щоб не перевантажувати основний текст великими таблицями.

Нефункціональні вимоги визначають, якою має бути система. Для цієї роботи важливими є простота локального запуску, зрозуміла структура проєкту, надійне збереження даних, читабельність коду, можливість перевірки API через Swagger і відсутність обов'язкової залежності від хмарних сервісів. Останній пункт є принциповим, оскільки система повинна працювати в умовах студентського захисту без ризику, що зовнішній сервіс стане недоступним.

Вимога локального запуску впливає на вибір технологій. PostgreSQL можна розгорнути локально або через Docker, Entity Framework Core дозволяє керувати міграціями, а ASP.NET Core Web API запускається як звичайний серверний застосунок. Такий набір технологій достатній для демонстрації системи й водночас не вимагає окремої cloud-інфраструктури [1; 3; 4; 18].

Оскільки frontend не розробляється, частина вимог переноситься на якість API. Endpoint-и повинні мати зрозумілі маршрути, приймати структуровані DTO, повертати передбачувані коди відповідей і коректно обробляти помилки. Swagger UI у цьому випадку виконує роль і документації, і демонстраційного інтерфейсу [6; 17].

Таблиця 1.4 – Узагальнений перелік ключових вимог

Група вимог	**Зміст**
Робота із заявками	Створення, перегляд, редагування, зміна статусів і дедлайнів
Контроль строків	Виявлення заявок із наближенням дедлайну та прострочених заявок
Нагадування	Автоматичне створення записів-нагадувань у визначених ситуаціях
Історія змін	Фіксація важливих подій, пов'язаних із заявкою
Локальний запуск	Запуск проєкту після клонування з GitHub без cloud deployment
Документування API	Використання Swagger UI для перевірки й опису endpoint-ів

Такі вимоги є достатніми для бакалаврської роботи, оскільки вони охоплюють повний цикл: від аналізу предметної області до реалізації та перевірки

працездатного програмного продукту. Водночас у них немає зайвої функціональності, яка могла б ускладнити розробку без прямої користі для теми.

1.5. Постановка задачі дипломної роботи

На основі проведеного аналізу задача дипломної роботи формулюється так: необхідно розробити веборієнтовану систему, яка дозволяє працювати із заявками, контролювати строки їх виконання та автоматично формувати нагадування у випадку наближення або порушення дедлайну.

Система має бути реалізована як серверний застосунок із REST API. Такий підхід дає змогу відокремити бізнес-логіку від способу взаємодії користувача із системою. У межах роботи взаємодія здійснюється через Swagger UI, але в майбутньому до того самого API можна підключити повноцінний frontend або інтегрувати його з іншими сервісами.

Вхідними даними системи є інформація про заявку: назва, опис, пріоритет, статус, дедлайн і виконавець. Додатково система працює з даними про користувачів, нагадування, історію змін і системні події. Вихідними даними є список заявок, детальна інформація про конкретну заявку, перелік нагадувань, записи історії та результати виконання API-запитів.

У межах реалізації потрібно забезпечити збереження даних у PostgreSQL, роботу з базою через Entity Framework Core, документування API через Swagger, а також фонову перевірку строків за допомогою BackgroundService. Кодові фрагменти винесено до додатків, оскільки за своїм характером вони є допоміжним матеріалом до основного тексту.

Очікуваним результатом є локально працездатний застосунок, який можна запустити після клонування репозиторію. Успішність реалізації перевіряється через набір тестових сценаріїв: створення заявки, зміна статусу, перевірка дедлайну, створення нагадування, фіксація прострочення та перегляд історії змін.

Висновки до розділу 1. У першому розділі було уточнено зміст поняття заявки, описано типовий життєвий цикл її обробки, визначено проблеми ручного контролю строків і порівняно можливі підходи до розв’язання задачі. На основі цього сформульовано вимоги до системи та визначено постановку задачі дипломної роботи. Подальший розділ присвячено проєктуванню архітектури, моделі даних, API та логіки автоматизованих нагадувань.

РОЗДІЛ 2

ПРОЄКТУВАННЯ ВЕБОРІЄНТОВАНОЇ СИСТЕМИ КОНТРОЛЮ СТРОКІВ ВИКОНАННЯ ЗАЯВОК

2.1. Обґрунтування вибору технологічного стеку

Технологічний стек для дипломного проєкту обирався не за принципом максимальної складності, а за принципом достатності. Система повинна мати серверне API, реляційну базу даних, засіб доступу до даних, механізм фонової перевірки строків і зручний спосіб перевірити endpoint-и без окремого frontend. З цих причин обрано ASP.NET Core Web API, Entity Framework Core, PostgreSQL, BackgroundService і Swagger UI.

ASP.NET Core Web API використовується як основа серверної частини. Платформа підтримує контролери, dependency injection, middleware, конфігурацію, логування і добре підходить для побудови HTTP API [1; 2]. Для цієї роботи важливо, що застосунок можна запускати локально, а структура проєкту залишається зрозумілою навіть без додаткової інфраструктури.

Entity Framework Core обрано як інструмент доступу до даних. Він дозволяє працювати з моделями предметної області, описувати зв’язки між сутностями, створювати міграції та виконувати LINQ-запити до бази даних [3]. Для бакалаврської роботи це зручно, оскільки модель даних можна показати і на рівні діаграми, і на рівні класів.

Як систему керування базами даних обрано PostgreSQL. Вона є безкоштовною, поширеною, стабільною і достатньо простою для локального розгортання. Для демонстраційного проєкту PostgreSQL можна встановити безпосередньо на комп’ютер або запустити через Docker. У роботі PostgreSQL використовується для зберігання заявок, користувачів, нагадувань, історії змін і системних подій [4; 18].

Swagger UI застосовується замість окремого frontend. Це не означає, що система не є веборієнтованою. Її основою все одно є HTTP API, а Swagger дає змогу взаємодіяти з ним через браузер, переглядати доступні endpoint-и, виконувати запити та аналізувати відповіді [6; 17]. Для дипломного MVP це дозволяє не витрачати час на інтерфейс, який не є центральним предметом роботи.

BackgroundService використовується для автоматичного контролю строків. Це вбудований механізм ASP.NET Core для виконання фонових задач у межах застосунку [7]. У даному випадку він періодично перевіряє активні заявки й створює нагадування, якщо дедлайн наближається або вже порушений. Без такого механізму нагадування довелося б формувати лише після ручного виклику API, що суперечило б ідеї автоматизації.

Таблиця 2.1 – Обґрунтування вибору основних технологій

Компонент	Обрана технологія	Причина вибору
Серверна частина	ASP.NET Core Web API	Підтримка REST API, DI, middleware, локального запуску
Доступ до даних	Entity Framework Core	Міграції, LINQ, зв’язок між моделями й таблицями
База даних	PostgreSQL	Безкоштовність, простий локальний запуск, реляційна модель
Документація API	Swagger / OpenAPI	Інтерактивна перевірка API без окремого frontend
Фонові задачі	BackgroundService	Автоматична перевірка строків у межах застосунку

Усі обрані технології добре поєднуються між собою. Це важливо для дипломного проєкту, оскільки надмірна кількість незалежних сервісів могла б ускладнити локальний запуск і захист роботи. Обраний стек дозволяє зберегти баланс між практичною цінністю й контрольованою складністю.

2.2. Архітектура системи

Архітектура системи побудована за шаровим принципом. На верхньому рівні знаходиться API-шар, який приймає HTTP-запити та повертає відповіді. Нижче

розташований шар сервісів, де зосереджена бізнес-логіка: створення заявок, зміна статусів, перевірка дедлайнів, формування історії та нагадувань. Окремий шар доступу до даних відповідає за взаємодію з PostgreSQL через Entity Framework Core.

Такий поділ обрано для того, щоб контролери не містили складної логіки. Контролер має прийняти запит, викликати відповідний сервіс і повернути результат. Якщо ж у контролерах розміщувати перевірку дедлайнів, роботу з історією та створення нагадувань, код швидко стане важким для підтримки. Тому логіка винесена в сервіси.

Фоновий компонент DeadlineReminderBackgroundService працює паралельно з API, але не зберігає довгоживучий AppDbContext. Він викликає DeadlineReminderProcessor, який для кожного циклу створює окремий dependency-injection scope і отримує новий контекст даних. Це важливо через різний життєвий цикл компонентів: BackgroundService існує протягом усього часу роботи застосунку, тоді як DbContext призначений для короткої одиниці роботи.

База даних є центральним сховищем стану. У ній зберігаються не тільки заявки, а й нагадування, історія змін і системні події. Це дозволяє після кожної операції відновити стан заявки та пояснити, чому вона отримала певний статус або нагадування.

Оскільки система запускається локально, архітектура не включає окремі cloud-компоненти. Це спрощує демонстрацію, але не обмежує майбутній розвиток. За потреби той самий Web API можна розгорнути в контейнері або на хмарній платформі, а PostgreSQL замінити на керований сервіс бази даних.

Рис. 2.1 – Архітектура веборієнтованої системи

На рис. 2.1 показано основні компоненти системи. Swagger UI використовується для взаємодії з API, але не містить бізнес-логіки. Уся логіка роботи із заявками зосереджена у сервісах. BackgroundService звертається до сервісу перевірки строків, а доступ до бази даних виконується через Entity Framework Core.

Таблиця 2.2 – Відповідальність компонентів системи

Компонент	Відповідальність
Swagger UI	Надання інтерактивної документації та виконання тестових запитів
API Controllers	Прийом HTTP-запитів, маршрутизація, повернення відповідей
Application Services	Виконання бізнес-операцій із заявками та нагадуваннями
BackgroundService	Періодична автоматична перевірка дедлайнів
EF Core / DbContext	Робота з таблицями, транзакціями та міграціями
PostgreSQL	Постійне збереження даних системи

2.3. Проєктування функціональної моделі системи

Функціональна модель системи описує, які дії може виконувати користувач і які операції система виконує автоматично. У цьому проєкті ролі зроблено простими, оскільки головна мета роботи — не складна авторизація, а контроль строків заявок. У мінімальному варіанті можна виділити ініціатора заявки, виконавця та адміністратора.

Ініціатор створює заявку й описує проблему або запит. Виконавець працює із заявкою, змінює її статус і, за потреби, уточнює дані через редагування основних полів. Адміністратор або відповідальна особа може переглядати всі заявки, аналізувати прострочення та контролювати стан нагадувань. У межах локального MVP ці ролі представлені спрощено, але вони потрібні для пояснення предметної області.

Основний сценарій починається зі створення заявки. Користувач передає назву, опис, пріоритет, дедлайн і виконавця. API перевіряє коректність даних, сервіс створює запис у базі, а також додає подію в історію. Після цього заявка з’являється у списку активних записів.

Другий важливий сценарій — зміна статусу. Наприклад, заявка переходить зі стану «Нова» до «У роботі», потім до «Виконана» і «Закрита». Кожна така зміна має бути записана в історію. Це потрібно не для формальності, а для можливості пояснити шлях заявки.

Третій сценарій виконується автоматично. BackgroundService періодично перевіряє активні заявки. Якщо дедлайн наближається, створюється нагадування типу DeadlineApproaching. Якщо строк уже минув, створюється нагадування типу Overdue і заявка може отримати ознаку простроченої. Цей сценарій не залежить від того, чи відкрив хтось список заявок у Swagger.

Flow обробки запиту щодо заявки

Рис. 2.2 – Flow обробки запиту щодо заявки

Flow формування автоматизованого нагадування

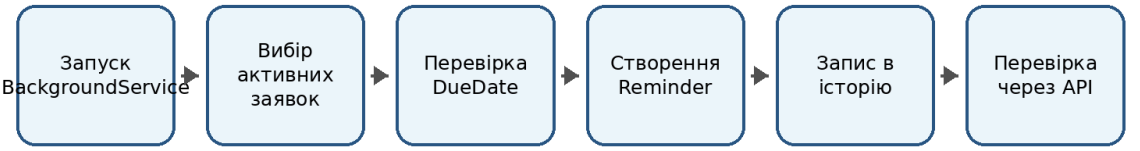

Рис. 2.3 – Flow формування автоматизованого нагадування

Рис. 2.2 і рис. 2.3 показують різницю між звичайною API-операцією та автоматичною фоновою операцією. У першому випадку ініціатором є користувач, який виконує HTTP-запит. У другому випадку ініціатором є сам застосунок, а саме BackgroundService.

2.4. Проєктування інформаційної моделі

Інформаційна модель фактичної системи побудована навколо чотирьох сутностей: AppUser, Ticket, TicketHistory і Reminder. Центральною є Ticket, у якій зберігається поточний стан заявки. Історія змін та нагадування винесені в окремі таблиці, тому поточне представлення не перевантажується, а попередні події не втрачаються.

Сутність AppUser описує користувача, який створює заявку, виконує її або вносить зміни. У поточному MVP повноцінна автентифікація не реалізована, тому користувачі використовуються як довідникові записи. Це дозволяє зберегти зв’язки між заявками та відповідальними особами, не розширюючи диплом окремим модулем ідентифікації.

Сутність Ticket містить назву, опис, статус, пріоритет, час створення й оновлення, дедлайн, автора та виконавця. Навігаційні колекції History і Reminders відображають зв’язки один-до-багатьох. Ознака прострочення не зберігається окремим полем, а обчислюється у відповіді API на основі DueAt і фінального статусу, тому не виникає ризику розбіжності між датою та окремим прапорцем.

Reminder описує сформоване системою нагадування. Запис містить тип DeadlineApproaching або Overdue, статус Sent чи Read, текст повідомлення, час створення, відправлення та прочитання. Для однієї заявки допускається не більше одного нагадування кожного типу; це правило закріплене унікальним індексом за TicketId і Type.

TicketHistory зберігає події життєвого циклу заявки: створення, зміну назви, опису, пріоритету, дедлайну, статусу або виконавця. Для події можна зафіксувати старе й нове значення, час і користувача, який ініціював зміну.

