Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система для організації збору коштів. Серверна частина для створення ініціатив та проведення зборів
(тема)

Виконав:
здобувач 4 року навчання
групи ПЗПІ-21-1

Михайло ЖМАЙЦЕВ
(Власне ім'я, ПРІЗВИЩЕ)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник ст.викл. кафедри ПІ Віталій ЛЯПОТА
(посада, Власне ім'я, ПРІЗВИЩЕ)

Допускається до захисту
Зав. кафедри _____ Кирило СМЕЛЯКОВ
(підпис) (Власне ім'я, ПРІЗВИЩЕ)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Жмайцеву Михайлу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи Програмна система для організації збору коштів. Серверна частина для створення ініціатив та проведення зборів
Затверджена наказом по університету від 19.05.2025 р. № 397 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 19. 06. 2025
3. Вихідні дані до роботи Розробити серверну частину програмного рішення, що забезпечує управління ініціативами, створення зборів, облік пожертв та інтеграцію з платіжними платформами, використовуючи архітектуру Clean Architecture, мову програмування C# та платформу .NET 8, для зберігання даних — СУБД PostgreSQL, RESTful API з аутентифікацією, обробкою зборів та генерацією статистики, платіжну інтеграцію через Stripe API.
4. Перелік питань, що потрібно опрацювати в роботі: Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, висновки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	05.04.2025	*виконано*
2	Створення специфікації ПЗ	12.04.2025	*виконано*
3	Проектування ПЗ	15.04.2025	*виконано*
4	Розробка ПЗ	16.04.2025	*виконано*
5	Тестування ПЗ	28.04.2025	*виконано*
6	Оформлення пояснювальної записки	30.04.2025	*виконано*
7	Підготовка презентації та доповіді	04.05.2025	*виконано*
8	Попередній захист	17.06.2025	*виконано*
9	Нормоконтроль, рецензування	17.06.2025	*виконано*
10	Здача роботи у електронний архів	17.06.2025	*виконано*
11	Допуск до захисту у зав. кафедри	18.06.2025	*виконано*

Дата видачі завдання «08» «квітня» 2025 р.

Здобувач _____ Михайло ЖМАЙЦЕВ
(підпис)

Керівник роботи _____ ст.викл. Віталій ЛЯПОТА
(підпис) (посада, Власне ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 59 стор., 10 рис., 1 табл., 12 джерел, 3 додатків.

ЗБІР КОШТІВ, ІНІЦІАТИВИ, C#, DONATION API, PAYPAL, STRIPE, .NET.

Об’єкт розробки – програмна система для організації та супроводу збору коштів через тематичні ініціативи.

Мета розробки – створити серверну частину програмного рішення, що забезпечує управління ініціативами, створення зборів, облік пожертв та інтеграцію з платіжними платформами.

Методи розробки – застосовано архітектуру Clean Architecture, мову C# та платформу .NET 7. Для зберігання даних використано СУБД PostgreSQL. Впроваджено RESTful API з аутентифікацією, обробкою зборів та генерацією статистики. Платіжна інтеграція реалізована через Stripe API.

Основні результати роботи:

– розроблено модулі для створення та керування ініціативами (CRUD);

– реалізовано механізм створення зборів із валідацією параметрів (термін, сума, категорія);

– забезпечено інтеграцію з платіжними системами (Stripe, PayPal);

– створено модулі статистики пожертв (загальні, щомісячні, топ-донори);

– розроблено API для інтеграції з зовнішніми платформами (donation export, webhook).

Результатом розробки стала функціональна серверна частина програмної системи, здатна обробляти запити, взаємодіяти з платіжними провайдерами, збирати пожертви в межах окремих ініціатив та формувати звітність у зручному форматі.

ABSTRACT

.NET, C#, DONATION API, PAYPAL, STRIPE, FUNDRAISING, INITIATIVES.

Object of Development – A software system for organizing and supporting fundraising through thematic initiatives.

Purpose of Development – To create the server-side component of a software solution that manages initiatives, creates fundraising campaigns, tracks donations, and integrates with payment platforms.

Development Methods – The system is built using the Clean Architecture approach, the C# programming language, and the .NET 7 platform. PostgreSQL is used as the database management system. A RESTful API has been implemented, featuring authentication, campaign handling, and statistics generation. Payment integration is implemented via the Stripe API.

Main Results of the Work:

– developed modules for creating and managing initiatives (CRUD);

– implemented a mechanism for creating fundraising campaigns with parameter validation (duration, amount, category);

– enabled integration with payment systems (Stripe, PayPal);

– created modules for donation statistics (overall, monthly, top donors);

– developed an API for integration with external platforms (donation export, webhook).

As a result, a fully functional server-side component of the software system was developed, capable of processing requests, interacting with payment providers, collecting donations within individual initiatives, and generating reports in a user-friendly format.

ЗМІСТ

ВСТУП

Актуальність роботи обумовлена широким застосуванням цифрових технологій у сфері збору коштів для соціальних, екологічних та медичних проєктів. Потреба в зручних та ефективних системах для управління ініціативами та донатами є очевидною, оскільки значна частина сучасних благодійних організацій і волонтерських груп активно використовують онлайн-інструменти для взаємодії зі своїми донорами та підтримки постійної комунікації з ними. Розробка програмної системи, що дозволяє централізовано та прозоро організовувати збір коштів, відслідковувати ефективність кампаній та оперативно сповіщати учасників, є актуальною задачею з огляду на стрімке зростання популярності волонтерських та благодійних ініціатив.

Метою кваліфікаційної роботи є створення програмної системи, яка забезпечить автоматизацію та ефективне управління процесами реєстрації користувачів, адміністрування донорів, надсилання сповіщень та контролю активності.

Об’єкт дослідження – процес автоматизації збору коштів для різноманітних ініціатив.

Предмет дослідження – програмна система для реєстрації, керування профілями донорів та автоматизації сповіщень користувачів.

Для реалізації поставлених завдань використовувалися сучасні технології розробки, а саме платформа .NET 8 (мова C#), веб-фреймворк ASP.NET Core, інструменти для роботи з базами даних (Entity Framework Core, PostgreSQL), сервіси авторизації (Identity, OAuth 2.0), сервіси сповіщень (Firebase Cloud Messaging, SMTP), а також мобільні технології React Native.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

У сучасному суспільстві стрімко зростає потреба у відкритих, прозорих та технологічно ефективних механізмах збору коштів. Соціальні ініціативи, громадські організації, благодійні фонди та окремі волонтерські об’єднання все частіше стикаються з необхідністю швидко мобілізувати фінансову підтримку на важливі проєкти: лікування, евакуацію, гуманітарну допомогу, підтримку ЗСУ, екологічні ініціативи, розвиток освітніх чи культурних програм.

Традиційні способи комунікації з донорами, як-от розміщення оголошень у соціальних мережах або використання банківських реквізитів вручну, є обмеженими в масштабованості, безпеці та зручності. Організаторам зборів не вистачає централізованих інструментів для створення та управління кампаніями, відстеження донатів, ведення звітності, а також прозорого зворотного зв’язку з аудиторією.

З іншого боку, користувачі – потенційні донори – бажають бачити достовірність та відкритість ініціатив, мати змогу легко зробити внесок через зручний спосіб (картка, Apple Pay, Google Pay тощо), переглядати статистику по кожному збору, бачити хід кампанії, і при цьому отримувати сповіщення про важливі оновлення.

Особливу актуальність проблема отримує в умовах воєнного часу або гуманітарних криз, коли швидкість збору коштів може мати критичне значення. Тут постає потреба у єдиній цифровій платформі, яка дозволяє:

- швидко створювати тематичні ініціативи із збором коштів;
- інтегрувати платіжні шлюзи для безпечного прийому пожертв;
- централізовано вести статистику по донорах та ініціативах;
- забезпечити рольову модель доступу та адміністрування;
- надсилати користувачам повідомлення про хід зборів.

Таким чином, існує очевидний запит з боку як громадськості, так і організацій – на створення гнучкої, безпечної та масштабованої програмної

системи для організації збору коштів, яка б поєднувала простоту використання з високим рівнем довіри [1].

Збір коштів через цифрові платформи виконує важливу соціальну функцію – він забезпечує оперативну мобілізацію фінансових ресурсів без необхідності проходити складні бюрократичні процедури чи залежати від централізованих фінансових інституцій. На відміну від традиційних джерел фінансування, як-от державні гранти, банківські кредити чи благодійні фонди з обмеженими регламентами, збори коштів базуються на ініціативі самих учасників – як організаторів, так і донорів – і не вимагають тривалого погодження або перевірки на відповідність певним критеріям.

У цьому підході особливу цінність має прозорість: кожна ініціатива може публічно продемонструвати свій поточний стан, суму зібраних коштів, динаміку внесків та досягнуті результати. Це формує довіру, яка вкрай важлива у благодійній або соціально орієнтованій діяльності. Крім того, збори дозволяють залучати кошти не лише від великих донорів, а й від широкої аудиторії звичайних користувачів, створюючи ефект «народного фінансування», де важливий кожен внесок.

У практичному сенсі, такий формат забезпечує високу швидкість запуску нових проєктів і гнучкість у розподілі зібраних ресурсів. Водночас цифрові платформи дають змогу в реальному часі реагувати на запити користувачів, збирати статистику, надсилати сповіщення, адаптувати цілі зборів або розширювати їх масштаб залежно від потреб ситуації. Це не лише технологічна перевага, а й механізм соціальної згуртованості, що дозволяє великій кількості людей відчути свою залученість до важливих змін.

Таким чином, збори коштів, реалізовані у вигляді спеціалізованих цифрових сервісів, виступають не просто засобом фінансування – вони стають інструментом демократичного розподілу відповідальності, підтримки довіри та колективної дії.

Окрім доступності та швидкодії, цифрові збори коштів надають унікальну можливість трансформації соціальної ініціативи в структурований проєкт із чіткою ціллю та відкритим зворотним зв’язком. У той час як інші фінансові механізми

часто працюють як закриті системи, недоступні для пересічного громадянина, платформи зборів відкривають процес і дозволяють кожному долучитися – не лише грошима, а й увагою, підтримкою, поширенням інформації. В результаті формується не просто фінансування певної потреби, а спільнота навколо ідеї.

Це особливо важливо в контексті гуманітарних, екологічних чи військових ініціатив, де значення має не лише результат, а й сама участь. Коли користувач бачить, як його внесок впливає на реальний прогрес, мотивація зростає, формується відчуття причетності, а сам процес збору перетворюється на комунікаційний міст між потребою і відгуком суспільства. Саме така відкритість, підкріплена технологічними інструментами – сповіщеннями, статистикою, інтеграцією з поштовими сервісами чи платіжними шлюзами – робить цифрові збори значно ефективнішими, ніж будь-який офлайн-аналогу або централізована система пожертв.

