НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

«До захисту допущено»
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
“___”_____2025р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою
«Цифрові технології в енергетиці»
зі спеціальності 122 «Комп’ютерні науки»

на тему: «Веб-додаток для взаємодії між волонтерами»

Виконав: студент 4 курсу, групи ТР-11

Бабенко Артем Анатолійович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник асистент каф. ЦТЕ, Володимир РУДИК _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Рецензент професор каф. ТАЕ, д.т.н., Михайло АБДУЛІН _____
(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ) (підпис)

Н.контроль старший викладач каф. ЦТЕ, Ольга БЕСПАЛА _____
(посада, ім’я, ПРІЗВИЩЕ) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ
Рівень вищої освіти – перший (бакалаврський)
спеціальність 122 «Комп'ютерні науки»
Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ
Завідувач кафедри ЦТЕ
_____ Наталія АУШЕВА
(підпис)
"___" _____ 2025 р.

З А В Д А Н Н Я
на дипломну роботу студенту

Бабенку Артему Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Веб-додаток для взаємодії між волонтерами»
Науковий керівник Рудик Володимир Іванович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від "02" червня 2025 року № 1875-с

2. Термін подання студентом: роботи 09.06.2025р

3. Вихідні дані до роботи: мова програмування C#, фреймворк ASP.NET, СУБД MS SQL, середовище розробки Visual Studio 2022

4. Перелік питань, які потрібно розробити: 1) провести аналіз задачі розробки веб-додатку; 2) визначити та обґрунтувати вибір технологій для реалізації веб-додатку; 3) реалізувати ключові функції: створення завдань, спілкування між волонтерами, систему оповіщен; 4) забезпечити комфортний інтерфейс для зручного користування; 5) розробити програмний застосунок для спрощення взаємодії між волонтерами.

5. Орієнтований перелік ілюстративного матеріалу: діаграма прецедентів (ролей), структура бази даних, скріншоти роботи користувачів з веб-додатком.

6. Дата видачі завдання 13.09.2024р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вибір теми роботи	13.10.24	Виконано
2.	Аналіз методів та засобів розв’язання задачі	20.04.2025	Виконано
3.	Розробка архітектури та загальної структури системи	27.04.25	Виконано
4.	Розробка окремих підсистем	04.05.25	Виконано
5.	Програмна реалізація системи	11.05.25	Виконано
6.	Оформлення пояснювальної записки	12.05.25	Виконано
7.	Захист програмного забезпечення	14.05.25	Виконано
8.	Передзахист	28.05.25	Виконано
9.	Захист		Виконано

Студент _____ Артем БАБЕНКО
(підпис) (ім’я, ПРІЗВИЩЕ)

Керівник _____ Володимир РУДИК
(підпис) (ім’я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 71 сторінках, містить 26 ілюстрацій, 1 додаток, 20 джерел в переліку посилань.

Метою роботи є розробка та впровадження веб-додатку для оптимізації взаємодії між волонтерами, що забезпечує швидкий обмін інформацією, координацію спільних дій та ефективне управління завданнями.

Методи та засоби: використано монолітну багатошарову архітектуру з принципами «чистої архітектури», патерн CQRS із бібліотекою MediatR, фреймворк ASP.NET Core, технологію gRPC із HTTP/2 для синхронної взаємодії, Apache Kafka для асинхронної обробки повідомлень, SignalR для оновлення даних у реальному часі, Entity Framework Core для роботи з базою даних MSSQL, а також Docker для контейнеризації.

Основний зміст дипломної роботи включає аналіз задачі розробки веб-додатку, вибір оптимальних методів і засобів, таких як архітектура системи, аутентифікація через JWT, міжсервісна взаємодія через gRPC та Kafka, а також програмну реалізацію, що охоплює розробку клієнтської частини на ASP.NET MVC, серверної частини, структури бази даних та сценаріїв роботи користувачів, зокрема реєстрацію, авторизацію, створення, редагування та перегляд оголошень, відповіді на них, відображення статистики та відгуків.

Розроблений веб-додаток може бути впроваджений у волонтерських організаціях для покращення координації дій, оперативного обміну інформацією та підвищення ефективності роботи волонтерів.

ABSTRACT

The diploma thesis is completed on 71 pages, contains 26 illustrations, 1 appendices, and 20 references in the list of sources.

The purpose of the study is to develop and implement a web application designed to optimize interaction among volunteers, facilitating rapid information exchange, coordination of joint efforts, and efficient task management.

Methods and tools: a monolithic multi-layered architecture based on the principles of «clean architecture» was employed, incorporating the CQRS pattern with the MediatR library, the ASP.NET Core framework, gRPC technology with HTTP/2 for synchronous interaction, Apache Kafka for asynchronous message processing, SignalR for real-time data updates, Entity Framework Core for database management with MSSQL, and Docker for containerization.

The main content of the diploma thesis encompasses an analysis of the web application development task, the selection of optimal methods and tools, including system architecture, authentication via JWT, inter-service interaction through gRPC and Kafka, as well as the software implementation, which includes the development of the client-side using ASP.NET MVC, the server-side, database structure, and user interaction scenarios, specifically registration, authorization, creation, editing, and viewing of announcements, responses to them, display of statistics, and user reviews.

The developed web application can be implemented in volunteer organizations to enhance coordination of activities, enable prompt information exchange, and improve the overall efficiency of volunteers' work

ЗМІСТ

ВСТУП

Волонтерство є невід’ємною складовою сучасного суспільства, воно сприяє вирішенню важливих соціальних, гуманітарних та екологічних потреб. З огляду на стрімкий розвиток інформаційних технологій, особливої актуальності набуває розробка практичних та ефективних інструментів, які дозволяють дуже гарно оптимізувати взаємодію між волонтерами, координувати їхні зусилля, ефективно використовувати ресурси, які вони мають.

Потреба в розробці сучасного веб-додатку, який забезпечує ефективну комунікацію між волонтерами, підкреслює високу актуальність даної дипломної роботи. Веб-додаток дозволить волонтерам дуже швидко обмінюватись інформацією, планувати спільні дії та ефективно розподіляти завдання.

Метою даної дипломної роботи є розробка та впровадження працездатного веб-додатку, який відповідатиме сучасним вимогам до продуктивності, зручності у користуванні, а також враховуватиме особливості волонтерської діяльності.

Завдання даної роботи включають проведення аналізу задачі розробки веб-додатку, визначення та обґрунтування вибору технологій для розробки програмного застосунку для спрощення взаємодії між волонтерами.

Для досягнення цієї мети було обрано оптимальні технології, які забезпечують ефективну реалізацію функціоналу. Також було розроблено логічну архітектуру системи та інтуїтивно зрозумілий інтерфейс, який відповідатиме потребам користувача. Після реалізації веб-додатку було проведено його всебічне тестування з метою перевірки працездатності.

Отже, створення цього веб-додатку сприятиме підвищенню ефективності волонтерських ініціатив, покращить координацію зусиль волонтерів і підвищить результативність роботи волонтерських організацій.

1 АНАЛІЗ ЗАДАЧІ РОЗРОБКИ ВЕБ-ДОДАТКУ ДЛЯ ВЗАЄМОДІЇ МІЖ ВОЛОНТЕРАМИ

Для ефективної розробки веб-додатку, який сприятиме взаємодії між волонтерами, необхідно чітко сформувати завдання та задачі, щоб забезпечити відповідність продукту потребам користувачів і сучасним стандартам якості.

Основним завданням є створення зручної та функціональної платформи, яка дозволить волонтерам швидко обмінюватися інформацією, координувати спільні дії та ефективно управляти ресурсами. Це вимагає детального аналізу вимог до функціоналу, безпеки, продуктивності та інтерфейсу користувача.

Не менш важливим є аналіз програмних засобів, який забезпечить вибір оптимальних технологій для реалізації проєкту.

На основі проведеного аналізу було сформовано технічне завдання, яке включає детальний опис функціональних і нефункціональних вимог, а також архітектури системи. Це дозволило закласти міцну основу для подальшого проєктування та розробки веб-додатку, який відповідатиме потребам волонтерів і сприятиме їхній ефективній взаємодії.

1.1 Постановка задачі

Розробка веб-додатку для взаємодії між волонтерами вимагає чіткого визначення задачі, яка враховуватиме потреби цільової аудиторії та особливості волонтерської діяльності. Основною задачею є створення платформи, яка забезпечить ефективну координацію волонтерів, спростить організацію їхньої роботи та сприятиме оперативному обміну інформацією.

Веб-додаток має виконувати такі ключові функції: надавати можливість створювати та розподіляти завдання, забезпечувати зручне спілкування між учасниками. Крім того, система повинна підтримувати функцію сповіщень, щоб волонтери могли розуміти, хто відкликнувся на їхнє завдання. Важливим аспектом

є забезпечення доступності платформи для користувачів із різним рівнем технічної підготовки, що вимагає створення простого та інтуїтивно зрозумілого інтерфейсу [20].

Задача також включає вибір технологій, які забезпечать високу продуктивність і можливість масштабування системи в майбутньому.

1.2 Аналіз існуючих рішень

Перед початком розробки було проведено аналіз існуючих рішень у сфері програмного забезпечення призначеного для обміну інформацією між волонтерами.

VolunteerMatch надає користувачам можливість створювати оголошення про волонтерські можливості, реагувати на них, а також шукати завдання за географічним розташуванням чи інтересами. Платформа має зручний інтерфейс, який дозволяє волонтерам швидко знаходити відповідні проєкти, а організаціям публікувати свої оголошення.

Проте платформа має певні недоліки. Одним із основних є обмежена функціональність для прямого спілкування між волонтерами в реальному часі, що може ускладнювати оперативну координацію. VolunteerMatch орієнтований переважно на ринок США, що робить його менш адаптованим до потреб локальних спільнот в інших країнах, зокрема в Україні.

Крім того, відсутність асинхронних систем сповіщень, таких як email-повідомленнями, знижує ефективність оперативного інформування.

Іншим існуючим рішенням є використання оголошень на OLX та волонтерських груп у месенджерах, таких як Telegram і Facebook.

Платформа OLX надає користувачам можливість створювати оголошення про волонтерські ініціативи, наприклад, збір допомоги чи пошук добровольців, що забезпечує широке охоплення аудиторії завдяки великій базі користувачів.

Перевагою OLX є простота розміщення оголошень і можливість категоризації, що дозволяє волонтерам швидко знаходити потрібні ініціативи за регіоном чи типом діяльності.

Волонтерські групи в Telegram і Facebook забезпечують швидкий обмін інформацією, дозволяючи учасникам обговорювати завдання в реальному часі, ділитися ресурсами та координувати дії.

Проте ці рішення мають суттєві недоліки. На OLX оголошення на платформі часто змішуються з комерційними пропозиціями, що знижує їхню видимість для цільової аудиторії.

Волонтерські групи в Telegram і Facebook страждають від низької структурованості, інформація може губитися в потоці повідомлень, а відсутність централізованого управління ускладнює контроль за виконанням завдань. Крім того, модерація в таких групах не завжди ефективна, що може призводити до поширення недостовірної інформаці

1.3 Огляд методів

Для створення ефективного веб-додатку, спрямованого на оптимізацію взаємодії між волонтерами, необхідно ретельно проаналізувати сучасні методи розробки програмного забезпечення.

У цьому підрозділі розглядаються ключові підходи до побудови архітектури системи, забезпечення безпеки через методи аутентифікації та організації міжсервісної взаємодії, які відповідають специфіці проєкту, реалізованого на мові C# із використанням сучасних технологій.

Кожен метод аналізується з точки зору його переваг, недоліків і доцільності застосування в контексті волонтерської платформи.

1.3.1 Варіанти архітектури системи

Архітектура системи визначає організацію компонентів веб-додатку, їхню взаємодію, а також впливає на можливості масштабування та зручність підтримки. Для розробки веб-додатку, призначеного для взаємодії між волонтерами, проведено аналіз різних архітектурних підходів, щоб визначити оптимальний варіант із урахуванням специфіки завдання.

Спочатку розглянуто монолітну архітектуру [5], яка передбачає об’єднання всіх складових системи, а саме клієнтської частини, серверної логіки та бази даних, у єдиний додаток, що розгортається як цілісна одиниця. Аналіз показав, що монолітна архітектура вирізняється простотою реалізації на початкових етапах розробки.

Наприклад, створення базового функціоналу для управління завданнями волонтерів, такого як додавання чи редагування записів, потребує менше зусиль, адже всі компоненти зосереджені в одному місці, що спрощує налаштування, тестування та відлагодження.

У даному контексті монолітна архітектура реалізована у багатошаровому вигляді, де є чіткий поділ на шари: шар представлення та шар бізнес-логіки, які взаємодіють через ефективний протокол зв’язку, а розгортання здійснюється за допомогою контейнеризації. Це забезпечує зручність розробки та підтримки на ранніх етапах, адже всі частини додатку тісно пов’язані, що полегшує управління залежностями.

Проте в ході дослідження виявлено суттєві обмеження цього підходу. Якщо кількість користувачів платформи зросте через активне залучення волонтерів до масштабних ініціатив, масштабувати доведеться всю систему, навіть якщо навантаження стосується лише однієї функції. Таким чином, багатошарова монолітна архітектура є прийнятним рішенням для невеликих проєктів із простим функціоналом, але для платформи волонтерів, яка потребує гнучкості та масштабованості, вона має певні обмеження.

Далі було проаналізовано мікросервісну архітектуру [17]. Цей підхід передбачає поділ системи на невеликі незалежні сервіси, кожен із яких відповідає за окрему функцію.

Наприклад, один сервіс може обробляти завдання волонтерів, інший може забезпечувати аутентифікацію, а ще один може відповідати за надсилання сповіщень. У ході аналізу встановлено, що мікросервісна архітектура дозволяє масштабувати лише ті компоненти, які зазнають найбільшого навантаження. Такий підхід також полегшує оновлення, тому що зміни в одному сервісі не зачіпають роботу інших, що дає змогу швидше впроваджувати нові функції.

Проте мікросервісна архітектура додає складності в управлінні. Кожен сервіс потрібно розгортати окремо, що потребує додаткових інструментів для координації, таких як контейнеризація, а також засобів для забезпечення взаємодії між сервісами, наприклад, через API чи брокери повідомлень.

Окрім того, інтеграційне тестування стає більш трудомістким, адже необхідно перевіряти коректність роботи всіх сервісів разом. Порівняно з багатошаровою монолітною архітектурою, мікросервіси забезпечують більшу гнучкість і масштабованість [19], але потребують значно більше зусиль для реалізації та підтримки, що може бути викликом для проєктів із обмеженими ресурсами.

Також було досліджено архітектуру, орієнтовану на події. У такому підході компоненти системи обмінюються асинхронними подіями через брокер повідомлень.

Наприклад, створення нового завдання може генерувати подію, яка обробляється сервісом сповіщень для надсилання повідомлень волонтерам. Аналіз показав, що цей підхід добре підходить для інтерактивних функцій, адже дозволяє обробляти події асинхронно, не блокуючи основну роботу системи.

Архітектура на основі подій підвищує стійкість системи, наприклад, якщо один сервіс тимчасово недоступний, подія залишиться в черзі та буде оброблена пізніше.

1.3.2 Методи аутентифікації

Безпека є критично важливим аспектом веб-додатку для волонтерів, адже платформа працює з персональними даними, такими як імена, контакти та деталі завдань. У цьому пункті проведено аналіз різних методів аутентифікації, щоб визначити, які з них найкраще відповідають потребам проєкту.

Спочатку розглянуто аутентифікацію за допомогою логіну та пароля. Цей метод є найпоширенішим і передбачає введення користувачем своїх облікових даних для доступу до системи. Аналіз показав, що такий підхід вирізняється простотою реалізації та зручністю для користувачів, адже більшість волонтерів уже знайомі з подібним способом входу.

Наприклад, волонтер може зареєструватися на платформі, вказавши електронну пошту як логін і створивши пароль, після чого використовувати ці дані для входу. Щоб зробити цей метод безпечнішим, можна додати шифрування паролів.

Однак у процесі дослідження було виявлено слабкі сторони цього підходу. Якщо користувачі обирають слабкі паролі, наприклад, занадто короткі чи передбачувані, система стає вразливою до атак. Для мінімізації цих ризиків необхідно впровадити вимоги до складності паролів і механізми захисту від атак, що додає складності в реалізації.

Далі було проаналізовано токен-базовану аутентифікацію за допомогою JSON Web Token (JWT). У цьому методі після успішного входу користувач отримує токен, який містить зашифровану інформацію, наприклад, ідентифікатор користувача та його роль.

Токен передається в заголовках HTTP-запитів для авторизації, що дозволяє користувачу отримувати доступ до захищених ресурсів, таких як список завдань. Токени також можна підписувати секретним ключем, щоб відкинути можливість їх підробити. Для підвищення зручності можна використовувати refresh tokens, які дозволяють оновлювати токени без повторного введення пароля.

Однак JWT потребує захисту від атак, таких як XSS (Cross-Site Scripting), що вимагає додаткових заходів на клієнтській стороні, наприклад, безпечного зберігання токенів.

Ще один підхід, який було розглянуто, це авторизація через OAuth 2.0. Цей метод дозволяє користувачам входити в систему через сторонні сервіси, такі як Google, Apple або Facebook.

Наприклад, волонтер може натиснути кнопку «Увійти через Google» і після підтвердження отримати доступ до платформи, не створюючи новий обліковий запис. Аналіз показав, що OAuth 2.0 зручний для користувачів, адже економить час на реєстрацію, а безпека забезпечується перевіреними провайдерами. Це також може бути корисно для волонтерів, які вже мають облікові записи в таких сервісах і не хочуть запам’ятовувати додаткові паролі.

Однак інтеграція з API сторонніх сервісів потребує додаткових зусиль, таких як налаштування OAuth-клієнта та обробка токенів доступу. Для платформи волонтерів, де більшість користувачів може віддавати перевагу локальним обліковим записам, OAuth 2.0 здається надмірним, але його можна розглядати як додаткову опцію в майбутньому.

Також було досліджено багатофакторну аутентифікацію (MFA). Цей метод поєднує кілька рівнів перевірки, наприклад, пароль і одноразовий код, надісланий через SMS, електронну пошту або додаток-аутентифікатор. У ході аналізу встановлено, що MFA значно підвищує безпеку, адже навіть якщо зловмисник дізнається пароль, він не зможе увійти без другого фактора.

Однак для звичайних волонтерів MFA може бути незручною, особливо якщо вони працюють у польових умовах із обмеженим доступом до інтернету. До того ж надсилання кодів через SMS потребує інтеграції з платними сервісами, що додає витрат.

1.3.3 Способи міжсервісної взаємодії

Міжсервісна взаємодія [8] відіграє ключову роль у забезпеченні злагодженої роботи компонентів веб-додатку, призначеного для взаємодії між волонтерами. Проведено аналіз різних підходів до організації такої взаємодії, щоб визначити оптимальні рішення для ефективного функціонування платформи.

Спочатку розглянуто використання REST API. Цей підхід передбачає обмін даними між компонентами через HTTP-запити, такі як GET для отримання інформації, POST для створення, PUT для оновлення та DELETE для видалення. Аналіз показав, що REST API є універсальним і широко сумісним із різними платформами, що полегшує інтеграцію.

Наприклад, запит на отримання списку завдань волонтерів може бути реалізований через GET-запит, який повертає дані у форматі JSON. Такий метод вирізняється простотою, розробники легко визначають призначення кожного запиту, а формат JSON спрощує обробку на стороні клієнта.

Проте в ході дослідження виявлено обмеження синхронного режиму роботи REST API. Якщо один із компонентів, стає недоступним або працює повільно, це може затримати всю операцію, що негативно впливає на користувацький досвід. Крім того, для систем із високим навантаженням REST API є менш ефективним, адже кожен запит потребує окремого з’єднання, що може призвести до перевантаження.

Також було проаналізовано використання gRPC із протоколом HTTP/2. Цей метод базується на швидкій передачі даних завдяки мультиплексуванню та стисненню, які забезпечує HTTP/2. У ході дослідження встановлено, що gRPC перевершує REST за продуктивністю, адже зменшує обсяг даних завдяки Protocol Buffers із сильною типізацією.

Наприклад, передача інформації про нове завдання від одного компонента до іншого може відбуватися значно швидше, що корисно для оперативної обробки запитів.

Проте цей підхід потребує додаткових зусиль на етапі налаштування, оскільки необхідно генерувати код на основі визначень Protocol Buffers. Окрім того, менша універсальність gRPC порівняно з REST може ускладнити інтеграцію з деякими платформами, які не підтримують HTTP/2. Незважаючи на це, gRPC є перспективним для сценаріїв, де критичною є висока швидкість обробки.

1.4 Огляд програмних засобів

Розробка веб-додатку для взаємодії між волонтерами потребує ґрунтовного аналізу програмних засобів, які здатні забезпечити високу продуктивність, надійність та зручність використання платформи.

Проведено всебічний огляд ключових категорій інструментів, зокрема систем управління базами даних, технологій для серверної та клієнтської частин, з урахуванням специфіки завдань, пов’язаних із координацією волонтерської діяльності.

Аналіз спрямований на визначення оптимальних рішень, які сприятимуть ефективній реалізації функціоналу, підтримці масштабованості та адаптивності системи до змінних потреб користувачів, з урахуванням використання мови програмування C# та сучасних технологій.

1.4.1 База даних

База даних є центральним елементом веб-додатку, призначеного для взаємодії між волонтерами, оскільки забезпечує зберігання та обробку даних про користувачів, оголошення, відповіді на оголошення, теги, статистику діяльності та інше.

Проведено аналіз різних систем управління базами даних, щоб визначити найбільш підходящі рішення для реалізації платформи з урахуванням вимог до продуктивності, надійності та сумісності з екосистемою C#.

Спочатку розглянуто Microsoft SQL Server (MSSQL) як реляційну систему управління базами даних. Аналіз показав, що MSSQL вирізняється високою продуктивністю та стабільністю, що є критично важливим для платформ із великою кількістю одночасних операцій. Підтримка транзакцій гарантує цілісність даних у сценаріях, де одночасно створюються завдання та додаються пов’язані записи, наприклад, коментарі.

Сумісність із Entity Framework Core, популярним інструментом у C#, полегшує інтеграцію та роботу з базою даних у контексті розробки. Проте MSSQL потребує значних обчислювальних ресурсів, що може стати обмеженням для проєктів із обмеженим бюджетом, а також вимагає ретельного планування для забезпечення масштабованості.

Далі було проаналізовано MySQL як альтернативну реляційну систему з відкритим кодом. Дослідження виявило, що MySQL забезпечує високу продуктивність і легкість у налаштуванні, що робить її привабливою для веб-додатків із середнім навантаженням, наприклад, для управління завданнями волонтерів.

Підтримка транзакцій і широкий спектр інструментів для індексації даних дозволяє ефективно обробляти запити, такі як пошук активних волонтерів за регіонами. Завдяки своїй безкоштовній природі та широкій спільноті MySQL є доступним рішенням для проєктів із динамічним розвитком.

Проте складність масштабування у високонавантажених системах і менша інтеграція з деякими специфічними бібліотеками C# порівняно з MSSQL можуть бути недоліками, що потребують додаткового аналізу.

Окремо розглянуто PostgreSQL як ще одну реляційну систему з відкритим кодом. Аналіз показав, що PostgreSQL підтримує роботу з різними типами даних, зокрема JSONB, що може бути корисним для зберігання неструктурованої інформації, наприклад, метаданих про волонтерські заходи.

Безкоштовність і активна спільнота забезпечують доступ до оновлень, але складність адміністрування, зокрема при налаштуванні реплікації чи резервного копіювання, може вплинути на стабільність за високого навантаження.

1.4.2 Серверна частина

Серверна частина веб-додатку відповідає за обробку запитів, реалізацію бізнес-логіки та інтеграцію з базою даних, що є ключовими аспектами для забезпечення стабільної роботи платформи волонтерів. Проведено аналіз технологій і фреймворків для реалізації серверної частини з урахуванням використання мови C# та сучасних підходів до розробки.

Спочатку проаналізовано ASP.NET Core як основу для серверної частини. Дослідження показало, що ASP.NET Core є потужним фреймворком, який забезпечує високу продуктивність і гнучкість завдяки кросплатформності та асинхронній моделі обробки запитів.

Цей фреймворк добре інтегрується з іншими технологіями C#, такими як Entity Framework Core для роботи з базою даних чи MediatR для реалізації патерну CQRS. ASP.NET Core підтримує модульну структуру, що дозволяє чітко розділити бізнес-логіку, наприклад, для управління розподілом завдань чи обробки сповіщень.

Вбудовані механізми захисту від типових уразливостей, таких як SQL-ін’єкції чи XSS, підвищують безпеку платформи при роботі з персональними даними.

Проте висока ресурсоємність у сценаріях із великою кількістю одночасних запитів потребує оптимізації, наприклад, через контейнеризацію для ефективного розподілу ресурсів.

Далі було досліджено патерн «чиста архітектура» для організації серверної частини. Аналіз виявив, що цей підхід забезпечує чіткий поділ системи на шари з ізольованими відповідальностями, що полегшує тестування та підтримку коду.

Наприклад, шар бізнес-логіки може бути відокремлений від інфраструктури, що дає змогу адаптувати систему до змін, таких як оновлення бази даних. У поєднанні з MediatR цей патерн підтримує CQRS, що оптимізує обробку команд і запитів, що є цінним для складних систем із великою кількістю операцій.

Проте реалізація цього підходу додає складності на етапі проєктування, що потребує ретельного планування.

Окремо проаналізовано Apache Kafka для асинхронної обробки. Дослідження показало, що Kafka дозволяє ефективно управляти потоками повідомлень, зокрема для відправки сповіщень на поштовий сервіс.

Використання Kafka забезпечує високу пропускну здатність і стійкість до збоїв, адже події зберігаються в черзі й обробляються пізніше, якщо сервіс недоступний. Проте інтеграція Kafka з екосистемою C# потребує додаткових бібліотек і налаштувань, що може ускладнити розробку.

1.4.3 Клієнтська частина

Клієнтська частина веб-додатку відіграє ключову роль у забезпеченні зручного доступу до платформи для волонтерів. Проведено аналіз технологій для створення адаптивного та інтуїтивного інтерфейсу з урахуванням використання C# і пов’язаних інструментів.