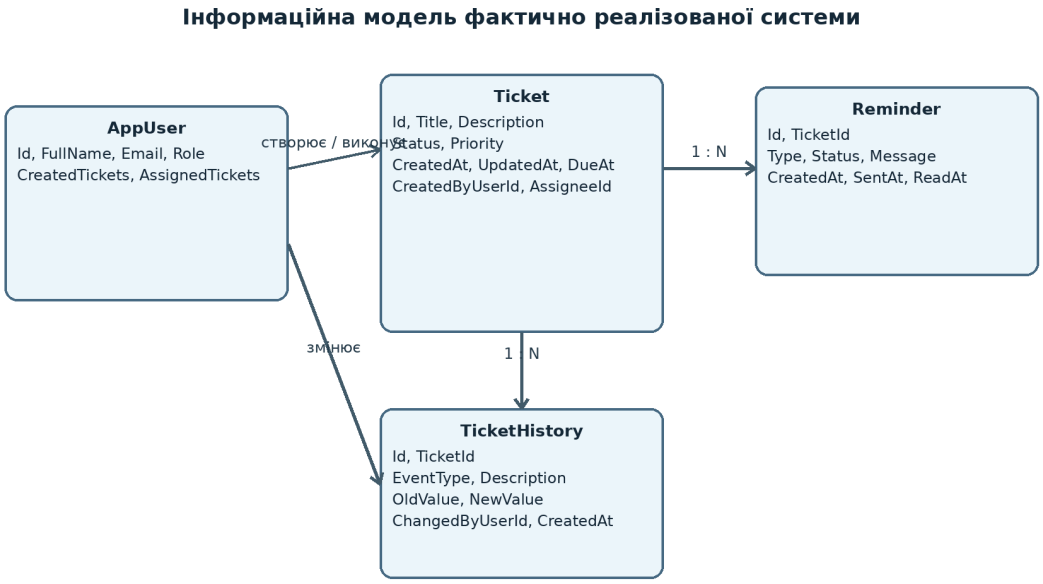

Рис. 2.4 – Фактична інформаційна модель системи

Таблиця 2.3 – Основні сутності інформаційної моделі

Сутність	**Призначення**	**Ключові зв'язки**
AppUser	Користувач, автор зміни або виконавець	Створює заявки; може бути виконавцем; пов'язаний з історією
Ticket	Поточний стан заявки	Належить автору; може мати виконавця; має історію й нагадування
TicketHistory	Журнал значущих змін	Належить Ticket; може посилатися на AppUser
Reminder	Нагадування про наближення або порушення строку	Належить Ticket; унікальне за TicketId + Type

Проєктування інформаційної моделі є важливим етапом, оскільки помилки на цьому рівні важко виправляти пізніше. Якщо, наприклад, не передбачити окрему історію змін, доведеться або втрачати важливі події, або зберігати їх у неструктурованому вигляді. Обрана модель дає достатню гнучкість для навчального проєкту й залишає можливість майбутнього розширення.

2.5. Проєктування логіки контролю строків і формування нагадувань

Логіка нагадувань є центральною частиною дипломної роботи. Вона визначає, коли система повинна реагувати на дедлайн і який запис має бути створено. У найпростішому випадку можна було б перевіряти строки лише під час відкриття списку заявок. Проте такий варіант не є повністю автоматизованим, тому в роботі використовується фоновий сервіс.

DeadlineReminderBackgroundService запускає першу перевірку одразу після старту host-a, а потім повторює її через інтервал, заданий у конфігурації. У демонстраційній версії CheckIntervalSeconds дорівнює 10 секундам, а NotifyBeforeMinutes — 5 хвилинам. Короткі значення обрані для тестування: на захисті не потрібно чекати годину або добу, щоб побачити результат.

Активні заявки читаються без відстеження змін за допомогою AsNoTracking і проєктуються у компактний внутрішній набір полів. Якщо DueAt уже настав, процесор обирає тип Overdue. Якщо строк ще не минув, але потрапляє до попереджувального вікна, формується DeadlineApproaching. Після створення або зміни заявки той самий процесор може перевірити один Ticket негайно, не очікуючи наступного фонового циклу.

Захист від дублювання реалізовано на кількох рівнях. Процесор перевіряє наявні пари TicketId і ReminderType перед додаванням запису, а в PostgreSQL створено унікальний індекс за цими полями. Паралельні запуски координуються через ProcessingLock, тому ручний endpoint і фоновий цикл не повинні одночасно створити однакові нагадування.

У майбутньому логіку можна розширити: додати різні правила для різних пріоритетів, кілька рівнів нагадувань, відправку повідомлень електронною поштою або інтеграцію з месенджерами. Проте для цієї дипломної роботи достатньо реалізувати внутрішні нагадування, які зберігаються в базі й доступні через API.

Таблиця 2.4 – Правила формування нагадувань

Умова	**Тип нагадування**	**Дія системи**
DueAt ≤ поточному UTC-часу	Overdue	Створити нагадування, якщо такого типу ще немає
now < DueAt ≤ now + NotifyBeforeMinutes	DeadlineApproaching	Створити попереджувальне нагадування
Статус Completed, Closed або Cancelled	Не створюється	Не включати заявку до активної вибірки
Пара TicketId + Type вже існує	Не створюється повторно	Пропустити; унікальний індекс захищає БД
Заявку створено або змінено	Негайна перевірка	Викликати ProcessTicketAsync

Такі правила роблять систему достатньо простою для реалізації, але водночас наближеною до реальних процесів. У ній є не тільки поле DueAt, а й поведінка, яка залежить від цього поля. Саме це відрізняє проєкт від звичайного CRUD-застосунку.

2.6. Проєктування API системи

API системи має бути зрозумілим і передбачуваним. Оскільки в роботі не розробляється окремий frontend, саме API стає основним способом взаємодії з програмним продуктом. Тому endpoint-и повинні мати логічні маршрути, використовувати стандартні HTTP-методи та повертати зрозумілі відповіді.

Фактичний API поділено на ресурси tickets, reminders, users і dictionaries. Для заявок реалізовано створення, отримання списку з фільтрами, отримання за ідентифікатором, оновлення основних полів, окрему зміну статусу, призначення виконавця та перегляд історії. Для нагадувань доступні читання, фільтрація, позначення прочитаними, ручний запуск процесора та отримання його діагностичного стану.

DTO-моделі відокремлюють зовнішній контракт API від внутрішніх сутностей бази даних. Це потрібно для того, щоб не віддавати зайві поля та не дозволяти користувачу змінювати те, що повинно контролюватися системою. Наприклад, ідентифікатор, дата створення або частина історії формуються сервером, а не передаються клієнтом.

Swagger UI використовується для перегляду контрактів API та виконання тестових запитів. Це зручно для захисту, тому що можна показати повний сценарій без Postman або frontend: створити заявку, змінити її статус, переглянути список, дочекатися нагадування й перевірити історію [6; 17].

Таблиця 2.5 – Основні endpoint-и API

Метод	**Маршрут**	**Призначення**
GET	/api/tickets	Список із фільтрами status, priority, assigneeId, onlyOverdue, onlyDueSoon
GET	/api/tickets/{id}	Отримання заявки за GUID
POST	/api/tickets	Створення заявки; 201 Created
PUT	/api/tickets/{id}	Оновлення основних редагованих полів
PATCH	/api/tickets/{id}/status	Зміна статусу
PATCH	/api/tickets/{id}/assign	Призначення виконавця
GET	/api/tickets/{id}/history	Історія змін заявки
GET	/api/reminders	Перегляд і фільтрація нагадувань
PATCH	/api/reminders/{id}/mark-as-read	Позначення нагадування прочитаним
POST	/api/reminders/process	Ручний запуск обробки дедлайнів
GET	/api/reminders/processing-status	Діагностика фонового процесу

2.7. Об’єктно-орієнтована модель і застосування принципів ООП

Мова C# і платформа .NET орієнтовані на об’єктно-орієнтовану розробку. У цьому проєкті ООП проявляється не через складну ієрархію типів, а через розподіл системи на об’єкти з чіткою відповідальністю. Ticket, Reminder, TicketHistory і AppUser представляють стан предметної області, контролери приймають HTTP-запити, сервіси виконують сценарії, а AppDbContext координує роботу з даними. Для невеликого застосунку така композиція є зрозумілішою за глибоке успадкування [26].

Абстракція використовується через ITicketService та IReminderService. Контролер знає доступний набір операцій, але не залежить від конкретних LINQ-запитів або структури таблиць PostgreSQL. Інкапсуляція проявляється в тому, що допоміжні операції TicketService, зокрема перевірка користувачів,

формування історії та перетворення сутностей у DTO, залишаються всередині сервісу.

Поліморфізм забезпечує dependency injection: TicketsController отримує ITicketService, а конкретний TicketService підставляється контейнером під час запуску. За потреби його можна замінити тестовою реалізацією без зміни контролера. Інтерфейси в C# визначають поведінковий контракт, який можуть реалізовувати різні типи [27].

Успадкування використане лише там, де воно природно випливає з фреймворку. Контролери успадковуються від ControllerBase, AppDbContext — від DbContext, а DeadlineReminderBackgroundService — від BackgroundService. Для сутностей предметної області спільний базовий клас не вводився, оскільки формальна ієрархія не дала б практичної користі.

Таблиця 2.6 – Застосування принципів ООП у проєкті

Принцип	**Реалізація у проєкті**	**Практичний результат**
Абстракція	ITicketService, IReminderService, DTO-контракти	Контролери не залежать від деталей EF Core
Інкапсуляція	Приватні допоміжні методи сервісів	Правила не розпорошуються по контролерах
Поліморфізм	Підстановка реалізацій через DI	Реалізацію можна замінити без зміни клієнтського коду
Успадкування	ControllerBase, DbContext, BackgroundService	Повторне використання поведінки фреймворку
Композиція	Сервіси використовують AppDbContext і процесор	Система складається з невеликих компонентів

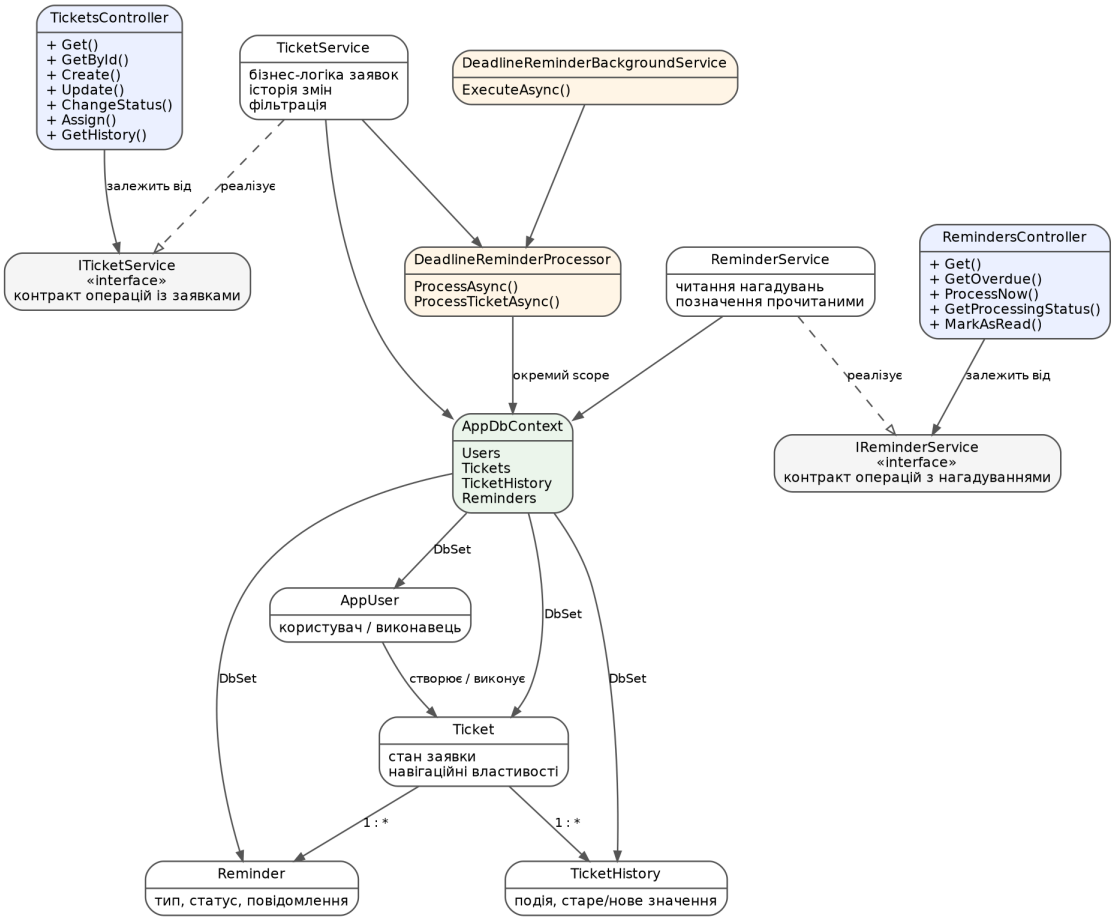

Рис. 2.5 – Основні класи, інтерфейси та залежності реалізованої системи

На рис. 2.5 видно, що доменні сутності не взаємодіють із контролерами напряму. HTTP-рівень залежить від сервісних контрактів, сервіси використовують AppDbContext, а фоновий компонент звертається до окремого процесора. Схема відповідає фактичній структурі репозиторію.

2.8. Принципи SOLID, архітектурні підходи та шаблони

Принципи SOLID у роботі використані як практичні орієнтири. Найпомітнішим є принцип єдиної відповідальності. TicketsController відповідає за HTTP-рівень, TicketService — за сценарії роботи із заявками, ReminderService — за читання та зміну стану нагадувань, DeadlineReminderProcessor — за правила формування нагадувань, а AppDbContext — за відображення об’єктів на реляційну модель.

