НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри

Комп'ютерних наук

_____Голуб Б.Л.

_“ ”_____2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему

««Програмне забезпечення інформаційної системи підтримки волонтерської діяльності»

Спеціальність 121 «Інженерія програмного забезпечення»

Гарант освітньої програми

К.т.н., доцент _____ /Вайганг Г.О. /
(Науковий ступень та вчене звання) (підпис) (ПІБ)

Керівник бакалаврської кваліфікаційної роботи

_____ _____ Бородкін Г. О.
(Науковий ступень та вчене звання) (підпис) (ПІБ)

Виконав _____ Мороз Я.В.
(підпис) (ПІБ)

КИЇВ-2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ

Завідувач кафедри
комп'ютерних наук

_____/ Голуб Б.Л., доцент, к.т.н /

підпис

"__"_____2025 р.

ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

студенту Морозу Ярославу Вікторовичу

Спеціальність 121 «Інженерія програмного забезпечення»

Тема бакалаврської кваліфікаційної роботи: Програмне забезпечення інформаційної системи підтримки волонтерської діяльності

Затверджена наказом ректора НУБіП України від 16.12.2024 № 2249 "С"

Термін подання завершеної роботи на кафедру _____

(рік, місяць, число)

Вихідні дані до роботи: опис програмного забезпечення

Перелік питань , які потрібно розробити:

Аналіз проблемної області, вибір та обґрунтування засобів для розробки системи, проектування інформаційної системи.

Дата видачі завдання "_____"_____20___ р.

Керівник бакалаврської кваліфікаційної роботи

___к.т.н., доцент_____ _____ ___Бородкін Г. О. _

(науковий ступінь та вчене звання) (підпис) (ПІБ)

Завдання прийняв до виконання _ _____ Мороз Я.В.

(підпис) (ПІБ студента

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API — Application Programming Interface, інтерфейс програмування застосунків, що забезпечує взаємодію між клієнтом і сервером.

DB — Database, база даних.

ER — Entity-Relationship, модель для побудови логічної структури бази даних на основі сутностей та їх зв’язків.

Express.js — фреймворк для Node.js, який використовується для створення REST API та серверної логіки.

JSON — JavaScript Object Notation, легкий формат обміну даними, що часто використовується в мережевій взаємодії між клієнтом і сервером.

MVC — Model-View-Controller, архітектурний шаблон для побудови структурованих додатків із розділенням бізнес-логіки, уявлення та контролю.

ORM — Object-Relational Mapping, підхід до взаємодії з базою даних через об’єктно-орієнтовані моделі.

PK — Primary Key, первинний ключ у таблиці бази даних.

REST — Representational State Transfer, архітектурний стиль для побудови API, що працюють через протокол HTTP.

SPA — Single Page Application, односторінковий вебзастосунок, що динамічно оновлює вміст без повного перезавантаження сторінки.

SQL — Structured Query Language, мова структурованих запитів для роботи з реляційними базами даних.

TLS/SSL — Transport Layer Security / Secure Sockets Layer, протоколи шифрування даних при передачі через інтернет.

UML — Unified Modeling Language, уніфікована мова моделювання, що використовується для візуалізації структури та поведінки програмних систем.

3НФ — Третя нормальна форма, рівень нормалізації бази даних, що усуває транзитивні залежності між неключовими атрибутами.

ІС — інформаційна система.

ПЗ — програмне забезпечення.

СУБД — система управління базами даних.

UI — User Interface, інтерфейс користувача.

UX — User Experience, досвід взаємодії користувача з системою

ВСТУП

Сучасні суспільні виклики, пов’язані з гуманітарними кризами, надзвичайними ситуаціями та соціальними проблемами, зумовлюють активний розвиток волонтерської діяльності як важливої форми громадської участі. Проте ефективна організація волонтерських ініціатив потребує технологічної підтримки, зокрема впровадження сучасного програмного забезпечення, що здатне автоматизувати процеси управління, комунікації та звітності.

Актуальність розробки програмного додатку полягає в тому, що існуючі платформи для волонтерства часто не відповідають повному спектру вимог: вони або надто складні для використання, або не підтримують розмежування ролей і модерацію проектів. У зв’язку з цим виникає потреба у створенні простої, адаптивної та безпечної інформаційної системи, яка дозволить волонтерам швидко приєднуватися до ініціатив координаторам — ефективно керувати завданнями, а донорам — контролювати цільове використання ресурсів.

Метою розробки є створення вебзастосунку, що дозволяє оптимізувати процеси взаємодії між волонтерами, координаторами, донорами та модераторами. Особлива увага приділяється безпеці доступу, зручності користування, прозорості управління проектами та гнучкому налаштуванню статусів (збір коштів, у процесі, завершено) і модерації (на розгляді, затверджено, відхилено), які відображаються вибірково залежно від ролі користувача.

Методи та технології, які застосовувалися при розробці, включають:

- **ReactJS** — для реалізації фронтенд-частини у вигляді односторінкового застосунку;
- **Node.js та Express.js** — для побудови REST API та реалізації бізнес-логіки;
- **PostgreSQL** — як система керування базами даних;

- **UML** — для моделювання структури та поведінки системи (використано діаграми прецедентів, класів, розгортання, ER-діаграми);
- Принципи MVC-архітектури та рольової авторизації.

Апробація програмного забезпечення здійснювалась у навчальному процесі в межах виконання лабораторних і курсових робіт, а також шляхом тестування у змодельованих сценаріях взаємодії різних ролей. Елементи системи розглядалися на студентських консультаціях та використовувались у рамках освітнього дослідження з теми діджиталізації волонтерського сектору.

Структура пояснювальної записки охоплює 45 сторінок друкованого тексту, містить 23 використаних джерела та 3 додатки. Дипломна робота складається з чотирьох розділів:

- У **першому розділі** розглянуто предметну область, виявлено актуальні потреби, сформульовано функціональні та нефункціональні вимоги до системи.
- У **другому розділі** побудовано модель інформаційного забезпечення, створено структуру бази даних та її взаємозв’язки.
- У **третьому розділі** здійснено вибір технологій, реалізовано програмну архітектуру клієнтської та серверної частин, а також логіку авторизації та ролей.
- У **четвертому розділі** проведено тестування, описано результати перевірки працездатності системи, наведено вимоги до середовища впровадження.

Результатом виконаної роботи стало створення працездатного прототипу системи, який повністю відповідає поставленим вимогам і демонструє можливість подальшого практичного використання.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Волонтерська діяльність є важливою складовою соціального розвитку, яка об’єднує небайдужих громадян навколо спільної мети — допомогти іншим. У сучасному світі потреба в волонтерському управлінні значно зросла.

Із залучення великої кількості учасників, координація завдань, організація фінансування, модерація проектів, звітність і безперебійна комунікація — це лише частина функцій, які мають бути реалізовані у межах однієї системи. Для досягнення продуктивності роботи ці процеси потребують технологічної підтримки.

Предметна область даної дипломної роботи охоплює процеси, пов’язані з автоматизацією управління волонтерською діяльністю за допомогою веб-орієнтованої інформаційної системи. Така система повинна забезпечувати підтримку повного життєвого циклу волонтерського проекту: від його створення координатором, перевірки модератором, до залучення волонтерів і донорів, виконання завдань і формування звітності.

Основні учасники системи (актори):

- **Волонтер** — виконує завдання в межах проекту, подає заявки, надає звіти.
- **Координатор** — створює та управляє проектами, розподіляє волонтерів, формує звітність.
- **Модератор** — перевіряє проекти, схвалює або відхиляє їх, контролює дотримання правил сайту.
- **Донор** — надає фінансову допомогу, переглядає звіти про використання ресурсів.
- **Неавторизований користувач** — має доступ лише до загальної інформації про затверджені проекти.

Інформаційна система повинна забезпечувати:

- Реєстрацію та автентифікацію користувачів з різними ролями.

- Можливість подати заявку на участь у проекті.
- Відображення лише дозволених статусів проекту залежно від ролі.
- Модерацію проектів та обмеження публічності до моменту затвердження.
- Фіксацію благодійних внесків і прозору звітність.

Таким чином, предметна область охоплює технологічну підтримку волонтерських процесів, їхню структуризацію, контроль, координацію та аналітику з урахуванням специфіки взаємодії учасників.

1.2 Огляд інформаційних джерел та існуючих рішень

У процесі створення інформаційної системи підтримки волонтерської діяльності важливим етапом є вивчення наявних рішень та платформ, що вже функціонують у цій сфері. Такий огляд дозволяє краще зрозуміти потреби користувачів, виявити сильні та слабкі сторони існуючих сервісів, уникнути дублювання функцій та визначити переваги майбутнього продукту.

Платформа dobro

Опис: dobro— це перша та найбільша в Україні благодійна онлайн-платформа, яка забезпечує кращу ефективність для збору коштів на соціальні та гуманітарні проекти. Платформа була створена у 2011 році як "Українська біржа благодійності" та у 2020 році перезапущена під новим брендом dobro.ua. Місією платформи є розвиток культури системної благодійності, заснованої на прозорості, відповідальності та залученні громадян.

Функціональні можливості:

- Публікування перевірених благодійних проектів з детальним описом та фінансовими цілями.
- Можливість для користувачів підтримувати проекти фінансово через зручні платіжні системи.

- Прозора звітність про зібрані та витрачені кошти для кожного проекту.
- Створення персональних кампаній для збору коштів на підтримку обраних проектів.
- Інтеграція з Viber та Telegram-ботами для зручності користувачів.

Переваги:

- Високий рівень довіри завдяки прозорості та звітності.
- Широкий спектр проектів у різних сферах: здоров'я, освіта, соціальна допомога тощо.
- Зручний та зрозумілий інтерфейс для користувачів.

Недоліки:

- Відсутність функціоналу для координації волонтерської діяльності, такого як управління завданнями або комунікація між волонтерами та координаторами.

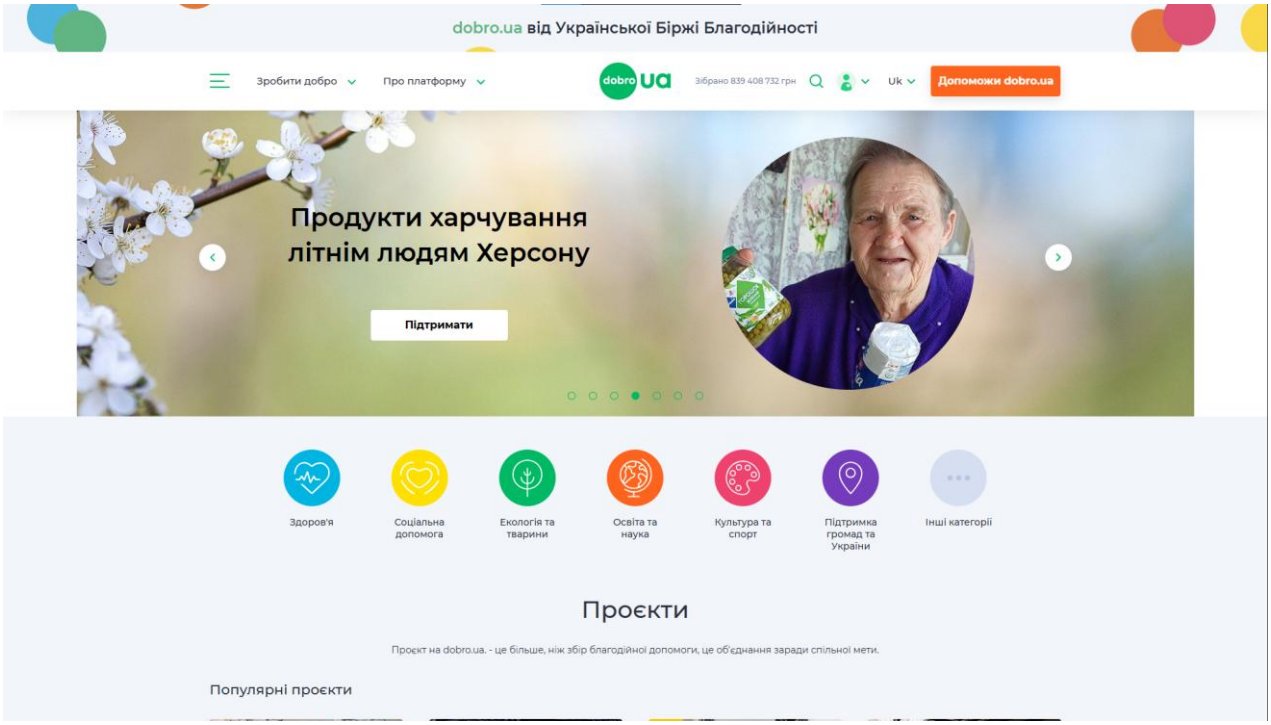

Рис. 1.1 — Головна сторінка платформи dobro

Платформа volonter

Опис: volonter — це українська онлайн-платформа, яка сприяє взаємодії між волонтерами та організаціями. Метою платформи є об'єднання громадських ініціатив та волонтерів для реалізації спільних проектів.

Функціональні можливості:

- Публікація завдань та проектів, до яких можуть долучитися волонтери.
- Можливість для волонтерів подавати заявки на участь у проектах.
- Комунікація між організаторами та волонтерами через платформу.

Переваги:

- Сприяння волонтерській діяльності та залученню нових учасників.
- Простий механізм взаємодії між волонтерами та організаціями.

Недоліки:

- Обмежене сприяння щодо управління проектами та завданнями.
- Відсутність інтеграції з фінансовими інструментами для збору коштів.
- Немає системи модерації або перевірки проектів.

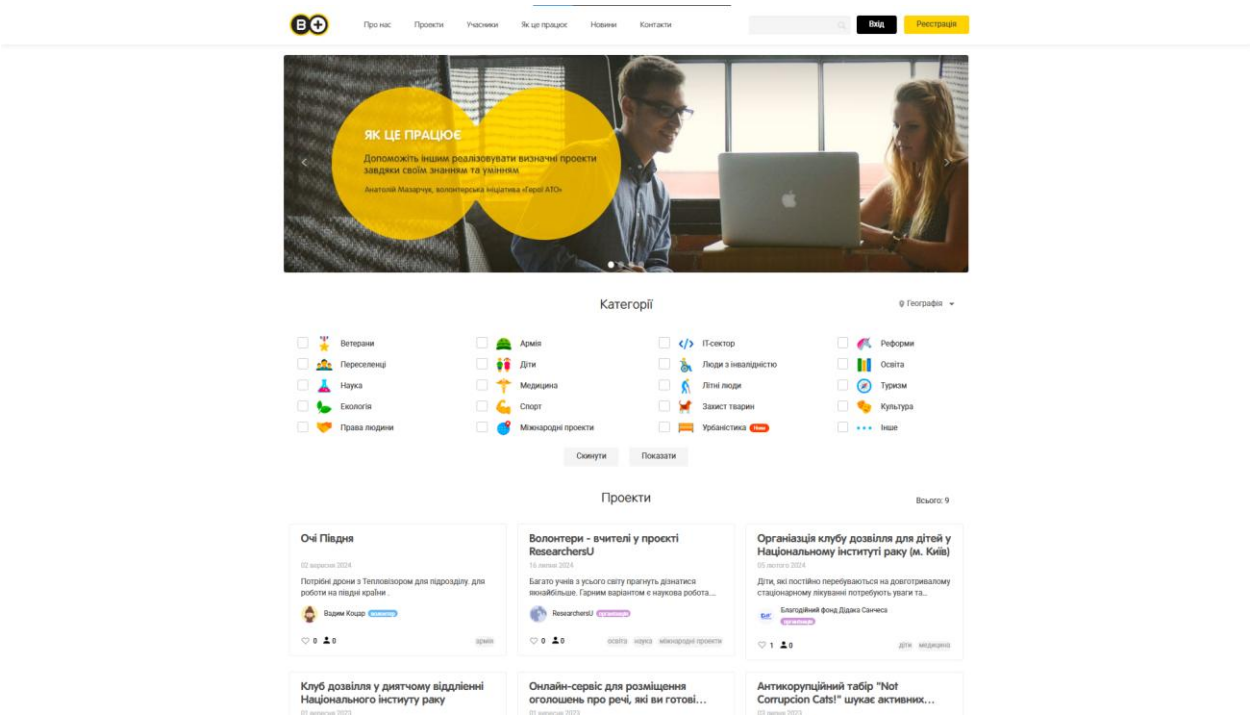

Рис.1.2 — Головна сторінка платформи volonter

Платформа spilno

Опис: spilno — це платформа, створена за ініціативи ЮНІСЕФ, яка об'єднує громадські ініціативи, волонтерські проекти та благодійні організації. Метою платформи є сприяння розвитку громадянського суспільства та підтримка соціальних проектів.

Функціональні можливості:

- Публікація соціальних та волонтерських проектів.
- Можливість для користувачів долучатися до проектів та ініціатив.
- Інформаційна підтримка та освітні ресурси для волонтерів та організацій.

Переваги:

- Підтримка міжнародної організації ЮНІСЕФ, що забезпечує довіру до платформи.
- Широкий спектр проектів у різних соціальних сферах.

Недоліки:

- Відсутність сприяння розвитку платформи для управління волонтерськими завданнями та проектами.
- Обмежені можливості для збору коштів безпосередньо через платформу.
- Немає системи модерації або перевірки проектів.

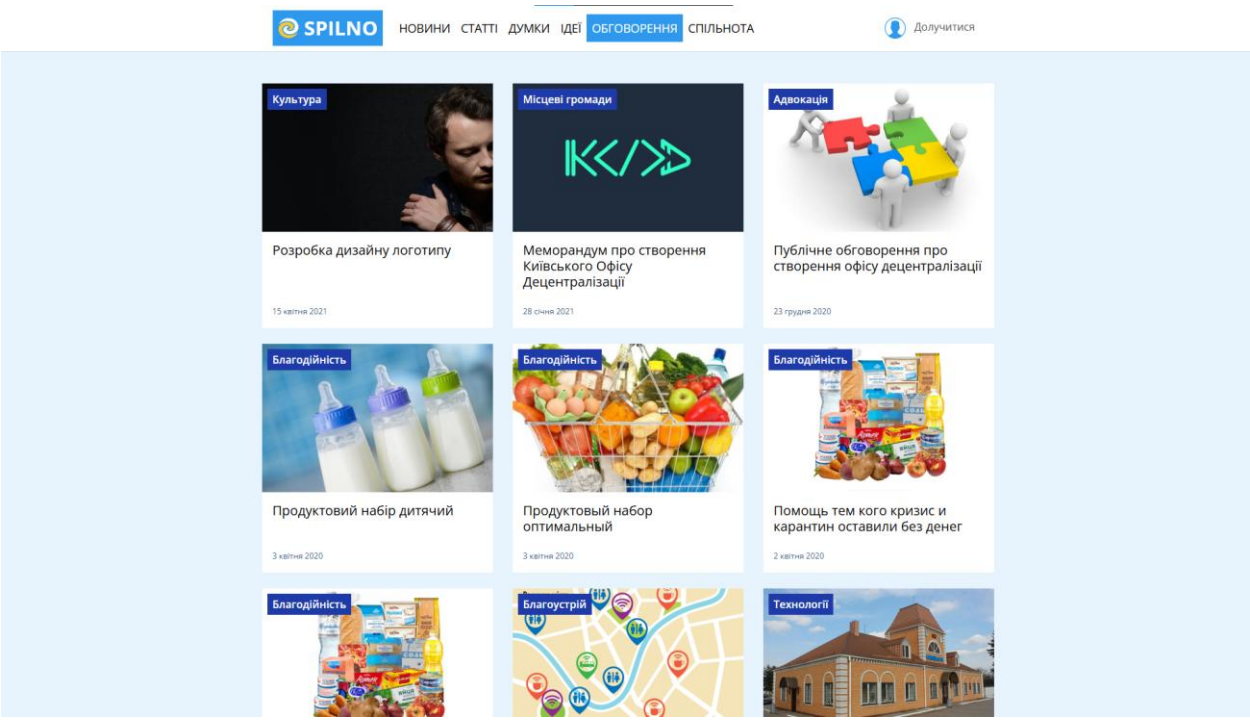

Рис.1.3 — Головна сторінка платформи spilno

Критерій	**Dobro**	**spilno**	**volonter.**
Система пожертв	+	-	-
Ролі користувачів	-	частково	-
Подання заявки волонтером	-	+	+
я завдань	-	-	частково
Модерація проектів	-	-	-

Інтеграція з фінансовими звітами	-	-	-
Прозорість етапів реалізації проекту	частково	частково	-
Підтримка мобільних пристроїв	+	+	+

Жодна з проаналізованих платформ не об'єднує в одне рішення всі ключові можливості: керування волонтерами, модерацію проектів, пожертви, статуси виконання, розмежування доступу за ролями, подання звітів та фінансову прозорість. Таким чином, запропонована система має інноваційний підхід до структурування волонтерської діяльності та заповнює наявні функціональні прогалини.

1.3 Аналіз вимог до програмної системи

Після аналізу предметної області виникає необхідність чітко визначити вимоги до інформаційної системи підтримки волонтерської діяльності. Формання вимог дозволяє структуровано описати очікувану функціональність, гарантувати коректну взаємодію між користувачами системи та її компонентами, а також уникнути суперечностей у реалізації бізнес-логіки.

Функціональні вимоги:

1. **Реєстрація та автентифікація користувачів** — система повинна дозволяти користувачам створювати облікові записи з роллю волонтера, донора або надіслати запит на роль координатора (який затверджується вручну).
2. **Рольовий доступ до функціоналу** — інтерфейс і доступні дії мають залежати від ролі:

- Волонтер — подання заявок, виконання завдань, подання звітів.

- Донор — перегляд проектів, внесення пожертв, перегляд фінансової звітності.
- Координатор — створення проектів і завдань, перевірка звітів, управління волонтерами.
- Модератор — перевірка та затвердження або відхилення проектів.

3. **Подання заявки на участь у проекті** — волонтер може надіслати заявку з мотиваційним повідомленням, яка потім розглядається координатором.
4. **Створення проектів** — координатор створює проекти, додає опис, мету, цільову суму пожертв, додає завдання та вимоги до волонтерів.
5. **Модерація проектів** — після створення проект потрапляє у статус "на розгляді" і доступний лише модератору. Тільки після затвердження проект стає публічним.
6. **Додання завдань до проекту** — координатор формує завдання з дедлайнами, інструкціями та вимогами.
7. **Подання звітів** — волонтер після виконання завдання надсилає текстовий опис і фото як підтвердження.
8. **Фінансування проекту** — донори мають змогу підтримати проект, переглянути ціль і відслідковувати витрати.
9. **Формування фінансових звітів** — координатор формує звіти про витрачені кошти, які доступні донорам.
10. **Контроль видимості статусів** — статуси виконання (збір коштів, у процесі, завершено) доступні всім, а статуси модерації (на розгляді, затверджено, відхилено) — лише координатору.

Нефункціональні вимоги

Продуктивність

Система повинна швидко реагувати на дії користувачів: час завантаження сторінок не повинен перевищувати 2 секунд при нормальному навантаженні та 5 секунд при піковому навантаженні.

Підтриманя одночасної роботи не менше 500 активних користувачів без значної втрати продуктивності.

Масштабованість

Система має бути здатна до масштабування без втрати продуктивності із збільшенням кількості користувачів і подій.

Підтримка горизонтального масштабування для обробки збільшених обсягів даних та одночасних підключень.

Надійність

Система має бути доступною не менше ніж 99,9% часу (мінімальний простій — не більше 8 годин на рік).

Підтримка функцій автоматичного резервного копіювання даних для запобігання втраті інформації у разі збоїв.

Безпека

Захист даних користувачів за допомогою шифрування (SSL/TLS) під час передачі та зберігання.

Вимога до використання двофакторної аутентифікації для адміністративних користувачів і доступу до конфіденційних даних.

Система має відповідати вимогам щодо захисту персональних даних (наприклад, GDPR).

Зручність використання (Юзабіліті)

Інтерфейс має бути зрозумілим та доступним для користувачів з мінімальними технічними навичками.

Забезпечення адаптивного дизайну для підтримки різних пристроїв (ПК, планшети, мобільні телефони).

Портативність

Система має бути доступною через різні операційні системи та веб-браузери, зокрема Windows, macOS, Linux, Android, iOS.

Підтримка основних браузерів (Google Chrome, Firefox, Safari, Microsoft Edge) без необхідності встановлення додаткових плагінів.

Технічні вимоги:

1. **Фронтенд:** ReactJS (односторінковий застосунок, SPA)
2. **Бекенд:** Node.js + Express.js (REST API, логіка маршрутизації, обробка запитів)
3. **База даних:** PostgreSQL (реляційна, з таблицями users, projects, tasks, applications, reports, donations тощо)
4. **Зберігання файлів:** локально або через стороннє сховище (залежно від хостингу)
5. **Інструменти моделювання:** UML (діаграми класів, розгортання, прецедентів), ER-діаграма
6. **Середовище виконання:** сучасний веб-браузер + серверна платформа з Node.js

Ці вимоги стали основою для моделювання структури та розробки функціональних модулів системи. Вони забезпечують цілісність, безпечність і практичну придатність реалізованого програмного рішення.

1.4 Моделювання предметної області

Моделювання предметної області — один із фундаментальних етапів у процесі створення інформаційної системи, оскільки саме він дозволяє ще до початку розробки сформувати точне уявлення про майбутню програмну систему. У межах цього етапу здійснюється абстрагування від реального світу, виділення ключових об’єктів і процесів, які потім трансформуються у структуровані моделі.

Для інформаційної системної підтримки волонтерської діяльності моделювання дозволяє:

- узагальнити ролі користувачів та їхні дії;
- зрозуміти потоки даних, які циркулюють у системі;
- описати внутрішню логіку обробки заявок, завдань, звітів та фінансування;
- забезпечити основу для створення архітектури, бази даних, API та UI;
- зменшити ризики помилок під час проектування й програмування.

У межах даного дослідження використовувалась **мова UML (Unified Modeling Language)** — сучасний стандарт для візуального представлення структурних і поведінкових аспектів програмного забезпечення. UML дає змогу представити логіку системи з різних точок зору: зовнішньої взаємодії (через прецеденти), внутрішніх процесів (через послідовність) та логіки виконання дій (через активність).

Основні переваги моделювання з UML:

- Уніфікована візуальна мова, зрозуміла як розробникам, так і аналітикам.
- Можливість розділити складну систему на окремі частини.
- Чітке визначення залежностей між об’єктами та користувачами.
- Гнучкість і масштабованість моделі при зміні вимог.