Також варто зазначити, що збори коштів не обов’язково зводяться лише до благодійності. Це інструмент підтримки освітніх стартапів, незалежних культурних подій, локальних ініціатив у малих громадах – скрізь, де є ідея, але бракує стартових ресурсів. І якщо традиційні канали підтримки часто ігнорують малі або нестандартні проєкти, система зборів дозволяє їм знайти підтримку напряму – без посередників, із довірою та миттєвим результатом.

1.2 Аналіз подібних систем

Системи цифрового фандрейзингу бувають різних типів. Існують як класичні краудфандингові платформи, так і мобільні/банківські додатки з благодійними функціями. Однак наразі саме традиційні канали – колективні скриньки та особисті волонтерські збори – залишаються найпопулярнішими: близько 29% українців роблять внески у стаціонарні скриньки в магазинах і супермаркетах, ще 11% – жертвують через волонтерів на публічних заходах. Натомість лише незначна частина громадян користувалася краудфандингом.З технічної точки зору найпоширенішими є мобільні застосунки банків: наприклад, у Monobank – функція «банки», а в Приват24 – сервіс «конверти», де користувач може створити збір із

описом, ціллю і оновленням прогресу. Також дедалі частіше застосовуються QR-донати і смс-пожертви, а платіжні шлюзи використовуються благодійними фондами для прийому онлайн-платежів.

Попри технічний прогрес, цифрова благодійність має низку суттєвих викликів. Головна з них – низька довіра: багато донорів остерігаються, що їхні кошти підуть не за призначенням, а небагато хто перевіряє прозорість фондів. Як наголошується у дослідженні з UX-дизайну, відсутність чіткої інформації про те, куди потрапляють гроші, й нечітка подача мети проекту миттєво знижують довіру користувача.

Аналогічно, будь-яка «фрикція» у процесі пожертви – зайві кроки, заплутана навігація або неадаптованість під мобільні – часто призводить до відмови від донату. Ще один психологічний бар'єр – перевантаження вибором. Класичні спостереження показують, що коли донору пропонують забагато параметрів (різні суми чи категорії пожертв, багато опцій у формі), він просто «заморожується» й відмовляється діяти. Тому зайва складність форм може нашкодити конверсії.

Інтерфейс додатку Donate24 (див. рис 1.1) показовий: він щоденно пропонує користувачам актуальні збори на потреби ЗСУ, дозволяючи відфільтрувати кампанії за темою, легко перейти до донату чи скопіювати реквізити з одним.

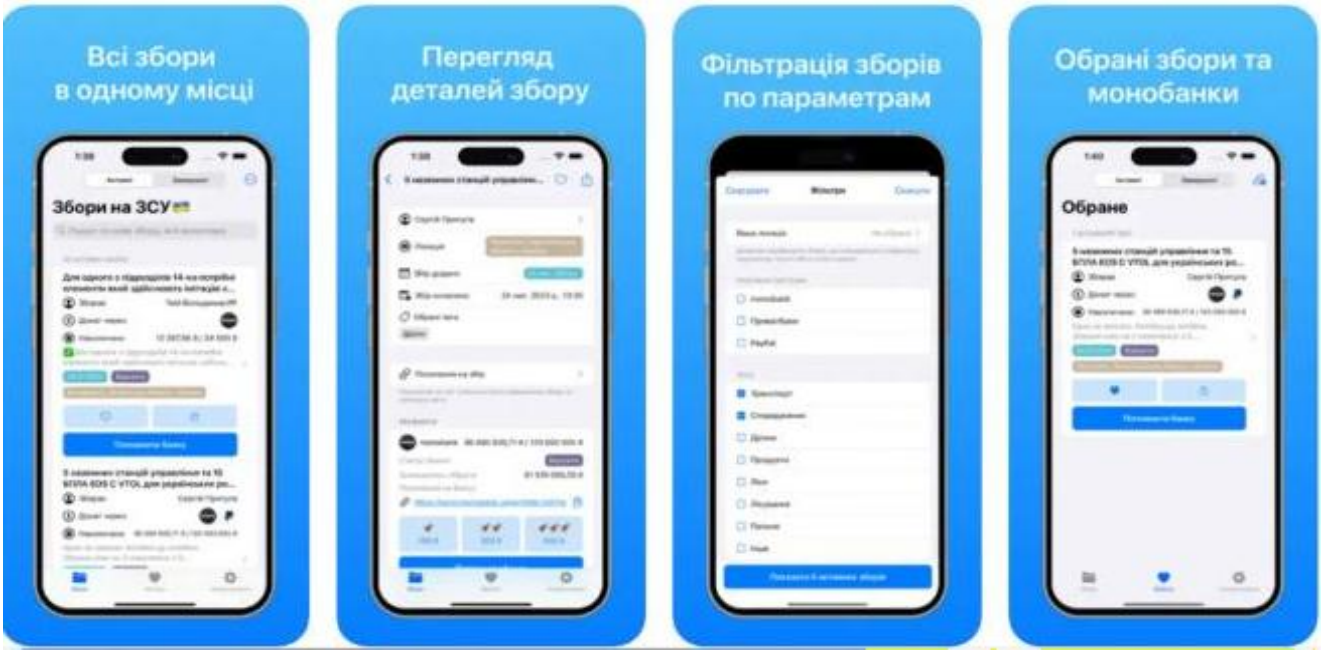

Рисунок 1.1 – Donate24 (за даними [2])

Такі рішення посилюють ефект миттєвої дії – якщо користувач бачить кнопку «Пожертвувати зараз» на початку заклику, це значно підвищує відгук. Також мобільні застосунки використовують push-повідомлення для рекрутингу донорів і мотивації «дати зараз»: вони спонукатимуть зробити внесок своєчасно та нагадують «давати часто». Не менш важливий аспект – соціальний резонанс: коли люди діляться у мережах інформацією про власний донат, це може генерувати нові залучення (соціальне підтвердження). кліком.

Найкращі практики у комунікації та технічному вирішенні формули руху донорів зосереджені на простоті та прозорості. Наприклад, показ реального прогресу кампанії у вигляді progress bar та конкретних числових статистик значно підвищують довіру користувача. Важливо також мінімізувати кількість полів у формі донату та забезпечити «UX у 2 кліки»: досвід показує, що перенесення головного заклику до дії на початок (наприклад, кнопка «Donate» у перших рядках листа) різко збільшує конверсію. Після завершення транзакції корисно одразу відображати екран-підтвердження з візуальною анімацією або коротким повідомленням-подякою, щоб донор відчув «ефект завершеності» (feedback) і розумів, що його пожертва зарахована. Автоматизовані листи чи підтвердження з докладними даними про витрати також допомагають утримати довіру і мотивують повторні внески.

Прикладами вдалих рішень є як українські розробки, так і світові сервіси. Зазначимо, що функція «банок» у мобільному додатку Monobank зібрала понад 1 млрд євро від початку війни, а минулого року через неї пройшло 43,68 млрд грн донатів (Monobank оприлюднив ці дані). Подібний інструмент – «Конверти» у Приват24 – дозволяє групувати збори та прозоро показувати збірникам опис і прогрес. Ще один приклад – сайт VolunteeringUkraine, який публічно відображає всі пожертви на фронтовий проєкт Front Line Kit, що надходять через платіжний шлюз WayForPay; так донори бачать повну прозорість усіх транзакцій. Міжнародні платформи як Patreon чи Donorbox стають стандартними для творчих і навчальних проєктів, а українські рішення активно запозичують ці підходи. Разом вони ілюструють прагнення зробити цифрову благодійність зручною та прозорою, що

важливо для подальшого зростання фондів і залучення довіри суспільства [2].

1.3 Аналіз проблем, з якими стикаються системи збору коштів

Незважаючи на популярність цифрових платформ для збору коштів, більшість існуючих рішень стикається з рядом системних обмежень, які суттєво впливають як на організаторів, так і на донорів. Найбільш помітною проблемою є дефіцит прозорості: користувач часто не має змоги повністю відстежити, як саме витрачаються кошти після завершення збору, або чи дійшли вони взагалі до адресата. Це породжує скепсис, недовіру та зменшує готовність людей повторно підтримувати ініціативи.

Ще однією серйозною складністю є фрагментованість функціональності. Багато платформ або орієнтовані виключно на прийом платежів, або ж мають мінімальний інструментарій для роботи з ініціативами – наприклад, відсутність аналітики, слабка підтримка категоризації зборів чи низький рівень взаємодії з користувачем після пожертви. Через це організатори змушені комбінувати кілька сервісів, що ускладнює облік, звітність і підвищує ризик технічних помилок.

Окрема проблема – обмеженість або дорожнеча інтеграцій з платіжними системами. У багатьох випадках для повноцінного підключення, наприклад, до Stripe чи PayPal, потрібна юридична особа, що ставить бар’єр перед невеликими ініціативами або волонтерами. Деякі безкоштовні рішення мають суттєві функціональні компроміси, інші – надто складні в налаштуванні. Це знижує доступність зборів для широкого кола користувачів, особливо в умовах обмежених технічних знань.

Також слід зазначити, що відсутність гнучкої системи ролей та моделі доступу ускладнює делегування обов’язків в команді проєкту. Часто організатори змушені ділитися одним обліковим записом або покладатися на зовнішні таблиці для контролю, що несе загрози як для безпеки, так і для узгодженості даних.

І нарешті, багато систем мають слабку підтримку автоматизованого інформування донорів про прогрес кампанії. Користувачі не отримують повідомлень про завершення збору, зміни його статусу чи досягнення цілі,

втрачаючи зв’язок з ініціативою. Це не лише знижує емоційну залученість, а й послаблює потенціал повторної участі в майбутньому.

1.4 Постановка задачі

У рамках проєкту розробки програмної системи для організації збору коштів основним завданням нашого розробника є створення логіки обробки ініціатив, механізмів збору пожертв та супровідної аналітики. Зважаючи на суспільну важливість точного відображення цілей, тематики та фінансового стану кожної кампанії, до системи висуваються вимоги щодо підтримки повного життєвого циклу ініціатив – від моменту їх створення до завершення збору.

Завданням стало забезпечити технічну можливість додавання, редагування, фільтрації та пошуку ініціатив, кожна з яких повинна мати чітко окреслену мету, категорію, опис та історію зборів. Користувач повинен мати змогу не тільки ініціювати кампанію, а й отримувати агреговану інформацію про її ефективність – зокрема, суму зібраних коштів, динаміку пожертв, список найбільш активних донорів.