Принцип відкритості/закритості підтримується завдяки сервісним контрактам і конфігурації. Нову реалізацію сервісу можна зареєструвати через DI без зміни контролерів. Водночас у MVP не створювалися абстракції наперед для кожної можливої зміни. Наприклад, окремий Repository поверх DbContext не додано, оскільки EF Core вже виконує роль одиниці роботи та надає операції над наборами сутностей.

Принцип підстановки Лісков стосується сервісних інтерфейсів: будь-яка реалізація ITicketService повинна зберігати очікувану семантику методів. Принцип розділення інтерфейсів дотримано тим, що операції із заявками й нагадуваннями не об’єднані в один великий контракт. RemindersController не отримує залежність від методів, які йому не потрібні.

Принцип інверсії залежностей реалізовано через конструкторну ін’єкцію. Контролери залежать від ITicketService та IReminderService, а не створюють конкретні об’єкти самостійно. Зіставлення інтерфейсів із реалізаціями виконується в Program.cs. Вбудований DI-контейнер ASP.NET Core також керує життєвими циклами Scoped і Singleton [8; 13].

З погляду загальної архітектури проєкт є модульним монолітом із шаровим розподілом. Усі компоненти працюють в одному процесі, що спрощує локальний запуск, але логічно розділені на Controllers, Services, Domain, Infrastructure і BackgroundServices. У проєкті застосовано Service Layer, DTO, Options pattern, фоновий Worker та Unit of Work через AppDbContext [12].

Таблиця 2.7 – Відображення принципів SOLID у структурі застосунку

Принцип	**Приклад у коді**	**Практичний зміст**
S	Контролери, сервіси, процесор і DbContext мають різні ролі	Зміна маршруту не вимагає переписування дедлайнів
O	Сервісні контракти та Options	Нові реалізації додаються через DI
L	TicketService виконує контракт ITicketService	Альтернативна реалізація зберігає поведінку методів
I	ITicketService та IReminderService розділені	Кожен контролер залежить від потрібних операцій
D	Конструкторна ін’єкція інтерфейсів	Високорівневі компоненти не створюють залежності через new

Таблиця 2.8 – Архітектурні підходи та шаблони, використані у проєкті

Підхід або шаблон	**Де застосовано**	**Призначення**
Layered architecture	Controllers → Services → AppDbContext → PostgreSQL	Розділення HTTP, логіки й даних
Dependency Injection / IoC	Program.cs і конструктори	Явні залежності та керування життєвими циклами
Service Layer	TicketService, ReminderService, Processor	Сценарії розміщені поза контролерами
DTO	Request/Response-класи	Відокремлення API від EF-сутностей
Options pattern	DeadlineReminderOptions	Типізована конфігурація інтервалів
Background Worker	DeadlineReminderBackgroundService	Періодична робота без HTTP-запиту
Unit of Work	AppDbContext	Узгоджене збереження змін
Idempotency guard	Lock, перевірка ключів, унікальний індекс	Відсутність дублікатів нагадувань

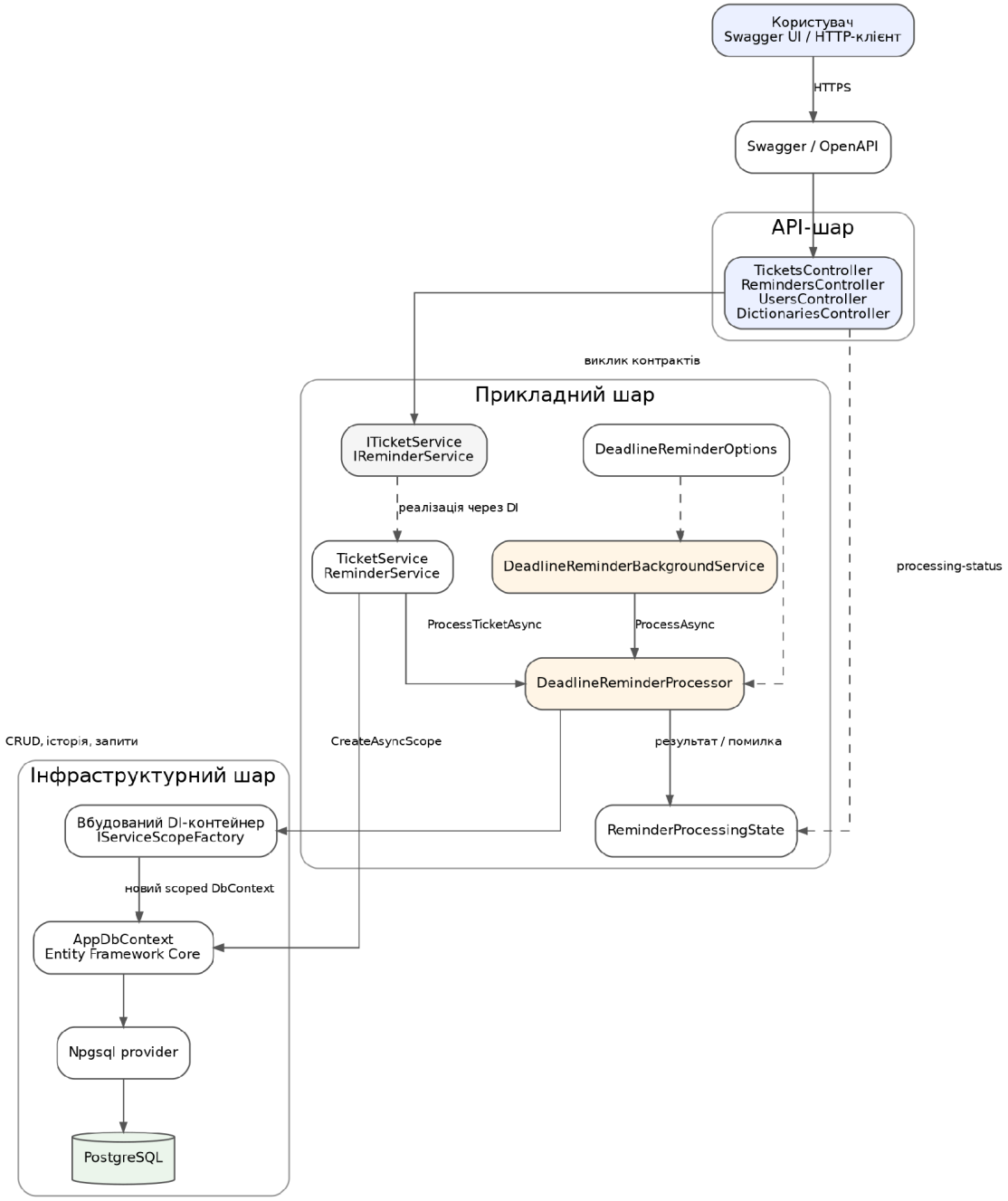

Рис. 2.6 – Деталізована архітектура фактично реалізованого застосунку

На рис. 2.6 окремо показано два способи запуску логіки нагадувань. Перший — регулярний виклик із DeadlineReminderBackgroundService. Другий — негайна перевірка конкретної заявки після створення або зміни. Обидва шляхи використовують один DeadlineReminderProcessor, тому правила не дублюються.

2.9. REST-підхід і обґрунтування HTTP-методів

Зовнішній контракт застосунку спроєктовано в ресурсно-орієнтованому стилі REST. Основними ресурсами є tickets, reminders і users, а маршрути використовують іменники, а не назви C#-методів. Сервер не зберігає стан клієнтської сесії між запитами: кожний запит містить дані, необхідні для виконання конкретної операції [11; 15; 28].

GET використовується для читання й не повинен змінювати стан системи. Тому списки заявок, одна заявка, історія та нагадування отримуються через GET. Параметри фільтрації винесені в query string, оскільки вони уточнюють представлення колекції. Наприклад, onlyOverdue=true додає до LINQ-запиту умову щодо DueAt і фінальних статусів.

POST /api/tickets створює новий ресурс, тому API повертає 201 Created через CreatedAtAction. Клієнт отримує GUID і адресу GET-операції. POST /api/reminders/process має командну семантику: він вручну запускає обробку. Такий endpoint залишено для діагностики й демонстрації, тоді як штатний запуск виконується автоматично.

PUT /api/tickets/{id} обрано для одночасного оновлення набору основних редагованих полів: назви, опису, пріоритету й дедлайну. Для вузьких змін використано PATCH. Статус, виконавець і ознака прочитання змінюються окремими маршрутами, оскільки це самостійні бізнес-операції. PATCH призначений для часткової модифікації ресурсу [28; 29].

DELETE у поточному API відсутній навмисно. Фізичне видалення заявки суперечило б вимозі збереження історії та могло б приховати факт прострочення. Життєвий цикл завершується статусами Closed або Cancelled. Для production-рішення можна додати архівування, однак для поставленої задачі воно не є необхідним.

HTTP-статуси є частиною контракту. Успішне читання або зміна повертає 200 OK, створення — 201 Created, а відсутній ідентифікатор — 404 Not Found.

Валідаційні помилки мають повертатися як 400 Bad Request. Подальшим покращенням може бути middleware, який перетворює доменні винятки у стандартизований Problem Details.

Таблиця 2.9 – Обґрунтування HTTP-методів у реалізованому API

Метод	**Приклади маршрутів**	**Причина вибору**
GET	/api/tickets; /api/tickets/{id}; /api/reminders	Безпечне читання ресурсів і колекцій
POST	/api/tickets; /api/reminders/process	Створення Ticket або запуск команди з побічним ефектом
PUT	/api/tickets/{id}	Оновлення набору основних полів заявки
PATCH	/status; /assign; /mark-as-read	Цільова часткова зміна одного аспекту ресурсу
DELETE	Не реалізовано	Історія зберігається; завершення відбувається через статус

Висновки до розділу 2. У другому розділі обґрунтовано технологічний стек, описано шарову архітектуру і фактичні залежності компонентів, спроєктовано інформаційну модель, логіку нагадувань та REST API. Показано застосування ООП, SOLID, dependency injection, Service Layer, DTO, Options pattern і фонового Worker. Кожне рішення пов’язане з конкретним класом, маршрутом або правилом у коді.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1. Реалізація серверної частини системи

Реалізація системи виконана як ASP.NET Core Web API-застосунок. Структура проєкту побудована так, щоб відокремити API-шар, бізнес-логіку та доступ до даних. Такий поділ не є формальністю: він полегшує читання коду й дозволяє змінювати окремі частини системи без переписування всього застосунку.

API-шар містить контролери, які приймають HTTP-запити. У контролерах немає складних правил обробки дедлайнів або створення нагадувань. Вони викликають відповідні сервіси та повертають результат. Це робить контролери коротшими й зрозумілішими.

Шар сервісів відповідає за сценарії роботи із заявками. Наприклад, під час створення заявки сервіс перевіряє вхідні дані, формує сутність Ticket, додає запис історії та зберігає зміни в базі. Під час зміни статусу сервіс не просто змінює поле Status, а також створює подію в TicketHistory.

Окремий сервіс відповідає за обробку дедлайнів. Його викликає BackgroundService, але за потреби той самий сервіс може бути викликаний із демонстраційного endpoint-а. Це корисно під час тестування, оскільки не завжди зручно чекати чергового автоматичного запуску фонової перевірки.

Конфігурація застосунку зберігається в appsettings.json. Там розміщується рядок підключення до PostgreSQL і параметри фонового сервісу, наприклад інтервал перевірки строків. Для реального production-середовища такі параметри краще виносити в змінні середовища, але для локального запуску дипломного проєкту файл конфігурації є достатнім.

Таблиця 3.1 – Орієнтовна структура програмного рішення

Каталог / файл	**Призначення**
Controllers	HTTP-маршрути для tickets, reminders, users і dictionaries

Каталог / файл	Призначення
Contracts	Request/response DTO для зовнішнього API-контракту
Domain/Entities	AppUser, Ticket, TicketHistory, Reminder
Domain/Enums	Статуси, пріоритети та типи нагадувань
Services	Сценарії роботи із заявками та нагадуваннями
BackgroundServices	Фоновий цикл, процесор і діагностичний стан
Infrastructure	AppDbContext і DbSeeder
Options	DeadlineReminderOptions
README.md, docs	Локальний запуск, тестовий план і пояснення БД

Фрагменти коду не розміщуються безпосередньо в основному тексті, оскільки вони займають багато місця й мають допоміжний характер. Найважливіші приклади винесено до додатків: реєстрація сервісів, реалізація BackgroundService, контролер заявок, DbContext і команди локального запуску.

3.2. Реалізація механізму доступу до даних

Доступ до даних реалізовано за допомогою Entity Framework Core. Центральним класом є AppDbContext, у якому оголошено чотири набори сутностей: Users, Tickets, TicketHistory і Reminders. У методі OnModelCreating задано назви таблиць, обмеження довжини, enum-конвертацію, зовнішні ключі, індекси та каскадну поведінку.

Для PostgreSQL використовується провайдер Npgsql для Entity Framework Core. У поточному навчальному MVP схема створюється під час старту через EnsureCreatedAsync, після чого DbSeeder додає демонстраційні записи. Це спрощує локальний запуск. Для production-версії доцільно перейти на міграції EF Core, оскільки вони краще фіксують послідовні зміни схеми [3; 5].

Основною сутністю є Ticket. Вона має зв’язки з користувачем-ініціатором, виконавцем, нагадуваннями та історією змін. Зв’язок між Ticket і Reminder є один-до-багатьох: одна заявка може мати кілька нагадувань, наприклад попереджувальне й нагадування про прострочення. Аналогічно одна заявка може мати багато записів історії.

Під час реалізації важливо уникати зберігання складної історії в одному текстовому полі. Такий підхід начебто простіший, але пізніше ускладнює фільтрацію, сортування й аналіз. Тому історія винесена в окрему таблицю. Це відповідає реляційному підходу й робить структуру даних зрозумілішою.

Для локального запуску PostgreSQL можна встановити на комп’ютер або підняти в Docker-контейнері. У README до проєкту доцільно вказати обидва варіанти. У межах захисту достатньо мати готову локальну базу й виконані міграції.