Побудовані діаграми

У ході моделювання предметної області було створено три ключові UML-діаграми, які охоплюють різні аспекти майбутньої системи.

Діаграма прецедентів

Ця діаграма демонструє взаємодію між основними акторами системи (волонтер, координатор, донор, модератор, неавторизований користувач) та функціональні сценарії, які вони виконують. Діаграма була побудована з метою ідентифікації основних дій у системі та ролей, відповідальних за їх ініціювання.

Перед побудовою діаграми був проаналізований опис предметної області для визначення акторів які злагоджено працюють в ній.

Актори:

1. **Волонтер** – користувач, який реєструється в системі, обирає завдання відповідно до своїх навичок, отримує сповіщення та звітує про виконану роботу.
2. **Координатор** – адміністратор волонтерської організації, який створює та керує завданнями, розподіляє волонтерів, контролює виконання та формує звіти.
3. **Модератор —** користувач, який контролює якість та відповідність контенту в системі, перевіряє створені проекти, блокує некоректних учасників (волонтерів або координаторів), а також публікує або відхиляє проекти, що подаються до реалізації
4. **Донор** – особа або організація, що надає фінансову чи матеріальну підтримку волонтерським ініціативам та має доступ до звітів про використання ресурсів.

Діаграма прецедентів UML знаходиться в Додатку А

Діаграма дозволяє побачити загальну структуру взаємодії без занурення в технічні деталі.

Прецеденти включають:

- для волонтера: реєстрація, подання заявки, подання звіту;
- для координатора: створення проекту, призначення завдань, перевірка звітів, формування фінансового звіту;
- для донора: перегляд проектів, надсилання пожертви;
- для модератора: схвалення або відхилення проекту;
- для неавторизованого користувача: перегляд затверджених проектів.

Прецеденти акторів:

Волонтер:

1. **Зареєструватися в системі** – створення облікового запису волонтера для подальшої участі в проектах.
2. **Перегляд доступних проектів** – отримання списку актуальних проектів, до яких волонтер може долучитися.
3. **Подання заявки на участь у проекті** – подання запиту на участь у вибраному проекті з можливістю додати коментар або мотивацію.
4. **Виконання завдання** – здійснення волонтерської діяльності відповідно до отриманого завдання.
5. **Отримання інструкцій** – доступ до опису дій, які потрібно виконати в межах завдання (розширювальний прецедент для "Виконання завдання").
6. **Запит на повторне виконання** – повторна участь у завданні, якщо попередня спроба була невдалою або запитана повторна дія.
7. **Завантаження фото доказів виконання** – надання візуальних підтверджень про виконану роботу (наприклад, фото до/після).
8. **Звітування про виконану роботу** – оформлення підсумкового звіту щодо виконаного завдання, з описом і прикріпленнями.

9. **Отримання зворотного зв’язку** – можливість переглянути коментарі, оцінки або рішення координатора щодо звіту чи виконаної роботи.

Координатор:

1. **Створити нове завдання** – ініціація нового завдання в межах існуючого або нового проекту.
2. **Визначити вимоги до волонтера** – встановлення критеріїв (навички, досвід), які повинен мати волонтер для участі.
3. **Розподілити волонтерів по завданнях** – визначення, хто з волонтерів буде виконувати які саме завдання в проекті.
4. **Перевірити виконання завдань** – перевірка звітів волонтерів та оцінка якості їхньої роботи.
5. **Запит на коригування роботи** – повернення звіту на доопрацювання або уточнення (розширення до перевірки).
6. **Формувати звіт** – створення зведених звітів за результатами діяльності волонтерів або реалізації проектів.
7. **Створити проект** – ініціаціювання нового волонтерського проекту з описом мети, потреб та завдань.
8. **Перевірити подані заявки** – ознайомлення з заявками волонтерів, які подались на проекти.
9. **Прийняти заявки** – схвалення участі волонтера у конкретному завданні/проекті.
10. **Відхилити заявки** – відмова у прийнятті заявки, якщо волонтер не відповідає вимогам.
11. **Переглянути звіти волонтерів** – аналіз поданих звітів від волонтерів за виконаними завданнями.

Модератор:

1. **Переглянути опубліковані проекти** – перегляд усіх проектів, що були схвалені до публікації.

2. **Перевірити проект** – детальне ознайомлення з інформацією про новий проект, перевірка на відповідність правил платформи.

3. **Опублікувати проект** – схвалення та відкриття доступу до проекту для волонтерів.

4. **Відхилити проект** – відмова в публікації через невідповідність критеріям або некоректність даних.

5. **Заблокувати волонтера / координатора** – обмеження доступу користувачеві у разі порушення правил платформи.

Донор:

1. **Переглянути доступні проекти** – ознайомлення з актуальними волонтерськими ініціативами, до яких можна долучитися фінансово.

2. **Надати фінансову допомогу** – здійснення пожертви або передача матеріальних ресурсів на підтримку певного проекту.

3. **Переглянути звітність про використання коштів** – доступ до фінансових звітів та інформації про те, як були використані донорські кошти.

Діаграма послідовності

Діаграма послідовності зосереджена на хронологічному порядку повідомлень між об’єктами під час виконання конкретного сценарію — наприклад, подання заявки волонтером і її обробка координатором.

Учасники сценарію:

- **Волонтер** — подає заявку, виконує завдання, звітує.
- **Координатор** — створює та супроводжує проекти, призначає завдання, перевіряє звіти.
- **Модератор** — переглядає нові проекти, приймає рішення про публікацію.
- **Донор** — надає фінансову допомогу та переглядає фінансові звіти.

Ключові взаємодії, зображені на діаграмі:

1. **Створення проекту координатором**:

- Координатор формує проект і надсилає його на перевірку.
- Модератор переглядає проект і:
  - Якщо він **відповідає правилам**, публікує його.
  - Якщо **порушує правила**, відхиляє.
  - Якщо координатор порушує правила систематично — його обліковий запис блокується (опція OPT).

2. **Реєстрація волонтера та подання заявки**:

- Волонтер реєструється.
- Вибирає проект і надсилає заявку на участь.
- Координатор переглядає заявку:
  - Якщо вона **відповідає вимогам**, її приймають, надсилають інструкції до завдання.
  - Якщо **не відповідає** — заявку відхиляють.

3. **Виконання завдання волонтером**:

- Волонтер виконує завдання.
- Завантажує звіт про виконану роботу.
- Координатор перевіряє звіт:
  - Якщо він **коректний**, приймає його.
  - Якщо **є помилки**, відправляє на доопрацювання, а волонтер коригує звіт і повторно надсилає.

4. **Фінансова підтримка та звітність**:

- Донор надає фінансову допомогу.

- Координатор формує фінансовий звіт.
- Донор переглядає звіт щодо витрачених коштів.

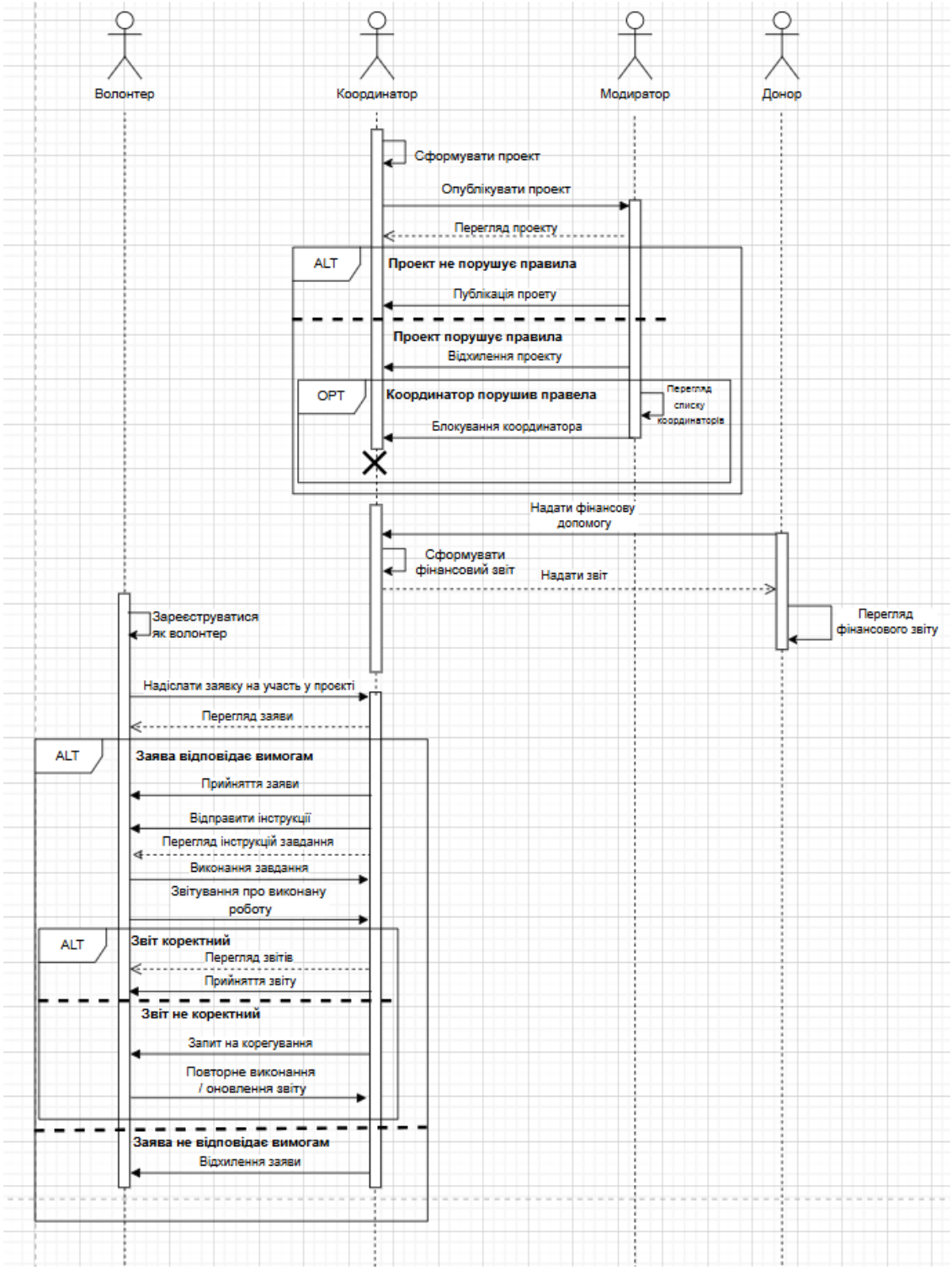

Рис. 1.5 – Діаграма послідовності UML

Завдяки цій діаграмі можна побачити, які компоненти системи взаємодіють між собою на кожному етапі виконання логіки.

Діаграма активності

Діаграма активності демонструє логіку виконання процесів, зокрема повний життєвий цикл участі волонтера у проекті — від подання заявки до подання звіту й затвердження виконаного завдання.

Основні етапи, представлені на діаграмі:

- Початок процесу — подання заявки;
- Рішення координатора — схвалити або відхилити;
- Призначення завдання волонтеру;
- Виконання завдання;
- Подання звіту;
- Перевірка координатором — прийняти або відправити на доопрацювання;
- Завершення завдання.

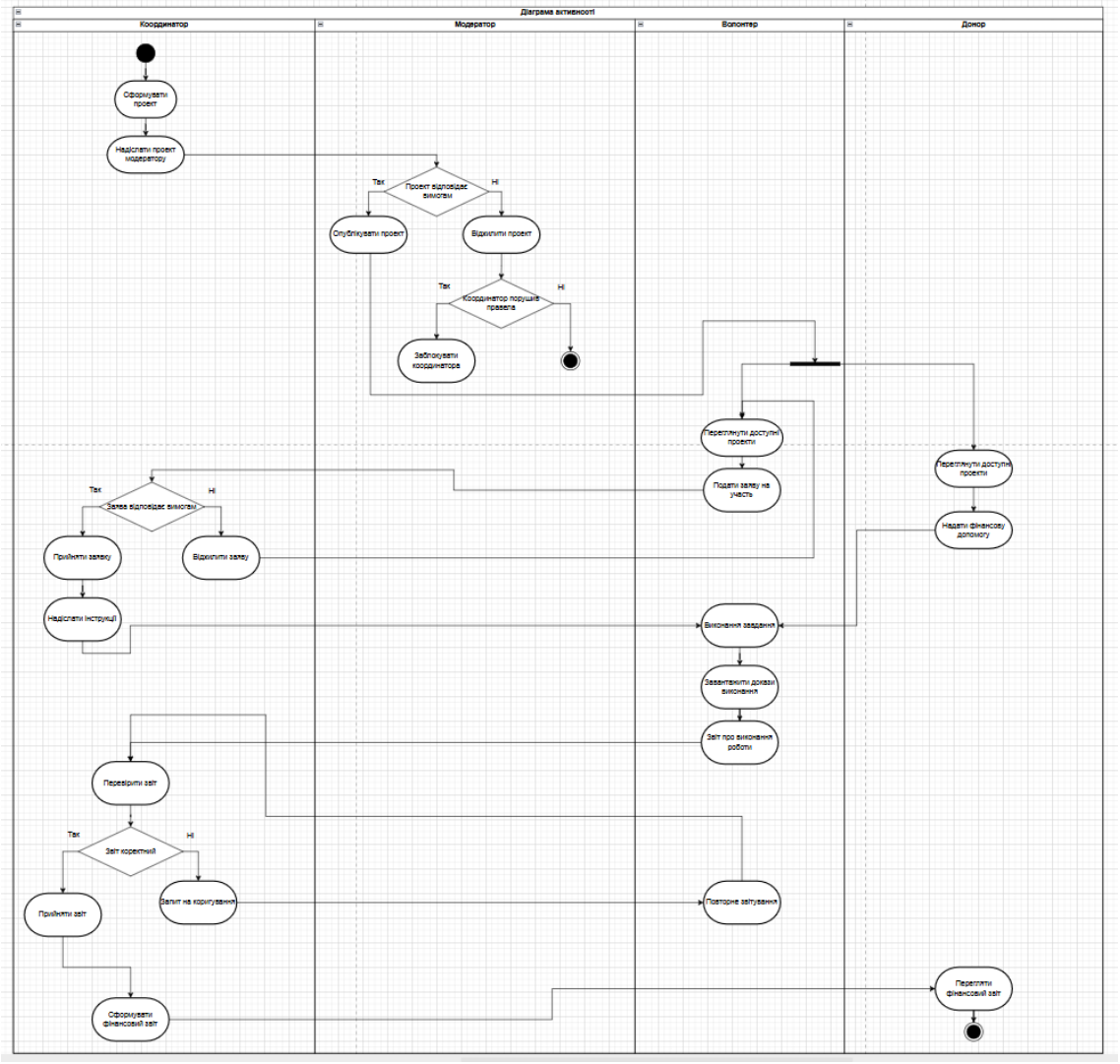

Рис. 1.6 – Діаграма активності UML

Ця діаграма наочно показує логіку роботи системи без прив’язки до технічної реалізації, з акцентом на послідовність дій і сценарії взаємодії.

Проведене моделювання предметної області дозволяє сформулювати чітке бачення архітектури та логіки роботи майбутньої системи. Побудовані діаграми стали основою для:

- проектування реляційної бази даних;
- формування маршрутів REST API;
- реалізації системи ролей і доступів;
- розробки інтерфейсів користувача.

Моделі також стали спільним засобом розуміння проекту всіма учасниками — розробником, керівником, аналітиком і потенційними користувачами.

1.5 Постановка завдання

На основі проведеного аналізу предметної області, огляду існуючих рішень, а також сформованих функціональних і нефункціональних вимог до майбутньої системи, можна сформулювати основне завдання, яке вирішується дипломною роботою.

Мета розробки: Створення веборієнтованого інформаційного середовища, що забезпечує цифрову підтримку волонтерських ініціатив, включаючи: управління проектами, реєстрацію волонтерів і донорів, контроль виконання завдань, модерацію проектів, звітування та прозорість фінансових операцій.

Об’єкт дослідження: Процеси організації волонтерської діяльності в онлайн-середовищі.

Предмет дослідження:Інформаційна система, що реалізує механізми управління волонтерами, координації проектів, збору допомоги, перевірки звітності та фільтрації доступу залежно від ролі користувача.

Основні задачі, які вирішуються у дипломній роботі:

1. **Розробити моделі предметної області** з використанням UML-діаграм (прецеденти, послідовності, активності) для формалізації взаємодії між основними суб’єктами системи: волонтером, координатором, модератором і донором.

2. **Спроектувати базу даних**, яка забезпечить збереження даних про користувачів, проекти, заявки, завдання, звіти, ролі та пожертви. При цьому слід урахувати нормалізацію, унікальні ключі та зв’язки між сутностями.

3. **Реалізувати багаторольову систему доступу** із поділом функціоналу між:

   - Волонтерами (подача заявки, виконання завдань, подання звітів);

- Координаторами (створення проектів, призначення завдань, перевірка звітів);
- Донорами (перегляд проектів, фінансова підтримка, перегляд фінзвітності);
- Модераторами (схвалення/відхилення проектів, блокування координаторів).

4. **Забезпечити логіку видимості статусів**, щоб уникнути витоку службової інформації:
   - Статуси виконання проектів (збір коштів, у процесі, завершено) — доступні всім.
   - Статуси модерації (на розгляді, затверджено, відхилено) — лише координатору.
5. **Розробити інтерфейс користувача**, що базується на React, з урахуванням адаптивності, простоти навігації та інтуїтивності.
6. **Реалізувати REST API** на базі Express.js, з урахуванням обробки ролей, статусів, безпеки авторизації, перевірки вводу та валідації даних.
7. **Забезпечити цілісність бізнес-логіки**, включаючи перевірку коректності звітів, повторне подання, перевірку дій координатора модератором, і фінансову прозорість для донорів.
8. **Впровадити базові перевірки (модерація, логіка допуску, блокування)** та зручну систему тестування із перевіркою вхідних сценаріїв.

Очікуваний результат:

Розробка багаторольової інформаційної системи, яка дозволяє організувати процеси волонтерської діяльності з високим ступенем прозорості, контролю та доступності, з урахуванням потреб користувачів і безпеки даних.

Висновки до розділу 1

У першому розділі дипломної роботи було здійснено системний аналіз предметної області «Підтримка волонтерської діяльності», що дозволило виявити ключові проблеми та визначити способи їх вирішення за допомогою сучасного вебзастосунку.

На основі вивчення тематичних джерел та огляду реальних українських платформ («dobro.ua», «volonter.org», «spilno.org») було виявлено відсутність комплексного рішення, яке поєднує управління завданнями, модерацію, пожертви, контроль звітності та чіткий поділ доступу.

У рамках моделювання було побудовано діаграми UML (прецедентів, послідовності, активності), які формалізували взаємодію між системними компонентами та користувачами. Особлива увага приділялася ролям, маршрутам заявок, перевірці звітів і механізмам модерації, що є надто важливими для реального функціонування такої системи.

Постановка завдання дозволила остаточно визначити структуру й функціональність системи, яка буде реалізована у наступних розділах, і задати чіткий напрям подальшої розробки, враховуючи вимоги до безпеки, зручності, прозорості та масштабованості.

2 ПРОЕКТУВАННЯ ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Логічна модель даних у вигляді ER-діаграми

2.1.1 Загальні положення

Одним із ключових етапів при проектуванні інформаційної системи є побудова логічної моделі даних, яка дозволяє формалізувати об'єкти предметної області, зв'язки між ними, а також основні атрибути, що необхідні для подальшого збереження, обробки й відображення інформації. Логічна модель даних — це представлення структури інформації незалежно від фізичного втілення в тій чи іншій системі управління базами даних. Така модель дозволяє не лише систематизувати дані, які оброблятиме система, але й виявити потенційні помилки ще до початку реалізації фізичної бази даних.

Для проекту "Програмне забезпечення інформаційної системи підтримки волонтерської діяльності" логічна модель даних була побудована у вигляді **ER-діаграми (Entity-Relationship Diagram)** — графічне подання основних сутностей, що описують об’єкти реального світу, атрибутів цих сутностей і зв’язків між ними. Завдяки цьому підходу розробники можуть на ранньому етапі виявити логічні помилки, суперечності, дублювання або відсутність ключових зв’язків у системі.

ER-діаграма виступає в ролі мосту між аналітиком та розробником: перший може формалізувати логіку бізнесу, а другий — перетворити її в реляційну структуру бази даних. Вона забезпечує однозначне трактування ролей і об’єктів системи незалежно від специфіки платформи реалізації.

2.1.2 Побудова ER-діаграми

Для реалізації логічної моделі була використана професійна система моделювання **ERwin Data Modeler**, яка дозволила побудувати ієрархічно-

структуровану ER-діаграму з дотриманням нормалізації (до 3НФ). Цей підхід забезпечив оптимальне представлення сутностей, уникаючи дублювання інформації та потенційної надлишковості.

На ER-діаграмі системи (Рис. 2.1) було відображено 9 ключових сутностей, понад 20 зв’язків, включаючи складні композиційні та асоціативні залежності, а також механізм розмежування доступу за ролями.

2.1.3 Види зв’язків між сутностями

У системі реалізовано різні типи зв’язків, що мають критичне значення для забезпечення цілісності даних. Зокрема:

- **Один до багатьох (1:N)** — наприклад, координатор створює багато проектів; один проект має багато завдань.
- **Багато до багатьох (M:N)** — волонтери можуть бути призначені до кількох завдань у різних проектах, і навпаки — одне завдання може бути розподілено між кількома волонтерами. Цей зв’язок реалізується через проміжну сутність (наприклад, Виконання).
- **Один до одного (1:1)** — наприклад, кожен фінансовий звіт належить до одного проекту.

Усі зв’язки мають атрибути, що забезпечують контроль статусів (затверджено, на перевірці, відхилено тощо) й дати змін, що є необхідним для формування журналів активності та автоматичного керування доступом.

2.1.4 Основні сутності та атрибути

Під час проектування бази даних були визначені ключові **сутності**, що безпосередньо відображають функціональні ролі та об’єкти системи:

1. Волонтер

- ID_Волонтера (PK)
- ПІБ, Пошта, Телефон, Досвід, Кількість виконаних завдань

Зв’язки:

- один-до-багатьох - Заява
- багато-до-багатьох - Завдання (через "Виконання")
- один-до-багатьох - Звіт

2. Заява

- ID_Заява (PK)
- Коментар, Дата подачі, Статус
- ID_Волонтера (FK)
- ID_Координатора (FK)

Зв’язки:

- кожна заява подається волонтером і розглядається координатором

3. Координатор

- ID_Координатора (PK)
- ПІБ, Пошта, Телефон, Статус акаунту

Зв’язки:

- один-до-багатьох - Проекти
- один-до-багатьох - Завдання
- один-до-багатьох - Звіти
- один-до-багатьох - Фінансові звіти
- один-до-багатьох - Заяви

4. Модератор

- ID_Модератора (PK)
- Ім’я, Пошта, Рівень доступу

Зв’язки:

- один-до-багатьох - Проекти

5. Проект

- ID_Проект (PK)
- Назва, Опис, Ціль, Дата створення, Статус
- ID_Модератора (FK), ID_Координатора (FK)

Зв’язки:

- один-до-багатьох - Завдання
- один-до-багатьох - Фінансовий звіт

6. Завдання

- ID_Завдання (PK)
- Назва, Опис, Інструкція, Дата завершення, Вимоги, Статус виконання
- ID_Проекту (FK), ID_Координатора (FK)

Зв’язки:

- багато-до-багатьох - Волонтер (через "Виконання")

7. Виконання (проміжна таблиця)

- ID_Завдання (PK, FK)
- ID_Волонтера (PK, FK)

- ID_Проекту (необов'язковий FK)

Зв’язки:

- проміжна таблиця для реалізації N:M зв’язку між Завданням і Волонтером

Це допоміжна таблиця, яка дозволяє реалізувати зв’язок багато-до-багатьох між волонтерами та завданнями. Вона містить атрибути: ID_Завдання, ID_Волонтера, ID_Проекту.

8. Звіт

- ID_Звіт (PK)
- Коментар, Фото/Документ, Дата, Статус перевірки
- ID_Волонтера (FK), ID_Координатора (FK)

Зв’язки:

- кожен звіт надсилається волонтером і перевіряється координатором

9. Фінансовий звіт

- ID_Фінансовий_звіт (PK)
- Дата створення, Сума, Призначення коштів, Стан
- ID_Координатора (FK)

Зв’язки:

- кожен фінансовий звіт прив’язаний до координатора і проекту

2.1.5 Відповідність моделі даних третій нормальній формі (3НФ)

На основі аналізу структури логічної моделі, представленої на **рис. 2.1**, можна стверджувати, що дана модель **повністю відповідає вимогам третьої нормальної форми (3НФ)**. Це підтверджується поетапною перевіркою нормалізації:

1. Перша нормальна форма (1НФ)

Усі атрибути у таблицях є **атомарними** — тобто містять лише одне значення. В таблицях не спостерігається масивів, списків або повторюваних груп. Наприклад:

- У таблиці Волонтер атрибут ПІБ не містить розділення на ім’я, прізвище, по батькові — але зберігає єдине значення.
- У таблиці Фінансовий звіт атрибут Сума містить одне числове значення — без розділення по валюті або категоріях витрат.

Отже, перша нормальна форма дотримана.

2. Друга нормальна форма (2НФ)

Усі неключові атрибути **повністю функціонально залежні від повного первинного ключа**. Жодна таблиця, що має складений первинний ключ (наприклад, Виконання з ключем ID_Завдання + ID_Волонтера), не містить атрибутів, які залежать лише від частини ключа.

- У таблиці Виконання — жоден неключовий атрибут (а таких немає) не залежить тільки від ID_Завдання або тільки від ID_Волонтера, що свідчить про дотримання 2НФ.
- Таблиці з простим ключем (Проект, Заявка, Координатор) мають атрибути, які залежні тільки від цілого ключа.

Таким чином, друга нормальна форма виконується.

3. Третя нормальна форма (3НФ)

Модель **не містить транзитивних залежностей** між неключовими атрибутами. Кожен неключовий атрибут залежить лише від первинного ключа своєї таблиці, і немає жодних атрибутів, які можна було б винести в окрему сутність.

Наприклад:

- У таблиці Заявка атрибут Статус стосується безпосередньо заявки, а не волонтера або координатора.
- У таблиці Фінансовий звіт атрибут Стан є частиною логіки звіту, і не залежить від атрибутів координатора.
- У таблиці Звіт атрибути ФотоДокумент, Коментар, Статус перевірки описують лише сам звіт, а не волонтера чи координатора.