Ключовою складовою реалізації стало також проєктування модуля зборів: система повинна дозволяти створювати фінансові кампанії в межах ініціативи, валідувати їх параметри (термін, сума, валюта), а також надавати прозорий інтерфейс для користувачів, які бажають зробити внесок. Платіжна інтеграція повинна гарантувати безпеку, простоту та гнучкість – для цього обрана модель взаємодії зі сторонніми сервісами на основі API таких платформ, як Stripe або PayPal.

Окремим завданням стала побудова модуля статистики, який дозволяє отримувати актуальні звіти про стан кожного збору, переглядати загальні та щомісячні суми надходжень, а також виявляти топ-донорів. Цей функціонал не лише підвищує прозорість проєкту, а й сприяє зміцненню довіри між організаторами та спільнотою.

Таким чином, основним завданням розробки стало створення повноцінної серверної логіки, яка забезпечить управління ініціативами, запуск зборів, прийом

коштів через інтегровані платіжні шлюзи та генерацію статистичної інформації в межах однієї моделі даних.

Крім окреслених вище вимог, проєкт передбачає також розробку всеосяжного механізму безпеки та відповідності нормативним стандартам. Система повинна забезпечувати наскрізне шифрування фінансових транзакцій, зберігання персональних даних згідно з GDPR та локальними законами про захист інформації, а також повноцінний аудит усіх дій користувачів у вигляді незмінних журналів. Такий підхід не лише мінімізує ризики несанкціонованого доступу чи шахрайства, а й формує надійну основу для зовнішніх перевірок і фінансового комплаєнсу, що особливо важливо для благодійних платформ, які працюють із міжнародними донорами та партнерами.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Функціональні вимоги

Програмна система для організації збору коштів повинна забезпечувати повний цикл взаємодії між користувачами, адміністраторами та платіжними сервісами. Система повинна підтримувати створення, адміністрування та участь у ініціативах зі збору коштів з боку звичайних користувачів, а також надати інструменти для контролю, моніторингу й модерації з боку адміністраторів.

Усі користувачі повинні мати змогу проходити реєстрацію та автентифікацію з використанням електронної пошти або сторонніх провайдерів (OAuth). Для створення зборів повинна передбачатися процедура верифікації з підтвердженням особи через email [3]. Після цього користувач отримує доступ до інтерфейсу створення ініціатив та кампаній збору коштів, які проходять модерацію та публікуються на платформі.

Система повинна підтримувати повноцінний механізм пожертв: від динамічної генерації форм оплат до прийому платежів через інтегровані платіжні сервіси. Після здійснення платежу користувач повинен отримати підтвердження, а дані про транзакцію – бути збережені у базі з можливістю подальшого аналізу.

Для забезпечення прозорості й аналітики система повинна вести облік пожертв, надавати статистику по ініціативам і зборам, а також реалізовувати підписку на сповіщення – як email, так і push-формату – для інформування користувачів про хід зборів, нові кампанії чи досягнення цілей.

З боку адміністраторів передбачається панель управління категоріями ініціатив, контроль за активністю користувачів, доступ до звітів і можливість вручну модерувати окремі ініціативи або користувачів.

Фронтенд мобільного додатку має забезпечувати можливість перегляду та підтримки ініціатив, управління власним профілем, відображення історії донатів та керування підписками. У свою чергу, серверна частина повинна бути побудована з урахуванням масштабованості, безпечного зберігання даних, і забезпечувати швидкий обробіток запитів з мобільного клієнта.

2.2 Нефункціональні залежності

Надійність, продуктивність та безпека є критично важливими аспектами розробки системи збору коштів, оскільки вона має справу з обробкою чутливої інформації та транзакційними даними користувачів. Відсутність належної реалізації цих вимог може призвести до втрати довіри, порушення цілісності даних або прямої фінансової шкоди. Далі розглянуто ці залежності:

– безпека. система повинна гарантувати шифрування переданих даних за допомогою протоколу https. усі персональні дані та фінансові транзакції повинні бути захищені відповідно до принципів gdpr. для доступу до адміністративних функцій має бути реалізовано багаторівневу авторизацію. json web token (jwt) повинен використовуватися для авторизації api-запитів [4];

– масштабованість. архітектура повинна дозволяти горизонтальне масштабування з можливістю обробки збільшеної кількості користувачів та зборів. система має бути підготовлена до розгортання в хмарному середовищі з підтримкою контейнеризації (docker) та оркестрації (наприклад, kubernetes або azure app services);

– доступність. програмне забезпечення повинно бути доступне користувачам 24/7 з мінімальним простоєм. у разі збоїв система має підтримувати механізми відновлення – резервне копіювання бази даних та логів, автоматичне масштабування, моніторинг ресурсів;

– продуктивність. середній час відповіді для більшості api-запитів не повинен перевищувати 500 мс за стандартного навантаження. при цьому допускається використання кешування, асинхронної обробки запитів та оптимізованих індексів у базі даних;

– супроводжуваність та розширюваність. кодова база повинна дотримуватися принципів чистої архітектури (clean architecture), що дозволяє легко вносити зміни та адаптувати систему до нових функціональних вимог. логіка має бути розділена на шари (домен, застосунок, інфраструктура), що дозволяє ізолювати бізнес-логіку від реалізацій;

– міжплатформність. серверна частина повинна працювати незалежно від типу клієнта (веб чи мобільний), надаючи уніфікований restful api. формати обміну – json із дотриманням узгоджених схем.

Таким чином, дотримання наведених нефункціональних вимог гарантує високу якість експлуатації системи в реальних умовах, а також дозволяє розглядати її як стабільну та безпечну платформу для взаємодії великої кількості користувачів.

2.3 Припущення та залежності

У процесі проєктування та реалізації системи було сформульовано низку припущень, які вплинули на вибір архітектурних рішень, технологічного стеку та організацію функціональних модулів. Водночас система має низку зовнішніх та внутрішніх залежностей, від яких безпосередньо залежить її працездатність та стабільність.

Припущення:

– очікується, що користувачі системи матимуть базові навички взаємодії з веб або мобільними застосунками, а також доступ до інтернету. це дозволяє спростити інтерфейс взаємодії, орієнтуючи його на мобільні пристрої та браузери з сучасною підтримкою без потреби в окремих інструкціях чи навчанні;

– припускається, що організатори зборів діятимуть добросовісно та матимуть на меті досягнення реальних цілей. система реалізує базові механізми перевірки, однак не бере на себе відповідальність за зміст ініціатив – за зміст відповідає користувач, що їх публікує;

– також вважається, що інтегровані платіжні сервіси (наприклад, stripe, paypal, liqpay) підтримуватимуть стабільну роботу api та не вимагатимуть значних змін у контрактах взаємодії протягом життєвого циклу системи.

Залежності:

– платформа критично залежить від доступності сторонніх сервісів, зокрема – платіжних шлюзів, поштових серверів (для верифікації та сповіщень), а також хостингової інфраструктури, на якій розгорнуто бекенд. збої у роботі будь-якого з цих елементів можуть призвести до часткової або повної недоступності окремих

функцій;

– система також залежить від стабільної роботи бази даних і коректного застосування міграцій. оскільки в основі лежить централізоване зберігання інформації про користувачів, ініціативи, збори та транзакції – цілісність структури бази даних є критично важливою.

3 АРХІТЕКТУРА ТА ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Проектування UML

На основі визначених функціональних і нефункціональних вимог було проведено моделювання сценаріїв взаємодії користувачів з розроблюваною програмною системою.

Для візуалізації цих сценаріїв було використано UML-діаграму прецедентів (Use Case Diagram), яка дозволяє чітко відобразити ролі основних акторів системи та типові взаємодії між ними.

Ключовими акторами запропонованої системи є:

– зареєстрований користувач здійснює реєстрацію та авторизацію, управляє своїм профілем, підписується на ініціативи та виконує пожертвування;

– адміністратор – виконує керування ініціативами, категоріями, проводить моніторинг активності користувачів та перевірку зборів;

– незареєстрований користувач (гість) – переглядає доступні ініціативи та збори коштів без можливості здійснювати пожертвування чи підписки;

– зовнішній сервіс – взаємодіє з системою через API для отримання інформації про донорів або ініціативи.

Створення Use Case Diagram дозволяє структуровано зобразити взаємозв'язок акторів із ключовими функціями системи та слугує основою для подальшого проєктування та реалізації функціоналу програмної системи (див. рис. 3.1).

Рисунок 3.1 – Use-case діаграма (рисунок виконано самостійно)

На наведеній Use-case діаграмі представлено основні актори та їх взаємодії з програмною платформою для збору коштів. Актори поділені на чотири категорії: Гість, Зареєстрований користувач, Адміністратор та API-клієнт. Гість має змогу лише переглядати та здійснювати пошук ініціатив без авторизації. Зареєстрований користувач виконує широкий спектр дій: від авторизації, налаштування профілю, підписки на ініціативи, отримання сповіщень до безпосереднього створення власних ініціатив та пожертвувань. Адміністратор системи здійснює модерацію контенту, управління категоріями ініціатив, моніторинг активності користувачів, а

також контролює та блокує потенційно шахрайські збори. API-клієнт використовує відкриті інтерфейси для отримання інформації про ініціативи та донорів із зовнішніх застосунків. Диференціація ролей на діаграмі дозволяє чітко визначити функціонал та повноваження кожного типу користувача системи.

Для забезпечення структурованості та зрозумілості внутрішньої організації програмної системи, у процесі проєктування було використано діаграму компонентів. Такий тип UML-діаграм дозволяє візуалізувати основні логічні блоки системи, їхні ролі та взаємозв’язки, що особливо важливо при роботі з розподіленою архітектурою та модульною структурою.

У контексті реалізації функціоналу, що охоплює створення та обробку ініціатив, керування зборами, взаємодію з платіжними сервісами та формування статистичних звітів, діаграма компонентів демонструє, як відповідні модулі взаємодіють між собою, з API-шаром і зовнішніми службами. Зокрема, вона показує, як ініціативи та збори представлені окремими сервісами в логіці застосунку, як саме модуль статистики агрегує дані про пожертви, та яким чином інтеграційні компоненти з'єднують систему з платіжними провайдерами та сторонніми платформами через API або вебхуки.

Завдяки цій діаграмі можна чітко простежити поділ відповідальностей між окремими частинами серверної частини, що критично важливо для підтримуваності коду, масштабування системи та зручності командної розробки. У нашому випадку вона також служить відображенням того, як саме реалізовано взаємозв’язок між користувачем, ініціативами, процесом збору коштів та зовнішніми сервісами, що відповідають за фінансові операції (див. рис.3.2).