Таблиця 3.2 – Приклад таблиць бази даних

Таблиця	**Основні дані**	**Призначення**
users	Id, FullName, Email, Role	Довідник авторів і виконавців
tickets	Id, Title, Description, Status, Priority, DueAt, CreatedByUserId, AssigneeId	Поточний стан заявки
ticket_history	Id, TicketId, EventType, OldValue, NewValue, ChangedByUserId, CreatedAt	Незмінний журнал змін
reminders	Id, TicketId, Type, Status, Message, SentAt, ReadAt	Нагадування DeadlineApproaching і Overdue

Збереження даних у PostgreSQL робить результат роботи відтворюваним. Після запуску міграцій структура бази створюється автоматично, а тестові дані можна додати через seed-метод або окремий endpoint для демонстраційного наповнення.

3.3. Реалізація механізму автоматизованих нагадувань

Механізм автоматизованих нагадувань реалізовано через BackgroundService. Його завдання полягає в тому, щоб періодично запускати перевірку дедлайнів незалежно від дій користувача. Це важливо: якщо нагадування створюється лише тоді, коли хтось відкриває список заявок, система не є справді автоматизованою.

Фоновий сервіс працює циклічно. Після запуску застосунку він очікує заданий інтервал, створює scope для отримання scoped-сервісів, викликає сервіс перевірки дедлайнів і знову переходить в очікування. Використання

IServiceScopeFactory потрібне тому, що BackgroundService є довгоживучим компонентом, а DbContext має scoped-життєвий цикл [7; 8].

Процесор дедлайнів отримує з бази лише активні заявки. Статуси Completed, Closed і Cancelled виключаються ще у LINQ-запиті. Дані читаються через AsNoTracking, після чого для кожного Ticket порівнюється DueAt з поточним UTC-часом і налаштованим порогом NotifyBeforeMinutes.

Під час реалізації враховано проблему дублювання. Без додаткової перевірки фоновий сервіс міг би створювати однакові нагадування при кожному запуску. Тому перед додаванням нового запису перевіряється, чи вже існує активне нагадування такого типу для цієї заявки. Це робить дані чистішими й спрощує демонстрацію.

У навчальному проєкті нагадування зберігаються в базі даних і переглядаються через API. Це свідоме обмеження. Надсилання повідомлень на email або в месенджер могло б бути корисним, але воно додало б інтеграційну складність, яка не є необхідною для розкриття теми дипломної роботи.

Таблиця 3.3 – Типи нагадувань у системі

Категорія	**Значення**	**Призначення**
ReminderType	DeadlineApproaching	Попередження, коли DueAt потрапляє у вікно NotifyBeforeMinutes
ReminderType	Overdue	Фіксація прострочення активної заявки
ReminderStatus	Sent	Нагадування створене та ще не прочитане
ReminderStatus	Read	Користувач виконав mark-as-read; заповнено ReadAt

Основна перевага підходу полягає в тому, що логіка нагадувань залишається всередині системи й перевіряється локально. Для демонстрації достатньо створити заявку з DueAt через кілька хвилин, після чого перевірити Reminder у Swagger або Adminer. Окремий endpoint /api/reminders/process дозволяє запустити той самий процесор вручну.

3.4. Реалізація API та перевірка роботи через Swagger

Swagger UI використовується як основний інструмент демонстрації. Після запуску застосунку користувач переходить за адресою /swagger і бачить перелік доступних endpoint-ів. Кожен endpoint можна відкрити, переглянути модель запиту, виконати запит і побачити відповідь сервера.

Для створення заявки виконується POST-запит до /api/tickets. У тілі запиту передаються назва, опис, пріоритет, дедлайн і виконавець. У разі успіху сервер повертає створену заявку з ідентифікатором. Якщо дані некоректні, API має повернути помилку валідації.

Для перевірки списку заявок використовується GET-запит до /api/tickets. У фактичній реалізації цей endpoint підтримує фільтри за статусом, пріоритетом, виконавцем, ознакою прострочення та наближенням дедлайну. Фільтри передаються як query-параметри, тому їх можна комбінувати без створення окремого маршруту для кожного сценарію.

Зміна статусу виконується окремим PATCH-запитом. Це краще, ніж повне оновлення заявки через PUT, оскільки зміна статусу є окремою бізнес-операцією. Вона повинна створювати запис в історії та може мати додаткові правила. Наприклад, не можна закрити заявку, яка не була виконана.

Swagger також використовується для перевірки нагадувань. Після створення заявки з близьким або простроченим дедлайном можна викликати endpoint перегляду нагадувань і переконатися, що BackgroundService створив відповідний запис. Це робить демонстрацію наочною навіть без frontend.

Таблиця 3.4 – Приклад демонстраційного сценарію через Swagger

Крок	**Дія**	**Очікуваний результат**
1	Отримати тестових користувачів через GET /api/users	Повертаються seeded GUID користувачів
2	Створити Ticket із DueAt через 2–3 хвилини	POST повертає 201 Created
3	Переглянути Ticket і таблицю ticket_history	Є подія Created

Крок	Дія	Очікуваний результат
4	Запустити POST /api/reminders/process або зачекати фоновий цикл	Створюється DeadlineApproaching
5	Дочекатися настання DueAt	Створюється Overdue; дублікатів немає
6	Виконати mark-as-read	Status стає Read, ReadAt заповнюється
7	Змінити статус на Completed	Нові нагадування для заявки не створюються

Такий сценарій добре підходить для захисту, оскільки показує не лише окремі endpoint-и, а зв’язок між ними: створення заявки впливає на базу даних, фоновий сервіс створює нагадування, а історія дозволяє перевірити, які події відбулися.

3.5. Тестування системи

Тестування системи проводиться за сценарним підходом. Оскільки проєкт є навчальним MVP, основну увагу приділено не навантажувальному тестуванню, а перевірці коректності бізнес-логіки. Потрібно переконатися, що заявки створюються, дедлайни зберігаються, статуси змінюються, нагадування не дублюються, а історія подій відповідає виконаним операціям.

Перший набір тестів стосується CRUD-операцій із заявками. Перевіряється створення заявки з коректними даними, відхилення некоректного запиту, отримання списку, перегляд деталей і оновлення полів. Ці тести підтверджують, що базова взаємодія з API працює правильно.

Другий набір тестів пов’язаний зі статусами. Система повинна дозволяти допустимі переходи й не повинна створювати суперечливий стан. Наприклад, закрита заявка не повинна знову потрапляти в активну перевірку дедлайнів. Така логіка може бути реалізована в сервісі заявок.

Третій набір тестів перевіряє BackgroundService і сервіс дедлайнів. Тут важливо створити кілька заявок із різними датами: майбутній дедлайн, дедлайн у межах попереджувального порогу, уже прострочений дедлайн і закрита заявка.

Очікується, що нагадування будуть створені тільки для тих записів, для яких це потрібно.

Четвертий набір перевіряє відсутність дублювання. Для цього сервіс дедлайнів запускається кілька разів поспіль. Якщо для заявки вже існує активне нагадування типу Overdue, повторний запуск не повинен створювати ще один такий самий запис.

П’ятий набір тестів стосується Swagger. Перевіряється, чи всі основні endpoint-и відображаються, чи мають зрозумілі схеми запитів і відповідей, а також чи можна виконати демонстраційний сценарій без сторонніх інструментів.

Рис. 3.1 – Кількість перевірок за групами тестових сценаріїв

Таблиця 3.5 – Приклади тестових сценаріїв

Сценарій	**Вхідні умови**	**Очікуваний результат**
Створення заявки	Коректні DTO та GUID користувачів	201 Created, Ticket і подія Created
Порожня назва або опис	Порушення DataAnnotations	400 Bad Request
Невідомий enum	Некоректний priority/status	400 Bad Request
Наближення дедлайну	DueAt у межах NotifyBeforeMinutes	Reminder DeadlineApproaching
Прострочення	DueAt менший за UTC now	Reminder Overdue
Повторна перевірка	Пара TicketId + Type уже існує	Новий запис не створюється

Сценарій	Вхідні умови	Очікуваний результат
Фінальний статус	Completed, Closed або Cancelled	Ticket виключається з фонової вибірки

Результати тестування показують, що система виконує основні функції, визначені в постановці задачі. Окремі обмеження залишаються: не реалізовано повноцінну автентифікацію, немає зовнішніх каналів надсилання нагадувань і відсутній окремий frontend. Проте ці обмеження не заважають перевірити головну ідею роботи.

3.6. Аналіз результатів реалізації

У результаті роботи отримано серверний застосунок, який демонструє повний цикл обробки заявок із контролем строків. Система дозволяє створювати заявки, зберігати їх у PostgreSQL, змінювати статуси, переглядати історію та отримувати нагадування, сформовані автоматично. При цьому застосунок не потребує хмарного розгортання й може бути запущений локально.

Одним із практичних результатів є те, що проєкт має зрозумілий сценарій демонстрації. На захисті можна показати не тільки статичні діаграми, а й роботу API: створити заявку, виконати перевірку, побачити нагадування та переглянути історію. Такий сценарій краще пояснює роботу системи, ніж простий опис класів або таблиць.

Сильним боком реалізації є відокремлення фонової логіки від API. BackgroundService не змішується з контролерами, а використовує окремий сервіс дедлайнів. Завдяки цьому логіку нагадувань можна тестувати й розвивати незалежно. У майбутньому до неї можна додати відправку email або інтеграцію з месенджером без повної перебудови системи.

Ще одним важливим результатом є використання PostgreSQL і EF Core. Реляційна модель підходить для цієї задачі, оскільки між заявками, користувачами, нагадуваннями та історією існують чіткі зв’язки. EF Core спрощує роботу з цими зв’язками на рівні коду й дозволяє відтворити структуру бази через міграції.

Разом із тим система має межі MVP. У ній немає складної системи ролей, повноцінної авторизації, frontend-інтерфейсу, інтеграції з поштою та статистичних звітів. Ці можливості можна розглядати як напрямки подальшого розвитку. Для бакалаврської роботи важливішим є те, що базова задача контролю строків і автоматизованих нагадувань реалізована цілісно.

У майбутньому систему можна розширити кількома способами. По-перше, додати JWT-автентифікацію та реальні ролі користувачів. По-друге, реалізувати frontend на React або Blazor. По-третє, додати зовнішні канали сповіщень. По-четверте, розширити аналітику: середній час виконання, кількість прострочених заявок, навантаження на виконавців. По-п’яте, підготувати Docker Compose для ще простішого локального запуску.

Таблиця 3.6 – Реалізовані результати та можливий розвиток

Напрям	**Реалізовано в роботі**	**Можливе розширення**
Робота із заявками	Створення, перегляд, зміна статусів	Розширені фільтри та пошук
Контроль строків	Перевірка дедлайнів через BackgroundService	Налаштовувані правила для різних пріоритетів
Нагадування	Внутрішні записи в БД	Email, Telegram або інші канали
Доступ до системи	Swagger UI для демонстрації	Окремий frontend та автентифікація
Аналітика	Базові дані для аналізу	Звіти, графіки, dashboard

Висновки до розділу 3. У третьому розділі описано реалізацію серверної частини, доступу до даних, фонового механізму нагадувань, перевірки API через Swagger і тестування основних сценаріїв. Розроблена система відповідає поставленій задачі та може бути запущена локально без хмарного розгортання. Отриманий результат має практичну цінність як навчальний програмний продукт і як основа для подальшого розвитку.

3.7. Локальний запуск, демонстраційні дані та відтворюваність результатів

Окремою вимогою до дипломного проєкту є можливість локального запуску. Це рішення було прийняте свідомо, оскільки для захисту бакалаврської роботи важливо мати контрольоване середовище. Якщо система залежить від хмарного сервісу, доступу до зовнішнього API або платного ресурсу, демонстрація може зірватися через причину, яка не пов’язана з якістю коду. Локальний запуск зменшує такі ризики.

Локальний сценарій не означає, що система є менш практичною. Навпаки, більшість backend-застосунків спочатку розробляється й перевіряється локально. У цьому режимі розробник бачить логи, може швидко змінювати конфігурацію, запускати міграції та створювати тестові дані. Для навчального проєкту це дозволяє показати не тільки кінцевий результат, а й інженерний процес.

Після клонування репозиторію користувач повинен мати зрозумілу інструкцію. У ній зазначаються необхідні компоненти: .NET SDK, PostgreSQL або Docker, доступ до командного рядка та, за бажанням, IDE. Далі користувач виконує відновлення пакетів, налаштовує connection string, застосовує міграції та запускає Web API. Після старту застосунку Swagger UI відкривається в браузері.

Для відтворюваності важливо, щоб у репозиторії були не лише файли проєкту, а й README з послідовністю команд. Якщо інша людина не може запустити проєкт без пояснень автора, це зменшує цінність роботи. Тому README є частиною практичного результату, хоча формально не належить до програмного коду.

Вихідний код і документація локального запуску доступні у відкритому GitHub-репозиторії: https://github.com/Artemmal/DeadlineControl. У репозиторії розміщено README, Docker Compose, план ручного тестування та діаграми.

Демонстраційні дані також мають значення. Порожня система погано показує логіку нагадувань, тому доцільно підготувати кілька заявок: одну з майбутнім дедлайном, одну з дедлайном у межах 24 годин, одну вже прострочену,

одну закриту та одну з високим пріоритетом. Такий набір дозволяє швидко перевірити різні гілки логіки.

У дипломному проєкті демонстраційні дані можуть створюватися двома способами. Перший спосіб — seed-метод, який заповнює базу після застосування міграцій. Другий спосіб — створення записів через Swagger. Для захисту зручніше мати обидва варіанти: seed прискорює старт, а Swagger показує, що API дійсно працює.

Відтворюваність результату також стосується версій інструментів. Якщо в README зазначити версію .NET SDK і PostgreSQL, перевіряючому буде легше підготувати середовище. Це не гарантує повної відсутності проблем, але суттєво зменшує кількість випадкових помилок під час запуску.