Спочатку розглянуто ASP.NET MVC як основу для клієнтської частини. Аналіз показав, що ASP.NET MVC дозволяє створювати динамічні сторінки через серверний рендеринг, що забезпечує швидке завантаження контенту, наприклад, списків завдань чи профілів волонтерів.

Цей підхід добре інтегрується з екосистемою C#, що спрощує розробку та підтримку. Також серверний рендеринг сприяє кращій індексації пошуковими системами, що може підвищити видимість платформи.

Проте обмежена інтерактивність, пов’язана з необхідністю нових запитів до сервера для кожної дії, може уповільнити користувацький досвід у сценаріях, де потрібні миттєві оновлення.

Також було досліджено SignalR для роботи в реальному часі. SignalR забезпечує двосторонній зв’язок між клієнтом і сервером, що є ідеальним для оновлення даних на інтерфейсі в реальному часі, наприклад, відображення нових оголошень. SignalR підтримує WebSocket для швидкої передачі даних і сумісний із альтернативними протоколами, що забезпечує гнучкість.

Інтеграція з ASP.NET MVC дозволяє оновлювати інтерфейс без перезавантаження сторінки, що покращує зручність для користувачів. Проте використання SignalR створює додаткове навантаження на сервер через постійні з’єднання, що потребує оптимізації, наприклад, через балансування навантаження.

2 ВИБІР ТА ОБҐРУНТУВАННЯ ВИКОРИСТАНИХ МЕТОДІВ І ЗАСОБІВ

Розробка веб-додатку для взаємодії між волонтерами вимагає ретельного вибору методів і програмних засобів, які забезпечать ефективну реалізацію функціоналу, високу продуктивність і масштабованість платформи.

У цьому розділі представлено обґрунтування вибору технологій та підходів, які застосовуються для реалізації веб-додатку, з урахуванням специфіки координації волонтерської діяльності та використання мови програмування C#.

Аналіз базується на попередніх дослідженнях архітектури та програмних засобів, а також включає необхідні характеристики та параметри для їхнього налаштування.

2.1 Засоби для реалізації серверної частини

Серверна частина веб-додатку відіграє центральну роль у забезпеченні обробки запитів від користувачів, реалізації складної бізнес-логіки та координації між різними компонентами системи в межах монолітної багатошарової архітектури.

Проведено ґрунтовну оцінку методів і технологій, які дозволяють створити надійну та ефективну основу для платформи, здатної витримувати значні навантаження, пов’язані з взаємодією між волонтерами, і адаптуватися до майбутнього зростання кількості користувачів.

У цьому підрозділі розглядаються ключові аспекти реалізації серверної частини, включаючи вибір фреймворку, архітектурного патерну, підходів до управління запитами та технології розгортання, з урахуванням їхнього впливу на загальну продуктивність і стабільність системи.

2.1.1 Використання фреймворку

Для реалізації серверної частини веб-додатку обрано фреймворк ASP.NET Core, який зарекомендував себе як потужний інструмент для створення високопродуктивних і масштабованих веб-додатків. Вибір ASP.NET Core [6] обґрунтовано його численними перевагами, які ідеально відповідають вимогам платформи для волонтерів.

Передусім, цей фреймворк забезпечує глибоку інтеграцію з екосистемою C#, що дозволяє ефективно використовувати широкий спектр бібліотек і інструментів, таких як Entity Framework Core для роботи з базами даних MSSQL, MediatR для реалізації патерну CQRS і gRPC для швидкої міжшарової взаємодії. ASP.NET Core підтримує асинхронну обробку запитів, що є критично важливим для забезпечення швидкої реакції на дії користувачів, наприклад, під час створення нових завдань, оновлення статусів оголошення чи фільтрацію оголошень за тегами.

Модульна архітектура ASP.NET Core сприяє чіткому розподілу відповідальностей між різними аспектами системи.

Наприклад, бізнес-логіка, пов’язана з розподілом завдань між волонтерами, ізолюється від механізмів автентифікації, які використовують JWT (bearer) токени, передаючи їх у контексті запитів для ідентифікації користувачів. Це дозволяє розробникам зосередитися на окремих аспектах функціоналу, не створюючи зайвих залежностей між компонентами.

Крім того, ASP.NET Core забезпечує вбудовані механізми для захисту від типових загроз, таких як SQL-ін’єкції чи атаки типу XSS (Cross-Site Scripting), що є особливо важливим для платформи, яка працює з персональними даними волонтерів, включаючи їхні імена, контактну інформацію та деталі завдань.

Фреймворк також підтримує інтеграцію з gRPC, що дозволяє швидко передавати дані між UI і бекендом, а також із Kafka для асинхронної обробки повідомлень, які надсилаються на поштовий сервіс.

Використання ASP.NET Core створює міцну основу для платформи, яка може адаптуватися до змінних вимог волонтерської діяльності, підтримуючи високу

швидкість і надійність роботи навіть за значного навантаження, наприклад, під час координації масштабних ініціатив із залученням тисяч користувачів.

2.1.2 Патерн архітектури застосунку

Для організації серверної частини застосовано патерн «чиста архітектура» [4], який забезпечує створення чітко структованої системи з ізольованими шарами відповідальностей у межах монолітної багатошарової архітектури.

Цей вибір обґрунтовано його здатністю забезпечувати гнучкість, легкість підтримки та адаптивність системи до еволюційних змін, що є ключовими вимогами для платформи, яка може розширюватися з урахуванням нових потреб волонтерських організацій. У рамках «чистої архітектури» [13] система поділена на чотири основні шари: доменний, додатку, інфраструктурний і представлення, кожен із яких виконує специфічні функції, сприяючи логічній цілісності, прозорості коду та ефективному управлінню складністю.

Доменний шар зосереджується на визначенні основних сутностей і правил бізнес-логіки платформи, формуючи концептуальну основу для всіх операцій, що виконуються в системі.

Цей шар відповідає за визначення ключових об’єктів, таких як користувачі, їхні ролі, оголошення, відповіді на оголошення та теги, які використовуються для категоризації контенту. Він уособлює логіку, що регулює взаємодію між цими об’єктами, наприклад, правила призначення ролей користувачам, створення нових оголошень із відповідними атрибутами чи обробку відгуків і рейтингів, забезпечуючи єдину й послідовну основу для бізнес-процесів платформи. Завдяки цій ізоляції доменний шар [20] залишається незалежним від зовнішніх систем, що полегшує його тестування та модифікацію.

Шар додатків відіграє роль координаційного рівня, який відповідає за обробку запитів і команд, отриманих через gRPC від клієнтської частини, і передачу їх до відповідних обробників за допомогою MediatR.

Цей шар забезпечує послідовність і логічну цілісність бізнес-процесів, трансформуючи вхідні дані, такі як запити на створення оголошень, у команди, які передаються для подальшої обробки.

Наприклад, команда створення нового оголошення, отримана через gRPC, проходить через шар додатків, де валідуються її атрибути, такі як автор, час створення та вміст, і передається до відповідного обробника для реалізації. Цей підхід дозволяє ефективно розподіляти навантаження між різними типами операцій і забезпечує чітку структуру для розширення функціональності платформи.

Інфраструктурний шар забезпечує практичну взаємодію з зовнішніми системами, включаючи базу даних MSSQL, яка підтримує всі необхідні таблиці та асоціації, інтеграцію з Apache Kafka для асинхронної відправки повідомлень на поштовий сервіс, а також використання gRPC для синхронної взаємодії з UI.

Цей шар відповідає за реалізацію механізмів доступу до даних через Entity Framework Core, обробку потоків повідомлень для забезпечення надійної доставки сповіщень і підтримку двостороннього зв’язку з клієнтською частиною.

Крім того, інфраструктурний шар забезпечує гнучкість шляхом підтримки переходу на MySQL як альтернативну базу даних, що дозволяє адаптувати платформу до різних інфраструктурних умов, наприклад, у разі зміни вимог до апаратного забезпечення чи бюджетних обмежень. Ця гнучкість є особливо цінною для забезпечення безперервної роботи платформи в різних середовищах.

Шар представлення відповідає за обробку вхідних запитів від UI, створеного на основі ASP.NET MVC, і передачу даних через gRPC для подальшої обробки на серверній стороні.

Цей шар забезпечує безперервний цикл взаємодії між клієнтом і сервером, наприклад, обробляючи запити на перегляд списків оголошень чи відправку відповідей і передаючи їх через gRPC до відповідних компонентів.

Він також координує оновлення даних у реальному часі за допомогою SignalR, що дозволяє користувачам миттєво бачити зміни, такі як нові оголошення. Така організація сприяє ефективному управлінню потоками даних і забезпечує високу чутливість інтерфейсу до дій користувачів.

Ця чітко визначена структура дозволяє легко адаптувати систему до змін, наприклад, при додаванні нових функцій, таких як інтеграція з геолокаційними сервісами для відображення завдань на карті, розширення можливостей сповіщень через інтеграцію з месенджерами чи додавання аналітичних модулів для оцінки активності волонтерів.

Вона також полегшує модифікацію існуючих процесів, таких як оновлення логіки автентифікації чи розширення бази даних.

Використання «чистої архітектури» сприяє створенню системи, яка є стійкою до еволюційних змін, зберігаючи логічну цілісність, прозорість коду та модульність, що є особливо важливим для платформи, яка може розширюватися з урахуванням нових потреб волонтерських організацій, таких як підтримка нових типів взаємодій, інтеграція з додатковими каналами зв’язку чи впровадження автоматизованих процесів координації.

2.1.3 Управління запитами

Для управління запитами на серверній частині застосовано патерн Command Query Responsibility Segregation (CQRS) у поєднанні з бібліотекою MediatR [7], що дозволяє ефективно розділити операції запису (команди) і читання (запити) у межах монолітної архітектури [9].

Цей вибір обґрунтовано його здатністю оптимізувати продуктивність, спрощувати обробку складних бізнес-процесів, підвищувати масштабованість системи та забезпечувати чітку організацію коду, що є критично важливим для платформи, яка має справлятися з великими обсягами одночасних запитів від користувачів.

Патерн CQRS базується на принципі розподілу відповідальностей, де операції, що модифікують стан системи (наприклад, створення нового оголошення), обробляються як команди, а операції, що лише отримують дані (наприклад, перегляд списку оголошень), виконуються як запити.

Такий підхід дозволяє налаштувати окремі механізми для кожної категорії операцій, що сприяє підвищенню ефективності та зниженню навантаження на сервер, особливо в пікові періоди активності, коли платформа може обробляти тисячі запитів одночасно.

Бібліотека MediatR виступає як посередник, який забезпечує маршрутизацію запитів і команд до відповідних обробників, що значно спрощує організацію коду. У рамках цієї реалізації запити та команди надходять через gRPC у вигляді структурованих proto-описів, які трансформуються за допомогою бібліотеки AutoMapper у відповідні об’єкти бізнес-логіки.

Наприклад, команда створення нового оголошення, отримана від UI через gRPC, перетворюється на об’єкт команди, що містить атрибути, такі як ідентифікатор автора, заголовок, вміст, час публікації тощо, а потім передається до обробника через MediatR. Аналогічно, запит на отримання списку активних оголошень із фільтрацією за статусом чи тегами перетворюється на об’єкт запиту, який обробляється окремим обробником для повернення відфільтрованих даних.

Використання MediatR дозволяє централізувати логіку маршрутизації, зменшуючи залежності між компонентами та полегшуючи додавання нових типів запитів чи команд у майбутньому, що є важливим для еволюції платформи.

Механізми ін’єкції залежностей, вбудовані в ASP.NET Core, відіграють ключову роль у забезпеченні доступу до необхідних ресурсів у межах обробників.

Наприклад, обробник команди створення оголошення може отримувати доступ до репозиторію через інтерфейс, реалізований у шарі інфраструктури за допомогою Entity Framework Core, для збереження даних у базі даних MSSQL. Аналогічно, обробник запиту на перегляд списків може використовувати той самий репозиторій для отримання даних і їхньої передачі через gRPC назад до UI, де SignalR забезпечує оновлення інтерфейсу в реальному часі.

Ця інтеграція дозволяє ефективно управляти залежностями, уникаючи жорстких зв’язків між шарами, і забезпечує гнучкість при зміні джерел даних чи методів їхньої обробки, наприклад, при переході з MSSQL на MySQL.

Такий підхід до управління запитами дозволяє оптимізувати продуктивність платформи, зменшуючи навантаження на систему за рахунок паралельної обробки команд і запитів.

Наприклад, під час масових волонтерських кампаній, коли одночасно створюються нові оголошення, оновлюються статуси і переглядаються списки завдань, CQRS розподіляє ці операції між окремими потоками, що знижує ризик перевантаження сервера. Це особливо важливо для забезпечення стабільної роботи платформи в реальному часі, коли затримки можуть вплинути на координацію дій.

Крім того, поділ операцій запису і читання дозволяє оптимізувати кожну з них окремо, наприклад, команди можуть бути спрямовані на базу даних із сильною консистентністю, тоді як запити можуть використовувати кешування для прискорення доступу до даних, що підвищує загальну ефективність системи.

Прозорість у реалізації бізнес-логіки, яку забезпечує CQRS у поєднанні з MediatR, є ще однією значною перевагою. Кожен тип операції має свій обробник, що полегшує відстеження логіки та її модифікацію.

Наприклад, обробник команди створення оголошення може включати валідатор для перевірки унікальності ідентифікатора чи коректності вмісту, тоді як обробник запиту на перегляд може включати фільтри за часом публікації чи статусом. Ця структурованість сприяє швидкому виявленню та виправленню помилок, а також спрощує тестування окремих компонентів, що є важливим для забезпечення якості платформи.

Крім того, використання AutoMapper для трансформації даних між шарами зменшує кількість ручної маніпуляції об’єктами, підвищуючи надійність і скорочуючи час розробки.

Використання CQRS із MediatR створює гнучку основу для обробки запитів, яка може бути легко розширена для підтримки нових функцій, зберігаючи при цьому стабільність і ефективність системи.

Наприклад, додавання аналітичного модуля для оцінки активності волонтерів (наприклад, підрахунок кількості виконаних завдань за місяць) може бути реалізовано через нові запити, а інтеграція з зовнішніми сервісами для

автоматичного надсилання сповіщень через SMS чи створення звітів про ефективність роботи платформи через нові команди. Ця гнучкість дозволяє платформі еволюціонувати разом із потребами користувачів, адаптуючись до нових сценаріїв використання.

Крім того, підхід CQRS сприяє підвищенню масштабовності платформи, дозволяючи розподілити навантаження між різними серверами чи кластерами. Наприклад, операції запису можуть бути спрямовані на один сервер із високою надійністю, тоді як операції читання будуть спрямовані на інший із оптимізованим кешем, що є перспективним для майбутнього розширення інфраструктури.

Така архітектура також полегшує інтеграцію з іншими технологіями, такими як черги повідомлень для асинхронної обробки чи системи аналізу даних для генерації звітів, що може бути використано для покращення досвіду користувачів і оптимізації роботи платформи.

У контексті волонтерської діяльності, де важлива швидкість реагування та надійність, цей підхід забезпечує стабільність і ефективність навіть за значного зростання кількості користувачів.

2.1.4 Контейнеризація

Для розгортання серверної частини застосовано технологію контейнеризації за допомогою Docker [12], що забезпечує ізоляцію середовищ, спрощує процес масштабування, підвищує надійність платформи та сприяє уніфікації інфраструктури в межах монолітної архітектури.

Вибір Docker обґрунтовано його здатністю створювати легковагові, ізольовані контейнери, які включають усі залежності, бібліотеки та конфігурації, необхідні для роботи окремих компонентів системи, таких як бекенд на ASP.NET Core, сервер GRPC для взаємодії з UI, брокер повідомлень Apache Kafka для асинхронної обробки сповіщень, сервіс SignalR для оновлення даних у реальному часі та база даних MSSQL.

Ця технологія усуває проблеми сумісності, які можуть виникати при розгортанні системи на різних серверах чи у різних середовищах, забезпечуючи однакові умови роботи на всіх етапах від розробки та тестування до продакшену та обслуговування.

Контейнеризація дозволяє ізолювати кожен компонент платформи, що підвищує її стабільність, безпеку та стійкість до збоїв, а також спрощує процес оновлення та масштабування.

Однією з ключових переваг використання Docker є можливість створення незалежних контейнерів для кожного компонента системи.

Наприклад, бекенд на ASP.NET Core може працювати в одному контейнері, сервер gRPC може працювати в іншому, Kafka буде у третьому, а база даних MSSQL ─ у четвертому. Така ізоляція запобігає конфліктам між залежностями, такими як різні версії бібліотек чи драйверів, і дозволяє незалежно оновлювати чи перезапускати кожен контейнер без впливу на інші частини системи. Це особливо корисно для платформи волонтерів, яка може функціонувати в різних інфраструктурних умовах, від локальних серверів невеликих волонтерських організацій до великих хмарних платформ, таких як AWS чи Azure.

Наприклад, оновлення SignalR для покращення продуктивності оновлення інтерфейсу чи заміна версії Kafka для підвищення пропускної здатності повідомлень може бути виконано без зупинки всієї платформи, що забезпечує безперервну роботу навіть під час технічних втручань.

Контейнеризація також сприяє ефективному розподілу ресурсів, що є важливим для масштабування платформи в разі зростання кількості користувачів. Docker дозволяє швидко додавати нові екземпляри контейнерів для бекенду чи обробника Kafka, щоб впоратися з різким підвищенням навантаження, наприклад, під час масових волонтерських кампаній, коли тисячі користувачів одночасно створюють оголошення, переглядають статуси чи отримують сповіщення. У цьому випадку Docker дає можливість запускати кілька контейнерів на одному сервері, оптимізуючи використання апаратних ресурсів, таких як пам’ять і процесор, і забезпечуючи високу доступність.

Наприклад, під час координації екстрених дій, коли платформа переживає пік активності, можна запустити додатковий контейнер із бекендом для обробки запитів і контейнер із Kafka для забезпечення доставки сповіщень, зберігаючи стабільність і швидкість роботи системи. Такий підхід дозволяє платформі адаптуватися до змінного навантаження без значних витрат на інфраструктуру, що є економічно вигідним для волонтерських організацій з обмеженим бюджетом.

Крім того, контейнеризація спрощує процес розгортання та оновлення системи, що є важливим для забезпечення її актуальності та безпеки. Використання Docker дозволяє розробникам створювати єдині образи контейнерів, які включають усі залежності, конфігураційні файли та версії програмного забезпечення, що гарантує ідентичну поведінку системи в різних середовищах.

Наприклад, образ контейнера для бекенду може включати ASP.NET Core, Entity Framework Core і конфігурацію gRPC, тоді як образ для Kafka може включати налаштування брокера повідомлень і підключення до поштового сервісу. Ці образи можуть бути легко розгорнуті на будь-якому сервері з встановленим Docker, що зменшує час на налаштування та мінімізує ризик людських помилок. Процес оновлення також спрощується: замість ручного оновлення кожного сервера достатньо зібрати новий образ контейнера з оновленою версією коду чи залежностей і замінити старий контейнер, що забезпечує швидке і безпечне впровадження змін.

Ще однією значною перевагою контейнеризації є її внесок у безпеку платформи. Оскільки кожен контейнер ізольований від інших, компрометація одного компонента (наприклад, через вразливість у бекенді) не впливає на інші частини системи, такі як база даних чи сервер SignalR. Ця ізоляція підтримується механізмами Linux Containers (LXC) чи іншими технологіями, які використовуються Docker, що дозволяє обмежити доступ між контейнерами та мінімізувати ризик поширення загроз.

Наприклад, контейнер із базою даних може бути налаштований із суворими правилами мережевої ізоляції, дозволяючи доступ лише з бекенду, що захищає дані волонтерів від несанкціонованого доступу.

Контейнеризація також сприяє спрощенню процесів розробки та тестування. Розробники можуть використовувати Docker Compose для створення локальних оточень, які точно імітують продакшен, включаючи всі компоненти, такі як бекенд, gRPC, Kafka, SignalR і базу даних. Це дозволяє тестувати нові функції, такі як інтеграція з геолокаційними сервісами чи оновлення логіки сповіщень, у відокремленому середовищі без ризику впливу на робочу платформу.

Наприклад, можна створити складену конфігурацію з кількома контейнерами, підключеними через віртуальну мережу, для моделювання сценарію масового завантаження, що допомагає виявити потенційні слабкі місця ще на етапі розробки. Такий підхід скорочує час на налагодження та підвищує якість програмного забезпечення.

Нарешті, контейнеризація сприяє довготривалій підтримці та еволюції платформи. Використання Docker дозволяє зберігати історію образів контейнерів у репозиторіях, таких як Docker Hub чи приватні реєстри, що полегшує відкат до попередніх версій у разі виникнення проблем.

Наприклад, якщо оновлення бекенду призводить до нестабільності, то можна швидко повернутися до попереднього стабільного образу, мінімізуючи час простою.

Крім того, контейнеризація підтримує потенційну еволюцію до мікросервісної архітектури, у міру зростання платформи окремі компоненти можуть бути виділені в окремі сервіси без радикальної переробки архітектури, що дозволяє плавний перехід від моноліта до гібридної моделі. Для волонтерської платформи це означає можливість поступового розширення, наприклад, додавання модулів для аналітики чи інтеграції з новими каналами зв’язку, зберігаючи при цьому стабільність і продуктивність.

Таким чином, використання Docker для контейнеризації створює надійну, гнучку та масштабовану основу для розгортання платформи, яка може адаптуватися до змінних умов експлуатації, забезпечувати безперервну роботу в умовах значних навантажень і підтримувати еволюцію системи відповідно до потреб користувачів.

2.2 Засоби для реалізації клієнтської частини

Клієнтська частина веб-додатку також відіграє важливу роль у забезпеченні зручного, інтуїтивно зрозумілого та доступного інтерфейсу для волонтерів, незалежно від їхнього віку чи досвіду роботи з цифровими платформами.

Проведено ретельний аналіз сучасних технологій для створення адаптивного, швидкого та функціонального інтерфейсу, який дозволяє користувачам ефективно обмінюватися інформацією, координувати дії, переглядати актуальні завдання, отримувати сповіщення в реальному часі та взаємодіяти з іншими учасниками платформи без зайвих затримок чи складнощів.

У цьому підрозділі детально розглядаються технології для реалізації взаємодії в реальному часі та створення інтерфейсу користувача, включаючи швидкість реагування, простоту використання та адаптивність до різних пристроїв, таких як настільні комп’ютери чи смартфони.

2.2.1 Технології взаємодії в реальному часі

Для реалізації взаємодії в реальному часі між клієнтською частиною та сервером обрано технологію SignalR [15], яка забезпечує двосторонній зв’язок і дозволяє оновлювати дані на інтерфейсі без затримок, створюючи ефект миттєвої синхронізації між діями користувачів і відображенням змін. SignalR застосовується виключно на головній сторінці платформи, для публікації нових оголошень на дошку оголошень у реальному часі. Вибір SignalR обґрунтовано його численними перевагами, які ідеально відповідають потребам волонтерської платформи.

По-перше, SignalR підтримує WebSocket як основний протокол для швидкої передачі даних, що забезпечує низьку затримку та високу пропускну здатність, необхідну для такого сценарію, як миттєве відображення нового оголошення на дошці для всіх підключених користувачів одразу після його створення. WebSocket дозволяє встановити постійне з’єднання між клієнтом і сервером, уникаючи

накладних витрат, пов’язаних із традиційним HTTP-запитом/відповіддю, що особливо важливо для головної сторінки, де оперативність відображення нових оголошень може вплинути на швидкість координації волонтерів.

По-друге, SignalR також має вбудовану сумісність із альтернативними транспортними протоколами, такими як Server-Sent Events (SSE) та Long Polling, що забезпечує гнучкість у різних умовах мережі.

Наприклад, якщо користувач на головній сторінці працює в умовах нестабільного з’єднання чи через застарілий браузер, який не підтримує WebSocket, SignalR автоматично перемикається на альтернативний протокол, зберігаючи функціональність платформи. Ця адаптивність є критично важливою для волонтерів, які можуть працювати в польових умовах із обмеженим доступом до мережі, наприклад, у сільській місцевості чи під час стихійних лих, коли стабільність інтернет-з’єднання не гарантується.

Також SignalR підтримує автоматичне відновлення з’єднання після тимчасових збоїв, що дозволяє користувачам на головній сторінці продовжувати переглядати дошку оголошень без необхідності вручну оновлювати сторінку чи повторно входити в систему, забезпечуючи безперервний доступ до нових оголошень.

На головній сторінці SignalR забезпечує миттєве оновлення дошки оголошень. Коли користувач створює нове оголошення, SignalR передає його на сервер через WebSocket, і це оголошення миттєво відображається на дошці для всіх підключених користувачів. Це дозволяє іншим волонтерам одразу побачити нове завдання та відреагувати на нього, що значно підвищує ефективність роботи головної сторінки як основного центру взаємодії для користувачів.

Наприклад, якщо автор публікує термінове оголошення про потребу в медичній допомозі, SignalR забезпечує його миттєве відображення для всіх активних користувачів, дозволяючи швидко залучити відповідних волонтерів.

Додатково до SignalR застосовано gRPC із протоколом HTTP/2 для потокової передачі даних між клієнтом і сервером, що забезпечує високу швидкість і ефективність обміну інформацією. gRPC використовує бінарний формат Protocol

Buffers (Protobuf) для серіалізації даних, що значно зменшує розмір повідомлень у порівнянні з традиційним JSON чи XML, а також підтримує стиснення заголовків і мультиплексування через HTTP/2, дозволяючи одночасно обробляти кілька потоків даних через одне з'єднання.

Для забезпечення стабільності та масштабованості gRPC інтегрується з іншими компонентами системи, такими як брокер повідомлень Kafka, який використовується для асинхронної відправки сповіщень на поштовий сервіс.

Наприклад, коли волонтер додає нове оголошення через головну сторінку, Kafka може паралельно відправити email із деталями завдання відповідним учасникам, що забезпечує багатоканальну комунікацію.