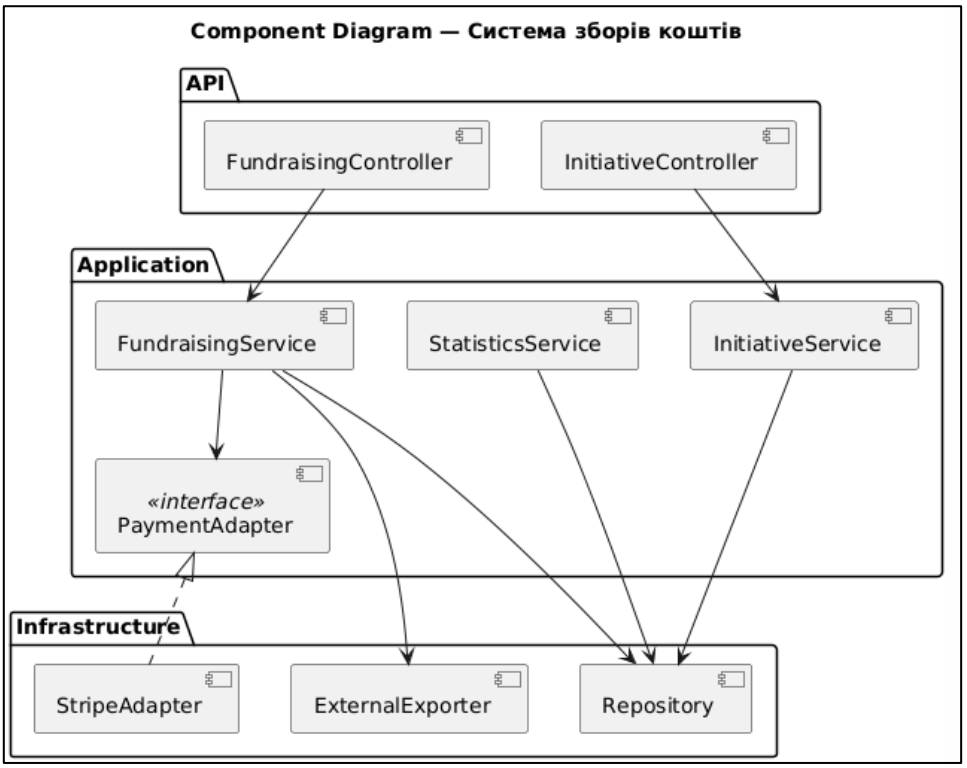

Рисунок 3.2 – Діаграма компонентів (рисунок виконано самостійно)

На діаграмі наведено наступний функціонал:

– контролери (ініціативи, збори) взаємодіють з відповідними сервісами. сервіси звертаються до репозиторію та платіжного адаптера;

– stripeadapter реалізує інтерфейс paymentadapter;

– експорт у зовнішні системи реалізується через externalexporter.

3.2 Проєктування архітектури ПЗ

У системі я використовую Clean-architecture архітектуру.

Завдяки чистій архітектурі доменний і прикладний рівні знаходяться в центрі дизайну, відомого як ядро системи. Бізнес-логіка розміщується в цих двох рівнях, хоча вони містять різні види бізнес-логіки. Вони розглядаються як деталі, і бізнес-рівні не повинні залежати від рівнів презентації та інфраструктури. Замість того, щоб бізнес-логіка залежала від доступу до даних або інших питань інфраструктури, ця залежність інвертується: деталі інфраструктури та реалізації залежать від прикладного рівня.

У процесі розробки серверної частини системи збору коштів було прийнято

рішення використати архітектурний підхід Clean Architecture, оскільки саме він найбільш вдало поєднує вимоги до масштабованості, підтримуваності та розділення відповідальностей. Особливості предметної області – зокрема, робота з критично важливими фінансовими транзакціями, часта взаємодія з зовнішніми сервісами (платіжні API), необхідність адаптації до нових сценаріїв зборів – вимагають від архітектури не лише структурованості, а й здатності до безпечного розширення без порушення існуючої логіки.

Clean Architecture дозволяє зосередити найважливішу частину системи – бізнес-логіку – у центральному шарі, ізольованому від зовнішніх залежностей, таких як веб-фреймворк, база даних чи сторонні інтеграції [5]. У межах нашого функціоналу це означає, що обробка створення ініціатив, запуск зборів, валідація параметрів і генерація статистики реалізовані незалежно від того, який саме контролер ініціює запит або яка саме СУБД використовується для зберігання результатів .

Переваги Clean Architecture у цьому проєкті:

– чітке розділення зон відповідальності: кожен шар системи має власну роль (домен, застосунок, інфраструктура, презентація), що мінімізує зв’язність коду;

– можливість легкого тестування: бізнес-логіка ізольована та легко покривається юніт-тестами без залежності від сторонніх сервісів чи бд;

– гнучкість при інтеграції: реалізація інтерфейсів на стороні інфраструктури дозволяє легко змінювати реалізації арі, сховищ, платіжних шлюзів;

– зручність командної розробки: чітка архітектура дозволяє розподіляти відповідальність між командами, не блокуючи одне одного;

– довгострокова підтримка і масштабування: при додаванні нового функціоналу структура залишається зрозумілою, а зміни не впливають на інші компоненти системи (див. рис. 3.3).

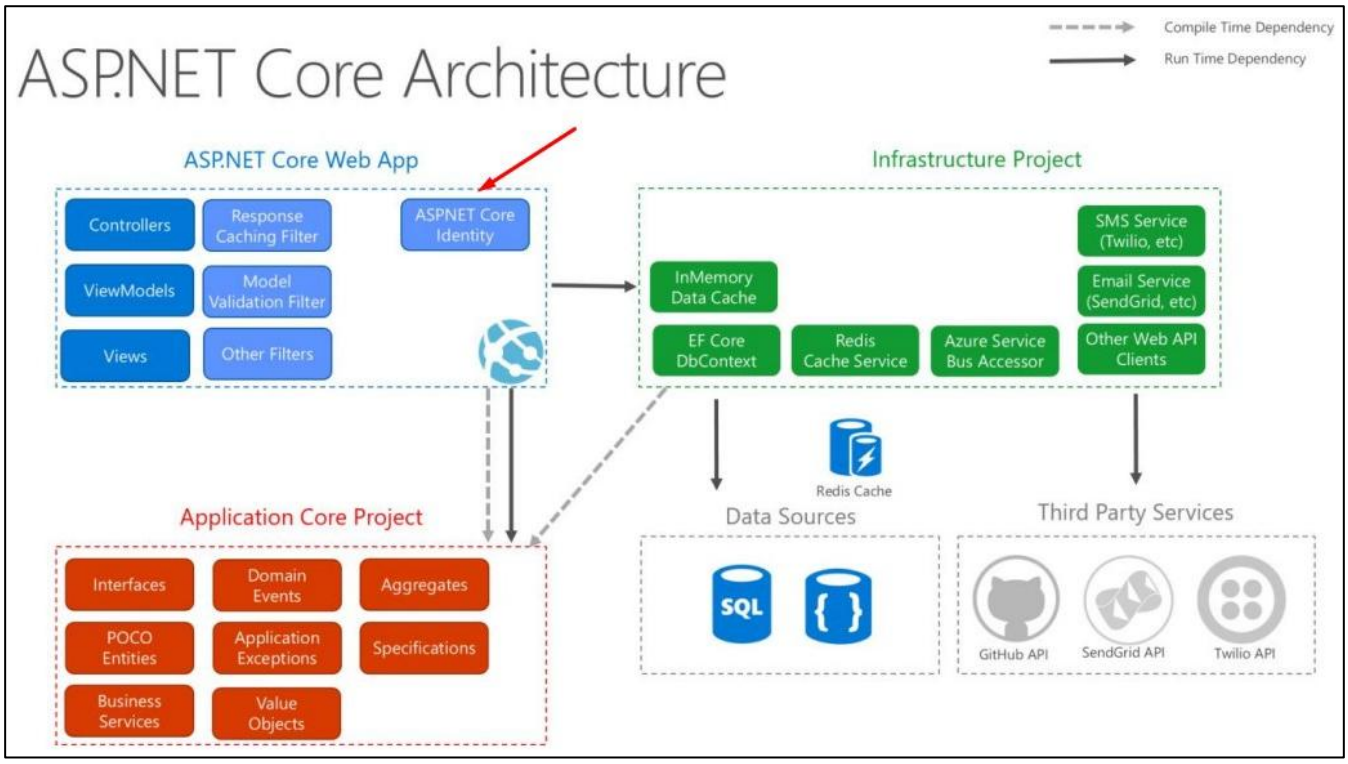

Рисунок 3.3 – Діаграма чистої архітектури (https://devtorium.com/blog/monolithic-vs-microservices-architecture-guide-to-custom-web-development-company-services/)

Такий підхід робить систему стійкою до змін – ми можемо замінити механізм оплати або змінити технологію обміну даними (наприклад, перейти від REST до gRPC) без необхідності переписувати всю бізнес-логіку. Це критично важливо для платформи, яка з часом буде розвиватися, розширювати можливості інтеграції, та, ймовірно, буде мати окремі мобільні чи десктопні клієнти.

3.3 Архітектура зберігання даних

У великому проєкті ніколи не буває достатньо однієї єдиної бази даних, і найчастіше на одному сервері розгортають щонайменше дві – бо розділення середовищ, навантаження та обов’язків дає значні переваги і безпеку.

По-перше, окремі бази даних для різних стадій життєвого циклу (розробка, тестування, staging і продакшн) дозволяють вносити зміни в структуру чи дані у «пісочниці» без будь-якого ризику вплинути на реальних користувачів: міграції, нові фічі, виправлення помилок можна безболісно проганяти на тестовій БД, а в разі невдачі просто скинути її назад, не втратити бізнес-дані.

По-друге, у виробничому середовищі часто рекомендують тримати окрему репліку або read-only базу для звітності та аналітики; навантаження на читаючі запити (звіти, дашборди, BI-інструменти) не впливатиме на швидкість критичних операцій запису – проведення платежів, реєстрація користувачів тощо. По-третє, на рівні безпеки ізольована база резервних копій («standby») або гарячого резерву може миттєво взяти на себе роботу, якщо основний екземпляр впаде: ви отримуєте високодоступний кластер, мінімізуєте RTO. І нарешті, часто буває зручно тримати дві фізично різні схеми: одну для транзакційних даних і одну для логів та аудиту – щоб архіви, «важкі» журнальні товари не роздували основні таблиці, а бек-енд працював стабільно. Власне тому на одному SQL-сервері зазвичай й розгортають дві (або більше) БД – кожна відповідає за свій набір завдань, ізольована по правам, навантаженню та життєвому циклу, що гарантує безпеку, продуктивність і гнучкість у тестуванні та масштабуванні.