Ще один практичний момент — конфігурація інтервалу BackgroundService. Для реальної системи інтервал перевірки може бути більшим, наприклад 5 або 15 хвилин. Для демонстрації краще використовувати коротший інтервал, щоб нагадування з’являлися швидко. Тому інтервал доцільно винести в appsettings.json, а не фіксувати безпосередньо в коді.

Таким чином, локальний запуск у цій роботі розглядається як повноцінний спосіб демонстрації, а не як тимчасова заміна cloud deployment. Він відповідає обсягу бакалаврського проєкту, зменшує залежність від зовнішніх сервісів і дозволяє детально показати роботу API, бази даних та фонового сервісу.

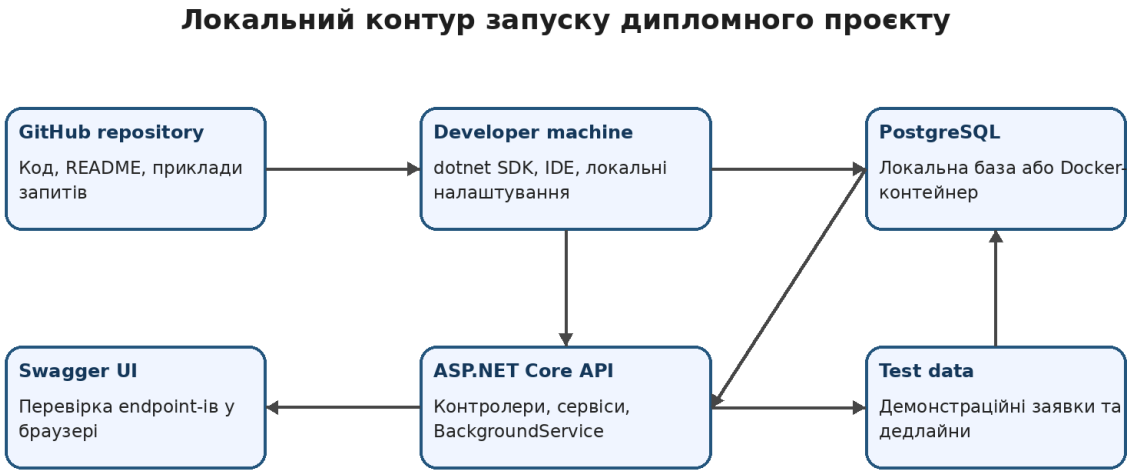

Рис. 3.2 – Локальний контур запуску дипломного проєкту

Таблиця 3.7 – Мінімальні вимоги до локального середовища

Компонент	**Призначення**	**Коментар**
.NET 8 SDK	Компіляція та запуск Web API	Версія вказана у README
Docker Desktop / Docker Engine	Запуск PostgreSQL та Adminer	Команда docker compose up -d
Git	Клонування репозиторію	Код доступний на GitHub
Browser	Swagger UI та Adminer	Окремий frontend не потрібен
IDE або CLI	Редагування й запуск застосунку	Visual Studio, Rider, VS Code або dotnet CLI

Під час підготовки до захисту варто перевірити локальний запуск на чистому середовищі або хоча б в окремій папці. Це допомагає помітити приховані залежності: наприклад, локальний файл конфігурації, який не потрапив до репозиторію, або міграцію, яку було створено, але не закомічено.

Ще одна корисна практика — мати короткий сценарій демонстрації. Він може складатися з п’яти кроків: запуск застосунку, відкриття Swagger, створення заявки, ручний запуск перевірки дедлайнів і перегляд створеного нагадування. Якщо такий сценарій відпрацьований, демонстрація виглядає послідовною й переконливою.

Локальний запуск також допомагає уникнути зайвих питань щодо вартості хмарної інфраструктури. У роботі не використовується Azure, AWS чи інша платформа, тому немає потреби створювати платні ресурси. Водночас архітектура не забороняє майбутнє розгортання: Web API і PostgreSQL можна перенести в контейнеризоване або хмарне середовище пізніше.

3.8. Обробка помилок, валідація та журналювання

Для системи контролю заявок важливо не лише виконувати успішні сценарії, а й коректно реагувати на помилки. Якщо користувач передає некоректний дедлайн, неіснуючий ідентифікатор виконавця або намагається змінити статус закритої заявки, API не повинен повертати випадкову помилку сервера. Відповідь має бути передбачуваною та зрозумілою.

Валідація починається на рівні DTO. Наприклад, назва заявки не повинна бути порожньою, дедлайн має бути коректною датою, а пріоритет повинен належати до допустимого набору значень. Така перевірка дозволяє відхилити помилковий запит ще до виконання бізнес-логіки.

Друга частина перевірок належить до доменної логіки. Наприклад, формально статус можна передати як рядок або enum, але не кожен перехід статусу є допустимим. Заявку не варто переводити зі стану «Закрита» назад у «Нова» без окремого правила. Такі перевірки повинні бути в сервісі, а не лише в атрибутах DTO.

Обробка помилок у контролерах має повертати відповідні HTTP-коди. Якщо заявку не знайдено, доцільно повертати 404 Not Found. Якщо вхідні дані некоректні, використовується 400 Bad Request. Якщо операція порушує правило бізнес-логіки, можна повернути 409 Conflict або 400 Bad Request із поясненням. Для непередбачених помилок сервер повертає 500, але такі ситуації потрібно логувати.

Журналювання допомагає зрозуміти, що відбулося в системі. Для дипломного MVP достатньо логувати запуск застосунку, запуск BackgroundService, кількість перевірених заявок, створені нагадування та помилки під час обробки. Це не лише допомагає в розробці, а й робить демонстрацію більш прозорою.

Окремо варто логувати роботу фонового сервісу. Якщо нагадування не створюються, без логів важко зрозуміти, чи сервіс не запустився, чи не знайшов активних заявок, чи виникла помилка при збереженні. Короткі структуровані записи значно спрощують діагностику.

У роботі не ставиться задача створити складну систему моніторингу. Проте базове логування через стандартні засоби ASP.NET Core є достатнім і відповідає обсягу проєкту [9]. У майбутньому ці логи можна було б передавати в зовнішню систему спостереження, але для локального запуску достатньо консолі та файлів логів.

Валідація й логування також впливають на якість API-документації. У Swagger бажано бачити не лише успішні відповіді, а й можливі помилки. Це допомагає користувачу API зрозуміти, які дані очікує система та як вона реагує на некоректні сценарії.

Таблиця 3.8 – Приклади помилкових сценаріїв і реакції API

Ситуація	**Фактична або очікувана реакція**	**Коментар**
Порожня назва/опис	400 Bad Request	Валідація DTO
Некоректний enum або дата	400 Bad Request	Помилка model binding
Ticket за id не знайдено	404 Not Found	Контролер повертає NotFound
Повторна обробка нагадувань	200 OK, createdReminderCount = 0	Дедуплікація працює ідемпотентно
Неіснуючий userId	Доменний виняток; потребує єдиного Problem Details middleware	Зафіксовано як напрям покращення
Помилка підключення до PostgreSQL	500 / помилка startup	Перевіряються connection string і Docker-контейнер

Підхід до помилок має бути однаковим у всій системі. Якщо один endpoint повертає детальне повідомлення, а інший лише загальний текст, користувачу API

важче зрозуміти поведінку застосунку. Тому варто використовувати єдину модель відповіді для помилок або хоча б однакові принципи формування повідомлень.

Для дипломної роботи достатньо показати кілька прикладів помилкових сценаріїв. Це демонструє, що система проєктувалася не тільки для «щасливого шляху». У реальних застосунках саме обробка помилок часто визначає, наскільки зручно підтримувати продукт після першого запуску.

З позиції програмної інженерії валідація, обробка помилок і журналювання є не другорядними деталями, а частиною якості системи. Вони не змінюють основну функцію контролю строків, але роблять її більш надійною та зрозумілою для користувача й розробника.

3.9. Обмеження MVP та практична оцінка результату

Розроблена система є MVP, тобто мінімально достатньою версією програмного продукту для демонстрації основної ідеї. Це означає, що вона не включає всі можливості, які могла б мати повноцінна корпоративна система. Такий підхід є виправданим для бакалаврської роботи, оскільки дозволяє глибше опрацювати ключову частину замість поверхневого додавання багатьох функцій.

Першим обмеженням є відсутність повноцінної автентифікації та авторизації. У реальному продукті користувачі повинні входити в систему, мати ролі та доступ лише до дозволених дій. У межах цієї роботи ролі розглядаються на рівні моделі та сценаріїв, але складна security-логіка не є центральною частиною теми.

Другим обмеженням є відсутність зовнішніх каналів сповіщень. Система створює нагадування як записи в базі даних, але не надсилає їх на email або в месенджери. З одного боку, це спрощує проєкт. З іншого боку, така реалізація все одно доводить головне: система здатна автоматично виявити потрібний стан заявки та зафіксувати нагадування.

Третім обмеженням є відсутність окремого frontend. Для користувача production-системи Swagger не є зручним інтерфейсом. Але для дипломного проєкту він має перевагу: дозволяє прозоро показати API, тіла запитів, відповіді, статус-коди й повну послідовність взаємодії.

Четвертим обмеженням є відсутність складної аналітики. Система зберігає дані, на основі яких у майбутньому можна побудувати звіти, але сама дипломна реалізація обмежується базовим переглядом заявок, нагадувань та історії. Це дозволяє залишити фокус на механізмі контролю строків.

Попри ці обмеження, практичний результат є цілісним. У системі є предметна модель, база даних, API, фоновий процес, документація через Swagger і тестові сценарії. Це не набір розрізнених фрагментів, а застосунок, компоненти якого працюють разом.

Оцінюючи результат, можна сказати, що проєкт виконує навчальну та прикладну функцію. Навчальна функція полягає в демонстрації підходів до розробки Web API, роботи з EF Core і реалізації фонової обробки. Прикладна функція полягає в тому, що систему можна адаптувати під реальний невеликий процес контролю заявок.

Якщо розвивати систему далі, найперше варто додати автентифікацію, ролі та frontend. Після цього можна підключити email-сповіщення, додати dashboard і розгорнути застосунок у хмарі або Docker-середовищі. Проте ці кроки є логічним продовженням, а не обов’язковою умовою для виконання поставленої задачі.

Загалом MVP добре відповідає меті дипломної роботи. Він достатньо простий, щоб бути зрозумілим і відтворюваним, але водночас містить реальну бізнес-логіку, а не лише базові CRUD-операції. Саме наявність фонової перевірки дедлайнів і нагадувань робить проєкт змістовним з погляду інженерії програмного забезпечення.

Таблиця 3.9 – Межі MVP і можливі наступні кроки

Обмеження поточної версії	Причина обмеження	Можливий розвиток
Немає повної авторизації	Фокус роботи на контролі строків	JWT, ролі, політики доступу
Немає frontend	Swagger достатній для демонстрації API	React, Angular або Blazor-інтерфейс
Немає email-сповіщень	Зменшення інтеграційної складності	SMTP або зовнішній notification-сервіс
Обмежена аналітика	MVP орієнтований на core-логіку	Dashboard, звіти, метрики виконання
Локальний запуск без cloud	Відтворюваність і відсутність платних ресурсів	Docker Compose, Azure App Service або AWS ECS

Висновки до додаткових підрозділів розділу 3. Локальний запуск, обробка помилок і оцінка меж MVP доповнюють опис реалізації. Вони показують, що програмний продукт розглядається не лише як код, а як відтворювана система з інструкцією запуску, тестовими даними, очікуваною поведінкою в помилкових сценаріях і зрозумілими напрямами розвитку.

Саме ці аспекти часто відрізняють навчальний проєкт від набору прикладів коду. Якщо застосунок можна клонувати, налаштувати, запустити, перевірити через Swagger і пояснити його обмеження, він стає достатньо переконливим результатом для бакалаврської дипломної роботи з інженерії програмного забезпечення.

3.10. Практичний сценарій використання системи у робочому процесі

Щоб оцінити корисність розробленої системи, варто розглянути не лише технічну реалізацію, а й практичний сценарій її використання. Нехай у невеликій організації є внутрішній процес обробки звернень від співробітників. Частина звернень стосується технічних проблем, частина — адміністративних питань, а частина — уточнення статусу певних внутрішніх робіт.

У ручному режимі такі звернення можуть надходити в чат, електронну пошту або особисті повідомлення. Спочатку це здається зручним, бо не потрібно впроваджувати окрему систему. Але з часом кількість звернень збільшується, і

стає складно зрозуміти, які з них уже виконані, які очікують реакції, а які залишилися без відповіді. Саме в такій ситуації потрібен централізований облік заявок.

Розроблена система може бути використана як невеликий внутрішній API-сервіс. Відповідальна особа створює заявку через Swagger або через майбутній frontend, зазначає коротку назву, опис, виконавця, пріоритет і дедлайн. Після цього заявка зберігається в базі даних і стає частиною контрольованого процесу.

Далі виконавець змінює статус заявки залежно від реального стану робіт. Якщо він почав роботу, статус змінюється на «У роботі». Якщо потрібне уточнення, заявку можна перевести у стан очікування або відредагувати її опис і дедлайн. Після завершення робіт заявка переводиться у виконаний або закритий стан. Кожна важлива зміна фіксується в історії.

Фоновий сервіс у цьому сценарії працює як незалежний контролер строків. Йому не потрібно, щоб користувач вручну відкривав кожну заявку. Він періодично аналізує дедлайни й створює нагадування. Це особливо корисно для заявок із середнім або низьким пріоритетом, які легко втратити серед терміновіших задач.

Наприклад, якщо дедлайн заявки настає завтра, система створює попереджувальне нагадування. Відповідальна особа бачить, що строк наближається, і може завчасно перевірити стан виконання. Якщо строк уже минув, система фіксує прострочення. Це дає можливість не лише реагувати на проблему, а й аналізувати, чому вона виникла.