Використання SignalR для головної сторінки та gRPC для інших частин платформи створює гнучку основу для реалізації взаємодії в реальному часі, яка може адаптуватися до різних сценаріїв використання та умов експлуатації.

2.2.2 Інтерфейс користувача

Для створення інтерфейсу користувача обрано фреймворк ASP.NET MVC [13], що забезпечує серверний рендеринг сторінок і дозволяє швидко завантажувати контент, наприклад списки завдань, профілі волонтерів чи історію взаємодій.

Вибір ASP.NET MVC обґрунтовано його глибокою інтеграцією з екосистемою C#, що спрощує розробку та підтримку коду завдяки єдиній мові програмування для серверної та клієнтської логіки. Фреймворк використовує модель MVC (Model-View-Controller), яка чітко розподіляє відповідальності між даними, відображенням і логікою управління, що сприяє структурованому і зрозумілому коду, легкому для масштабування та модифікації.

Серверний рендеринг забезпечує швидке завантаження сторінок навіть за повільного інтернет-з'єднання. Наприклад, сторінка зі списком завдань генерується на сервері і передається браузеру у вигляді готового HTML, що

зменшує навантаження на клієнтський пристрій і прискорює відображення. Це також покращує індексацію пошуковими системами, що підвищує видимість платформи.

Фреймворк ASP.NET MVC підтримує адаптивний дизайн через Razor-синтаксис та інтеграцію з CSS-фреймворками, такими як Bootstrap. Це дозволяє інтерфейсу автоматично адаптуватися до різних пристроїв: наприклад, на настільному комп'ютері список завдань відображається як таблиця, а на смартфоні ─ як вертикальний список карток, що зручно для користувачів у дорозі.

На головній сторінці SignalR інтегровано для оновлення дошки оголошень у реальному часі: нові оголошення відображаються миттєво без перезавантаження сторінки, що економить час і знижує навантаження на сервер. Для інших сторінок ASP.NET MVC використовує gRPC для швидкого завантаження даних.

Фреймворк ASP.NET MVC забезпечує зручний, адаптивний і продуктивний інтерфейс, який відповідає потребам користувачів, дозволяючи швидко отримувати доступ до інформації та ефективно взаємодіяти в рамках платформи.

2.3 Засоби для роботи з базами даних

База даних є основою для зберігання та обробки інформації, необхідної для функціонування веб-додатку, включаючи дані про волонтерів, завдання, оголошення тощо.

Проведено аналіз технологій для роботи з базами даних, з акцентом на їхню продуктивність, надійність, сумісність із C# та здатність забезпечувати стабільну роботу платформи в умовах значного навантаження.

2.3.1 Система управління базами даних

Для зберігання даних обрано Microsoft SQL Server (MSSQL) [2] як основну систему управління базами даних.

Вибір MSSQL обґрунтовано його високою продуктивністю при обробці великих обсягів даних, що є ключовим для платформи, яка може одночасно обслуговувати сотні чи тисячі волонтерів.

База MSSQL [16] забезпечує надійну підтримку транзакцій, що гарантує цілісність даних під час складних операцій, таких як одночасне оновлення статусів завдань чи створення оголошень. Його оптимізовані механізми індексації та кешування дозволяють швидко отримувати дані, наприклад, списки активних завдань чи профілі волонтерів, що критично важливо під час пікових навантажень, коли платформа обробляє численні запити в реальному часі.

База даних MSSQL також пропонує розширені функції безпеки, такі як шифрування даних у спокої та під час передачі, а також гнучке управління доступом через ролі та політики. Це забезпечує надійний захист конфіденційних даних волонтерів, таких як імена, контактні дані, що є пріоритетом для платформи, яка працює з чутливою інформацією.

Також MSSQL підтримує складні зв’язки між сутностями, що дозволяє ефективно моделювати структури даних, такі як зв’язки між волонтерами та оголошеннями. Це забезпечує гнучкість у розробці бізнес-логіки та адаптацію платформи до нових вимог, таких як додавання аналітичних звітів чи інтеграції з геолокаційними сервісами.

2.3.2 Інструменти ORM

Для роботи з базами даних застосовано Entity Framework Core (EF Core) [2] як основний інструмент ORM (Object-Relational Mapping). Вибір EF Core обґрунтовано його глибокою інтеграцією з C# і ASP.NET Core, що дозволяє

розробникам працювати з даними на рівні об’єктів, а не прямих SQL-запитів, що значно спрощує розробку, знижує ризик помилок і підвищує безпеку. EF Core автоматично мапує об’єкти бізнес-логіки на таблиці MSSQL, дозволяючи зосередитися на логіці додатку, а не на низькорівневих операціях з базою даних.

Структура EF Core [9] підтримує складні запити, включаючи фільтрацію, сортування, групування та об’єднання таблиць, що дозволяє ефективно отримувати дані, наприклад, відфільтровані списки завдань за тегом чи деталізовані профілі волонтерів.

Інструмент також забезпечує механізми міграцій, які автоматизують оновлення схеми бази даних при зміні моделей, наприклад, при додаванні нового поля до таблиці оголошень чи створенні індексів для підвищення продуктивності. Підтримка транзакцій у EF Core гарантує цілісність даних під час складних операцій, таких як одночасне створення оголошення та оновлення статусу волонтера, що є важливим для підтримки консистентності платформи.

Також EF Core забезпечує безпечну взаємодію з MSSQL, автоматично параметризуючи запити, що унеможливлює SQL-ін’єкції, які можуть загрожувати конфіденційності даних. Використання оптимізованих драйверів для MSSQL дозволяє EF Core максимально використовувати можливості бази, такі як індексація та кешування запитів, що прискорює доступ до даних. Наприклад, запити до таблиці завдань із великою кількістю записів виконуються швидко завдяки індексам, створеним через EF Core.

Додатково EF Core підтримує асинхронні операції, що підвищує продуктивність платформи за рахунок паралельної обробки звернень до бази даних. Під час масового перегляду списків завдань або створення оголошень асинхронні запити дозволяють серверу обробляти кілька запитів одночасно, зменшуючи час очікування для користувачів і забезпечуючи плавну роботу навіть у пікові періоди.

Використання Entity Framework Core створює надійну основу для роботи з даними, забезпечуючи швидкий, безпечний і ефективний доступ до інформації в MSSQL. Цей інструмент дозволяє платформі відповідати високим вимогам

волонтерської діяльності, підтримуючи як базові операції, так і складні сценарії обробки даних, зберігаючи при цьому стабільність і масштабованість.

2.4 Засоби для реалізації міжсервісної взаємодії

Міжсервісна взаємодія є ключовим елементом архітектури веб-додатку, оскільки вона забезпечує ефективну координацію між різними компонентами системи, такими як бекенд, база даних, поштовий сервіс та інтерфейс користувача.

Для реалізації міжсервісної взаємодії використано як синхронні, так і асинхронні підходи, щоб оптимально відповідати різноманітним сценаріям використання платформи.

Синхронна взаємодія застосовується там, де потрібна миттєва відповідь, наприклад, під час завантаження даних для відображення, а асинхронна застосовується для обробки операцій, які можуть виконуватися у фоновому режимі, таких як відправка сповіщень.

У цьому підрозділі детально розглядаються технології, обрані для реалізації синхронної та асинхронної взаємодії, їх переваги та відповідність вимогам волонтерської платформи.

2.4.1 Синхронна взаємодія

Для синхронного обміну даними між модулями платформи обрано технологію gRPC [7] із підтримкою HTTP/2, що забезпечує швидкий і ефективний зв’язок. gRPC використовує бінарний формат Protocol Buffers (Protobuf) для передачі даних, що значно зменшує їхній обсяг у порівнянні з текстовими форматами, такими як JSON. Це дозволяє оперативно передавати складні структури даних, наприклад, деталізовані списки оголошень із тегами, що необхідно для швидкого відображення інформації в інтерфейсі користувача.

Протокол HTTP/2, який лежить в основі gRPC, забезпечує можливість одночасної обробки кількох потоків через одне з’єднання завдяки мультиплексуванню. Це зменшує затримки під час інтенсивного обміну даними, наприклад, коли координатор одночасно запитує інформацію про оголошення чи волонтерів.

Технолоія gRPC пропонує гнучкість завдяки підтримці різних типів викликів. Наприклад, унарні виклики використовуються для швидкого отримання статичних даних, таких як профіль волонтера, тоді як серверний стрімінг застосовується для передачі оновлень у реальному часі, наприклад, під час відображення змін у координатах на карті завдань. Інтеграція gRPC із C# дозволяє легко генерувати клієнтський і серверний код із файлів Protobuf, що забезпечує чітку типізацію та зменшує ризик помилок під час передачі даних між модулями.

2.4.2 Асинхронна взаємодія

Для асинхронного обміну даними між компонентами платформи обрано брокер повідомлень Apache Kafka [1], який забезпечує ефективну обробку подій у фоновому режимі. Kafka дозволяє системі працювати з великою кількістю повідомлень, що виникають під час активності користувачів, наприклад, коли одночасно створюються десятки нових оголошень, оновлюються їхні статуси або надсилаються повідомлення про нові відповіді на них.

Система Kafka використовує модель «публікація-підписка», де події, такі як створення оголошення, публікуються в спеціальні теми, а підписані сервіси, наприклад, модуль відправки email-повідомлень, обробляють їх незалежно. Наприклад, після додавання нового оголошення бекенд генерує подію, яка містить ідентифікатор оголошення, його автора та список одержувачів. Ця подія потрапляє в тему Kafka, звідки модуль обробки повідомлень витягує її та надсилає email усім зацікавленим волонтерам, повідомляючи про нове завдання.

Для інтеграції Kafka з платформою використано бібліотеку Confluent.Kafka, яка забезпечує зручний доступ до функціоналу брокера з C#. Це дозволяє бекенду швидко публікувати події, а споживачам ефективно їх обробляти, наприклад, для генерації звітів про активність користувачів чи надсилання push-повідомлень через зовнішні сервіси. Kafka також підтримує механізми підтвердження обробки, що гарантує, що жодна подія не буде втрачена, забезпечуючи надійність комунікації.

Використання Kafka дозволяє платформі залишатися чутливою до дій користувачів, оскільки фонові операції не блокують основний потік обробки. Це забезпечує безперебійну роботу платформи, дозволяючи волонтерам оперативно отримувати важливі повідомлення та координувати свої дії.

2.5 Метод аутентифікації

Для забезпечення безпечного доступу до веб-додатку обрано токен-базовану аутентифікацію за допомогою JSON Web Token (JWT) із використанням refresh tokens. Цей метод забезпечує надійний механізм ідентифікації користувачів, що є критично важливим для платформи. Після успішного введення логіна та пароля користувач отримує доступний токен, який містить закодовані дані, включаючи ідентифікатор користувача та його роль (волонтер чи адміністратор). Токен передається в заголовках HTTP-запитів, дозволяючи серверу швидко перевіряти авторизацію без повторного звернення до бази даних для кожної дії.

Використання refresh tokens дозволяє автоматично оновлювати основний токен після його закінчення терміну дії, що підвищує зручність для волонтерів, які можуть працювати з платформою протягом тривалого часу, наприклад, під час багатоденних кампаній. Це усуває необхідність частого повторного входу, зберігаючи при цьому безпеку завдяки короткому терміну дії основних токенів. Для захисту від компрометації JWT підтримує цифровий підпис із секретним ключем, що унеможливлює підробку, а також шифрування, яке захищає дані під час передачі.

Додаткові заходи безпеки включають зберігання токенів у захищеному вигляді на клієнтському боці (наприклад, у HTTP-only cookies), що зменшує ризик витоку через XSS-атаки. У разі підозри на компрометацію користувач може вручну завершити сесію, що автоматично анулює всі активні токени. Цей підхід забезпечує баланс між зручністю та захистом.

2.6 Додаткові засоби розробки

Розробка веб-додатку потребує використання сучасних інструментів, які оптимізують процес створення, тестування та супроводу платформи. У цьому розділі розглядаються ключові засоби, що сприяють підвищенню продуктивності розробників, забезпечують контроль версій і створюють комфортне середовище для роботи над проєктом. Вибрані інструменти інтегруються з екосистемою C# і підтримують монолітну архітектуру, забезпечуючи стабільність і ефективність на всіх етапах розробки.

2.6.1 Середовище розробки

Для розробки веб-додатку обрано Visual Studio 2022 як основне інтегроване середовище розробки (IDE), яке забезпечує повну підтримку C# і всіх пов’язаних технологій, таких як ASP.NET Core, Entity Framework Core і gRPC. Visual Studio 2022 пропонує зручний інтерфейс із вбудованими інструментами для налагодження, автодоповнення коду та інтеграції з Docker для тестування контейнеризованих компонентів. Це дозволяє розробникам швидко виявляти помилки, оптимізувати запитів до MSSQL і перевіряти асинхронні операції з Kafka чи SignalR.

2.6.2 Система контролю версій

Для управління версіями коду обрано Git із хостингом на GitHub, що забезпечує централізоване зберігання проєкту. Git дозволяє відстежувати зміни в коді, створювати гілки для експериментальних функцій (наприклад, інтеграції геолокації) і об’єднувати їх після тестування. Використання GitHub полегшує співпрацю, надаючи інструменти для рецензування коду і відстеження завдань, що оптимізує процес розробки.

Конфігурація репозиторію включає базові гілки, що забезпечує впорядкований робочий процес. Інтеграція з діями GitHub Actions дозволяє автоматизувати тести та розгортання контейнерів Docker, скорочуючи час на ручну перевірку. Це гарантує, що кожна зміна, наприклад, оновлення логіки SignalR, проходить автоматичну валідацію перед інтеграцією, підвищуючи надійність платформи.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У цьому розділі представлено детальний опис програмної реалізації веб-додатку для взаємодії між волонтерами, розробленого на мові програмування C#. Реалізація охоплює архітектуру системи, структуру бази даних, розробку клієнтської та серверної частин, а також міжсервісну взаємодію.

3.1 Архітектура веб-застосунку

Веб-додаток побудовано на основі монолітної багатошарової архітектури з використанням принципів «чистої архітектури», що забезпечує чіткий розподіл відповідальностей між компонентами. Система складається з клієнтської частини (UI), реалізованої на ASP.NET MVC, і серверної частини (бекенд), створеної з використанням ASP.NET Core. Взаємодія між цими компонентами здійснюється через gRPC із протоколом HTTP/2, що забезпечує швидкий і ефективний обмін даними. Уся інфраструктура розгортається за допомогою Docker, що гарантує ізоляцію компонентів і спрощує масштабування.

На серверній стороні застосовано «чисту архітектуру», яка поділяє систему на шари: доменний, додатку, інфраструктурний і представлення.

Доменний шар містить основні сутності, такі як користувачі, оголошення та теги, а також бізнес-правила, наприклад, логіку створення оголошень.

Шар додатку координує обробку запитів через CQRS і MediatR, де gRPC-запити, описані у proto-файлах, мапляться через AutoMapper на команди або запити і передаються до обробників.

Інфраструктурний шар забезпечує доступ до MSSQL через EF Core, а також інтеграцію з Kafka для асинхронної відправки повідомлень на поштовий сервіс.

Шар представлення обробляє вхідні gRPC-запити від UI і повертає відповіді, використовуючи SignalR для оновлення головної сторінки в реальному часі.

Клієнтська частина, побудована на ASP.NET MVC, відповідає за відображення інтерфейсу та взаємодію з користувачами.

Для головної сторінки використано SignalR, який через WebSocket оновлює дошку оголошень у реальному часі, а для інших сторінок використано gRPC для швидкого завантаження даних.

Така архітектура забезпечує високу продуктивність, модульність і можливість адаптації до нових вимог волонтерських організацій.

3.2 Структура бази даних

База даних веб-додатку (рисунок 3.1) реалізована за допомогою Microsoft SQL Server (MSSQL), що забезпечує надійне зберігання даних про користувачів, оголошення, відповіді, теги та статистику. Доступ до бази даних здійснюється через Entity Framework Core (EF Core), який дозволяє працювати з даними на рівні об’єктів C# і забезпечує безпечну взаємодію шляхом параметризації запитів.

Центральною таблицею є Announcements, яка зберігає інформацію про публікації волонтерів, включаючи заголовок, зміст, автора, час створення, адресу та статус повідомлення. Відповіді користувачів на оголошення зберігаються у таблиці Responses, де фіксується текст повідомлення, автор відповіді та час надсилання.

Таблиця AspNetUsers акумулює контактні дані користувачів, параметри авторизації, автентифікації та індивідуальні налаштування. Керування доступом до функціональності реалізовано через таблиці AspNetRoles, AspNetUserRoles, AspNetUserClaims та AspNetRoleClaims, що забезпечують підтримку ролей і пов’язаних із ними прав доступу відповідно до стандарту ASP.NET Identity.

Для тематичного структурування контенту використовується таблиця Tags, яка містить перелік доступних тегів, та таблиця UserTags, що пов’язує користувачів з відповідними тематиками через Email та ідентифікатор тегу. Це дозволяє

реалізувати систему фільтрації та оповіщення за інтересами й покращити релевантність відображення інформації.

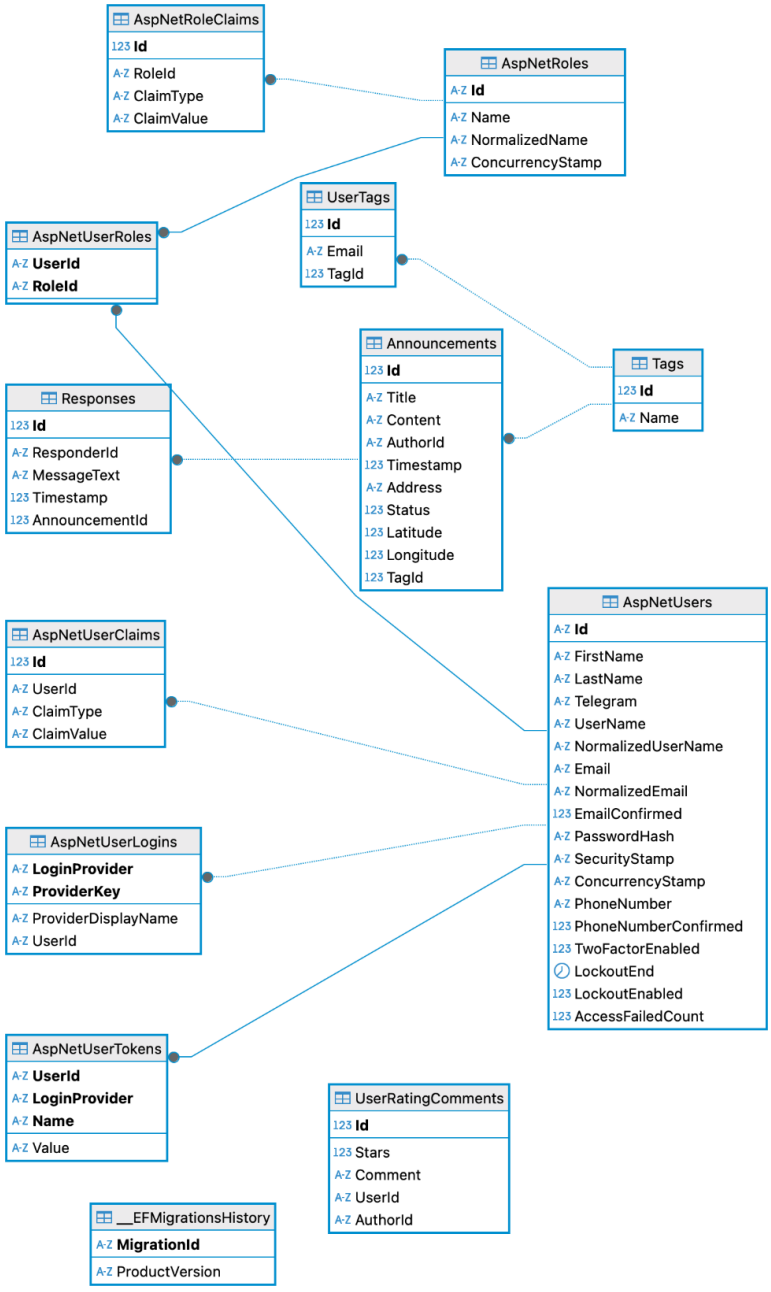

Рисунок 3.1 — Структура бази даних

Загалом, структура бази даних побудована таким чином, щоб забезпечити гнучке управління контентом, користувачами та доступом у межах волонтерської платформи.

3.3 Розробка користувацької частини

Користувацька частина веб-додатку розроблена на основі ASP.NET MVC, що забезпечує створення інтуїтивно зрозумілого інтерфейсу з адаптивним дизайном. UI взаємодіє з бекендом через gRPC, а для оновлення головної сторінки в реальному часі використовується SignalR із WebSocket. Розробка охоплює функціонал реєстрації, авторизації, роботи з оголошеннями та відображення статистики, що відповідає потребам волонтерів.

3.3.1 Реалізація реєстрації та авторизації

Реєстрація та авторизація користувачів реалізовані з використанням токен-базованої аутентифікації на основі JSON Web Token (JWT) із bearer-токенами. При реєстрації користувач вводить електронну пошту та пароль, які зберігаються в таблиці Users у MSSQL. Паролі хешуються за допомогою алгоритму BCrypt для забезпечення безпеки. Після успішної реєстрації користувач може авторизуватися, ввівши свої облікові дані.

При авторизації сервер перевіряє коректність даних, генерує JWT-токен із інформацією про користувача (ідентифікатор, роль) і підписує його секретним ключем. Токен передається клієнту і зберігається в HTTP-only cookies. Bearer-токен додається до контексту кожного gRPC-запиту, дозволяючи серверу ідентифікувати користувача. Наприклад, при запиті на створення оголошення бекенд перевіряє токен і визначає, чи має користувач права на цю дію.

3.3.2 Реалізація публікації та редагування оголошень

Функціонал публікації та редагування оголошень реалізований через ASP.NET MVC із використанням gRPC для взаємодії з бекендом.

Користувач на сторінці створення оголошення вводить заголовок, вміст і теги, після чого дані через gRPC-запит відправляються на сервер. Запит, описаний у proto-файлі, мапиться через AutoMapper на команду, яка обробляється через MediatR. Обробник команди в шарі додатку валідує дані (наприклад, перевіряє унікальність заголовка) і через репозиторій зберігає оголошення в таблиці Announcements у MSSQL.

Після створення оголошення SignalR через WebSocket оновлює дошку оголошень на головній сторінці для всіх підключених користувачів, відображаючи нове оголошення в реальному часі.

Для редагування оголошення користувач відкриває сторінку з відповідним оголошення, де через gRPC завантажуються поточні дані оголошення, після чого оновлені дані відправляються аналогічним шляхом. Зміни також відображаються на головній сторінці через SignalR, забезпечуючи миттєву синхронізацію для всіх волонтерів.

3.3.3 Відображення статистики

Функція відображення статистики дозволяє користувачам переглядати дані про їхню активність на платформі, такі як кількість створених оголошень, отриманих відповідей та взаємодій із тегами.

Сторінка статистики, побудована на ASP.NET MVC, надсилає gRPC-запит до бекенду, який через MediatR обробляє запит на вибірку даних із бази даних. Обробник у шарі додатку використовує репозиторій для асинхронного отримання даних із таблиць Announcements, Responses, AspNetUsers, UserTags та пов’язаних таблиць, застосовуючи фільтрацію за ідентифікатором користувача.

Дані повертаються клієнту у вигляді структурованого об’єкта через gRPC і відображаються у вигляді текстових блоків, створених за допомогою CSS-фреймворку Bootstrap для адаптивного дизайну.

Наприклад, користувач може побачити, що створив 10 оголошень і написав 15 відповідей за останній місяць. Оновлення статистики відбувається асинхронно після нових дій.

3.4 Розробка серверної частини

Серверна частина веб-додатку розроблена на основі ASP.NET Core з використанням «чистої архітектури» та патерну CQRS із MediatR, що забезпечує чіткий розподіл логіки та високу тестопридатність.

Бекенд обробляє запити від клієнтського інтерфейсу через gRPC із протоколом HTTP/2, де структура даних визначається у proto-файлах. Ці запити мапляються через AutoMapper на команди або запити, які передаються до обробників у шарі додатку через MediatR. У хендлерах реалізовано ін’єкцію залежностей (DI) для підключення репозиторіїв, оголошених у шарі додатку та реалізованих у шарі інфраструктури, що забезпечує доступ до бази даних MSSQL через Entity Framework Core (EF Core).

Прикладом є обробка створення оголошення, gRPC-запит із відповідним proto-описом перетворюється на команду, яку обробник валідує і зберігає в базі через репозиторій.

Для асинхронного сповіщення користувачів (наприклад, про нові оголошення) застосовується Apache Kafka, після обробки команди дані записуються в Kafka-топік, а окремий бекграунд-сервіс зчитує їх і відправляє повідомлення через SMTP-сервер.

Такий підхід гарантує обробку великих обсягів даних навіть під час піків навантаження. Розгортання серверної частини здійснюється через Docker-контейнери, які ізолюють бекенд, gRPC-сервіси, Kafka та MSSQL, забезпечуючи модульність і готовність до масштабування.

4 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

У цьому розділі описано особливості роботи користувача з розробленим веб-додатком для взаємодії між волонтерами, включаючи системні вимоги, основні сценарії використання та інструкції з експлуатації.

Також тут описано use-case діаграму і усі сторінки веб-застосунку. Веб-додаток забезпечує зручний інтерфейс для реєстрації, створення та перегляду оголошень, відповідей на них, а також перегляду статистики активності, що відповідає потребам користувачів платформи.

4.1 Системні вимоги

Для коректної роботи веб-додатку необхідно дотримуватися певних системних вимог, які стосуються як клієнтської, так і серверної частин системи. Веб-додаток розроблений із урахуванням сучасних стандартів, що забезпечує його доступність на різних пристроях і операційних системах.

Для роботи з веб-додатком користувачу потрібен пристрій із доступом до інтернету та сучасним веб-браузером, таким як Google Chrome, Mozilla Firefox, Microsoft Edge або Safari. Браузер має підтримувати WebSocket для коректної роботи SignalR, що забезпечує оновлення головної сторінки в реальному часі, а також JavaScript для обробки інтерактивних елементів інтерфейсу, створених на базі ASP.NET MVC.