Рішення впровадити гетерогенну систему керування реляційними базами даних (RDBMS), до складу якої входять PostgreSQL і Microsoft SQL Server, ґрунтується на комплексному аналізі функціональних та нефункціональних вимог системи, а також нарахунку переваг кожного продукту в контексті архітектури нашого рішення.

Постачальники сучасних RDBMS реалізують різні варіанти MVCC (Multi-Version Concurrency Control) та механізми реплікації. Використовуючи PostgreSQL для основних транзакційних операцій, ми отримуємо високу ступінь масштабованості за рахунок ефективної роботи із паралельними запитами та розподіленими репліками. Microsoft SQL Server, в свою чергу, задовольняє потреби аналітичних навантажень і підтримує оптимізовані інструменти для OLAP-запитів та побудови складних звітів.

Обидві СУБД суворо дотримуються принципів ACID (Atomicity, Consistency, Isolation, Durability). Вибір PostgreSQL обумовлений його стійкістю до write-skew та чудовою підтримкою точного виконання транзакцій у найскладніших сценаріях. MSSQL доповнює це розширеною підтримкою In-Memory OLTP, оптимізацією запитів на рівні планувальника, а також вбудованими засобами захисту даних

(TDE, Always Encrypted).

PostgreSQL забезпечує модульність через механізм extension, що дозволяє під’єднувати додаткові типи індексів (GiST, SP-GiST), JSONB-поля та геопросторові можливості (PostGIS) без підняття версії сервера.

Microsoft SQL Server інтегрований із набором корпоративних рішень (SSIS, SSRS, Azure Data Factory), що спрощує побудову ETL-процесів та централізовану моніторинг-аналітику в середовищі Windows/.NET.

PostgreSQL поширюється за відкритою ліцензією BSD, що мінімізує загальні витрати на придбання та оновлення ПЗ, а також дозволяє вільно модифікувати ядро системи. Для критичних корпоративних модулів, де важлива офіційна підтримка від виробника, використовується MSSQL, що забезпечує прямий контракт із Microsoft, доступ до оновлень без простоїв і SLA на рівні 99,99 %.

Сумісність із .NET та EF Core – обидві СУБД мають нативні провайдери для Entity Framework Core:

Npgsql.EntityFrameworkCore.PostgreSQL для PostgreSQL забезпечує повну підтримку LINQ-конструкцій, схемних міграцій і складних SQL-функцій,

Microsoft.EntityFrameworkCore.SqlServer для MSSQL надає розширення для Table-Valued Functions та синхронного виконання транзакцій у контексті IdentityDbContext.

Комбінування двох СУБД дозволяє розподілити навантаження за контекстами застосунку (транзакційні дані в PostgreSQL, історичні/аналітичні у MSSQL), підвищити загальну доступність сервісів і зменшити час відновлення після можливих збоїв. Така архітектура забезпечує баланс між продуктивністю, надійністю та економічною ефективністю, задовольняючи найсуворіші вимоги корпоративних систем [6].

Також важливо розглянути набір сутностей та їх взаємодія у нашій БД, цю ER-діаграму наведену на рисунку 3.4.

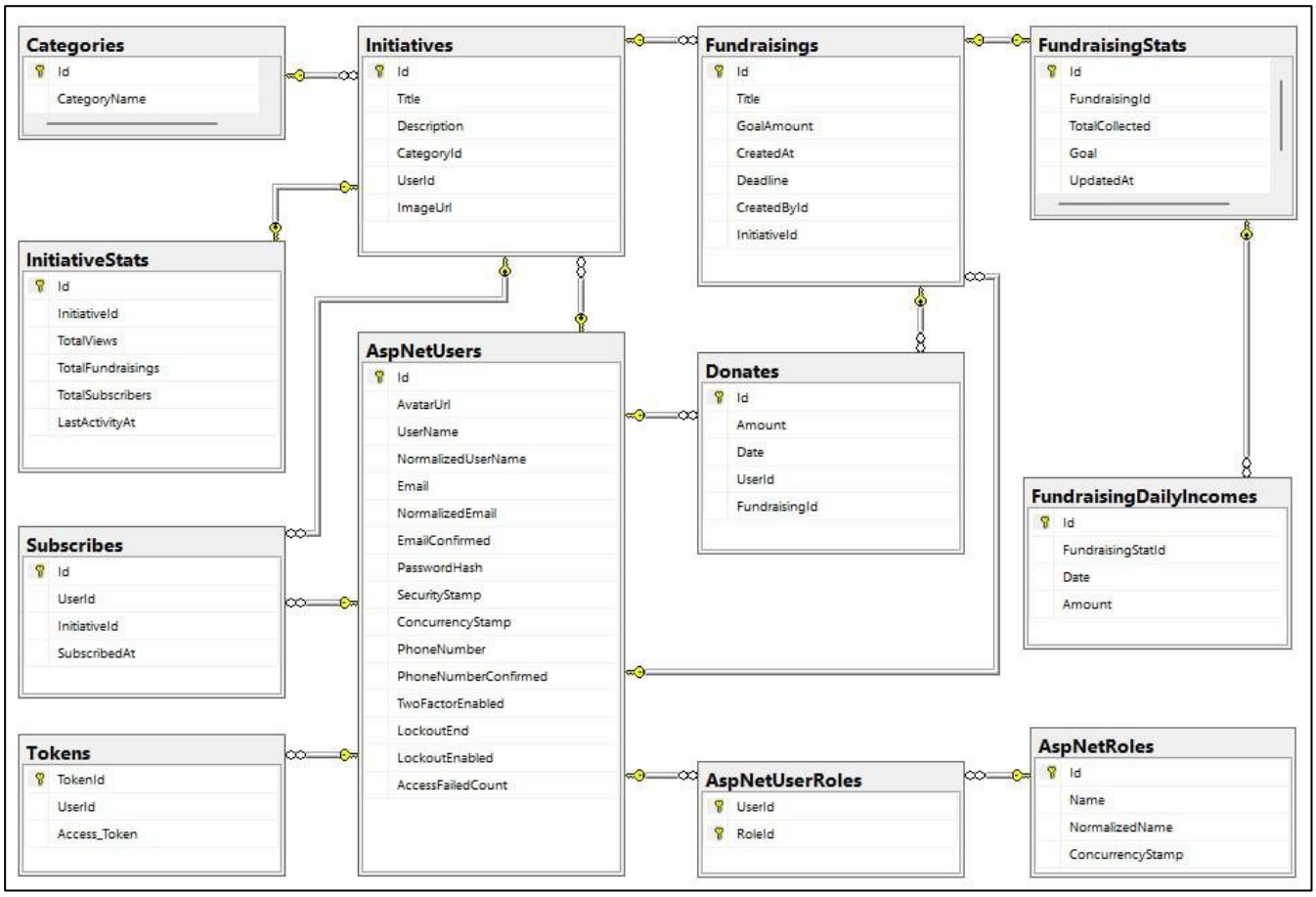

Рисунок 3.4 – ER-діаграма (рисунок виконано самостійно)

ER-діаграма, наведена на рисунку, ілюструє п’ять основних сутностей: User, Initiative, Foundrassing, Donates, Subscribes, а також відображає між ними ключові зв’язки.

User – центральна сутність, що представляє зареєстрованого користувача платформи. Один користувач може створювати або підтримувати кілька ініціатив, підписуватись на них та робити пожертви.

Initiative – описує ініціативу, яка може включати одну або кілька кампаній зі збору коштів. Ініціатива також може бути об’єктом підписки для користувача.

Foundrassing (Fundraising) – сутність, що описує окрему кампанію збору коштів. Кожна така кампанія належить одній ініціативі, має власну суму, термін дії та стан. Це дозволяє створювати кілька послідовних або паралельних зборів у межах однієї теми.

Donates – зв’язувальна сутність між користувачем збором. Вона фіксує факт пожертви, включаючи ідентифікатор донора, суму, дату та пов’язану кампанію.

Subscribes – відображає підписку користувача на певну ініціативу. Це дає змогу надсилати користувачеві сповіщення про нові збори, оновлення кампанії чи досягнення цілі.

У схемі присутні зв’язки «один до багатьох»:

– один User може бути пов’язаний із багатьма Donates;

– один Foundrassing може мати багато записів Donates;

– один Initiative може містити багато Foundrassing;

– один User може підписатись на багато Initiative через Subscribes, і навпаки – що вказує на багато-до-багатьох зв’язок із проміжною таблицею.

Завдяки такій структурі база даних дозволяє зберігати цілісну історію взаємодії користувача із системою, аналізувати активність донорів, генерувати статистику зборів та забезпечувати персоналізоване інформування підписників.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Файлова структура

Далі варто розглянути файлову структуру проекту та проаналізувати структуру та призначення слоїв архітектурної логіки. Файлову структуру наведено на рисунку 4.1.

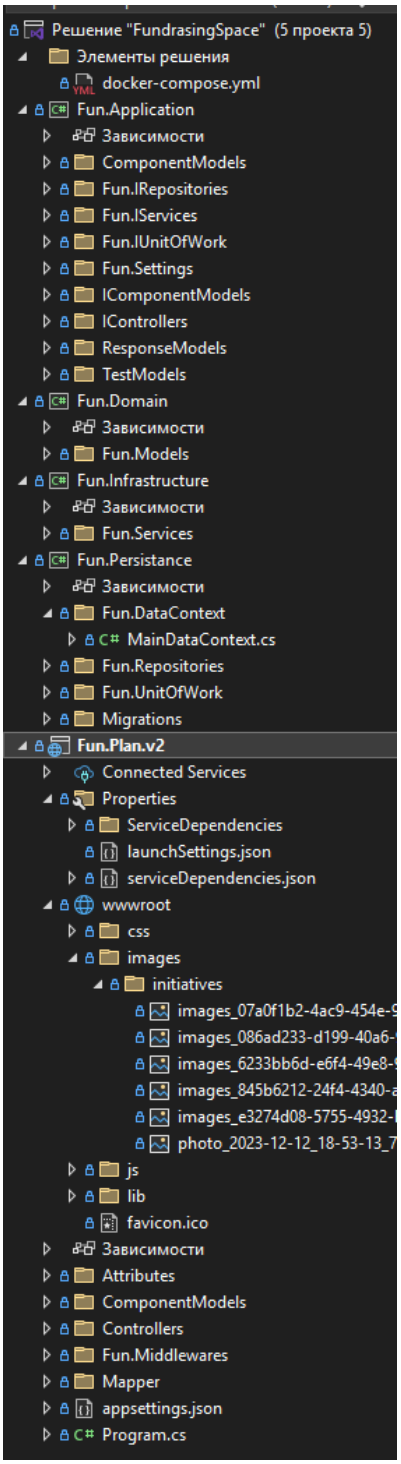

Рисунок 4.1 – Файлова структура системи (рисунок виконано самостійно)

Нижче розглянуто детальніше всі компоненти архітектури серверної частини додатку.