Важливо, що система не приймає управлінські рішення замість людини. Вона не визначає, хто винен у простроченні, і не змінює автоматично пріоритети. Її задача інша — зробити стан процесу видимим. Для інформаційної системи це дуже важлива роль: вона не замінює організаційні правила, але допомагає їх виконувати.

Практична цінність також полягає в тому, що дані зберігаються структуровано. Якщо через кілька тижнів потрібно зрозуміти, чому певна заявка була виконана пізніше, можна переглянути її історію. Там видно, коли заявку створили, коли змінили статус, коли сформувалося нагадування і чи було прострочення.

Такий підхід відрізняється від звичайної таблиці. У таблиці теж можна зберігати дедлайн, але вона не має вбудованої поведінки. Вона не перевірить строк сама, не створить нагадування, не захистить від випадкового дублювання запису і не забезпечить однаковий API для майбутньої інтеграції. У розробленій системі дані та поведінка працюють разом.

У практичному сценарії важливим є і те, що система може бути поступово розширена. Спочатку вона використовується лише через Swagger. Потім до API можна додати простий frontend. Після цього — email-сповіщення. Далі — ролі та авторизацію. Така еволюція є природною, оскільки серверна частина вже має окрему модель даних і чіткі endpoint-и.

З погляду розробника, така система є зручною для підтримки. Якщо змінюється правило нагадування, потрібно редагувати сервіс дедлайнів, а не всі контролери. Якщо змінюється структура таблиць, це фіксується через міграції EF Core. Якщо потрібно додати новий endpoint, він може використати вже існуючі сервіси.

З погляду користувача, основна перевага полягає в прозорості. Користувач бачить, що заявка не просто «десь є», а має конкретний статус, виконавця і строк. Якщо строк наближається, система робить це помітним. Якщо строк порушено, факт прострочення не губиться в листуванні.

З погляду керівника або відповідальної особи, система допомагає контролювати навантаження. Навіть без складної аналітики можна переглянути активні та прострочені заявки. У майбутньому ці дані можуть бути основою для

dashboard, де відображатиметься кількість відкритих заявок, середній час виконання та частка прострочених задач.

Окремо слід зазначити, що локальний формат запуску не заважає практичному сценарію. Навпаки, він дозволяє швидко розгорнути систему в навчальному або тестовому середовищі. Якщо організації потрібне production-використання, той самий проєкт можна підготувати до розгортання через Docker або хмарний сервіс.

У межах дипломної роботи практичний сценарій виконує ще одну функцію: він показує, що реалізована система має зміст за межами синтаксису коду. Це не просто приклад використання ASP.NET Core або EF Core, а модель конкретного процесу, де є заявки, строки, відповідальні особи, нагадування та історія подій.

Для демонстрації такого сценарію достатньо підготувати кілька записів. Наприклад, створити заявку з високим пріоритетом і дедлайном сьогодні, заявку з дедлайном завтра, прострочену заявку та закриту заявку. Після запуску перевірки можна показати, що система обробляє їх по-різному: для одних створює нагадування, інші пропускає.

Цей сценарій також добре пояснює вибір BackgroundService. Без нього логіку дедлайнів довелося б запускати вручну або прив’язувати до перегляду списку. У такому разі система не виконувала б автоматизований контроль у повному сенсі. BackgroundService робить поведінку ближчою до реальної серверної системи.

Практичний сценарій показує, що система може бути корисною навіть у мінімальній версії. Вона не має всіх можливостей великих task management-платформ, але вирішує конкретну проблему: допомагає не втрачати заявки й контролювати строки. Для бакалаврської роботи це достатньо переконливий результат.

Таким чином, розроблений застосунок можна оцінювати як основу для подальшого розвитку. Він має чітко визначене ядро, зрозумілий спосіб запуску,

можливість перевірки через Swagger і логіку, яка відповідає темі роботи. Подальші доробки не змінюють основної ідеї, а лише розширюють її практичне використання.

Таблиця 3.10 – Практичний сценарій використання системи

Етап процесу	**Дія користувача або системи**	**Результат**
Створення	POST /api/tickets з автором, виконавцем і DueAt	Ticket і TicketHistory.Created
Призначення	PATCH /api/tickets/{id}/assign	AssigneeId оновлено, історія доповнена
Виконання	PATCH /api/tickets/{id}/status	Фіксується StatusChanged
Наближення строку	Processor порівнює DueAt з reminderThreshold	Створюється DeadlineApproaching
Порушення строку	DueAt минув, статус не фінальний	Створюється Overdue
Прочитання	PATCH /api/reminders/{id}/mark-as-read	Status = Read, заповнено ReadAt
Завершення	Completed, Closed або Cancelled	Ticket більше не обробляється фоново

Завдяки такому сценарію можна чітко пояснити практичну логіку роботи під час захисту. Спочатку демонструється створення заявки, потім зміна її стану, далі робота фонового сервісу і в кінці перегляд результатів. Це послідовна історія використання системи, а не набір окремих технічних операцій.

У підсумку практичний сценарій підтверджує, що тема дипломної роботи розкрита не тільки теоретично. Розроблена система має визначену предметну область, реалізовану серверну частину, модель даних, автоматизовану поведінку та зрозумілий спосіб перевірки результату.

3.11. Реалізація архітектурних рішень і усунення технічних проблем

Після завершення початкової реалізації архітектурні рішення були перевірені у реальному локальному запуску. Тестування через Swagger і перегляд записів у Adminer виявили кілька проблем, які не були очевидними на етапі проєктування. Їх виправлення стало важливою частиною практичного результату.

Запити onlyOverdue і onlyDueSoon спочатку викликали допоміжний метод IsFinalStatus усередині IQueryable.Where. Для звичайної C#-колекції такий код

коректний, але EF Core повинен перетворити вираз на SQL і не може транслювати довільний метод. Фільтр переписано як прямі порівняння enum-значень. Обчислення залишилося на стороні PostgreSQL, а не переносилося в пам’ять через AsEnumerable.

Інша проблема виникла під час одночасної роботи процесора нагадувань і контексту запиту. Повторне використання tracked-об’єктів призводило до DbUpdateConcurrencyException. У фінальній реалізації DeadlineReminderProcessor не зберігає DbContext у полі. Для кожного запуску він створює новий DI scope, отримує окремий AppDbContext і читає заявки через AsNoTracking.

Щоб нагадування не залежало лише від таймера, після створення, редагування дедлайну або зміни статусу TicketService викликає ProcessTicketAsync. Якщо строк уже входить до попереджувального вікна, Reminder створюється відразу. Фоновий сервіс залишається резервним механізмом, який регулярно перевіряє всі активні заявки.

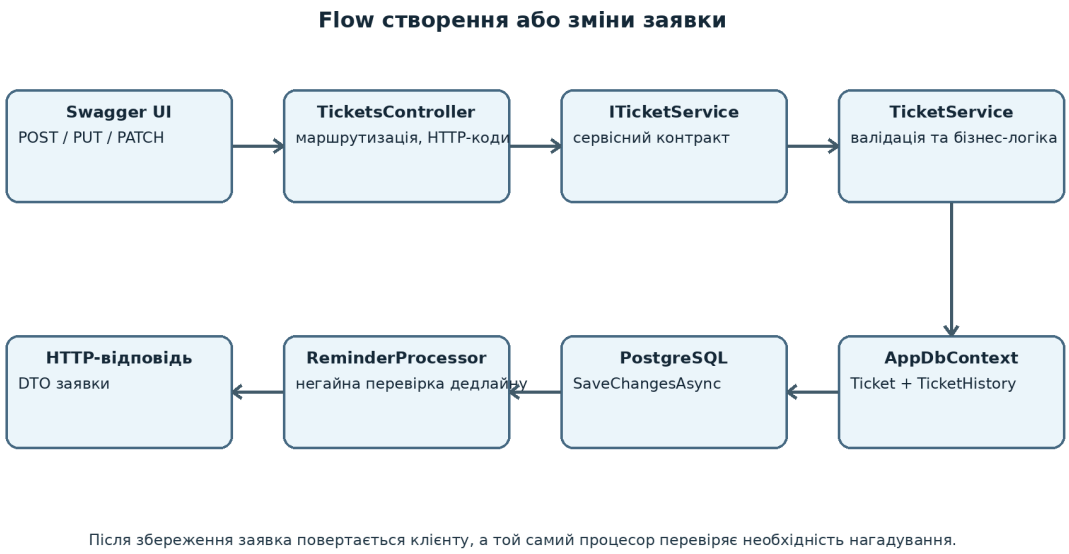

Рис. 3.3 – Фактичний flow операції створення або зміни заявки

У flow на рис. 3.3 контролер делегує роботу сервісу. TicketService перевіряє пов’язані дані, змінює Ticket, додає TicketHistory і зберігає транзакцію. Після успішного збереження виконується негайна перевірка дедлайну. Це розділяє основну транзакцію заявки та додаткову обробку нагадувань.

Рис. 3.4 – Алгоритм безпечної фонової обробки дедлайнів

Фоновий flow показано на рис. 3.4. Перед створенням scope процесор отримує ProcessingLock. Далі читаються активні заявки й наявні ключі нагадувань. Після завершення ReminderProcessingState зберігає діагностичний результат, який повертається endpoint-ом processing-status.

Таблиця 3.11 – Ключові класи фактичної реалізації та їх відповідальність

Клас або контракт	**Життєвий цикл / роль**	**Основна відповідальність**
TicketsController	HTTP-рівень	Маршрутизація, status codes, делегування ITicketService
ITicketService / TicketService	Scoped	CRUD, фільтри, статус, виконавець, історія, негайна перевірка
IReminderService / ReminderService	Scoped	Читання нагадувань і MarkAsRead
DeadlineReminderBackgroundService	Hosted singleton	Періодичний запуск процесора
DeadlineReminderProcessor	Singleton без DbContext у полі	Окремий scope, дедуплікація і створення Reminder
ReminderProcessingState	Singleton	Lock і діагностичні дані останнього запуску
AppDbContext	Scoped	EF Core mapping і збереження одиниці роботи

Таблиця 3.12 – Технічні проблеми та прийняті рішення

Проблема	**Причина**	**Рішення**
500 для onlyOverdue / onlyDueSoon	EF Core не транслював IsFinalStatus у SQL	Прямі enum-порівняння у IQueryable
Reminder не з’являвся відразу	Перевірка виконувалась лише за таймером	ProcessTicketAsync після змін заявки
DbUpdateConcurrencyException	Конфлікт tracked-стану і життєвого циклу DbContext	Новий scope та AsNoTracking для кожного запуску
Можливі дублікати	Повторні або паралельні перевірки	ProcessingLock, перевірка ключів, унікальний індекс
Складна діагностика	Помилки фонового процесу не видно у Swagger	ProcessingState і processing-status endpoint

Фрагменти фактичного коду, на які спирається опис, наведено в Додатках В–К. Вони взяті з робочої версії проєкту після виправлення фільтрації та конфлікту DbContext.

ВИСНОВКИ

У бакалаврській дипломній роботі було розглянуто задачу розробки веборієнтованої системи контролю строків виконання заявок та автоматизованих нагадувань. Обрана тема має практичний зміст, оскільки в багатьох організаціях існує потреба не лише створювати заявки, а й контролювати їх виконання в часі.

У першому розділі було проаналізовано предметну область. Заявку розглянуто як об’єкт із життєвим циклом, що включає створення, призначення виконавця, контроль дедлайну, зміну статусів, виконання та закриття. Також визначено основні проблеми ручного контролю строків: розпорошеність інформації, залежність від людського фактора, відсутність історії змін і складність аналітики.

У другому розділі виконано проєктування системи. Обґрунтовано вибір ASP.NET Core Web API, Entity Framework Core, PostgreSQL, Swagger UI та BackgroundService. Описано шарову архітектуру, доменні сутності, сервісні контракти, принципи ООП і SOLID, використання dependency injection, Service Layer, DTO та Options pattern. REST-маршрути й HTTP-методи обрано відповідно до характеру кожної операції.

У третьому розділі описано фактичну реалізацію та тестування. Показано структуру репозиторію, AppDbContext, роботу TicketService і DeadlineReminderProcessor, локальний запуск PostgreSQL та Adminer через Docker Compose, а також перевірку через Swagger. Розглянуто виправлення LINQ-запитів, негайну обробку дедлайнів і створення нового scoped DbContext для кожного фонового циклу.

Поставлену мету роботи досягнуто. Розроблено веборієнтований серверний застосунок, який створює та редагує заявки, фіксує історію, фільтрує прострочені й наближені до дедлайну записи, автоматично формує нагадування та надає діагностику фонового процесу. Система має документований REST API і відтворюваний локальний спосіб запуску після клонування репозиторію.

Подальший розвиток роботи може включати додавання повноцінної автентифікації, ролей користувачів, frontend-інтерфейсу, зовнішніх каналів сповіщень, розширеної аналітики та контейнеризації. Проте навіть у поточному вигляді проєкт демонструє основні інженерні підходи до побудови прикладної веборієнтованої системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft. ASP.NET Core overview. URL: https://learn.microsoft.com/en-us/aspnet/core/ (дата звернення: 31.05.2026).

2. Microsoft. Create web APIs with ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/web-api/ (дата звернення: 31.05.2026).

3. Microsoft. Entity Framework Core documentation. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 31.05.2026).

4. PostgreSQL Global Development Group. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 31.05.2026).

5. Npgsql. Entity Framework Core Provider for PostgreSQL. URL: https://www.npgsql.org/efcore/ (дата звернення: 31.05.2026).

6. Microsoft. ASP.NET Core web API documentation with Swagger / OpenAPI. URL: https://learn.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger (дата звернення: 31.05.2026).

7. Microsoft. Background tasks with hosted services in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services (дата звернення: 31.05.2026).

8. Microsoft. Dependency injection in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection (дата звернення: 31.05.2026).

9. Microsoft. Logging in .NET Core and ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/logging/ (дата звернення: 31.05.2026).