Отже, транзитивних залежностей у моделі немає, що відповідає умовам 3НФ.

Візуальна модель

На основі вищезазначених сутностей, атрибутів та зв’язків у середовищі ERwin було побудовано діаграму, яка повністю відображає логічну структуру бази даних.

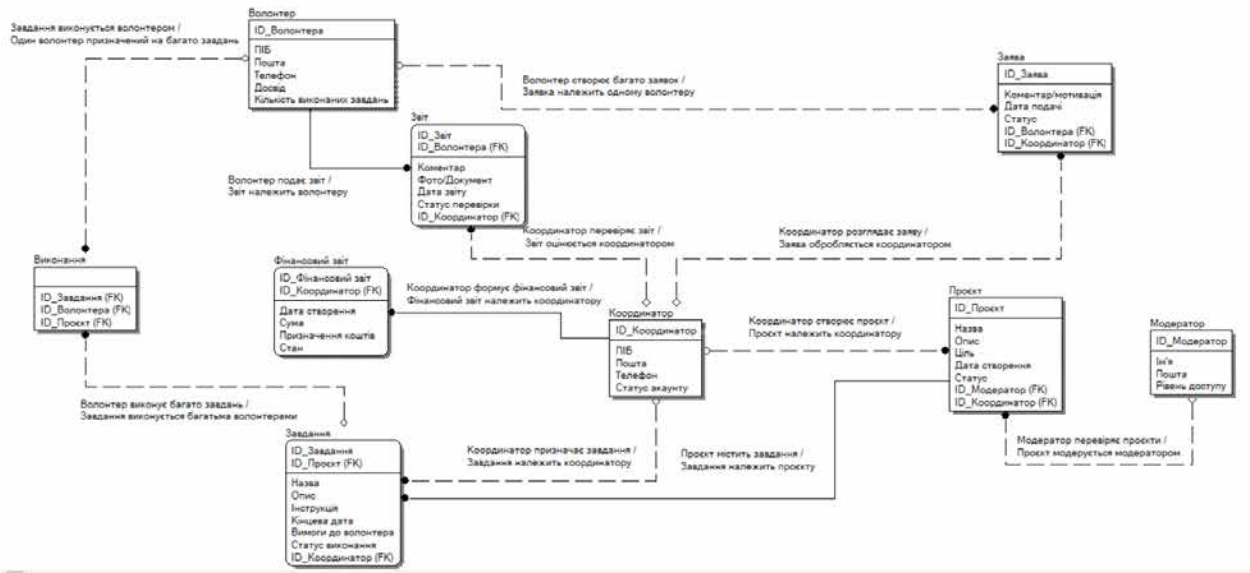

Рис. 2.1 – ER-діаграма логічної моделі бази даних для системи підтримки волонтерської діяльності

Логічна модель даних у вигляді ER-діаграми є фундаментом для побудови стійкої та масштабованої інформаційної системи підтримки волонтерської діяльності. Чітке визначення сутностей, атрибутів і зв’язків забезпечує високий рівень узгодженості, прозорості та безпеки при роботі з даними. Завдяки використанню ERwin Data Modeler діаграма є не лише візуальним елементом, а й технічним інструментом підтримки реалізації на всіх етапах — від проектування до запуску у виробництво.

2.2 Діаграма класів та кооперацій

Проектування архітектури програмного забезпечення передбачає визначення основних об’єктів системи, їхніх властивостей, методів та взаємозв’язків. Одним із найефективніших підходів до цього є використання UML-моделей, які дозволяють графічно та структуровано представити об'єктну модель системи. У цьому пункті розглядаються дві ключові моделі:

- діаграма класів — для формалізації структури системи;
- діаграми кооперацій — для демонстрації сценаріїв взаємодії між об'єктами під час виконання функціональних задач.

2.2.1 Діаграма класів

Діаграма класів є фундаментальним інструментом об’єктно-орієнтованого проектування, який дозволяє візуалізувати та формалізувати основну структуру системи. Вона показує об’єкти системи у вигляді класів, описує їх атрибути (властивості), методи (функції) та типи зв’язків між ними (асоціації, композиції, агрегації, залежності, успадкування). Побудова діаграми класів є критично важливою для забезпечення узгодженості логіки системи на етапах реалізації, тестування та масштабування. Вона забезпечує перехід від логічної моделі до програмної реалізації й безпосередньо впливає на структуру коду, що буде реалізовано.

Ключові цілі побудови діаграми класів:

- **Моделювання структури об'єктів** предметної області (користувачів, проектів, звітів, заявок тощо);
- **Визначення обов’язків** кожного об’єкта: які дії він може виконувати (методи), які дані зберігає (атрибути);
- **Встановлення зв’язків** між об'єктами, що взаємодіють;
- **Підготовка до генерації коду** для реалізації системи у вигляді програмних класів

На діаграмі класів (рис. 2.2.1) представлені основні учасники системи, об’єкти, з якими вони взаємодіють, та їх структурні зв’язки.

- **Клас «Volunteer» (Волонтер)** — відповідає за представлення користувачів, які подають заявки та виконують завдання. Він асоційований з класом «Application» (Заявка), який описує запити на участь у проектах, що створюються координаторами.
- **Клас «Coordinator» (Координатор)** — особа, що створює проекти, завдання та керує волонтерами. Має прямі зв’язки з класами «Project» (Проект), «Task» (Завдання) і «Report» (Звіт).

- **Клас «Moderator» (Модератор)** — представлений як окремий клас, пов’язаний з «Project», оскільки саме він приймає або відхиляє проекти перед їх публікацією.
- **Клас «Donor» (Донор)** — має асоціацію з «FinancialReport» (Фінансовий звіт), що дозволяє йому переглядати прозорість фінансування проектів.
- **Клас «Project» (Проект)** — центральна сутність, з якої формуються завдання. Має зв’язки з «Coordinator», «Volunteer» (через заявки), а також з класом «Task».
- **Клас «Task» (Завдання)** — конкретизує обсяг роботи, що виконується волонтерами. Пов’язаний із класом «Report», через який формується зворотній зв’язок по виконанню.
- **Клас «Application» (Заявка)** — описує запит волонтера на участь у проекті. Має атрибути: статус, мотиваційний коментар, дата подачі. Пов’язаний із класами «Volunteer», «Project» і обробляється координатором.
- **Клас «Report» (Звіт)** — описує підсумок виконаного завдання: може містити фото, текстовий коментар, статус перевірки.
- **Клас «FinancialReport» (Фінансовий звіт)** — клас, що фіксує звітність координатора про витрати по проекту. Пов’язаний із «Project» та відкривається для перегляду «Donor».

Усі асоціації між класами реалізовані згідно з логікою предметної області:

- один координатор може створити багато проектів;
- кожен проект містить багато завдань;
- кожен волонтер може подати кілька заявок;
- заявка може бути прив’язана лише до одного проекту;
- завдання може мати декілька призначених волонтерів, і кожен волонтер може виконувати багато завдань (через асоціативну сутність);

- кожне завдання має один звіт;
- кожен проект має не більше одного фінансового звіту.

Таким чином, діаграма класів не тільки окреслює основну структуру системи, а й демонструє основи логіки доступу, обов’язків користувачів та циклу обробки даних у рамках системи

Діаграма класів інформаційної системи підтримки волонтерської діяльності знаходиться в Додатку Б

2.2.2 Діаграми кооперацій

Якщо діаграма класів описує **статичну структуру системи**, то діаграми кооперацій (cooperation diagrams або collaboration diagrams) дозволяють змоделювати **динамічну взаємодію об’єктів** у межах конкретних сценаріїв використання. Вони демонструють, як екземпляри класів (об’єкти) обмінюються повідомленнями під час виконання певної задачі — з вказанням черговості, ролей та логіки виконання.

Для інформаційної системи підтримки волонтерської діяльності були змодельовані найтиповіші сценарії кооперацій, які охоплюють ключові етапи життєвого циклу проекту та волонтерської участі в ньому. Ці сценарії описані нижче та представлені на діаграмах (див. рис. 2.2.2 – 2.2.5).

У процесі побудови архітектури програмного забезпечення важливо не лише визначити структуру системи у вигляді класів, а й змоделювати сценарії їх взаємодії під час виконання ключових функціональних процесів. Для цього використовуються **UML-діаграми кооперацій** (collaboration diagrams), які наочно демонструють динамічну поведінку системи в межах конкретного механізму чи задачі.

На основі діаграми класів, побудованої в попередній лабораторній роботі, було ідентифіковано **чотири основні механізми (кооперації)**, що

охоплюють логіку життєвого циклу волонтерської діяльності в межах розроблюваної системи. Для кожного з механізмів виділено відповідні класи та сформовано окрему діаграму, яка ілюструє їхню взаємодію.

1. Подання заявки волонтером

Ця кооперація відображає початкову взаємодію користувача типу «волонтер» із системою. Волонтер обирає проект, заповнює заявку із зазначенням коментаря або мотивації, після чого вона передається координатору на розгляд. У системі заявці автоматично присвоюється статус **«на розгляді»**.

Залучені класи:

- Волонтер
- Заявка
- Проект
- Координатор

Опис процесу:

1. Волонтер створює заявку на основі відкритого проекту.
2. Система перевіряє, чи не існує вже активної заявки.
3. Якщо все вірно — створюється новий об’єкт Заявка.
4. Клас Координатор отримує доступ до заявки для подальшого розгляду.

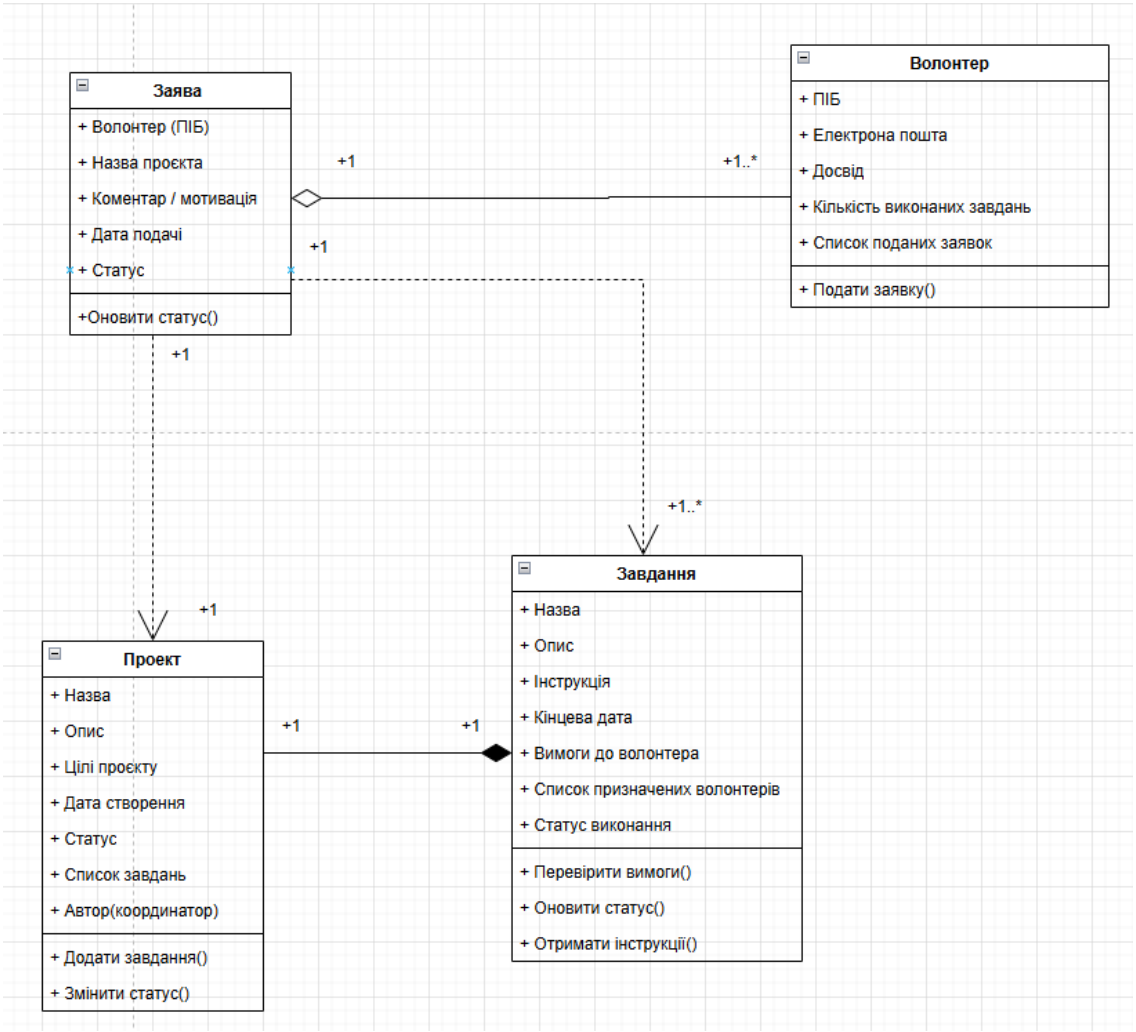

Рис. 2.3 – Діаграма Участь волонтера в проекті

2. Виконання та звітування по завданню

Після того як заявку волонтера схвалено, координатор призначає йому відповідне завдання. Після виконання завдання волонтер формує звіт, який передає координатору на перевірку. Цей процес ілюструє реалізацію відповідальності, зворотного зв’язку та контролю якості виконаної роботи.

Залучені класи:

- Завдання
- Волонтер
- Звіт про виконання
- Координатор

Опис процесу:

1. Координатор призначає завдання волонтеру.
2. Волонтер виконує завдання згідно з інструкцією.
3. Формує об'єкт Звіт про виконання, додає коментар, докази (фото).
4. Система передає звіт координатору.
5. Координатор перевіряє та змінює статус звіту (наприклад, **"прийнято"** або **"на доопрацювання"**).

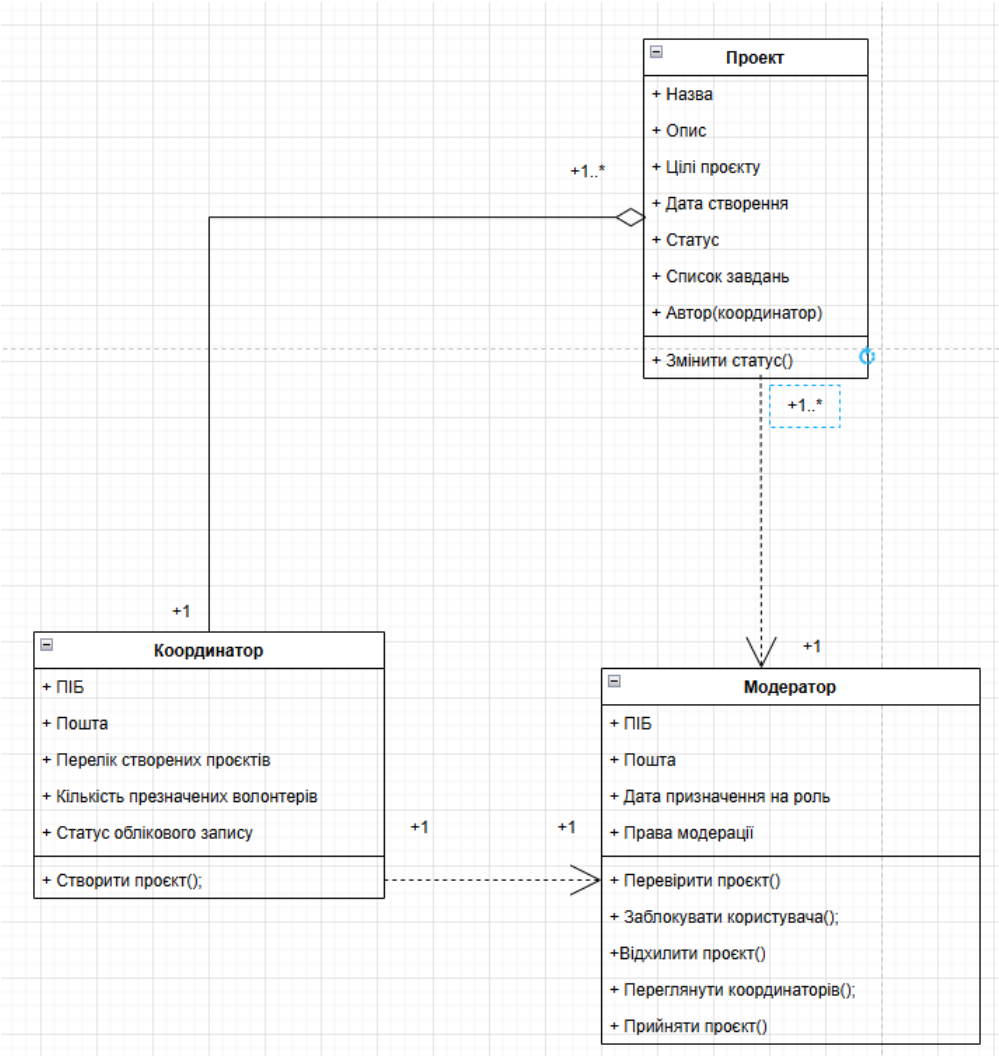

Рис. 2.4 – Діаграма Звітність по завданню

3. Модерація проектів перед публікацією

Перед тим як створений координатором проект стане доступним волонтерам або донорам, він має бути перевірений модератором. Це ключовий етап забезпечення якості контенту та верифікації інформації. Лише після схвалення модератором проект змінює статус на **«опубліковано»**.

Залучені класи:

- Проект
- Координатор
- Модератор

Опис процесу:

1. Координатор створює новий проект.
2. Система автоматично встановлює статус **«на перевірці»**.
3. Модератор отримує доступ до списку нових проектів.
4. Перевіряє інформацію, і приймає рішення:

o Якщо проект відповідає вимогам — статус змінюється на **«опубліковано»**.

o Якщо є недоліки — статус стає **«відхилено»**.

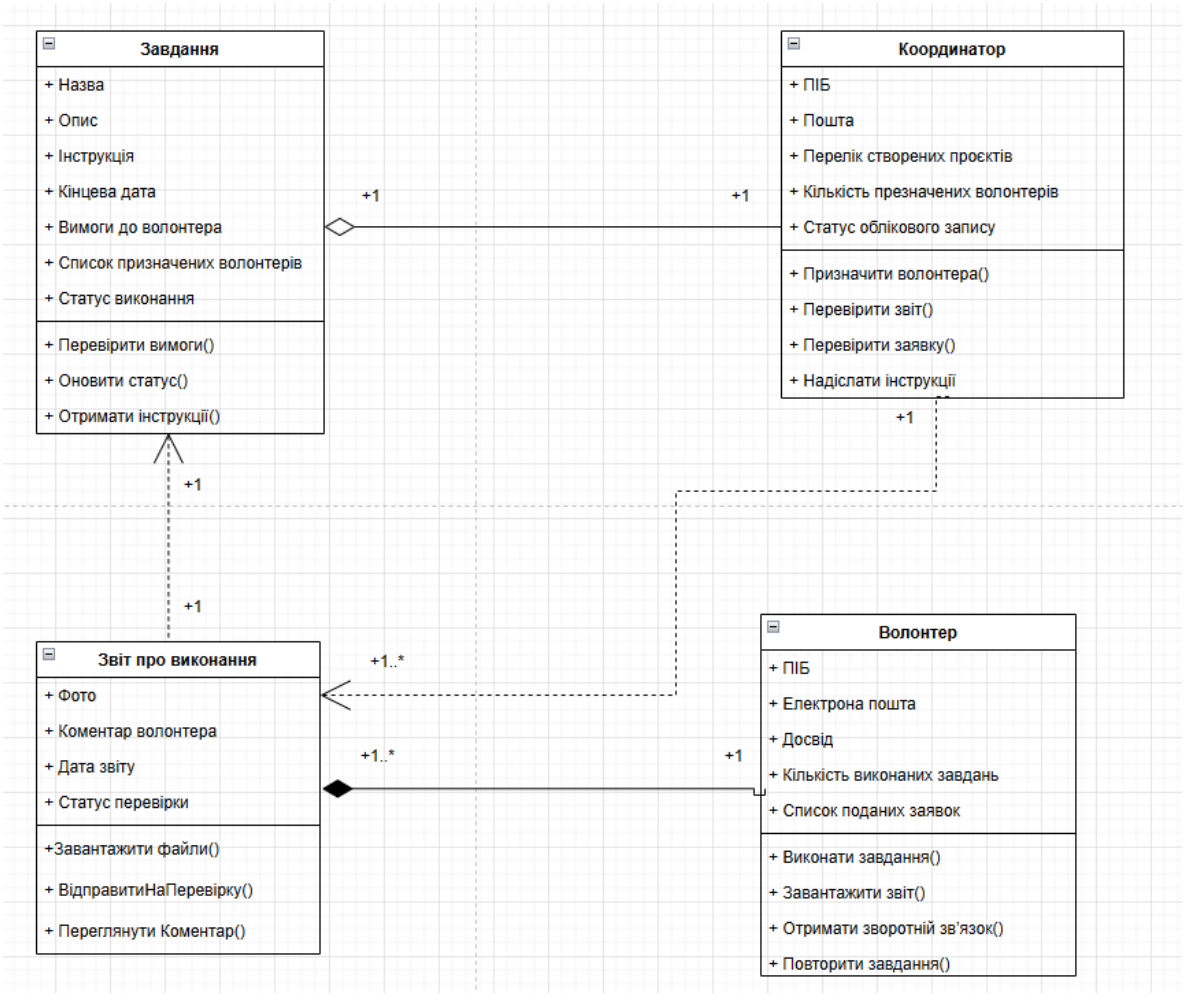

Рис. 2.5 – Діаграма Модерація проектів

4. Формування фінансової звітності для донорів

Прозорість використання коштів є критично важливою в роботі з донорами. Після завершення або в процесі реалізації проекту координатор формує фінансовий звіт, який детально описує витрати. Цей звіт відкривається для перегляду донорам, які підтримують платформу або конкретні проекти.

Залучені класи:

- Фінансовий звіт
- Проект
- Координатор
- Донор

Опис процесу:

1. Координатор заповнює об'єкт Фінансовий звіт, вказуючи витрати, призначення коштів.
2. Система фіксує звіт за відповідним проектом.
3. Донор отримує доступ до перегляду опублікованих звітів.
4. На основі цього звіту донор приймає рішення щодо майбутнього фінансування.

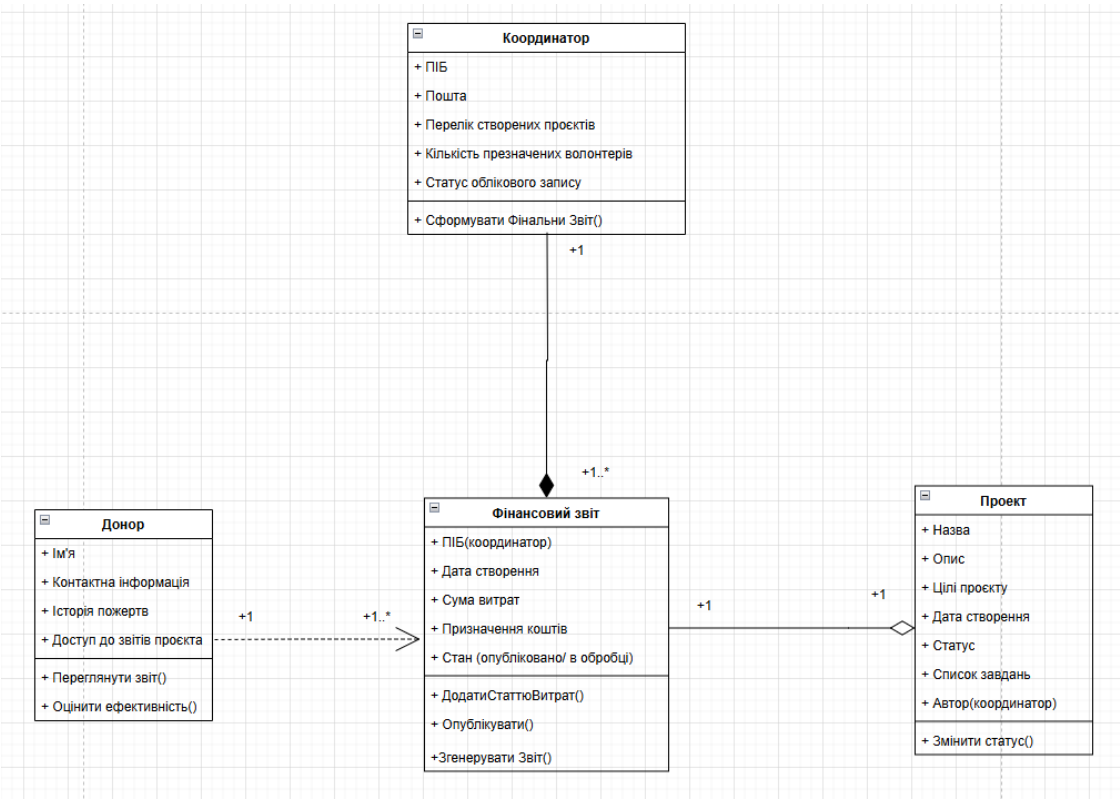

Рис. 2.6 – Діаграма Фінансова прозорість

Розроблені діаграми кооперацій демонструють ключові механізми взаємодії об'єктів у межах системи підтримки волонтерської діяльності. Завдяки розбиттю загальної логіки на окремі сценарії, стає можливим поетапне впровадження функціоналу, його модульне тестування та подальше масштабування. Також ці кооперації закладають основу для майбутнього побудування компонентів, контролерів, API-інтерфейсів та шаблонів представлення інформації на клієнтському рівні.

2.3 Діаграма пакетів

У процесі архітектурного проектування програмного забезпечення важливо не лише описати окремі функціональні компоненти, а й структурно організувати їх у логічні блоки, що забезпечують зрозумілу і модульну архітектуру системи. З цією метою використовується діаграма пакетів (package diagram), яка є одним з інструментів мовою UML для представлення високорівневої організації коду або логіки системи.

Діаграма пакетів дозволяє візуалізувати поділ великої системи на підсистеми або модулі (пакети), які виконують конкретні ролі у програмному середовищі. Такий підхід підвищує рівень абстракції та спрощує розуміння структури програми, особливо при її масштабуванні або модифікації.

