НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри
Артем ВОЛОКИТА
(підпис)
“__” _____2026 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на темі: Система управління операційною діяльністю волонтерських фондів у військовій сфері

Виконав : студент __4__ курсу, групи ІМ-22
(шифр групи)

Басараб Станіслав Анатолійович
(прізвище, ім’я, по батькові) (підпис)

Керівник старший викладач, доктор філософії комп’ютерної інженерії Міщенко Л. Д.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) старший викладач, доктор філософії комп’ютерної інженерії Міщенко Л. Д.
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент доцент кафедри ІПІ, к. т. н., Олійник Ю. О.
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2026 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____ Артем ВОЛОКИТА
(підпис)

“__” _____2026 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Басараба Станіслава Анатолійовича

1. Тема проєкту Система управління операційною діяльністю волонтерських фондів у військовій сфері
керівник проєкту Міщенко Л. Д., старший викладач, доктор філософії комп’ютерної інженерії
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 3 червня 2026 року №2055-с.

2. Термін здачі студентом закінченого проєкту *2 червня 2026 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області та огляд існуючих рішень.

Розділ 2. Обрані підходи та технології до розробки.

Розділ 3. Програмна реалізація системи

Розділ 4. Огляд розробленої системи

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структура системи (структурна схема), діаграма зв'язок-сутність (функціональна схема), алгоритм дій системи (принципова схема).

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Міщенко Л. Д		

7. Дата видачі завдання «06» лютого 2026 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*25.11.2025-10.02.2026*	
2.	*Вивчення та аналіз завдання*	*10.02.2026-12.03.2026*	
3.	*Розробка архітектури та загальної структури системи*	*10.03.2026-25.03.2026*	
4.	*Розробка структур окремих підсистем*	*25.03.2026-20.04.2026*	
5.	*Програмна реалізація системи*	*20.04.2026-17.05.2026*	
6.	*Оформлення пояснювальної записки*	*20.04.2026-04.06.2026*	
7.	*Захист програмного продукту*	*27.05.2026*	
8.	*Передзахист*	*02.06.2026*	
9.	*Захист*	*16.06.2026*	

Студент-дипломник _____ Станіслав БАСАРАБ
(підпис)

Керівник проєкту _____ Людмила МІЩЕНКО
(підпис)

АНОТАЦІЯ

У дипломному проєкті розроблено інформаційну систему управління операційною діяльністю волонтерських фондів, що здійснюють матеріально-технічне забезпечення Збройних Сил України. Проаналізовано предметну область та наявні програмні рішення класу управління взаємовідносинами з контрагентами, на основі чого сформовано вимоги до системи. Розроблена система забезпечує ведення бази контрагентів, супроводження запитів від військових підрозділів, облік благодійних надходжень, управління закупівлями й складськими запасами та формування звітності. Спроєктовано архітектуру програмного забезпечення та реалізовано клієнтську і серверну частини системи.

Ключові слова: волонтерська діяльність, військова сфера, управління взаємовідносинами з контрагентами, облік надходжень, NestJS, PostgreSQL, Prisma, React, TypeScript.

ANNOTATION

This diploma project presents the development of an information system for managing the operational activities of volunteer foundations that provide logistical support to the Armed Forces of Ukraine. The subject domain and existing customer relationship management software solutions were analysed, and on this basis the system requirements were defined. The developed system maintains a database of counterparties, handles requests from military units, records charitable contributions, manages procurement and warehouse stock, and generates reports. The software architecture was designed, and the client and server parts of the system were implemented.

Keywords: volunteer activity, military sphere, customer relationship management, contribution accounting, NestJS, PostgreSQL, Prisma, React, TypeScript.

Довідки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система управління операційною	4		
			діяльністю волонтерських фондів			
			у військовій сфері			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система управління операційною	98		
			діяльністю волонтерських фондів			
			у військовій сфері			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система управління операційною	1		
			діяльністю волонтерських фондів			
			у військовій сфері			
			Структура системи			
	А4	***ІАЛЦ.467200.005 Д2***	Система управління операційною	1		
			діяльністю волонтерських фондів			
			у військовій сфері			
			Діаграма сутність-зв’язок			
	А4	***ІАЛЦ.467200.006 Д3***	Система управління операційною	1		
			діяльністю волонтерських фондів			
			у військовій сфері			
			Алгоритм дій системи			
	А4	***ІАЛЦ.467200.007 Д4***	Система управління операційною	11		
			діяльністю волонтерських фондів			
			у військовій сфері			
			Текст програмного коду			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	

ІАЛЦ.467200.001 ОА

Розроб	Басараб С.А.			*Система управління операційною діяльністю волонтерських фондів у військовій сфері*	Літ.	Аркуш	Аркушів
Перев	Міщенко Л.Д.					1	1
				Опис альбому	***КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-22***		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система управління операційною діяльністю волонтерських фондів у військовій сфері*»

Київ – 2026

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Це технічне завдання поширюється на розроблення інформаційної системи управління операційною діяльністю волонтерських фондів у військовій сфері, а також на подальший супровід та вдосконалення зазначеної системи.

Областю застосування цієї системи є автоматизація управлінських та обліково-аналітичних процесів волонтерських організацій, які здійснюють діяльність із забезпечення матеріально-технічних потреб Збройних Сил України, зокрема благодійних фондів, громадських організацій та волонтерських ініціатив відповідного спрямування.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розроблення даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», яке було затверджено факультетом «Інформатики та обчислювальної техніки» кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою дипломного проєкту є підвищення ефективності операційної діяльності волонтерських фондів військового спрямування шляхом розроблення спеціалізованої веб-платформи, що забезпечує централізоване ведення обліку контрагентів, супроводження запитів від військових підрозділів, контроль руху благодійних надходжень та матеріальних цінностей, а також формування відповідної звітності.

Призначення системи полягає в автоматизації типових процесів волонтерської діяльності, зменшенні частки ручної праці під час обробки запитів

та обліку ресурсів, а також забезпеченні прозорості діяльності фонду перед благодійниками.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розроблення дипломного проєкту слугують офіційні технічні документації застосовуваних програмних засобів, тематичні наукові публікації, фахова література з питань управління взаємовідносинами з контрагентами та проєктування інформаційних систем, актуальні інтернет-ресурси з досліджуваної галузі, а також консультації з представниками волонтерських організацій.

5 ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до розробленого продукту

- Зручний та інтуїтивно зрозумілий інтерфейс користувача
- Ведення єдиної бази контрагентів (благодійників-партнерів, представників військових підрозділів, постачальників)
- Реєстрація та супроводження запитів на матеріально-технічне забезпечення
- Облік благодійних надходжень у грошовій та натуральній формах
- Управління закупівлями та контроль виконання замовлень
- Ведення обліку складських запасів та руху матеріальних цінностей
- Розмежування прав доступу користувачів на основі рольової моделі
- Формування звітності щодо діяльності фонду в різних розрізах
- Механізм пошуку та фільтрації даних за основними сутностями системи
- Документація прикладного програмного інтерфейсу для забезпечення можливості подальших інтеграцій

5.2 Вимоги до програмного забезпечення

- Операційна система: Windows 10/11, macOS 10.15 або новіша, Linux з графічним інтерфейсом
- Встановлений веб-браузер (Google Chrome, Mozilla Firefox, Microsoft Edge або Safari актуальних версій) з увімкненою підтримкою JavaScript
- Активне підключення до мережі Інтернет

5.3. Вимоги до апаратної частини

- Оперативна пам'ять обсягом не менше 4 ГБ
- Мережеве підключення зі швидкістю не менше 10 Мбіт/с
- Роздільна здатність екрана не менше 1280×720 пікселів

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	25.11.2025-10.02.2026
Вивчення та аналіз завдання	10.02.2026-12.03.2026
Розробка архітектури та загальної структури системи	10.03.2026-25.03.2026
Розробка структур окремих частин системи	25.03.2026-20.04.2026
Програмна реалізація системи	20.04.2026-17.05.2026
Виправлення помилок	30.04.2026-02.06.2026
Оформлення пояснювальної записки	20.05.2026-04.06.2026

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система управління операційною діяльністю волонтерських фондів у військовій сфері*»

Київ – 2026

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ACID	(Atomicity, Consistency, Isolation, Durability) Атомарність, узгодженість, ізольованість, довговічність – властивості транзакцій
API	(Application Programming Interface) Прикладний програмний інтерфейс
CRM	(Customer Relationship Management) Управління взаємовідносинами з контрагентами
HTML	(HyperText Markup Language) Мова розмітки гіпертексту
HTTP	(HyperText Transfer Protocol) Протокол передавання гіпертексту
JSON	(JavaScript Object Notation) Формат текстового обміну даними на основі мови JavaScript
JSONB	(JSON Binary) Бінарне представлення JSON у системі керування базами даних PostgreSQL
JSX	(JavaScript XML) Синтаксичне розширення мови JavaScript
ORM	(Object-Relational Mapping) Об’єктно-реляційне відображення
QCi	(Quality Competitive Index) Модель оцінки якості управління контрагентами
RBAC	(Role-Based Access Control) Рольова модель розмежування доступу
REST	(Representational State Transfer) Архітектурний стиль побудови розподілених застосунків
SaaS	(Software as a Service) Програмне забезпечення як послуга

SOAP	(Simple Object Access Protocol) Простий протокол доступу до об’єктів
SOQL	(Salesforce Object Query Language) Мова запитів до об’єктів платформи Salesforce
SPA	(Single Page Application) Односторінковий веб-застосунок
SQL	(Structured Query Language) Структурована мова запитів
SSR	(Server-Side Rendering) Серверний рендеринг сторінок
UI	(User Interface) Користувацький інтерфейс
URL	(Uniform Resource Locator) Уніфікований локатор ресурсу
СКБД	Система керування базами даних

ВСТУП

Повномасштабна збройна агресія Російської Федерації проти України, що розпочалася у 2022 році, обумовила виникнення безпрецедентного волонтерського руху, спрямованого на матеріально-технічне забезпечення Збройних Сил України. За даними Опендатабот, лише у 2024 році українці переказали благодійним фондам понад 17 мільярдів гривень, при цьому більша частина цих коштів спрямовується саме на потреби армії [1].

Незважаючи на значні обсяги діяльності, переважна частина волонтерських організацій продовжує використовувати для управління своїми процесами універсальні програмні засоби, не призначені для відповідних завдань: електронні таблиці, месенджери, поштові сервіси та розрізнені нотатки. Такий підхід призводить до втрати інформації, дублювання роботи, ускладнення обліку та додаткових витрат часу.

Областю застосування розроблюваної системи є інформаційне забезпечення операційної діяльності волонтерських фондів. Розроблена веб-платформа призначена для централізованого ведення обліку контрагентів, супроводження запитів на матеріально-технічне забезпечення, обліку благодійних надходжень, управління закупівлями і складськими запасами та формування відповідної звітності.

Актуальність роботи обумовлена відсутністю спеціалізованих програмних рішень для управління операційною діяльністю волонтерських фондів військового спрямування, а також потребою в підвищенні ефективності та прозорості зазначеної діяльності в умовах триваючого збройного протистояння. У цьому дипломному проєкті досліджено особливості предметної області та розроблено інформаційну систему, що забезпечує комплексну автоматизацію основних процесів волонтерської діяльності.

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 Постановка проблеми та її актуальність

Незалежно від організаційно-правової форми, операційна діяльність волонтерського фонду військового спрямування передбачає обробку гетерогенних потоків даних, що надходять із численних незалежних джерел та підлягають збереженню, узгодженню й аналітичній обробці. До основних типів інформаційних об'єктів, з якими оперує фонд, належать:

1) запити від військових підрозділів – структурована інформація про потребу в матеріально-технічних засобах (номенклатура, кількість, технічні характеристики, термін, координатор з боку підрозділу) разом з неструктурованими додатками (фотоматеріали, голосові повідомлення, документи);
2) записи про закупівлі та логістичні операції – інформація про постачальників, замовлення, рахунки, акти приймання-передавання, маршрути доставки;
3) відомості про контрагентів – благодійників, представників військових підрозділів, постачальників;
4) записи про публічну звітність – зведені звіти за визначені періоди для розміщення на офіційних ресурсах фонду.

Між зазначеними об'єктами існує розгалужена система зв'язків: одне надходження може фінансувати кілька закупівель, одна закупівля – задовольняти кілька запитів, один запит – потребувати кількох постачань. Сукупність зв'язків формує орієнтований граф операцій, обхід якого є основою для побудови аудиторського сліду та формування звітності.

На практиці переважна більшість волонтерських фондів продовжує здійснювати управління операційною діяльністю за допомогою сукупності розрізнених програмних засобів загального призначення, не призначених для побудови цілісної інформаційної системи: електронних таблиць (Google Sheets, Microsoft Excel) для обліку операцій, месенджерів (Telegram, Signal, Viber) для приймання запитів та комунікації, хмарних сховищ (Google Drive, Dropbox) для зберігання документів і фотоматеріалів, а також засобів нотування (Notion, Trello) для внутрішньої координації роботи команди.

З точки зору інженерного аналізу, описана конфігурація становить сукупність ізольованих сховищ даних, що не утворюють єдиної інформаційної системи. Такий підхід має низку істотних недоліків, які можна класифікувати за категоріями.

Перший із недоліків полягає у відсутності єдиної моделі даних. Кожний із застосовуваних засобів оперує власною моделлю представлення інформації: електронні таблиці зберігають дані у вигляді денормалізованих відношень із плоскою структурою, месенджери – у вигляді хронологічних послідовностей повідомлень, сервіси форм – у вигляді сукупностей пар «поле – значення». Інформація про один інформаційний об'єкт (наприклад, запит від військового підрозділу) одночасно існує у кількох незалежних формах представлення, узгодженість між якими не забезпечується автоматизованими засобами та повністю покладається на людину.

Другий недолік полягає у відсутності контролю цілісності даних. Універсальні інструменти не реалізовують механізмів перевірки цілісності зв'язків між інформаційними об'єктами (referential integrity), обмежень на унікальність ключових атрибутів та транзакційності операцій. Як наслідок, поширеними є помилки на кшталт фіксації закупівлі без посилання на запит, що її ініціював, або відсутності зв'язку між надходженням та закупівлею, що ним була профінансована. Це унеможливлює формальну побудову аудиторського сліду операцій.

Окрему групу недоліків становлять інтеграційні розриви між джерелами даних. Перенесення даних між застосовуваними сервісами здійснюється переважно ручним способом. Кожен такий перехід є джерелом помилок та втрати інформації, а також становить істотну частку трудовитрат координатора фонду.

Суттєвою проблемою є також обмежена масштабованість застосовуваних засобів. Електронні таблиці виявляють істотне зниження продуктивності за обсягів даних, що накопичуються у міру тривалої роботи фонду. Месенджери як засіб приймання запитів не передбачають їх категоризації, фільтрації та пошуку за структурованими атрибутами, що ускладнює роботу з історією запитів у міру її накопичення.

Останнім із розглянутих недоліків є відсутність механізмів розмежування доступу. Універсальні засоби не передбачають побудови рольової моделі доступу (RBAC) до даних із розмежуванням повноважень координаторів, бухгалтерів, аудиторів та публічних користувачів. Як наслідок, або всі учасники команди мають доступ до всіх даних, або доступ обмежується ручним способом, що не масштабується.

1.2 Теоретичні засади управління взаємовідносинами з контрагентами

Цей розділ присвячено систематизації теоретико-методологічної бази, на якій ґрунтується дослідження та проєктування інформаційних систем класу управління взаємовідносинами з контрагентами.

1.2.1 Поняття та сутність систем управління взаємовідносинами

Управління взаємовідносинами з контрагентами у сучасному науковому дискурсі розглядається не як вузька технологічна категорія, а як

міждисциплінарна концепція, що поєднує елементи менеджменту, маркетингу взаємовідносин та інформаційних технологій. У роботі А. Пейна та П. Фроу [2] CRM визначається як стратегічний підхід, що використовує інформацію про контрагентів для створення цінності для них самих та цінності для власників організації, причому реалізація цього підходу потребує крос-функціональної, процесно-орієнтованої взаємодії між підрозділами організації. Таке широке трактування протиставляється вузькому, за якого CRM ототожнюється виключно з програмним забезпеченням для обліку контактів.

Важливим для подальшого викладу є поняття контрагента. Під контрагентом розуміють суб'єкта – фізичну або юридичну особу, – що перебуває у договірних або потенційно договірних відносинах з організацією та виступає однією зі сторін господарської, інформаційної чи комунікаційної взаємодії. У контексті систем CRM до категорії контрагентів зараховують як наявних та потенційних клієнтів, так і постачальників, партнерів, посередників тощо, відомості про яких підлягають структурованому обліку.

Окремого пояснення потребує поняття взаємодії. Взаємодією вважають окремий, зафіксований у часі акт інформаційного, комерційного або сервісного обміну між організацією та контрагентом, що відбувається через визначений канал комунікації – телефонний дзвінок, електронний лист, особисту зустріч, повідомлення у месенджері тощо. Сукупність таких актів формує історію відносин з конкретним контрагентом.

Із поняттям взаємодії тісно пов'язане поняття життєвого циклу відносин, під яким розуміють послідовність якісно різнорідних етапів, через які проходять відносини організації з контрагентом від моменту первинного контакту до завершення співпраці. Типова модель життєвого циклу охоплює стадії залучення (acquisition), утримання (retention), розвитку (development) та, у випадку втрати контрагента, повернення (win-back).

Технологічною основою сучасних CRM-систем виступає веб-застосунок. Веб-застосунком називають програмне забезпечення, побудоване за архітектурою «клієнт-сервер», у якому клієнтська частина виконується у середовищі веб-браузера, а серверна – на віддаленому вузлі та відповідає за обробку даних, реалізацію бізнес-логіки та взаємодію із системою керування базами даних. Обмін даними між клієнтом і сервером здійснюється переважно через протокол передавання гіпертексту (HTTP) із застосуванням архітектурного стилю REST або інших прикладних протоколів. На відміну від традиційних настільних застосунків, веб-застосунок не потребує локального встановлення на пристрій користувача, є кросплатформним за своєю природою та централізовано оновлюється на стороні сервера, що забезпечує миттєве отримання змін усіма користувачами одночасно [3]. Саме архітектура веб-застосунку є переважною формою реалізації сучасних CRM-систем, оскільки вона забезпечує мережевий доступ до єдиного інформаційного середовища з будь-якого пристрою.

Не менш важливим є поняття клієнтського інтерфейсу. Клієнтський інтерфейс становить сукупність візуальних, інтерактивних та програмних засобів, що забезпечують двосторонній обмін інформацією між користувачем і програмною системою: подавання даних із системи у формі, придатній для сприйняття людиною, та передавання користувацьких команд і вхідних даних у систему для подальшої обробки. Клієнтський інтерфейс CRM-системи зазвичай охоплює форми введення інформації про контрагентів, табличні та графічні подання даних, навігаційні елементи, інформаційні панелі (dashboards) та засоби формування звітності. Якість клієнтського інтерфейсу безпосередньо впливає на користувацький досвід, продуктивність роботи операторів системи та повноту й коректність даних, що накопичуються в інформаційному середовищі [3].

1.2.2 Загальна класифікація систем управління взаємовідносинами

Багатоманітність наявних на ринку рішень класу CRM зумовлює потребу у їх класифікації за низкою незалежних ознак. У сучасній науковій та галузевій літературі найбільш усталеними є три класифікаційні розрізи: за функціональним призначенням, за моделлю розгортання та за галузевою спрямованістю [4, 5]. Узагальнену класифікацію систем управління взаємовідносинами з контрагентами за переліченими ознаками наведено на рисунку 1.1.

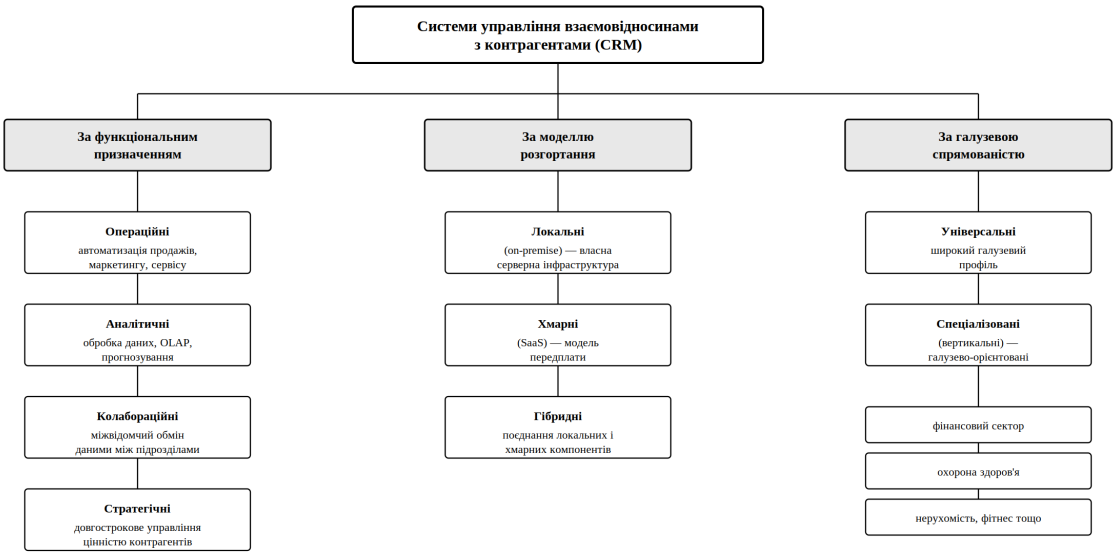


Рисунок 1.1 – Узагальнена класифікація систем управління взаємовідносинами з контрагентами

Поділ за функціональним призначенням відображає переважну спрямованість системи: від автоматизації операційної взаємодії з контрагентами до аналітичної обробки накопичених даних та підтримки стратегічних рішень. Класифікація за моделлю розгортання характеризує спосіб розміщення програмних та апаратних компонентів системи й

визначає розподіл відповідальності за інфраструктуру між організацією та постачальником рішення. Поділ за галузевою спрямованістю відмежовує універсальні рішення, придатні до широкого кола застосувань, від спеціалізованих, функціональність яких від початку врахована під потреби конкретного сектору. Наведені класифікаційні розрізи є взаємно незалежними: одна й та сама система одночасно характеризується ознакою з кожного з них.

1.2.3 Класифікація за функціональним призначенням

Згідно з підходом, започаткованим у роботах із крос-функціонального моделювання CRM [2] та подальшим розвитком у галузевих джерелах [4, 5], виокремлюють чотири базові типи систем.

Операційні CRM-системи (operational CRM) спрямовані на автоматизацію поточних бізнес-процесів, що супроводжують безпосередню взаємодію з контрагентами, – управління продажами, маркетинговими комунікаціями та сервісним обслуговуванням. До типового функціонального наповнення таких систем належать: ведення єдиного реєстру контрагентів, управління воронкою продажів, автоматизація маркетингових кампаній, реєстрація звернень і робота з тикетами служби підтримки. Прикладами операційних CRM є HubSpot, Pipedrive, Zoho CRM.