Серверна інфраструктура розгортається на операційній системі Linux, MacOS або Windows Server . Для роботи бекенду, побудованого на ASP.NET Core, потрібна наявність .NET 6.0 або новіша версія. База даних MSSQL розгортається на Microsoft SQL Server 2019 або новішій версії. Для асинхронної обробки повідомлень необхідна установка Apache Kafka. Усі компоненти (бекенд, gRPC-

сервіси, Kafka, MSSQL) розгортаються через Docker, тому сервер має підтримувати Docker і Docker Compose.

Для надсилання сповіщень через SMTP-сервер необхідно налаштувати доступ до поштового сервісу із відповідними обліковими даними. Для забезпечення безпеки рекомендується використовувати HTTPS із SSL-сертифікатом для шифрування даних під час передачі між клієнтом і сервером.

Ці вимоги забезпечують стабільну роботу веб-додатку, дозволяючи користувачам ефективно взаємодіяти з платформою.

4.2 Сценарії роботи користувача з системою

Веб-додаток підтримує взаємодію між двома типами користувачів, а саме волонтерами та адміністраторами, кожен із яких має унікальний функціонал, адаптований до їхніх ролей (рисунок 4.1).

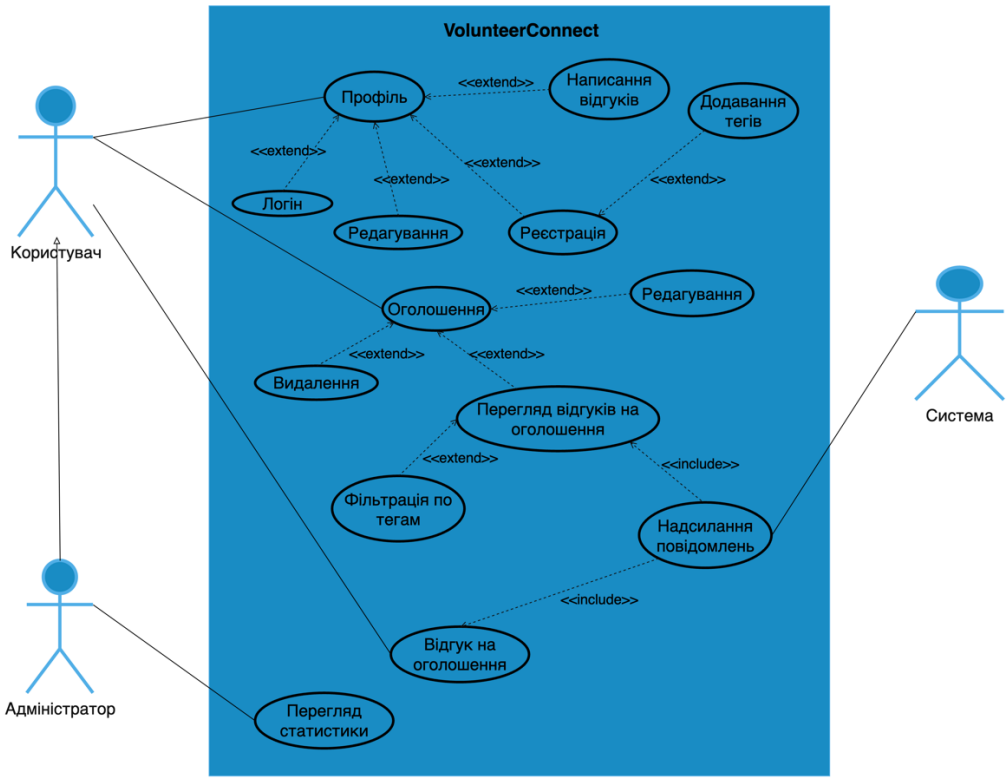

Рисунок 4.1 — Use-case діаграма

Сценарії роботи з системою охоплюють основні операції, доступні кожному типу користувача, з урахуванням їхніх прав доступу, визначених через ASP.NET Identity.

4.2.1 Реєстрація та авторизація

Процес реєстрації був розроблений для того, щоб нові користувачі мали змогу створити облікові записи. Після натискання на «Зареєструватися» (рисунок 4.2), користувач переходить на сторінку реєстрації.

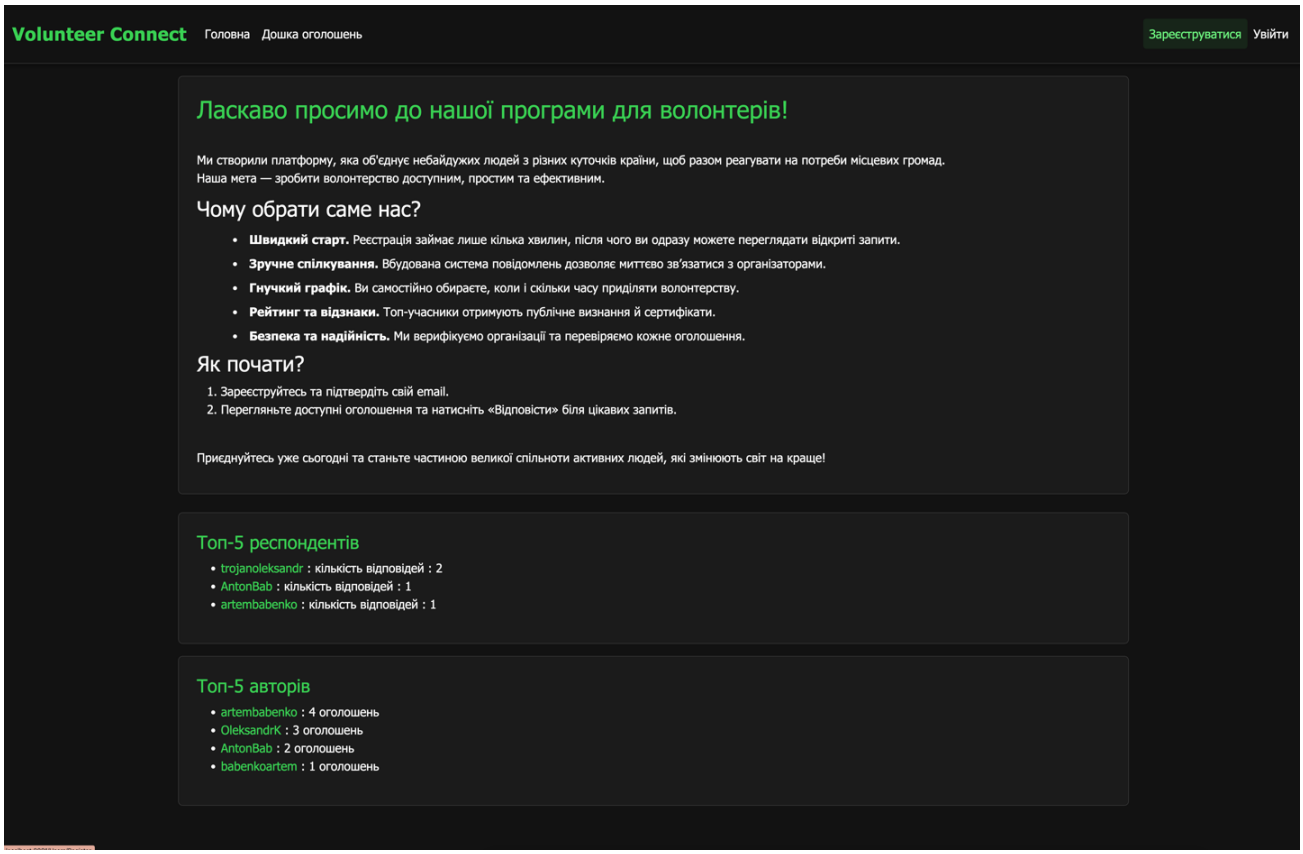

Рисунок 4.2 – Головна сторінка веб-додатку

На сторінці реєстрації (рисунок 4.3) користувач має змогу заповнити поля: ім’я, прізвище, електронна пошта, телеграм, логін, пароль.

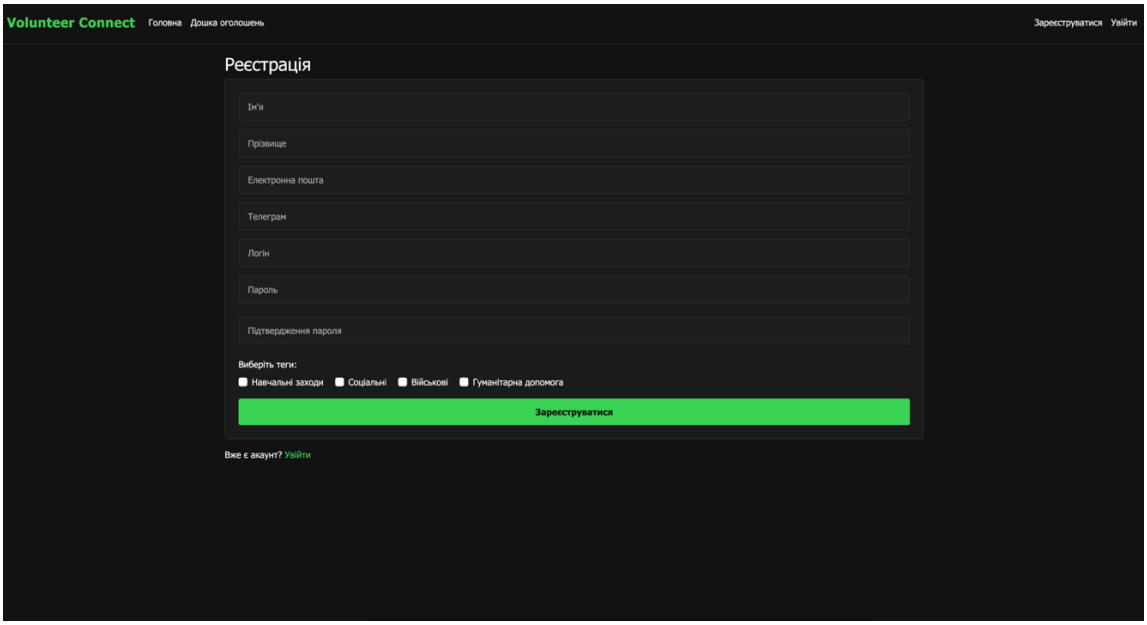

Рисунок 4.3 — Сторінка реєстрації

Додатково користувач має можливість обрати теги, які його цікавлять. Після цього він буде отримувати листи на електронну пошту, яку вказав при реєстрації, про нові оголошення з тегами, які його цікавлять.

Якщо при реєстрації користувач ввів некоректні дані, то система буде відображати помилку з підсказками (рисунок 4.4).

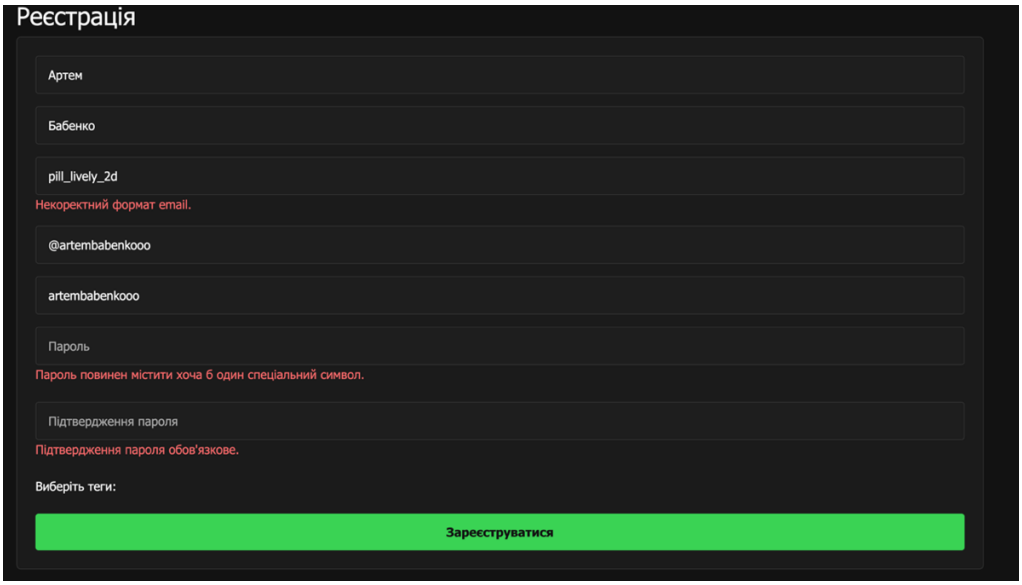

Рисунок 4.4 — Помилка про некоректно введені дані

Після введення коректних даних користувач натискає кнопку «Зареєструватися». Система валідує дані, хешує пароль, зберігає інформацію та створює унікальний токен підтвердження. Після чого користувач потрапляє на сторінку, де система повідомляє його про необхідність підтвердити електронну пошту (рисунок 4.5).

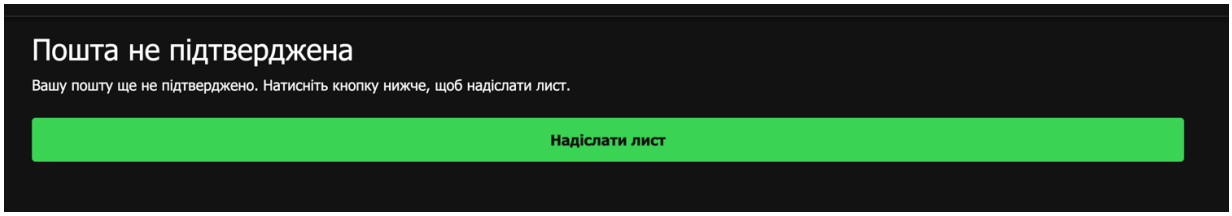

Рисунок 4.5 – Повідомлення про необхідність підтвердити пошту

Токен підтвердження застосовується для ідентифікації користувача та підтвердження правильності його електронної адреси.

Після генерації токена система створює електронний лист, який містить посилання для верифікації поштової адреси (рисунок 4.6).

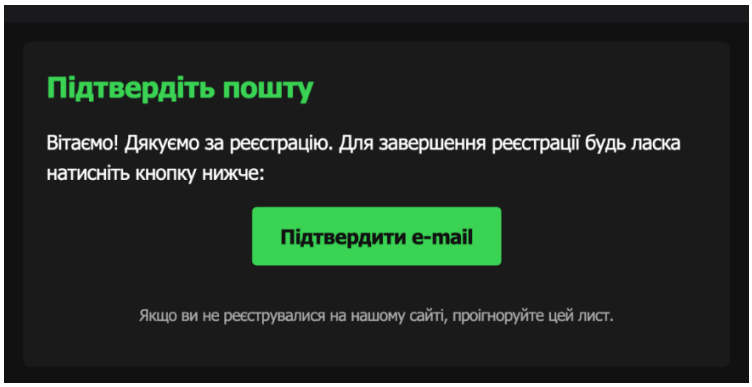

Рисунок 4.6 – Лист з посиланням для верифікації поштової адреси

При активації кнопки підтвердження, яка містить унікальний токен у складі URL-посилання, клієнтський інтерфейс генерує запит до серверної частини. Виконується перевірка токена, зокрема його наявність, відповідність конкретному користувачу та термін дії. У разі валідності токена статус облікового запису

оновлюється в базі даних, а користувач переходить на сторінку з повідомленням про успішне підтвердження пошти (рисунок 4.7).

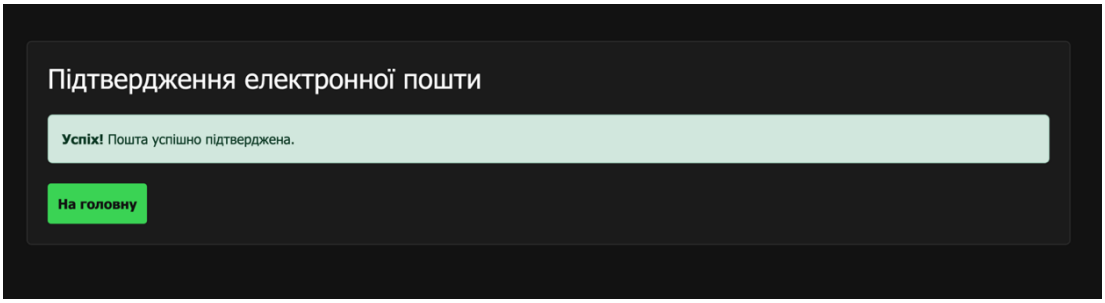

Рисунок 4.7 – Повідомлення про успішне підтвердження пошти

У разі невідповідності токена чи завершення його терміну дії система повертає повідомлення про помилку, пропонуючи користувачу повторно надіслати лист із новим посиланням.

Такий підхід забезпечує безпеку, ефективність обробки запитів і зручність взаємодії з платформою, відповідаючи вимогам до масштабованості та надійності.

Уже зареєстрований користувач має змогу авторизуватися (рисунок 4.8), для цього йому потрібно ввести електронну пошту та його пароль.

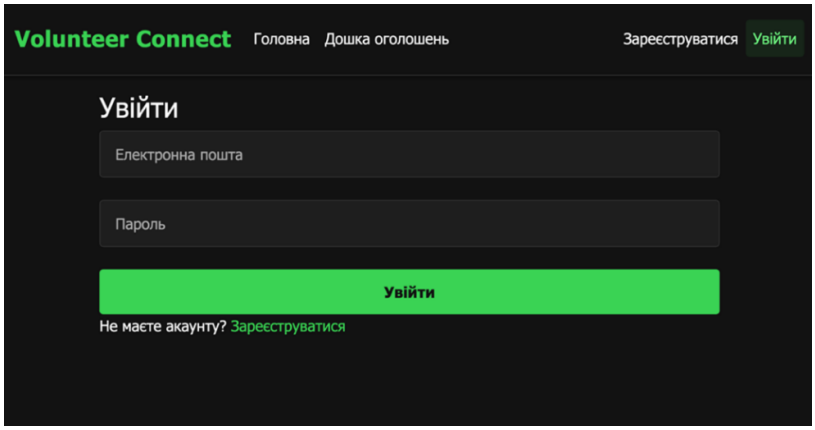

Рисунок 4.8 – Сторінка з формою для авторизації

Якщо введено некоректні дані, то система повідомляє про помилку (рисунок 4.9).

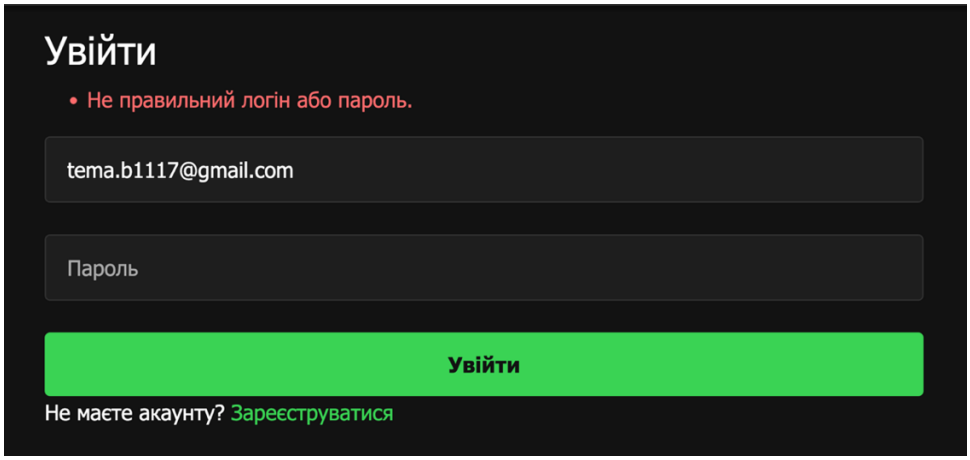

Рисунок 4.9 – Повідомлення про некоректний пароль або логін

Якщо користувач вводить коректні дані, то він потрапляє на головну сторінку (рисунок 4.2), де він бачить повідомлення з інформацією про веб-додаток та статистику користувачів.

На сторінці профіля користувача (рисунок 4.10) волонтер бачить усі свої активні та виконані оголошення, які він створив за весь час. Також тут розташовані відгуки про користувача та дані користувача, які включають ім’я, email, telegram, теги, які він обирав під час реєстрації, та середній рейтинг користувача.

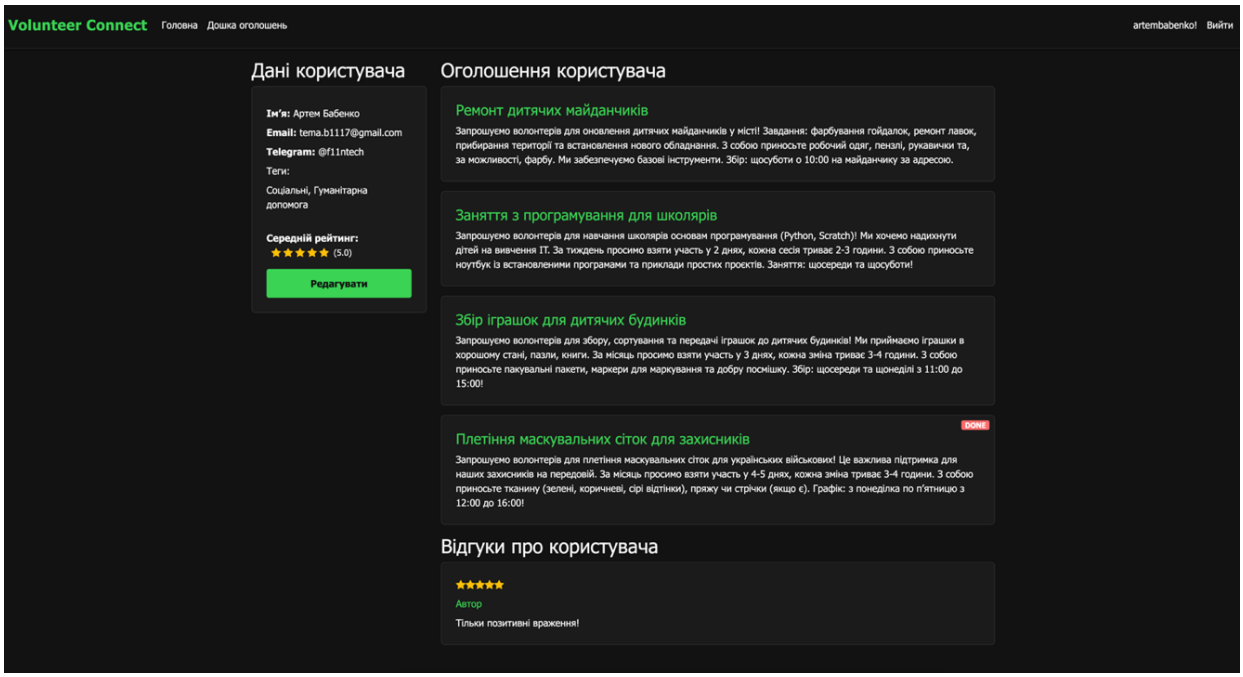

Рисунок 4.10 – Профіль користувача

За потреби користувач має змогу редагувати основну інформацію про себе (рисунок 4.11), а саме ім’я, прізвище, username в telegram.

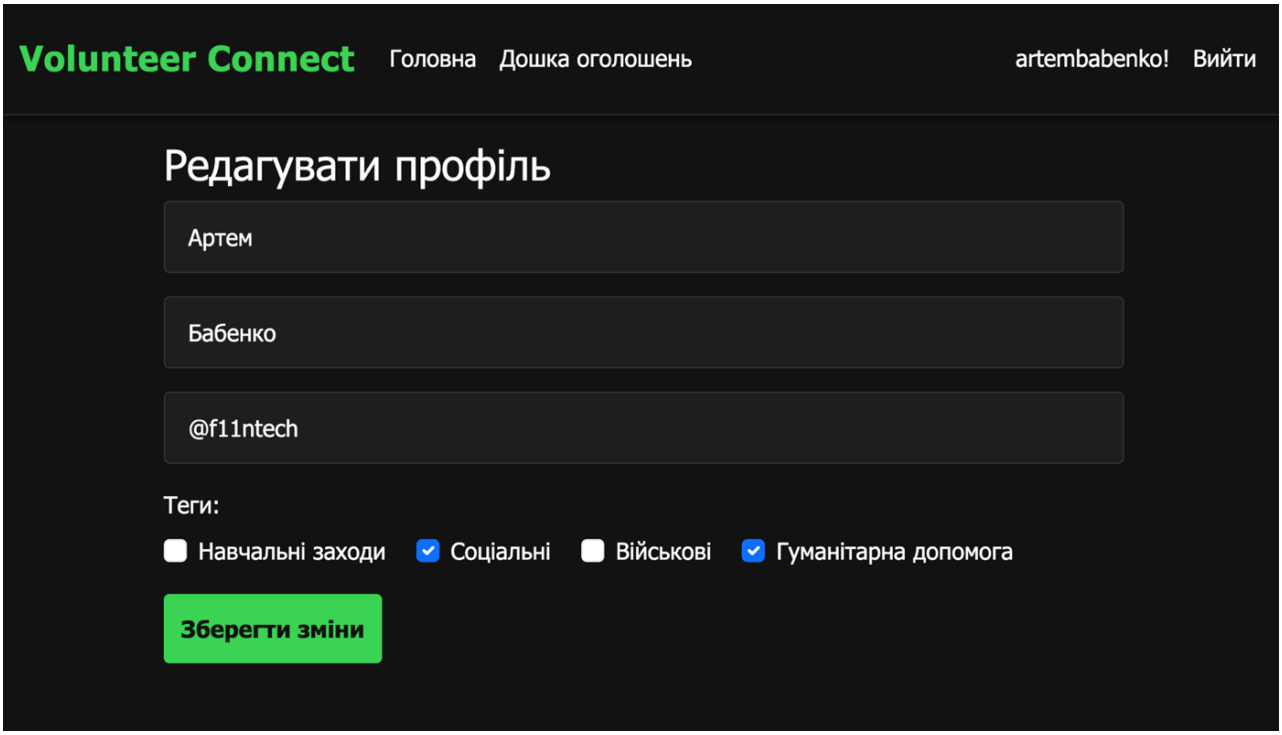

Рисунок 4.11 — Редагування профіля користувача

Також користувач може відредагувати теги, які вказував при реєстрації. Можливість редагування електронної пошти відсутня.