Застосовний шар (Fun.Application):

– componentmodels – моделі передачі даних між компонентами застосунку;

– fun.irepositories – інтерфейси репозиторіїв для ін’єкції залежностей;

– fun.iservices – інтерфейси сервісів бізнес-логіки;

– fun.iunitofwork – інтерфейси для патерну unit of work;

– fun.settings – конфігураційні об’єкти для налаштування системи;

– icomponentmodels – додаткові проміжні моделі або dto;

– controllers – контролери для обробки http-запитів;

– responsemodels – об’єкти для формування відповіді клієнту;

– testmodels – допоміжні моделі для тестування.

– доменний шар (fun.domain):

– fun.models – сутності, що описують основні бізнес-об’єкти системи.

– інфраструктурний шар (fun.infrastructure):

– fun.services – сервіси для взаємодії з зовнішніми системами (api, email, тощо).

Шар доступу до даних (fun.persistance):

– fun.datacontext – контекст бази даних (entity framework);

– fun.repositories – реалізації репозиторіїв для доступу до даних;

– fun.unitofwork – реалізація патерну unit of work;

– migrations – сценарії міграцій для оновлення схеми бд.

Веб-шар (fun.plan.v2):

– wwwroot – публічні статичні ресурси (css, js, зображення);

– attributes – власні атрибути для контролерів або моделей;

– componentmodels – повторне використання моделей інтерфейсу;

– controllers – контролери api для взаємодії з користувачами;

– fun.middlewares – проміжне пз для глобальної обробки запитів;

– mapper – класи для мапінгу між dto та сутностями;

– appsettings.json – конфігураційні файли середовища

– program.cs – точка входу до програми (головний клас запуску).

4.2 Реалізація Postgres + SQL Server

Ми встановили практикою розділення бази даних на дві окремі, тож на рисунку 4.2 наведено приклад зв'язки елементів механізму переходу між базами даних Postgres та SQL Server.

```
{
  "ConnectionStrings": {
    "SqlConnection": "Server=Solis;  Database=FundraisingV2; Trusted_Connection=True; TrustServerCertificate=True; MultipleActiveResultSets=True;"
    "Postgres": "Host=localhost;Port=5432;Database=fundb;Username=postgres;Password=postgres"

  },
  "DatabaseProvider": "SqlConnection",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
builder.Services.AddDbContext<MainDataContext>(options =>
  {
      if (string.Equals(dbProvider, "Postgres", StringComparison.OrdinalIgnoreCase))
      {
          var cs = configuration.GetConnectionString("Postgres");
          options.UseNpgsql(cs);
      }
      else
      {
          var cs = configuration.GetConnectionString("SqlConnection");
          options.UseSqlServer(cs);
      }
});
```

Рисунок 4.2 – Програмний код зв'язку елементів механізму переходу (рисунок виконано самостійно)

У стартовій конфігурації (appsettings.json) я передбачив два рядки підключення – для SQL Server (SqlConnection) і для PostgreSQL (Postgres) – а також параметр «DatabaseProvider», який визначає, яку саме СУБД використовувати в поточному середовищі.

Далі, у момент реєстрації DbContext у контейнері DI я перевіряю значення dbProvider і якщо воно рівне «Postgres», то викликаю options.UseNpgsql(...), передаючи йому рядок підключення до Postgres; інакше – options.UseSqlServer(...) з рядком підключення до MSSQL. Таким чином, щоб переключити всю аплікацію між цими двома СУБД, досить змінити лише значення DatabaseProvider (наприклад, через змінні оточення чи різні appsettings.{Environment}.json), не чіпаючи жодного рядка коду [7].

4.3 Цікаві алгоритми та методи

Надалі варто розглянути pipline для Stripe API, цей елемент є чи не най ключовим у нашій системі, забезпечуючи зручну інтеграцію із платіжною системою нашого програмного застосунку.

У процесі реалізації функціоналу збору коштів критично важливим стало завдання забезпечити надійну, захищену та просту в інтеграції платіжну систему. Після аналізу доступних сервісів було прийнято рішення використовувати Stripe API як основну технологію для прийому пожертв.

Stripe – це сучасна платіжна платформа, яка надає потужні інструменти для обробки онлайн-платежів, зокрема у форматі благодійних внесків, підтримки одноразових і регулярних транзакцій, динамічної обробки валют, а також створення кастомізованих платіжних форм. У нашому випадку це дозволило швидко реалізувати безпечний інтерфейс для донатів, який можна інтегрувати у веб- або мобільний клієнт без збереження платіжних даних на стороні сервера – що особливо важливо з точки зору відповідності вимогам безпеки (PCI-DSS).

Крім того, Stripe API забезпечує зручну обробку webhook-повідомлень, завдяки чому система в режимі реального часу може реагувати на події – наприклад, підтвердження успішного платежу, помилку чи повернення коштів. Це значно підвищує достовірність внутрішньої статистики зборів і дозволяє формувати актуальні звіти без затримок.

Ще одним аргументом на користь Stripe стала його гнучкість і хороша документація – завдяки чому було забезпечено швидку інтеграцію в архітектуру з Clean Architecture, ізоляцію бізнес-логіки від сторонніх залежностей і можливість у майбутньому додати альтернативні платіжні сервіси (наприклад, PayPal або LiqPay) без порушення поточної структури коду.

Таким чином, Stripe API у нашій системі виконує роль надійного шлюзу, що забезпечує прозорий, масштабований і безпечний механізм обробки платежів, критично необхідний для будь-якої платформи зі збору коштів. Далі наведено пайплайн запиту (див. рис. 4.3).

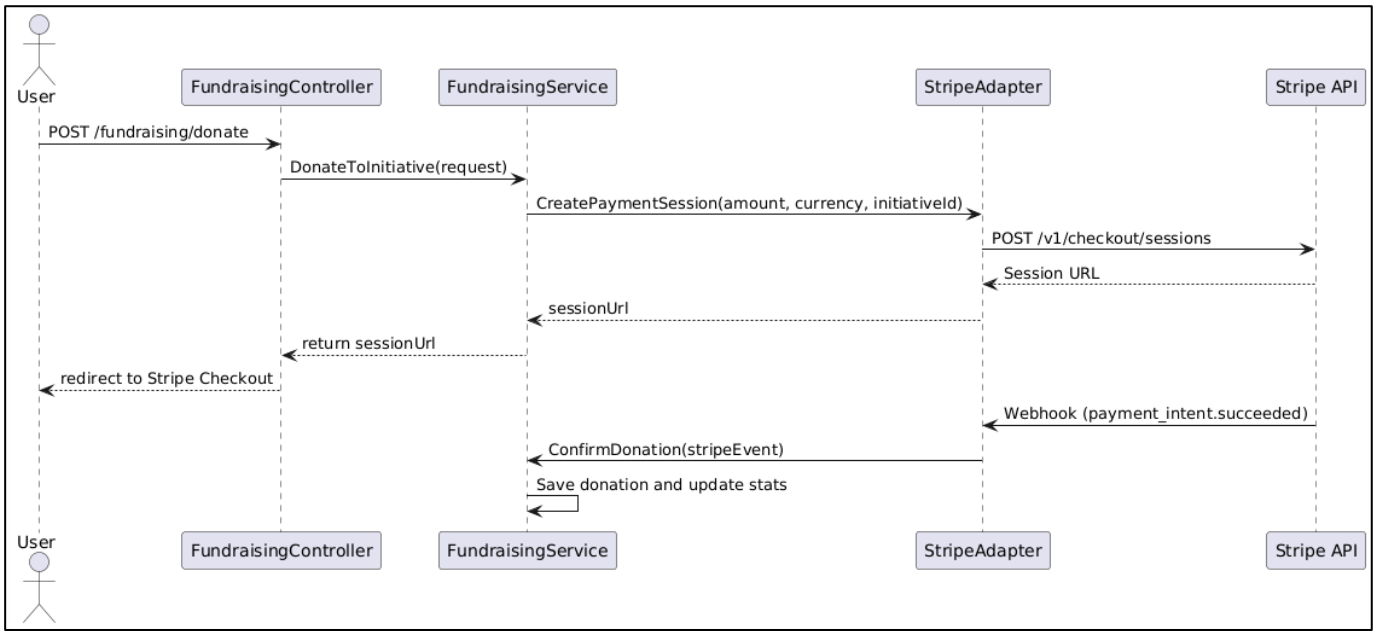

Рисунок 4.3 – Діаграма Stripe pipline (рисунок виконано самостійно)

Ця діаграма показує:

– користувач надсилає запит на пожертву для ініціативи (/fundraising/donate).

– контролер передає запит у сервіс, який викликає stripeadapter для створення сесії оплати;

– stripe повертає посилання на checkout-сесію, і користувач перенаправляється туди;

– після оплати stripe надсилає webhook з підтвердженням транзакції;

– адаптер обробляє webhook і викликає сервіс, який зберігає інформацію про донат і оновлює статистику;

– це чітко показує, як stripe інтегрується у ваш пайплайн і як ваша система реагує на події оплати.

У нашій системі статистичні показники фандрейзингової активності організовані на двох рівнях: агрегатному (сутність FundraisingStat) і щоденному (FundraisingDailyIncome). Кожен виклик методу DonateAsync виконує атомарне створення запису у таблиці Donates (з фіксацією суми та часу) та відразу оновлює відповідний FundraisingStat – інкрементує TotalCollected і виставляє UpdatedAt.

Після цього в таблиці FundraisingDailyIncome знаходять або створюють запис за сьогоднішньою датою: якщо існує – додають суму, інакше – вставляють

новий рядок із базовими полями (FundraisingStatId, Date, Amount). Для отримання статистики передбачено REST-ендпоінт GET /api/fundraisings/{id}/stats, який через EF Core з жадібним завантаженням (Include) одночасно отримує сутність FundraisingStat та всі пов’язані записи FundraisingDailyIncome.

Отримані доменні об’єкти трансформуються у DTO, що містить накопичену суму, часову мітку оновлення та впорядкований за датами масив пар (дата, сума), що забезпечує клієнтам можливість виконувати як макро-, так і мікро-аналіз без додаткової обробки.