Аналітичні CRM-системи (analytical CRM) орієнтовані на обробку накопичених даних про контрагентів з метою виявлення закономірностей, сегментації, оцінки прибутковості та прогнозування поведінки. Їх основу складають сховища даних, інструменти багатовимірного аналізу (OLAP), засоби інтелектуального аналізу даних та машинного навчання.

Колаборaційні CRM-системи (collaborative CRM) забезпечують узгодженість дій різних підрозділів організації – продажів, маркетингу, сервісу – за рахунок спільного доступу до єдиної інформації про

контрагента. Цей тип систем особливо релевантний для організацій із розгалуженою організаційною структурою, де взаємодія з контрагентом може здійснюватися різними каналами та різними співробітниками.

Стратегічні CRM-системи (strategic CRM) у трактуванні А. Пейна та П. Фроу [2] виходять за межі суто технологічного рішення та позиціонуються як підтримка стратегічного управління, орієнтованого на побудову довгострокових відносин і максимізацію цінності клієнтського портфеля. Стратегічний рівень CRM не існує як окремий програмний продукт, а реалізується через інтеграцію операційного, аналітичного та колаборaційного контурів і регламентацію відповідних бізнес-процесів.
Слід зауважити, що сучасні промислові CRM-платформи (Salesforce, Microsoft Dynamics 365, SAP CRM) практично завжди суміщають усі чотири функціональні контури в межах єдиного рішення, що дозволяє говорити про конвергенцію типів [5].

1.2.4 Класифікація за моделлю розгортання

За способом фізичного та логічного розміщення обчислювальних ресурсів CRM-системи поділяються на три категорії.

Локальні (on-premise) системи розгортаються на власній серверній інфраструктурі організації-замовника. Перевагами такого підходу є повний контроль над даними та можливість глибокої кастомізації, недоліками – високі початкові капітальні витрати, необхідність утримання кваліфікованого IT-персоналу та складність масштабування.

Хмарні (cloud-based, SaaS) системи розгортаються в інфраструктурі постачальника та надаються користувачеві за моделлю передплати. Ця модель забезпечує швидке введення в експлуатацію, передбачувану структуру операційних витрат, автоматичне оновлення та доступ із будь-

якого місця, однак передбачає певні обмеження щодо кастомізації та накладає вимоги до якості мережевого з'єднання.

Гібридні системи поєднують обидва підходи: критично важливі або чутливі дані зберігаються у локальному контурі, тоді як менш чутливі функціональні модулі реалізуються у хмарі. Такий компроміс дозволяє узгодити вимоги інформаційної безпеки з перевагами хмарних технологій.

1.2.5 Класифікація за галузевою спрямованістю

За цією ознакою CRM-системи поділяють на універсальні та спеціалізовані (вертикальні). Універсальні системи (Salesforce, HubSpot, Microsoft Dynamics 365) розраховані на широкий спектр галузей і забезпечують базовий набір функцій, який далі адаптується під потреби конкретної організації шляхом налаштувань або розробки розширень. Спеціалізовані системи із самого початку проєктуються під специфіку певної галузі – фінансових послуг, охорони здоров'я, нерухомості, фітнес-індустрії тощо – і містять у складі базової функціональності галузеві сутності та бізнес-процеси.

1.2.6 Моделі управління взаємовідносинами з контрагентами

У теорії управління взаємовідносинами з контрагентами розроблено низку концептуальних моделей, що описують структуру та послідовність процесів взаємодії організації з її контрагентами. Ці моделі є методологічною основою для проєктування функціональної архітектури CRM-систем, оскільки визначають, які саме процеси мають бути автоматизовані та які дані підлягають обліку. Нижче розглянуто три найбільш поширені моделі: модель IDIC, модель ланцюжка цінності CRM та модель QCi.

Модель IDIC була розроблена Д. Пепперсом та М. Роджерс у середині 1990-х років та остаточно сформульована у їхній монографії 2004 року [6]. Модель пропонує організації виконувати чотири послідовні дії для побудови індивідуалізованих відносин з кожним контрагентом. На рисунку 1.2 наведено етапи моделі IDIC.

Рисунок 1.2 – Етапи модель IDIC

Назва моделі утворена початковими літерами чотирьох її етапів. На першому етапі – ідентифікації (Identify) – здійснюється формування повного та узгодженого профілю кожного контрагента. Другий етап – диференціація (Differentiate) – передбачає сегментацію контрагентів за їх цінністю для організації та специфікою потреб. На третьому етапі – взаємодії (Interact) – реалізується двостороння комунікація з контрагентом з урахуванням контексту попередньої історії відносин. Четвертий етап – кастомізація (Customize) – полягає в адаптації пропозиції та сервісу під конкретного контрагента. Як показано на рисунку 1.2, перші два етапи утворюють аналітичний контур CRM, а два наступні – операційний; при цьому модель є циклічною: дані, отримані під час взаємодії, через зворотний зв'язок збагачують профіль контрагента та використовуються на наступних ітераціях. Саме така структура моделі обґрунтовує склад функціональних модулів CRM-системи, які мають забезпечувати як накопичення й аналіз даних про контрагентів, так і підтримку безпосередньої взаємодії з ними.

Модель ланцюжка цінності CRM розроблена британським дослідником Ф. Баттлом і остаточно сформульована у його роботах 2001–2004 років [7]. Слід зауважити, що ця модель не є тотожною загальновідомому «ланцюжку цінності» М. Портера 1985 року, який стосується конкурентоспроможності організації в цілому; модель Баттла є самостійною конструкцією, спеціалізованою саме на управлінні відносинами з контрагентами.

Модель QCi розроблена консультаційною компанією QCi Assessment (Н. Вудкок, М. Старкі, М. Стоун та ін.) у 2002 році [8]. Її автори свідомо вживають термін «управління контрагентами» (customer management) замість «управління взаємовідносинами», акцентуючи увагу на сукупності операційних дій, а не на абстрактному понятті відносин. Модель є радше діагностичним інструментом для оцінювання зрілості CRM-практик організації, ніж нормативною методологією [9].

1.3 Огляд існуючих програмних рішень

Для проведення огляду було відібрано чотири системи: Salesforce, HubSpot, Bitrix24 та Zoho CRM. Цей вибір обумовлений такими чинниками. По-перше, зазначені платформи репрезентують різні сегменти ринку – від рішень корпоративного класу (Salesforce) до інструментів для малого та середнього бізнесу (HubSpot, Zoho), а також гібридних платформ із розширеним переліком модулів (Bitrix24). По-друге, всі чотири системи мають значну частку користувачів та документовану практику впровадження у некомерційному секторі, що дозволяє оцінювати їхню придатність для волонтерських організацій. По-третє, кожна з них пропонує користувацький інтерфейс, що є критично важливою вимогою до системи, яка проєктується.

1.3.1 Огляд системи Salesforce

Salesforce [10] є хмарною платформою для управління взаємовідносинами з клієнтами, заснованою у 1999 році. Платформа реалізована за моделлю SaaS та доступна виключно через веб-браузер або фірмові мобільні застосунки.

Функціональне наповнення Salesforce структуроване у вигляді так званих хмар (clouds), кожна з яких є окремим продуктом із власним призначенням. Sales Cloud забезпечує управління воронкою продажів, обліком потенційних клієнтів та угод. Service Cloud призначений для роботи з клієнтськими запитами та технічною підтримкою. Marketing Cloud містить інструменти для проведення маркетингових кампаній та сегментації аудиторії. Окремо виділяється модуль Nonprofit Cloud, спеціально розроблений для некомерційного сектора, який включає функціональність управління донорами, грантами, програмами та волонтерами. Платформа додатково надає Experience Cloud для побудови порталів зовнішніх користувачів та Analytics Cloud для розширеної аналітики.

Особливістю моделі даних Salesforce є висока гнучкість, що досягається через метаданий підхід: усі сутності описуються як метадані та можуть бути модифіковані без зміни вихідного коду. Це дозволяє адміністраторам створювати нестандартні об'єкти та розширювати стандартну модель даних відповідно до потреб організації. Для програмного розширення функціональності використовується власна мова Apex, синтаксично схожа на Java, та декларативна мова запитів SOQL. Приклад інтерфесу Salesforce наведений на рисунку 1.3.

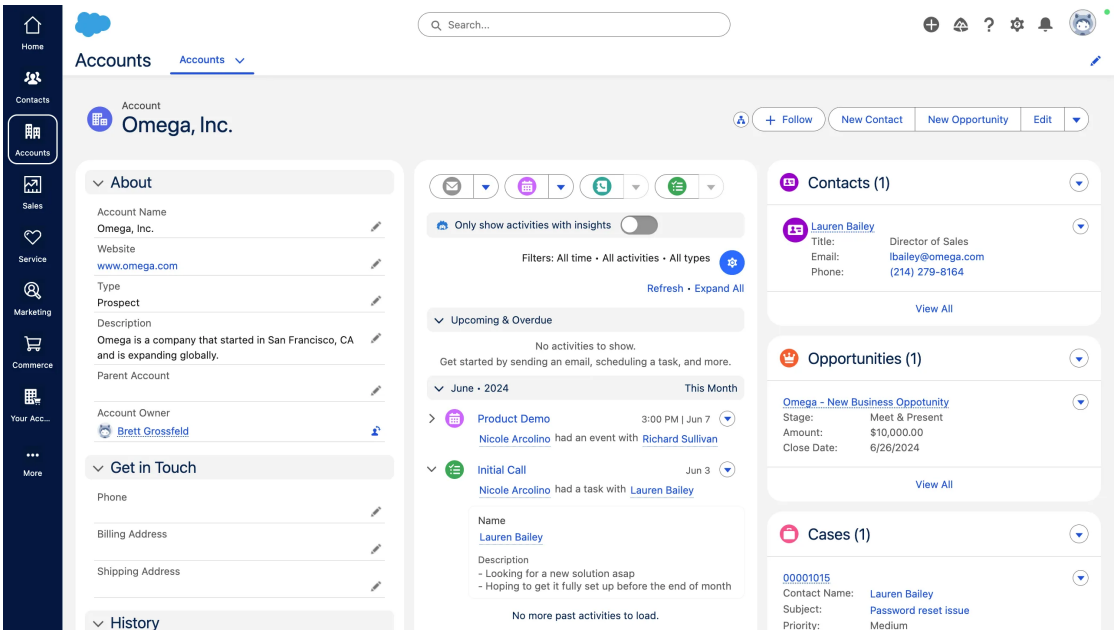


Рисунок 1.3 – Приклад інтерфейсу Salesforce [11]

Як видно з рисунка 1.3, картка контрагента в Salesforce агрегує в межах єдиного екрана відомості про сам обліковий запис, перелік пов'язаних контактних осіб, відкриті комерційні можливості, звернення до служби підтримки та хронологічну стрічку взаємодій. Така організація інтерфейсу ілюструє згаданий вище принцип об'єднання операційного, аналітичного та сервісного контурів навколо єдиного запису про контрагента.

Аналіз сильних сторін Salesforce у контексті досліджуваної задачі дозволяє виокремити кілька позитивних рис. Передусім це наявність спеціалізованого модуля Nonprofit Cloud, який містить готову модель даних для роботи з донорами, програмами допомоги та волонтерами, що частково відповідає потребам волонтерських організацій. Істотною перевагою є також висока гнучкість моделі даних: вона дозволяє адаптувати систему під специфічні процеси координації постачання технічних засобів військовим підрозділам, зокрема через створення власних об'єктів для обліку підопічних, заявок на спорядження та звітів про використання. Крім того, платформа має надійний та задокументований прикладний інтерфейс, що

підтримує протоколи REST та SOAP і забезпечує можливість інтеграції з іншими інформаційними системами.

Водночас застосування Salesforce для задач волонтерської координації має суттєві обмеження. Найвагомішим із них є надзвичайно висока вартість володіння системою. Базова ліцензія Sales Cloud Enterprise коштує приблизно 165 доларів США на одного користувача на місяць, а повноцінне впровадження Nonprofit Cloud потребує додаткових інвестицій у налаштування та навчання персоналу. Хоча компанія Salesforce пропонує програму Power of Us для некомерційних організацій із десятьма безкоштовними ліцензіями, цього обсягу недостатньо для розгортання у мережі волонтерських ініціатив із десятками координаторів. Окремою перешкодою є складність системи, що створює високий поріг входу для нових користувачів. Дослідження користувацького досвіду показують, що повноцінне освоєння Salesforce потребує щонайменше декількох тижнів навчання навіть для адміністраторів, що є неприйнятним для волонтерських організацій, де персонал часто змінюється та значна частина учасників працює на безоплатній основі. Слід зважити й на те, що офіційна локалізація інтерфейсу українською мовою відсутня: хоча платформа підтримує понад тридцять мов, української серед них немає, що створює мовний бар'єр для більшості користувачів.

1.3.2 Огляд системи HubSpot

HubSpot [12] є хмарною платформою для управління взаємовідносинами з клієнтами та маркетингом, заснованою у 2006 році у місті Кембридж, штат Массачусетс. Спочатку компанія розвивала концепцію вхідного маркетингу (inbound marketing), згодом перетворивши її на повноцінну CRM-екосистему. Особливістю позиціонування HubSpot є

орієнтація на малий та середній бізнес, чим пояснюється спрощений інтерфейс та наявність повноцінної безкоштовної версії.

Функціональне наповнення HubSpot структуроване як набір продуктових модулів, що називаються хабами (hubs). Marketing Hub містить інструменти ведення електронних розсилок, налаштування форм збору контактів, аналітики поведінки відвідувачів та автоматизації маркетингових ланцюжків. Sales Hub надає функції управління угодами, послідовностями листування, плануванням зустрічей та прогнозуванням продажів. Service Hub забезпечує роботу з клієнтськими зверненнями, тікет-системою та базою знань. Окремо виділяються CMS Hub для управління вмістом веб-сайтів та Operations Hub для синхронізації даних між системами [12]. У центрі цих модулів розташована безкоштовна CRM-платформа, що забезпечує єдину базу контактів, компаній та угод. Приклад інтерфейсу наведений на рисунку 1.4.

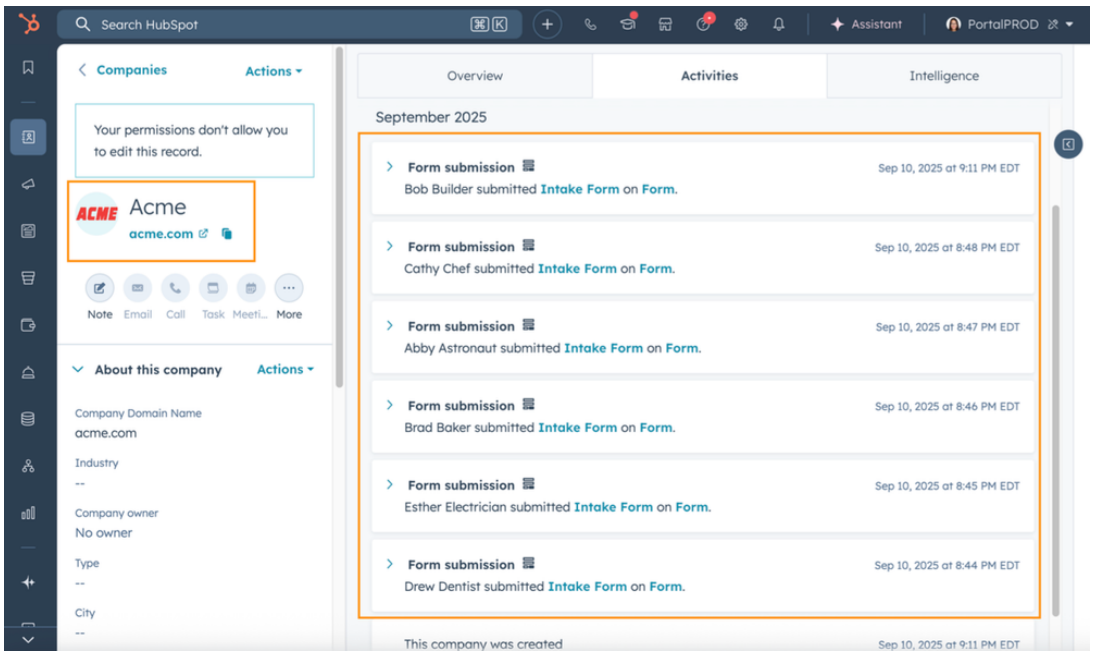


Рисунок 1.4 – Приклад інтерфесу HubSpot [13]

Як видно з рисунка 1.4, картка компанії в HubSpot поєднує в межах єдиного екрана основні відомості про обліковий запис, засоби швидкого створення нотаток, листів, дзвінків і завдань, а також хронологічну стрічку дій, у якій зафіксовано історію взаємодій із контрагентом. Подання історії у вигляді послідовності подій ілюструє орієнтацію платформи на цілісний облік усіх контактів з контрагентом.

До сильних сторін HubSpot у контексті досліджуваної задачі належить кілька особливостей. Передусім наявність безкоштовної версії дозволяє організаціям розпочати використання системи без початкових інвестицій, що особливо важливо для волонтерських ініціатив, які часто діють в умовах обмеженого фінансування. Істотною перевагою є інтуїтивний інтерфейс та невисокий поріг входу, які спрощують навчання нових координаторів, що відповідає потребі швидкого підключення нових учасників до волонтерської роботи. Платформа також пропонує сучасний та задокументований програмний інтерфейс із підтримкою протоколу REST та механізму сповіщень webhook, що уможливлює інтеграцію з іншими сервісами. Окремо слід відзначити підтримку української локалізації інтерфейсу, що знімає мовний бар'єр для українських користувачів.

Поряд із перевагами застосування HubSpot до задачі координації волонтерської діяльності виявляє низку обмежень. Найвідчутнішим із них є значні функціональні обмеження безкоштовної версії: у ній відсутні можливості автоматизації бізнес-процесів, обмежено власні поля та звіти, недоступне рольове управління доступом, а розширення цих можливостей вимагає переходу на платні плани, де ціна зростає нелінійно. Суттєвим обмеженням є також жорстка структурованість моделі даних навколо концепцій «контакт», «компанія» та «угода», що не повною мірою відповідає природі сутностей волонтерської діяльності, де ключовими об'єктами є «підопічний підрозділ», «заявка на постачання», «партія допомоги» та «акт передачі»; адаптація стандартної моделі під ці потреби

можлива через створення нестандартних об'єктів, проте ця можливість доступна лише у плані Enterprise. Слід зважити й на те, що HubSpot орієнтований передусім на комерційну діяльність, тому окремого готового модуля для некомерційного сектора не передбачено, на відміну від Salesforce, а спеціальні знижки для благодійних організацій передбачені лише для деяких категорій.

1.3.3 Огляд системи Bitrix24

Bitrix24 [14] є комплексною платформою для управління бізнесом, що поєднує функціональність CRM-системи, корпоративного порталу, системи управління проєктами та засобу внутрішніх комунікацій. Функціональне наповнення Bitrix24 містить кілька основних складників. Модуль CRM забезпечує управління контактами, компаніями, угодами та воронками продажів. Модуль управління завданнями та проєктами підтримує методології Канбан та діаграми Ганта. Модуль внутрішніх комунікацій містить корпоративний чат, відеоконференції та стрічку новин. Окремо виділяються модулі обліку робочого часу, документообігу та конструктор веб-сайтів. Платформа також пропонує телефонну інтеграцію та засіб електронних розсилок [14]. Приклад інтерфесу Bitrix24 представлено на рисунку 1.5.

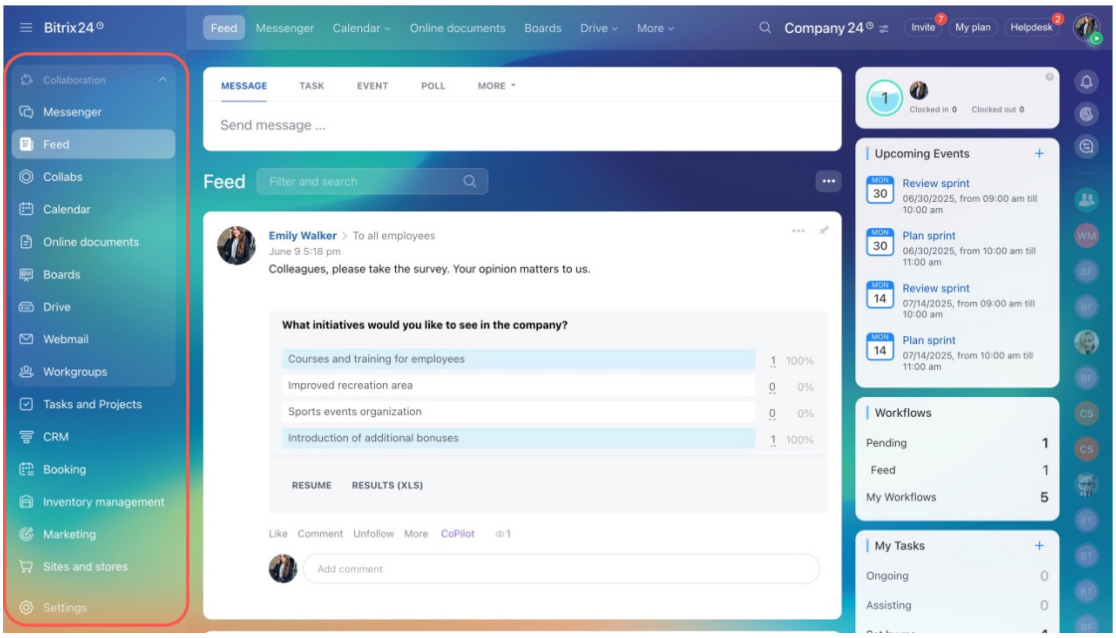


Рисунок 1.5 – Приклад інтерфейсу Bitrix24 [15]

Як видно з рисунка 1.5, усі функціональні модулі платформи зібрані в єдиному веб-інтерфейсі та доступні через вертикальну панель навігації, що ілюструє згаданий вище комплексний характер Bitrix24.

До сильних сторін Bitrix24 у контексті досліджуваної задачі належить кілька особливостей. Передусім історично платформа мала повноцінну українську локалізацію інтерфейсу та документації, що знімало мовний бар'єр для користувачів. Істотною перевагою була наявність локального варіанту розгортання, який забезпечував повний контроль над даними, що особливо важливо у сфері, де інформація має чутливий характер. Привабливим для волонтерських організацій із великою кількістю учасників є також безкоштовний план без обмежень на кількість користувачів. Крім того, комплексний характер платформи дозволяє об'єднати в одному середовищі CRM, управління завданнями та комунікації, що зменшує потребу в інтеграції з зовнішніми сервісами.