4.2.2 Додавання нового оголошення

Функціонал додавання нового оголошення в веб-додатку розроблений для забезпечення зручного створення публікацій волонтерами.

Авторизований волонтер з підтвердженою поштою на сторінці з оголошеннями має змогу опублікувати також і своє. Відкривши форму для створення (рисунок 4.12), користувач повинен ввести необхідні дані, а саме заголовок, опис, адресу, обрати тег з випадаючого списку та поставити локацію на карті.

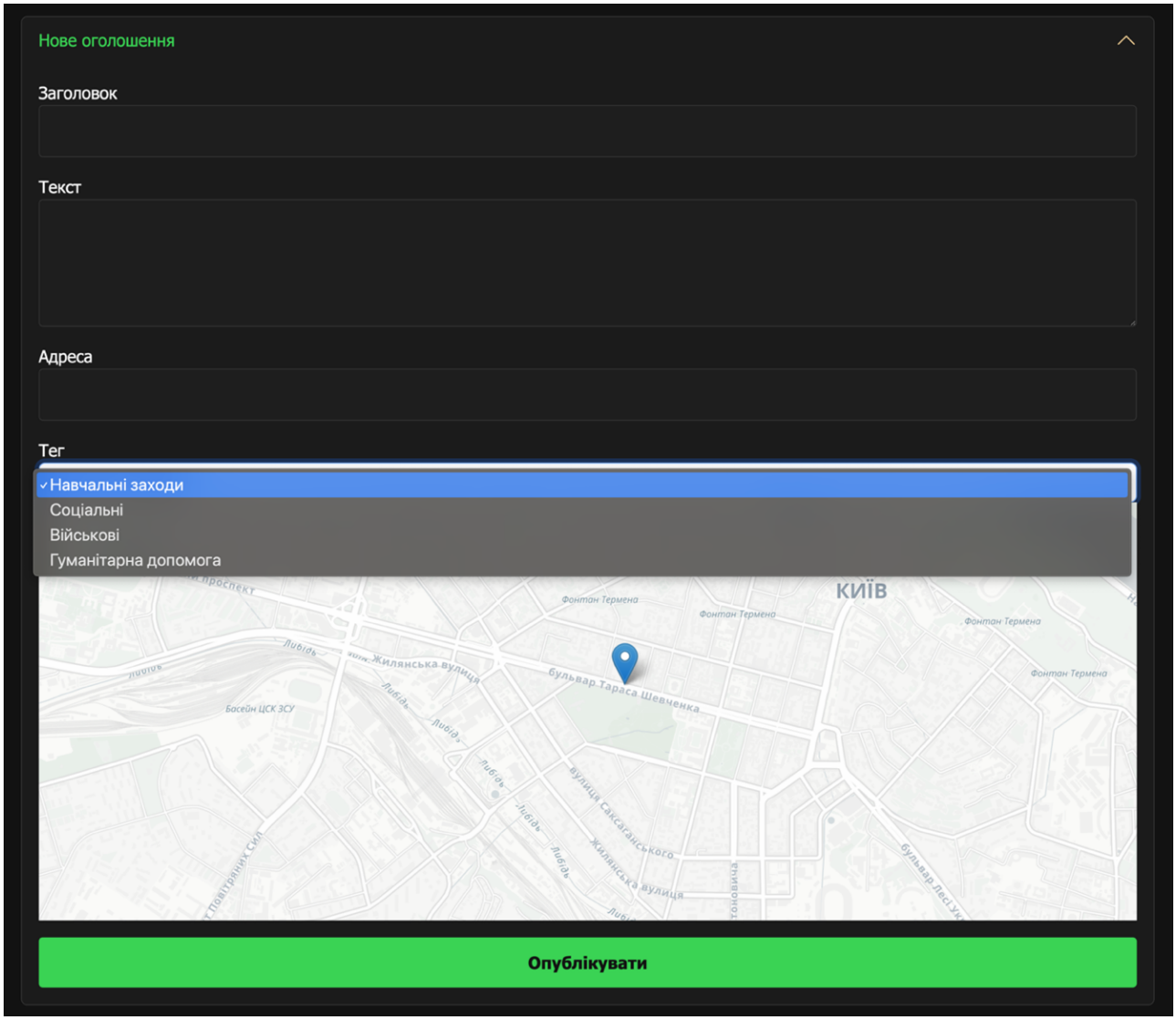

Рисунок 4.12 – Форма для створення оголошення

Після публікації, оголошення в режимі реального часу потрапляє на дошку оголошень (рисунок 4.13). В правому верхньому куту автор оголошення може бачити позначку «OWN», що означає, що він автор та позначку «NEW», яка зв’являється на всіх нових публікаціях, які потрапляють на дошку, та зникає після оновлення сторінки або після перегляду цієї публікації.

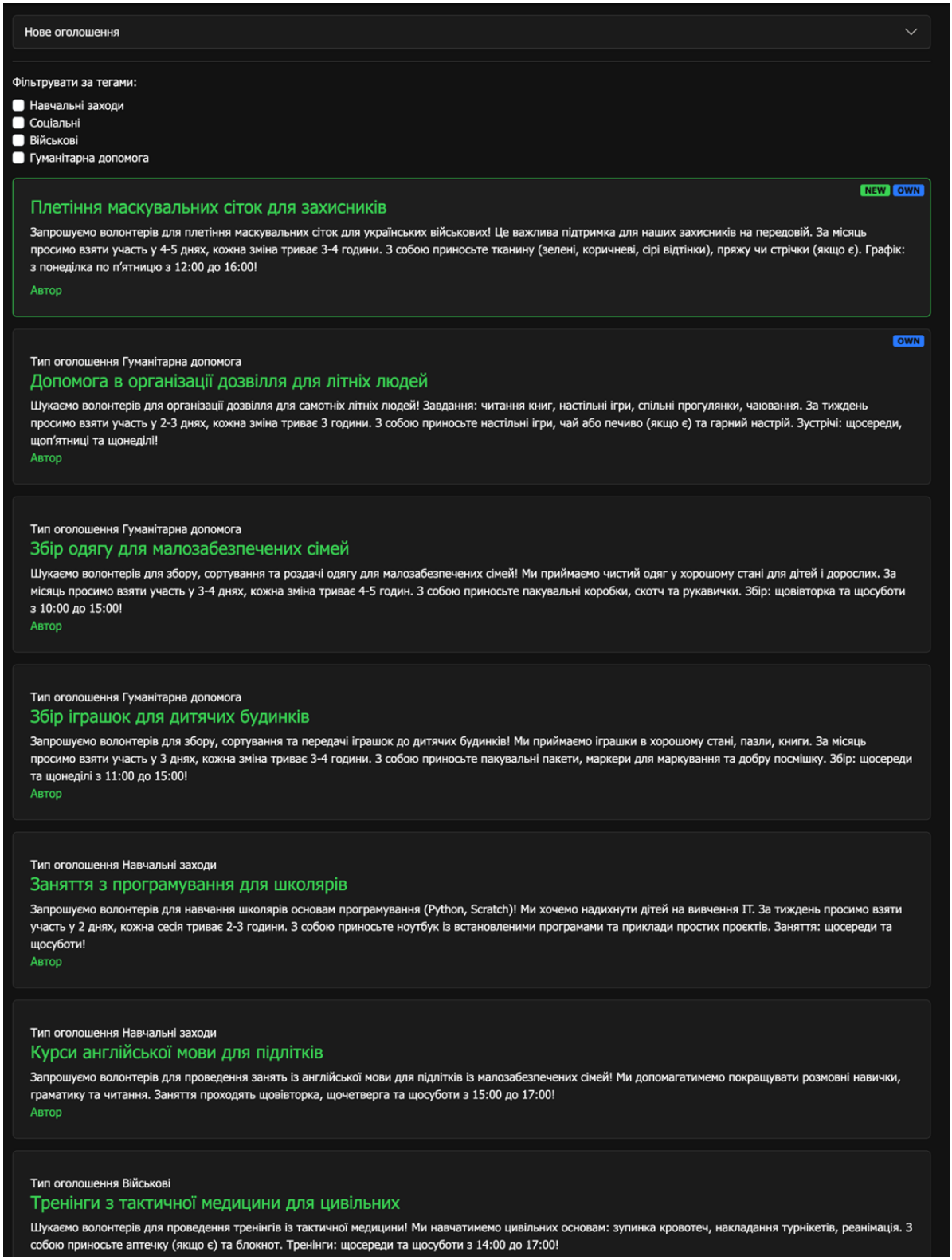

Рисунок 4.13 – Вигляд нового оголошення на дошці

Після публікацій нових оголошень на дошці, усі користувачі отримують лист на пошту (рисунок 4.14) про нове оголошення з тегом, який був обраний при реєстрації або був доданий чи відредагований в профілі користувача.

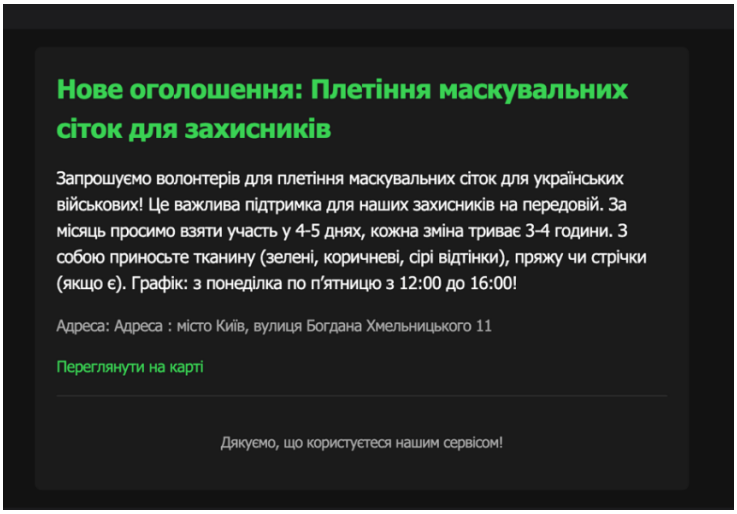

Рисунок 4.14 — Лист про появу нового оголошення

Користувач має змогу переглянути локацію проведення заходу, натиснувши на текст з активним посиланням «Переглянути на карті», після чого він перейде на сторінку Google Maps (рисунок 4.15).

Рисунок 4.15 — Відображення локації на Google Maps

Користувач має змогу побачити локацію на мапі, яка позначена червоною крапкою та її координати. Це дає можливість швидко прокласти маршрут від його поточного місцезнаходження та зрозуміти чи зможе він допомогти.

4.2.3 Редагування, перегляд та відповідь на оголошень

Авторизований користувач обирає оголошення зі списку на головній сторінці. Для зручності він може відфільтрувати оголошення за тегами, які його цікавлять, обравши, наприклад, категорію військові та соціальні (рисунок 4.16). Фільтрація може відбувається як за одним тегом так і за декількома.

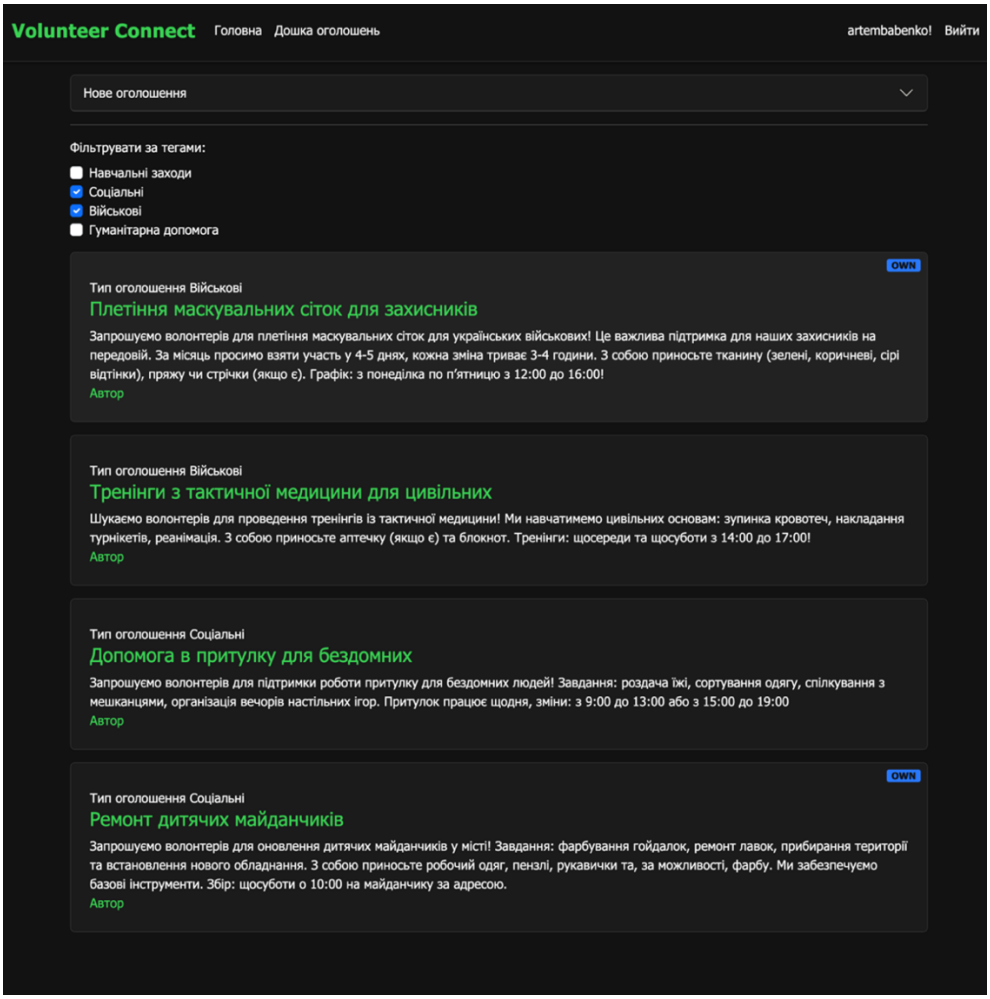

Рисунок 4.16 – Відфільтровані оголошення

Користувач обирає оголошення зі списку на головній сторінці, клікнувши на нього. Система відображає деталі оголошення (рисунок 4.17), а саме заголовок, опис із деталями, адресу з мапою, категорію за тегом та час публікації. Також користувач має змогу відповісти на оголошення та переглянути усі відповіді, які були раніше.

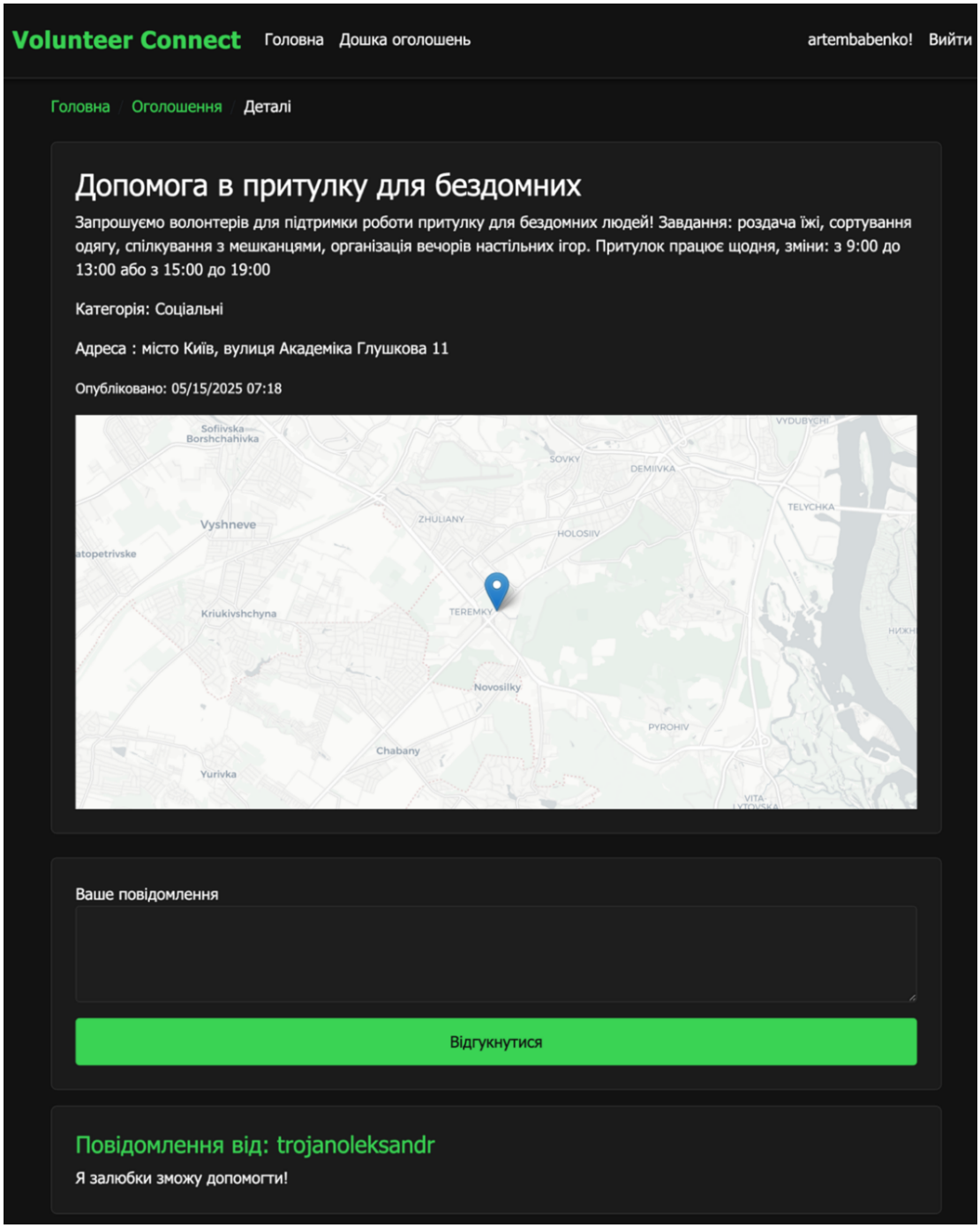

Рисунок 4.17 — Перегляд оголошення

Якщо користувач захоче відгукнутись на оголошення, то йому потрібно ввести текст повідомлення в форму, яка розташована одразу під оголошенням, після чого настиснути кнопку «Відгукнутися».

Після цього відгук з’явиться під оголошенням (рисунок 4.18) та буде доступний для перегляду всім користувачам, які також перейдуть на це оголошення.

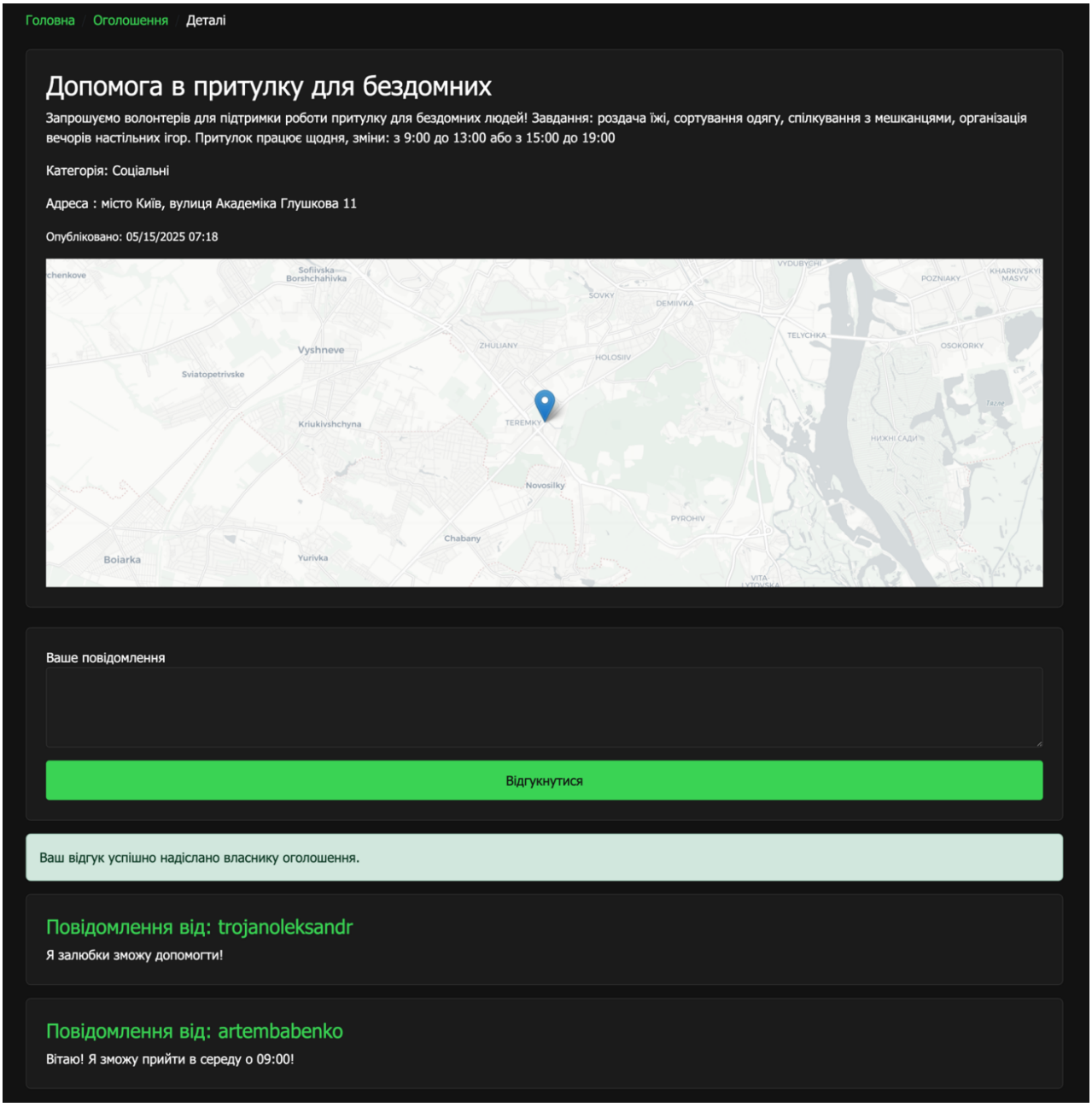

Рисунок 4.18 – Відображення відгуку на оголошення

Автор оголошення, на яке відгукнувся користувач, отримає лист на пошту з детальною інформацією про користувачав, що відгукнувся, а також сам коментар відгуку (рисунок 4.19).

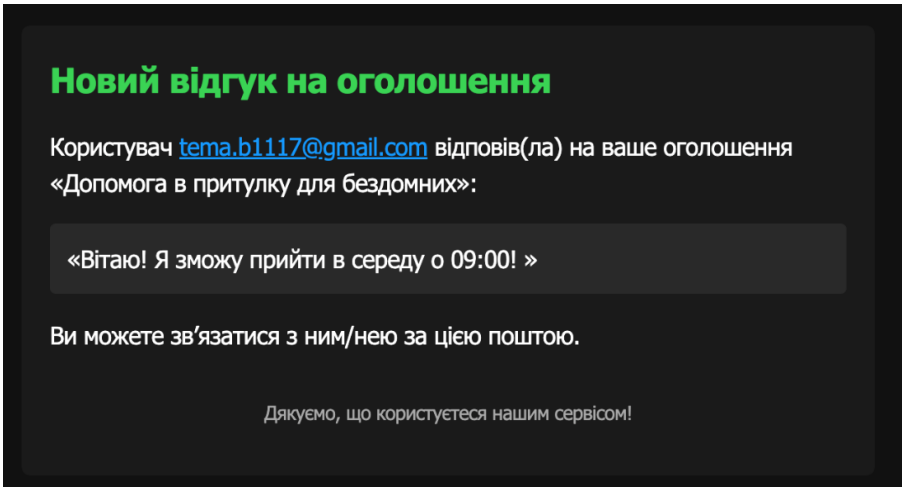

Рисунок 4.19 – Лист про отримання нового відгуку на оголошення

Автор оголошення має змогу редагувати та видаляти власні оголошення за потреби (рисунок 4.20).

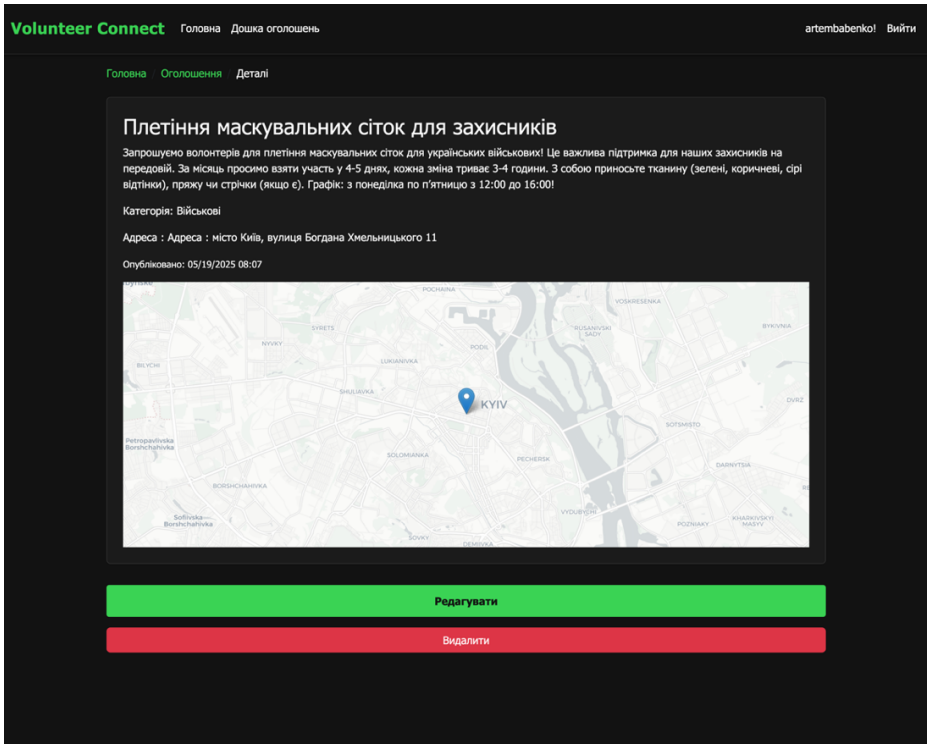

Рисунок 4.20 – Перегляд власного оголошення

Якщо користувач захоче відредагувати оголошення, то він зможе змінити заголовок, текст, адресу та статус (рисунок 4.21).