2.3.1 Структура діаграми

На рис. 2.3 наведено діаграму пакетів програмного забезпечення інформаційної системи підтримки волонтерської діяльності. Дана діаграма була створена у середовищі draw.io і розроблена відповідно до вимог логічної декомпозиції архітектури системи.

У представленій моделі виділено п’ять основних інтерфейсних рівнів відповідно до ролей користувачів:

1. ModeratorInterface

Цей пакет відповідає за інтерфейс модератора, включає два підмодулі:

- ProjectModeration – модерація проектів: перевірка та схвалення/відхилення.
- UserModeration – перегляд і керування списком координаторів.

2. CoordinatorInterface

Центральний адміністративний інтерфейс координатора, поділений на 4 ключові підпакети:

- VolunteerManagement – перевірка заяв волонтерів, розподіл між проектами.
- TaskManagement – створення, контроль, редагування завдань.
- ProjectManagement – створення та керування проектами.
- FinancialReporting – формування фінансових звітів після завершення збору.

3. VolunteerInterface

Інтерфейс для волонтерів, що охоплює:

- ApplicationSubmission – подання заявок на участь у проектах.
- TaskExecution – доступ до завдань, виконання інструкцій.
- Reporting – створення звіту та завантаження доказів виконаної роботи.

4. UserInterface

Універсальний публічний інтерфейс системи, що містить:

- ProjectViewing – перегляд активних і завершених проектів.
- FundSupport – здійснення пожертв.
- FinancialReview – ознайомлення з фінансовими звітами.

2.3.2 Взаємозв’язки між пакетами

Між пакетами існують залежності (відображені пунктирними стрілками), які показують, які модулі взаємодіють між собою. Наприклад:

- FundSupport залежить від FinancialReporting, оскільки інформація про витрати формується координатором.
- ApplicationSubmission у волонтерському інтерфейсі пов’язане з VolunteerManagement у координаторському модулі, адже заявка надходить саме туди.
- ProjectModeration у модераторському пакеті залежить від ProjectManagement, адже саме координатор створює проект, а модератор його перевіряє.

Такий підхід дозволяє не лише зрозуміти, як компоненти системи пов’язані між собою, а й спроектувати **слабко зв’язані (loosely coupled)** та **високомодульні** блоки.

2.3.3 Призначення та переваги

Поділ системи на логічні пакети забезпечує низку архітектурних переваг:

- **Масштабованість** — до системи можна легко додавати нові пакети (наприклад, для нових ролей або підсистем).
- **Зрозумілість структури** — кожен пакет виконує чітко визначену роль, спрощуючи навігацію у великому проекті.
- **Розподіл обов’язків** — дозволяє делегувати розробку окремих пакетів різним командам.
- **Полегшене тестування** — модулі можна ізолювати для юніт-тестування.
- **Підтримка безпечного доступу** — логічна ізоляція дозволяє застосовувати політики авторизації на рівні пакетів (наприклад, лише координатор може створювати завдання).

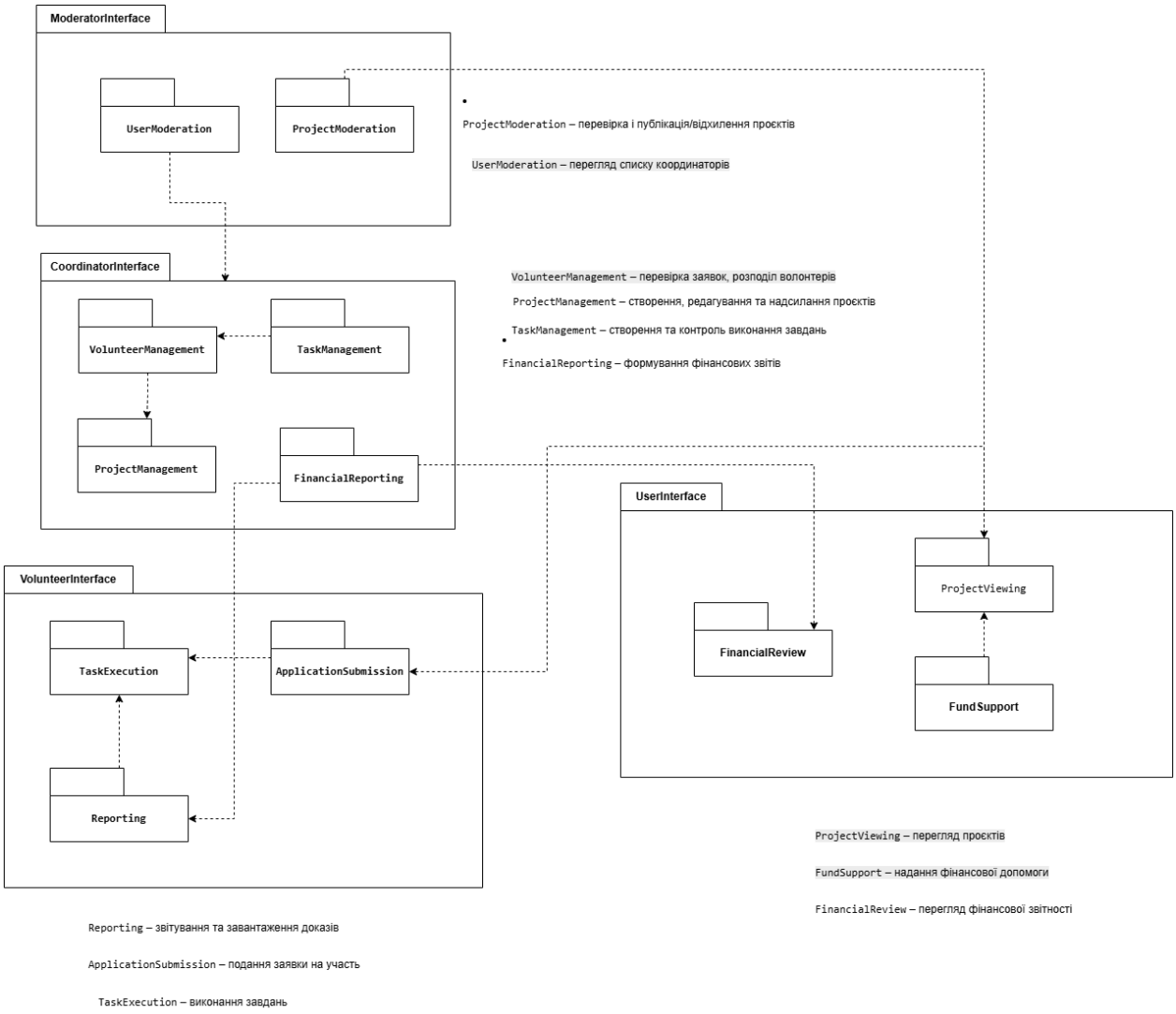

Рис. 2.7 – Діаграма пакетів інформаційної системи підтримки волонтерської діяльності

Усі залежності між пакетами в діаграмі позначено пунктирними стрілками, які демонструють взаємодію одного інтерфейсу з іншим. Наприклад, модуль FundSupport взаємодіє з FinancialReporting, що дозволяє донору переглядати фінансову статистику проекту перед здійсненням внеску. А модуль TaskExecution безпосередньо пов’язаний із TaskManagement, що забезпечує отримання актуальних завдань волонтерами.

Таке розбиття на інтерфейси та внутрішні пакети забезпечує не лише логічну структурованість, а й сприяє модульності, повторному використанню коду, зручному супроводу й тестуванню програмного забезпечення.

Завдяки діаграмі пакетів можна легко виявити потенційно зайві залежності, скоригувати архітектурну складність і забезпечити стабільну основу для масштабування системи в майбутньому. Кожен пакет може реалізовуватися незалежно, що особливо важливо для командної розробки.

Таким чином, діаграма пакетів дозволяє візуально окреслити архітектурну організацію системи, ієрархію її підсистем та логічний поділ за ролями. Така візуалізація є незамінною для етапів проектування, супроводу, рефакторингу та командної роботи над масштабним проектом.

2.4 Діаграма компонентів

Побудова компонентної архітектури є важливим етапом проектування інформаційної системи, адже вона дозволяє візуалізувати ключові модулі програмного забезпечення, способи їх взаємодії, обмін даними та залежності між функціональними частинами системи. У контексті програмного забезпечення **інформаційної системи підтримки волонтерської діяльності** компонентний підхід допомагає забезпечити масштабованість, зрозумілу структуру та легкість у підтримці.

Діаграма компонентів у UML (Component Diagram) ілюструє фізичні частини системи — програмні компоненти (бібліотеки, підсистеми, сервіси), які реалізують певну функціональність і можуть бути повторно використані в інших системах.

2.4.1 Структура компонентної архітектури

У системі підтримки волонтерської діяльності було виділено наступні **основні підсистеми** (компоненти), кожна з яких реалізує окремий напрям функціональності:

1. Реєстрація та авторизація користувача

- **Компонент:** Обробка реєстрації та авторизації
- **Призначення:** забезпечення аутентифікації та контролю доступу користувачів різних ролей
- **Взаємодії:** передає авторизованого користувача до компоненту інтерфейсу

2. Інтерфейс користувача

- **Компоненти:**
  - Інтерфейс волонтера
  - Інтерфейс координатора
  - Система доступу до статистики
  - Система створення заявок
- **Призначення:** організація доступу користувачів до функцій системи відповідно до ролі

3. Комунікаційна система

- **Компонент:** Обробка повідомлень
- **Призначення:** надсилання сповіщень, відповідей, запитів між користувачами та системою

4. Звітність та аналітика

- **Компонент:** Генерація звітів
- **Призначення:** створення фінансових звітів, аналітичних висновків, логів активності

5. Інтеграція з зовнішніми платформами

- **Компонент:** Обмін даними з API

- **Призначення:** взаємодія з іншими інформаційними системами (наприклад, фінансовими або державними)

6. Управління завданнями

- **Компонент:** Призначення завдань
- **Призначення:** розподіл завдань між волонтерами, зберігання статусів, перевірка відповідності

7. Управління проектами

- **Компонент:** Ініціалізація та супровід проектів
- **Призначення:** створення, оновлення, завершення проектів координаторами

8. Сховище даних

- **Призначення:** централізоване зберігання інформації про користувачів, заявки, завдання, звіти

Діаграма компонентів інформаційної системи підтримки волонтерської діяльності знаходиться в Додатку В

2.4.2 Взаємозв'язки між компонентами

Діаграма демонструє такі ключові взаємодії:

- **Інтерфейс користувача** — є основною точкою маршрутизації запитів. Він взаємодіє з системою авторизації, отримує доступ до модулів завдань, заявок, звітності.
- **Компонент логіки обробки завдань і проектів** — відповідає за координацію процесів. Отримує команди з інтерфейсу користувача, обробляє їх та передає дані у сховище.

- **Комунікаційний компонент** — реагує на дії, формує повідомлення (наприклад, підтвердження заявки чи зміна статусу).
- **Аналітика** — формує звіти на основі даних, збережених у БД.
- **Сховище даних** — взаємодіє зі всіма модулями, забезпечуючи надійне зберігання інформації.

2.4.3 Переваги використаної архітектури

1. **Модульність**: кожен компонент відповідає за чітко визначений набір функцій, що дозволяє ізолювати логіку та спростити тестування.
2. **Гнучкість**: можна легко замінити компонент або інтегрувати новий (наприклад, модуль підтримки Telegram-бота).
3. **Безпека**: доступ до бази даних мають лише внутрішні серверні компоненти.
4. **Масштабованість**: систему легко розширювати, додаючи нові сервіси або API.
5. **Підтримка**: модульний підхід полегшує усунення помилок і впровадження змін без порушення всієї логіки.

Висновки до розділу 2

У другому розділі було здійснено комплексне проектування інформаційного та програмного забезпечення системи підтримки волонтерської діяльності. Цей етап відіграє ключову роль у формуванні архітектури системи, визначенні її логіки, структури даних та взаємодії між функціональними модулями.

На першому етапі було побудовано **ER-діаграму**, яка відобразила логічну модель даних, що охоплює основні сутності предметної області: користувачів різних типів (волонтер, координатор, модератор, донор), заявки, проекти, завдання, звіти та фінансові операції. Створена модель пройшла процес нормалізації до **третьої нормальної форми (3НФ)**, що гарантує

відсутність надлишкових залежностей, покращену структурованість та гнучкість подальшого використання в реляційній СУБД.

Далі було сформовано **діаграму класів**, яка деталізує об’єктно-орієнтовану структуру системи: класи, їх атрибути, методи та зв’язки. Ця діаграма дозволила перейти від концептуального бачення до проектного рівня, з якого надалі буде реалізовано програмну логіку.

Для кращого розуміння сценаріїв взаємодії між об’єктами було створено **діаграми кооперацій**, що демонструють процеси подання заявки, її розгляду, призначення завдань та подання звітів. Це дозволило чітко зафіксувати поведінку системи в динаміці, зокрема у взаємодії між волонтером, координатором і модератором.

Наступним кроком стало створення **діаграми пакетів**, яка забезпечила логічне групування функціональних блоків системи на високому рівні абстракції. В межах цієї структури були виділені окремі інтерфейсні зони (інтерфейси користувача) для кожної ролі системи: волонтера, координатора, модератора та донора. Така модульність дозволяє легше масштабувати систему, впроваджувати нові ролі або функції без порушення вже існуючої логіки.

Завершальним етапом стало формування **діаграми компонентів**, яка дозволила відобразити фізичну архітектуру програмного забезпечення: ключові модулі, такі як інтерфейс користувача, система авторизації, обробка заявок, керування завданнями, модулі звітності та аналітики, а також центральне сховище даних. Було проаналізовано їхні взаємозв’язки та точки інтеграції, що дозволяє забезпечити ефективну координацію між модулями та стабільність системи в умовах розширення.

Таким чином, розділ 2 повністю охоплює процес формування технічної структури програмного забезпечення, що забезпечує основу для його надійної

реалізації, масштабованості, зручності супроводу та відповідності функціональним вимогам. Усі створені моделі відображають актуальні потреби предметної області та дозволяють максимально ефективно реалізувати функціональність інформаційної системи підтримки волонтерської діяльності.

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Система управління інформаційною базою

У процесі розробки інформаційної системи підтримки волонтерської діяльності важливим кроком є вибір системи управління інформаційною базою. Від якості цієї системи залежить швидкодія, масштабованість, цілісність та безпека оброблюваних даних, що особливо важливо для систем, які працюють з персональними даними користувачів, фінансовими транзакціями та звітністю.

З огляду на обсяг, складність та структуру даних, які зберігаються в системі, для реалізації інформаційної бази було обрано **PostgreSQL** — потужну, безкоштовну реляційну СУБД з відкритим кодом, яка підтримує об’єктно-реляційну модель даних.

PostgreSQL поєднує в собі надійність, високу продуктивність та гнучкість, що робить її ідеальною для проектів, які орієнтовані на зростання, зміну функціоналу та широке коло користувачів. Її використання виправдане наступними факторами:

- **Об’єктно-реляційна модель:** дозволяє поєднувати класичні таблиці з підтримкою об’єктно-орієнтованих підходів, таких як наслідування таблиць, створення власних типів даних і функцій.
- **Підтримка JSON/JSONB:** надає змогу ефективно зберігати напівструктуровані дані, що може бути корисним для зберігання додаткової інформації про заявки, завдання чи дії користувачів.
- **Транзакційність:** повна підтримка транзакцій дозволяє забезпечити цілісність операцій, що особливо важливо при роботі з волонтерами, пожертвами, перевірками заявок тощо.

- **Висока продуктивність:** PostgreSQL демонструє стабільну роботу з великими обсягами даних, має оптимізований рушій виконання запитів та підтримує індекси, тригери, збережені процедури.
- **Безпека:** СУБД підтримує ролі, обмеження доступу до окремих об’єктів БД, шифрування та журнали дій, що дозволяє реалізувати багаторівневу систему авторизації для волонтерів, координаторів, модераторів та донорів.
- **Масштабованість:** PostgreSQL підходить як для невеликих автономних рішень, так і для масштабних проектів, що розгортаються у хмарному середовищі або в Docker-контейнерах.
- **Підтримка хмар та контейнеризації:** СУБД легко інтегрується в сучасну DevOps-інфраструктуру, має офіційні образи для Docker, підтримується у хмарних платформах AWS, GCP, Azure.

Перевага PostgreSQL також у її активній спільноті, наявності великої кількості документації, можливості гнучкого розширення функціоналу через плагіни та модулі (наприклад, PostGIS для геоданих або pgAudit для розширеного логування).

У межах цього проекту PostgreSQL виконує роль основного сховища для:

- облікових записів користувачів (волонтерів, координаторів, модераторів, донорів),
- інформації про проекти, їх статуси, строки та координаторів,
- даних про завдання, заявки, звіти, фінансові транзакції.

Розроблена логічна модель даних у вигляді ER-діаграми, яка раніше була представлена у розділі 2.1, реалізована у фізичній структурі бази даних за допомогою SQL-запитів у PostgreSQL. Усі сутності та їх зв’язки збережено,

забезпечено референтну цілісність, введено ключі, обмеження, а також індексацію по критичних полях (email, ID користувача, ID заявки тощо).

Таким чином, обраний інструмент — PostgreSQL — повністю відповідає потребам проекту, забезпечуючи оптимальну продуктивність, зручність адміністрування, підтримку складної бізнес-логіки та надійне зберігання даних.

3.2 Розробка інформаційної бази

На основі логічної моделі даних, реалізованої у вигляді ER-діаграми, було здійснено побудову фізичної моделі інформаційної бази. Розробка інформаційної бази є фундаментальним етапом при створенні програмного забезпечення, оскільки вона забезпечує стійке збереження, швидкий доступ і ефективну обробку критично важливих даних.

У межах програмного забезпечення інформаційної системи підтримки волонтерської діяльності було створено повноцінну інформаційну базу з використанням СУБД PostgreSQL. Її структура визначена в файлі schema.ts, де описані всі сутності, що відповідають ключовим компонентам системи: користувачі, проекти, завдання, заявки, звіти, донори, пожертви, а також статуси модерації проектів.

Структура таблиць та атрибутів

Для забезпечення відповідності логічній моделі, фізична модель містить такі основні таблиці:

- **users** — містить дані про всіх користувачів системи (волонтерів, координаторів, модераторів, донорів). Передбачає поля для збереження імені, електронної пошти, пароля, ролі, дати створення тощо.

- **projects** — таблиця, що описує волонтерські проекти. Містить назву, опис, ціль, бюджет, статус модерації, дату створення та ідентифікатор координатора.
- **tasks** — зберігає завдання, пов’язані з проектами. Поля включають назву, опис, дату дедлайну, статус виконання, а також зовнішній ключ на відповідний проект.
- **applications** — таблиця заявок волонтерів на участь у проектах.
- **reports** — звіти волонтерів, що додаються після виконання завдань, включаючи підтверджувальні матеріали (наприклад, фото або текстовий опис).
- **donations** — інформація про фінансові або матеріальні внески від донорів, а також їхнє призначення (посилання на проект).

Усі таблиці містять первинні ключі та використовують зовнішні ключі для встановлення зв’язків між сутностями. Наприклад, зв’язок між волонтером і його заявками реалізується через поле user_id, а між проектом і завданнями — через project_id.

Структура бази даних

Уся структура бази описана в файлі schema.ts, де за допомогою бібліотеки drizzle-orm визначені основні таблиці. Наприклад, опис таблиці користувачів виглядає наступним чином:

```
export const users = pgTable("users", {
  id: serial("id").primaryKey(),
  name: varchar("name", { length: 255 }).notNull(),
  email: varchar("email", { length: 255 }).notNull().unique(),
  role: varchar("role", { length: 50 }).notNull(), // volunteer, coordinator, moderator, donor
  createdAt: timestamp("created_at").defaultNow(),
});
```

Також визначено таблиці для проектів (projects), заявок (applications), завдань (tasks), пожертв (donations) тощо. Ось приклад структури для таблиці projects:

```
export const projects = pgTable("projects", {
  id: serial("id").primaryKey(),
  title: varchar("title", { length: 255 }).notNull(),
  description: text("description").notNull(),
  goal: varchar("goal", { length: 255 }),
  createdAt: timestamp("created_at").defaultNow(),
  coordinatorId: integer("coordinator_id").references(() => users.id),
  moderationStatus: varchar("moderation_status", { length: 50
}).default("pending"),
});
```

Це забезпечує збереження важливих атрибутів кожного проекту, його статусу модерації, прив'язку до координатора тощо.

Початкове заповнення бази

Для ініціалізації тестових даних використовується файл seedData.ts. Там створюються початкові координатори, проекти та завдання, що дозволяє швидко запустити демонстраційну версію системи. Наприклад:

```
await db.insert(users).values({
  name: "Test Coordinator",
  email: "coordinator@example.com",
  role: "coordinator",
});
```

Валідація та обмеження цілісності

У схемі реалізовані всі необхідні обмеження на рівні бази даних:

- Унікальність електронної пошти користувача (email).

- Обов’язковість заповнення основних полів (not null).
- Індекси на зовнішні ключі для підвищення продуктивності при виконанні запитів.
- Статуси модерації (pending, approved, rejected) у таблиці projects реалізовані через перелік (enum), що забезпечує контроль над допустимими значеннями.

Ініціалізація бази даних

Для початкового заповнення бази даних використовується файл seedData.ts, який містить приклади створення початкових користувачів (координаторів, волонтерів, модераторів), тестових проектів та завдань. Це дає змогу швидко розгорнути систему для демонстраційних або тестових цілей.

Використання ORM

Замість прямого написання SQL-запитів, у проекті застосовано бібліотеку drizzle-orm, яка забезпечує типобезпечну, декларативну побудову запитів до бази. У файлі storage.ts реалізовано набір функцій для виконання найпоширеніших дій: отримання користувача, збереження проекту, додавання заявки, обробка звіту тощо.

Взаємодія з базою даних

У файлі db.ts налаштовано з’єднання з PostgreSQL, використовуючи відповідні змінні оточення для гнучкого налаштування. Весь доступ до бази здійснюється через абстрактний шар збереження (IStorage), що відповідає принципам інкапсуляції й дозволяє за потреби змінити СУБД без змін у прикладному коді.

У файлі db.ts конфігурується підключення до PostgreSQL:

```
import { drizzle } from "drizzle-orm/node-postgres";
import { Pool } from "pg";
```

```
const pool = new Pool({ connectionString: process.env.DATABASE_URL });
export const db = drizzle(pool);
```

Завдяки такій абстракції — запити до бази можна виконувати через ORM, що підвищує безпеку (захист від SQL-ін’єкцій), покращує читабельність і підтримуваність коду.

3.3 Вибір інструментарію для створення прикладного програмного забезпечення

При розробці програмного забезпечення для інформаційної системи підтримки волонтерської діяльності надзвичайно важливим етапом є вибір відповідного інструментарію. Обрана технологічна основа безпосередньо впливає на зручність реалізації функціоналу, масштабованість, продуктивність, інтеграцію з базою даних та взаємодію з кінцевим користувачем.

Для створення даної системи було прийнято рішення використовувати стек JavaScript/TypeScript, що поєднує простоту розробки, ефективність обробки запитів у реальному часі та можливість інтеграції з сучасними інструментами для побудови UI/UX.

Серверна частина (Backend)

У якості середовища виконання бекенду було обрано **Node.js**, що дозволяє реалізувати неблокуючу обробку запитів. Разом з цим використовується **Express** — популярний фреймворк, що спрощує побудову REST API.

Організація маршрутів, логіки авторизації, доступу до БД реалізована з використанням таких модулів:

- express — базовий HTTP сервер;

- express-session — підтримка сесій;
- drizzle-orm — типобезпечний доступ до PostgreSQL;
- pg — драйвер PostgreSQL.

Файл routes.ts структурує API маршрути, зокрема:

```
router.get("/projects", authMiddleware, getProjectsHandler);
router.post("/apply", authMiddleware, submitApplicationHandler);
```

Клієнтська частина (Frontend)

Для реалізації клієнтської частини системи було обрано **React** у поєднанні з **Tailwind CSS** як основу інтерфейсного фреймворку. Це забезпечує компонентний підхід до побудови UI, адаптивність до будь-яких екранів та швидкість розробки. Уся логіка клієнта була реалізована мовою **TypeScript**, що дозволяє отримати переваги типобезпечності та зменшити ймовірність помилок на етапі компіляції.

Інтерфейс реалізує:

- подання заявок волонтерами;
- управління завданнями координаторами;
- перегляд проектів для донорів та волонтерам;
- модерацію — для модераторів.

Основні сторінки та їх логіка

- **HomePage (home-page.tsx)**

Реалізує головну вітальну сторінку з динамічною підтримкою темної/світлої теми та мультимовності. Користується бібліотекою next-themes для керування візуальною темою, а також next-i18next для локалізації інтерфейсу.

- **AuthPage (auth-page.tsx)**

Сторінка авторизації користувача. Вона визначає, чи авторизований користувач, і в залежності від ролі (волонтер, координатор, донор, адміністратор) автоматично перенаправляє його на відповідну панель керування.

- **CoordinatorDashboard (coordinator-dashboard.tsx)**

Інтерфейс координатора. Виводить список створених ним проектів та дозволяє керувати ними: переглядати заявки волонтерів, статуси модерації, змінювати інформацію, ініціювати завершення збору коштів тощо.

- **CreateProject (create-project.tsx)**

Форма для створення нового проекту координатором. Передбачено введення назви, опису, фінансової мети, дедлайну, категорії та початкового статусу. Дані валідуються перед надсиланням.

- **DonorDashboard (donor-dashboard.tsx)**

Доступна лише авторизованим донорам. Виводить список активних проектів, дозволяє здійснювати пожертви, переглядати звіти щодо витрат за підтриманими проектами.

- **DonatePage / DonatePageSimple (donate-page.tsx, donate-page-simple.tsx)**

Інтерфейси, що дозволяють внести фінансову допомогу на проект. Включають інформацію про цілі, зібрану суму, а також інтеграцію з платіжною системою через API.

- **AdminDashboard (admin-dashboard.tsx)**

Панель адміністратора-модератора, що дозволяє затверджувати або відхиляти проекти, переглядати звіти та блокувати користувачів у разі порушення правил.

- **ContactsPage та AboutPage (contacts-page.tsx, about-page.tsx)**

Інформаційні сторінки з описом платформи, контактною інформацією та загальними відомостями про функціонування проекту.