Однак у застосуванні Bitrix24 до задачі координації волонтерської діяльності у військовій сфері виявляються істотні обмеження. Найвагомішим із них є фактор походження компанії-розробника та пов'язані

з ним ризики інформаційної безпеки: Bitrix24 належить до продуктів російського походження, що робить використання платформи у волонтерських організаціях, які працюють із військовими підрозділами Збройних сил України, неприйнятним з міркувань захисту чутливої інформації. Навіть якщо абстрагуватися від фактора походження, модель даних Bitrix24 повторює стандартні концепції комерційної CRM-системи і не пристосована до специфіки гуманітарної допомоги, а її адаптація вимагає значних зусиль із доопрацювання. Окремою перешкодою є те, що інтерфейс платформи характеризується високою щільністю елементів управління та значною кількістю функцій, що створює складність для нових користувачів без попереднього досвіду роботи з корпоративними системами. Слід зважити й на те, що прикладний програмний інтерфейс Bitrix24, хоча й існує, має репутацію недостатньо задокументованого та нестабільного між версіями, що ускладнює надійну інтеграцію зі сторонніми застосунками.

1.3.4 Огляд системи Zoho CRM

Zoho CRM [16] є хмарною системою управління взаємовідносинами з клієнтами, що входить до екосистеми Zoho – пакету бізнес-застосунків. На відміну від HubSpot, Zoho CRM має ширше охоплення функціональних областей та позиціонується як вартісно-ефективна альтернатива Salesforce.

Функціональне наповнення Zoho CRM охоплює стандартні модулі: управління потенційними клієнтами (Leads), контактами та компаніями (Contacts, Accounts), угодами (Deals), завданнями та зустрічами. Розширені модулі забезпечують прогнозування продажів, аналітику ефективності, управління територіями та правилами призначення. Окремої уваги заслуговує модуль Zia – вбудований штучний інтелект, що пропонує прогнозування ймовірності закриття угод, виявлення аномалій у даних та автоматизовану категоризацію записів. Платформа інтегрована з іншими

сервісами Zoho – поштовим клієнтом Zoho Mail, системою управління проєктами Zoho Projects, бухгалтерською системою Zoho Books та понад сорока п'ятьма іншими застосунками.

Особливістю моделі даних Zoho CRM є можливість створення нестандартних модулів (custom modules) у плані Professional та вище, що дозволяє адаптувати систему під специфічні сутності предметної області – наприклад, створити модуль «Підопічний підрозділ» або «Заявка на спорядження». Платформа надає декларативний інструмент Canvas для візуальної модифікації інтерфейсу карток без написання коду, а також скриптову мову Deluge для реалізації бізнес-логіки. Приклад інтерфейсу наведений на рисунку 1.6.

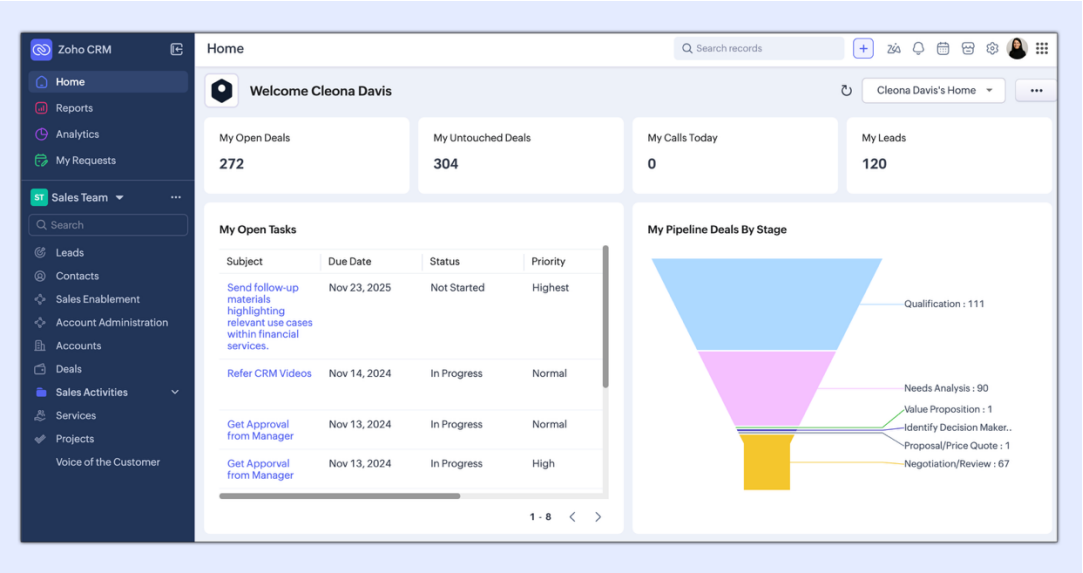


Рисунок 1.6 – Приклад інтерфейсу Zoho CRM [17]

Як видно з рисунка 1.6, інтерфейс системи побудований за принципом інформаційної панелі: ключові кількісні показники винесено у верхню частину екрана, а нижче розміщено докладніші подання даних, що ілюструє аналітичну спрямованість платформи.

До сильних сторін Zoho CRM у контексті досліджуваної задачі належить кілька особливостей. Передусім помірна вартість ліцензій порівняно з Salesforce робить Zoho CRM доступною для волонтерських

організацій із обмеженим бюджетом: зокрема, план Professional за приблизно 23 євро на користувача забезпечує функціональність, що в Salesforce коштує у декілька разів дорожче. Перевагою є й наявність української локалізації інтерфейсу, що спрощує освоєння системи україномовними координаторами. Гнучкість моделі даних із підтримкою нестандартних модулів дозволяє адаптувати систему під специфічні процеси волонтерської діяльності. Слід відзначити також задокументований прикладний програмний інтерфейс на основі протоколу REST, який підтримує всі основні операції створення, читання, оновлення та видалення записів, а також сповіщення webhook про події у системі. Нарешті, екосистема Zoho надає інтегровані інструменти для документообігу, обліку фінансів та комунікацій, що зменшує потребу у пошуку додаткових сервісів.

Поряд із перевагами Zoho CRM виявляє у досліджуваній сфері й низку обмежень. Передусім, попри загальну зручність, інтерфейс Zoho CRM поступається сучасним рішенням у частині візуального оформлення та логіки навігації, що відзначають користувачі у відгуках на платформах G2 та Capterra. Суттєвим недоліком є те, що локалізація українською мовою виконана непослідовно: окремі модулі та діалоги залишаються англійською, а термінологія в перекладі не завжди уніфікована. Окремим обмеженням є відсутність спеціалізованого модуля для некомерційних організацій на кшталт Salesforce Nonprofit Cloud, через що адаптацію під потреби волонтерських організацій доводиться виконувати від базової моделі.

1.3.5 Порівняльний аналіз розглянутих систем

Для систематизації результатів огляду чотирьох систем доцільно звести їхні характеристики у єдину порівняльну таблицю. Критеріями порівняння обрано показники, що безпосередньо впливають на придатність платформи для розв’язання задачі координації волонтерської діяльності у

військовій сфері: повнота функціонального наповнення, наявність української локалізації, вартість володіння, гнучкість моделі даних, наявність API для інтеграції та відповідність специфіці предметної області.

Таблиця 1.1 – Порівняльна характеристика розглянутих CRM-систем

Критерій	**Salesforce**	**HubSpot**	**Bitrix24**	**Zoho CRM**
Безкоштовна версія	–	+	+	+
Українська локалізація	–	+	+	+
Помірна вартість володіння	–	–	+	+
Гнучкість моделі даних	+	–	–	+
Наявність документованого API	+	+	–	+
Спеціалізований модуль для некомерційних організацій	+	–	–	–
Прийнятність для роботи з військовою інформацією	+	+	–	+
Низький поріг входу для користувачів	–	+	–	+

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі виконано аналіз предметної області, постановку проблеми та огляд наявних програмних рішень класу управління взаємовідносинами з контрагентами.

Сформульовано проблему: відсутність цілісної інформаційної системи для управління операційною діяльністю волонтерських фондів військового спрямування змушує організації використовувати сукупність розрізнених програмних засобів загального призначення, що зумовлює відсутність єдиної моделі даних, контролю цілісності зв'язків, наявність інтеграційних розривів, обмежену масштабованість та неможливість автоматизованого формування звітності.

Систематизовано теоретико-методологічну базу управління взаємовідносинами з контрагентами: розкрито сутність концепції CRM, визначено ключові поняття предметної області, наведено класифікацію систем за функціональним призначенням, моделлю розгортання та галузевою спрямованістю.

Проведено порівняльний аналіз чотирьох провідних CRM-систем – Salesforce, HubSpot, Bitrix24 та Zoho CRM. Встановлено, що жодна з розглянутих платформ не задовольняє повного переліку вимог: універсальні комерційні системи не враховують специфіки сутностей волонтерської діяльності та потребують значного доопрацювання, рішення з найповнішою функціональністю мають надмірну для волонтерських організацій вартість володіння та високий поріг входу, а платформа Bitrix24 є неприйнятною з міркувань інформаційної безпеки через походження компанії-розробника.

Визначено проблематику та напрям роботи, які полягають у потребі створення спеціалізованої інформаційної системи, що об'єднує в єдиному інформаційному середовищі облік контрагентів, заявок на постачання,

надходжень та операцій передачі допомоги і враховує специфіку операційної діяльності волонтерських фондів військового спрямування.

На основі обґрунтування проблематики сформульовано мету дипломного проєкту – підвищення ефективності операційної діяльності волонтерських фондів військового спрямування шляхом розроблення веб-застосунку для управління взаємовідносинами з контрагентами, що забезпечує цілісний облік даних, контроль їх цілісності та автоматизоване формування звітності.

Для досягнення поставленої мети дипломного проєкту необхідно виконати такий перелік завдань: проєктування архітектури веб-застосунку та моделі даних, що відповідає специфіці операційної діяльності волонтерських фондів військового спрямування; програмна реалізація серверної та клієнтської частини веб-застосунку із забезпеченням рольової моделі розмежування доступу та задокументованого прикладного програмного інтерфейсу; тестування розробленого веб-застосунку та аналіз його ефективності у порівнянні з розглянутими аналогами.

РОЗДІЛ 2 ОБРАНІ ПІДХОДИ ТА ТЕХНОЛОГІЇ ДО РОЗРОБКИ СИСТЕМИ

2.1 Архітектурний підхід до побудови системи

Для розроблюваної системи управління операційною діяльністю волонтерських фондів вибір архітектури повинен враховувати декілька визначальних чинників: характер користувацької аудиторії, що працюють із різних пристроїв та локацій; необхідність централізованого зберігання та обробки чутливих даних; обмежені технічні ресурси волонтерських організацій, що звужують можливості адміністрування складної інфраструктури; потребу в подальшому розвитку системи після завершення дипломного проєкту, що вимагає чіткої внутрішньої структури коду.

У цьому підрозділі обґрунтовано два ключових архітектурних рішення: вибір клієнт-серверної архітектури з розділенням на клієнтську (frontend) та серверну (backend) частини як загальної моделі побудови застосунку, а також вибір модульного моноліту як архітектурного шаблону для серверної частини.

2.1.1 Обґрунтування вибору клієнт-серверної архітектури з розділенням на frontend та backend

Для реалізації системи розглянуто три принципово різні моделі побудови застосунку: десктопний застосунок, що встановлюється на робочу станцію користувача; веб-застосунок із серверним формуванням сторінок (server-side rendering, SSR), у якому браузер отримує готову розмітку HTML для кожного запиту; односторінковий веб-застосунок (single page application, SPA), що взаємодіє із серверною частиною через прикладний програмний інтерфейс, побудований за принципами REST.

Перший із розглянутих варіантів – десктопний застосунок. Він забезпечує безпосередній доступ до ресурсів операційної системи та може працювати без постійного підключення до мережі. Проте для розглядуваної предметної області така модель має суттєві недоліки. Волонтерські фонди залучають широке коло учасників, які користуються різноманітними пристроями та операційними системами (Windows, macOS, Linux, мобільні платформи), що вимагало б розроблення та підтримки декількох версій застосунку. Встановлення спеціалізованого програмного забезпечення на робочі станції створює адміністративний бар'єр для нових волонтерів та збільшує ризик роботи із застарілими версіями. Крім того, основна частина операцій системи виконується у польових умовах із мобільних пристроїв, на яких встановлення десктопного клієнта є технічно неможливим. Зважаючи на ці чинники, варіант із десктопним застосунком визнано непридатним для розв'язання поставленої задачі.

Другий варіант – веб-застосунок із серверним формуванням сторінок. Така модель передбачає, що при кожній навігації браузер отримує від сервера сформовану сторінку HTML. Це забезпечує швидке початкове завантаження та краще індексування пошуковими системами. Проте у контексті розроблюваної системи останній аргумент не є релевантним, оскільки система за своєю суттю не призначена для публічного індексування. Натомість серверне формування сторінок має суттєві обмеження для систем зі складною інтерактивністю: кожна вагома зміна стану потребує повного або часткового перезавантаження сторінки, що сприймається користувачем як затримка та ускладнює реалізацію оптимістичних оновлень інтерфейсу – функціональності, особливо цінної в умовах нестабільного зв'язку, характерних для прифронтових територій.

Третій варіант – односторінковий веб-застосунок із прикладним програмним інтерфейсом на основі REST. Він передбачає чітке розділення системи на дві частини: клієнтську, що виконується у браузері користувача

та відповідає за відображення інтерфейсу й первинну перевірку введених даних, та серверну, що обробляє запити, виконує бізнес-логіку та взаємодіє з базою даних. Обмін інформацією між частинами здійснюється через запити HTTP, структуровані за принципами REST, із використанням формату JSON для передавання даних.

Така модель забезпечує низку істотних переваг для розглядуваної предметної області. Передусім це доступність із будь-якого пристрою без встановлення: користувачеві достатньо мати сучасний веб-браузер та доступ до мережі. Це критично важливо для волонтерів, що працюють у польових умовах із наявних на місці пристроїв, та для представників військових частин, на чиї робочі станції встановлення стороннього програмного забезпечення може бути обмежено внутрішніми регламентами. Істотною перевагою є винесення критичної бізнес-логіки на захищений сервер: усі операції, що визначають коректність даних та права доступу – затвердження фінансових транзакцій, формування звітів для донорів, ведення журналу аудиту, – виконуються виключно на серверній стороні, недоступній для модифікації з боку клієнта. Клієнтська частина в моделі SPA не повинна вважатися довіреним джерелом даних: будь-яка перевірка, виконана у браузері, дублюється на сервері, оскільки користувач технічно здатний обійти клієнтські перевірки шляхом прямих звернень до прикладного програмного інтерфейсу. Архітектура із розділенням на дві частини забезпечує також розмежування зон відповідальності та можливість незалежного розвитку: вона дозволяє вести розроблення клієнтського та серверного коду паралельно, ґрунтуючись на узгодженому контракті інтерфейсу, і спрощує подальший розвиток системи, адже за потреби клієнтська частина може бути доповнена окремим мобільним застосунком, що звертається до того самого інтерфейсу, без модифікації серверної логіки; аналогічно, серверна частина може використовуватися сторонніми системами через інтеграційний інтерфейс – наприклад,

бухгалтерською програмою фонду для автоматизованого формування звітів. Нарешті, модель SPA забезпечує природну підтримку інтерактивних сценаріїв роботи: багато операцій у системі мають характер тривалої взаємодії – формування закупівлі складається з вибору постачальника, додавання позицій, прикріплення документів, попереднього перегляду, узгодження, – і ця модель дозволяє виконувати такі кроки без перезавантаження сторінки, зберігаючи проміжний стан у пам'яті браузера та надсилаючи на сервер лише ті дані, які дійсно потребують обробки.

На основі проведеного аналізу для реалізації системи обрано модель односторінкового веб-застосунку із серверною частиною, що надає прикладний програмний інтерфейс на основі REST. Така архітектура найповніше відповідає характеру предметної області та типовим сценаріям роботи користувачів, сформованим у першому розділі.

2.1.2 Обґрунтування вибору архітектурного шаблону для серверної частини

Після визначення загальної моделі взаємодії між клієнтом і сервером постає питання внутрішньої організації серверної частини. У сучасній практиці розроблення веб-застосунків розрізняють три основні архітектурні шаблони: класичний моноліт, модульний моноліт та мікросервісну архітектуру. Кожен із цих шаблонів має власну сферу застосування, і помилковий вибір на цьому етапі призводить до значних труднощів на подальших етапах розроблення та експлуатації.

Першим із розглянутих шаблонів є класичний моноліт. У межах цього шаблону весь серверний код являє собою єдиний застосунок, у якому всі частини бізнес-логіки взаємодіють напряму через виклики функцій і поділяють спільний стан. Перевагами такого підходу є простота реалізації, відсутність накладних витрат на міжмодульну взаємодію та простота

початкового розгортання. Недоліком, що особливо проявляється у проєктах з розвиненою бізнес-логікою, є тенденція до утворення сильно зв'язаного коду, у якому межі між функціональними областями розмиваються, а внесення змін до однієї частини непередбачувано впливає на інші. Для розроблюваної системи, що оперує декількома чітко окресленими доменними областями, такий підхід призведе до накопичення технічного боргу вже на ранніх етапах розвитку.

Другим шаблоном є мікросервісна архітектура. Мікросервісний підхід передбачає розділення серверної частини на декілька незалежних застосунків, кожен з яких має власну базу даних, розгортається окремо та взаємодіє з іншими через мережевий протокол. Серед декларованих переваг – незалежне масштабування окремих компонентів, ізоляція збоїв, можливість використання різних технологічних засобів для різних сервісів. Однак мікросервісна архітектура несе суттєві експлуатаційні витрати: потребу в системі оркестрування, розподілене журналювання, відстеження запитів між сервісами, забезпечення цілісності даних за відсутності розподілених транзакцій. Крім того, поточний масштаб системи не виправдовує переходу на мікросервіси: вони доцільні насамперед для систем, у яких різні домени мають кардинально відмінні профілі навантаження та потребують незалежного масштабування, чого в розглядуваній системі не спостерігається.

Третім шаблоном є модульний моноліт, який становить компроміс між двома попередніми. Серверний код розгортається як єдиний застосунок, проте всередині нього виділяються чітко відмежовані модулі, кожен з яких відповідає за окрему доменну область, має власні внутрішні структури даних та надає назовні явно визначений інтерфейс. Взаємодія між модулями відбувається виключно через ці інтерфейси, що запобігає неконтрольованому проникненню деталей реалізації одного модуля в

інший. Такий підхід зберігає простоту експлуатації моноліту та водночас забезпечує внутрішню структурованість, властиву мікросервісам.

Для розроблюваної системи модульний моноліт обрано як оптимальний архітектурний шаблон з огляду на кілька міркувань. Передусім масштаб дипломного проєкту та орієнтовний обсяг навантаження не виправдовують додаткової складності мікросервісної архітектури. Істотним є й те, що цілісність даних має критичне значення: група пов'язаних операцій – наприклад, реєстрація передачі допомоги, оновлення залишків на складі та запис до журналу аудиту – повинна виконуватися атомарно, що природно реалізується в межах єдиної транзакції бази даних, доступної модульному моноліту. Слід зважити також на обмежені адміністративні ресурси волонтерських організацій: для них просте розгортання у вигляді одного контейнера, що містить серверний застосунок, є суттєво кращим варіантом, ніж розгортання та супровід оркестрованої системи з багатьох взаємопов'язаних сервісів. Нарешті, чітка модульна структура коду закладає основу для можливого майбутнього виділення окремих модулів у самостійні сервіси, якщо такий поділ стане виправданим унаслідок зростання навантаження або розширення функціональності: модульний моноліт не закриває шляху до мікросервісів, а лише відкладає це рішення до моменту, коли воно стане обґрунтованим.

Обраний архітектурний шаблон формує підґрунтя для подальшого вибору серверного каркаса, який має забезпечувати ефективну реалізацію модульного підходу.

2.2 Вибір технологій серверної частини

Після визначення архітектурного підходу постає завдання вибору конкретних технологій реалізації серверної частини. Серверна сторона системи виконує найбільш відповідальні функції: обробляє запити

користувачів, виконує бізнес-логіку, забезпечує цілісність даних, реалізує механізми автентифікації та авторизації, веде журнал аудиту. У зв'язку з цим вибір технологій серверної частини потребує особливо ретельного обґрунтування з огляду на критичність коректної роботи цих компонентів для функціонування системи в цілому.

Технологічний стек серверної частини базується на середовищі виконання Node.js, що дозволяє використовувати TypeScript для написання серверного коду. У наступних підрозділах послідовно обґрунтовано вибір серверного фреймворка та системи керування базами даних – двох визначальних компонентів, що формують основу серверної інфраструктури системи.

2.2.1 Фреймворк для побудови серверного застосунку

Серверний фреймворк визначає спосіб організації коду, доступні засоби обробки запитів HTTP, механізми взаємодії з базою даних, набір вбудованих засобів забезпечення безпеки та інші ключові аспекти реалізації. Для екосистеми Node.js доступна значна кількість серверних фреймворків, проте найбільш поширеними та зрілими рішеннями є Express, Fastify та NestJS. Кожне з цих рішень має власну концепцію та найкраще підходить для певного типу проєктів.

Першим із розглянутих рішень є Express – мінімалістичний серверний фреймворк, що тривалий час залишається фактичним стандартом розроблення на Node.js [18]. Express надає базовий шар обробки запитів HTTP та механізм проміжного програмного забезпечення (middleware), залишаючи розробникові свободу у виборі решти компонентів – структури проєкту, механізму перевірки даних, способу взаємодії з базою даних, реалізації автентифікації. Така мінімалістичність є перевагою для невеликих сервісів зі специфічними вимогами, проте для розглядуваної системи вона

перетворюється на недолік. Розроблення системи управління операційною діяльністю волонтерських фондів передбачає реалізацію значної кількості однотипних задач: перевірку вхідних даних, перевірку прав доступу, журналювання дій, формування узгоджених відповідей про помилки. Express не надає готових рішень для цих задач, тож кожен такий аспект розробник реалізує самостійно або інтегрує сторонні бібліотеки, що неминуче призводить до неоднорідності коду в межах одного проєкту та збільшує ймовірність помилок у критично важливій логіці.

Другим рішенням є Fastify – відносно новий серверний фреймворк, що позиціонується як високопродуктивна альтернатива Express [19]. Ключовою його перевагою є вбудована схемна перевірка вхідних та вихідних даних на основі специфікації JSON Schema, що одночасно забезпечує контроль коректності даних та підвищення продуктивності за рахунок попереднього компілювання перетворювачів JSON. Fastify також має кращу підтримку TypeScript порівняно з Express та структурованіше розширення через систему модулів-розширень. Однак Fastify зберігає мінімалістичний підхід щодо загальної структури проєкту: він не диктує спосіб організації коду в модулі, не надає вбудованих механізмів упровадження залежностей (dependency injection) та залишає вибір структури проєкту на розсуд розробника. Для системи, у якій передбачено декілька доменних модулів зі складною взаємодією, відсутність визначеної структури з часом призведе до тих самих труднощів, що й у випадку Express.

Третім рішенням є NestJS – фреймворк вищого рівня, що надає не лише засоби обробки запитів HTTP, але й структурну основу для побудови серверних застосунків [20]. NestJS реалізує концепцію модульної архітектури як обов'язковий елемент організації коду: серверний застосунок поділяється на модулі, кожен з яких інкапсулює власні контролери, сервіси та інші компоненти. Така структура природно відповідає обраному в пункті 2.1.2 архітектурному шаблону модульного моноліту. Слід зазначити, що

NestJS має й певні обмеження. Зокрема, він висуває вищі вимоги до початкового освоєння порівняно з Express через використання концепцій, нетипових для базової екосистеми JavaScript, – декораторів, упровадження залежностей, обов'язкової модульної структури. Застосунки на NestJS мають також дещо вищі накладні витрати під час запуску та обробки запитів порівняно з Fastify. Проте для розроблюваної системи перша обставина компенсується тим, що структурний підхід окуповується вже на ранніх етапах розвитку складної бізнес-логіки, а друга – тим, що очікуване навантаження є невисоким, і різниця в продуктивності між фреймворками не є визначальним чинником.