10. Microsoft. Configuration in ASP.NET Core. URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/ (дата звернення: 31.05.2026).

11. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine, 2000.

12. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.

13. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Prentice Hall, 2017.

14. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

15. Richardson L., Amundsen M., Ruby S. RESTful Web APIs. O’Reilly Media, 2013.

16. Microsoft Learn. Загальні відомості про основи ASP.NET Core. URL: https://learn.microsoft.com/uk-ua/training/paths/aspnet-core-fundamentals/ (дата звернення: 31.05.2026).

17. Microsoft Learn. Документація ASP.NET Core Web API за допомогою Swagger/OpenAPI. URL: https://learn.microsoft.com/uk-ua/aspnet/core/tutorials/web-api-help-pages-using-swagger (дата звернення: 31.05.2026).

18. SQL.ua. Вивчай SQL та PostgreSQL українською. URL: https://sql.ua/ (дата звернення: 31.05.2026).

19. OpenAPI Initiative. OpenAPI Specification. URL: https://spec.openapis.org/oas/latest.html (дата звернення: 31.05.2026).

20. Docker. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 31.05.2026).

21. GitHub Docs. About repositories. URL: https://docs.github.com/en/repositories (дата звернення: 31.05.2026).

22. OWASP Foundation. REST Security Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html (дата звернення: 31.05.2026).

23. OWASP Foundation. Input Validation Cheat Sheet. URL: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html (дата звернення: 31.05.2026).

24. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes.

25. ISO/IEC/IEEE 29148:2018. Systems and software engineering — Life cycle processes — Requirements engineering.

26. Microsoft. Object-oriented programming (C#). URL: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop (дата звернення: 08.06.2026).

27. Microsoft. Interfaces — define behavior for multiple types (C#). URL: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/interfaces (дата звернення: 08.06.2026).

28. IETF. RFC 9110: HTTP Semantics. URL: https://www.rfc-editor.org/rfc/rfc9110.html (дата звернення: 08.06.2026).

29. IETF. RFC 5789: PATCH Method for HTTP. URL: https://www.rfc-editor.org/rfc/rfc5789.html (дата звернення: 08.06.2026).

ДОДАТКИ

Додаток А. Розгорнута матриця функціональних вимог до системи

Таблиця А.1 – Функціональні вимоги

Код	**Вимога**	**Критерій перевірки**
FR-01	Створення заявки з назвою, описом, пріоритетом, автором, виконавцем і DueAt	POST /api/tickets повертає 201
FR-02	Перегляд і фільтрація заявок	GET /api/tickets повертає колекцію
FR-03	Отримання заявки за GUID	GET /api/tickets/{id} повертає Ticket або 404
FR-04	Оновлення полів, статусу та виконавця	PUT і PATCH змінюють стан та історію
FR-05	Автоматичне попередження про наближення строку	Створюється Reminder DeadlineApproaching
FR-06	Автоматична фіксація прострочення	Створюється Reminder Overdue
FR-07	Збереження історії змін	GET /api/tickets/{id}/history повертає події
FR-08	Позначення нагадування прочитаним	PATCH mark-as-read встановлює Read і ReadAt
FR-09	Діагностика та ручний запуск процесора	process і processing-status доступні через Swagger

Додаток Б. Розгорнута матриця нефункціональних вимог до системи

Таблиця Б.1 – Нефункціональні вимоги

Код	**Вимога**	**Пояснення**
NFR-01	Локальний відтворюваний запуск	Клонування GitHub + Docker Compose + dotnet run
NFR-02	Документований API	Swagger UI та OpenAPI
NFR-03	Реляційне збереження даних	PostgreSQL і EF Core/Npgsql
NFR-04	Підтримувана структура коду	Controllers, Contracts, Domain, Services, Infrastructure, BackgroundServices
NFR-05	Фоновий процес не блокує HTTP	BackgroundService виконується окремо
NFR-06	Відсутність дублікатів Reminder	Lock, перевірка ключів та унікальний індекс
NFR-07	Безпечний життєвий цикл DbContext	Окремий scope і AsNoTracking для фонового запуску

Додаток В. Фактична конфігурація застосунку в Program.cs

```
using System.Text.Json.Serialization;
using DeadlineControl.Api.BackgroundServices;
using DeadlineControl.Api.Infrastructure;
using DeadlineControl.Api.Options;
using DeadlineControl.Api.Services.Reminders;
using DeadlineControl.Api.Services.Tickets;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.Converters.Add(new JsonStringEnumConverter());
    });

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.Configure<DeadlineReminderOptions>(builder.Configuration.GetSection("DeadlineRem
inder"));

var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<AppDbContext>(options =>
{
    options.UseNpgsql(connectionString);
});

builder.Services.AddScoped<ITicketService, TicketService>();
builder.Services.AddScoped<IReminderService, ReminderService>();
builder.Services.AddSingleton<ReminderProcessingState>();
builder.Services.AddSingleton<DeadlineReminderProcessor>();
builder.Services.AddHostedService<DeadlineReminderBackgroundService>();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.MapControllers();

using (var scope = app.Services.CreateScope())
{
    var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();
    await dbContext.Database.EnsureCreatedAsync();
    await DbSeeder.SeedAsync(dbContext);
}

await app.RunAsync();
```

Додаток Г. Контролер REST-операцій із заявками

```
using DeadlineControl.Api.Contracts.Tickets;
using DeadlineControl.Api.Domain.Enums;
using DeadlineControl.Api.Services.Tickets;
using Microsoft.AspNetCore.Mvc;

namespace DeadlineControl.Api.Controllers;

[ApiController]
[Route("api/[controller]")]
public class TicketsController : ControllerBase
{
    private readonly ITicketService _ticketService;

    public TicketsController(ITicketService ticketService)
    {
        _ticketService = ticketService;
    }

    [HttpGet]
    public async Task<ActionResult<IReadOnlyCollection<TicketResponse>>> Get(
        [FromQuery] TicketStatus? status,
        [FromQuery] TicketPriority? priority,
        [FromQuery] Guid? assigneeId,
        [FromQuery] bool? onlyOverdue,
        [FromQuery] bool? onlyDueSoon,
        CancellationToken cancellationToken)
    {
        var tickets = await _ticketService.GetAsync(status, priority, assigneeId, onlyOverdue,
onlyDueSoon, cancellationToken);
        return Ok(tickets);
    }

    [HttpGet("{id:guid}")]
    public async Task<ActionResult<TicketResponse>> GetById(Guid id, CancellationToken
cancellationToken)
    {
        var ticket = await _ticketService.GetByIdAsync(id, cancellationToken);
        return ticket is null ? NotFound() : Ok(ticket);
    }

    [HttpPost]
    public async Task<ActionResult<TicketResponse>> Create(CreateTicketRequest request,
CancellationToken cancellationToken)
    {
        var ticket = await _ticketService.CreateAsync(request, cancellationToken);
        return CreatedAtAction(nameof(GetById), new { id = ticket.Id }, ticket);
    }

    [HttpPut("{id:guid}")]
    public async Task<ActionResult<TicketResponse>> Update(Guid id, UpdateTicketRequest request,
CancellationToken cancellationToken)
    {
        var ticket = await _ticketService.UpdateAsync(id, request, cancellationToken);
        return ticket is null ? NotFound() : Ok(ticket);
    }

    [HttpPatch("{id:guid}/status")]
    public async Task<ActionResult<TicketResponse>> ChangeStatus(Guid id,
ChangeTicketStatusRequest request, CancellationToken cancellationToken)
    {
        var ticket = await _ticketService.ChangeStatusAsync(id, request, cancellationToken);
        return ticket is null ? NotFound() : Ok(ticket);
    }

    [HttpPatch("{id:guid}/assign")]
    public async Task<ActionResult<TicketResponse>> Assign(Guid id, AssignTicketRequest request,
CancellationToken cancellationToken)
```

```
    {
        var ticket = await _ticketService.AssignAsync(id, request, cancellationToken);
        return ticket is null ? NotFound() : Ok(ticket);
    }

    [HttpGet("{id:guid}/history")]
    public async Task<ActionResult<IReadOnlyCollection<TicketHistoryResponse>>> GetHistory(Guid
id, CancellationToken cancellationToken)
    {
        var history = await _ticketService.GetHistoryAsync(id, cancellationToken);
        return Ok(history);
    }
}
```

Додаток Д. Реалізація фільтрації, створення та зміни статусу в TicketService

```
public class TicketService : ITicketService
{
    public async Task<IReadOnlyCollection<TicketResponse>> GetAsync(
        TicketStatus? status,
        TicketPriority? priority,
        Guid? assigneeId,
        bool? onlyOverdue,
        bool? onlyDueSoon,
        CancellationToken cancellationToken)
    {
        var now = DateTime.UtcNow;
        var dueSoonLimit = now.AddHours(24);

        var query = _dbContext.Tickets
            .AsNoTracking()
            .Include(x => x.CreatedByUser)
            .Include(x => x.Assignee)
            .Include(x => x.Reminders)
            .AsQueryable();

        if (status.HasValue)
        {
            query = query.Where(x => x.Status == status.Value);
        }

        if (priority.HasValue)
        {
            query = query.Where(x => x.Priority == priority.Value);
        }

        if (assigneeId.HasValue)
        {
            query = query.Where(x => x.AssigneeId == assigneeId.Value);
        }

        if (onlyOverdue == true)
        {
            query = query.Where(x =>
                x.DueAt < now &&
                x.Status != TicketStatus.Completed &&
                x.Status != TicketStatus.Closed &&
                x.Status != TicketStatus.Cancelled);
        }

        if (onlyDueSoon == true)
        {
            query = query.Where(x =>
                x.DueAt >= now &&
                x.DueAt <= dueSoonLimit &&
                x.Status != TicketStatus.Completed &&
                x.Status != TicketStatus.Closed &&
                x.Status != TicketStatus.Cancelled);
        }

        var tickets = await query
            .OrderBy(x => x.DueAt)
            .ToListAsync(cancellationToken);

        return tickets.Select(x => ToTicketResponse(x, now)).ToList();
    }

    public async Task<TicketResponse?> GetByIdAsync(Guid id, CancellationToken cancellationToken)
    {
        var now = DateTime.UtcNow;

        var ticket = await _dbContext.Tickets
            .AsNoTracking()
```

```
            .Include(x => x.CreatedByUser)
            .Include(x => x.Assignee)
            .Include(x => x.Reminders)
            .FirstOrDefaultAsync(x => x.Id == id, cancellationToken);

        return ticket is null ? null : ToTicketResponse(ticket, now);
    }

    public async Task<TicketResponse> CreateAsync(CreateTicketRequest request, CancellationToken
cancellationToken)
    {
        await EnsureUserExistsAsync(request.CreatedByUserId, cancellationToken);

        if (request.AssigneeId.HasValue)
        {
            await EnsureUserExistsAsync(request.AssigneeId.Value, cancellationToken);
        }

        var now = DateTime.UtcNow;
        var ticket = new Ticket
        {
            Id = Guid.NewGuid(),
            Title = request.Title.Trim(),
            Description = request.Description.Trim(),
            Priority = request.Priority,
            Status = TicketStatus.New,
            CreatedAt = now,
            UpdatedAt = now,
            DueAt = request.DueAt.ToUniversalTime(),
            CreatedByUserId = request.CreatedByUserId,
            AssigneeId = request.AssigneeId
        };

        _dbContext.Tickets.Add(ticket);
        AddHistory(ticket.Id, "Created", "Заявку створено.", null, ticket.Status.ToString(),
request.CreatedByUserId, now);

        await _dbContext.SaveChangesAsync(cancellationToken);

        // If the ticket is already inside the notification window (or overdue),
        // evaluate it immediately. The BackgroundService continues to handle
        // tickets that enter the window later.
        await _reminderProcessor.ProcessTicketAsync(ticket.Id, cancellationToken);

        var createdTicket = await GetByIdAsync(ticket.Id, cancellationToken);
        return createdTicket!;
    }

    public async Task<TicketResponse?> ChangeStatusAsync(Guid id, ChangeTicketStatusRequest
request, CancellationToken cancellationToken)
    {
        await EnsureUserExistsAsync(request.ChangedByUserId, cancellationToken);

        var ticket = await _dbContext.Tickets.FirstOrDefaultAsync(x => x.Id == id,
cancellationToken);
        if (ticket is null)
        {
            return null;
        }

        var now = DateTime.UtcNow;
        var oldStatus = ticket.Status;

        ticket.Status = request.Status;
        ticket.UpdatedAt = now;
        ticket.CompletedAt = request.Status is TicketStatus.Completed or TicketStatus.Closed ?
now : null;

        AddHistory(ticket.Id, "StatusChanged", "Змінено статус заявки.", oldStatus.ToString(),
request.Status.ToString(), request.ChangedByUserId, now);
```

```
            await _dbContext.SaveChangesAsync(cancellationToken);
            await _reminderProcessor.ProcessTicketAsync(ticket.Id, cancellationToken);

            return await GetByIdAsync(id, cancellationToken);
        }
    }
```

Додаток Е. Фоновий цикл DeadlineReminderBackgroundService

```
using DeadlineControl.Api.Options;
using Microsoft.Extensions.Options;

namespace DeadlineControl.Api.BackgroundServices;

public sealed class DeadlineReminderBackgroundService : BackgroundService
{
    private readonly DeadlineReminderProcessor _processor;
    private readonly ILogger<DeadlineReminderBackgroundService> _logger;
    private readonly DeadlineReminderOptions _options;

    public DeadlineReminderBackgroundService(
        DeadlineReminderProcessor processor,
        IOptions<DeadlineReminderOptions> options,
        ILogger<DeadlineReminderBackgroundService> logger)
    {
        _processor = processor;
        _logger = logger;
        _options = options.Value;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        var intervalSeconds = Math.Max(_options.CheckIntervalSeconds, 5);
        _logger.LogInformation(
            "Deadline reminder background service started. Check interval: {CheckIntervalSeconds}
seconds; notify before: {NotifyBeforeMinutes} minutes.",
            intervalSeconds,
            Math.Max(_options.NotifyBeforeMinutes, 1));

        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                // The first check is performed immediately after the host starts.
                // Each processor call creates its own DI scope and DbContext.
                await _processor.ProcessAsync(stoppingToken);
            }
            catch (OperationCanceledException) when (stoppingToken.IsCancellationRequested)
            {
                break;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "An error occurred while checking ticket deadlines.");
            }

            try
            {
                await Task.Delay(TimeSpan.FromSeconds(intervalSeconds), stoppingToken);
            }
            catch (OperationCanceledException) when (stoppingToken.IsCancellationRequested)
            {
                break;
            }
        }

        _logger.LogInformation("Deadline reminder background service stopped.");
    }
}
```