Додаткові особливості

- **Інтернаціоналізація (i18n):**

Здійснена через бібліотеку next-i18next, що дозволяє зберігати текстові ресурси в окремих JSON-файлах і автоматично обирати мову в залежності від локалі користувача.

- **Комунікація з backend:**

Усі сторінки взаємодіють із backend через REST API. Запити реалізовані за допомогою fetch або axios, із урахуванням авторизації через JWT.

База даних

Для зберігання структурованої інформації про користувачів, проекти, завдання, звіти тощо використано **PostgreSQL**. Це об’єктно-реляційна СУБД, що забезпечує:

- транзакційність і цілісність даних;
- підтримку JSON-полів (для складних структур);
- масштабованість при високих навантаженнях.

Схема БД реалізована через schema.ts, з прикладом опису таблиць:

```
export const donations = pgTable("donations", {
```

```
  id: serial("id").primaryKey(),
  amount: integer("amount").notNull(),
  userId: integer("user_id").references(() => users.id),
  projectId: integer("project_id").references(() => projects.id),
  createdAt: timestamp("created_at").defaultNow(),
});
```

Інтеграційні та сервісні модулі

Також у складі системи використовуються такі додаткові компоненти:

- **dotenv** — для безпечного зберігання конфігурації;
- **bcrypt** — для хешування паролів;
- **jsonwebtoken** — для створення та перевірки JWT-токенів;
- **CORS middleware** — для налаштування безпечної комунікації між клієнтом і сервером.

Розгортання

Для локального і продакшн-розгортання використовується **Vite** як збирач для фронтенду, а також можливість розгортання на:

- **Render**, **Railway** або **Vercel** (для front);
- **Docker** (за потреби для контейнеризації backend+DB).

Таким чином, обраний стек технологій:

- **React + Tailwind CSS** (Frontend)
- **Node.js + Express + Drizzle + PostgreSQL** (Backend + DB)
- **JWT / Sessions + bcrypt** (Security)
- **Vite + TypeScript** (DevTools)

є цілком придатним для реалізації сучасної, безпечної та масштабованої інформаційної системи для підтримки волонтерської діяльності. Усі компоненти логічно поєднані, підтримують модульність та відповідають вимогам сучасного веброзробництва.

3.4 Алгоритмізація та програмування програмних модулів

Алгоритмізація та програмування програмних модулів є важливим етапом розробки інформаційної системи підтримки волонтерської діяльності. На цьому етапі розробляються алгоритми основних бізнес-процесів системи та створюються програмні модулі, які забезпечують роботу цих алгоритмів.

Розглянемо детальніше реалізацію ключових алгоритмів, таких як реєстрація волонтера в системі, подання заявки волонтера на участь у проекті, а також алгоритм розгляду заявки координатором.

Алгоритм реєстрації волонтера в системі

Першим етапом роботи волонтера з системою є реєстрація. Блок-схема цього алгоритму наведена на рисунку 3.1. Процес розпочинається з введення волонтером своїх персональних даних (ім'я, електронна пошта, пароль). Далі здійснюється валідація введених даних на коректність та перевірка унікальності email у базі даних. У разі, якщо такі дані вже існують або введені некоректно, реєстрація не відбувається, і система генерує повідомлення про помилку. Якщо перевірка пройшла успішно, інформація зберігається у базі даних, після чого користувач отримує підтвердження про успішну реєстрацію.

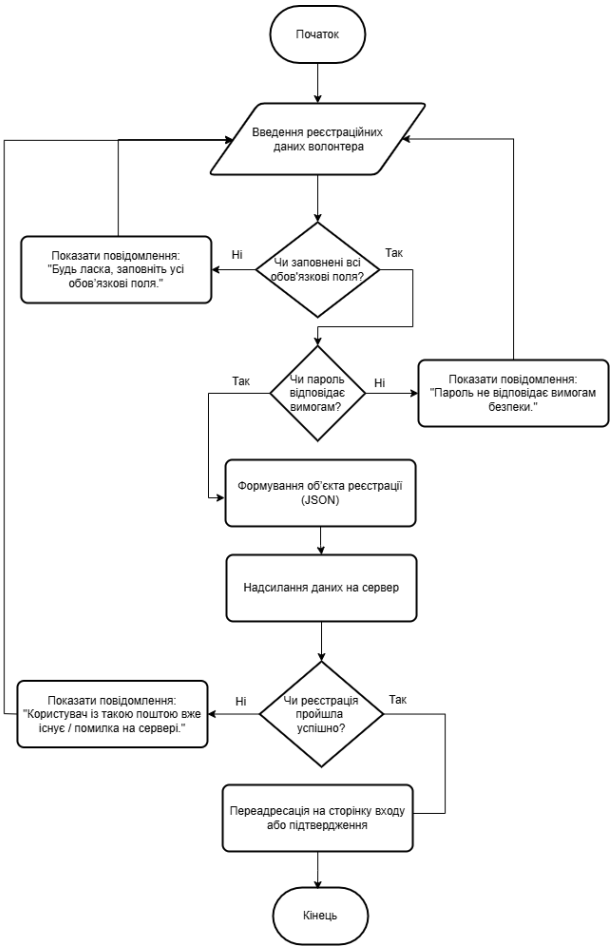

Рис. 3.1 – Блок-схема алгоритму реєстрації волонтера

На серверній частині цей процес реалізується наступним чином:

```
app.post('/register', async (req, res) => {
  const { name, email, password } = req.body;

  const existingUser = await db.getUserByEmail(email);
  if (existingUser) {
    return res.status(400).json({ message: "Користувач з таким email вже існує" });
  }

  const hashedPassword = await bcrypt.hash(password, 10);
  await db.insertUser({ name, email, password: hashedPassword });

  res.status(201).json({ message: "Реєстрація успішна" });
});
```

Алгоритм подання заявки волонтера на участь у проекті

Наступним важливим алгоритмом є подання заявки волонтера на участь у конкретному проекті. Блок-схема цього алгоритму зображена на рисунку 3.2. Волонтер обирає доступний проект та подає заявку через клієнтський інтерфейс. Після цього відбувається перевірка, чи волонтер вже не подав заявку на цей проект раніше. Якщо заявка є унікальною, вона зберігається у базі даних зі статусом «на розгляді». У разі виявлення дубліката волонтер отримує відповідне повідомлення.

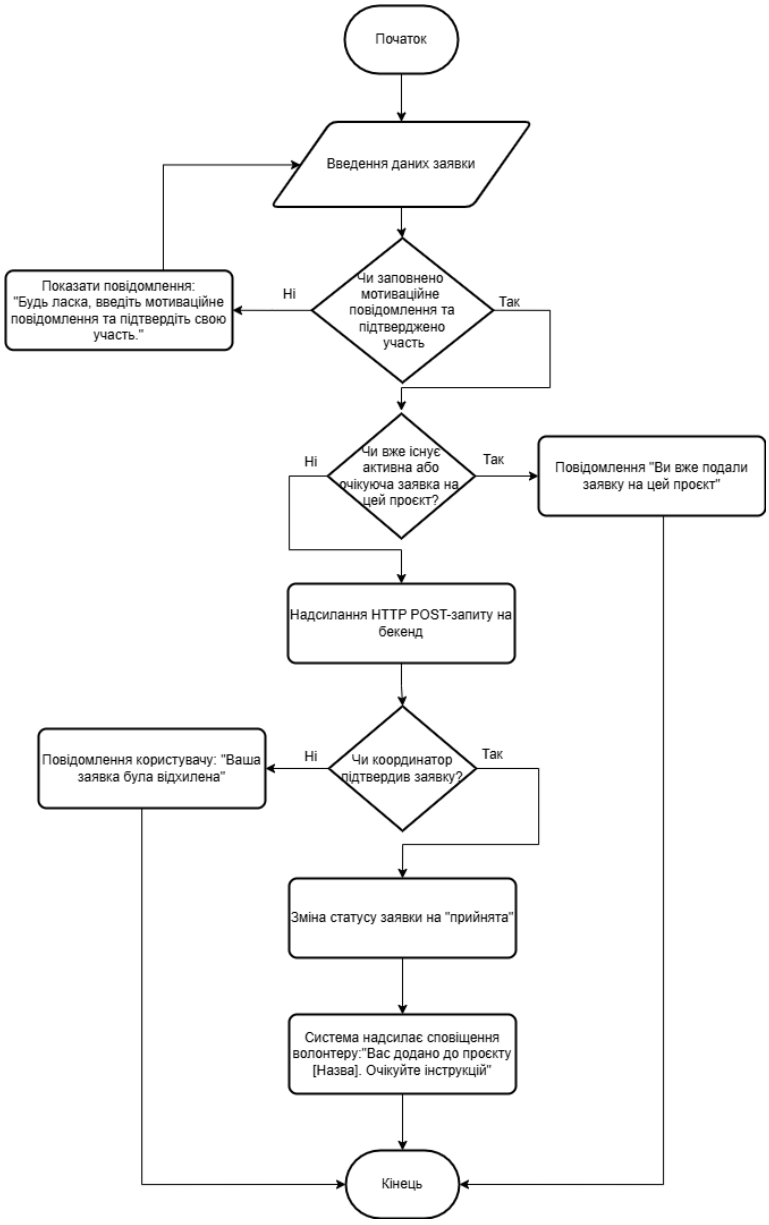

Рис. 3.2 – Блок-схема алгоритму подання заявки на участь у проекті

Реалізація цього алгоритму на сервері виглядає наступним чином:

```
app.post('/projects/:projectId/apply', async (req, res) => {
  const { projectId } = req.params;
  const { userId } = req.body;

  const existingApplication = await db.getApplicationByUserAndProject(userId,
projectId);
  if (existingApplication) {
    return res.status(400).json({ message: "Заявка на цей проект вже подана" });
  }

  await db.insertApplication({ userId, projectId, status: 'pending' });
  res.status(201).json({ message: "Заявка успішно подана" });
});
```

Алгоритм розгляду заявки координатором

Останній важливий алгоритм – це алгоритм розгляду заявки волонтера координатором. Цей процес також зображений на блок-схемі (рисунок 3.3). Координатор отримує перелік усіх заявок зі статусом «на розгляді» і приймає рішення про схвалення або відхилення кожної заявки. Після прийняття рішення статус заявки у базі даних оновлюється відповідно. Волонтер отримує повідомлення про рішення координатора.

Рис. 3.3 – Блок-схема алгоритму розгляду заявки координатором

Серверна частина алгоритму має таку реалізацію:

```
app.post('/applications/:id/review', async (req, res) => {
  const { id } = req.params;
  const { decision } = req.body; // decision може бути 'approved' або 'rejected'

  await db.updateApplicationStatus(id, decision);
  res.status(200).json({ message: `Заявка ${decision === 'approved' ? 'схвалена' :
'відхилена'}` });
});
```

Використання наведених вище алгоритмів та їх програмної реалізації дозволяє забезпечити ефективну роботу системи підтримки волонтерської діяльності, гарантувати коректність обробки заявок та уникнути помилок на кожному з етапів взаємодії користувачів із системою.

Висновки до розділу 3

Під час виконання третього розділу дипломної роботи було здійснено безпосередню розробку інформаційного та програмного забезпечення системи підтримки волонтерської діяльності. Для цього було спершу визначено систему управління інформаційною базою. В якості СУБД було обрано **PostgreSQL**, яка зарекомендувала себе як потужне, надійне та масштабоване рішення з відкритим вихідним кодом, здатне ефективно працювати з великими обсягами даних і забезпечувати високий рівень продуктивності та безпеки.

Наступним важливим кроком було розроблення інформаційної бази даних системи. Фізична модель даних створювалась на основі раніше розробленої логічної моделі, що гарантує її ефективність і відповідність бізнес-логіці проекту. Було створено схеми таблиць бази даних, зв’язки між ними та структури даних, необхідні для ефективного функціонування всієї системи. Реалізація інформаційної бази включає налаштування таблиць, полів та зв'язків, що підтверджує високу ступінь структурованості та відповідність розробленій логічній моделі.

При виборі інструментарію для створення прикладного програмного забезпечення проведено аналіз сучасних технологій, що можуть ефективно вирішувати поставлені задачі. На основі аналізу було прийнято рішення використати такі технології:

- **Frontend**: було використано React разом із фреймворком Vite і бібліотекою Tailwind CSS. Це забезпечило зручність реалізації адаптивного, швидкого та сучасного користувацького інтерфейсу.

- **Backend**: серверну частину системи було розроблено з використанням Node.js та Express. Цей стек дозволив створити продуктивний, асинхронний сервер, який ефективно обробляє запити користувачів та взаємодіє з базою даних.
- Для забезпечення взаємодії з базою даних було використано ORM-систему **Drizzle**, яка дозволила суттєво спростити написання запитів та підвищити стабільність і безпеку роботи з даними.

Останнім етапом цього розділу стало детальне розроблення та програмування ключових бізнес-процесів системи у вигляді алгоритмів та програмних модулів. Було наведено блок-схеми та код для таких процесів, як реєстрація волонтера, подання заявки на проект та розгляд заявки координатором. Ці реалізації забезпечують чітке розуміння послідовності дій користувачів, гарантують прозору логіку роботи та надійність виконання основних функціональних задач системи.

Таким чином, у третьому розділі роботи було забезпечено перехід від теоретичних моделей до практичної реалізації інформаційного та програмного забезпечення, що створює надійний фундамент для повноцінного функціонування системи підтримки волонтерської діяльності та її подальшого розширення.

4 РЕКОМЕНДАЦІЇ ЩОДО ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЇ СИСТЕМИ

4.1 Тестування системи

Тестування програмного забезпечення є важливим і необхідним етапом процесу розробки. Воно дозволяє переконатися, що система функціонує відповідно до визначених вимог і забезпечує належний рівень якості. Основна мета тестування — виявлення дефектів, недоліків, а також перевірка відповідності реалізованого функціоналу очікуванням користувачів та специфікації програмної системи.

Тестування — це процес перевірки поведінки програмного продукту, що дозволяє виявити помилки до моменту його використання в реальних умовах.

У процесі розробки програмного забезпечення інформаційної системи підтримки волонтерської діяльності було застосовано системне тестування з акцентом на ручний підхід. Системне тестування охоплює перевірку роботи програмного продукту в цілому, включаючи всі основні компоненти та їх взаємодії. Такий підхід дозволив комплексно перевірити основні сценарії використання системи та імітувати типові дії користувачів у реальному середовищі.

Під час тестування особлива увага приділялася таким функціональним компонентам:

1. Реєстрація користувачів

Тестування реєстрації користувачів включало перевірку наступних ситуацій:

- Реєстрація нового користувача з коректними даними.
- Спроба реєстрації з використанням вже існуючих облікових даних.

- Перевірка поведінки системи при введенні некоректних даних (наприклад, неповна інформація або некоректний формат).

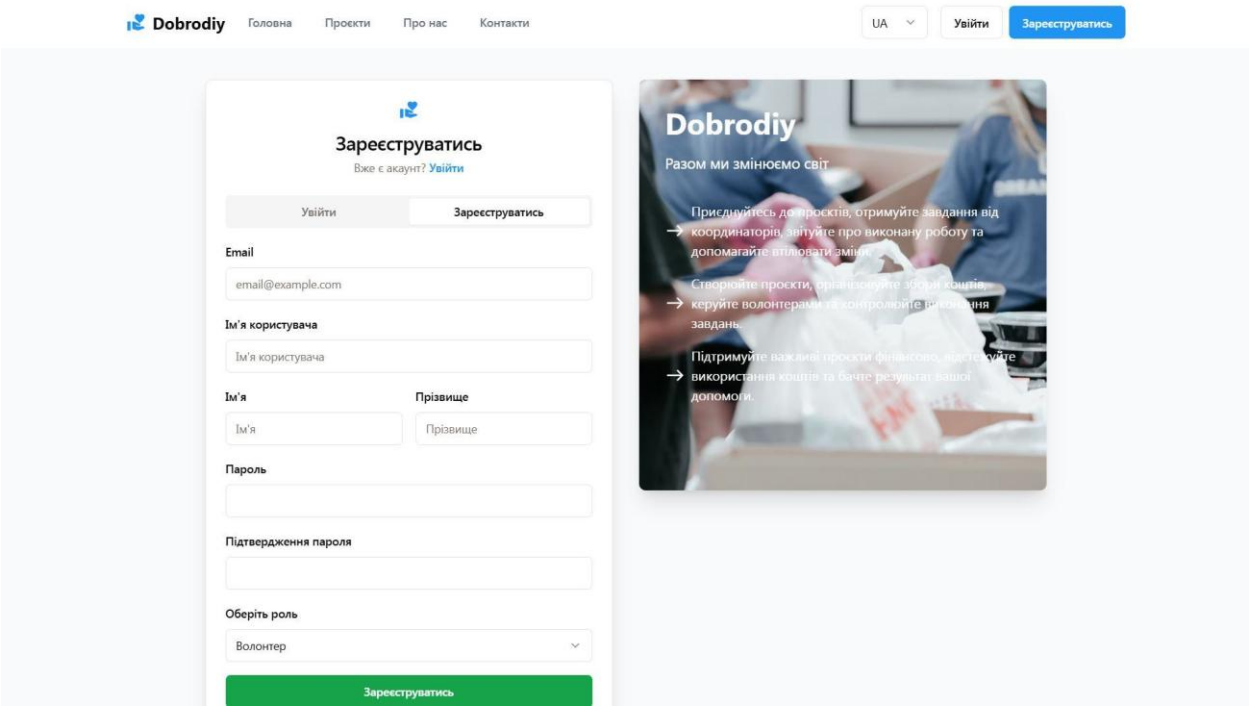

Рис.4.1 Сторінка реєстрації

Результати тестування показали, що система коректно обробляє як успішні сценарії реєстрації, так і коректно інформує користувача у разі помилок.

2. Авторизація користувачів

Авторизація тестувалася з метою перевірки процесу входу до системи зареєстрованими користувачами. Було перевірено:

- Вхід з правильними логіном і паролем.
- Спроби входу з невірними обліковими даними.
- Перевірка захисту від багаторазових невдалих спроб авторизації.

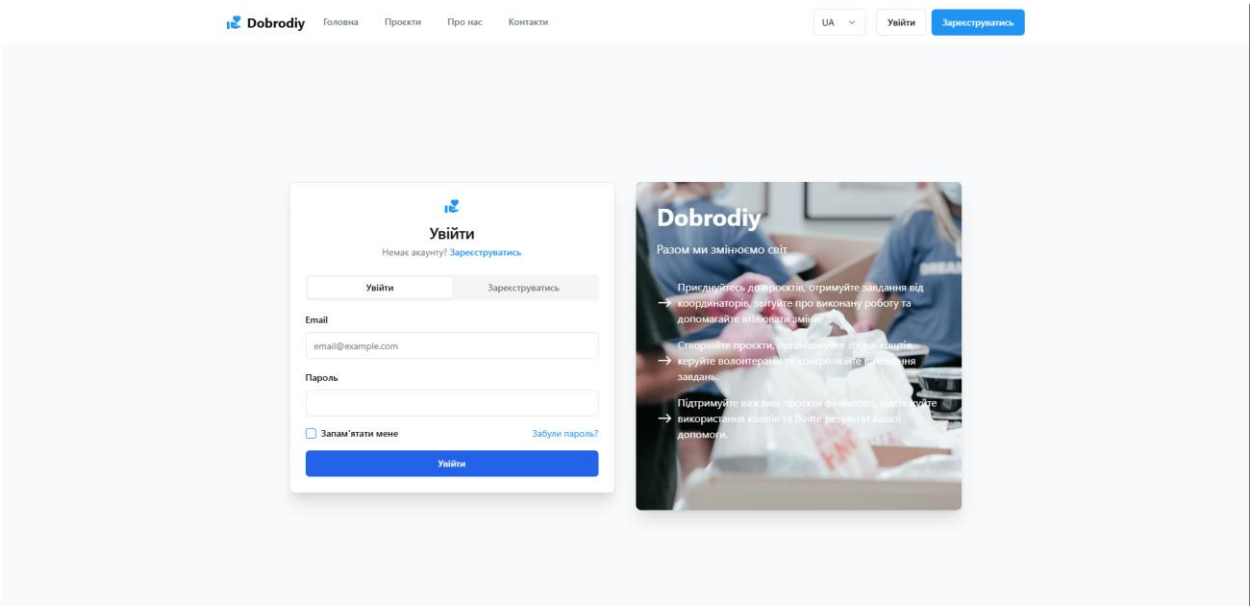

Рис.4.2 Сторінка реєстрації

Всі тестові сценарії успішно пройшли перевірку, підтвердивши стабільність і надійність процесу авторизації в системі.

4. Створення та модерація проектів

Було проведено тестування таких ключових процесів:

- Створення нового проекту координатором із заповненням необхідних даних.
- Перевірка коректності збереження даних у базі та відображення створеного проекту.
- Процес схвалення або відхилення проекту модератором, включаючи перевірку відповідних повідомлень користувачу.