На основі проведеного аналізу для реалізації серверної частини системи обрано фреймворк NestJS. Цей вибір зумовлений збігом між модульною концепцією NestJS та обраним у пункті 2.1.2 архітектурним шаблоном модульного моноліту, повноцінною підтримкою TypeScript, наявністю вбудованих засобів перевірки даних та авторизації, а також можливістю автоматичного формування документації прикладного програмного інтерфейсу.

2.2.2 Система керування базами даних

Вибір системи керування базами даних є одним із найвідповідальніших технологічних рішень, оскільки воно визначає спосіб зберігання даних на тривалу перспективу та накладає обмеження на типи запитів, які можна ефективно виконувати. Зміна системи керування базами даних на пізніших етапах розвитку проєкту є складною та трудомісткою операцією, тому початковий вибір повинен бути обґрунтований із врахуванням довгострокових потреб системи.

Дані, якими оперує розроблювана система, мають характеристики, що визначають вимоги до системи керування базами даних: структурованість,

складні зв'язки між сутностями, потребу в транзакційній цілісності та необхідність складних аналітичних запитів.

Серед реляційних систем керування базами даних з відкритим кодом, що підходять для розглядуваної системи, провідними рішеннями є PostgreSQL та MySQL. MySQL є популярною та зрілою системою, проте PostgreSQL традиційно вирізняється повнішою підтримкою стандарту SQL, ширшим набором вбудованих типів даних та більш розвиненим механізмом розширень [21]. Документоорієнтовані рішення (MongoDB) та інші моделі (сховища типу «ключ – значення», графові бази) розглянуто, але визнано непридатними для основного навантаження системи з огляду на структурований характер даних та потребу в транзакційній цілісності.

Як основну систему керування базами даних обрано PostgreSQL з огляду на сукупність її властивостей, що відповідають наведеним вимогам. Передусім PostgreSQL забезпечує повноцінну підтримку транзакцій із дотриманням властивостей ACID: атомарне виконання послідовностей операцій у межах транзакції з можливістю налаштування рівня ізольованості, включно з найсуворішим рівнем serializable. У разі будь-якого збою посередині операції стан бази повертається до початкового, не залишаючи частково оновлених даних; для системи, що оперує важливою інформацією, ця властивість є не зручністю, а вимогою, оскільки часткове виконання транзакції потенційно призводить до неузгодженостей. Істотним є й те, що PostgreSQL підтримує зовнішні ключі для забезпечення посилальної цілісності: вона дозволяє декларативно описувати обмеження посилання між таблицями (foreign key constraints), які перевіряються самою системою керування базами даних на найнижчому рівні, що гарантує цілісність даних незалежно від коректності прикладного коду. Окремою перевагою є підтримка типу JSONB для слабоструктурованих метаданих – бінарного подання JSON, що зберігається ефективно та підтримує індексування за полями всередині документа [21]. Це дозволяє в межах

однієї бази даних поєднувати реляційне зберігання основних сутностей із гнучким зберіганням метаданих змінної структури – наприклад, історичних знімків стану записів для журналу аудиту, налаштувань шаблонів звітів, додаткових атрибутів закупівель, що залежать від типу товару, – і в такий спосіб усуває потребу інтеграції окремої документоорієнтованої бази даних, зберігаючи можливість виконання запитів SQL до всіх даних системи. Нарешті, PostgreSQL вирізняється розширюваністю та підтримкою спеціалізованих запитів: її механізм розширень дозволяє додавати нові типи даних, оператори та індекси без модифікації ядра, а вбудована повнотекстова пошукова підсистема, доступна без додаткових пакунків, забезпечує пошук за описами позицій, накладних та коментарів із морфологічним нормуванням слів для української мови.

2.2.3 Інструмент для роботи з базою даних

Прямий доступ серверного коду до бази даних може реалізовуватися різними засобами, що відрізняються рівнем абстрагування над мовою SQL: написанням запитів SQL вручну, використанням конструктора запитів (query builder) або повноцінного об'єктно-реляційного відображення (ORM).

Написання запитів SQL вручну забезпечує повний контроль над запитами та передбачувану продуктивність, проте позбавляє код типобезпеки: запити зберігаються як рядкові літерали, помилки в назвах стовпців виявляються лише під час виконання, а результати повертаються як нетипізовані об'єкти. Конструктори запитів на кшталт Knex розв'язують задачу безпечного формування запитів та надають вбудоване керування міграціями [22], проте не забезпечують наскрізної типобезпеки – схема бази залишається невідомою компілятору TypeScript. Для системи, що оперує фінансовими даними, відсутність типобезпеки є визначальним недоліком обох підходів.

Сучасні засоби об'єктно-реляційного відображення для екосистеми TypeScript забезпечують типобезпечну взаємодію з базою на основі описаної схеми. Серед провідних рішень – Prisma, що використовує декларативний опис схеми в окремому файлі та формує типобезпечного клієнта [23], та Drizzle, що описує схему безпосередньо в коді TypeScript і формує запити, близькі за продуктивністю до написаних вручну [24]. Prisma вирізняється зрілістю та обсягом доступних матеріалів, вбудованим механізмом керування міграціями та інструментом графічного перегляду даних Prisma Studio. Drizzle є молодшим інструментом з меншою екосистемою.

Для розроблюваної системи обрано засіб об'єктно-реляційного відображення Prisma з огляду на наскрізну типобезпеку від схеми бази до коду серверних обробників, декларативний опис схеми у єдиному файлі, що підвищує її читність, вбудований механізм міграцій, який зберігає історію змін схеми разом із кодом проєкту, та зрілість інструменту, що знижує ризики освоєння в межах часових рамок дипломного проєкту. Можлива втрата продуктивності в окремих складних запитах не є визначальною для системи з помірним очікуваним навантаженням, а в крайніх випадках Prisma надає механізм виконання довільних запитів SQL зі збереженням типобезпеки результату.

2.3 Вибір технологій клієнтської частини

Клієнтська частина системи виконується безпосередньо у браузері користувача та відповідає за відображення інтерфейсу, обробку дій користувача, формування запитів до серверного API та представлення отриманих даних. На відміну від серверної частини, що працює в контрольованому середовищі, клієнтський код виконується в умовах значної варіативності – різні браузери, пристрої, швидкість мережевого

з'єднання, продуктивність обчислювальних ресурсів. Це накладає на вибір клієнтських технологій додаткові вимоги: широка сумісність, ощадливе використання ресурсів пристрою, стійкість до повільного або переривчастого з'єднання.

2.3.1 Бібліотека для побудови інтерфейсу користувача

Бібліотека для побудови інтерфейсу користувача визначає спосіб опису компонентів інтерфейсу, механізм відстеження змін стану та оновлення відображення, а також сумісність із зовнішніми бібліотеками. Для сучасного веб-розроблення на TypeScript провідними рішеннями є React, Vue та Svelte. Кожне з них реалізує компонентний підхід, але відрізняється внутрішньою організацією та екосистемою.

Першим із розглянутих рішень є React – бібліотека для побудови інтерфейсу, розроблена компанією Meta та поширювана з відкритим кодом [25]. React реалізує компонентну архітектуру з декларативним описом інтерфейсу через синтаксичне розширення JSX, що дозволяє вкладати розмітку, подібну до HTML, безпосередньо в код мовами JavaScript або TypeScript. Зміни в інтерфейсі відбуваються через оновлення стану компонента, після чого React самостійно визначає мінімальний набір змін, які потрібно застосувати до відображення (механізм узгодження, reconciliation). React має одну з найбільших спільнот серед усіх веб-бібліотек, що означає наявність готових рішень для більшості типових задач, велику кількість навчальних матеріалів та активну підтримку. Підтримка TypeScript у React є зрілою та повноцінною: усі офіційні програмні інтерфейси мають точні типові декларації, а синтаксис JSX інтегрується із системою типів TypeScript.

Другим рішенням є Vue – самостійна бібліотека для побудови інтерфейсу, що реалізує власний підхід, за якого компоненти подаються у

вигляді однофайлових модулів, які поєднують розмітку, логіку та стилі [26]. Vue має репутацію зручного для початкового освоєння інструменту та широко використовується, зокрема, у Китаї та Європі. Підтримка TypeScript у Vue 3 значно покращена порівняно з попередніми версіями, проте інтеграція TypeScript із шаблонним синтаксисом Vue історично залишається менш безшовною, ніж у випадку JSX у React. Розмір спільноти Vue менший за React, що відбивається на меншій кількості готових бібліотек для специфічних потреб.

Третім рішенням є Svelte, який вирізняється принципово іншим підходом: замість виконання бібліотечного коду в браузері Svelte є компілятором, що перетворює описи компонентів на оптимізований імперативний код мовою JavaScript на етапі збирання [27]. Це забезпечує менший розмір кінцевого пакета коду та вищу продуктивність відображення. Проте Svelte має значно меншу спільноту та екосистему порівняно з React і Vue, що особливо помітно для специфічних задач: бібліотек складних таблиць із сортуванням, фільтруванням та віртуалізацією для Svelte значно менше, як і бібліотек побудови графіків та форм зі складною перевіркою даних. Для розглядуваної системи це є істотним обмеженням.

На основі проведеного аналізу для побудови інтерфейсу користувача обрано React. Цей вибір зумовлений найбільшим розміром спільноти серед розглянутих бібліотек, що забезпечує доступність готових рішень для всіх типів елементів інтерфейсу, передбачених у системі; зрілою та повноцінною підтримкою TypeScript, що відповідає загальній стратегії наскрізної типобезпеки; широким досвідом використання React у проєктах із подібним профілем – бізнес-застосунках зі складними формами та таблицями, – що підтверджує придатність для розглядуваної задачі. Принциповий недолік React – відсутність найкращої продуктивності серед розглянутих варіантів,

оскільки Svelte є ефективнішим, – не є визначальним для системи з очікуваним помірним навантаженням на клієнтську сторону.

2.3.2 Інструмент збірки клієнтського застосунку

Сучасні клієнтські застосунки на JavaScript і TypeScript не виконуються браузером безпосередньо: вихідний код проходить етап збірки, на якому він транспілюється, об'єднується з модулями залежностей та оптимізується. Інструмент збірки визначає швидкість циклу розробки – час холодного старту сервера розробки, тривалість гарячого перезавантаження модулів (HMR) – та характеристики кінцевого бандла, що завантажується користувачем.

Webpack тривалий час залишається найпоширенішим інструментом збірки в екосистемі JavaScript [28]. Він має винятково гнучку конфігурацію, проте побудований на принципі повної збірки всього застосунку, що призводить до повільного холодного старту та складної конфігурації навіть для типових задач. Vite принципово переосмислює процес розробки: під час розробки він використовує нативні модулі ES, які підтримуються сучасними браузерами безпосередньо, тож сервер не виконує повної збірки, а лише транспілює окремі файли на запит браузера [29]. Це забезпечує практично миттєвий холодний старт незалежно від розміру проєкту та швидке гаряче перезавантаження. Для виробничої збірки Vite використовує Rollup, що генерує оптимізовані бандли. Turbopack – інструмент нового покоління, написаний на Rust, що обіцяє вищу продуктивність за рахунок інкрементальної компіляції [30], проте на момент проєктування системи залишається відносно молодим: офіційно стабільним його використання поза екосистемою Next.js не визнане, а екосистема плагінів значно менш розвинена, ніж у Vite.

Як інструмент збірки обрано Vite з огляду на практично миттєвий холодний старт сервера розробки, швидке гаряче перезавантаження модулів, оптимізовані виробничі бандли та мінімалістичну конфігурацію. Зрілість Vite та наявність офіційного шаблону для зв'язки React та TypeScript роблять його найбільш виваженим вибором у межах обмежених ресурсів дипломного проєкту.

2.3.3 Підхід до клієнтської маршрутизації

Розроблювана система містить значну кількість логічно відмежованих розділів (користувачі, пожертви, закупівлі, склад, звітність, аудит), кожен з яких поділяється на сторінки переліку, картки окремого запису та форми. У межах однієї картки сутності реалізовано декілька вкладок – наприклад, картка закупівлі містить розділи «позиції», «документи», «логістика», «отримувач». Така структура потребує механізму маршрутизації, який відображає поточний стан інтерфейсу в URL-адресі. Клієнтська маршрутизація для обраної моделі односторінкового застосунку є природним вибором: вона дозволяє змінювати відображуваний розділ без перезавантаження сторінки, зберігаючи стан застосунку між переходами, а також забезпечує можливість обміну посиланнями на конкретні записи та повернення через історію браузера.

Для екосистеми React провідними рішеннями є React Router та TanStack Router. React Router є найпоширенішим рішенням для маршрутизації в React-застосунках та підтримує опис маршрутів у вигляді ієрархічної структури з вкладеними дочірніми маршрутами [31]. TanStack Router є відносно новим рішенням від тієї ж команди, що розробляє TanStack Query [32]. Ключовою перевагою TanStack Router є наскрізна типобезпека маршрутів: вони описуються у спосіб, який дозволяє TypeScript перевіряти коректність переходів, параметрів URL та параметрів пошуку на

етапі компіляції, тоді як у React Router типізація параметрів є частковою. TanStack Router також має вбудовану підтримку завантаження даних на рівні маршруту (route loaders), що інтегрується з механізмом кешування TanStack Query.

Обидві бібліотеки підтримують вкладені маршрути – необхідну умову для реалізації карток сутностей із внутрішніми вкладками. На прикладі закупівлі: маршрут /purchases/:id відображає основну картку зі спільною верхньою частиною (заголовок, реквізити, кнопки дій) та областю, у якій залежно від поточного вкладеного маршруту відображається один із розділів – /items, /documents, /recipient. Така структура дозволяє зберігати спільну частину інтерфейсу нерухомою під час перемикання між вкладками, зберігати в URL поточну активну вкладку для прямого переходу за посиланням та кешувати дані кожного розділу окремо.

Для розроблюваної системи обрано TanStack Router з огляду на наскрізну типобезпеку маршрутів, узгодженість екосистеми з обраним TanStack Query, що забезпечує єдиний підхід до завантаження та кешування даних, та вбудовану підтримку route loaders. Зворотним боком вибору є дещо менший обсяг навчальних матеріалів порівняно з React Router, проте це обмеження компенсується виграшем у типобезпеці, особливо суттєвим для системи зі складною структурою маршрутів.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі обґрунтовано вибір технологій та архітектурних підходів для розробки системи управління операційною діяльністю волонтерських фондів у військовій сфері. Технологічні рішення обрано з урахуванням сформованих у першому розділі функціональних та нефункціональних вимог, специфіки предметної області та обраного типу цільового продукту – веб-застосунку.

На рівні архітектурних рішень обрано модель односторінкового веб-застосунку із серверною частиною, що надає REST API, із чітким розмежуванням меж довіри між клієнтом і сервером. Для серверної частини обрано шаблон модульного моноліту як компроміс між простотою експлуатації класичного моноліту та внутрішньою структурованістю мікросервісної архітектури, що відповідає масштабу дипломного проєкту та обмеженим ресурсам волонтерських організацій на адміністрування інфраструктури.

Для серверної частини обрано фреймворк NestJS, концепція модульної архітектури якого збігається з обраним архітектурним шаблоном. NestJS забезпечує наскрізну підтримку TypeScript, вбудовані механізми валідації та авторизації, що є критичним для системи, у якій коректність даних має визначальне значення. Як систему керування базами даних обрано PostgreSQL з огляду на повноцінну підтримку ACID-транзакцій, референційну цілісність через зовнішні ключі, гнучке зберігання слабоструктурованих метаданих через тип JSONB та розширюваність для повнотекстового пошуку. Для типобезпечної взаємодії серверного коду з базою даних обрано Prisma ORM із вбудованим механізмом керування міграціями.

Для клієнтської частини обрано бібліотеку React, що вирізняється найбільшим серед розглянутих рішень обсягом екосистеми готових

компонентів для типових задач системи (складні таблиці, форми, графіки). Як інструмент збірки обрано Vite, що забезпечує швидкий цикл розробки та оптимізовані виробничі бандли. Для клієнтської маршрутизації обрано TanStack Router з наскрізною типобезпекою маршрутів та інтеграцією з рештою екосистеми TanStack.

Обраний технологічний стек формує надійне підґрунтя для подальшої реалізації системи, описаної в третьому розділі.

РОЗДІЛ 3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Структура проєкту та середовище розробки

Перед безпосередньою реалізацією системи визначено організацію вихідного коду та налаштовано середовище розробки. Система складається з двох частин, що розгортаються окремо: серверної, яка надає прикладний програмний інтерфейс на основі REST, та клієнтської, реалізованої як односторінковий веб-застосунок. Обидві частини зберігаються в єдиному репозиторії, організованому за принципом монорепозиторію (monorepo) із застосуванням робочих просторів пакетного менеджера pnpm та системи керування складанням Turborepo.

Версію середовища виконання Node.js закріплено у файлі .nvmrc, а пакетний менеджер pnpm активується через механізм Corepack із фіксованою версією. Склад робочих просторів описано у файлі pnpm-workspace.yaml, який відносить до проєкту каталоги apps/* та packages/*. Каталог packages зарезервовано для спільного коду серверної та клієнтської частин. Система Turborepo запускає однотипні сценарії – розробницький запуск, складання, перевірку стилю коду та перевірку типів – для всіх пакетів єдиною командою з кореня репозиторію.

Серверну частину (apps/server) поділено на чотири рівні відповідальності. Каталог common містить наскрізні засоби, спільні для всіх модулів: захисники доступу, декоратори та спільні об'єкти передавання даних. Каталог config відповідає за конфігурацію застосунку та перевірку коректності змінних середовища. Каталог infrastructure об'єднує інфраструктурні служби, насамперед взаємодію з базою даних, поштовим сервісом та файловим сховищем. Каталог modules містить доменні модулі,

кожен з яких відповідає за окрему предметну область, як-от заявки, закупівлі чи склад. Клієнтську частину (apps/web) структуровано за функціональними ознаками: маршрути системи зосереджено в каталозі routes, реалізацію окремих розділів – у каталозі features, багаторазові елементи інтерфейсу – у каталозі components, а допоміжні служби, такі як клієнт прикладного інтерфейсу, керування сеансом і правила розмежування доступу, – у каталозі lib. Структуру каталогів монорепозиторію наведено на рисунку 3.1.

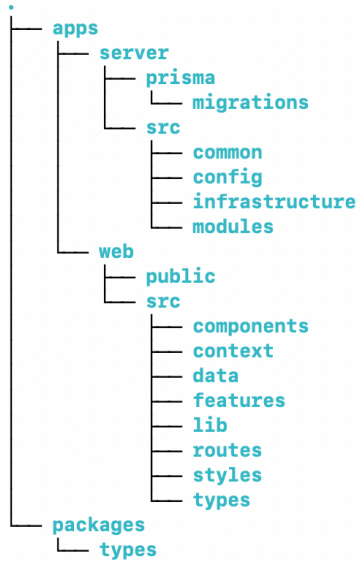


Рисунок 3.1 – Узагальнена структура каталогів монорепозиторію

Як видно з рисунка 3.1, у межах каталогу apps розміщено серверну та клієнтську частини, а каталог packages відведено під спільний код. Усередині серверної частини відображено поділ на наскрізні засоби, конфігурацію, інфраструктуру та доменні модулі, що відповідає структурі модульного моноліту, описаній у другому розділі.

3.2 Рівень доступу до даних

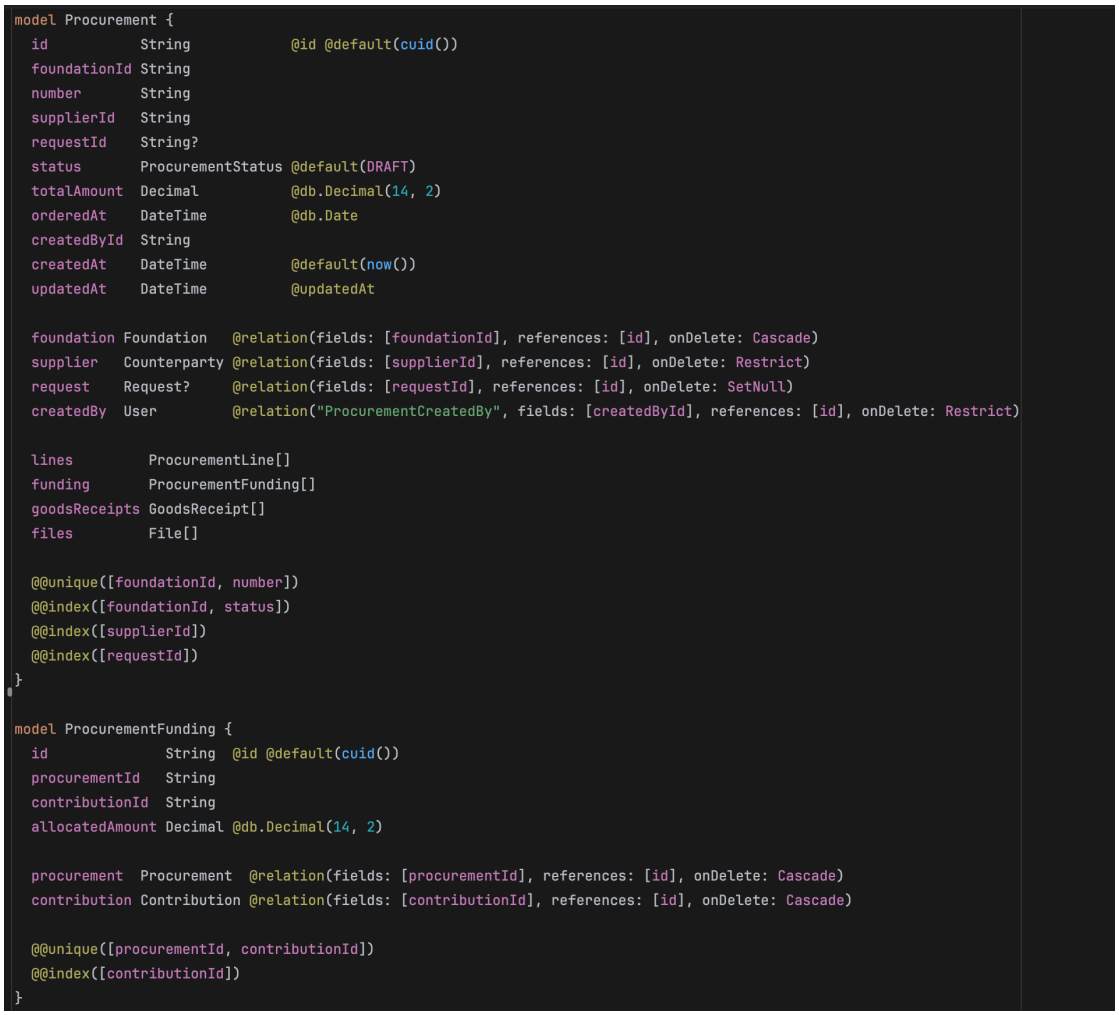
Рівень доступу до даних реалізовано на системі керування базами даних PostgreSQL із застосуванням засобу об'єктно-реляційного

відображення Prisma. Реалізацію зосереджено навколо єдиного декларативного опису схеми у файлі schema.prisma, на основі якого Prisma формує міграції бази даних та типобезпечного клієнта для серверного коду.