Додаток Ж. Процесор автоматизованих нагадувань

```
using DeadlineControl.Api.Domain.Entities;
using DeadlineControl.Api.Domain.Enums;
using DeadlineControl.Api.Infrastructure;
using DeadlineControl.Api.Options;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;

namespace DeadlineControl.Api.BackgroundServices;

/// <summary>
/// Creates deadline reminders using a fresh dependency-injection scope and a fresh
/// DbContext for every processing run. The processor intentionally works with
/// no-tracking projections and only inserts new Reminder entities, so it does not
/// keep stale Ticket entities in the change tracker.
/// </summary>
public sealed class DeadlineReminderProcessor
{
    private readonly IServiceScopeFactory _scopeFactory;
    private readonly DeadlineReminderOptions _options;
    private readonly ReminderProcessingState _state;
    private readonly ILogger<DeadlineReminderProcessor> _logger;

    public DeadlineReminderProcessor(
        IServiceScopeFactory scopeFactory,
        IOptions<DeadlineReminderOptions> options,
        ReminderProcessingState state,
        ILogger<DeadlineReminderProcessor> logger)
    {
        _scopeFactory = scopeFactory;
        _options = options.Value;
        _state = state;
        _logger = logger;
    }

    public async Task<ReminderProcessingResult> ProcessAsync(CancellationToken cancellationToken)
    {
        await _state.ProcessingLock.WaitAsync(cancellationToken);
        var startedAtUtc = DateTime.UtcNow;
        _state.MarkStarted(startedAtUtc);

        try
        {
            await using var scope = _scopeFactory.CreateAsyncScope();
            var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

            var tickets = await dbContext.Tickets
                .AsNoTracking()
                .Where(x => x.Status != TicketStatus.Completed &&
                            x.Status != TicketStatus.Closed &&
                            x.Status != TicketStatus.Cancelled)
                .Select(x => new TicketReminderCandidate(
                    x.Id,
                    x.Title,
                    x.DueAt))
                .ToListAsync(cancellationToken);

            var result = await ProcessTicketsAsync(
                dbContext,
                tickets,
                startedAtUtc,
                cancellationToken);

            _state.MarkCompleted(result);
            return result;
        }
        catch (Exception ex)
        {
```

```
            _state.MarkFailed(startedAtUtc, ex);
            _logger.LogError(ex, "Deadline reminder processing failed.");
            throw;
        }
        finally
        {
            _state.ProcessingLock.Release();
        }
    }

    public async Task<ReminderProcessingResult> ProcessTicketAsync(
        Guid ticketId,
        CancellationToken cancellationToken)
    {
        await _state.ProcessingLock.WaitAsync(cancellationToken);
        var startedAtUtc = DateTime.UtcNow;
        _state.MarkStarted(startedAtUtc);

        try
        {
            await using var scope = _scopeFactory.CreateAsyncScope();
            var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

            var ticket = await dbContext.Tickets
                .AsNoTracking()
                .Where(x => x.Id == ticketId &&
                            x.Status != TicketStatus.Completed &&
                            x.Status != TicketStatus.Closed &&
                            x.Status != TicketStatus.Cancelled)
                .Select(x => new TicketReminderCandidate(
                    x.Id,
                    x.Title,
                    x.DueAt))
                .FirstOrDefaultAsync(cancellationToken);

            IReadOnlyCollection<TicketReminderCandidate> tickets = ticket is null
                ? Array.Empty<TicketReminderCandidate>()
                : new[] { ticket };

            var result = await ProcessTicketsAsync(
                dbContext,
                tickets,
                startedAtUtc,
                cancellationToken);

            _state.MarkCompleted(result);
            return result;
        }
        catch (Exception ex)
        {
            _state.MarkFailed(startedAtUtc, ex);
            _logger.LogError(ex, "Deadline reminder processing failed for ticket {TicketId}.",
ticketId);
            throw;
        }
        finally
        {
            _state.ProcessingLock.Release();
        }
    }

    private async Task<ReminderProcessingResult> ProcessTicketsAsync(
        AppDbContext dbContext,
        IReadOnlyCollection<TicketReminderCandidate> tickets,
        DateTime startedAtUtc,
        CancellationToken cancellationToken)
    {
        var now = DateTime.UtcNow;
        var notifyBeforeMinutes = Math.Max(_options.NotifyBeforeMinutes, 1);
        var reminderThreshold = now.AddMinutes(notifyBeforeMinutes);
```

```
var ticketIds = tickets.Select(x => x.Id).ToArray();

var existingReminderKeys = ticketIds.Length == 0
    ? new HashSet<ReminderKey>()
    : (await dbContext.Reminders
        .AsNoTracking()
        .Where(x => ticketIds.Contains(x.TicketId))
        .Select(x => new ReminderKey(x.TicketId, x.Type))
        .ToListAsync(cancellationToken))
        .ToHashSet();

var remindersToCreate = new List<Reminder>();
var deadlineApproachingCreatedCount = 0;
var overdueCreatedCount = 0;

foreach (var ticket in tickets)
{
    ReminderType? reminderType = null;

    if (ticket.DueAt <= now)
    {
        reminderType = ReminderType.Overdue;
    }
    else if (ticket.DueAt <= reminderThreshold)
    {
        reminderType = ReminderType.DeadlineApproaching;
    }

    if (!reminderType.HasValue)
    {
        continue;
    }

    var key = new ReminderKey(ticket.Id, reminderType.Value);
    if (!existingReminderKeys.Add(key))
    {
        continue;
    }

    remindersToCreate.Add(new Reminder
    {
        Id = Guid.NewGuid(),
        TicketId = ticket.Id,
        Type = reminderType.Value,
        Status = ReminderStatus.Sent,
        Message = BuildMessage(ticket, reminderType.Value),
        CreatedAt = now,
        SentAt = now
    });

    if (reminderType == ReminderType.Overdue)
    {
        overdueCreatedCount++;
    }
    else
    {
        deadlineApproachingCreatedCount++;
    }
}

if (remindersToCreate.Count > 0)
{
    dbContext.Reminders.AddRange(remindersToCreate);
    await dbContext.SaveChangesAsync(cancellationToken);
}

var result = new ReminderProcessingResult
{
    StartedAtUtc = startedAtUtc,
```

```
            CompletedAtUtc = DateTime.UtcNow,
            CheckedTicketCount = tickets.Count,
            CreatedReminderCount = remindersToCreate.Count,
            DeadlineApproachingCreatedCount = deadlineApproachingCreatedCount,
            OverdueCreatedCount = overdueCreatedCount
        };

        _logger.LogInformation(
            "Deadline reminder check completed. Checked: {CheckedTicketCount}; created:
{CreatedReminderCount}; approaching: {ApproachingCount}; overdue: {OverdueCount}; UTC now:
{UtcNow}; threshold: {ThresholdUtc}.",
            result.CheckedTicketCount,
            result.CreatedReminderCount,
            result.DeadlineApproachingCreatedCount,
            result.OverdueCreatedCount,
            now,
            reminderThreshold);

        return result;
    }

    private static string BuildMessage(
        TicketReminderCandidate ticket,
        ReminderType type)
    {
        return type switch
        {
            ReminderType.DeadlineApproaching =>
                $"Строк виконання заявки '{ticket.Title}' наближається. Дедлайн:
{ticket.DueAt:yyyy-MM-dd HH:mm} UTC.",
            ReminderType.Overdue =>
                $"Заявка '{ticket.Title}' прострочена. Дедлайн був: {ticket.DueAt:yyyy-MM-dd
HH:mm} UTC.",
            _ => $"Нагадування по заявці '{ticket.Title}'."
        };
    }

    private sealed record TicketReminderCandidate(
        Guid Id,
        string Title,
        DateTime DueAt);

    private sealed record ReminderKey(
        Guid TicketId,
        ReminderType Type);
}
```

Додаток З. Конфігурація реляційної моделі в AppDbContext

```
using DeadlineControl.Api.Domain.Entities;
using DeadlineControl.Api.Domain.Enums;
using Microsoft.EntityFrameworkCore;

namespace DeadlineControl.Api.Infrastructure;

public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
    {
    }

    public DbSet<AppUser> Users => Set<AppUser>();
    public DbSet<Ticket> Tickets => Set<Ticket>();
    public DbSet<TicketHistory> TicketHistory => Set<TicketHistory>();
    public DbSet<Reminder> Reminders => Set<Reminder>();

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

        modelBuilder.Entity<AppUser>(entity =>
        {
            entity.ToTable("users");
            entity.HasKey(x => x.Id);
            entity.Property(x => x.FullName).HasMaxLength(150).IsRequired();
            entity.Property(x => x.Email).HasMaxLength(150).IsRequired();
            entity.Property(x => x.Role).HasMaxLength(50).IsRequired();
            entity.HasIndex(x => x.Email).IsUnique();
        });

        modelBuilder.Entity<Ticket>(entity =>
        {
            entity.ToTable("tickets");
            entity.HasKey(x => x.Id);
            entity.Property(x => x.Title).HasMaxLength(200).IsRequired();
            entity.Property(x => x.Description).HasMaxLength(2000).IsRequired();
            entity.Property(x => x.Status).HasConversion<string>().HasMaxLength(50).IsRequired();
            entity.Property(x =>
x.Priority).HasConversion<string>().HasMaxLength(50).IsRequired();
            entity.HasIndex(x => x.Status);
            entity.HasIndex(x => x.Priority);
            entity.HasIndex(x => x.DueAt);
            entity.HasIndex(x => x.AssigneeId);

            entity.HasOne(x => x.CreatedByUser)
                .WithMany(x => x.CreatedTickets)
                .HasForeignKey(x => x.CreatedByUserId)
                .OnDelete(DeleteBehavior.Restrict);

            entity.HasOne(x => x.Assignee)
                .WithMany(x => x.AssignedTickets)
                .HasForeignKey(x => x.AssigneeId)
                .OnDelete(DeleteBehavior.Restrict);
        });

        modelBuilder.Entity<TicketHistory>(entity =>
        {
            entity.ToTable("ticket_history");
            entity.HasKey(x => x.Id);
            entity.Property(x => x.EventType).HasMaxLength(100).IsRequired();
            entity.Property(x => x.Description).HasMaxLength(1000).IsRequired();
            entity.Property(x => x.OldValue).HasMaxLength(1000);
            entity.Property(x => x.NewValue).HasMaxLength(1000);
            entity.HasIndex(x => x.TicketId);
            entity.HasIndex(x => x.CreatedAt);
```

```
            entity.HasOne(x => x.Ticket)
                .WithMany(x => x.History)
                .HasForeignKey(x => x.TicketId)
                .OnDelete(DeleteBehavior.Cascade);

            entity.HasOne(x => x.ChangedByUser)
                .WithMany()
                .HasForeignKey(x => x.ChangedByUserId)
                .OnDelete(DeleteBehavior.Restrict);
        });

        modelBuilder.Entity<Reminder>(entity =>
        {
            entity.ToTable("reminders");
            entity.HasKey(x => x.Id);
            entity.Property(x => x.Type).HasConversion<string>().HasMaxLength(50).IsRequired();
            entity.Property(x => x.Status).HasConversion<string>().HasMaxLength(50).IsRequired();
            entity.Property(x => x.Message).HasMaxLength(1000).IsRequired();
            entity.HasIndex(x => x.TicketId);
            entity.HasIndex(x => new { x.TicketId, x.Type }).IsUnique();
            entity.HasIndex(x => x.Status);

            entity.HasOne(x => x.Ticket)
                .WithMany(x => x.Reminders)
                .HasForeignKey(x => x.TicketId)
                .OnDelete(DeleteBehavior.Cascade);
        });
    }
}
```

Додаток И. Docker Compose для PostgreSQL і Adminer

```
services:
  postgres:
    image: postgres:16-alpine
    container_name: deadline-control-postgres
    restart: unless-stopped
    environment:
      POSTGRES_DB: deadline_control_db
      POSTGRES_USER: deadline_user
      POSTGRES_PASSWORD: deadline_password
    ports:
      - "5432:5432"
    volumes:
      - deadline_postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U deadline_user -d deadline_control_db"]
      interval: 5s
      timeout: 5s
      retries: 10

  adminer:
    image: adminer:4
    container_name: deadline-control-adminer
    restart: unless-stopped
    depends_on:
      postgres:
        condition: service_healthy
    ports:
      - "8080:8080"

volumes:
  deadline_postgres_data:
```

Додаток К. Типізована конфігурація механізму нагадувань

```
namespace DeadlineControl.Api.Options;

public class DeadlineReminderOptions
{
    public int CheckIntervalSeconds { get; set; } = 30;
    public int NotifyBeforeMinutes { get; set; } = 2;
}


{
  "ConnectionStrings": {
    "DefaultConnection":
"Host=localhost;Port=5432;Database=deadline_control_db;Username=deadline_user;Password=deadline_p
assword"
  },
  "DeadlineReminder": {
    "CheckIntervalSeconds": 10,
    "NotifyBeforeMinutes": 5
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning",
      "Microsoft.EntityFrameworkCore.Database.Command": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Додаток Л. Репозиторій програмного проєкту

Актуальна версія програмного проєкту, README та план тестування доступні за адресою: https://github.com/Artemmal/DeadlineControl