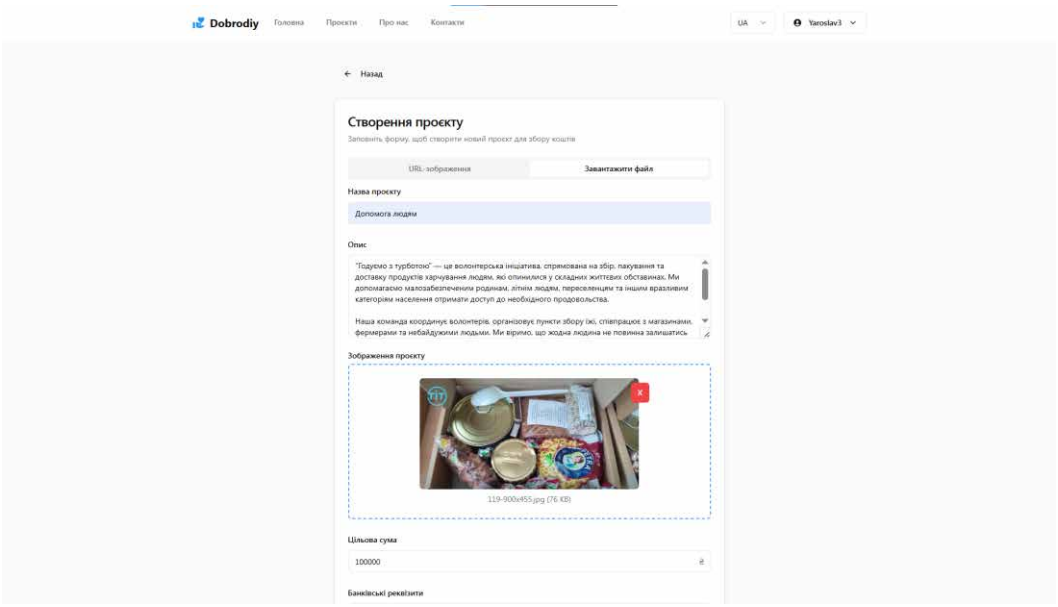

Рис 4.3 Сторінка створення проекту

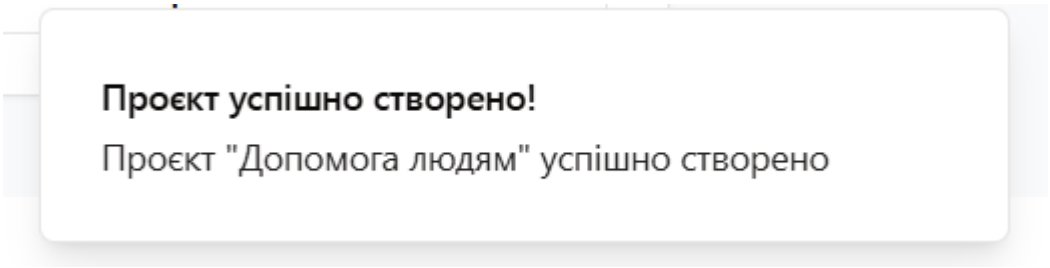

Рис.4.4 Повідомлення про успішне створення проекту

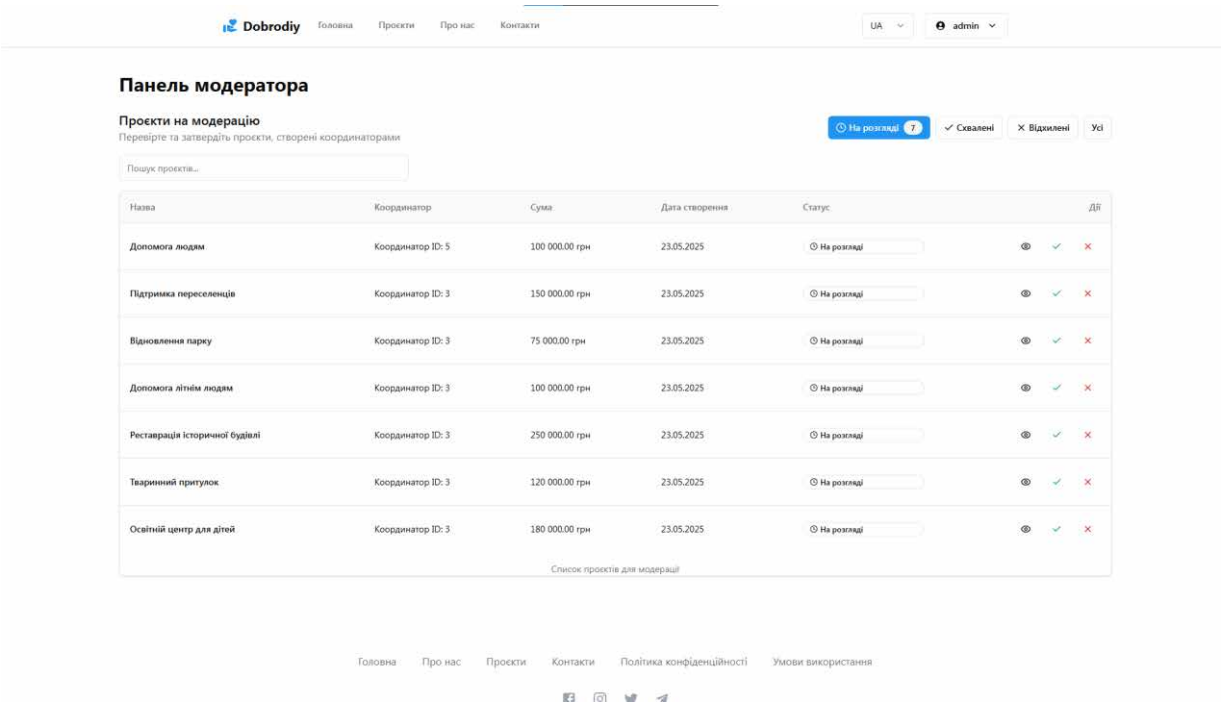

Рис 4.5 Сторінка модерації

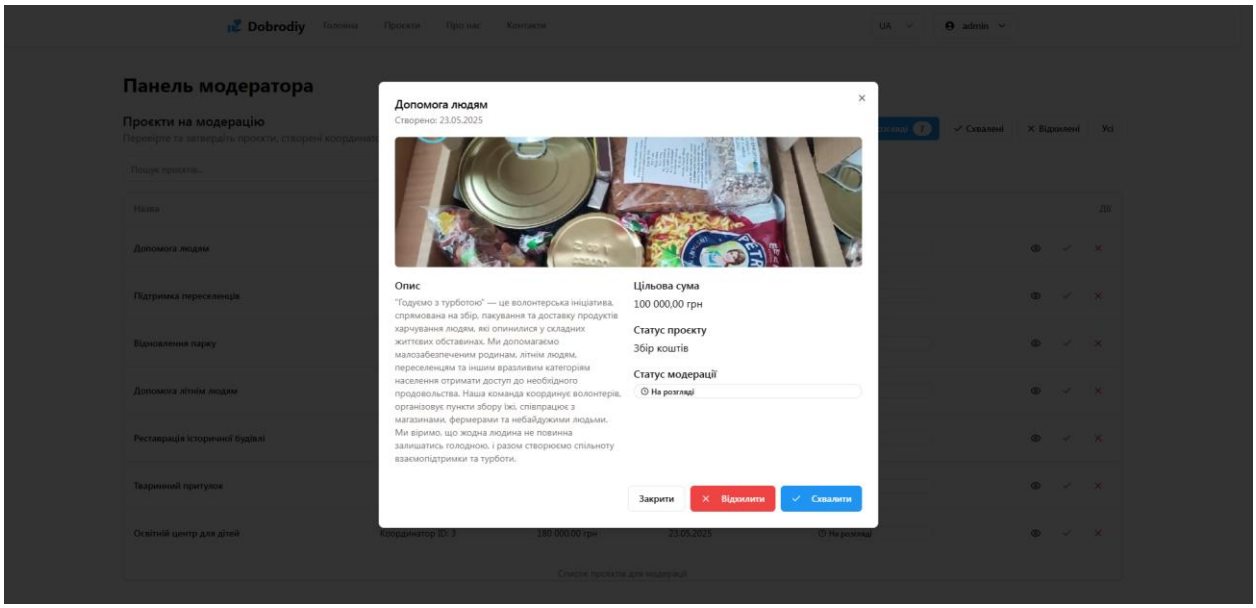

Рис.4.6 Сторінка прийняття або відхилення проекту

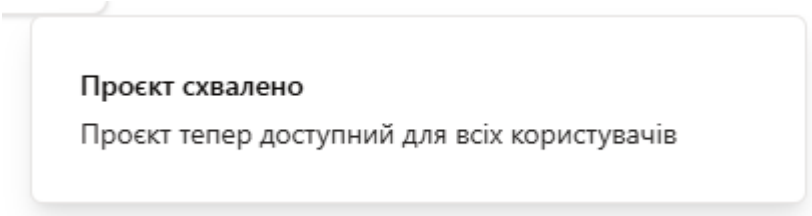

Рис. 4.7 Повідомлення про схвалення проекту модератором

Рис. 4.8 Вкладка схвалених проектів

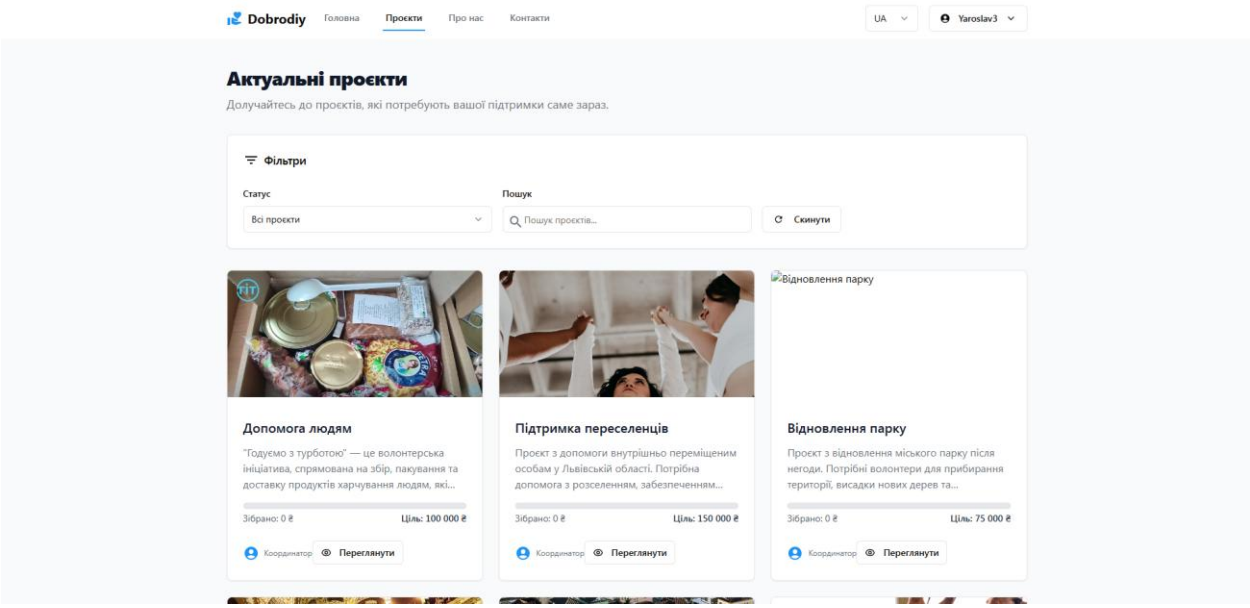

Рис. 4.9 Сторінка проектів з створеним координатором та схваленим модератором проект

Тестування підтвердило, що система стабільно та коректно реагує на всі вказані сценарії, належним чином інформуючи користувачів про статуси проектів.

5. Управління пожертвами

Було перевірено функціонал здійснення фінансових пожертв донором:

- Надсилання допомоги проекту.
- Відображення підтвердження після здійснення пожертви.
- Коректність відображення загальної суми пожертв для конкретного користувача.

Під час тестування було підтверджено, що всі транзакції та повідомлення працюють належним чином, забезпечуючи високий рівень довіри до фінансових операцій у системі.

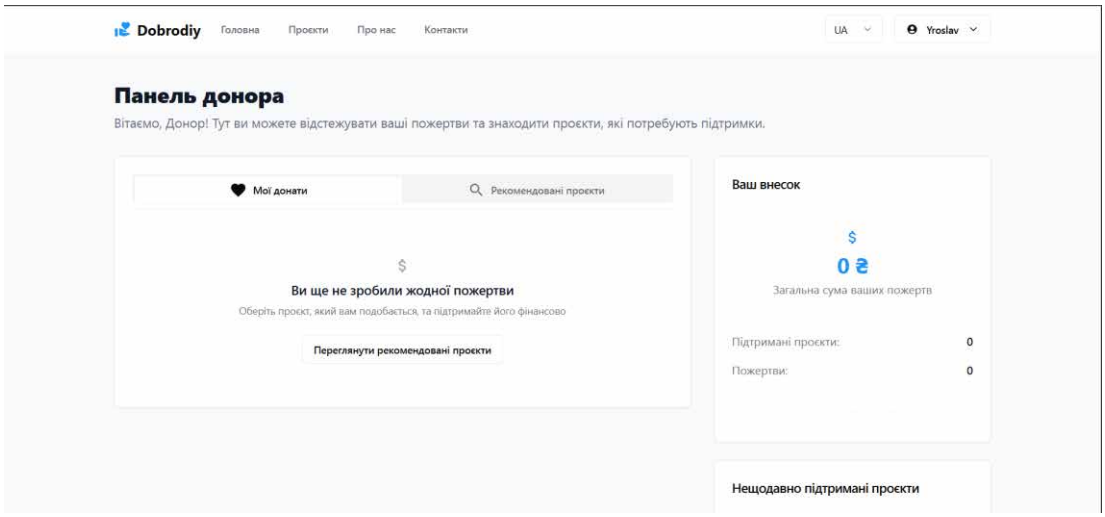

Рис. 4.10 Панель донора

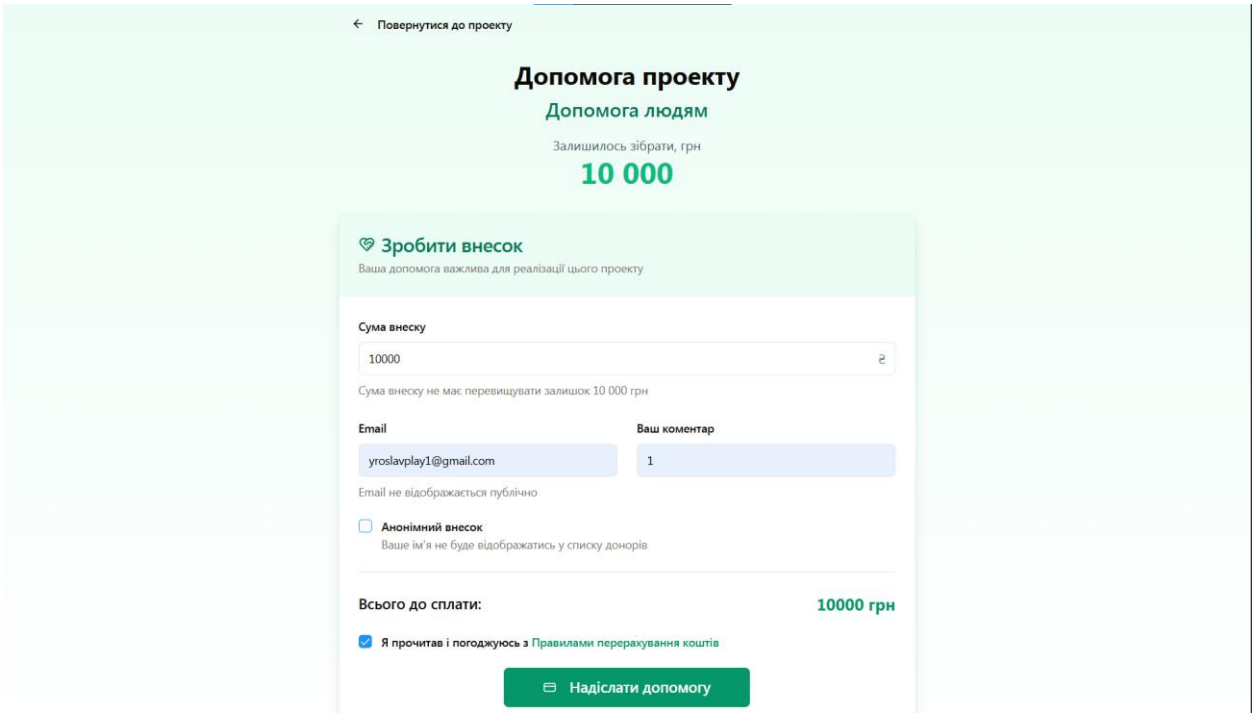

Рис. 4.11 Сторінка надсилання допомоги проекту

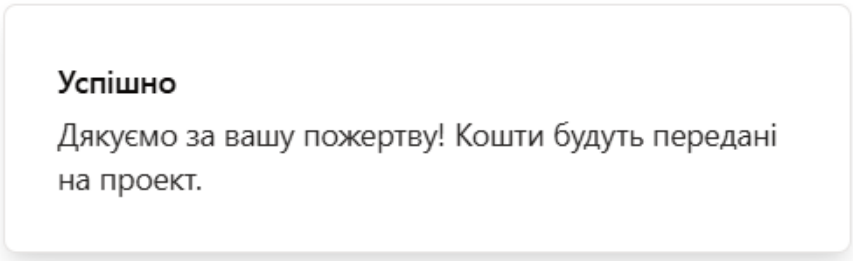

Рис. 4.12 Повідомлення про успішну пожертву коштів на проект

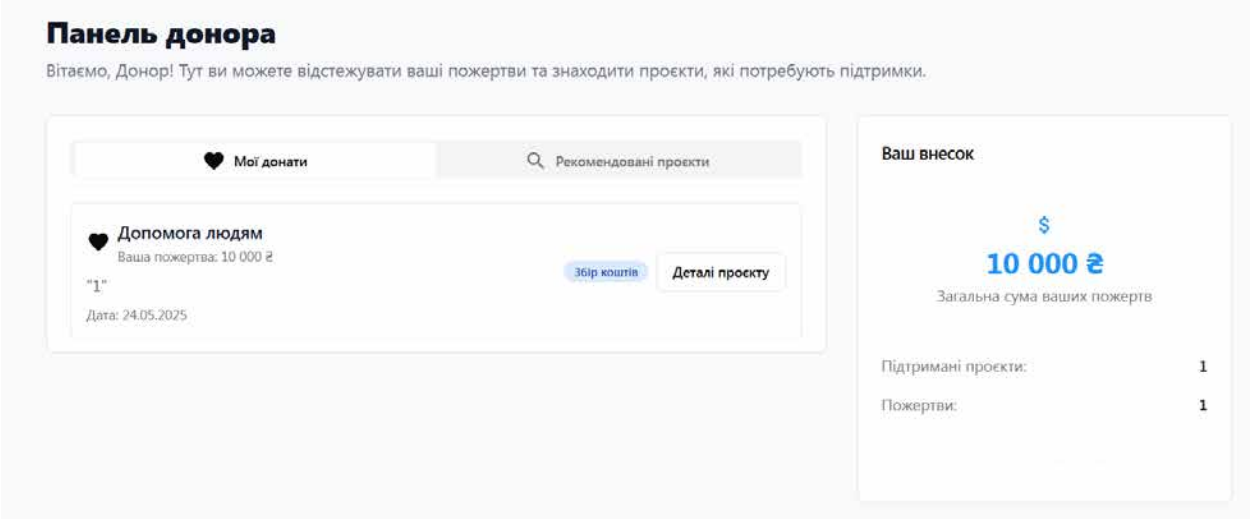

Рис. 4.13 Панель донора в якому відображається список проектів які донор підтримав та загальну суму потрачену донором на благодійність

5. Управління волонтерами

Тестування роботи з волонтерами включало:

- Подання заявки волонтером на участь у проекті.
- Схвалення або відхилення заявки координатором.
- Відповідне інформування волонтера про зміну статусу його заявки.

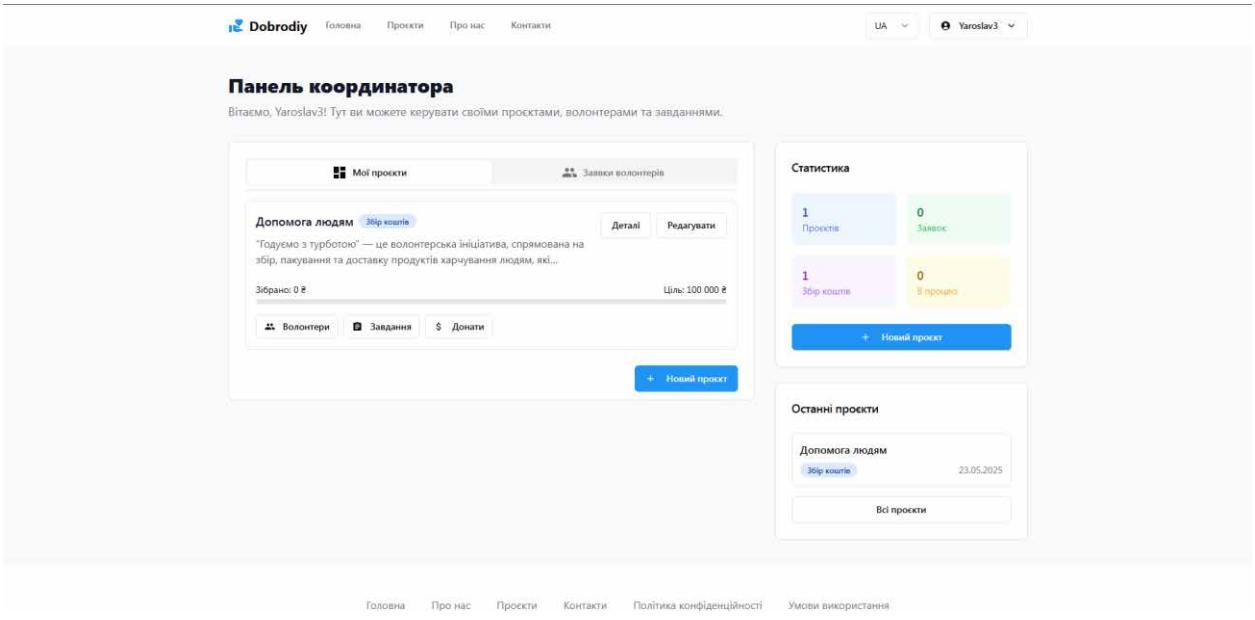

Рис. 4.14 Панель координатора

Система продемонструвала стабільність в усіх сценаріях, надаючи координаторам простий та зрозумілий інструмент управління заявками волонтерів.

Таким чином, проведене системне тестування підтвердило, що програмне забезпечення інформаційної системи підтримки волонтерської діяльності відповідає заявленим вимогам і готове до подальшого впровадження в експлуатацію

4.2 Вимоги до апаратного та програмного забезпечення

Ефективна робота інформаційної системи підтримки волонтерської діяльності потребує чіткого визначення апаратних і програмних вимог до всіх компонентів, які беруть участь у функціонуванні платформи. якості візуального представлення архітектури використовується **діаграма розгортання**, наведена на рис. 4.2. Вона дозволяє побачити, як компоненти системи розміщуються на фізичних та логічних вузлах, і як вони взаємодіють між собою.

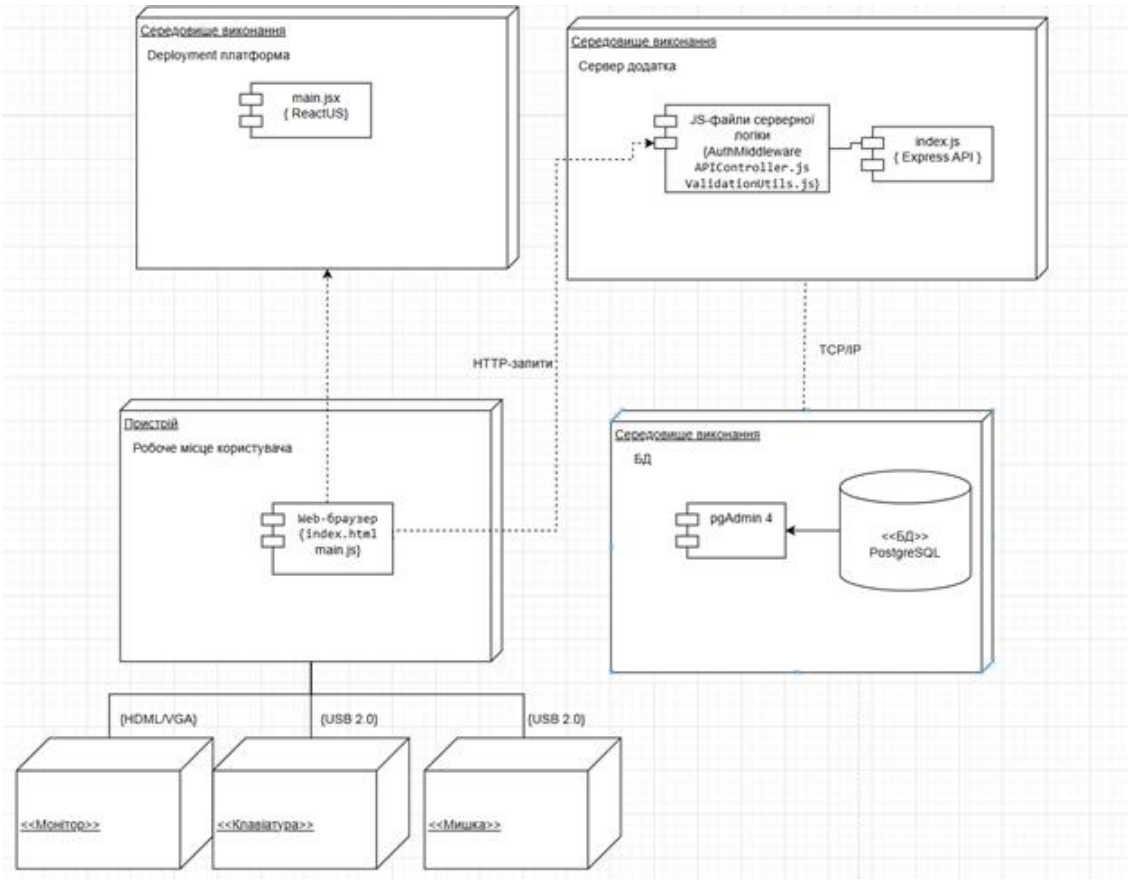

Рис. 4.2 – Діаграма розгортання системи

На діаграмі зображено чотири основні середовища виконання:

1. **Робоче місце користувача (Клієнтський пристрій)**:

- Взаємодія користувача з системою відбувається через **веб-браузер**, у якому завантажуються основні файли інтерфейсу (index.html, main.js).
- Фізично користувач працює за пристроєм, обладнаним стандартним набором периферії (монітор, клавіатура, миша), підключеною через USB/HDMI.

2. **Середовище виконання клієнтського інтерфейсу (Deployment платформа)**:

- Тут розгортається React-компонент main.jsx, що забезпечує функціонування фронтенду. Цей компонент обробляє події користувача та формує HTTP-запити до сервера.

3. **Сервер додатка**:

- Представлено файл index.js, який виконує роль входу до **серверної частини**, реалізованої на основі Express.js.
- Додаткові компоненти серверної логіки (AuthMiddleware, APIController.js, ValidationUtils.js) реалізують авторизацію, маршрутизацію запитів і валідацію даних.
- Сервер приймає HTTP-запити від клієнтів та надсилає відповіді після взаємодії з базою даних.

4. **Середовище виконання БД**:

- Як база даних використовується **PostgreSQL**, що керується через інтерфейс **pgAdmin 4**.
- Передача даних між сервером додатку та БД здійснюється по протоколу TCP/IP.

Архітектура систем

Система реалізована за класичною **трирівневою клієнт-серверною архітектурою**, що включає:

- Клієнтський пристрій (User Device)
- Веб-сервер (App Server)
- Сервер бази даних (DB Server)

Ця архітектура забезпечує модульність, зручне розділення відповідальностей, можливість горизонтального масштабування та гнучке адміністрування.

Вимоги до клієнтських пристроїв

Інформаційна система доступна через браузер, тому вимоги до клієнтських пристроїв є мінімальними:

Апаратне забезпечення:

- Процесор: від 2 ядер, 1.8 ГГц
- ОЗП: від 2 ГБ
- Екран: мінімум 1024x768 (рекомендовано 1366x768+)
- Інтернет-з’єднання: від 1 Мбіт/с

Програмне забезпечення:

- Операційна система: Windows 10+, macOS, Linux, Android 9+, iOS 13+
- Веб-браузери: останні версії Google Chrome, Mozilla Firefox, Microsoft Edge, Safari
- Підтримка JavaScript, HTML5, CSS3

Вимоги до веб-сервера

Веб-сервер виконує роль центрального обробника всіх запитів клієнтів. Він обробляє логіку, авторизацію, взаємодіє з базою даних, API та іншими службами.

Таблиця 4.1 — Апаратні вимоги до веб-сервера

Параметр	Мінімальні вимоги	Рекомендовані вимоги
Процесор	4 ядра, 2.5 ГГц	8 ядер, 3.0 ГГц+
ОЗП	8 ГБ	16 ГБ+
Накопичувач	SSD 256 ГБ	SSD 512 ГБ+

Мережа	1 Гбіт/с	10 Гбіт/с

Таблиця 4.2 — Програмні вимоги до веб-сервера

Компонент	**Мінімальні вимоги**	**Рекомендовані вимоги**
ОС	Ubuntu 20.04 / Windows SRV	Ubuntu 22.04 / Windows SRV 2022
Платформа	Node.js 18.x	Node.js 20+
Пакетний менеджер	npm 9.x або вище	npm 10.x
Сервіс запуску	PM2 / Docker	Docker + CI/CD (наприклад GitHub Actions)
Сертифікати SSL	Let’s Encrypt	Автоматичне оновлення SSL (Certbot/Cloudflare)
Фреймворк	Express + React (Vite)	Express + React + Tailwind + i18n

Вимоги до сервера бази даних

Сервер БД обробляє всі запити до структурованих даних: користувачі, проекти, заявки, пожертви, звіти. Надійність і продуктивність цієї частини системи — критично важливі.

Таблиця 4.3 — Апаратні вимоги до сервера бази даних

Параметр	**Мінімальні вимоги**	**Рекомендовані вимоги**
Процесор	4 ядра, 2.5 ГГц	8 ядер, 3.2 ГГц+
ОЗП	16 ГБ	32 ГБ+

Накопичувач	SSD 512 ГБ	SSD 1 ТБ+
Мережа	1 Гбіт/с	10 Гбіт/с

Таблиця 4.4 — Програмні вимоги до сервера бази даних

Компонент	**Мінімальні вимоги**	**Рекомендовані вимоги**
ОС	Ubuntu 20.04 / Windows SRV	Ubuntu 22.04 / Windows SRV 2022
СУБД	PostgreSQL 14.x	PostgreSQL 16.x
Резервування	pg_dump (ручне)	Автоматизовані бекапи + WAL
Безпека	TLS 1.2	TLS 1.3 + шифрування на рівні таблиць

Узагальнення

Розглянута архітектура системи дозволяє:

- ефективно обробляти паралельні запити від багатьох користувачів;
- масштабувати окремі компоненти (наприклад, перенести базу даних на окремий фізичний сервер або в хмару);
- забезпечити безпеку завдяки ізоляції рівнів (frontend — backend — database);
- використовувати сучасні інструменти автоматичного розгортання (CI/CD, Docker).

Завдяки оптимально підібраним вимогам до апаратного і програмного забезпечення, система підтримки волонтерської діяльності готова до

продуктивного, масштабованого та надійного розгортання як у локальному середовищі, так і в хмарних сервісах (AWS, GCP, Azure).