Кореневою сутністю моделі даних є фонд (Foundation). Кожна доменна сутність містить атрибут foundationId із зовнішнім ключем на сутність фонду та політикою каскадного видалення, унаслідок чого видалення фонду призводить до видалення всіх пов'язаних із ним даних. Ізоляція даних між фондами виконується на рівні запитів: усі операції читання й запису фільтруються за ідентифікатором фонду, отриманим із токена доступу, – відповідний механізм розглянуто в підрозділі 3.3. Обмеження унікальності визначено в межах окремого фонду: наприклад, складений унікальний ключ з fundationId та number діє на номер заявки лише в межах одного фонду.

Граф операцій, описаний у першому розділі, реалізовано зовнішніми ключами між сутностями з трьома політиками видалення. Політику каскадного видалення (Cascade) застосовано до позицій заявки та закупівлі, записів фінансування закупівлі, а також до всіх сутностей фонду. Політику заборони видалення (Restrict) застосовано до посилань на постачальника, благодійника, підрозділ, автора запису та позицію номенклатури в актах приймання й видачі. Політику обнулення посилання (SetNull) застосовано до зв'язку закупівлі із заявкою та до посилання на відповідального користувача. Перевірку цих обмежень виконує сама система керування базами даних.

Між закупівлями та надходженнями реалізовано зв'язок «багато до багатьох» із додатковим атрибутом суми. Для цього в схему введено окрему зв'язувальну сутність ProcurementFunding, фрагмент якої разом із сутністю закупівлі наведено на рисунку 3.2.



```
model Procurement {
  id           String           @id @default(cuid())
  foundationId String
  number       String
  supplierId   String
  requestId    String?
  status       ProcurementStatus @default(DRAFT)
  totalAmount  Decimal           @db.Decimal(14, 2)
  orderedAt    DateTime          @db.Date
  createdById  String
  createdAt    DateTime          @default(now())
  updatedAt    DateTime          @updatedAt

  foundation Foundation   @relation(fields: [foundationId], references: [id], onDelete: Cascade)
  supplier   Counterparty @relation(fields: [supplierId], references: [id], onDelete: Restrict)
  request    Request?     @relation(fields: [requestId], references: [id], onDelete: SetNull)
  createdBy  User         @relation("ProcurementCreatedBy", fields: [createdById], references: [id], onDelete: Restrict)

  lines         ProcurementLine[]
  funding       ProcurementFunding[]
  goodsReceipts GoodsReceipt[]
  files         File[]

  @@unique([foundationId, number])
  @@index([foundationId, status])
  @@index([supplierId])
  @@index([requestId])
}

model ProcurementFunding {
  id              String  @id @default(cuid())
  procurementId   String
  contributionId  String
  allocatedAmount Decimal @db.Decimal(14, 2)

  procurement  Procurement  @relation(fields: [procurementId], references: [id], onDelete: Cascade)
  contribution Contribution @relation(fields: [contributionId], references: [id], onDelete: Cascade)

  @@unique([procurementId, contributionId])
  @@index([contributionId])
}
```

Рисунок 3.2 – Фрагмент схеми Prisma (сутності Procurement та ProcurementFunding)

Як показано на рисунку 3.2, сутність ProcurementFunding містить посилання на закупівлю, посилання на надходження та суму, виділену з цього надходження на цю закупівлю (атрибут allocatedAmount), а складений унікальний ключ з procurementId та contributionId діє на пару «закупівля – надходження». На цьому самому фрагменті відображено описані політики видалення: посилання закупівлі на фонд є каскадним, посилання на постачальника – захищеним від видалення, а необов'язкове посилання на заявку – обнуленим.

Для метаданих змінної структури застосовано тип JSON. Його використано у двох випадках: для атрибута changes у журналі аудиту, що

зберігає знімок змінених полів, склад яких залежить від типу сутності, та для атрибута snapshot у сутності звіту, що фіксує дані звіту на момент публікації. Грошові суми зберігаються у типі з фіксованою точністю (Decimal).

Звернення до Prisma зосереджено в окремому шарі репозиторіїв, розміщеному в каталозі infrastructure/database/repos. Кожній сутності відповідає окремий клас репозиторію, позначений декоратором @Injectable, що залежить від служби доступу до бази даних. Сервіси доменних модулів звертаються до методів репозиторіїв, а не до клієнта Prisma безпосередньо. Фрагмент репозиторію закупівель наведено на рисунку 3.3.

```
@Injectable()
export class ProcurementRepository {
  constructor(private readonly prisma: PrismaService) {}

  create(input: CreateProcurementInput) {
    const { lines, funding, ...rest } = input;
    return this.prisma.procurement.create({
      data: { ...rest, lines: { create: lines }, funding: { create: funding } },
    });
  }

  replaceFunding(procurementId: string, funding: FundingInput[]) {
    return this.prisma.$transaction([
      this.prisma.procurementFunding.deleteMany({ where: { procurementId } }),
      this.prisma.procurementFunding.createMany({
        data: funding.map((f) => ({ ...f, procurementId })),
      }),
    ]);
  }
}
```

Рисунок 3.3 – Фрагмент репозиторію закупівель (ProcurementRepository)

Як видно з рисунка 3.3, репозиторій отримує службу доступу до бази даних через конструктор і надає методи, що оперують доменними поняттями. Метод створення формує закупівлю разом із її позиціями та

записами фінансування єдиною вкладеною операцією. Метод заміни фінансування виконує видалення попередніх записів і додавання нових у межах однієї транзакції.

Єдиною точкою з'єднання серверного коду з базою даних є інфраструктурний сервіс PrismaService. Цей сервіс успадковує згенерований клієнт Prisma та конфігурується адаптером @prisma/adapter-pg, якому передається рядок з'єднання з конфігурації застосунку. Сервіс інтегровано в життєвий цикл застосунку NestJS: з'єднання з базою даних встановлюється під час ініціалізації модуля, а роз'єднання виконується під час завершення роботи застосунку. PrismaService упроваджено в усі репозиторії.

Зміни схеми бази даних відстежуються механізмом міграцій Prisma. У середовищі розробки команда формування міграції створює та застосовує інструкції мовою визначення даних, зберігаючи їх в історії міграцій разом із кодом проєкту. У середовищі розгортання точка входу контейнера застосовує всі непримінені міграції перед запуском сервера. Початкове наповнення бази даних виконує окремий сценарій, який створює обліковий запис платформного адміністратора за даними зі змінних середовища; під час першого входу цей обліковий запис проходить процедуру налаштування двофакторної автентифікації.

3.3 Структура серверної частини та наскрізні механізми

Серверну частину реалізовано як модульний моноліт на фреймворку NestJS, що виконується поверх вебсервера Fastify. Кожна предметна область оформлена окремим модулем; усі модулі – автентифікації, користувачів, контрагентів, заявок, внесків, закупівель, складу, звітності, аудиту та інші – реєструються в кореневому модулі застосунку. У межах модуля код поділено на три рівні: контролер приймає запит HTTP і делегує його сервісу,

сервіс виконує бізнес-логіку, а доступ до даних здійснюється через шар репозиторіїв, описаний у підрозділі 3.2. Нижче розглянуто наскрізні (cross-cutting) механізми, спільні для всіх модулів.

3.3.1 Автентифікація

Система має дві незалежні гілки входу – для користувачів фондів і для адміністратора платформи, – що поділяють спільні служби роботи з токенами. Вхід виконується у два кроки. На першому кроці контролер приймає електронну пошту та пароль, сервіс перевіряє пароль за збереженим хешем і повертає виклик двофакторної автентифікації. Якщо двофакторну автентифікацію для облікового запису ще не налаштовано, виклик має стан SETUP і містить новий секрет, рядок otpauth та зображення QR-коду; якщо її вже налаштовано – стан VERIFY. На другому кроці клієнт надсилає шестизначний одноразовий код, сформований за алгоритмом TOTP, після чого видається пара токенів.

Тимчасовий код двофакторної автентифікації є випадковим значенням, спільним для сервера й застосунку-автентифікатора, на основі якого обидві сторони незалежно обчислюють однакові одноразові коди. Під час налаштування з цього секрету, найменування системи та електронної пошти облікового запису формується рядок схеми otpauth, який кодується у зображення QR-коду. Після зчитування рядка застосунок-автентифікатор зберігає секрет і кожні тридцять секунд обчислює новий шестизначний код за поточним часом, тоді як рядок для введення вручну подається як запасний спосіб перенесення секрету. Щоб усунути можливі розбіжності в показниках годинників сервера та пристрою користувача, під час перевірки коду допускається також значення із сусіднього тридцятисекундного проміжку. На рисунку 3.4 наведено функцію формування двохфакторної автентифікації.

```
async setupTwoFactor(dto: TwoFactorDto): Promise<AuthSessionResponse> {
  const payload = await this.ticket.verify(dto.ticket, 'SETUP', 'USER');
  if (
    !payload.secret ||
    !(await this.totp.verifyCode(payload.secret, dto.code))
  ) {
    throw new BadRequestException('Невірний код');
  }
  await this.users.setTotp(payload.sub, payload.secret, new Date());
  return this.issueSession(payload.sub);
}

private async buildChallenge(user: {
  id: string;
  email: string;
  totpEnabledAt: Date | null;
}): Promise<TwoFactorChallengeResponse> {
  const principal = { sub: user.id, type: 'USER' as const };
  if (user.totpEnabledAt) {
    return {
      stage: 'VERIFY',
      ticket: await this.ticket.signVerify(principal),
    };
  }
  const secret = this.totp.createSecret();
  const otpauthUri = this.totp.keyUri(user.email, secret);
  return {
    stage: 'SETUP',
    ticket: await this.ticket.signSetup(principal, secret),
    secret,
    otpauthUri,
    qrDataUrl: await this.totp.qr(otpauthUri),
  };
}
```

Рисунок 3.4 – Формування виклику двофакторної автентифікації та її налаштування

Як видно з рисунка 3.4, для облікового запису без налаштованої двофакторної автентифікації генерується секрет і формується QR-код, а під час підтвердження коду секрет зберігається й видається сеанс. Квиток між кроками є окремим токеном JWT, підписаним похідним секретом і позначеним станом (SETUP або VERIFY) та аудиторією (користувач або адміністратор), тому його не можна використати як токен доступу. Після успішного підтвердження видається токен доступу – підписаний JWT з ідентифікатором користувача, фонду та роллю – і токен оновлення – випадкове значення, що зберігається в базі даних у вигляді хешу. Під час

звернення до точки оновлення наявний токен оновлення відкликається й видається нова пара, тобто токени оновлення є одноразовими. На клієнті токен доступу додається до кожного запиту, а в разі завершення його строку дії сеанс прозоро поновлюється через токен оновлення з повторенням початкового запиту.

3.3.2 Авторизація за ролями та ізоляція клієнтів

Автентифікацію застосовано глобально: у кореневому модулі клас JwtAuthGuard зареєстровано як загальний guard застосунку, тому кожен запит, окрім позначених декоратором @Public, потребує дійсного токена доступу. Стратегія перевірки токена формує об'єкт-принципал: для користувача фонду він містить ідентифікатор користувача, ідентифікатор фонду та роль, а для адміністратора платформи – лише ідентифікатор. Розмежування доступу за ролями виконує окремий клас RolesGuard, наведений на рисунку 3.5, який застосовується до окремих маршрутів разом із декоратором @Roles.

```
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const required = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!required || required.length === 0) {
      return true;
    }
    const { user } = context
      .switchToHttp()
      .getRequest<{ user?: AuthenticatedPrincipal }>();
    return !!user && user.type === 'USER' && required.includes(user.role);
  }
}
```

Рисунок 3.5 – клас розмежування доступу за ролями

Метод canActivate зчитує перелік дозволених ролей, оголошений декоратором на методі або контролері, і пропускає запит лише тоді, коли принципал належить користувачеві фонду, роль якого міститься в цьому переліку. Маршрути зміни даних оголошують ролі без аудитора, унаслідок чого аудитор має доступ лише для читання. Ідентифікатор фонду береться виключно з токена, а не з вхідних даних запиту: декоратор @CurrentFoundationUser надає принципал обробникові, а сервіси використовують його ідентифікатор фонду для всіх операцій і перевіряють належність кожного запису тому самому фондові перед його зміною. Так у межах єдиного розгортання досягається ізоляція даних між фондами.

3.3.3 Перевірка вхідних даних

Перевірку вхідних даних застосовано глобально через стандартний механізм валідації NestJS, налаштований у точці входу з параметрами whitelist, forbidNonWhitelisted та transform. Унаслідок цього у вхідних структурах відкидаються поля, не оголошені у відповідному об'єкті передавання даних (DTO), а наявність зайвих полів спричиняє помилку. Самі правила перевірки оголошуються декораторами бібліотеки class-validator безпосередньо у полях об'єктів передавання даних – наприклад, перевірка формату електронної пошти та непорожнього рядка пароля. Помилкові запити відхиляються з кодом стану 400 та узгодженим описом помилки. Бізнес-помилки сервіси повідомляють через стандартні винятки NestJS – некоректний запит, не знайдено, відмова в доступі, – що формують єдину структуру відповіді.

3.3.4 Транзакційність і журнал аудиту

Групи пов'язаних операцій, що мають виконуватися неподільно, обгорнуто у транзакцію бази даних засобом prisma.$transaction. Так

виконуються, зокрема, створення фонду разом із першим обліковим записом адміністратора та запрошенням, а також заміна позицій і записів фінансування закупівлі. Кожна вагома зміна стану супроводжується записом до журналу аудиту через відповідний репозиторій. Запис журналу містить ідентифікатор фонду, ідентифікатор виконавця, дію, тип і ідентифікатор сутності, короткий опис, а також необов'язковий знімок змінених полів у форматі JSON. Журнал є доповнюваним: записи лише додаються й не змінюються, що формує наскрізний аудиторський слід операцій фонду.

3.3.5 Документація прикладного інтерфейсу

Документація прикладного інтерфейсу формується автоматично засобом NestJS Swagger на основі декораторів контролерів та об'єктів передавання даних.

Конфігурацію документації побудовано за принципом послідовного складання: на об'єкті-будівнику почергово задаються назва, опис і версія системи, схема авторизації за токеном, а також перелік тематичних груп. Кожна така група відповідає окремій функційній ділянці системи – від аутентифікації та керування користувачами до обліку внесків, закупівель, складського руху й звітності, – тому впорядкування методів у документації повторює модульну будову застосунку. Оскільки опис формується з тих самих декораторів та об'єктів передавання даних, що керують перевіркою вхідних запитів, він має єдине джерело узгодженості й не потребує окремого ручного супроводу. Схема авторизації за токеном дає змогу одноразово пройти підтвердження в інтерактивному інтерфейсі та звертатися до захищених методів безпосередньо під час ознайомлення з документацією. Ініціалізацію документації наведено на рисунку 3.6.

```
const swaggerConfig = new DocumentBuilder()
  .setTitle(
    'Система управління операційною діяльністю волонтерських фондів · API',
  )
  .setDescription('Операційна платформа волонтерських фондів')
  .setVersion('1.0')
  .addBearerAuth()
  .addTag('Auth', 'Аутентифікація користувачів фонду')
  .addTag('Platform', 'Аутентифікація суперадміністратора платформи')
  .addTag('Foundations', 'Створення та перелік фондів-клієнтів')
  .addTag('Users', 'Запрошення та керування користувачами фонду')
  .addTag('Counterparties', 'Контрагенти: підрозділи, донори, постачальники')
  .addTag('Requests', 'Заявки від військових підрозділів')
  .addTag('Files', 'Вкладення (локальне сховище)')
  .addTag('Contributions', 'Благодійні внески: грошові та натуральні')
  .addTag('Procurements', 'Закупівлі та замовлення у постачальників')
  .addTag('Warehouses', 'Склади фонду')
  .addTag('Stock', 'Склад: залишки, рух ТМЦ, прийом і видача')
  .addTag('Reports', 'Звітність: внутрішні та публічні звіти')
  .addTag('Audit', 'Аудит та історія: журнал усіх змін у фонді')
  .addTag('Dashboard', 'Дашборд: зведені показники фонду')
  .addTag('Nav', 'Навігація: лічильники активних розділів')
  .addTag('Search', 'Глобальний пошук по записах фонду')
  .build();
const document = SwaggerModule.createDocument(app, swaggerConfig);
SwaggerModule.setup('docs', app, document);
```

Рисунок 3.6 – Ініціалізація Swagger

На рисунку 3.6 описано заголовок, версію, схему авторизації за токеном та групи розділів, після чого документ публікується за адресою /docs у вигляді інтерактивного інтерфейсу та за адресою /docs-json у вигляді опису OpenAPI. Кожен метод контролера супроводжується анотаціями підсумку та типу відповіді, тому опис інтерфейсу залишається синхронним із реалізацією.

3.4 Реалізація операційного процесу

Цей підрозділ демонструє взаємодію модулів на прикладі наскрізного операційного процесу, описаного в першому розділі: від реєстрації заявки підрозділу до видачі матеріальних цінностей. Процес розглянуто послідовно у чотири етапи.

3.4.1 Реєстрація заявки підрозділу

Процес починається з реєстрації заявки від військового підрозділу. Координатор створює заявку, у якій зазначає підрозділ-замовник (контрагент типу «підрозділ»), контактну особу, пріоритет, канал надходження та перелік позицій із зазначенням найменування, кількості, одиниці виміру та за потреби технічних характеристик. Кожна позиція може посилатися на номенклатуру фонду або залишатися довільним описом. Заявці присвоюється номер, унікальний у межах фонду, і початковий статус «нова». Створення заявки супроводжується записом до журналу аудиту.

3.4.2 Формування закупівлі та її фінансування

На підставі заявки формується закупівля у постачальника. Сервіс закупівель перевіряє, що вказаний контрагент є постачальником, і за наявності прив'язує закупівлю до заявки. Для кожної позиції обчислюється сума рядка як добуток кількості та ціни, а загальна сума закупівлі – як їх підсумок. Окремо зазначаються джерела фінансування – перелік благодійних внесків і виділених із них сум, що зберігаються у зв'язувальній сутності фінансування, описаній у підрозділі 3.2. Перед збереженням сервіс перевіряє, що сума, виділена з кожного внеску, не перевищує його нерозподіленого залишку; для цього обчислюється сума вже виділених із цього внеску коштів. Закупівля разом із позиціями та записами фінансування створюється єдиною вкладеною операцією. Статус закупівлі змінюється за визначеною послідовністю, а кожна зміна фіксується в журналі аудиту.

3.4.3 Приймання на склад

Після отримання товару від постачальника виконується приймання на склад, реалізоване у сервісі приймання. Сервіс перевіряє, що закупівля перебуває у стані, придатному для приймання (замовлено або оплачено), і що кожна прийнята позиція належить номенклатурі фонду.

Як видно з рисунка 3.7, спершу створюється акт приймання з його позиціями, після чого для кожної позиції формується вхідний рух матеріальних цінностей (тип IN) та збільшується залишок відповідної позиції на складі. Після оприбуткування всіх позицій закупівлі присвоюється статус «отримано», а операція фіксується в журналі аудиту. Кожен запис руху зберігає посилання на акт приймання, що дозволяє простежити походження залишку.

```
const receipt = await this.receipts.create({
  foundationId: actor.foundationId,
  procurementId,
  warehouseId: warehouse.id,
  receivedById: actor.sub,
  note: dto.note?.trim() || null,
  lines: receiptLines,
});

const numbers = await this.movements.nextNumbers(
  actor.foundationId,
  receiptLines.length,
);
for (const [index, line] of receiptLines.entries()) {
  await this.movements.create({
    foundationId: actor.foundationId,
    number: numbers[index],
    type: MovementType.IN,
    itemId: line.itemId,
    warehouseId: warehouse.id,
    quantity: line.quantity,
    performedById: actor.sub,
    goodsReceiptId: receipt.id,
  });
  await this.stockLevels.increment(
    line.itemId,
    warehouse.id,
    line.quantity,
  );
}

await this.procurements.setStatus(
  procurementId,
  ProcurementStatus.RECEIVED,
);
await this.audits.create({
  foundationId: actor.foundationId,
  actorId: actor.sub,
  action: 'Прийнято на склад',
  targetType: 'PROCUREMENT',
  targetId: procurementId,
  summary: `Прийнято на склад «${warehouse.name}» (${receiptLines.length} поз.)`,
});
```

Рисунок 3.7 – Операція приймання на склад

3.4.4 Видача за заявкою

Завершальним етапом є видача матеріальних цінностей за заявкою, реалізована у сервісі видачі. Перед видачею сервіс перевіряє, що заявка не закрита й не відхилена, та що на складі достатньо кожної позиції. Логіку видачі та перерахунку статусу заявки наведено на рисунку 3.8.

```
function nextStatus(lines: { quantity: number; receivedQuantity: number }[]) {
  const anyReceived = lines.some((l) => l.receivedQuantity > 0);
  const allReceived = lines.every((l) => l.receivedQuantity >= l.quantity);
  if (allReceived) return RequestStatus.FULFILLED;
  if (anyReceived) return RequestStatus.PARTIALLY_FULFILLED;
  return null;
}
```

```
    for (const [index, line] of dto.lines.entries()) {
      await this.movements.create({
        foundationId: actor.foundationId,
        number: numbers[index],
        type: MovementType.OUT,
        itemId: line.itemId,
        warehouseId: warehouse.id,
        quantity: line.quantity,
        performedById: actor.sub,
        issuanceId: issuance.id,
      });
      await this.stockLevels.decrement(
        line.itemId,
        warehouse.id,
        line.quantity,
      );
      await this.fulfilRequestLine(request, line.itemId, line.quantity);
```

Рисунок 3.8 – Видача зі складу та перерахунок статусу заявки

Як видно з рисунка 3.8, для кожної позиції формується вихідний рух матеріальних цінностей (тип OUT), зменшується відповідний залишок на складі та збільшується отримана кількість у позиції заявки. Після видачі статус заявки перераховується скінченним автоматом: якщо отримано всі позиції повністю, заявці присвоюється статус «виконано», якщо отримано лише частину – «частково виконано». Операція видачі фіксується в журналі аудиту із зазначенням номера заявки та складу.