Нижче наведено код пайплайну статистики:

```
     public         async         Task<FundraisingStatisticsComponentModel>
GetStatisticsAsync(int fundraisingId)
      {
          var fund = await _ctx.Fundraisings
              .AsNoTracking()
              .Include(f => f.Stat)
                  .ThenInclude(s => s.DailyIncomes)
              .FirstOrDefaultAsync(f => f.Id == fundraisingId);

          if (fund is null)
              throw new KeyNotFoundException($"Fundraising #{fundraisingId}
not found.");
          var stat = fund.Stat;
          return new FundraisingStatisticsComponentModel
          {
              FundraisingId = fund.Id,
              Goal = /*stat.Goal*/fund.GoalAmount,
              TotalCollected = stat.TotalCollected,
              UpdatedAt = stat.UpdatedAt,
              DailyIncomes = stat.DailyIncomes
                  .OrderBy(d => d.Date)
                  .Select(d => new FundraisingDailyIncomeComponentModel
                  {
                      Date = d.Date,
                      Amount = d.Amount
                  })
                  .ToList()
          };
```

Також цікаво дослідити пайплайн роботи донатів автоипзованих юзерів, який повинен оновлювати коректно пов'язані із ними таблиці. Схема наведена на рисунку 4.4.

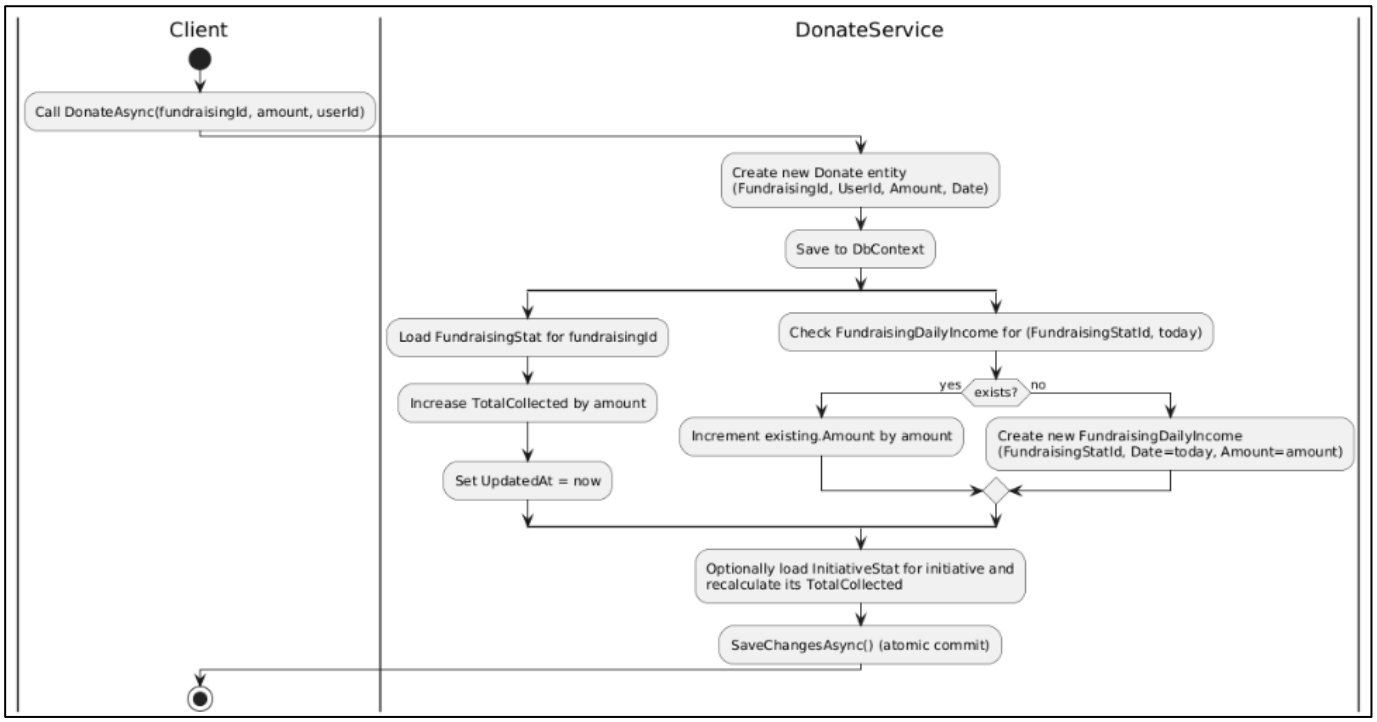

Рисунок 4.4 – Діаграма пайплайну донату (рисунок виконано самостійно)

Нижче наведено код сервісу:

```
public class DonateService: IDonateService
{
    private readonly ICRUDRepository<Fundraising> _repo;
    private readonly MainDataContext _db;
    private readonly IHttpContextAccessor _httpCtx;
    public DonateService(
        ICRUDRepository<Fundraising> repo,
        MainDataContext db,
        IHttpContextAccessor httpCtx)
    {
        _repo = repo;
        _httpCtx = httpCtx;
        _db = db;
    }
    public async Task DonateAsync(int fundraisingId, decimal amount,
int userId)
    {
        var donation = new Donate
        {
            FundraisingId = fundraisingId,
            UserId = userId,
            Amount = amount,
            Date = DateTime.UtcNow
        };
        _db.Donates.Add(donation);
        var stat = await _db.FundraisingStats
            .FirstOrDefaultAsync(s => s.FundraisingId ==
fundraisingId)
            ?? throw new KeyNotFoundException($"FundraisingStat for
#{fundraisingId} not found.");
        stat.TotalCollected += amount;
```

```
            stat.UpdatedAt = DateTime.UtcNow;
            var today = DateTime.UtcNow.Date;
            var daily = await _db.FundraisingDailyIncomes
                .FirstOrDefaultAsync(d => d.FundraisingStatId == stat.Id
&& d.Date == today);
            if (daily == null)
            {
                _db.FundraisingDailyIncomes.Add(new FundraisingDailyIncome
                {
                    FundraisingStatId = stat.Id,
                    Date = today,
                    Amount = amount
                });
            }
            else
            {
                daily.Amount += amount;
            }
            await _db.SaveChangesAsync();
        }
    }
```

У цьому сервісі метод DonateAsync виконує повний цикл обробки донату: спочатку він створює новий об’єкт Donate з переданими fundraisingId, userId та сумою, позначає поточний час, додає його до контексту (_db.Donates.Add). Далі він отримує відповідний запис FundraisingStat (або кидає виняток, якщо такий запис відсутній), нарощує в ньому TotalCollected на суму донату та оновлює поле UpdatedAt.

Після цього сервіс перевіряє, чи існує для цього FundraisingStat запис FundraisingDailyIncome на сьогоднішню дату: якщо ні – створює новий із переданою сумою, інакше збільшує існуюче поле Amount. Нарешті, всі зміни зберігаються в базі єдиним викликом SaveChangesAsync(), що гарантує атомарність операції.

Нижче розглянуто пайплайн:

– надходить виклик методу donateasync(fundraisingid, amount, userid).

– додається новий запис у таблицю donates (fundraisingid, userid, amount, date);

– завантажується fundraisingstat за fundraisingid і збільшується totalcollected, updatedat = зараз;

– шукається запис у fundraisingdailyincomes з fundraisingstatid та датою = сьогодні;

– якщо знайдений – amount += amount;

– якщо не знайдений – створюється новий рядок з fundraisingstatid, date = сьогодні, amount = amount;

– (опційно) оновлюється initiativestat: сумуються totalcollected усіх fundraisingstat для відповідної initiativeid;

– викликається savechangesasync() – всі зміни в donates, fundraisingstat, fundraisingdailyincomes (і initiativestat) зберігаються атомарно.

Метод DonateAsync у сервісі донатів реалізує повністю атомарну обробку надходження пожертвування: спочатку створює новий запис у таблиці Donates із фіксацією суми та UTC-датою, потім через запит до FundraisingStats знаходить агрегований запис для цієї кампанії, збільшує його поле TotalCollected на передану суму та оновлює часову мітку UpdatedAt, після чого перевіряє наявність щоденного запису в FundraisingDailyIncomes для сьогоднішньої дати – у разі відсутності створює новий рядок із відповідною сумою, а якщо такий запис уже є, просто додає до його поля Amount.

Усі зміни накопичуються в контексті Entity Framework Core і зберігаються одним викликом SaveChangesAsync(), що забезпечує консистентність даних між таблицями пожертвувань, агрегованої статистики та щоденних доходів [8].

4.4 Docker Compose

У рамках проєкту було використано Docker Compose – інструмент, який значно спрощує налаштування та розгортання середовища розробки. Docker Compose дозволяє одночасно піднімати кілька сервісів (зокрема базу даних, API, зовнішні інструменти для моніторингу або тестування), визначаючи їхню конфігурацію у єдиному файлі.

Нижче наведено код .yaml файлу для білду бази даних та проведення міграцій [10]:

```
version: "3.8"
services:
  db:
    image: postgres:13
    container_name: fun-postgres
```

```
    restart: always
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: fundb
    ports:
      - "5432:5432"
    volumes:
      - fun-pgdata:/var/lib/postgresql/data
  migrate:
    image: mcr.microsoft.com/dotnet/sdk:6.0
    container_name: fun-migrate
    depends_on:
      - db
    working_dir: /src/Fun.Persistence
    volumes:
      - ./:/src
    entrypoint:
      - bash
      - -lc
      - |
        echo "==> Installing EF Core global tool…"
        dotnet tool install --global dotnet-ef --version 6.0.29 \
          && export PATH="$PATH:/root/.dotnet/tools"
        echo "==> Applying EF migrations…"
        n=1
        max=5
        until [ "$n" -gt "$max" ]; do
          echo "Attempt $n of $max"
          if dotnet ef database update --context MainDataContext; then
            echo "Migrations applied"
            exit 0
          fi
          n=$((n+1))
          sleep 2
        done
        echo "Failed to apply migrations after $max attempts" >&2
        exit 1
volumes:
  fun-pgdata:
```

Цей docker-compose.yml описує два сервіси у версії 3.8:

– db піднімає контейнер з образом postgres:13, задає змінні оточення для користувача, паролю і назви бази, мапить порт 5432 і монтує volume fun-pgdata у /var/lib/postgresql/data для збереження даних поза межами контейнера;

– migrate виконується на базі образу .NET SDK 6.0, залежить від сервісу db і монтує весь локальний репозиторій у /src з робочою текою /src/Fun.Persistence. В точці входу через Bash спочатку глобально встановлює інструмент dotnet-ef версії 6.0.29 і додає його у PATH, а потім у циклі до 5 разів намагається виконати dotnet ef database update --context MainDataContext, чекаючи готовності Postgres та

роблячи паузу між спробами. Якщо міграції застосовані вдалого – контейнер зупиняється з кодом 0, у протилежному разі завершується з помилкою;

– оголошено volume fun-pgdata для збереження даних Postgres.