4.3 Склад інсталяційного пакету

У зв’язку з тим, що програмна система підтримки волонтерської діяльності реалізована як **веб-додаток** із клієнтською та серверною частинами, її інсталяційний пакет являє собою набір файлів, що дозволяє швидко розгорнути систему на хостингу або власному сервері. Система базується на стеку **Node.js + TypeScript + React + PostgreSQL**, тому її встановлення потребує налаштування відповідного середовища.

Структура інсталяційного пакету

Інсталяційний пакет складається з таких основних компонентів:

Компонент	**Призначення**
package.json, vite.config.ts	Конфігураційні файли Node.js-проекту, де вказані залежності та build-процеси
server/	Серверна частина: обробка запитів, логіка збереження, авторизація, API
client/	Клієнтська частина на React, зібрана з використанням Vite
schema.ts, db.ts	Опис структури бази даних та підключення до PostgreSQL
storage.ts	Основна логіка взаємодії з БД через ORM (Drizzle ORM)
seedData.ts	Скрипт для початкового заповнення бази тестовими даними
auth.ts	Модуль авторизації та аутентифікації

routes.ts	Основні маршрути API
dist/ або build/	Готовий для розгортання скомпільований фронтенд

Етапи встановлення системи

1. **Підготовка середовища сервера:**
   - Встановлення Node.js (рекомендовано v20+)
   - Встановлення PostgreSQL (версія 14 або вище)
   - За потреби – Docker для контейнерного розгортання

2. **Налаштування змінних середовища:**

У файлі .env або config.ts вказуються:

   - URL бази даних
   - JWT секрет (для авторизації)
   - Порт запуску

3. **Ініціалізація бази даних:**
   - Створення структури: npm run migrate
   - Заповнення початковими даними: npm run seed

4. **Збірка та запуск:**
   - Клієнтська частина: npm run build
   - Сервер: npm run start або pm2 start server/index.ts

Альтернативний варіант: контейнеризація

Для зручного розгортання система може бути зібрана в Docker-контейнер:

```
FROM node:20
WORKDIR /app
```

```
COPY . .
RUN npm install && npm run build
EXPOSE 3000
CMD ["node", "server/index.js"]
```

Також створюється docker-compose.yml для одночасного підняття сервера та PostgreSQL:

```
version: '3.8'

services:
  db:
    image: postgres:16
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: volunteer_system
    ports:
      - "5432:5432"

  app:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - db
```

Після розгортання

Після копіювання інсталяційного пакету на сервер:

- Здійснюється запуск backend-серверу (npm run start)
- Встановлюється reverse-proxy (Nginx) для перенаправлення на порт 3000
- Налаштовується SSL через Let's Encrypt
- Здійснюється підключення доменного імені
- Тестується доступність: https://your-domain.com

Таким чином, інсталяційний пакет системи включає повний набір файлів для швидкого розгортання як у локальному середовищі, так і на VPS або хмарній платформі. Завдяки сучасному стеку та модульній архітектурі, він дозволяє гнучко адаптувати платформу до потреб громади або волонтерської організації.

Висновки до розділу 4

У розділі 4 було детально розглянуто процес впровадження та експлуатації інформаційної системи підтримки волонтерської діяльності. Одним із ключових етапів стало **тестування системи**, яке дозволило виявити та усунути помилки на ранньому етапі, а також переконатися у коректній роботі всіх основних функціональних модулів. Тестування проводилося переважно вручну, із фіксацією результатів у тестових таблицях. Це забезпечило стабільність роботи системи та відповідність реалізованої логіки заявленим вимогам.

У підпункті 4.2 було сформульовано **вимоги до апаратного та програмного забезпечення**, необхідного для ефективного функціонування платформи. Зокрема, система розгортається за **архітектурою клієнт-сервер**, де користувач взаємодіє з інтерфейсом через браузер, а логіка та база даних обробляються на серверній стороні. Описані мінімальні та рекомендовані характеристики для клієнтських пристроїв, веб-сервера та сервера бази даних, що дозволяє планувати масштабування системи відповідно до потреб.

У підпункті 4.3 було визначено **склад інсталяційного пакету**, який охоплює всі необхідні компоненти для розгортання веб-застосунку: серверну частину (на Node.js), клієнтську частину (на React), конфігураційні файли, базу даних (PostgreSQL) та супровідну документацію. Описано процес розгортання як на фізичному сервері, так і у хмарному середовищі, з урахуванням налаштувань безпеки, збереження даних і масштабованості.

Таким чином, система готова до впровадження в реальних умовах. Вона відповідає вимогам до надійності, зручності використання та безпеки, а її архітектура забезпечує можливість подальшого розвитку та масштабування в залежності від потреб громади чи організації.

ВИСНОВКИ

У процесі виконання дипломної роботи було розроблено повнофункціональну **інформаційну систему підтримки волонтерської діяльності**, яка дозволяє ефективно координувати співпрацю між волонтерами, координаторами, донорами та адміністраторами. Основна мета — створення веб-платформи для реєстрації, подання, модерації, супроводу та фінансування соціальних ініціатив — була успішно досягнута.

Розробка системи охоплювала всі основні етапи створення сучасного програмного забезпечення: від аналізу предметної області до побудови модулів та розгортання рішення.

На **етапі системного аналізу** було досліджено функціональні потреби користувачів, визначено ролі, побудовано діаграми прецедентів, послідовності та активності, які чітко ілюструють логіку використання системи. Це дозволило сформувати повний перелік вимог до майбутнього програмного продукту.

На **етапі проектування** було створено логічну модель даних у вигляді **ER-діаграми**, побудовано **діаграми класів**, **кооперацій**, **пакетів** і **компонентів**. Усі ці артефакти наочно відображають структуру системи, взаємозв’язки між її частинами та забезпечують відповідність архітектури принципам модульності, масштабованості та підтримуваності.

Розробка інформаційної бази здійснювалася з використанням **PostgreSQL** як надійної реляційної СУБД. Для реалізації логіки застосунку використовувалися **Node.js** та **TypeScript** на серверній частині, а також **React + Tailwind CSS** на клієнтській. У розробці застосовувалися сучасні практики, зокрема **чистої архітектури**, **шаблони проектування**, ORM та валідація. Важливі процеси, такі як автентифікація, управління проектами, донати,

фільтрація та перевірка статусів, були реалізовані через окремі програмні модулі.

На етапі **впровадження та тестування** було проведено ручне функціональне тестування системи. Перевірка охопила ключові бізнес-процеси: реєстрацію, авторизацію, створення та модерацію проектів, підтримку донатів, що підтвердило коректну роботу усіх модулів.

Окремо були визначені **апаратні та програмні вимоги** до клієнтських пристроїв, веб-сервера та бази даних. Побудована **діаграма розміщення** допомогла візуалізувати архітектуру системи та визначити ролі кожного компонента. Також описано **склад інсталяційного пакета** для розгортання платформи на сервері або в хмарному середовищі.

В результаті розроблена система є повністю готовою до розгортання і відповідає усім поставленим функціональним та нефункціональним вимогам. Вона забезпечує інтуїтивну взаємодію, рольову модель доступу, безпечне управління проектами та відкриває широкі можливості для масштабування. Надалі систему можна доповнити такими модулями, як мобільний застосунок, інтеграція з платіжними шлюзами або аналітичними панелями для адміністративного управління.

Таким чином, розробка цього продукту стала не лише реалізацією складного технічного проекту, але й вагомим внеском у соціальну сферу, демонструючи, як IT-рішення можуть підтримувати волонтерську діяльність і посилювати громадську взаємодію.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Табунщик Г. В., Каплієнко Т. І., Петрова О. А. Проектування та моделювання програмного забезпечення сучасних інформаційних систем: навч. посіб. – Запоріжжя: Дике Поле, 2016. – 250 с. (bookshelf.erwin.com)

2. Мартін Р. С. Чиста архітектура. Мистецтво розробки програмного забезпечення / Роберт Сесіл Мартін. – Київ: Наш формат, 2020. – 336 с. (cleanarchitecturebook.com)

3. Яшина О. В., Земляна С. В., Мозгова І. В. Мова об’єктно-орієнтованого проектування UML: навч. посіб. – Дніпро: РВВ ДДТУ, 2003. – 58 с. (uml-diagrams.org)

4. Джон Дакетт. HTML і CSS. Розробка і дизайн веб-сайтів. – Індіанаполіс, 2011. – 478 с. (johnduckett.com)

5. Офіційна документація PostgreSQL – (postgresql.org/docs)

6. Офіційна документація React – (reactjs.org/docs)

7. Офіційна документація Tailwind CSS – (tailwindcss.com/docs)

8. Офіційна документація Vite – (vitejs.dev/guide)

9. Офіційна документація TypeScript – (typescriptlang.org/docs)

10. Авраменко А. С., Авраменко В. С., Косенюк Г. В. Тестування програмного забезпечення: навч. посіб. – Черкаси: ЧНУ ім. Б. Хмельницького, 2017. – 284 с.

11. Visual Studio Code: офіційний сайт редактора – (code.visualstudio.com/docs)

12. Козак О. Л. Аналіз вимог до програмного забезпечення: конспект лекцій. – Тернопіль, 2011. – 56 с.

13. ERWin Data Modeler. Release Notes. Version 2020 R1 – (bookshelf.erwin.com/2020R1)

Додаток А

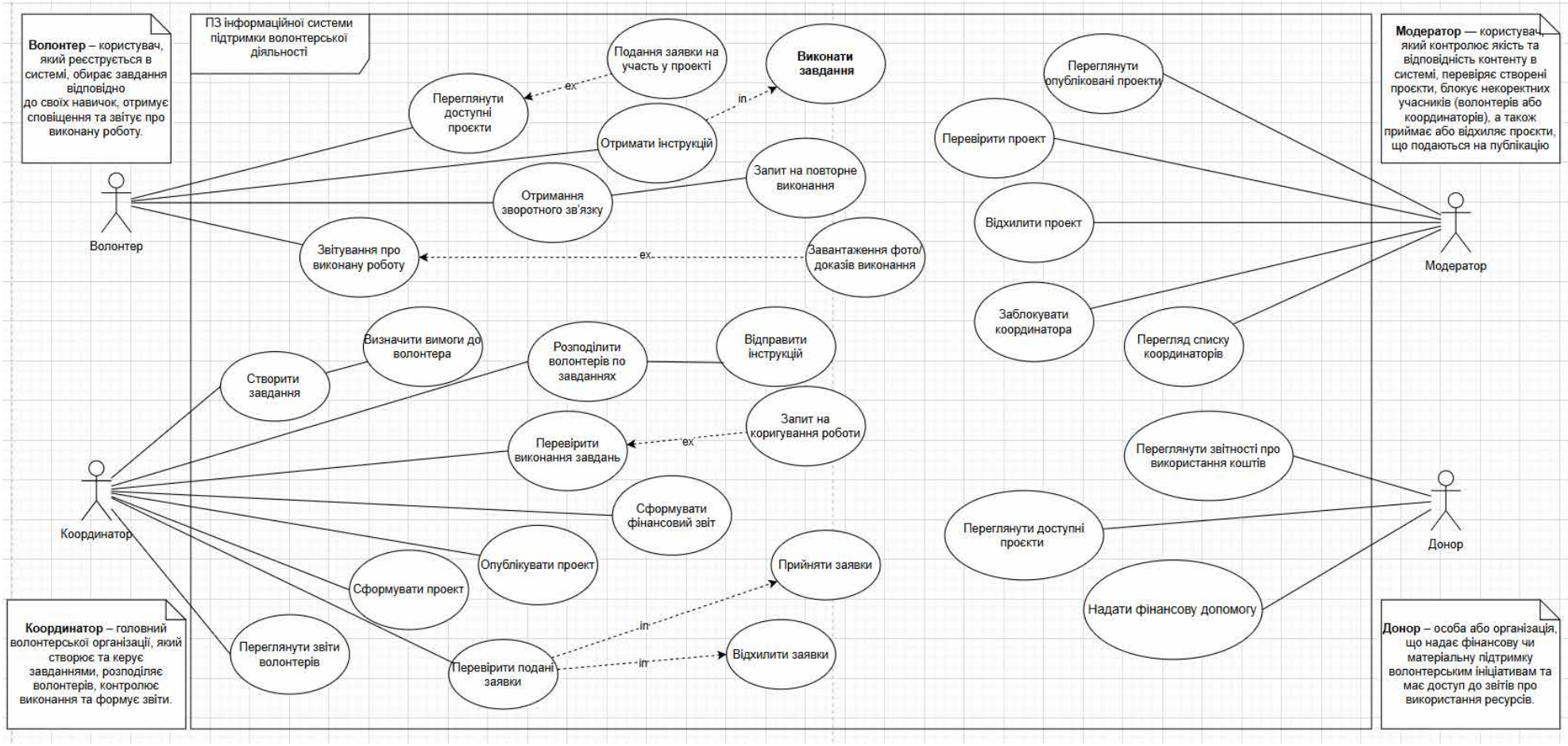

Рис. 1.4 Діаграма прецедентів UML

Додаток Б

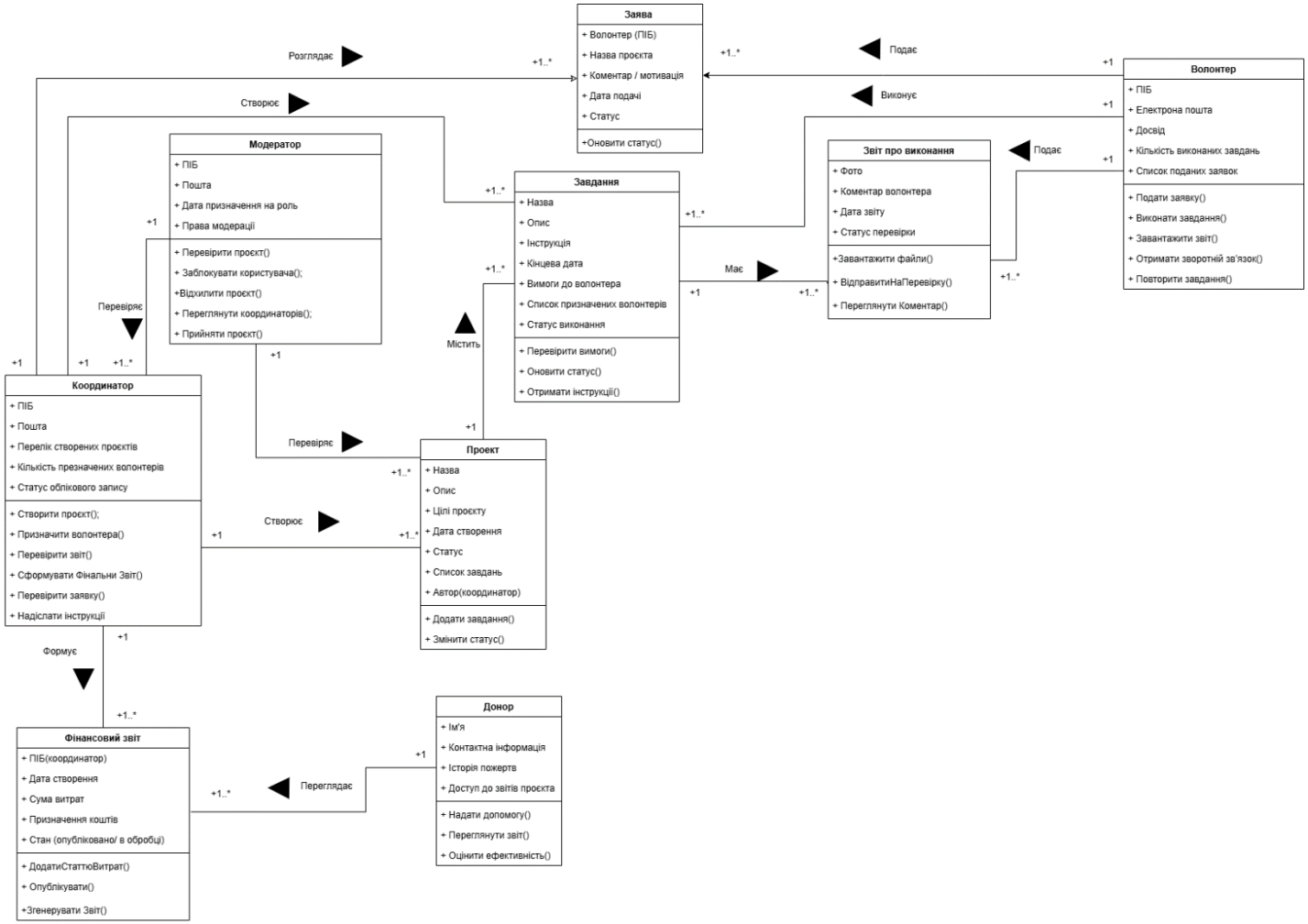

Рис. 2.2 – Діаграма класів інформаційної системи підтримки волонтерської діяльності

Додаток В

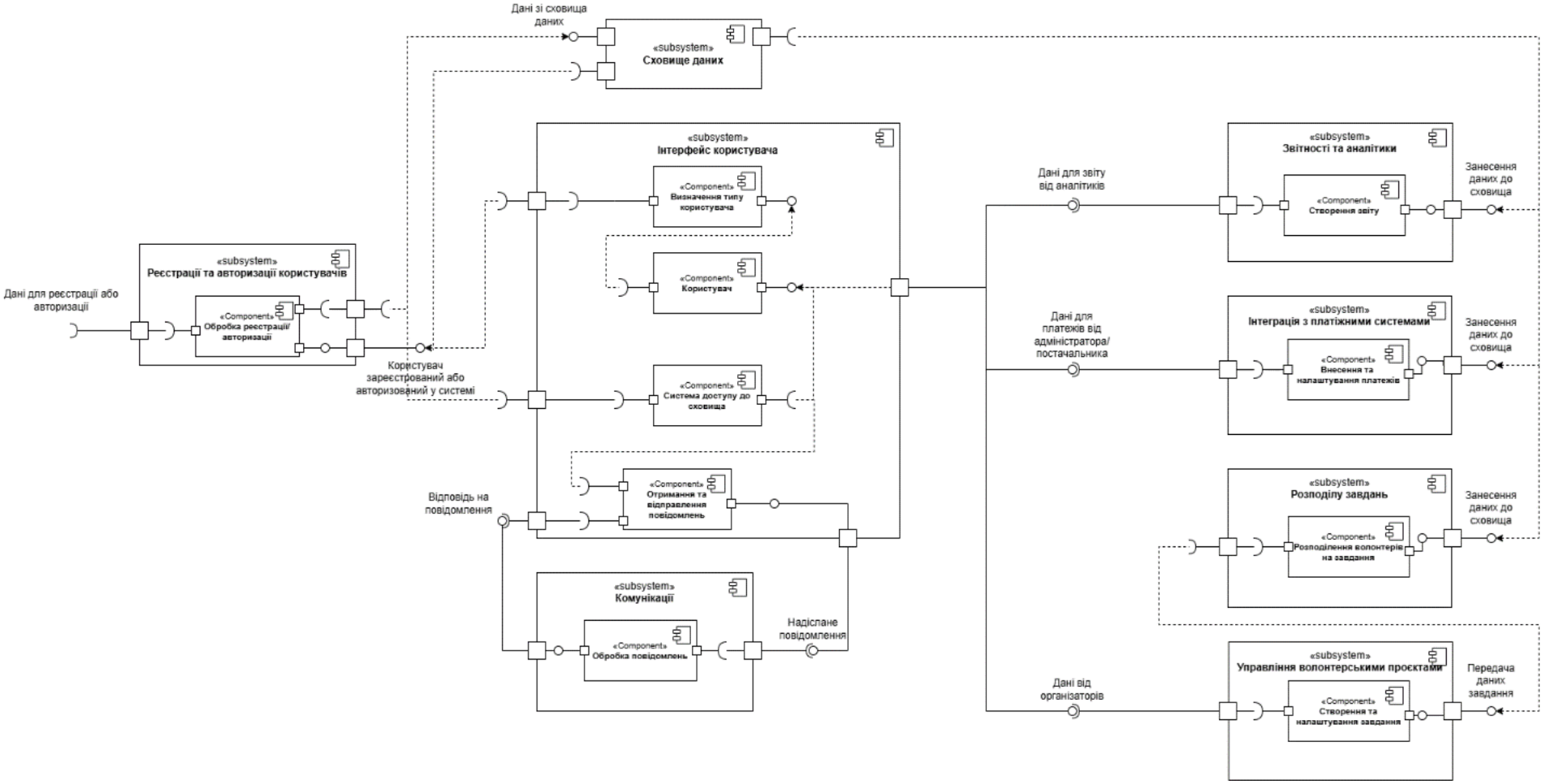

Рис. 2.8 – Діаграма компонентів інформаційної системи підтримки волонтерської діяльності

Додаток Г Код серверної частини

```
import { pgTable, text, serial, integer, boolean, timestamp, doublePrecision, pgEnum
} from "drizzle-orm/pg-core";
import { createInsertSchema, createSelectSchema } from "drizzle-zod";
import { z } from "zod";
import { relations } from "drizzle-orm";



// export type ModerationStatus = z.infer<typeof moderationStatusEnum>;

// Enum for user roles
export const userRoleEnum = pgEnum('user_role', ['volunteer', 'coordinator', 'donor',
'admin', 'moderator']);

// Enum for project status
export const projectStatusEnum = pgEnum('project_status', ['funding', 'in_progress',
'completed']);

// Enum for project moderation status
export const moderationStatusEnum = pgEnum('moderation_status', ['pending',
'approved', 'rejected']);


// export const moderationStatusEnum = z.enum([
//   "pending",    // на перевірці
//   "approved",   // схвалено
//   "rejected",   // відхилено
// ]);


// Enum for task status
export const taskStatusEnum = pgEnum('task_status', ['pending', 'in_progress',
'completed']);

// Enum for application status
export const applicationStatusEnum = pgEnum('application_status', ['pending',
'approved', 'rejected']);

// Users table
export const users = pgTable("users", {
  id: serial("id").primaryKey(),
  username: text("username").notNull().unique(),
  email: text("email").notNull().unique(),
  password: text("password").notNull(),
  role: userRoleEnum("role").notNull(),
  firstName: text("first_name"),
  lastName: text("last_name"),
  isVerified: boolean("is_verified").default(false).notNull(),
  isBlocked: boolean("is_blocked").default(false).notNull(),
  verificationToken: text("verification_token"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});

// Projects table
```

```
export const projects = pgTable("projects", {
  id: serial("id").primaryKey(),
  name: text("name").notNull(),
  description: text("description").notNull(),
  imageUrl: text("image_url"),
  targetAmount: doublePrecision("target_amount").notNull(),
  collectedAmount: doublePrecision("collected_amount").default(0).notNull(),
  status: projectStatusEnum("status").default('funding').notNull(),
  //
  moderationStatus:
moderationStatusEnum("moderation_status").default("pending").notNull(),
  isPublished: boolean("is_published").default(false).notNull(), // опціонально для
швидкої перевірки
  //
  coordinatorId: integer("coordinator_id").references(() => users.id).notNull(),
  bankDetails: text("bank_details"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
  updatedAt: timestamp("updated_at").defaultNow().notNull(),
});

// Project Moderation table
export const projectModerations = pgTable("project_moderations", {
  id: serial("id").primaryKey(),
  projectId: integer("project_id").references(() => projects.id).notNull(),
  status: moderationStatusEnum("status").default('pending').notNull(),
  comment: text("comment"),
  moderatorId: integer("moderator_id").references(() => users.id),
  createdAt: timestamp("created_at").defaultNow().notNull(),
  updatedAt: timestamp("updated_at").defaultNow().notNull(),
});

// Tasks table
export const tasks = pgTable("tasks", {
  id: serial("id").primaryKey(),
  title: text("title").notNull(),
  description: text("description").notNull(),
  projectId: integer("project_id").references(() => projects.id).notNull(),
  volunteerId: integer("volunteer_id").references(() => users.id),
  status: taskStatusEnum("status").default('pending').notNull(),
  deadline: timestamp("deadline"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
  updatedAt: timestamp("updated_at").defaultNow().notNull(),
});

// Reports table
export const reports = pgTable("reports", {
  id: serial("id").primaryKey(),
  taskId: integer("task_id").references(() => tasks.id).notNull(),
  imageUrl: text("image_url"),
  comment: text("comment"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});

// Volunteer applications
export const applications = pgTable("applications", {
  id: serial("id").primaryKey(),
```

```
  projectId: integer("project_id").references(() => projects.id).notNull(),
  volunteerId: integer("volunteer_id").references(() => users.id).notNull(),
  status: applicationStatusEnum("status").default('pending').notNull(),
  message: text("message"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});

// Donations table
export const donations = pgTable("donations", {
  id: serial("id").primaryKey(),
  projectId: integer("project_id").references(() => projects.id).notNull(),
  donorId: integer("donor_id").references(() => users.id),
  amount: doublePrecision("amount").notNull(),
  comment: text("comment"),
  createdAt: timestamp("created_at").defaultNow().notNull(),
});

// Define relations
export const usersRelations = relations(users, ({ many }) => ({
  projects: many(projects, { relationName: "coordinator_projects" }),
  tasks: many(tasks, { relationName: "volunteer_tasks" }),
  applications: many(applications, { relationName: "volunteer_applications" }),
  donations: many(donations, { relationName: "donor_donations" }),
}));

export const projectsRelations = relations(projects, ({ one, many }) => ({
  coordinator: one(users, {
    fields: [projects.coordinatorId],
    references: [users.id],
    relationName: "coordinator_projects"
  }),
  tasks: many(tasks),
  applications: many(applications),
  donations: many(donations),
}));

export const tasksRelations = relations(tasks, ({ one, many }) => ({
  project: one(projects, {
    fields: [tasks.projectId],
    references: [projects.id],
  }),
  volunteer: one(users, {
    fields: [tasks.volunteerId],
    references: [users.id],
    relationName: "volunteer_tasks"
  }),
  reports: many(reports),
}));

export const reportsRelations = relations(reports, ({ one }) => ({
  task: one(tasks, {
    fields: [reports.taskId],
    references: [tasks.id],
  }),
}));
```

```
export const applicationsRelations = relations(applications, ({ one }) => ({
  project: one(projects, {
    fields: [applications.projectId],
    references: [projects.id],
  }),
  volunteer: one(users, {
    fields: [applications.volunteerId],
    references: [users.id],
    relationName: "volunteer_applications"
  }),
}));

export const donationsRelations = relations(donations, ({ one }) => ({
  project: one(projects, {
    fields: [donations.projectId],
    references: [projects.id],
  }),
  donor: one(users, {
    fields: [donations.donorId],
    references: [users.id],
    relationName: "donor_donations"
  }),
}));

// Zod schemas for validation
export const insertUserSchema = createInsertSchema(users).omit({ id: true, createdAt:
true });
export const selectUserSchema = createSelectSchema(users);

export const insertProjectSchema = createInsertSchema(projects).omit({
  id: true,
  createdAt: true,
  updatedAt: true,
  collectedAmount: true,
  status: true,
  coordinatorId: true
});
export const selectProjectSchema = createSelectSchema(projects);

export const insertTaskSchema = createInsertSchema(tasks).omit({ id: true, createdAt:
true, updatedAt: true });
export const selectTaskSchema = createSelectSchema(tasks);

export const insertReportSchema = createInsertSchema(reports).omit({ id: true,
createdAt: true });
export const selectReportSchema = createSelectSchema(reports);

export const insertApplicationSchema = createInsertSchema(applications).omit({ id:
true, createdAt: true, status: true });
export const selectApplicationSchema = createSelectSchema(applications);

export const insertDonationSchema = createInsertSchema(donations).omit({ id: true,
createdAt: true });
export const selectDonationSchema = createSelectSchema(donations);

// Types
```

```
export type User = typeof users.$inferSelect;
export type InsertUser = z.infer<typeof insertUserSchema>;
export type SelectUser = z.infer<typeof selectUserSchema>;

export type Project = typeof projects.$inferSelect;
export type InsertProject = z.infer<typeof insertProjectSchema>;
export type SelectProject = z.infer<typeof selectProjectSchema>;

export type Task = typeof tasks.$inferSelect;
export type InsertTask = z.infer<typeof insertTaskSchema>;
export type SelectTask = z.infer<typeof selectTaskSchema>;

export type Report = typeof reports.$inferSelect;
export type InsertReport = z.infer<typeof insertReportSchema>;
export type SelectReport = z.infer<typeof selectReportSchema>;

export type Application = typeof applications.$inferSelect;
export type InsertApplication = z.infer<typeof insertApplicationSchema>;
export type SelectApplication = z.infer<typeof selectApplicationSchema>;

export type Donation = typeof donations.$inferSelect;
export type InsertDonation = z.infer<typeof insertDonationSchema>;
export type SelectDonation = z.infer<typeof selectDonationSchema>;
```