3.5 Реалізація додаткових підсистем

Окрім основного операційного процесу, система містить кілька допоміжних підсистем. Підсистема звітності формує два види звітів. Внутрішні звіти будуються конструктором за обраним типом, періодом і розрізами та вивантажуються у форматах XLSX, PDF або CSV (бібліотеки exceljs та pdfkit); сформований файл зберігається як вкладення звіту. Публічний звіт для донорів формується інакше: на момент публікації обчислюється знімок показників за період, який зберігається у полі snapshot сутності звіту разом із унікальним посиланням (publicSlug), після чого звіт стає доступним за відкритим посиланням без автентифікації. Формування та віддавання публічного звіту наведено на рисунку 3.9.

```
async publishPublic(
  actor: UserPrincipal,
  dto: PublicReportDto,
): Promise<{ slug: string }> {
  const snapshot = await this.publicReports.compute(
    actor.foundationId,
    new Date(dto.periodStart),
    endOfDay(dto.periodEnd),
    dto.sections,
  );
  const slug = randomUUID();
  const title = `Публічний звіт · ${formatRange(dto.periodStart, dto.periodEnd)}`;
  const report = await this.reports.create({
    foundationId: actor.foundationId,
    title,
    kind: ReportKind.PUBLIC,
    format: ReportFormat.HTML,
    periodStart: new Date(dto.periodStart),
    periodEnd: new Date(dto.periodEnd),
    generatedById: actor.sub,
    isPublished: true,
    publishedAt: new Date(),
    publicSlug: slug,
    snapshot,
  });
  await this.audit(actor, report.id, 'Опубліковано звіт', title);
  return { slug };
}

async getPublic(slug: string): Promise<PublicReportResponse> {
  const report = await this.reports.findBySlug(slug);
  if (!report || !report.isPublished) {
    throw new NotFoundException('Звіт не знайдено');
  }
  return {
    foundationName: report.foundation.name,
    periodStart: report.periodStart.toISOString(),
    periodEnd: report.periodEnd.toISOString(),
    snapshot: report.snapshot as PublicSnapshot,
  };
}
```

Рисунок 3.9 – Публікація та віддавання публічного звіту зі знімка

Як видно з рисунка 3.9, під час публікації обчислений знімок зберігається разом із посиланням і позначкою про публікацію, а відкрита сторінка віддає саме збережений знімок, а не поточні дані. Тому опублікований звіт лишається незмінним навіть після подальших змін у системі.

Підсистема глобального пошуку надає єдине поле пошуку і виконує пошук по записах фонду – заявках, контрагентах, внесках і закупівлях. Результати формуються запитом до бази даних і фільтруються за роллю користувача, тому відображаються лише розділи, доступні цій ролі. Підсистема файлів забезпечує прикріплення вкладень до заявок, внесків, закупівель і звітів: завантаження виконується через механізм багаточастинних запитів (@fastify/multipart), а самі файли зберігаються в каталозі файлової системи, який у контейнерному середовищі монтується як окремий том. Кожен файл зберігає посилання на пов'язану сутність, що дозволяє відобразити його у відповідній картці.

3.6 Реалізація клієнтської частини

Клієнтську частину реалізовано як односторінковий застосунок на бібліотеці React зі складанням засобом Vite. Код організовано за призначенням: каталог routes містить визначення маршрутів, features – реалізацію окремих розділів, components – багаторазові елементи інтерфейсу, а каталог lib – допоміжні модулі, серед яких типобезпечний клієнт прикладного інтерфейсу, сховища сеансу та правила розмежування доступу. Стилі оформлено на основі CSS-змінних без сторонніх бібліотек інтерфейсу.

Взаємодію із сервером інкапсульовано в окремому модулі-клієнті. Узагальнена функція виконує запит за відносним шляхом, додає до нього токен доступу та розбирає відповідь у типізований результат, а доменні

модулі (заявки, закупівлі, звіти тощо) надають типізовані функції поверх неї. Поновлення сеансу вбудовано в цю функцію: у разі відповіді про завершення строку дії токена (код 401) виконується одноразове поновлення пари токенів і повторення початкового запиту. Узагальнену функцію-клієнт та приклад доменної функції наведено на рисунку 3.10.

```
export async function apiFetch<T>(path: string, options: RequestOptions = {}): Promise<T> {
  let response = await send(path, options, options.token)

  if (
    response.status === 401 &&
    options.token &&
    path !== '/auth/refresh' &&
    authHooks?.getRefreshToken()
  ) {
    const newToken = await refreshAccessToken()
    if (newToken) {
      response = await send(path, options, newToken)
    } else {
      authHooks.onAuthFailure()
    }
  }

  if (!response.ok) {
    throw new ApiError(response.status, await extractMessage(response))
  }
  if (response.status === 204) {
    return undefined as T
  }
  return (await response.json()) as T
}
```

Рисунок 3.10 – Типізований клієнт прикладного інтерфейсу

Як видно з рисунка 3.10, у разі коду 401 функція звертається до точки оновлення, отримує нову пару токенів і повторює запит уже з новим токеном, а у разі невдачі повідомляє про завершення сеансу. Доменні функції лише задають шлях, метод і тип результату, делегуючи саме звернення узагальненій функції.

Маршрутизацію реалізовано засобом TanStack Router із файловим описом маршрутів. Захищені розділи згруповано під спільним маршрутом-каркасом, який перед завантаженням перевіряє наявність сеансу й переспрямовує неавтентифікованого користувача на сторінку входу, а в межах каркаса доступ до конкретного розділу додатково перевіряється за

роллю. Перелік розділів та дозволених ролей оголошено в модулі правил доступу, а функція canAccess визначає, чи доступний поточний шлях ролі користувача. Маршрут-каркас захищених розділів і перевірку доступу наведено на рисунку 3.11.

```
export const Route = createFileRoute('/_app')({
  beforeLoad: () => {
    if (!sessionStore.isAuthenticated()) {
      throw redirect({ to: '/login' })
    }
  },
  loader: async (): Promise<AppLayoutData> => {
    const token = sessionStore.getAccessToken()
    if (!token) return { counts: { requests: 0, procurements: 0 } }
    return { counts: await getNavCounts(token) }
  },
  component: AppLayout,
})

function AppLayout() {
  const { role } = useAuth()
  const pathname = useLocation({ select: (location) => location.pathname })
  const denied = role !== null && !canAccess(role, pathname)

  return <AppShell>{denied ? <AccessDenied /> : <Outlet />}</AppShell>
}
```

Рисунок 3.11 – Маршрут-каркас захищених розділів і перевірка доступу за роллю

Як видно з рисунка 3.11, маршрут-каркас переспрямовує користувача без сеансу на сторінку входу, а для автентифікованого користувача порівнює його роль із ролями, дозволеними для поточного розділу: за відсутності доступу замість вмісту відображається повідомлення про відмову. Дані для бічної панелі (лічильники активних розділів) завантажуються на рівні маршруту й кешуються механізмом TanStack Query.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі здійснено програмну реалізацію системи управління операційною діяльністю волонтерських фондів у військовій сфері на основі рішень, обґрунтованих у другому розділі.

Проєкт побудовано як монорепозиторій, а серверну частину структуровано за шаблоном модульного моноліту з поділом на рівні контролерів, сервісів і репозиторіїв. Рівень доступу до даних реалізовано на PostgreSQL та Prisma: описано модель даних із наскрізною мультиорендністю, реалізацію операційного графа зовнішніми ключами з диференційованими політиками видалення та зв'язувальну сутність фінансування закупівель.

Реалізовано наскрізні механізми серверної частини: двофакторну автентифікацію за алгоритмом TOTP, розмежування доступу за ролями та ізоляцію даних між фондами, перевірку вхідних даних, транзакційність групових операцій із наскрізним журналом аудиту та автоматичне формування документації прикладного інтерфейсу.

Реалізовано основний операційний процес – від реєстрації заявки підрозділу через формування й фінансування закупівлі та приймання на склад до видачі за заявкою – зі збереженням посилальної цілісності та автоматичною зміною статусів. Додатково реалізовано підсистеми звітності, глобального пошуку й файлових вкладень, а клієнтську частину – як односторінковий застосунок із типобезпечним клієнтом інтерфейсу, файловою маршрутизацією та розмежуванням за ролями.

Отже, у третьому розділі отримано працездатну програмну реалізацію системи, що забезпечує комплексну автоматизацію основних процесів операційної діяльності волонтерського фонду.

РОЗДІЛ 4 ОГЛЯД РОЗРОБЛЕНОЇ СИСТЕМИ

У розділі оглянуто основний функціонал розробленої системи з боку користувача; послідовно розглянуто ключові робочі області відповідно до операційного циклу фонду.

4.1 Автентифікація та керування доступом

Робота із системою можлива лише після входу до облікового запису: усі робочі розділи закриті для неавторизованого відвідувача, а перелік доступних розділів визначається роллю користувача. Вхід виконується за електронною поштою та паролем з обов'язковим двофакторним підтвердженням (2FA). Сторінку входу наведено на рисунку 4.1.

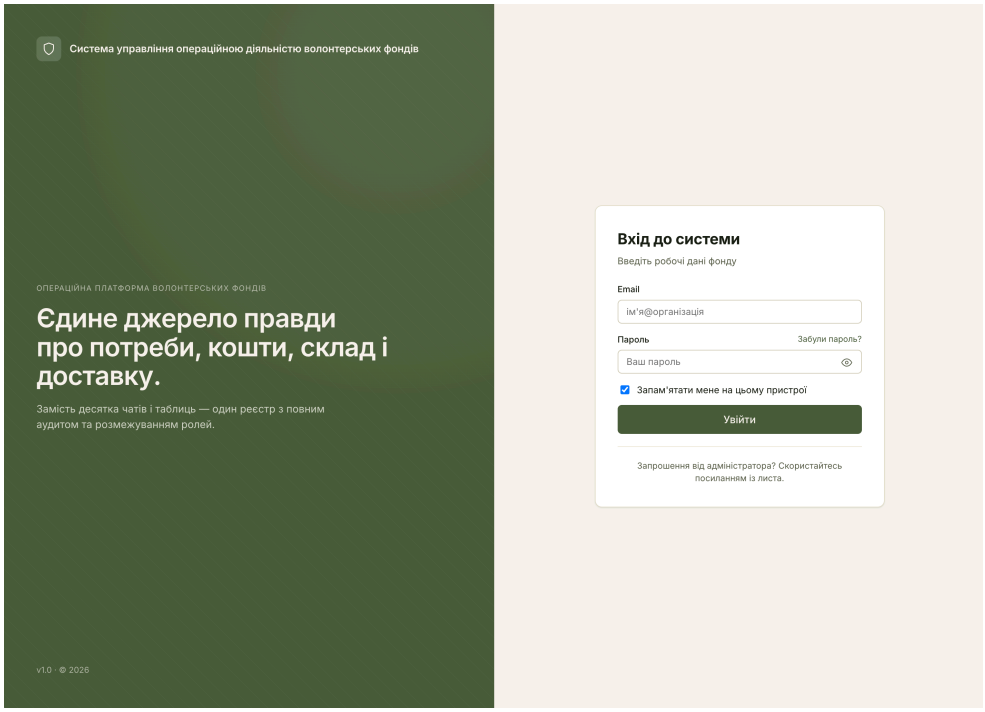


Рисунок 4.1 – Сторінка входу

Залежно від стану облікового запису другий крок входу має два варіанти. Якщо двофакторне підтвердження для запису ще не налаштовано,

система пропонує додати обліковий запис до застосунку-автентифікатора, відсканувавши наданий графічний код (QR-код), і ввести згенерований одноразовий код для завершення налаштування. Якщо підтвердження вже налаштовано, користувач лише вводить поточний одноразовий код зі свого застосунку-автентифікатора. Сторінку введення одноразового коду наведено на рисунку 4.2. Сеанс відкривається тільки після успішного підтвердження, тож знання самого пароля недостатнє для доступу до системи.

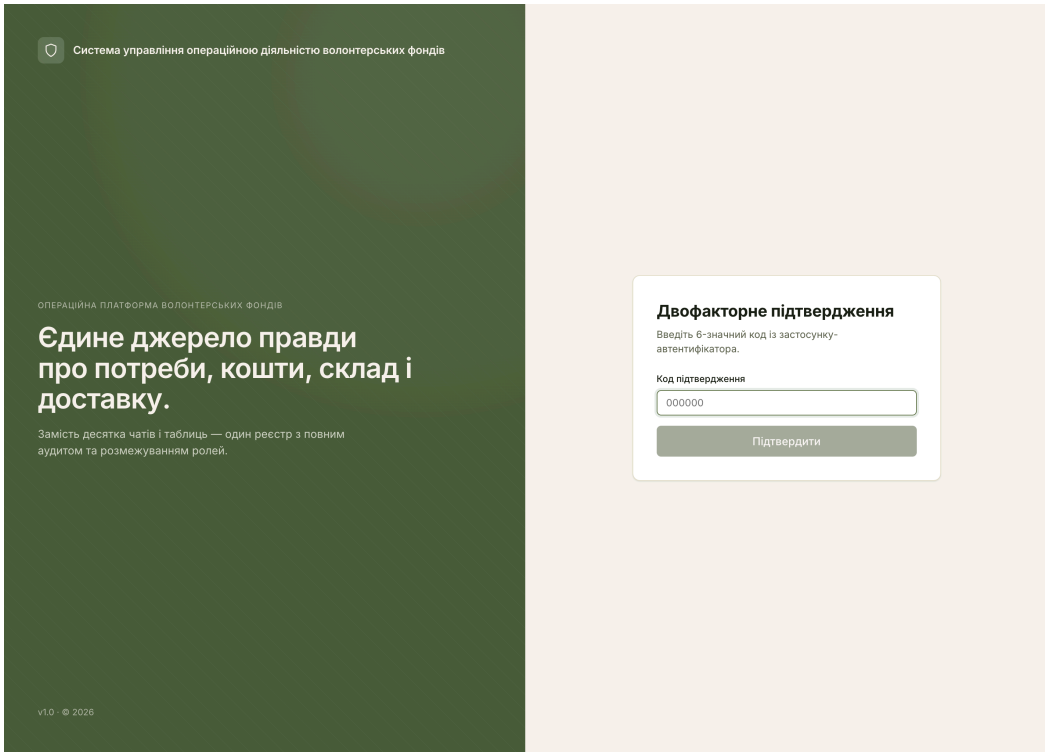


Рисунок 4.2 – Сторінка введення одноразового коду

Окремий обліковий запис у системі не реєструється самостійно: новий співробітник додається адміністратором фонду через запрошення, надіслане на електронну пошту із заздалегідь визначеною роллю. За посиланням із запрошення відкривається сторінка активації, де відображено назву фонду, призначену роль і адресу пошти, а від користувача вимагається лише задати власний пароль (не менше десяти символів із літерами та цифрами). Після цього обліковий запис стає активним. Сторінку активації за запрошенням

наведено на рисунку 4.3. Посилання із запрошення дійсне обмежений час; якщо строк його дії минув, активацію виконати неможливо, і запрошення потрібно надіслати повторно. На випадок втрати пароля передбачено його відновлення: за вказаною поштою надсилається посилання обмеженої дії для встановлення нового пароля.

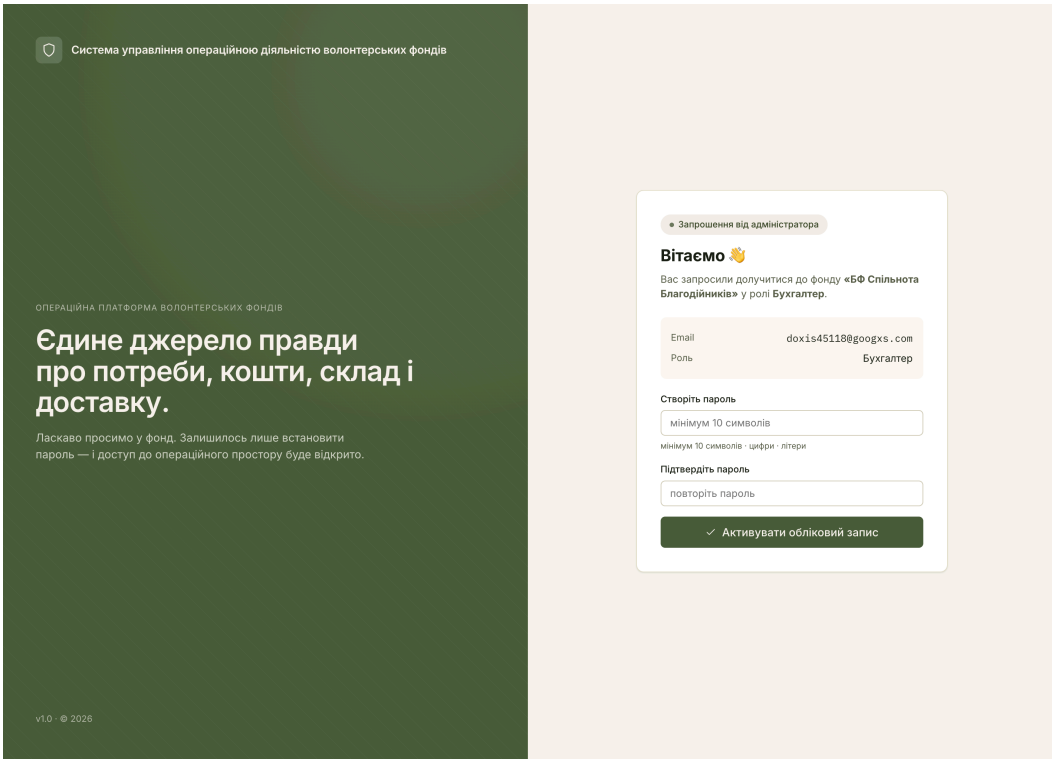


Рисунок 4.3 – Сторінка активації запрошення

Розмежування доступу побудовано на чотирьох ролях – адміністратор фонду, координатор, бухгалтер та аудитор, – і саме роль визначає, які розділи бічного меню та які дії доступні користувачеві. Координатор працює з операційними розділами (заявки, закупівлі, склад, контрагенти), але не має доступу до фінансового обліку надходжень. Бухгалтер, навпаки, відповідає за облік благодійних внесків, закупівлі та звітність, проте не опрацьовує заявки й не керує складом. Адміністратор має доступ до всіх розділів, включно з адмініструванням складу користувачів фонду. Роль аудитора є наскрізною й водночас доступною лише для перегляду: аудитор

бачить більшість розділів і журнал аудиту, але не може створювати чи змінювати записи. Така побудова забезпечує принцип найменших привілеїв – кожен користувач отримує рівно той обсяг доступу, який потрібен для виконання його функцій. Поза описаними ролями існує окремий вхід платформного адміністратора, призначений для керування переліком фондів, а не для операційної роботи всередині окремого фонду.

4.2 Зведена панель фонду

Після входу користувач потрапляє на зведену панель – стартову сторінку, що дає швидкий огляд поточного стану фонду без переходу до окремих розділів. Як заголовок панелі, так і набір поданих на ній показників залежать від ролі користувача, тож кожен співробітник бачить ті відомості, що відповідають його зоні відповідальності.

Для координатора панель має операційне спрямування й зведена на рисунку 4.4. На ній відображено кількість відкритих заявок із приростом за тиждень, кількість заявок у роботі з виокремленням позицій критичного пріоритету, кількість і суму закупівель у роботі, а також кількість позицій із низькими залишками, що потребують поповнення. Нижче розміщено перелік останніх заявок і стрічку останніх подій із журналу аудиту, що дає змогу одразу оцінити останню активність у фонді. У верхній частині панелі доступні кнопки швидких дій – створення нової заявки чи закупівлі та видача зі складу, – які скорочують шлях до найчастіших операцій.

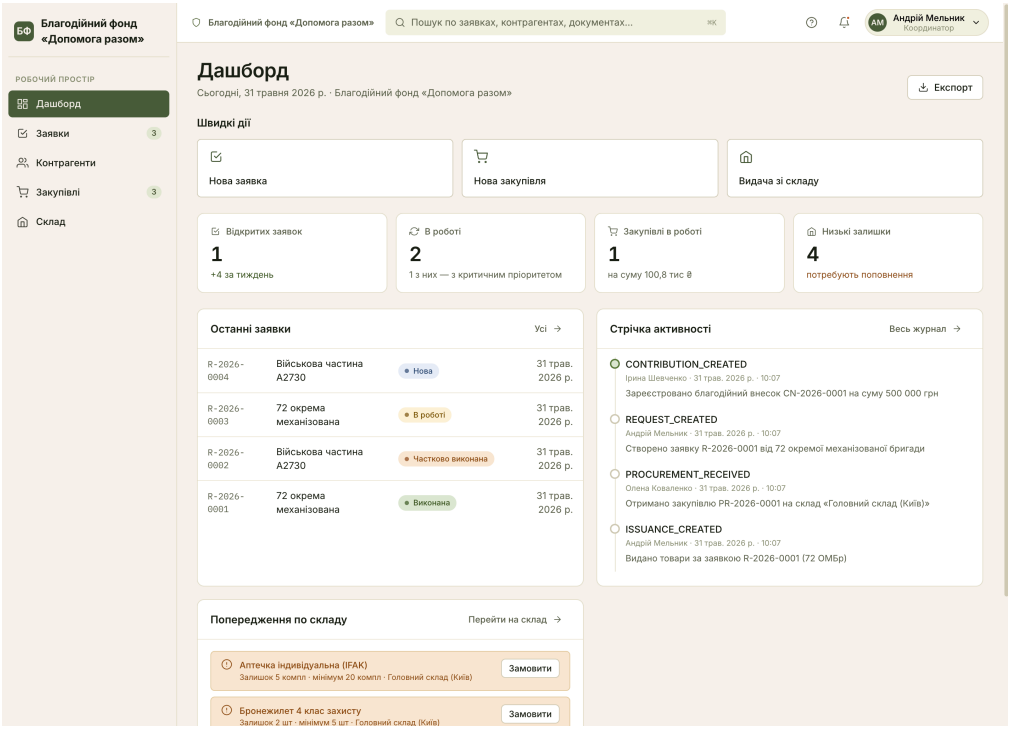


Рисунок 4.4 – Панель координатора

Для бухгалтера панель набуває фінансового вигляду й має заголовок «Фінансовий дашборд» (рисунок 4.5). Основні показники тут – сума благодійних надходжень за обраний місяць із зазначенням зміни щодо попереднього місяця, невеликий графік динаміки надходжень за дванадцять місяців, а також обсяг закупівель у роботі та перелік останніх надходжень. Серед швидких дій бухгалтерові доступна реєстрація нового внеску. Для показників, прив'язаних до періоду, передбачено вибір місяця, що дозволяє переглядати стан фонду за попередні періоди.