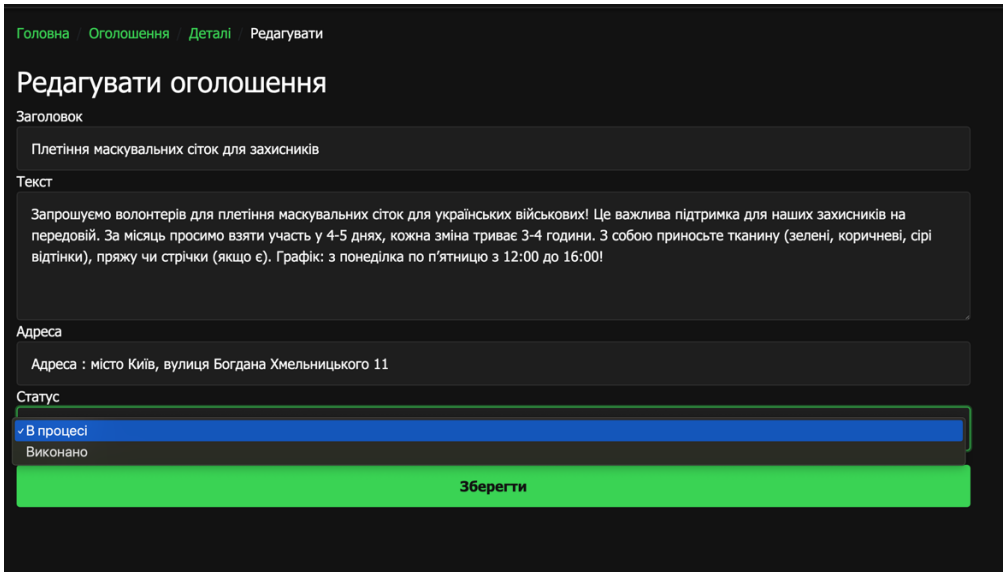

Рисунок 4.21 – Сторінка редагування оголошення

При зміні статусу на «Виконано», оголошення зникає з дошки оголошення, але залишається в профілі користувача з позначкою «DONE» (рисунок 4.22).

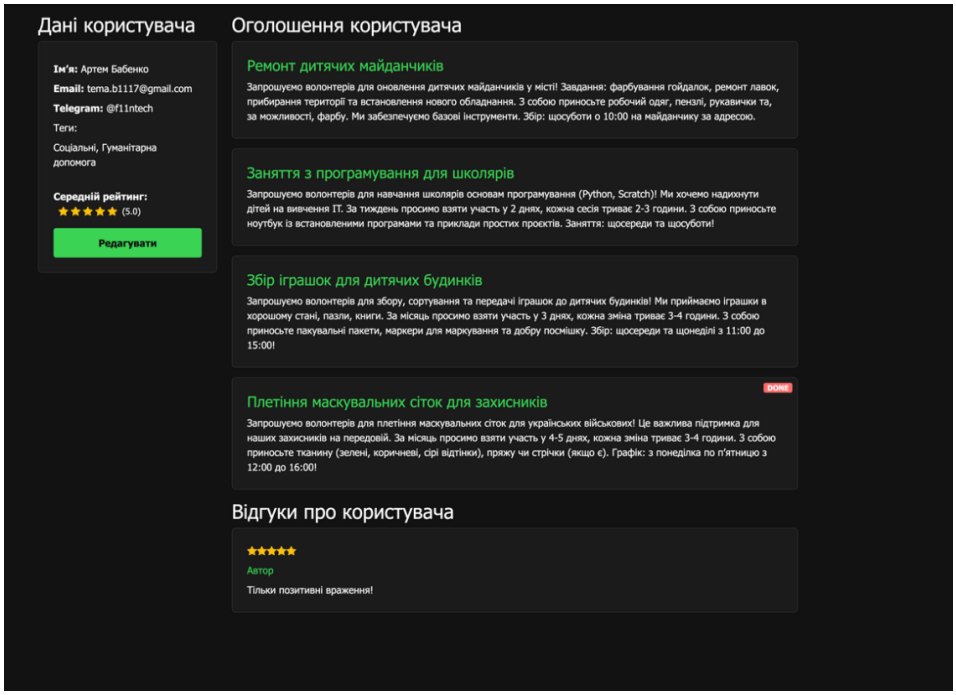

Рисунок 4.22 – Вигляд виконаного оголошення в профілі автора

Інші користувачі зможуть його побачити тільки якщо перейдуть на профіль автора, але уже не зможуть відгукнутись на нього (рисунок 4.23).

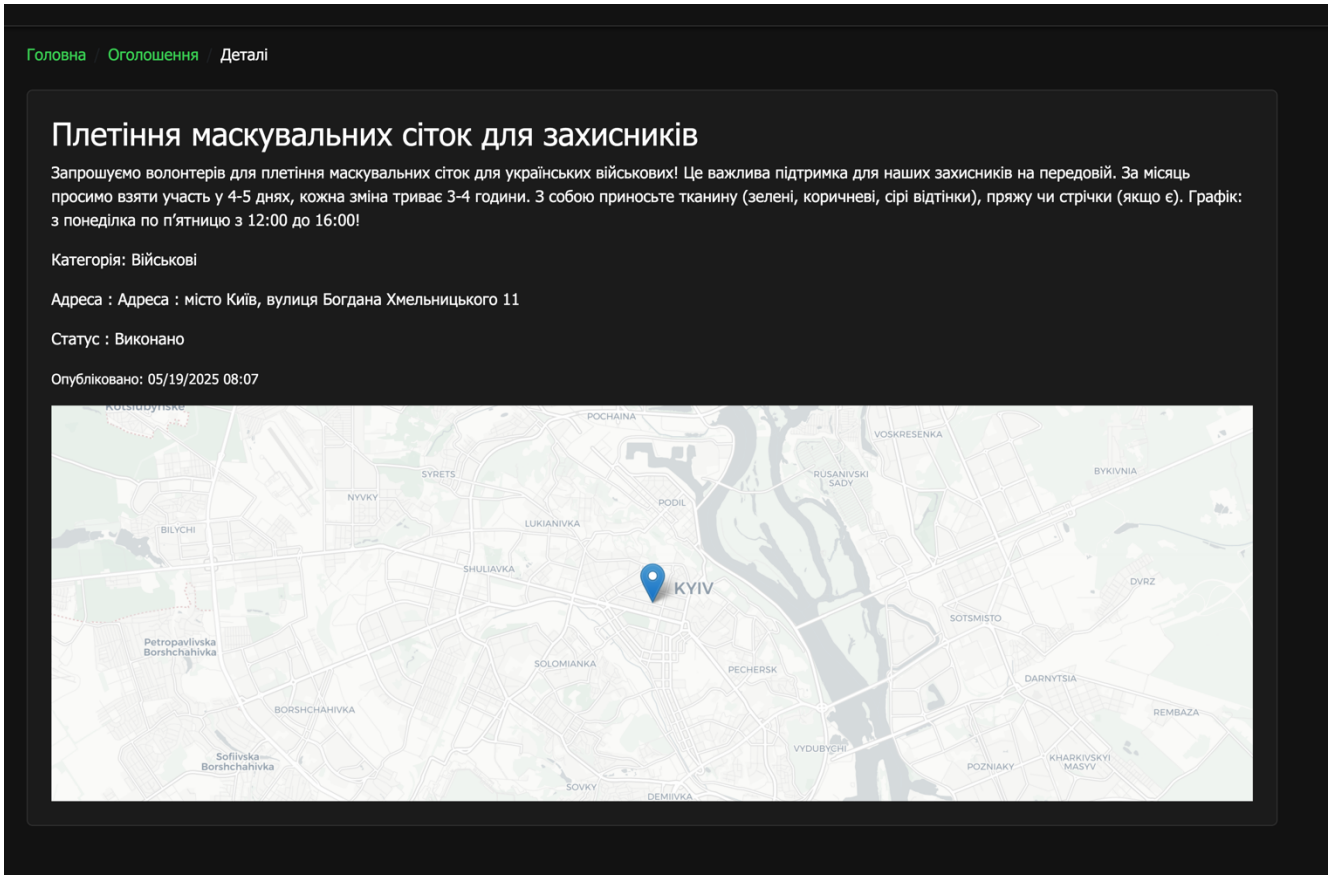

Рисунок 4.23 – Перегляд деталей виконаного оголошення

Рисунок 4.23 демонструє приклад перегляду деталей виконаного оголошення.

4.2.4 Відгуки користувачів

Користувачі веб-додатку мають змогу залишати відгуки з оцінками по 5-бальній системі оцінювання один одному (рисунок 4.24).

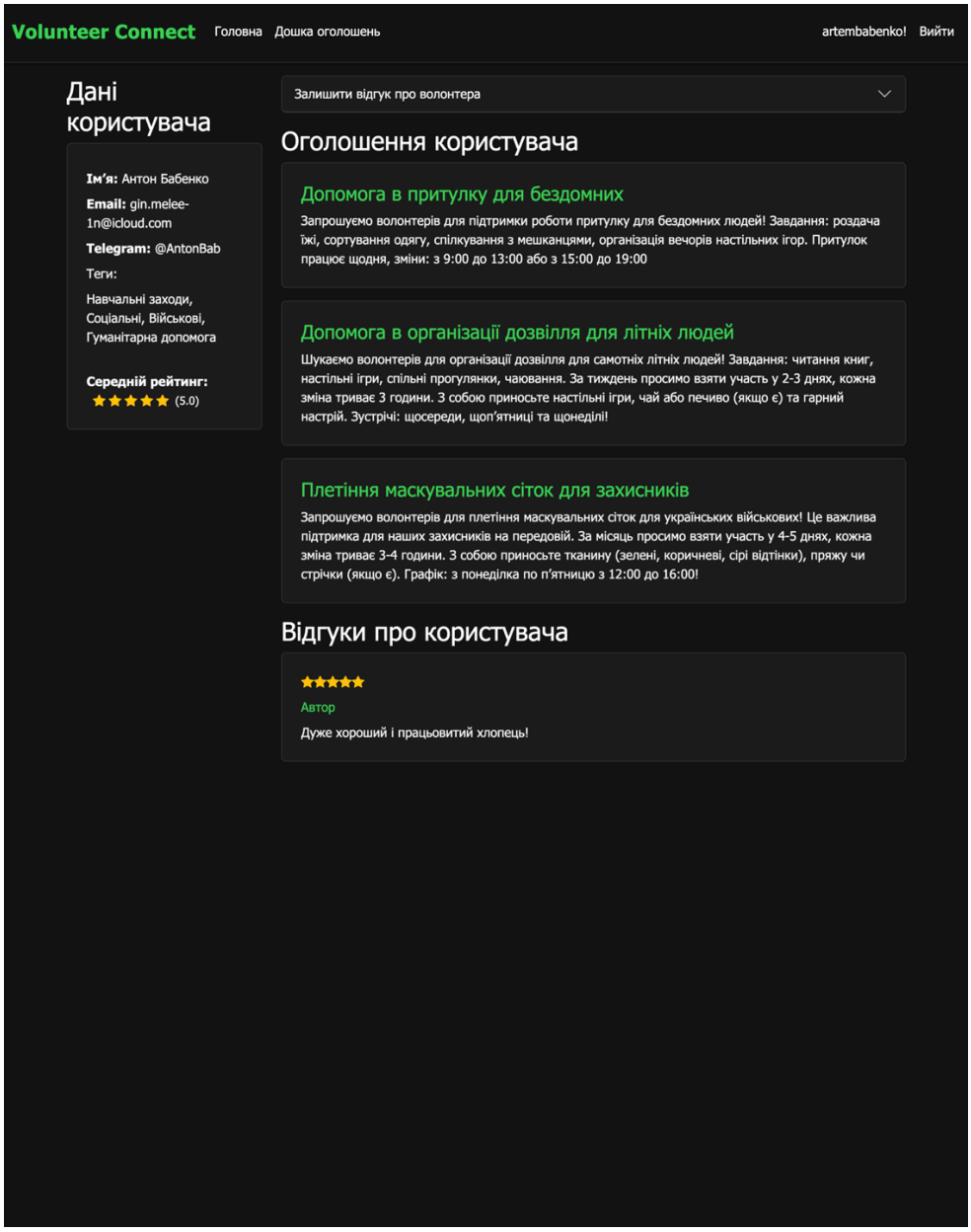

Рисунок 4.24 – Відображення відгуків в профілі користувача

Логіка написання відгуків реалізована таким чином, що коли користувач відгукується на оголошення, то автор цього оголошення має змогу написати відгук про даного користувача та залишити оцінку по 5-бальній системі оцінювання (рисунок 4.25).

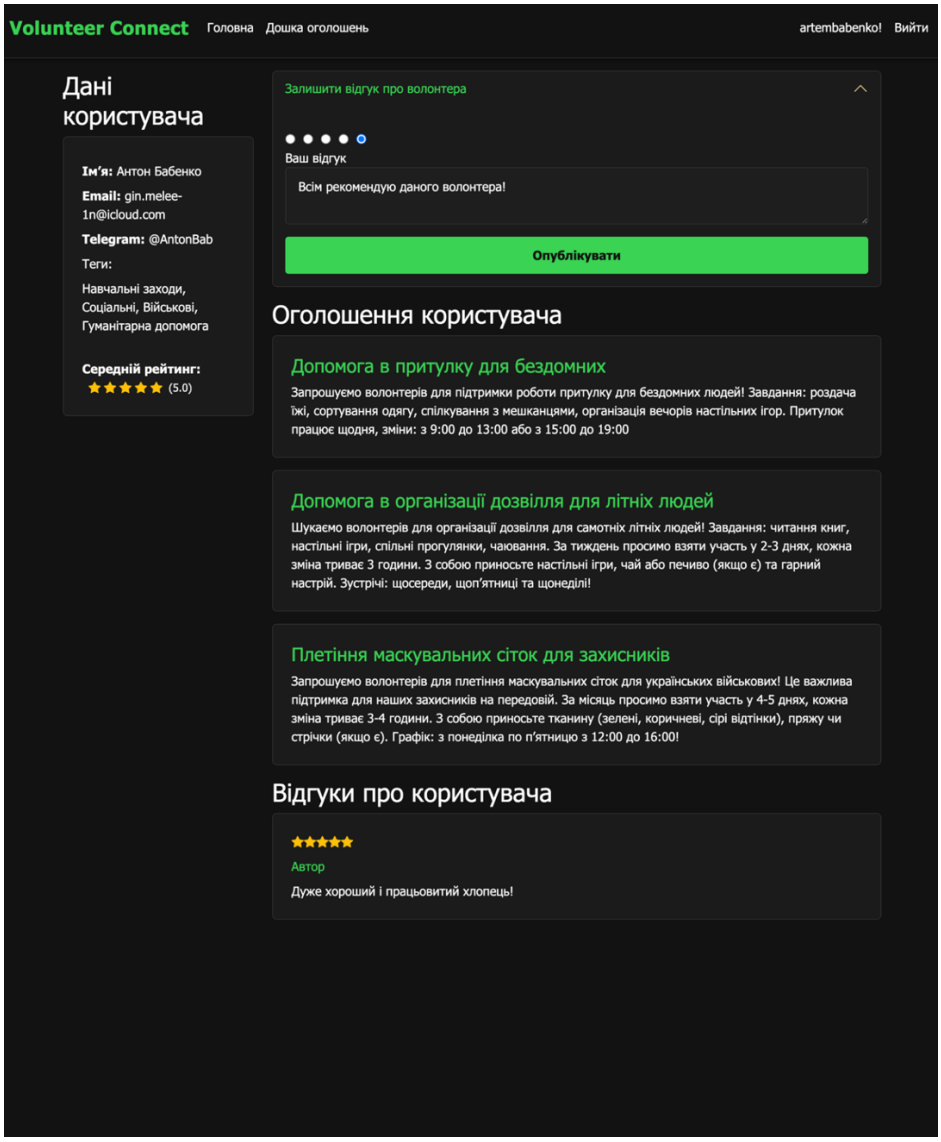

Рисунок 4.25 – Форма для написання відгуку

Слід зауважити, що будь-який користувач не може написати відгук іншому волонтеру. Це зроблено для того, щоб рейтинг та самі відгуки були максимально чесні та справедливі.

4.2.4 Перегляд статистики використання додатку

В системі є два типи користувача, один з них це адміністратор. Він має трішки інший функціонал, на відміну від волонтера. Адміністратор не може публікувати оголошення, але він може редагувати та видаляти оголошення всіх

користувачів. Також в адміністратора є доступ до перегляду статистики (рисунок 4.26).

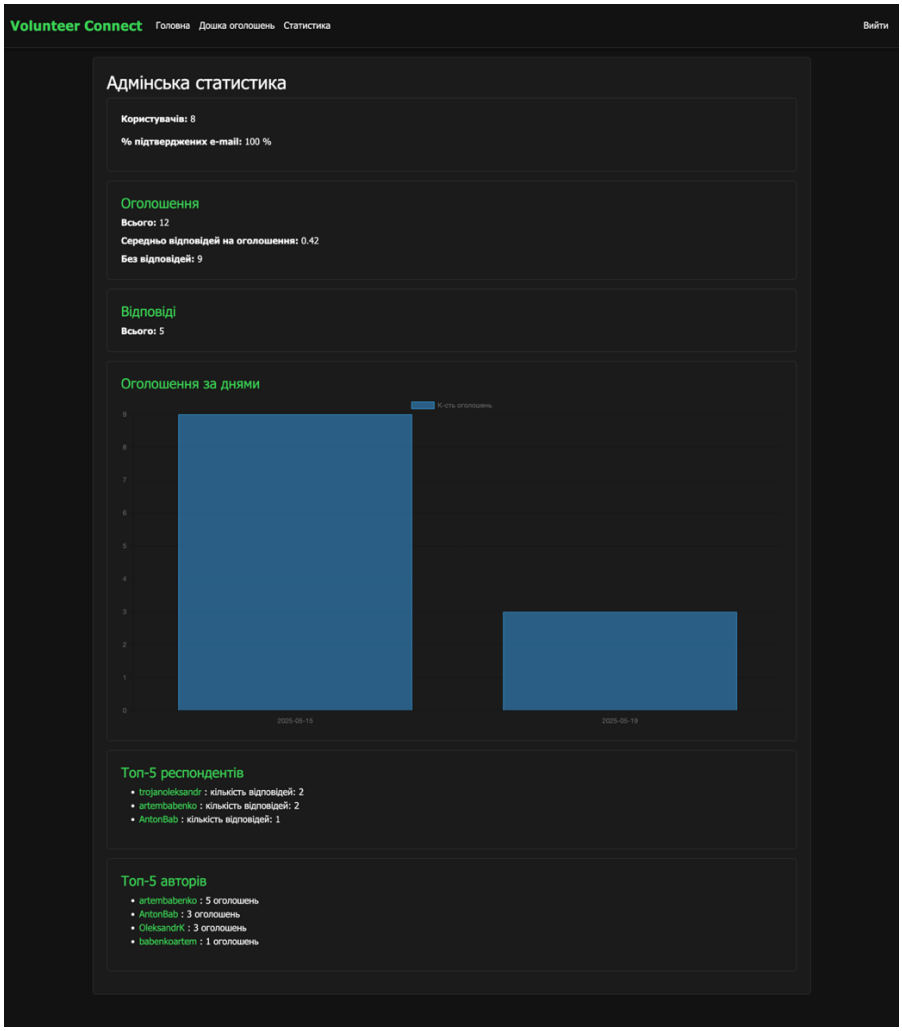

Рисунок 4.26 – Сторінка із статистикою адміністратора

На рисунку 4.26 ми бачимо, що адміністратор має змогу переглядати загальну кількість користувачів, яка зареєстрована, переглядати % користувачів, в яких підтверджена електронна пошта. Також статистика включає в себе загальну кількість оголошень, кількість оголошень без відповіді та загальну кількість відповідей на усі оголошення. Також можна побачити загальну кількість оголошень за днями та топ-5 авторів і респондентів.

ВИСНОВКИ

У результаті виконаної дипломної роботи проведено комплексне дослідження та розробку веб-додатку, спрямованого на оптимізацію взаємодії між волонтерами, що забезпечує швидкий обмін інформацією, координацію спільних дій та ефективне управління завданнями.

На основі ретельного аналізу сучасних підходів, методів та алгоритмів розв’язання поставленої задачі обґрунтовано доцільність використання монолітної багатошарової архітектури з принципами "чистої архітектури", яка забезпечує модульність і легкість підтримки системи.

Застосування патерну CQRS у поєднанні з бібліотекою MediatR дозволило оптимізувати обробку команд і запитів, підвищивши продуктивність платформи.

Для забезпечення безпеки використано токен-базовану аутентифікацію за допомогою JSON Web Token (JWT) із refresh tokens, що гарантує надійний захист персональних даних користувачів.

Синхронна міжсервісна взаємодія реалізована через gRPC із протоколом HTTP/2, що забезпечує високу швидкість передачі даних, тоді як асинхронна обробка повідомлень здійснюється за допомогою Apache Kafka, що дозволяє ефективно управляти потоками подій у фоновому режимі. Ці методи в сукупності створюють міцну основу для стабільної та гнучкої роботи платформи, відповідаючи вимогам волонтерської діяльності.

Аналіз програмних засобів дозволив обґрунтувати вибір оптимального технологічного стеку для реалізації веб-додатку.

Фреймворк ASP.NET Core обрано для серверної частини завдяки його кросплатформності, високій продуктивності та глибокій інтеграції з екосистемою C#.

Для клієнтської частини використано ASP.NET MVC, що забезпечує швидке завантаження сторінок і адаптивний інтерфейс через серверний рендеринг.

Робота з базою даних Microsoft SQL Server здійснюється за допомогою Entity Framework Core, що спрощує взаємодію з даними, забезпечує безпеку запитів і підтримує складні операції.

Технологія SignalR застосована для оновлення даних у реальному часі на головній сторінці, що значно підвищує зручність користування платформою.

Контейнеризація за допомогою Docker забезпечує ізоляцію компонентів, спрощує розгортання та сприяє масштабованості системи. Використання цих інструментів дозволило створити надійну, продуктивну та зручну у підтримці платформу, яка відповідає сучасним стандартам розробки програмного забезпечення.

Розроблений веб-додаток успішно реалізує задані функції, забезпечуючи волонтерам зручний інструмент для взаємодії. Система підтримує реєстрацію та авторизацію користувачів, створення, редагування й перегляд оголошень, обробку відповідей, відображення статистики та відгуків, а також інтеграцію з email-сповіщеннями через Apache Kafka.

Архітектура платформи дозволяє ефективно обробляти великі обсяги запитів, зберігаючи стабільність навіть у пікові періоди активності. Проведене тестування підтвердило коректність роботи програмного забезпечення, його відповідність функціональним і нефункціональним вимогам, а також здатність забезпечувати швидку реакцію на дії користувачів і безпечну обробку даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Date C. J. An Introduction to Database Systems. 8th ed. Lebanon, Indiana, U.S.A : Pearson, 2003. 1024 p.

2. Documentation. Apache Kafka. URL: https://kafka.apache.org/documentation/ (date of access: 19.05.2025).

3. Documentation. gRPC. URL: https://grpc.io/docs/ (date of access: 19.05.2025).

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

5. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p.

6. Freeman A. Pro ASP.NET Core 6. Berkeley, CA : Apress, 2022. URL: https://doi.org/10.1007/978-1-4842-7957-1 (date of access: 19.05.2025).

7. GitHub - jbogard/MediatR: Simple, unambitious mediator implementation in .NET. GitHub. URL: https://github.com/jbogard/MediatR (date of access: 19.05.2025).

8. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017. 624 p.

9. Kumar A. CQRS (Command Query Responsibility Segregation). Independently Published, 2019. 210p.

10. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson, 2017. 432 p.

11. Merkel D. Docker: Lightweight linux containers for consistent development and deployment. Linux J., 2014. 239p.

12. Microsoft docs. Signalr documentation. Microsoft. URL: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-9.0.

13. Miller R., Lerman J. Programming Entity Framework: Dbcontext. O'Reilly Media, Incorporated, 2012. 258p.

14. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Incorporated, 2021. 612 p.

15. Overview of ASP.NET Core MVC. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/aspnet (date of access: 19.05.2025).

16. Overview of ASP.NET Core SignalR. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/aspnet/core/signalr (date of access: 19.05.2025).

17. Overview of Entity Framework Core - EF Core. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-us/ef/core/ (date of access: 19.05.2025).

18. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media, 2020. 432 p.

19. Vernon V. Implementing Domaindriven Design. Addison-Wesley Professional, 2012. 656p.

20. Wohlin C., Runeson P., Höst M. Experimentation in Software Engineering. 2012th ed. Berlin, Germany : Springer, 2012. 260 p.