storage.ts

```
import {
  users,
  projects,
  tasks,
  reports,
  applications,
  donations,
  projectModerations,
  type User,
  type InsertUser,
  type Project,
  type InsertProject,
  type Task,
  type InsertTask,
  type Report,
  type InsertReport,
  type Application,
  type InsertApplication,
  type Donation,
  type InsertDonation
} from "@shared/schema";
import { db } from "./db";
import { eq, and, like, or, sql, gte, lte, isNull, desc } from "drizzle-orm";
import connectPg from "connect-pg-simple";
import session from "express-session";
import memorystore from "memorystore";

// Fix the typing issue with session store
const PostgresSessionStore = connectPg(session as any);
```

```
const MemoryStore = memorystore(session);

export interface IStorage {
  // User methods
  getUser(id: number): Promise<User | undefined>;
  getUserByUsername(username: string): Promise<User | undefined>;
  getUserByEmail(email: string): Promise<User | undefined>;
  getUserByVerificationToken(token: string): Promise<User | undefined>;
  createUser(user: InsertUser): Promise<User>;
  verifyUser(id: number): Promise<User>;
  blockUser(id: number): Promise<User>;
  getAllUsers(): Promise<User[]>;

  // Project methods
  getProjects(options?: { status?: string; search?: string; limit?: number; offset?: number }): Promise<Project[]>;
  getProjectById(id: number): Promise<Project | undefined>;
  getProjectsByCoordinatorId(coordinatorId: number): Promise<Project[]>;
  getProjectsForVolunteer(volunteerId: number): Promise<Project[]>;
  createProject(project: any): Promise<Project>;
  updateProjectStatus(id: number, status: string): Promise<Project>;
  updateProjectCollectedAmount(id: number, amount: number): Promise<Project>;
  updateProject(id: number, project: Partial<InsertProject>): Promise<Project>;
  deleteProject(id: number): Promise<void>;

  // Project Moderation methods
  getProjectModerations(projectId: number): Promise<any[]>;
  createProjectModeration(moderation: { projectId: number; status: string; comment: string | null; moderatorId:
number }): Promise<any>;

  // Task methods
  getTaskById(id: number): Promise<Task | undefined>;
  getTasksByProjectId(projectId: number): Promise<Task[]>;
  getTasksForVolunteer(volunteerId: number): Promise<Task[]>;
  createTask(task: InsertTask): Promise<Task>;
  assignTaskToVolunteer(id: number, volunteerId: number): Promise<Task>;
  updateTaskStatus(id: number, status: string): Promise<Task>;

  // Report methods
  getReportsByTaskId(taskId: number): Promise<Report[]>;
  createReport(report: InsertReport): Promise<Report>;

  // Application methods
  getApplicationById(id: number): Promise<Application | undefined>;
  getApplicationsByProjectId(projectId: number): Promise<Application[]>;
  getApplicationByVolunteerAndProject(volunteerId: number, projectId: number): Promise<Application |
undefined>;
  createApplication(application: InsertApplication): Promise<Application>;
  updateApplicationStatus(id: number, status: string): Promise<Application>;

  // Donation methods
  getDonationsByProjectId(projectId: number): Promise<Donation[]>;
  getDonationsByUserId(userId: number): Promise<Donation[]>;
  createDonation(donation: InsertDonation): Promise<Donation>;

  // Helper methods
  isVolunteerAssignedToProject(volunteerId: number, projectId: number): Promise<boolean>;
  getVolunteersByProjectId(projectId: number): Promise<User[]>;
```

```
  // Session store
  sessionStore: any;
}

export class DatabaseStorage implements IStorage {
  sessionStore: any;

  constructor() {
    this.sessionStore = new PostgresSessionStore({
      conObject: {
        connectionString: process.env.DATABASE_URL,
      },
      createTableIfMissing: true,
    });
  }

  // User methods
  async getUser(id: number): Promise<User | undefined> {
    const [user] = await db.select().from(users).where(eq(users.id, id));
    return user;
  }

  async getUserByUsername(username: string): Promise<User | undefined> {
    const [user] = await db.select().from(users).where(eq(users.username, username));
    return user;
  }

  async getUserByEmail(email: string): Promise<User | undefined> {
    const [user] = await db.select().from(users).where(eq(users.email, email));
    return user;
  }

  async getUserByVerificationToken(token: string): Promise<User | undefined> {
    const [user] = await db
      .select()
      .from(users)
      .where(eq(users.verificationToken, token));
    return user;
  }

  async createUser(insertUser: InsertUser): Promise<User> {
    const [user] = await db
      .insert(users)
      .values(insertUser)
      .returning();
    return user;
  }

  async verifyUser(id: number): Promise<User> {
    const [user] = await db
      .update(users)
      .set({ isVerified: true, verificationToken: null })
      .where(eq(users.id, id))
      .returning();
    return user;
  }
```

```
  async blockUser(id: number): Promise<User> {
    const [user] = await db
      .update(users)
      .set({
        isBlocked: true
      })
      .where(eq(users.id, id))
      .returning();
    return user;
  }

  async getAllUsers(): Promise<User[]> {
    return await db
      .select()
      .from(users)
      .orderBy(users.username);
  }

  // Project methods
  async getProjects(options?: { status?: string; search?: string; limit?: number; offset?: number }):
Promise<Project[]> {
    let query = db.select().from(projects);

    if (options) {
      if (options.status) {
        query = query.where(eq(projects.status, options.status));
      }

      if (options.search) {
        query = query.where(
          or(
            like(projects.name, `%${options.search}%`),
            like(projects.description, `%${options.search}%`)
          )
        );
      }

      if (options.limit) {
        query = query.limit(options.limit);
      }

      if (options.offset) {
        query = query.offset(options.offset);
      }
    }

    return await query.orderBy(desc(projects.createdAt));
  }

  async getProjectById(id: number): Promise<Project | undefined> {
    const [project] = await db.select().from(projects).where(eq(projects.id, id));
    return project;
  }

  async getProjectsByCoordinatorId(coordinatorId: number): Promise<Project[]> {
    return await db
```

```
    .select()
    .from(projects)
    .where(eq(projects.coordinatorId, coordinatorId))
    .orderBy(desc(projects.createdAt));
  }

  async getProjectsForVolunteer(volunteerId: number): Promise<Project[]> {
    // Get projects where volunteer's application was approved
    const approvedProjects = await db
      .select({
        project: projects
      })
      .from(applications)
      .innerJoin(projects, eq(applications.projectId, projects.id))
      .where(
        and(
          eq(applications.volunteerId, volunteerId),
          eq(applications.status, "approved")
        )
      );

    return approvedProjects.map(row => row.project);
  }

  async createProject(insertProject: InsertProject & { coordinatorId: number; status?: string; collectedAmount?:
number }): Promise<Project> {
    const [project] = await db
      .insert(projects)
      .values(insertProject)
      .returning();
    return project;
  }

  async updateProjectStatus(id: number, status: string): Promise<Project> {
    const [project] = await db
      .update(projects)
      .set({ status })
      .where(eq(projects.id, id))
      .returning();
    return project;
  }

  async updateProjectCollectedAmount(id: number, amount: number): Promise<Project> {
    const [project] = await db
      .update(projects)
      .set({
        collectedAmount: sql`${projects.collectedAmount} + ${amount}`
      })
      .where(eq(projects.id, id))
      .returning();
    return project;
  }

  async updateProject(id: number, projectData: Partial<InsertProject>): Promise<Project> {
    const [project] = await db
      .update(projects)
      .set(projectData)
```

```
    .where(eq(projects.id, id))
    .returning();
  return project;
}

async deleteProject(id: number): Promise<void> {
  await db
    .delete(projects)
    .where(eq(projects.id, id));
}

// Project Moderation Methods
async getProjectModerations(projectId: number): Promise<any[]> {
  try {
    // Перевіряємо спочатку, чи існує таблиця
    const moderations = await db
      .select()
      .from(projectModerations)
      .where(eq(projectModerations.projectId, projectId))
      .orderBy(desc(projectModerations.createdAt));

    return moderations;
  } catch (error) {
    console.error('Error getting project moderations:', error);
    // Якщо таблиця ще не створена, повертаємо порожній масив
    return [];
  }
}

async createProjectModeration(moderation: {
  projectId: number;
  status: string;
  comment: string | null;
  moderatorId: number
}): Promise<any> {
  try {
    // Перевіряємо спочатку, чи існує таблиця
    const [result] = await db
      .insert(projectModerations)
      .values(moderation)
      .returning();

    return result;
  } catch (error) {
    console.error('Error creating project moderation:', error);
    // Якщо таблиця ще не створена, повертаємо об'єкт як ніби запис був створений
    return {
      id: Date.now(),
      ...moderation,
      createdAt: new Date(),
      updatedAt: new Date()
    };
  }
}

// Task methods
async getTaskById(id: number): Promise<Task | undefined> {
```

```
  const [task] = await db.select().from(tasks).where(eq(tasks.id, id));
  return task;
}

async getTasksByProjectId(projectId: number): Promise<Task[]> {
  return await db
    .select()
    .from(tasks)
    .where(eq(tasks.projectId, projectId))
    .orderBy(desc(tasks.createdAt));
}

async getTasksForVolunteer(volunteerId: number): Promise<Task[]> {
  return await db
    .select()
    .from(tasks)
    .where(eq(tasks.volunteerId, volunteerId))
    .orderBy(desc(tasks.createdAt));
}

async createTask(insertTask: InsertTask): Promise<Task> {
  const [task] = await db
    .insert(tasks)
    .values(insertTask)
    .returning();
  return task;
}

async assignTaskToVolunteer(id: number, volunteerId: number): Promise<Task> {
  const [task] = await db
    .update(tasks)
    .set({ volunteerId, status: "in_progress" })
    .where(eq(tasks.id, id))
    .returning();
  return task;
}

async updateTaskStatus(id: number, status: string): Promise<Task> {
  const [task] = await db
    .update(tasks)
    .set({ status })
    .where(eq(tasks.id, id))
    .returning();
  return task;
}

// Report methods
async getReportsByTaskId(taskId: number): Promise<Report[]> {
  return await db
    .select()
    .from(reports)
    .where(eq(reports.taskId, taskId))
    .orderBy(desc(reports.createdAt));
}

async createReport(insertReport: InsertReport): Promise<Report> {
  const [report] = await db
```

```
    .insert(reports)
    .values(insertReport)
    .returning();
  return report;
 }

 // Application methods
 async getApplicationById(id: number): Promise<Application | undefined> {
  const [application] = await db.select().from(applications).where(eq(applications.id, id));
  return application;
 }

 async getApplicationsByProjectId(projectId: number): Promise<Application[]> {
  return await db
    .select()
    .from(applications)
    .where(eq(applications.projectId, projectId))
    .orderBy(desc(applications.createdAt));
 }

 async getApplicationByVolunteerAndProject(volunteerId: number, projectId: number): Promise<Application |
undefined> {
  const [application] = await db
    .select()
    .from(applications)
    .where(
      and(
        eq(applications.volunteerId, volunteerId),
        eq(applications.projectId, projectId)
      )
    );
  return application;
 }

 async createApplication(insertApplication: InsertApplication): Promise<Application> {
  const [application] = await db
    .insert(applications)
    .values(insertApplication)
    .returning();
  return application;
 }

 async updateApplicationStatus(id: number, status: string): Promise<Application> {
  const [application] = await db
    .update(applications)
    .set({ status })
    .where(eq(applications.id, id))
    .returning();
  return application;
 }

 // Donation methods
 async getDonationsByProjectId(projectId: number): Promise<Donation[]> {
  return await db
    .select()
    .from(donations)
    .where(eq(donations.projectId, projectId))
```

```
    .orderBy(desc(donations.createdAt));
  }

  async getDonationsByUserId(userId: number): Promise<Donation[]> {
    return await db
      .select()
      .from(donations)
      .where(eq(donations.donorId, userId))
      .orderBy(desc(donations.createdAt));
  }

  async createDonation(insertDonation: InsertDonation): Promise<Donation> {
    const [donation] = await db
      .insert(donations)
      .values(insertDonation)
      .returning();
    return donation;
  }

  // Helper methods
  async isVolunteerAssignedToProject(volunteerId: number, projectId: number): Promise<boolean> {
    const [application] = await db
      .select()
      .from(applications)
      .where(
        and(
          eq(applications.volunteerId, volunteerId),
          eq(applications.projectId, projectId),
          eq(applications.status, "approved")
        )
      );
    return !!application;
  }

  async getVolunteersByProjectId(projectId: number): Promise<User[]> {
    const volunteers = await db
      .select({
        user: users
      })
      .from(applications)
      .innerJoin(users, eq(applications.volunteerId, users.id))
      .where(
        and(
          eq(applications.projectId, projectId),
          eq(applications.status, "approved")
        )
      );

    return volunteers.map(row => row.user);
  }
}

export class MemStorage implements IStorage {
  private users: Map<number, User>;
  private projects: Map<number, Project>;
  private tasks: Map<number, Task>;
  private reports: Map<number, Report>;
```

```
private applications: Map<number, Application>;
private donations: Map<number, Donation>;
private projectModerations: Map<number, any>;

currentUserId: number;
currentProjectId: number;
currentTaskId: number;
currentReportId: number;
currentApplicationId: number;
currentDonationId: number;
currentModerationId: number;
sessionStore: any;

constructor() {
  this.users = new Map();
  this.projects = new Map();
  this.projectModerations = new Map();
  this.tasks = new Map();
  this.reports = new Map();
  this.applications = new Map();
  this.donations = new Map();

  this.currentUserId = 1;
  this.currentProjectId = 1;
  this.currentTaskId = 1;
  this.currentReportId = 1;
  this.currentApplicationId = 1;
  this.currentDonationId = 1;
  this.currentModerationId = 1;

  this.sessionStore = new MemoryStore({
    checkPeriod: 86400000,
  });
}

// User methods
async getUser(id: number): Promise<User | undefined> {
  return this.users.get(id);
}

async getUserByUsername(username: string): Promise<User | undefined> {
  return Array.from(this.users.values()).find(
    (user) => user.username === username,
  );
}

async getUserByEmail(email: string): Promise<User | undefined> {
  return Array.from(this.users.values()).find(
    (user) => user.email === email,
  );
}

async getUserByVerificationToken(token: string): Promise<User | undefined> {
  return Array.from(this.users.values()).find(
    (user) => user.verificationToken === token,
  );
}
```

```
  async createUser(insertUser: InsertUser): Promise<User> {
    const id = this.currentUserId++;
    const now = new Date();
    const user: User = {
      ...insertUser,
      id,
      isVerified: false,
      createdAt: now
    };
    this.users.set(id, user);
    return user;
  }

  async verifyUser(id: number): Promise<User> {
    const user = this.users.get(id);
    if (!user) {
      throw new Error("User not found");
    }

    const updatedUser = {
      ...user,
      isVerified: true,
      verificationToken: null
    };

    this.users.set(id, updatedUser);
    return updatedUser;
  }

  async blockUser(id: number): Promise<User> {
    const user = this.users.get(id);
    if (!user) {
      throw new Error("User not found");
    }

    const updatedUser = {
      ...user,
      isBlocked: true
    };

    this.users.set(id, updatedUser);
    return updatedUser;
  }

  async getAllUsers(): Promise<User[]> {
    return Array.from(this.users.values())
      .sort((a, b) => a.username.localeCompare(b.username));
  }

  // Project methods
  async getProjects(options?: { status?: string; search?: string; limit?: number; offset?: number }):
Promise<Project[]> {
    let result = Array.from(this.projects.values());

    if (options) {
      if (options.status) {
```

```
      result = result.filter(project => project.status === options.status);
    }

    if (options.search) {
      const searchLower = options.search.toLowerCase();
      result = result.filter(project =>
        project.name.toLowerCase().includes(searchLower) ||
        project.description.toLowerCase().includes(searchLower)
      );
    }

    // Sort by created date descending
    result.sort((a, b) => {
      return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
    });

    if (options.offset) {
      result = result.slice(options.offset);
    }

    if (options.limit) {
      result = result.slice(0, options.limit);
    }
  }

  return result;
}

async getProjectById(id: number): Promise<Project | undefined> {
  return this.projects.get(id);
}

async getProjectsByCoordinatorId(coordinatorId: number): Promise<Project[]> {
  return Array.from(this.projects.values())
    .filter(project => project.coordinatorId === coordinatorId)
    .sort((a, b) => {
      return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
    });
}

async getProjectsForVolunteer(volunteerId: number): Promise<Project[]> {
  // Get all approved applications for this volunteer
  const approvedApplications = Array.from(this.applications.values())
    .filter(app => app.volunteerId === volunteerId && app.status === "approved");

  // Get the corresponding projects
  return approvedApplications
    .map(app => this.projects.get(app.projectId))
    .filter((project): project is Project => !!project)
    .sort((a, b) => {
      return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
    });
}

async createProject(insertProject: InsertProject): Promise<Project> {
  const id = this.currentProjectId++;
  const now = new Date();
```

```
  const project: Project = {
    ...insertProject,
    id,
    status: "funding",
    collectedAmount: 0,
    createdAt: now,
    updatedAt: now
  };
  this.projects.set(id, project);
  return project;
}

async updateProjectStatus(id: number, status: string): Promise<Project> {
  const project = this.projects.get(id);
  if (!project) {
    throw new Error("Project not found");
  }

  const updatedProject = {
    ...project,
    status,
    updatedAt: new Date()
  };

  this.projects.set(id, updatedProject);
  return updatedProject;
}

async updateProjectCollectedAmount(id: number, amount: number): Promise<Project> {
  const project = this.projects.get(id);
  if (!project) {
    throw new Error("Project not found");
  }

  const updatedProject = {
    ...project,
    collectedAmount: project.collectedAmount + amount,
    updatedAt: new Date()
  };

  this.projects.set(id, updatedProject);
  return updatedProject;
}

async updateProject(id: number, projectData: Partial<InsertProject>): Promise<Project> {
  const project = this.projects.get(id);
  if (!project) {
    throw new Error("Project not found");
  }

  const updatedProject = {
    ...project,
    ...projectData,
    updatedAt: new Date()
  };

  this.projects.set(id, updatedProject);
```

```
  return updatedProject;
}

async deleteProject(id: number): Promise<void> {
  if (!this.projects.has(id)) {
    throw new Error("Project not found");
  }

  this.projects.delete(id);

  // Remove related tasks
  Array.from(this.tasks.entries())
    .filter(([_, task]) => task.projectId === id)
    .forEach(([taskId, _]) => this.tasks.delete(taskId));

  // Remove related applications
  Array.from(this.applications.entries())
    .filter(([_, app]) => app.projectId === id)
    .forEach(([appId, _]) => this.applications.delete(appId));

  // Remove related donations
  Array.from(this.donations.entries())
    .filter(([_, donation]) => donation.projectId === id)
    .forEach(([donationId, _]) => this.donations.delete(donationId));
}

// Task methods
async getTaskById(id: number): Promise<Task | undefined> {
  return this.tasks.get(id);
}

async getTasksByProjectId(projectId: number): Promise<Task[]> {
  return Array.from(this.tasks.values())
    .filter(task => task.projectId === projectId)
    .sort((a, b) => {
      return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
    });
}

async getTasksForVolunteer(volunteerId: number): Promise<Task[]> {
  return Array.from(this.tasks.values())
    .filter(task => task.volunteerId === volunteerId)
    .sort((a, b) => {
      return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
    });
}

async createTask(insertTask: InsertTask): Promise<Task> {
  const id = this.currentTaskId++;
  const now = new Date();
  const task: Task = {
    ...insertTask,
    id,
    status: "pending",
    createdAt: now,
    updatedAt: now
  };
```

```
  this.tasks.set(id, task);
  return task;
}

async assignTaskToVolunteer(id: number, volunteerId: number): Promise<Task> {
  const task = this.tasks.get(id);
  if (!task) {
    throw new Error("Task not found");
  }

  const updatedTask = {
    ...task,
    volunteerId,
    status: "in_progress",
    updatedAt: new Date()
  };

  this.tasks.set(id, updatedTask);
  return updatedTask;
}

async updateTaskStatus(id: number, status: string): Promise<Task> {
  const task = this.tasks.get(id);
  if (!task) {
    throw new Error("Task not found");
  }

  const updatedTask = {
    ...task,
    status,
    updatedAt: new Date()
  };

  this.tasks.set(id, updatedTask);
  return updatedTask;
}

// Report methods
async getReportsByTaskId(taskId: number): Promise<Report[]> {
  return Array.from(this.reports.values())
    .filter(report => report.taskId === taskId)
    .sort((a, b) => {
      return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
    });
}

async createReport(insertReport: InsertReport): Promise<Report> {
  const id = this.currentReportId++;
  const now = new Date();
  const report: Report = {
    ...insertReport,
    id,
    createdAt: now
  };
  this.reports.set(id, report);
  return report;
}
```

```
  // Application methods
  async getApplicationById(id: number): Promise<Application | undefined> {
    return this.applications.get(id);
  }

  async getApplicationsByProjectId(projectId: number): Promise<Application[]> {
    return Array.from(this.applications.values())
      .filter(application => application.projectId === projectId)
      .sort((a, b) => {
        return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
      });
  }

  async getApplicationByVolunteerAndProject(volunteerId: number, projectId: number): Promise<Application |
undefined> {
    return Array.from(this.applications.values())
      .find(app => app.volunteerId === volunteerId && app.projectId === projectId);
  }

  async createApplication(insertApplication: InsertApplication): Promise<Application> {
    const id = this.currentApplicationId++;
    const now = new Date();
    const application: Application = {
      ...insertApplication,
      id,
      status: "pending",
      createdAt: now
    };
    this.applications.set(id, application);
    return application;
  }

  async updateApplicationStatus(id: number, status: string): Promise<Application> {
    const application = this.applications.get(id);
    if (!application) {
      throw new Error("Application not found");
    }

    const updatedApplication = {
      ...application,
      status
    };

    this.applications.set(id, updatedApplication);
    return updatedApplication;
  }

  // Donation methods
  async getDonationsByProjectId(projectId: number): Promise<Donation[]> {
    return Array.from(this.donations.values())
      .filter(donation => donation.projectId === projectId)
      .sort((a, b) => {
        return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
      });
  }
```

```
  async getDonationsByUserId(userId: number): Promise<Donation[]> {
    return Array.from(this.donations.values())
      .filter(donation => donation.donorId === userId)
      .sort((a, b) => {
        return new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime();
      });
  }

  async createDonation(insertDonation: InsertDonation): Promise<Donation> {
    const id = this.currentDonationId++;
    const now = new Date();
    const donation: Donation = {
      ...insertDonation,
      id,
      createdAt: now
    };
    this.donations.set(id, donation);
    return donation;
  }

  // Helper methods
  async isVolunteerAssignedToProject(volunteerId: number, projectId: number): Promise<boolean> {
    return Array.from(this.applications.values())
      .some(app =>
        app.volunteerId === volunteerId &&
        app.projectId === projectId &&
        app.status === "approved"
      );
  }

  async getVolunteersByProjectId(projectId: number): Promise<User[]> {
    // Get all approved applications for this project
    const approvedApplications = Array.from(this.applications.values())
      .filter(app => app.projectId === projectId && app.status === "approved");

    // Get the corresponding volunteers
    return approvedApplications
      .map(app => this.users.get(app.volunteerId))
      .filter((user): user is User => !!user);
  }

  // Project Moderation methods
  async getProjectModerations(projectId: number): Promise<any[]> {
    return Array.from(this.projectModerations.values())
      .filter(moderation => moderation.projectId === projectId)
      .sort((a, b) => new Date(b.createdAt).getTime() - new Date(a.createdAt).getTime());
  }

  async createProjectModeration(moderation: { projectId: number; status: string; comment: string | null;
moderatorId: number }): Promise<any> {
    const id = this.currentModerationId++;
    const now = new Date();
    const newModeration = {
      id,
      projectId: moderation.projectId,
      status: moderation.status,
      comment: moderation.comment,
```

```
      moderatorId: moderation.moderatorId,
      createdAt: now,
      updatedAt: now
    };

    this.projectModerations.set(id, newModeration);
    return newModeration;
  }
}

// Switch to MemStorage for development to avoid database connectivity issues
export const storage = new MemStorage();
```