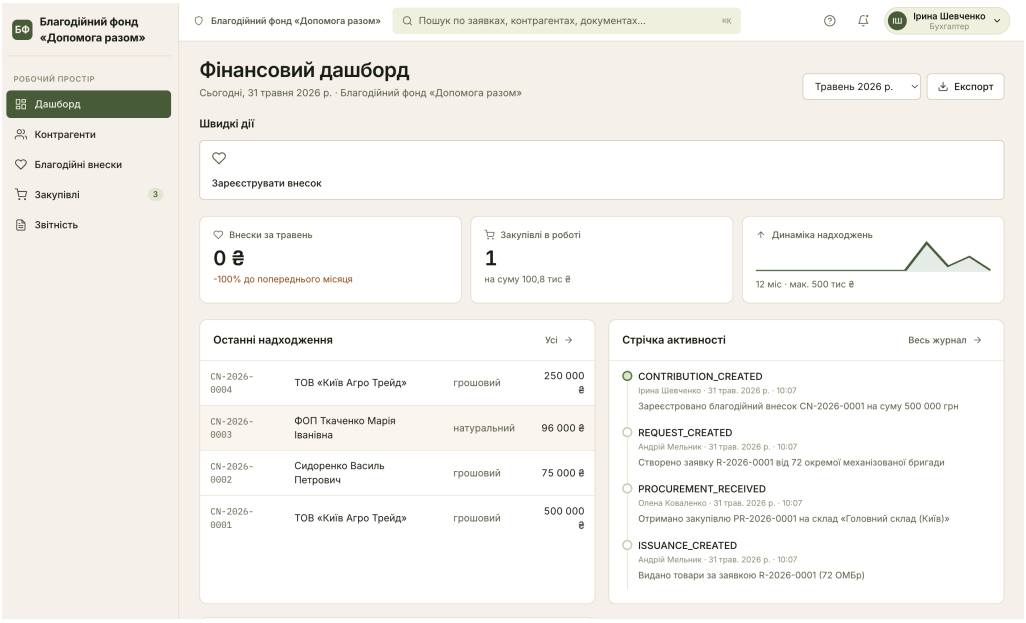


Рисунок 4.5 – Фінансовий дашборд

Подання панелі для решти ролей побудоване за тим самим принципом. Адміністратор бачить узагальнений набір операційних і фінансових показників, а для аудитора панель має заголовок «Огляд активності» й зосереджена на кількості операцій за період і стрічці подій без можливості внесення змін. Незалежно від ролі, зведені показники панелі можна вивантажити у форматі CSV для подальшого використання поза системою.

4.3 Облік контрагентів

Розділ контрагентів забезпечує ведення єдиної бази всіх сторін, з якими взаємодіє фонд. Контрагенти поділяються на три типи: військові підрозділи, що є отримувачами допомоги; благодійні партнери, що є джерелами надходжень; постачальники, у яких фонд здійснює закупівлі. Кожному контрагентові присвоюється власний обліковий код і заводиться картка з реквізитами – назвою, організаційно-правовою формою (юридична чи фізична особа, фізична особа-підприємець, неюридична особа), контактною особою, телефоном, електронною поштою, основним каналом

зв'язку (Signal, Telegram, електронна пошта чи телефон), приміткою та датою першого контакту. Перелік контрагентів наведено на рисунку 4.6; він підтримує фільтрування за типом і пошук, а керування записами (створення й редагування) доступне адміністраторові та координаторові, тоді як решта ролей мають доступ лише до перегляду.

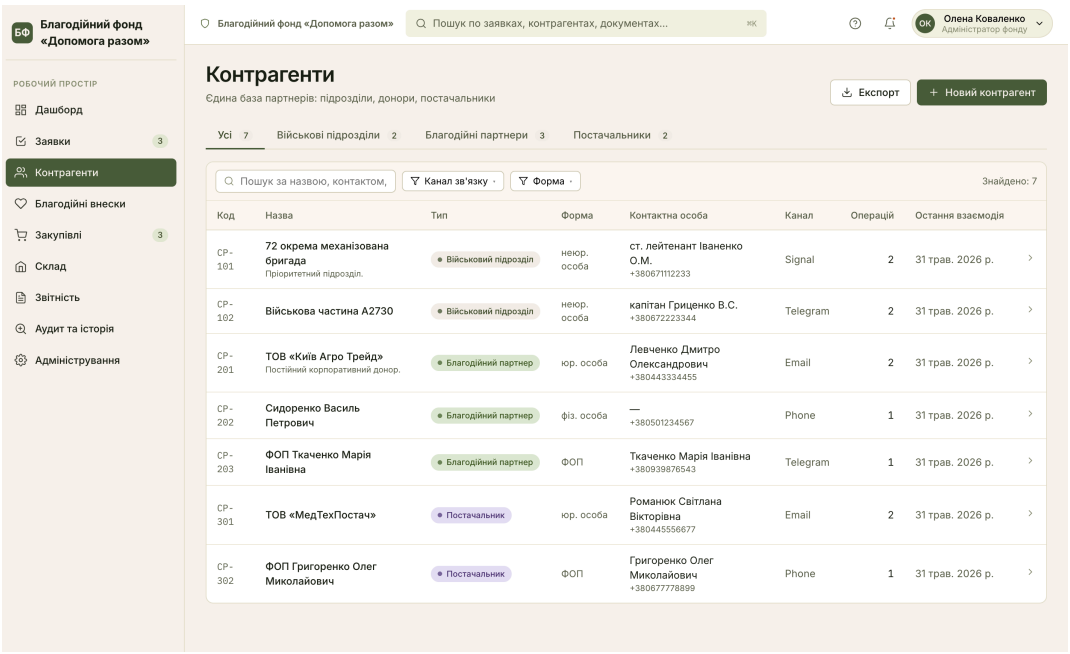


Рисунок 4.6 – Перелік контрагентів

Окрім реквізитів, картка контрагента відображає історію взаємодії з ним (рисунок 4.7): пов'язані заявки, надходження та закупівлі, а також кількість операцій і дату останньої взаємодії. Завдяки цьому з однієї картки видно всю співпрацю фонду з конкретною стороною. Єдина база контрагентів є опорною для решти розділів: на неї посилаються заявки (через підрозділ), надходження (через благодійника) і закупівлі (через постачальника), що усуває дублювання даних і забезпечує узгодженість обліку.

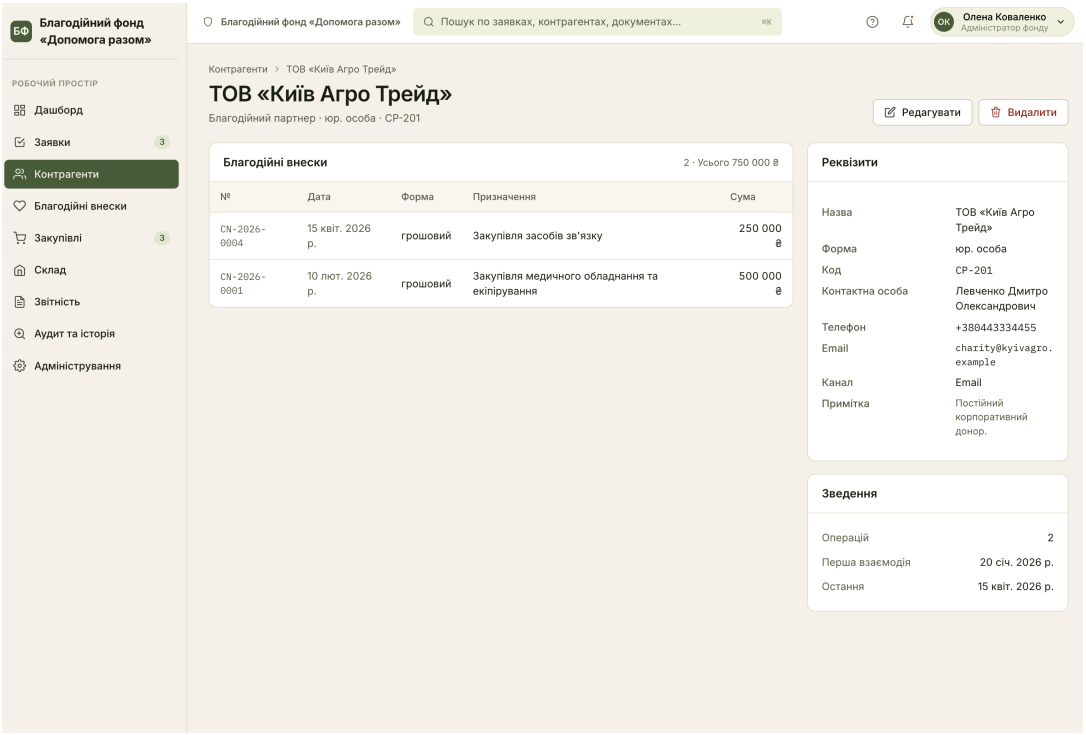


Рисунок 4.7 – Картка контрагента

Реєстр контрагентів можна вивантажити у форматі CSV для використання поза системою. Реквізити будь-якого запису доступні для редагування, при цьому раніше виконані з контрагентом операції залишаються незмінними.

4.4 Супроводження заявок підрозділів

Заявка фіксує звернення військового підрозділу щодо матеріально-технічного забезпечення. Під час реєстрації обирається підрозділ із бази контрагентів, зазначається його контактна особа, пріоритет (високий, середній або низький), за потреби – бажаний строк виконання, канал, яким надійшло звернення, і відповідальний працівник фонду. До заявки додається перелік потрібних позицій (щонайменше одна), для кожної з яких вказують назву, артикул, кількість, одиницю виміру та технічні характеристики. Звернення підрозділів часто містять неструктуровані додатки, тож до заявки можна прикріпити супровідні матеріали –

фотографії, документи й голосові повідомлення. Перелік заявок наведено на рисунку 4.8: він відображає номер, підрозділ, пріоритет, поточний статус, стислий склад позицій, орієнтовну вартість і строк виконання.

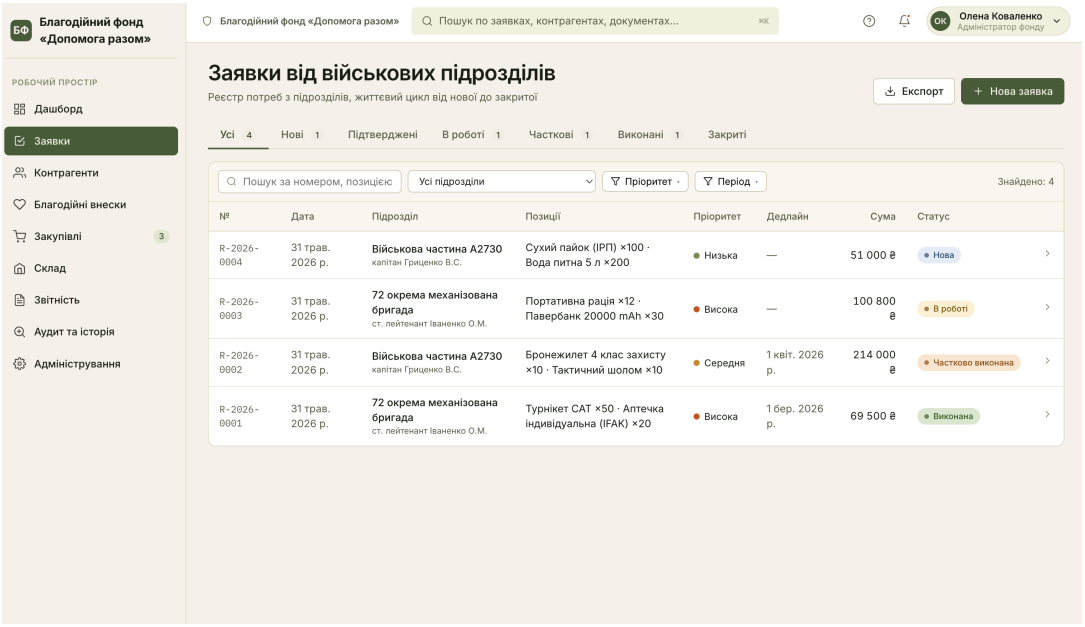


Рисунок 4.8 – Перелік заявок

Картка заявки (рисунок 4.9) подає повну інформацію про звернення – реквізити, перелік позицій із зазначенням уже отриманої кількості, пов'язані закупівлі та історію змін. Опрацювання заявки відбувається за визначеним життєвим циклом: від стану «нова» через підтвердження і взяття в роботу до повного або часткового виконання та закриття, із можливістю відхилення на проміжних етапах. Система допускає лише коректні переходи між станами, унеможливлюючи, наприклад, закриття заявки, яку ще не виконано; редагувати дозволено лише нові та підтверджені заявки. Статуси «виконано» й «частково виконано» виставляються автоматично за результатом видачі матеріальних цінностей зі складу, а кожна зміна стану фіксується в журналі аудиту.

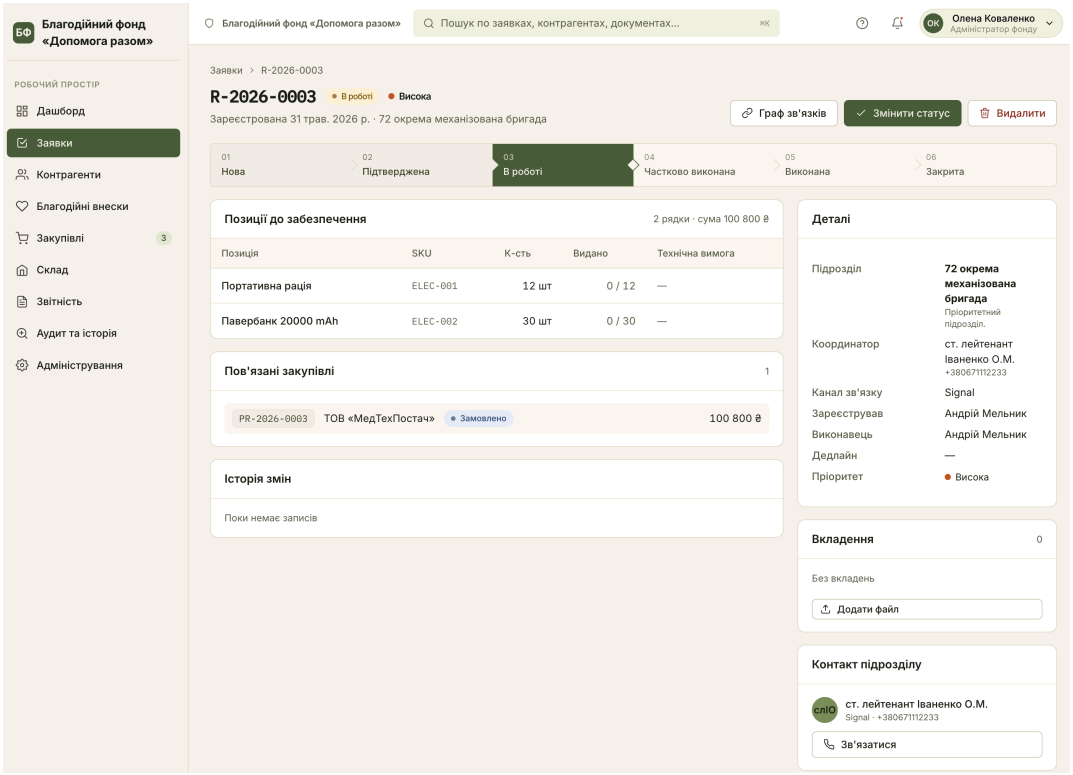


Рисунок 4.9 – Картка заявки

Кожній заявці система присвоює власний обліковий номер, за яким її зручно відстежувати в подальших операціях. Призначення відповідального працівника дає змогу розподіляти опрацювання звернень між координаторами фонду.

4.5 Облік благодійних надходжень

Розділ надходжень призначений для обліку допомоги, що надходить до фонду від благодійних партнерів. Передбачено дві форми надходжень – грошову та натуральну (товари чи послуги). Під час реєстрації (рисунок 4.10) обирається благодійник із бази контрагентів, зазначається форма надходження, а далі – сума й валюта для грошового внеску або найменування та кількість позиції для натурального; додатково вказуються призначення, документ-основа і дата надходження. Для підтвердження операції до запису прикріплюють відповідні документи – платіжні

доручення, договори про благодійну допомогу, акти приймання-передавання.

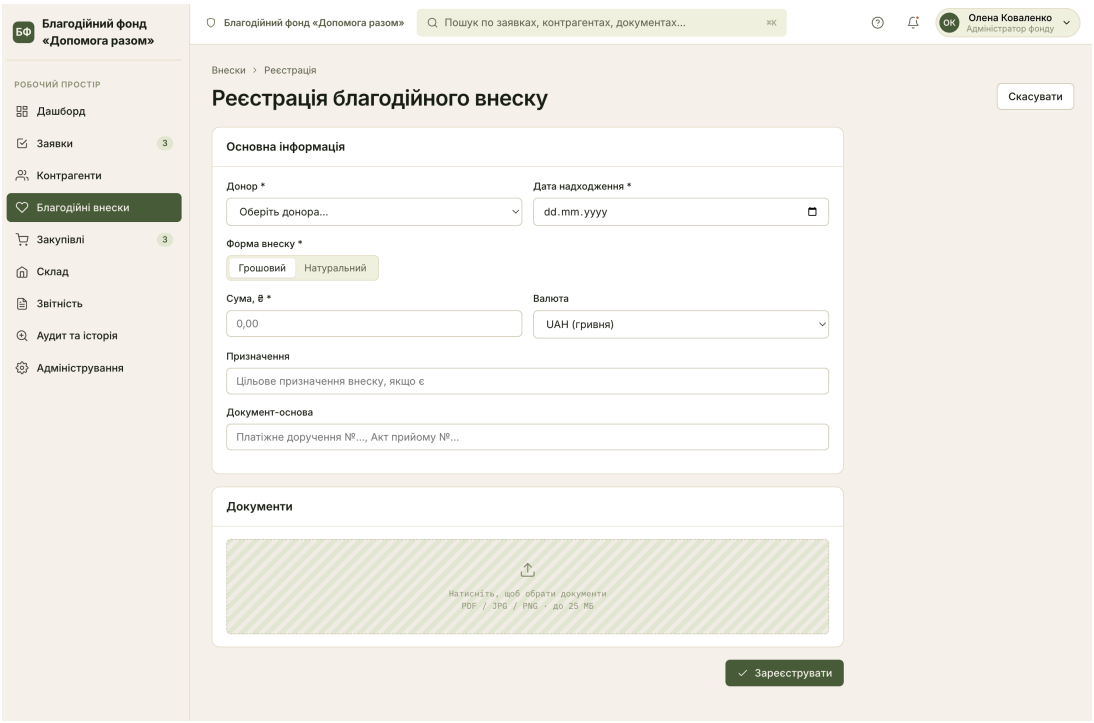


Рисунок 4.10 – Форма реєстрації благодійного внеску

Перелік надходжень наведено на рисунку 4.11; він відображає номер, благодійника, форму, суму, призначення, дату, а також кількість пов'язаних закупівель і вже розподілену суму. Картка надходження додатково показує нерозподілений залишок коштів і перелік закупівель, профінансованих за рахунок цього внеску. Такий зв'язок «надходження – закупівля» забезпечує цільове використання коштів і прозорість витрачання благодійної допомоги, а всі дії з надходженнями реєструються в журналі аудиту.

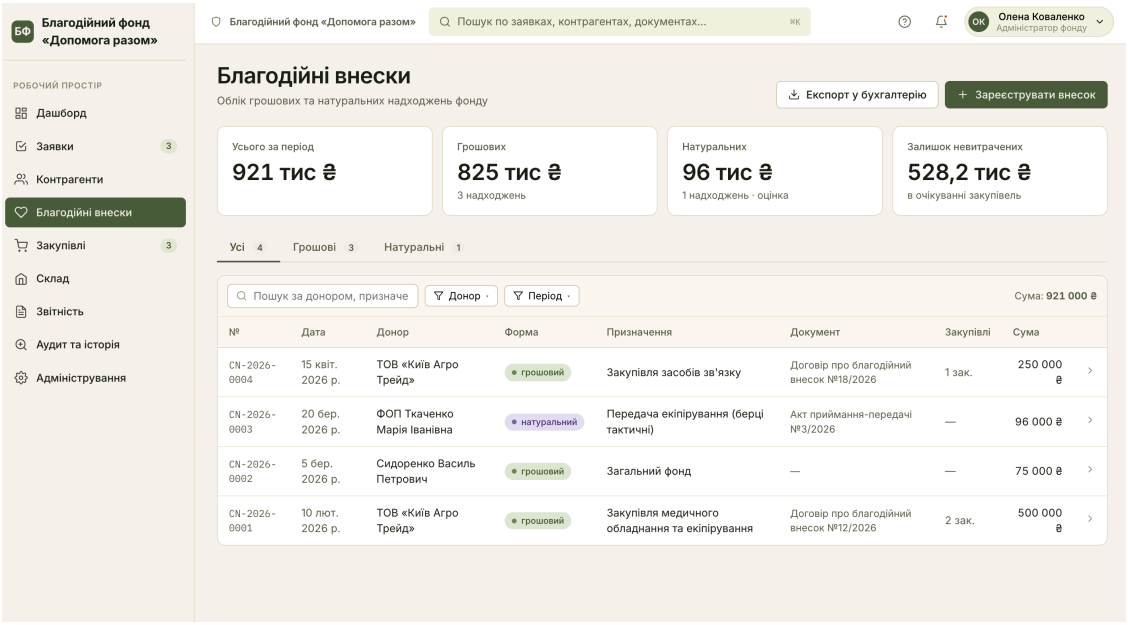


Рисунок 4.11 – Перелік благодійних внесків

Надходження обов'язково прив'язується до контрагента типу «благодійник» – контрагентів інших типів система для цього не приймає. Сума грошового внеску обліковується у вказаній валюті, а обсяг натурального визначається найменуванням і кількістю переданих позицій.

4.6 Закупівля та складський облік

Закупівля фіксує придбання матеріальних цінностей у постачальника. Під час її формування обирається постачальник із бази контрагентів, за потреби закупівля прив'язується до конкретної заявки, зазначається дата замовлення та перелік позицій, для кожної з яких вказують назву, артикул, кількість і ціну за одиницю; загальна сума обчислюється автоматично. Окремо задаються джерела фінансування – суми, що виділяються на цю закупівлю з конкретних грошових надходжень, завдяки чому видно, за рахунок яких внесків її оплачено. Закупівля переходить між станами – від чернетки до замовлення й оплати. Закупівлю з прив'язаним фінансуванням наведено на рисунку 4.12.