ДОДАТОК А

«Веб-додаток для взаємодії між волонтерами»

Лістинг основних класів

УКР.НТУУ_КПІ_ЦТЕ_НН ІАТЕ_ТР-11

Аркушів 23

Київ – 2025

```
syntax = "proto3";

package account;
option csharp_namespace = "AccountService.Grpc";

import "google/protobuf/empty.proto";

message LoginRequest {
  string Email    = 1;
  string Password = 2;
}

message LoginResponse {
  string Id            = 1;
  string UserName      = 2;
  string Email         = 3;
  string Telegram      = 4;
  string Token         = 5;
  bool         EmailConfirmed = 6;
}

message RegisterRequest {
  string FirstName = 1;
  string LastName  = 2;
  string Email     = 3;
  string Telegram  = 4;
  string UserName  = 5;
  string Password  = 6;
  repeated int32 SelectedTagIds = 7;
}

message ConfirmEmailRequest {
  string Token = 1;
}

message SendConfirmationRequest {
  string BaseUrl = 1;
}

message ProfileResponse{
  string Id                                 = 1;
  string UserName                   = 2;
  string FirstName                  = 3;
  string LastName                   = 4;
  string Email                              = 5;
  string Telegram                   = 6;
}

message ProfileRequest
{
        string UserId = 1;
}


message UpdateProfileRequest {
  string firstName   = 1;
  string lastName    = 2;
  string telegram    = 3;
  string userName    = 4;
  repeated int32 selectedTagIds = 5;
}

service Account {
  rpc Login(LoginRequest)                                                                   returns (LoginResponse);
  rpc Register(RegisterRequest)                                                     returns (LoginResponse);
  rpc Profile(ProfileRequest)                                       returns (ProfileResponse);
  rpc Account(google.protobuf.Empty)                                        returns (ProfileResponse);
  rpc ConfirmEmail(ConfirmEmailRequest)       returns (LoginResponse);
  rpc SendConfirmation(SendConfirmationRequest) returns (google.protobuf.Empty);
  rpc UpdateProfile (UpdateProfileRequest) returns (google.protobuf.Empty);
}

syntax = "proto3";

package announcements;
option csharp_namespace = "AnnouncementsService.Grpc";

import "google/protobuf/empty.proto";

message Announcement {
```

```
  int32 id        = 1;
  string title     = 2;
  string content   = 3;
  string authorId  = 4;
  string address  = 5;
  bool status  = 6;
  int64  timestamp = 7;
  Tag tag = 8;
}

message AllTag {
        repeated Tag tags = 1;
}

message Tag{
        int32 id = 1;
        string name = 2;
}
message AnnouncementWithInfoResponse {
  int32 id        = 1;
  string title     = 2;
  string content   = 3;
  string authorId  = 4;
  string authorEmail  = 5;
  string address  = 6;
  bool status  = 7;
  int64  timestamp = 8;
  Tag tag = 9;
  repeated AnnouncementResponse responses = 10;
  double latitude   = 11;
  double longitude   = 12;
}

message AnnouncementResponse
{
  string authorId  = 1;
  string authorEmail  = 2;
  string authorUserName  = 3;
  string messageText    = 4;
}


message UpdateAnnouncementRequest {
  int32 id        = 1;
  string title     = 2;
  string content   = 3;
  string address   = 4;
  bool status  = 5;
}

message PublishRequest {
  string title     = 1;
  string content   = 2;
  string address   = 3;
  int32 tagId   = 4;
  double latitude   = 5;
  double longitude   = 6;
}

message GetAnnouncementRequest {
  int32 id = 1;
}

message DeleteAnnouncementRequest {
  int32 id = 1;
}

message ListAnnouncementsResponse {
  repeated Announcement announcements = 1;
}

message RespondRequest {
  int32 announcementId = 1;
  string messageText    = 3;
}

message AnnocmentsByUserRequest
{
        string userId = 1;
}
```

```
message UserRatingByUserRequest
{
		string userId = 1;
}

message UserRatingCommentResponse {
  int32 stars = 1;
  string comment = 2;
  string authorId = 3;
}


message CreateUserRatingCommentRequest {
  int32 stars = 1;
  string comment = 2;
  string userId = 3;
}

message ListUserRatingComment {
  repeated UserRatingCommentResponse userRatingCommentResponse = 1;
}


message CanWriteStarCommentResponse{
		bool canWrite = 1;
}

message CanWriteStarCommentRequest{
		string userId = 1;
}

message TagsRequest {
   repeated int32 tag_ids = 1;
}

message GetTagsRequest{
		string UserId = 1;
}

message GetTagsResponse{
		repeated Tag Tags = 1;
}

service Announcements {
  rpc PublishAnnouncement(PublishRequest) returns (google.protobuf.Empty);
  rpc SubscribeAnnouncements(google.protobuf.Empty) returns (stream Announcement);
  rpc ListAnnouncements(Tag) returns (ListAnnouncementsResponse);
  rpc GetAnnouncement(GetAnnouncementRequest) returns (AnnouncementWithInfoResponse);
  rpc GetAnnouncementsByUser(AnnocmentsByUserRequest) returns (ListAnnouncementsResponse);
  rpc UpdateAnnouncement(UpdateAnnouncementRequest) returns (google.protobuf.Empty);
  rpc RespondToAnnouncement(RespondRequest) returns (google.protobuf.Empty);
  rpc DeleteAnnouncment(DeleteAnnouncementRequest) returns (google.protobuf.Empty);
  rpc GetTags(google.protobuf.Empty) returns (AllTag);
  rpc GetUserRating(UserRatingByUserRequest) returns (ListUserRatingComment);
  rpc CreateUserRating(CreateUserRatingCommentRequest) returns (google.protobuf.Empty);
  rpc CanWriteStar(CanWriteStarCommentRequest) returns (CanWriteStarCommentResponse);
  rpc ListAnnouncementsByTags(TagsRequest) returns (ListAnnouncementsResponse);
  rpc GetTagsByUser (GetTagsRequest) returns (GetTagsResponse);
}

using Grpc.Core;

using Google.Protobuf.WellKnownTypes;

using Microsoft.AspNetCore.Authorization;

using MediatR;

using AutoMapper;

using Application.Features.Accounts.Requests.Commands;

using AccountService.Grpc;
using GrpcApi.Extensions;
using Application.Features.Accounts.Requests.Queries;
using Application.Features.Announcements.Requests.Queries;
```

```
namespace GrpcApi.Services;

public class AccountGrpcService : Account.AccountBase
{
	private readonly IMediator _mediator;
	private readonly IMapper _mapper;

	public AccountGrpcService(IMediator mediator,IMapper mapper)
	{
		_mediator = mediator;
		_mapper = mapper;
	}

	[AllowAnonymous]
	public override async Task<LoginResponse> Login(LoginRequest request, ServerCallContext context)
	{
		var command = _mapper.Map<LoginAccountCommand>(request);
		var result = await _mediator.Send(command, context.CancellationToken);
		return _mapper.Map<LoginResponse>(result);
	}

	[AllowAnonymous]
	public override async Task<LoginResponse> Register(RegisterRequest request, ServerCallContext context)
	{
		var command = _mapper.Map<RegisterAccountCommand>(request);
		var result = await _mediator.Send(command, context.CancellationToken);
		return _mapper.Map<LoginResponse>(result);
	}

	[Authorize]
	public override async Task<LoginResponse> ConfirmEmail(ConfirmEmailRequest request, ServerCallContext context)
	{
		var userId = context.GetUserId();
		var cmd = new ConfirmEmailCommand(userId, request.Token);
		var result = await _mediator.Send(cmd, context.CancellationToken);
		return _mapper.Map<LoginResponse>(result);
	}

	[Authorize]
	public override async Task<Empty> SendConfirmation(SendConfirmationRequest request, ServerCallContext context)
	{
		var userId = context.GetUserId();
		var cmd = new SendEmailConfirmationCommand(request.BaseUrl, userId);
		await _mediator.Send(cmd, context.CancellationToken);
		return new Empty();
	}

	public override async Task<ProfileResponse> Profile(ProfileRequest request, ServerCallContext context)
	{
		var cmd = new GetProfileQuery(request.UserId);
		var result = await _mediator.Send(cmd, context.CancellationToken);

		var dto = _mapper.Map<ProfileResponse>(result);

		return dto;
	}

	[Authorize]
	public override async Task<Empty> UpdateProfile(UpdateProfileRequest request, ServerCallContext context)
	{
		var userId = context.GetUserId();

		var cmd = new UpdateProfileAccountCommand(
			userId,
			request.FirstName,
			request.LastName,
			request.Telegram,
			request.UserName,
			request.SelectedTagIds.ToList()
		);

		await _mediator.Send(cmd, context.CancellationToken);
		return new Empty();
	}
}

using AnnouncementsService.Grpc;
using Application.Features.Announcements.Requests.Commands;
using Application.Features.Announcements.Requests.Queries;
using Application.Features.Stats.Requests.Commands;
```

```
using Application.Features.UserRating.Requests.Commands;
using Application.Features.UserRating.Requests.Queries;
using AutoMapper;

using Google.Protobuf.WellKnownTypes;

using Grpc.Core;
using GrpcApi.Extensions;
using MediatR;

using Microsoft.AspNetCore.Authorization;


namespace GrpcApi.Services;

public class AnnouncementsGrpcService : Announcements.AnnouncementsBase
{
	private readonly IMediator _mediator;
	private readonly IMapper _mapper;

	public AnnouncementsGrpcService(IMediator mediator, IMapper mapper)
	{
		_mediator = mediator;
		_mapper = mapper;
	}

	[Authorize]
	public override async Task<Empty> PublishAnnouncement(PublishRequest request, ServerCallContext context)
	{
		var userId = context.GetUserId();

		await _mediator.Send(new PublishAnnouncementCommand(
			request.Title,
			request.Content,
			userId,
			request.Address,
			request.TagId,request.Latitude,request.Longitude),
			context.CancellationToken);

		return new Empty();
	}

	public override async Task SubscribeAnnouncements(Empty request, IServerStreamWriter<Announcement> responseStream,
ServerCallContext context)
	{
		var stream = await _mediator.Send(new SubscribeAnnouncementsQuery(), context.CancellationToken);

		await foreach (var dto in stream.WithCancellation(context.CancellationToken))
		{
			var proto = _mapper.Map<Announcement>(dto);
			await responseStream.WriteAsync(proto);
		}
	}

	public override async Task<ListAnnouncementsResponse> ListAnnouncements(Tag request, ServerCallContext context)
	{
		var dtos = await _mediator.Send(new GetRecentAnnouncementsQuery() { TagId = request.Id }, context.CancellationToken);

		var response = new ListAnnouncementsResponse();

		response.Announcements.AddRange(_mapper.Map<List<Announcement>>(dtos));

		return response;
	}

	[Authorize]
	public override async Task<Empty> RespondToAnnouncement(RespondRequest request, ServerCallContext context)
	{
		var userId = context.GetUserId();

		var cmd = new RespondToAnnouncementCommand(userId, request.AnnouncementId, request.MessageText);

		await _mediator.Send(cmd, context.CancellationToken);

		return new Empty();
	}

	public override async Task<AnnouncementWithInfoResponse> GetAnnouncement(GetAnnouncementRequest request, ServerCallContext
context)
	{
```

```
                var cmd = new GetAnnouncementQuery(request.Id);

                var result = await _mediator.Send(cmd, context.CancellationToken);

                var dto = _mapper.Map<AnnouncementWithInfoResponse>(result);

                return dto;
        }

        [Authorize]
        public override async Task<Empty> UpdateAnnouncement(UpdateAnnouncementRequest request, ServerCallContext context)
        {
                var userId = context.GetUserId();

                var cmd = new UpdateAnnouncementCommand(request.Id, request.Title, request.Content, userId, request.Address, request.Status);

                await _mediator.Send(cmd, context.CancellationToken);

                return new Empty();
        }

        public override async Task<ListAnnouncementsResponse> GetAnnouncementsByUser(AnnocmentsByUserRequest request, ServerCallContext
context)
        {
                var dtos = await _mediator.Send(new GetAnnouncementsByUserQuery(request.UserId), context.CancellationToken);

                var response = new ListAnnouncementsResponse();

                response.Announcements.AddRange(_mapper.Map<List<Announcement>>(dtos));

                return response;
        }

        [Authorize]
        public override async Task<Empty> DeleteAnnouncment(DeleteAnnouncementRequest request, ServerCallContext context)
        {

                var cmd = new DeleteAnnouncementCommand(request.Id);

                await _mediator.Send(cmd, context.CancellationToken);

                return new Empty();
        }

        public override async Task<AllTag> GetTags(Empty request, ServerCallContext context)
        {
                var query = new GetAllTagsQuery();

                var dtos = await _mediator.Send(query, context.CancellationToken);

                var allTags = new AllTag();

                allTags.Tags.AddRange(_mapper.Map<List<Tag>>(dtos));

                return allTags;
        }


        public override async Task<ListUserRatingComment> GetUserRating(UserRatingByUserRequest request, ServerCallContext context)
        {
                var query = new GetUserRatingCommentByUserQuery(request.UserId);

                var dtos = await _mediator.Send(query, context.CancellationToken);

                var listUserRatingComment = new ListUserRatingComment();

                listUserRatingComment.UserRatingCommentResponse.AddRange(_mapper.Map<List<UserRatingCommentResponse>>(dtos));

                return listUserRatingComment;
        }


        [Authorize]
        public override async Task<Empty> CreateUserRating(CreateUserRatingCommentRequest request, ServerCallContext context)
        {
                var userId = context.GetUserId();

                var cmd = new CreateUserRatingCommentCommand(request.Stars, request.Comment, request.UserId, userId);

                await _mediator.Send(cmd, context.CancellationToken);
```

```
                return new Empty();
        }


        [Authorize]
        public override async Task<CanWriteStarCommentResponse> CanWriteStar(CanWriteStarCommentRequest request, ServerCallContext
context)
        {
                var userId = context.GetUserId();

                var cmd = new CanWriteStarCommentCommand(userId, request.UserId);

                var result = await _mediator.Send(cmd, context.CancellationToken);

                return new CanWriteStarCommentResponse() { CanWrite = result };

        }

        public override async Task<ListAnnouncementsResponse> ListAnnouncementsByTags(TagsRequest request, ServerCallContext context)
        {
                var dtos = await _mediator.Send(new GetAnnouncementsByTagsQuery { TagIds = request.TagIds.ToList() },
context.CancellationToken);
                var response = new ListAnnouncementsResponse();
                response.Announcements.AddRange(_mapper.Map<List<Announcement>>(dtos));
                return response;
        }

        public override async Task<GetTagsResponse> GetTagsByUser(GetTagsRequest request, ServerCallContext context)
        {
                var cmd = new GetTagsQuery(request.UserId);

                var result = await _mediator.Send(cmd, context.CancellationToken);

                var response = new GetTagsResponse();

                response.Tags.AddRange(_mapper.Map<List<Tag>>(result));

                return response;
        }
}

using Application.Features.Stats.Requests.Queries;

using AutoMapper;

using Google.Protobuf.WellKnownTypes;

using Grpc.Core;

using MediatR;

using Microsoft.AspNetCore.Authorization;

using StatsService.Grpc;


namespace GrpcApi.Services;


public class StatsGrpcService : Stats.StatsBase
{
        private readonly IMediator _mediator;
        private readonly IMapper _mapper;

        public StatsGrpcService(IMediator mediator, IMapper mapper)
        {
                _mediator = mediator;
                _mapper = mapper;
        }

        [Authorize(Roles = "Administrator")]
        public override async Task<AdminStatsResponse> GetAdminStats(Empty request, ServerCallContext context)
        {
                var dto = await _mediator.Send(new GetAdminStatsQuery(), context.CancellationToken);

                var grpcResponse = _mapper.Map<AdminStatsResponse>(dto);

                return grpcResponse;
        }
```

```
	public override async Task<MainStatsResponse> GetMainStats(Empty request, ServerCallContext context)
	{
		var dto = await _mediator.Send(new GetMainStatsQuery(), context.CancellationToken);

		var grpcResponse = _mapper.Map<MainStatsResponse>(dto);

		return grpcResponse;
	}
}
```

```
using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;
using UI.Contracts;


namespace UI.Controllers;

[Authorize(Roles = "Administrator")]
public class AdminStatsController : Controller
{
	private readonly IAdminStatsService _adminStatsService;

	public AdminStatsController(IAdminStatsService adminStatsService)
	{
		_adminStatsService = adminStatsService;
	}

	public async Task<IActionResult> Index()
	{
		var result = await _adminStatsService.GetAdminStats();

		if (!result.Succeeded)
		{
			foreach (var err in result.Errors)
				ModelState.AddModelError("", string.Join("; ", err.Value));

		}

		return View(result.Data);
	}
}
```

```
using Grpc.Core;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using UI.Contracts;
using UI.Models.Announcements;


namespace UI.Controllers;

public class AnnouncementsController : Controller
{
	private readonly IPublishAnnouncementsService _publishAnnouncementsService;
	private readonly ILogger<AnnouncementsController> _logger;

	public AnnouncementsController(IPublishAnnouncementsService publishAnnouncementsService,
	                               ILogger<AnnouncementsController> logger)
	{
		_publishAnnouncementsService = publishAnnouncementsService;
		_logger = logger;
	}

	public async Task<IActionResult> Index([FromQuery] List<int> tagIds)
	{
		var tagsResponse = await _publishAnnouncementsService.GetAllTagsAsync();
		if (!tagsResponse.Succeeded)
		{
			foreach (var err in tagsResponse.Errors)
				ModelState.AddModelError("", string.Join("; ", err.Value));
		}
		var tags = tagsResponse.Data;

		var annResponse = await _publishAnnouncementsService.GetAnnouncementsByTagsAsync(tagIds);
		if (!annResponse.Succeeded)
		{
			foreach (var err in annResponse.Errors)
```

```
                ModelState.AddModelError("", string.Join("; ", err.Value));
        }
        var announcements = annResponse.Data;

        var me = User.FindFirstValue("uid");
        foreach (var a in announcements)
        {
                if (a.AuthorId == me) a.Badges.Add("OWN");
                if (a.Status) a.Badges.Add("done");
        }

        ViewBag.Tags = tags;
        ViewBag.SelectedTagIds = tagIds;

        return View(announcements);
}


[HttpGet]
public async Task<IActionResult> Filter(int TagId)
{
        return RedirectToAction("Index", new { tagId = TagId });
}

[HttpPost, Authorize, ValidateAntiForgeryToken]
public async Task<IActionResult> Publish(string title, string content, string address, int tagId, double latitude, double longitude)
{
        await _publishAnnouncementsService.PublishAnnouncement(title, content, address, latitude, longitude, tagId);
        return Ok();
}

public async Task<IActionResult> Details(int id)
{
        var result = await _publishAnnouncementsService.GetAnnouncementAsync(id);

        if (!result.Succeeded)
        {
                foreach (var err in result.Errors)
                        ModelState.AddModelError("", string.Join("; ", err.Value));

        }

        return View(result.Data);
}


[HttpGet, Authorize]
public async Task<IActionResult> Update(int id)
{
        var result = await _publishAnnouncementsService.GetAnnouncementAsync(id);

        if (!result.Succeeded)
        {
                foreach (var err in result.Errors)
                        ModelState.AddModelError("", string.Join("; ", err.Value));

        }

        var vm = new UpdateAnnouncementVM
        {
                Id = result.Data.Id,
                Title = result.Data.Title,
                Content = result.Data.Content,
                Address = result.Data.Address,
                Status = result.Data.Status,
        };

        return View(vm);
}

[HttpPost, Authorize, ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id)
{
        await _publishAnnouncementsService.DeleteAnnouncementAsync(id);

        return RedirectToAction(nameof(Index));
}


[HttpPost, Authorize, ValidateAntiForgeryToken]
```

```
	public async Task<IActionResult> Update(UpdateAnnouncementVM updateAnnouncementVM)
	{
		await _publishAnnouncementsService.UpdateAnnouncementAsync(updateAnnouncementVM);

		return RedirectToAction(nameof(Details), new { id = updateAnnouncementVM.Id });
	}


	[HttpPost, ValidateAntiForgeryToken]
	public async Task<IActionResult> Respond(int announcementId, string messageText)
	{
		await _publishAnnouncementsService.RespondToAnnouncement(announcementId, messageText);

		TempData["ResponseSent"] = "Ваш відгук успішно надіслано власнику оголошення.";

		return RedirectToAction(nameof(Details), new { id = announcementId });
	}
}
using Microsoft.AspNetCore.Mvc;

using System.Diagnostics;
using UI.Contracts;
using UI.Models;


namespace UI.Controllers;

public class HomeController : Controller
{
	private readonly ILogger<HomeController> _logger;
	private readonly IAdminStatsService _adminStatsService;

	public HomeController(ILogger<HomeController> logger, IAdminStatsService adminStatsService)
	{
		_logger = logger;
		_adminStatsService = adminStatsService;
	}

	public async Task<IActionResult> Index()
	{
		var result = await _adminStatsService.GetMainStats();

		if (!result.Succeeded)
		{
			return BadRequest();
		}

		return View(result.Data);
	}

	[ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
	public IActionResult Error()
	{
		return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
	}
}

using Grpc.Core;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewEngines;
using System.Security.Claims;
using UI.Contracts;
using UI.Models.Announcements;
using UI.Models.Authentication;


namespace UI.Controllers;

public class UsersController : Controller
{
	private readonly IPublishAnnouncementsService _publishAnnouncementsService;
	private readonly IAuthenticationService _authService;
	private readonly ILogger<UsersController> _logger;

	public UsersController(IAuthenticationService authService,
	                       IPublishAnnouncementsService publishAnnouncementsService,
	                       ILogger<UsersController> logger)
	{
		_publishAnnouncementsService = publishAnnouncementsService;
```

```
        _authService = authService;
        _logger = logger;
}

[HttpGet]
public IActionResult Login(string returnUrl = null)
{
        ViewData["ReturnUrl"] = returnUrl;
        return View();
}


[HttpGet, Authorize]
public async Task<IActionResult> MyProfile()
{
        var userId = User.FindFirstValue("uid");

        return await PublicProfile(userId);
}

[HttpGet]
public async Task<IActionResult> PublicProfile(string userId)
{
        var result = await _publishAnnouncementsService.GetAnnouncementsByUserAsync(userId);

        var profileInfoResult = await _authService.GetProfileInfo(userId);

        var startRating = await _publishAnnouncementsService.GetStarsRating(userId);

        var canWrite = await _publishAnnouncementsService.CanWrite(userId);

        var tegs = await _publishAnnouncementsService.GetTagsAsync(userId);

        if (!result.Succeeded)
                return BadRequest(result.Errors);

        if (!profileInfoResult.Succeeded)
                return BadRequest(result.Errors);

        foreach (var announcement in result.Data)
        {
                if (announcement.Status)
                {
                        announcement.Badges.Add("done");
                }
        }


        var vm = new ProfilePageVM
        {
                Announcements = result.Data,
                User = profileInfoResult.Data,
                UserRatingComments = startRating.Data,
                CanWrite = canWrite,
                Tags = tegs.Data.Select(x=> x.Name).ToList(),
        };

        return View("Profile", vm);
}

[HttpPost, ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginVM login, string returnUrl = null)
{
        ViewData["ReturnUrl"] = returnUrl;

        if (!ModelState.IsValid)
                return View(login);

        _logger.LogInformation("Login attempt for email: {Email}", login.Email);
        var result = await _authService.Authenticate(login.Email, login.Password);

        if (!result.Succeeded)
        {
                foreach (var err in result.Errors)
                        ModelState.AddModelError("", string.Join("; ", err.Value));

                _logger.LogWarning("Login failed for user: {Email}", login.Email);
                return View(login);
        }
```

```
                if (result.Data.EmailConfirmed == false)
                {
                        _logger.LogInformation("Email not confirmed for user: {Email}", login.Email);
                        return RedirectToAction("EmailNotConfirmed", new { email = login.Email });
                }

                _logger.LogInformation("Login succeeded for user: {Email}", login.Email);
                return LocalRedirect(returnUrl ?? Url.Content("~/"));
        }

        [HttpGet, Authorize]
        public IActionResult EmailNotConfirmed()
        {
                return View();
        }

        [HttpPost, Authorize, ValidateAntiForgeryToken]
        public async Task<IActionResult> EmailSent()
        {
                await _authService.SendConfirmationEmailLink();
                return View();
        }

        [HttpGet, Authorize]
        public async Task<IActionResult> EmailConfirmed(string token)
        {
                var result = await _authService.ConfirmEmail(token);

                ViewBag.EmailConfirmed = result.Succeeded;

                if (result.Succeeded)
                {
                        ViewBag.StatusMessage = "Пошта успішно підтверджена.";
                }
                else
                {
                        var errors = string.Join("; ", result.Errors.SelectMany(e => e.Value));
                        ViewBag.StatusMessage = $"Помилка підтвердження пошти: {errors}";
                }
                return View("EmailConfirmationResult");
        }

        [HttpGet]
        public async Task<IActionResult> Register()
        {
                var tagsResponse = await _publishAnnouncementsService.GetAllTagsAsync();
                if (!tagsResponse.Succeeded)
                {
                        foreach (var err in tagsResponse.Errors)
                                ModelState.AddModelError("", string.Join("; ", err.Value));
                }

                ViewBag.AvailableTags = tagsResponse.Data
                        .Select(t => new TagVM { Id = t.Id, Name = t.Name })
                        .ToList();

                return View(new RegisterVM());
        }


        [HttpPost,ValidateAntiForgeryToken]
        public async Task<IActionResult> Register(RegisterVM registration)
        {
                if (!ModelState.IsValid)
                        return View(registration);

                _logger.LogInformation("Registration attempt for email: {Email}", registration.Email);
                var result = await _authService.Register(registration);

                if (result.Succeeded)
                {
                        if (result.Data.EmailConfirmed == false)
                        {
                                _logger.LogInformation("Registration succeeded but email not confirmed for user: {Email}",
registration.Email);
                                return RedirectToAction("EmailNotConfirmed", new { email = registration.Email });
                        }
                        _logger.LogInformation("Registration succeeded for user: {Email}", registration.Email);
                        return LocalRedirect(Url.Content("~/"));
                }
```

```
		foreach (var err in result.Errors)
			ModelState.AddModelError(err.Key, string.Join("; ", err.Value));

		_logger.LogWarning("Registration failed for user: {Email}", registration.Email);
		return View(registration);
}

[HttpPost]
public async Task<IActionResult> Logout(string returnUrl = null)
{
		await _authService.Logout();
		return LocalRedirect(returnUrl ?? Url.Content("~/"));
}


[HttpPost]
[Authorize]
[ValidateAntiForgeryToken]
public async Task<IActionResult> StarRating(int stars, string comment, string userId)
{
		if (stars < 1 || stars > 5)
			ModelState.AddModelError(nameof(stars), "Оцініть від 1 до 5 зірок");

		if (string.IsNullOrWhiteSpace(comment))
			ModelState.AddModelError(nameof(comment), "Коментар не може бути порожнім");

		if (!ModelState.IsValid)
		{
			return RedirectToAction("PublicProfile", new { userId = userId });
		}

		await _publishAnnouncementsService.AddStarRating(stars, comment, userId);

		return RedirectToAction("PublicProfile", new { userId = userId });
}

[HttpGet, Authorize]
public async Task<IActionResult> EditProfile()
{
		var userId = User.FindFirstValue("uid");

		var profileResult = await _authService.GetProfileInfo(userId);
		if (!profileResult.Succeeded)
		{
			foreach (var err in profileResult.Errors)
				ModelState.AddModelError("", string.Join("; ", err.Value));
			return View("ProfileError");
		}

		var allTags = await _publishAnnouncementsService.GetAllTagsAsync();
		var userTags = await _publishAnnouncementsService.GetTagsAsync(userId);

		var vm = new UpdateProfileVM
		{
			FirstName = profileResult.Data.FirstName,
			LastName = profileResult.Data.LastName,
			Telegram = profileResult.Data.Telegram,

			AvailableTagIds = allTags.Succeeded