На рисунку 4.4 наведено інтерфейс докеру на якому видно створені образи та деталі їх роботи.

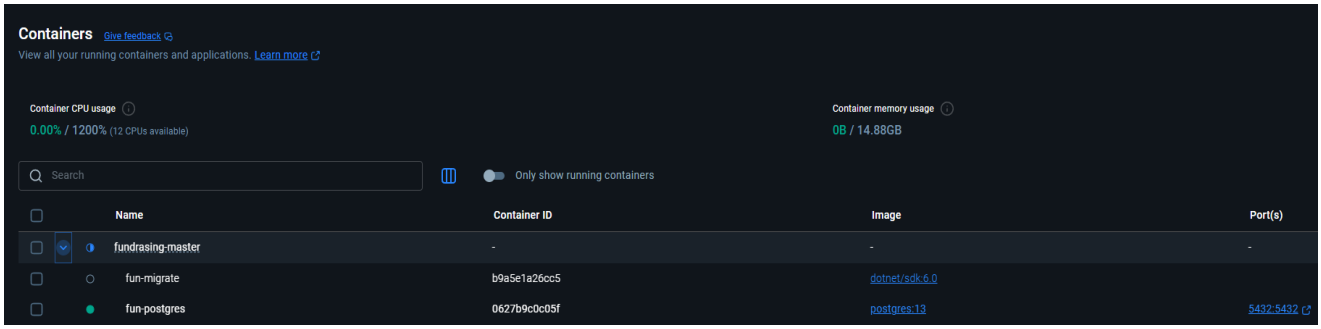

Рисунок 4.4 – Docker Compose (рисунок виконано самостійно)

Це важливо для забезпечення узгодженості середовищ між різними учасниками команди розробки: кожен може запускати однакове середовище з мінімальними налаштуваннями, що знижує ризик помилок через відмінності у локальних конфігураціях. У результаті це не тільки прискорює налаштування нових робочих місць, а й дозволяє швидко відтворювати та тестувати необхідні умови для різних сценаріїв роботи системи, підвищуючи ефективність командної взаємодії та забезпечуючи стабільність програмного рішення на всіх етапах розробки.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У цьому розділі розглянуто декілька сценаріїв тест-кейсів для візуального розуміння процесу розробки та прогресу виконання кожного етапу відповідно технічному завданню. Далі у таблиці 5.1 наведено тестові сценарії.

Таблиця 5.1 – Тест-кейс №1 (таблиця виконана самостійно)

Інформація про тест-кейс			
Ідентифікатор тесту:	Тест-кейс №1		
Власник тесту:	Жмайцев Михайло Олександрович		
Дата створення:	23.05.2025		
Мета тесту:	переконатися, що API для управління ініціативами та збором коштів працює коректно, а інтеграція зі Stripe обробляє платежі та вебхуки бездоганно.		
CRUD & фільтрація ініціатив			
№	Опис випадку	Очікуваний результат	Висновок
1	POST /api/initiatives з валідним JSON (назва, опис, категорія = «medical»)	201 Created, тіло відповіді містить id, status="Published"	Пройдено
2	GET /api/initiatives/{id} для створеної ініціативи	200 OK, поля збігаються з переданими, totalCollected=0	Пройдено
3	PUT /api/initiatives/{id} змінити опис	200 OK, новий опис збережений, updatedAt > createdAt	Пройдено
4	DELETE /api/initiatives/{id}	204 No Content, запис помічено IsDeleted=true	Пройдено
5	GET /api/initiatives?category=medical	Вибірка містить щонайменше одну ініціативу тієї категорії	Пройдено

Продовження таблиці 5.1.

№	Опис випадку	Очікуваний результат	Висновок
6	GET /api/initiatives?search=partOfName	Повертаються ініціативи, назви яких містять підрядок; іншого шуму немає	Пройдено
7	POST без «опису» (required)	400 Bad Request, JSON-помилка валідації Description	Пройдено
8	PUT до неіснуючого id	404 Not Found	Пройдено
9	GET від імені Guest (без JWT) необов'язкові файли	200 OK, але повертаються лише опубліковані ініціативи	Пройдено
10	DELETE із роллю *User* (не Admin)	403 Forbidden	Пройдено
Пожертва через Stripe			
№	Опис випадку	Очікуваний результат	Висновок
1	POST /api/fundraisings/{id}/donate { amount = 50.00 }	200 OK, тіло повертає clientSecret від Stripe API	Пройдено
2	Підтвердити платіж у Stripe Checkout	Stripe показує «Payment successful», користувач редиректиться на successURL	Пройдено
3	Отримати webhook payment_intent.succeeded	Система зберігає новий запис Donates; FundraisingStat.TotalCollected += 50	Пройдено
4	GET /api/fundraisings/{id}/stats	totalCollected відображає +50, dailyIncomes має сьогоднішній запис	Пройдено

Кінець таблиці 5.1.

№	Опис випадку	Очікуваний результат	Висновок
5	Повторити платежі до досягнення цілі, перевірити webhook charge.succeeded	Після перевищення goalAmount кампанія позначена status="Completed"	Пройдено
6	POST donate зі значенням 0 або від’ємною сумою	400 Bad Request, помилка валідації Amount (Range 0.01..)	Пройдено
7	Під час обробки webhook надіслати підроблений підпис	400 Signature Verification Failed; записів у БД не створено	Пройдено
8	Зробити платіж, потім надіслати payment_intent.payment_failed	Статистика не змінюється, запис Donates не створюється	Пройдено
9	GET /api/fundraisings/{invalidId}/stats	404 Not Found	Пройдено
10	Перервати оплату до підтвердження → повернутися на cancelURL	Система не створює донат; стан кампанії незмінний	Пройдено

ВИСНОВОК

У результаті виконання дипломної роботи було розроблено програмну систему для організації збору коштів, яка реалізує повноцінний набір функцій для взаємодії з користувачами, донорами, адміністраторами та зовнішніми сервісами. Основна увага у процесі розробки була приділена модулям реєстрації та авторизації користувачів (включаючи інтеграцію з Google OAuth 2.0), веденню історії донатів, формуванню профілів, а також системі сповіщень – автоматизованих email та push-повідомлень. Додатково створено адміністративний інтерфейс для модерації ініціатив, перевірки зборів та управління категоріями.

Розробка базувалася на архітектурному підході Clean Architecture із чітким розділенням на доменну, прикладну, інфраструктурну та презентаційну частини. Це дозволило досягти високої підтримуваності коду, модульності та легкості масштабування. Було успішно впроваджено технології .NET 8, ASP.NET Core, EF Core, PostgreSQL, Firebase Cloud Messaging, а також Stripe/LiqPay для здійснення онлайн-платежів.

У ході роботи також були виявлені та вирішені типові проблеми, притаманні цифровим платформам для збору коштів: низька прозорість, відсутність гнучкої системи верифікації користувачів, складність інтеграції платіжних інструментів і брак налаштованих механізмів сповіщення. Запропоноване рішення дозволяє ефективно взаємодіяти з великою кількістю донорів, підтримує масштабування ініціатив та забезпечує прозоре адміністрування.

Проведене тестування підтвердило стабільність функціонування основних компонентів системи. Запропоноване рішення відповідає сучасним вимогам до безпечних, зручних та гнучких систем цифрового фандрейзингу, є готовою до практичного застосування у рамках реальних соціальних або благодійних ініціатив.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз предметної області [Електронний ресурс]. – Режим доступу: https://www.clearchannel.co.uk/new-to-out-of-home/billboard-advertising/how-effective-is-billboard-advertising. – Дата звернення: 01.04.2025.

2. Donate24 – AppStore: https://apps.apple.com/us/app/donate24-%D0%B7%D0%B1%D0%BE%D1%80%D0%B8-%D0%BF%D1%96%D0%B4%D1%82%D1%80%D0%B8%D0%BC%D0%BA%D0%B0/id6450211519 - Дата звернення: 18.06.2025

3. Аналіз існуючих рішень [Електронний ресурс]. – Режим доступу: https://antaris.in.ua/ua. – Дата звернення: 05.05.2025.

4. OAuth 2.0 Authorization Code Flow [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-auth-code-flow. – Дата звернення: 07.05.2025.

5. Configure JWT Bearer authentication in ASP.NET Core [Електронний ресурс]. – Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/configure-jwt-bearer-authentication?view=aspnetcore-9.0. – Дата звернення: 08.05.2025.

6. Clean Architecture in .NET [Електронний ресурс]. – Режим доступу: https://code-maze.com/dotnet-clean-architecture/. – Дата звернення: 11.05.2025.

7. Maymala, Jayadevan. PostgreSQL for Data Architects [Електронний ресурс] / Jayadevan Maymala. — Birmingham : Packt Publishing, 2015. — 272 с. — ISBN 978-1-78328-860-1. — Режим доступу: https://www.amazon.com/PostgreSQL-Data-Architects-Jayadevan-Maymala/dp/1783288604. — Дата звернення: 14.05.2025.

8. Falatiuk H., Shirokopetleva M., Dudar Z. Investigation of Architecture and Technology Stack for e-Archive System // 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 8–11 October 2019, Kyiv, Ukraine. – 2019. – Режим доступу: https://doi.org/10.1109/picst47496.2019.9061407. – Дата звернення: 17.05.2025.

9. Stoyan Y. G., Smelyakov S. V. An approach to the problems of routing optimization in the regions of intricate shape // Information Processing Letters. – 1981. –

Vol. 13, no. 1. – P. 39–43. – Режим доступу: https://doi.org/10.1016/0020-0190(81)90148-4. – Дата звернення: 18.05.2025.

10. Netanel Basal. Docker Compose: Up and Running [Електронний ресурс]. — Sebastopol, CA: O'Reilly Media, 2023. — 220 с. — ISBN 978-1-492-04186-4. — Режим доступу: https://www.oreilly.com/library/view/docker-compose-up/9781492041864/. — Дата звернення: 21.05.2025.

11. YAML Tutorial: A Complete Language Guide with Examples [Електронний ресурс]. – Режим доступу: https://spacelift.io/blog/yaml. – Дата звернення: 24.05.2025.

12. GitHub репозиторій [Електронний ресурс]. – Режим доступу: https://github.com/NureZhmaitsevMykhailo/2025_B_PI_PZPI-21-1_Zhmaitsev_M_O. – Дата звернення: 16.06.2025.