				? allTags.Data.Select(t => t.Id).ToList()
				: new List<int>(),

			SelectedTagIds = userTags.Succeeded
				? userTags.Data.Select(t => t.Id).ToList()
				: new List<int>()
		};

		ViewBag.AvailableTags = allTags.Succeeded
			? allTags.Data.Select(t => new TagVM { Id = t.Id, Name = t.Name }).ToList()
			: new List<TagVM>();

		return View(vm);
}

[HttpPost, Authorize, ValidateAntiForgeryToken]
public async Task<IActionResult> EditProfile(UpdateProfileVM vm)
{
		if (!ModelState.IsValid)
		{
```

```
                        var allTags = await _publishAnnouncementsService.GetAllTagsAsync();

                        ViewBag.AvailableTags = allTags.Succeeded
                                ? allTags.Data.Select(t => new TagVM { Id = t.Id, Name = t.Name }).ToList()
                                : new List<TagVM>();

                        return View(vm);
                }

                var result = await _authService.UpdateProfileAsync(vm);

                if (!result.Succeeded)
                {
                        foreach (var err in result.Errors)
                                ModelState.AddModelError(err.Key, string.Join("; ", err.Value));

                        var allTags = await _publishAnnouncementsService.GetAllTagsAsync();
                        ViewBag.AvailableTags = allTags.Succeeded
                                ? allTags.Data.Select(t => new TagVM { Id = t.Id, Name = t.Name }).ToList()
                                : new List<TagVM>();
                        return View(vm);
                }

                return RedirectToAction("MyProfile");
        }
}
using Application.Contracts.Persistence;
using Application.Features.Announcements.Requests.Commands;
using MediatR;

namespace Application.Features.Announcements.Handlers.Commands;


public class DeleteAnnouncementCommandHandler : IRequestHandler<DeleteAnnouncementCommand, Unit>
{
        private readonly IAnnouncementRepository _announcementRepository;

        public DeleteAnnouncementCommandHandler(IAnnouncementRepository announcementRepository)
        {
                _announcementRepository = announcementRepository;
        }

        public async Task<Unit> Handle(DeleteAnnouncementCommand request, CancellationToken cancellationToken)
        {

                var entity = await _announcementRepository.GetAsync(request.Id, cancellationToken);

                await _announcementRepository.DeleteAsync(entity, cancellationToken);

                return Unit.Value;
        }
}

using Application.Contracts.Infrastructure;
using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Exceptions;
using Application.Features.Announcements.Requests.Commands;

using AutoMapper;

using Domain;

using MediatR;
using RTools_NTS.Util;


namespace Application.Features.Announcements.Handlers.Commands;

public class PublishAnnouncementCommandHandler : IRequestHandler<PublishAnnouncementCommand, Unit>
{
        private readonly IAnnouncementRepository _announcementRepository;
        private readonly IEmailTemplateBuilder _emailTemplateBuilder;
        private readonly IMessageBus _messageBus;
        private readonly IUserTagRepository _userTagRepository;
        private readonly IAnnouncementNotificationPublisher _notifier;
        private readonly IMapper _mapper;

        public PublishAnnouncementCommandHandler(IAnnouncementNotificationPublisher notifier,
```

```
                                                                                IAnnouncementRepository
announcementRepository,
                                                                                IEmailTemplateBuilder
emailTemplateBuilder,
                                                                                IMessageBus messageBus,
                                                                                IUserTagRepository
userTagRepository,
                                                                                IMapper mapper)
        {
                _announcementRepository = announcementRepository;
                _emailTemplateBuilder = emailTemplateBuilder;
                _messageBus = messageBus;
                _userTagRepository = userTagRepository;
                _notifier = notifier;
                _mapper = mapper;
        }

        public async Task<Unit> Handle(PublishAnnouncementCommand request, CancellationToken cancellationToken)
        {
                var entity = _mapper.Map<Announcement>(request);

                entity.Timestamp = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds();

                await _announcementRepository.AddAsync(entity, cancellationToken);

                var emailList = await _userTagRepository.GetEmailsByTag(entity.TagId, cancellationToken);

                foreach (var email in emailList)
                {

                        var emailTemplate = _emailTemplateBuilder.CreateNewAnnouncementEmail(email, entity);

                        if (!await _messageBus.PublishAsync(emailTemplate, cancellationToken))
                        {
                                continue;
                        }
                }

                var dto = _mapper.Map<AnnouncementDTO>(entity);

                await _notifier.PublishAsync(dto);

                return Unit.Value;
        }
}

using Application.Contracts.Identity;
using Application.Contracts.Infrastructure;
using Application.Contracts.Persistence;
using Application.Exceptions;
using Application.Features.Announcements.Requests.Commands;

using Domain;

using MediatR;


namespace Application.Features.Announcements.Handlers.Commands;

public class RespondToAnnouncementCommandHandler
        : IRequestHandler<RespondToAnnouncementCommand, Unit>
{
        private readonly IAnnouncementResponseRepository _announcementResponseRepository;
        private readonly IAnnouncementRepository _announcementRepository;
        private readonly IAuthenticationService _authenticationService;
        private readonly IEmailTemplateBuilder _emailTemplateBuilder;
        private readonly IMessageBus _messageBus;

        public RespondToAnnouncementCommandHandler(IAnnouncementResponseRepository announcementResponseRepository,
                                                                                IAnnouncementRepository
announcementRepository,
                                                                                IAuthenticationService
authenticationService,
                                                                                IEmailTemplateBuilder
emailTemplateBuilder,
                                                                                IMessageBus messageBus)
        {
                _announcementResponseRepository = announcementResponseRepository;
                _announcementRepository = announcementRepository;
                _authenticationService = authenticationService;
```

```
                _emailTemplateBuilder = emailTemplateBuilder;
                _messageBus = messageBus;
        }

        public async Task<Unit> Handle(RespondToAnnouncementCommand request, CancellationToken cancellationToken)
        {

                var announcement = await _announcementRepository.GetAsync(request.AnnouncementId, cancellationToken);

                var author = await _authenticationService.GetUserByIdAsync(announcement.AuthorId) ?? throw new UserNotFoundException();

                var responder = await _authenticationService.GetUserByIdAsync(request.ResponderId) ?? throw new UserNotFoundException();

                var email = _emailTemplateBuilder.CreateRespondToAnnouncementEmail(author.Email,
                        responder.Email, request.MessageText, announcement.Title);


                if (!await _messageBus.PublishAsync(email, cancellationToken))
                {
                        throw new EmailNotSendException();
                }

                var response = new AnnouncementResponse
                {
                        AnnouncementId = request.AnnouncementId,
                        ResponderId = request.ResponderId,
                        MessageText = request.MessageText,
                        Timestamp = DateTimeOffset.UtcNow.ToUnixTimeMilliseconds()
                };

                await _announcementResponseRepository.AddAsync(response, cancellationToken);

                return Unit.Value;
        }
}

using Application.Contracts.Persistence;
using Application.Features.Announcements.Requests.Commands;

using AutoMapper;

using Domain;

using MediatR;


namespace Application.Features.Announcements.Handlers.Commands;

public class UpdateAnnouncementCommandHandler : IRequestHandler<UpdateAnnouncementCommand, Unit>
{
        private readonly IAnnouncementRepository _announcementRepository;
        private readonly IMapper _mapper;

        public UpdateAnnouncementCommandHandler(IAnnouncementRepository announcementRepository,
                                                                                                IMapper mapper)
        {
                _mapper = mapper;
                _announcementRepository = announcementRepository;
        }

        public async Task<Unit> Handle(UpdateAnnouncementCommand request, CancellationToken cancellationToken)
        {
                var existingAnnouncement = await _announcementRepository.GetAsync(request.Id, cancellationToken);

                existingAnnouncement.Title = request.Title;
                existingAnnouncement.Content = request.Content;
                existingAnnouncement.Address = request.Address;
                existingAnnouncement.AuthorId = request.AuthorId;
                existingAnnouncement.Status = request.Status;

                await _announcementRepository.UpdateAsync(existingAnnouncement, cancellationToken);

                return Unit.Value;
        }
}

using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;
using AutoMapper;
```

```
using MediatR;

namespace Application.Features.Announcements.Handlers.Queries;

public class GetAllTagsQueryHandler : IRequestHandler<GetAllTagsQuery, IEnumerable<TagDTO>>
{
	private readonly ITagRepository _tagRepository;
	private readonly IMapper _mapper;

	public GetAllTagsQueryHandler(ITagRepository tagRepository, IMapper mapper)
	{
		_tagRepository = tagRepository;
		_mapper = mapper;
	}

	public async Task<IEnumerable<TagDTO>> Handle(GetAllTagsQuery request, CancellationToken cancellationToken)
	{
		return _mapper.Map<IEnumerable<TagDTO>>(await _tagRepository.GetAllAsync(cancellationToken));
	}
}

using Application.Contracts.Identity;
using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;

using AutoMapper;

using MediatR;


namespace Application.Features.Announcements.Handlers.Queries;

public class GetAnnouncementQueryHandler : IRequestHandler<GetAnnouncementQuery, AnnouncementWithInfoDTO>
{
	private readonly IAnnouncementRepository _announcementRepository;
	private readonly IAuthenticationService _authenticationService;
	private readonly IMapper _mapper;

	public GetAnnouncementQueryHandler(IAnnouncementRepository announcementRepository,
                                                                                IAuthenticationService authenticationService,
                                                                                IMapper mapper)
	{
		_announcementRepository = announcementRepository;
		_authenticationService = authenticationService;
		_mapper = mapper;
	}

	public async Task<AnnouncementWithInfoDTO> Handle(GetAnnouncementQuery request, CancellationToken cancellationToken)
	{
		var announcement = await _announcementRepository.GetAnnouncementByIdWithIncludeAsync(request.AnnouncementId,
cancellationToken);

		var result = _mapper.Map<AnnouncementWithInfoDTO>(announcement);

		if (result.AnnouncementsResponses != null)
		{
			foreach (var item in result.AnnouncementsResponses)
			{
				var user = await _authenticationService.GetUserByIdAsync(item.AuthorId);

				item.AuthorEmail = user.Email;
				item.AuthorUserName = user.UserName;
			}
		}

		return result;
	}
}

using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;
using AutoMapper;
using MediatR;

namespace Application.Features.Announcements.Handlers.Queries;

public class GetAnnouncementsByTagsHandler : IRequestHandler<GetAnnouncementsByTagsQuery, IEnumerable<AnnouncementDTO>>
{
```

```
	private readonly IAnnouncementRepository _repo;
	private readonly IMapper _mapper;

	public GetAnnouncementsByTagsHandler(IAnnouncementRepository repo, IMapper mapper)
	{
		_repo = repo;
		_mapper = mapper;
	}

	public async Task<IEnumerable<AnnouncementDTO>> Handle(GetAnnouncementsByTagsQuery request, CancellationToken ct)
	{
		var list = await _repo.GetActiveAnnouncementsByTagsAsync(request.TagIds, ct);
		return _mapper.Map<IEnumerable<AnnouncementDTO>>(list);
	}
}

using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;

using AutoMapper;

using MediatR;


namespace Application.Features.Announcements.Handlers.Queries;

public class GetAnnouncementsByUserQueryHandler : IRequestHandler<GetAnnouncementsByUserQuery, IEnumerable<AnnouncementDTO>>
{
	private readonly IAnnouncementRepository _announcementRepository;
	private readonly IMapper _mapper;

	public GetAnnouncementsByUserQueryHandler(IMapper mapper, IAnnouncementRepository announcementRepository)
	{
		_announcementRepository = announcementRepository;
		_mapper = mapper;
	}

	public async Task<IEnumerable<AnnouncementDTO>> Handle(GetAnnouncementsByUserQuery request, CancellationToken
cancellationToken)
	{
		var announcement = await _announcementRepository.GetAnnouncementsByUserAsync(request.UserId, cancellationToken);

		return _mapper.Map<List<AnnouncementDTO>>(announcement);
	}
}

using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;

using AutoMapper;

using MediatR;


namespace Application.Features.Announcements.Handlers.Queries;

public class GetRecentAnnouncementsQueryHandler: IRequestHandler<GetRecentAnnouncementsQuery, IEnumerable<AnnouncementDTO>>
{
	private readonly IAnnouncementRepository _announcementRepository;
	private readonly IMapper _mapper;

	public GetRecentAnnouncementsQueryHandler(IAnnouncementRepository announcementRepository, IMapper mapper)
	{
		_announcementRepository = announcementRepository;
		_mapper = mapper;
	}

	public async Task<IEnumerable<AnnouncementDTO>> Handle(GetRecentAnnouncementsQuery request, CancellationToken
cancellationToken)
	{
		var list = await _announcementRepository.GetActiveAnnouncementAsync(request.TagId, cancellationToken);

		return _mapper.Map<IEnumerable<AnnouncementDTO>>(list);
	}
}

using Application.Contracts.Identity;
using Application.Contracts.Persistence;
```

```
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;
using AutoMapper;
using MediatR;

namespace Application.Features.Announcements.Handlers.Queries;

public class GetTagsQueryHandler : IRequestHandler<GetTagsQuery, List<TagDTO>>
{
	private readonly IAuthenticationService _authenticationService;
	private readonly ITagRepository _tagRepository;
	private readonly IUserTagRepository _userTagRepository;
	private readonly IMapper _mapper;

	public GetTagsQueryHandler(IAuthenticationService authenticationService,
	                           ITagRepository tagRepository,
	                           IUserTagRepository userTagRepository, IMapper mapper)
	{
		_authenticationService = authenticationService;
		_tagRepository = tagRepository;
		_userTagRepository = userTagRepository;
		_mapper = mapper;
	}

	public async Task<List<TagDTO>> Handle(GetTagsQuery request, CancellationToken cancellationToken)
	{
		var user = await _authenticationService.GetUserByIdAsync(request.UserId);


		var tags = await _userTagRepository.ListByEmailAsync(user.Email, cancellationToken);

		var filtred = tags.Select(x => x.Tag);

		return _mapper.Map<List<TagDTO>>(filtred);
	}
}

using Application.Contracts.Infrastructure;
using Application.DTOs.Announcement.Responses;
using Application.Features.Announcements.Requests.Queries;

using MediatR;


namespace Application.Features.Announcements.Handlers.Queries;

public class SubscribeAnnouncementsQueryHandler: IRequestHandler<SubscribeAnnouncementsQuery, IAsyncEnumerable<AnnouncementDTO>>
{
	private readonly IAnnouncementNotificationPublisher _notifier;

	public SubscribeAnnouncementsQueryHandler(IAnnouncementNotificationPublisher notifier)
	{
		_notifier = notifier;
	}

	public Task<IAsyncEnumerable<AnnouncementDTO>> Handle(SubscribeAnnouncementsQuery req, CancellationToken ct)
	{
		return Task.FromResult(_notifier.Subscribe());
	}
}

using Application.Contracts.Persistence;
using Application.Features.UserRating.Requests.Commands;
using Domain;
using MediatR;

namespace Application.Features.UserRating.Handlers.Commands;

public class CreateUserRatingCommentCommandHandler : IRequestHandler<CreateUserRatingCommentCommand, Unit>
{
	private readonly IUserRatingCommentRepository _userRatingCommentRepository;

	public CreateUserRatingCommentCommandHandler(IUserRatingCommentRepository userRatingCommentRepository)
	{
		_userRatingCommentRepository = userRatingCommentRepository;
	}

	public async Task<Unit> Handle(CreateUserRatingCommentCommand request, CancellationToken cancellationToken)
	{
		await _userRatingCommentRepository.AddAsync(new UserRatingComment() {
```

```
                AuthorId  = request.AuthorId,
                Comment = request.Comment,
                Stars = request.Stars,
                UserId = request.UserId
            },cancellationToken);

            return Unit.Value;
        }
}

using Application.Contracts.Persistence;
using Application.DTOs.Announcement.Responses;
using Application.Features.UserRating.Requests.Queries;
using AutoMapper;
using MediatR;

namespace Application.Features.UserRating.Handlers.Queries;

public class GetUserRatingCommentByUserQueryHandler : IRequestHandler<GetUserRatingCommentByUserQuery,
IEnumerable<UserRatingCommentDTO>>
{
    private readonly IUserRatingCommentRepository _userRatingCommentRepository;
    private readonly IMapper _mapper;

    public GetUserRatingCommentByUserQueryHandler(IUserRatingCommentRepository userRatingCommentRepository, IMapper mapper)
    {
        _userRatingCommentRepository = userRatingCommentRepository;
        _mapper = mapper;
    }
    public async Task<IEnumerable<UserRatingCommentDTO>> Handle(GetUserRatingCommentByUserQuery request, CancellationToken
cancellationToken)
    {

        var dtos = await _userRatingCommentRepository.GetAllByUserIdAsync(request.UserId, cancellationToken);

        return _mapper.Map<IEnumerable<UserRatingCommentDTO>>(dtos);
    }
}

using Application.Contracts.Persistence;
using Application.Features.Stats.Requests.Commands;
using MediatR;
using Microsoft.AspNetCore.Authorization;

namespace Application.Features.Stats.Handlers.Commands
{
    public class CanWriteStarCommentCommandHandler : IRequestHandler<CanWriteStarCommentCommand, bool>
    {
        private readonly IAnnouncementResponseRepository _responseRepository;

        public CanWriteStarCommentCommandHandler(IAnnouncementResponseRepository responseRepository)
        {
            _responseRepository = responseRepository;
        }

        public async Task<bool> Handle(CanWriteStarCommentCommand request, CancellationToken cancellationToken)
        {
            var allResponses = await _responseRepository.GetAllWithAnnouncement(cancellationToken);

            if (allResponses == null)
            {
                return false;
            }

            bool hasResponded = allResponses
                .Any(r => r.ResponderId == request.TargetUserId
                                        && r.Announcement.AuthorId == request.WriterUserId);

            return hasResponded;
        }
    }
}

using Application.Contracts.Identity;
using Application.Contracts.Persistence;
using Application.DTOs.Stats.Responses;
using Application.Features.Stats.Requests.Queries;
using MediatR;
```

```
namespace Application.Features.Stats.Handlers.Queries;

public class GetAdminStatsQueryHandler : IRequestHandler<GetAdminStatsQuery, AdminStatsResponseDTO>
{
	private readonly IAnnouncementRepository _announcementRepo;
	private readonly IAnnouncementResponseRepository _responseRepo;
	private readonly IAuthenticationService _userService;

	public GetAdminStatsQueryHandler(IAnnouncementRepository announcementRepo,
                                         IAnnouncementResponseRepository responseRepo,
                                         IAuthenticationService userService)
	{
		_announcementRepo = announcementRepo;
		_responseRepo = responseRepo;
		_userService = userService;
	}

	public async Task<AdminStatsResponseDTO> Handle(GetAdminStatsQuery request, CancellationToken cancellationToken)
	{

		var announcements = await _announcementRepo.GetAllAsync(cancellationToken);
		var responses = await _responseRepo.GetAllAsync(cancellationToken);
		var users = await _userService.GetAllUsersAsync();

		var announcementDict = announcements.ToDictionary(a => a.Id, a => a.Title);
		var userEmailDict = users.ToDictionary(u => u.Id, u => u.Email);
		var userNameDict = users.ToDictionary(u => u.Id, u => u.UserName);


		var totalUsers = users.Count;
		var confirmedEmails = users.Count(u => u.EmailConfirmed);
		var pctEmailConfirmed = totalUsers == 0
                                        ? 0
                                        : Math.Round((double)confirmedEmails / totalUsers *
100, 2);



		var totalAnnouncements = announcements.Count;
		var announcementsPerDay = announcements
			.GroupBy(a => DateTimeOffset.FromUnixTimeMilliseconds(a.Timestamp).Date)
			.Select(g => new AnnouncementStatDTO
			{
				Date = g.Key.ToString("yyyy-MM-dd"),
				PublishedCount = g.Count()
			})
			.ToList();


		var totalResponses = responses.Count;
		var avgResponsesPerAnnouncement = totalAnnouncements == 0
			? 0
			: Math.Round((double)totalResponses / totalAnnouncements, 2);

		var announcementsWithResponses = responses
			.Select(r => r.AnnouncementId)
			.ToHashSet();

		var announcementsWithoutResponses = announcements
			.Count(a => !announcementsWithResponses.Contains(a.Id));

		var responsesByAnnouncement = responses
			.GroupBy(r => r.AnnouncementId)
			.Select(g => new ResponseStatDTO
			{
				AnnouncementTitle = announcementDict.TryGetValue(g.Key, out var title)
                                                    ? title
                                                    : "<Unknown>",
				ResponseCount = g.Count()
			})
			.ToList();


		var allResponses = responses
			.Select(r => new ResponseDetailDTO
			{
				AnnouncementTitle = announcementDict.TryGetValue(r.AnnouncementId, out var title)
                                                            ? title
                                                            : "<Unknown>",
				ResponderEmail = userEmailDict.TryGetValue(r.ResponderId, out var email)
```

```
                                                                                ? email
                                                                                : "<Unknown>",
                                        RespondedAt = DateTimeOffset
                .FromUnixTimeMilliseconds(r.Timestamp)
                                                                                .ToString("yyyy-MM-ddTHH:mm:ssZ")
                                })
                                .ToList();


                        var topResponders = responses
                                .GroupBy(r => r.ResponderId)
                                .OrderByDescending(g => g.Count())
                                .Take(5)
                                .Select(g => new TopResponderDTO
                                {
                                        ResponderEmail = userEmailDict.TryGetValue(g.Key, out var email)
                                                                                ? email
                                                                                : "<Unknown>",
                                        ResponderUserId = g.Key,
                                        ResponderUserName = userNameDict[g.Key],
                                        ResponseCount = g.Count()
                                })
                                .ToList();

                        var topAuthors = announcements
                                .GroupBy(a => a.AuthorId)
                                .OrderByDescending(g => g.Count())
                                .Take(5)
                                .Select(g => new TopAuthorDTO
                                {
                                        AuthorEmail = userEmailDict.TryGetValue(g.Key, out var email)
                                                                                ? email
                                                                                : "<Unknown>",
                                        AuthorUserId = g.Key,
                                        AuthorUserName = userNameDict[g.Key],
                                        AnnouncementCount = g.Count()
                                })
                                .ToList();


                        return new AdminStatsResponseDTO
                        {
                                TotalUsers = totalUsers,
                                PctEmailConfirmed = pctEmailConfirmed,

                                TotalAnnouncements = totalAnnouncements,
                                AnnouncementsPerDay = announcementsPerDay,

                                TotalResponses = totalResponses,
                                AvgResponsesPerAnnouncement = avgResponsesPerAnnouncement,
                                AnnouncementsWithoutResponses = announcementsWithoutResponses,
                                ResponsesByAnnouncement = responsesByAnnouncement,

                                AllResponses = allResponses,
                                TopResponders = topResponders,
                                TopAnnouncementAuthors = topAuthors
                        };
                }
}

using Application.Contracts.Identity;
using Application.Contracts.Persistence;
using Application.DTOs.Stats.Responses;
using Application.Features.Stats.Requests.Queries;

using MediatR;


namespace Application.Features.Stats.Handlers.Queries;

public class GetMainStatsQueryHandler : IRequestHandler<GetMainStatsQuery, MainStatsDTO>
{
        private readonly IAnnouncementResponseRepository _announcementResponseRepository;
        private readonly IAnnouncementRepository _announcementRepository;
        private readonly IAuthenticationService _userService;

        public GetMainStatsQueryHandler(IAnnouncementResponseRepository announcementResponseRepository,
                                                                                IAnnouncementRepository announcementRepository,
                                                                                IAuthenticationService userService)
```

```
        {
            _announcementResponseRepository = announcementResponseRepository;
            _announcementRepository = announcementRepository;
            _userService = userService;
        }

        public async Task<MainStatsDTO> Handle(GetMainStatsQuery request, CancellationToken cancellationToken)
        {
            var users = await _userService.GetAllUsersAsync();
            var announcements = await _announcementRepository.GetAllAsync(cancellationToken);
            var responses = await _announcementResponseRepository.GetAllAsync(cancellationToken);

            var userEmailDict = users.ToDictionary(u => u.Id, u => u.Email);
            var userNameDict = users.ToDictionary(u => u.Id, u => u.UserName);

            var topResponders = responses
                .GroupBy(r => r.ResponderId)
                .OrderByDescending(g => g.Count())
                .Take(5)
                .Select(g => new TopResponderDTO
                {
                    ResponderEmail = userEmailDict.TryGetValue(g.Key, out var email)
                                                                                    ? email
                                                                                    : "<Unknown>",
                    ResponderUserId = g.Key,
                    ResponderUserName = userNameDict[g.Key],
                    ResponseCount = g.Count()
                })
                .ToList();

            var topAuthors = announcements
                .GroupBy(a => a.AuthorId)
                .OrderByDescending(g => g.Count())
                .Take(5)
                .Select(g => new TopAuthorDTO
                {
                    AuthorEmail = userEmailDict.TryGetValue(g.Key, out var email)
                                                                                    ? email
                                                                                    : "<Unknown>",
                    AuthorUserId = g.Key,
                    AuthorUserName = userNameDict[g.Key],
                    AnnouncementCount = g.Count()
                })
                .ToList();

            return new MainStatsDTO
            {
                TopResponders = topResponders,
                TopAnnouncementAuthors = topAuthors
            };
        }
}
```