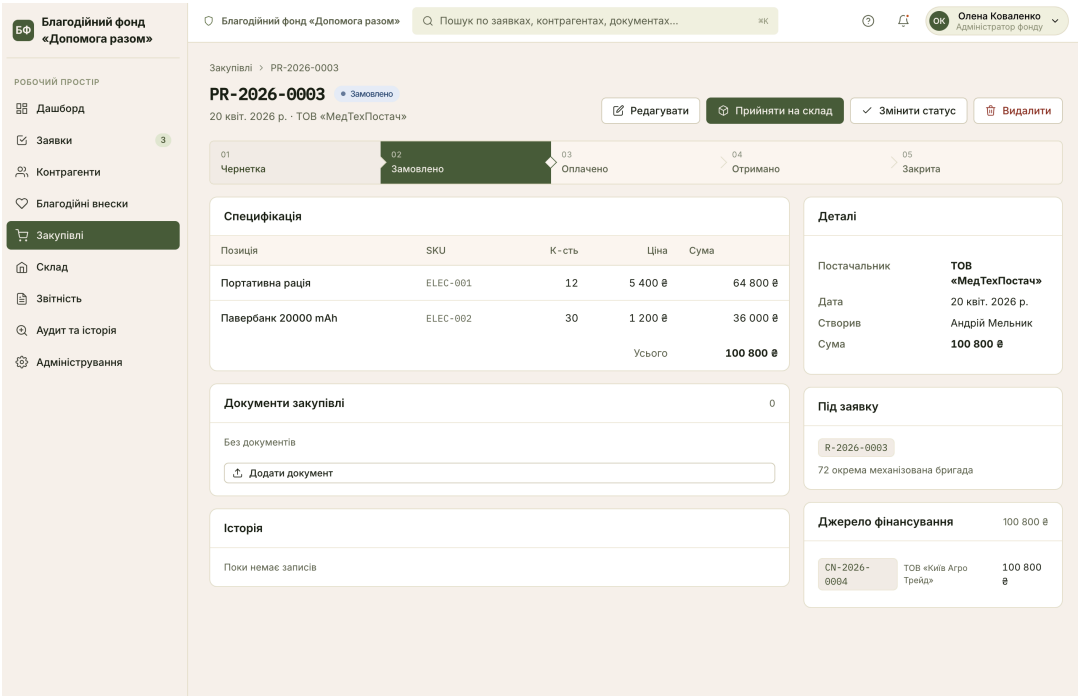


Рисунок 4.12 – Картка закупівлі

Після надходження придбаного виконується приймання на склад замовленої або оплаченої закупівлі: обирається склад і зазначаються прийняті позиції з кількістю. Під час приймання формуються вхідні рухи матеріальних цінностей, на відповідні позиції збільшуються складські залишки, а закупівля позначається як прийнята. Приймання на склад наведено на рисунку 4.13.

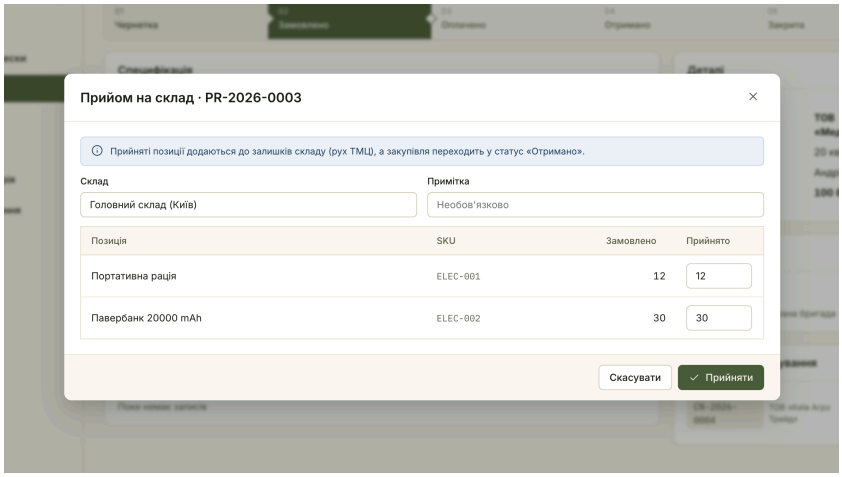


Рисунок 4.13 – Форма прийому на склад

Видача матеріальних цінностей виконується за відкритою заявкою підрозділу: обирається склад і позиції до видачі. Перед видачею система перевіряє, що заявку не закрито й не відхилено, а на складі достатньо кожної позиції, інакше операцію буде відхилено. За успішної видачі формуються вихідні рухи матеріальних цінностей, відповідні залишки зменшуються, у позиціях заявки зростає отримана кількість, а статус заявки автоматично перераховується на «частково виконано» або «виконано». Видачу за заявкою наведено на рисунку 4.14.

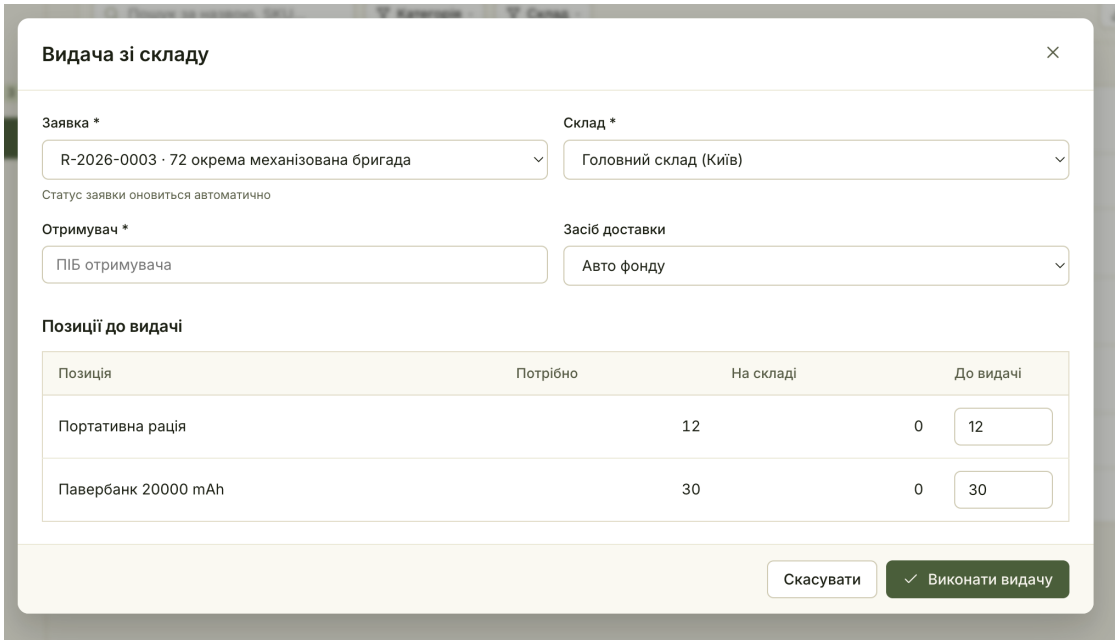


Рисунок 4.14 – Форма видачі за заявкою

Поточний стан складу подається в окремому розділі, який відображає залишки за позиціями та складами, а також рух матеріальних цінностей. Позиції, кількість яких опустилася нижче встановленого мінімального рівня, позначаються як низькі залишки й потребують поповнення. Залишки на складі наведено на рисунку 4.15.

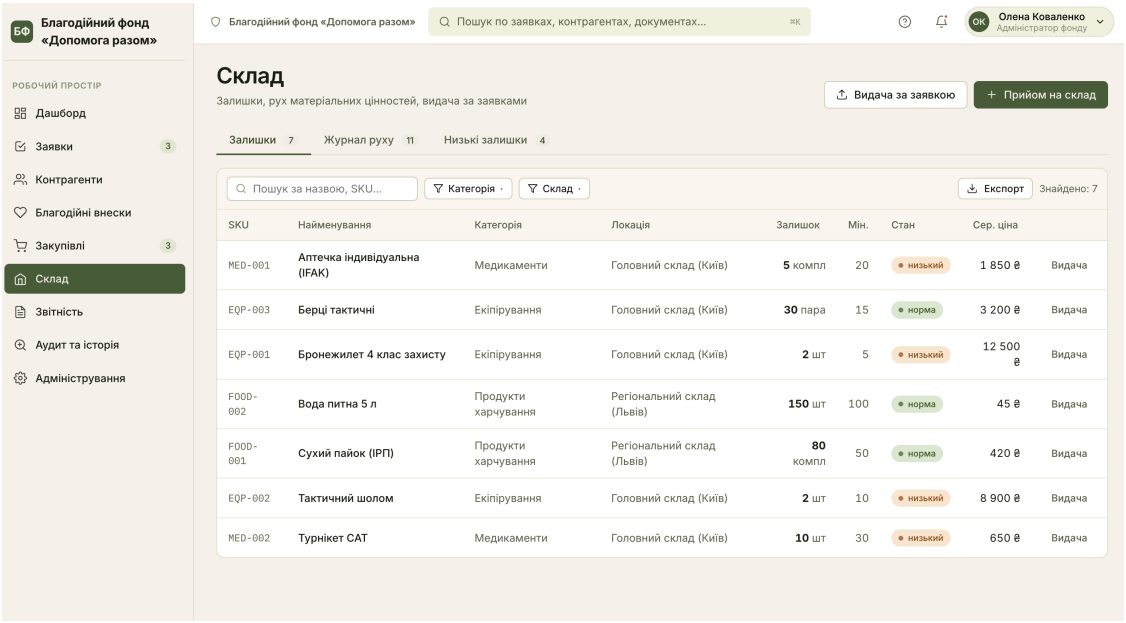


Рисунок 4.15 – Перелік залишків на складі

Кожному руху матеріальних цінностей присвоюється власний номер, за яким його можна відстежити в історії складу. Склад створюється автоматично за вказаною назвою під час першого приймання чи видачі, тож окремого попереднього налаштування складів не потрібно.

4.7 Формування звітності

Внутрішня звітність будується конструктором звітів. Користувач обирає тип звіту, звітний період, розріз даних і формат вивантаження – XLSX, PDF або CSV, – після чого система формує звіт і зберігає готовий файл як вкладення, доступне для подальшого завантаження. Раніше сформовані звіти зберігаються переліком із зазначенням назви, періоду, типу, дати формування, користувача та розміру файлу. Конструктор внутрішнього звіту наведено на рисунку 4.16.

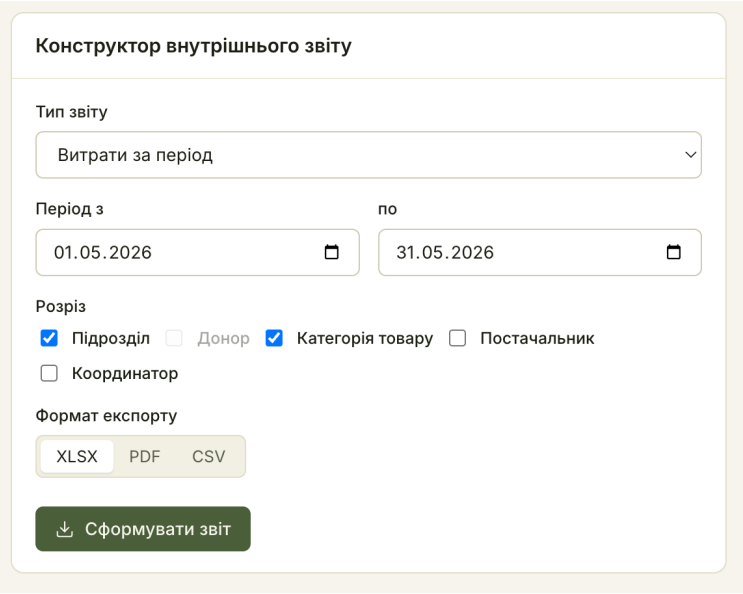


Рисунок 4.16 – Конструктор внутрішнього звіту

Окремо передбачено публічну звітність перед донорами. Для публічного звіту обирається звітний період і розділи, які він має містити (зокрема зібрані кошти, структура витрат, закриті потреби та благодійники), після чого звіт можна попередньо переглянути й опублікувати. На момент публікації обчислюється знімок показників за період, який зберігається разом з унікальним посиланням; після цього звіт стає доступним за відкритим посиланням без потреби входити в систему. Опублікований публічний звіт наведено на рисунку 4.17.

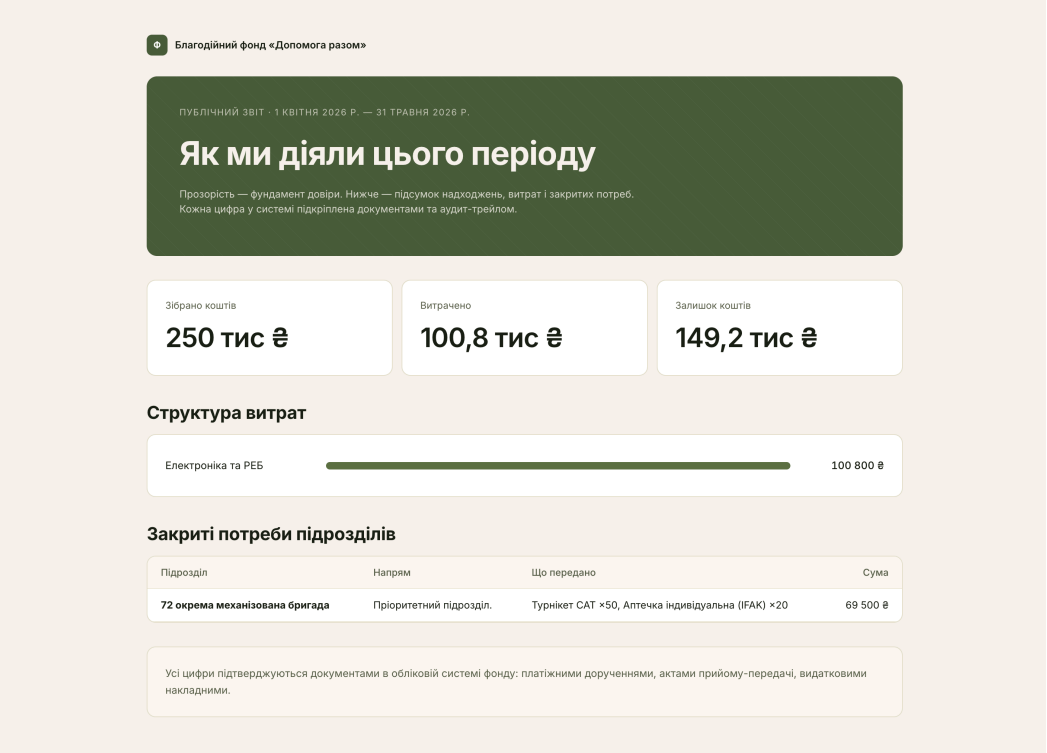


Рисунок 4.17 – Опублікований зовнішній звіт

Внутрішні звіти формуються в кількох типах і розрізах, що дає змогу отримувати дані під різні потреби фонду. Публічний звіт віддає саме збережений на момент публікації знімок, тому подальші операції в системі не змінюють уже оприлюднених показників.

4.8 Журнал аудиту та адміністрування

Журнал аудиту забезпечує простежуваність операцій, фіксуючи ключові зміни в системі – у заявках, надходженнях, закупівлях, рухах на складі, звітах, облікових записах користувачів, реквізитах фонду та довідниках. Кожен запис журналу містить виконавця дії, саму дію, тип і об'єкт, якого вона стосується, стислий опис і час. Записи журналу доступні лише для перегляду й не підлягають редагуванню, а доступ до нього мають адміністратор та аудитор. Журнал аудиту наведено на рисунку 4.18.

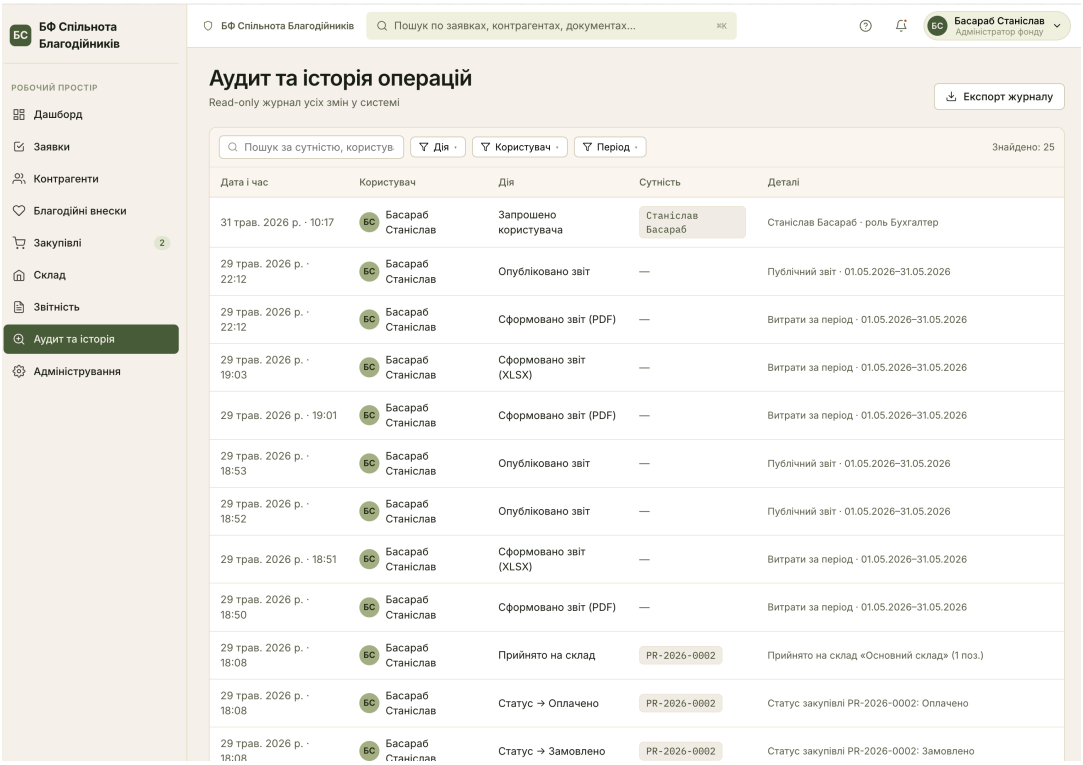


Рисунок 4.18 – Аудит логів та операцій

Розділ адміністрування, доступний лише адміністраторові фонду, призначений для керування складом користувачів. У ньому можна запросити нового співробітника, вказавши його електронну пошту й роль, змінити роль наявного користувача, заблокувати чи розблокувати обліковий запис, а також повторно надіслати запрошення. Кожен користувач має стан – активний, запрошений або заблокований. Керування користувачами фонду наведено на рисунку 4.19.

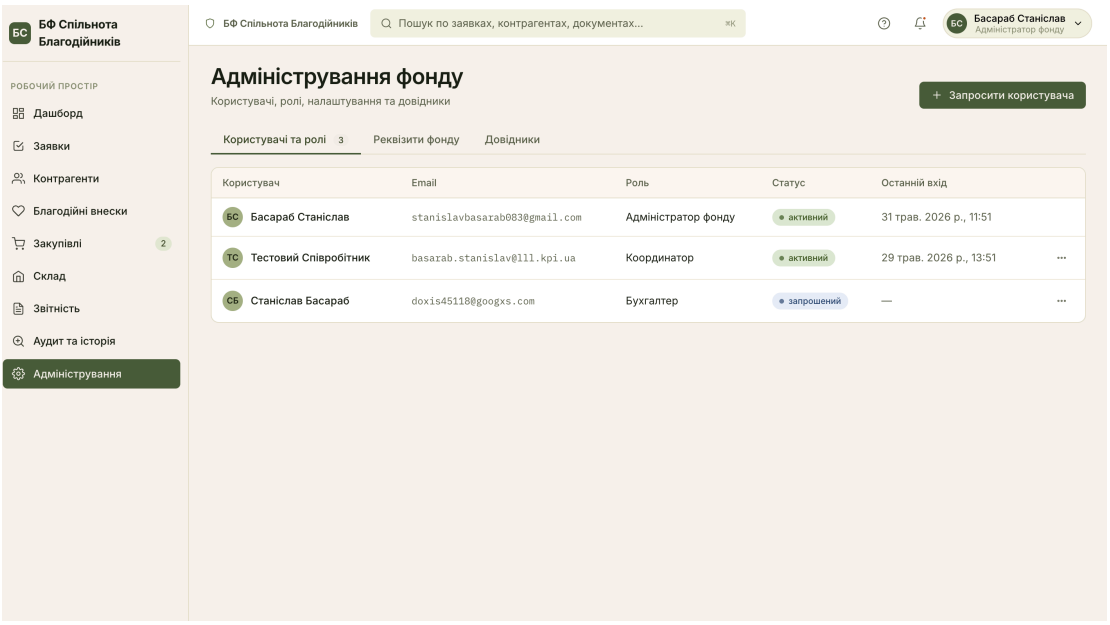


Рисунок 4.19 – Панель адміністрування

Заблокований користувач втрачає доступ до системи, проте раніше виконані ним дії залишаються в журналі аудиту. Окрім користувачів, адміністрування охоплює також редагування реквізитів фонду та довідників номенклатури.

4.9 Оцінка ефективності розробленої системи

За базу для порівняння ефективності розробленої взято наявний спосіб ведення діяльності, описаний у першому розділі, – використання розрізнених програмних засобів загального призначення (електронних таблиць, месенджерів та окремих облікових застосунків), за якого дані не пов'язані між собою й потребують повторного ручного перенесення. Під ефективністю розуміється співвідношення витрачених користувачем ресурсів до досягнутого результату, а основним кількісним показником обрано кількість елементарних дій (кроків), потрібних для виконання типової операції. Оцінку проведено за трьома вимірами, що відповідають призначенню системи: зменшенням частки ручної праці, рівнем

автоматизації типових процесів і забезпеченням прозорості діяльності перед благодійниками.

Під кроком розуміється окрема змістовна дія користувача – введення чи перенесення даних, перемикання між інструментами або ручне обчислення, – що потребує його уваги. Для оцінки взято три типові операції фонду. Реєстрація благодійного внеску з підтвердними документами за розрізненого підходу потребує близько чотирьох дій (запис у фінансову таблицю, окреме збереження документа, ручне зіставлення з картотекою благодійників та зазначення призначення), тоді як у системі вона виконується однією операцією через єдину форму, тобто скорочується на сімдесят п'ять відсотків. Формування звіту за період, що за розрізненого підходу вимагає збирання даних із кількох таблиць, ручного обчислення підсумків і компонування документа – щонайменше шість дій, – у системі зводиться до двох (налаштування параметрів звіту та його формування), тобто на дві третини менше.

Найпоказовішою є наскрізна операція забезпечення підрозділу за заявкою. За розрізненого підходу вона охоплює фіксування звернення, перенесення його до таблиці заявок, звіряння реквізитів підрозділу в окремому файлі, формування закупівлі в окремій таблиці, ручне зіставлення джерела її фінансування, оновлення складських залишків під час приймання та окремо під час видачі, ручну зміну статусу заявки й підсумкове зведення даних для звіту – щонайменше дев'ять дій у трьох-чотирьох не пов'язаних між собою інструментах із повторним введенням тих самих даних. У розробленій системі ця операція зводиться до чотирьох кроків у єдиному застосунку: реєстрації заявки, формування закупівлі з прив'язкою джерела фінансування, приймання на склад і видачі за заявкою, – причому дані вводяться одноразово й автоматично поширюються між пов'язаними записами. Скорочення кількості кроків більш ніж удвічі, поряд з усуненням

повторного введення даних, безпосередньо зменшує частку ручної праці та час на опрацювання звернення.

Зменшення кількості кроків досягається передусім за рахунок автоматизації дій, які за розрізненого підходу виконуються вручну. Статуси заявок «частково виконано» та «виконано» система виставляє самостійно за результатом видачі матеріальних цінностей; складські залишки збільшуються під час приймання й зменшуються під час видачі без окремого ручного коригування; загальна сума закупівлі обчислюється автоматично; звіти формуються конструктором на основі вже накопичених даних. У такий спосіб дії, що раніше становили окремі кроки, зникають із послідовності операцій, а ймовірність помилок через неузгоджене ручне введення усувається.

Третім виміром ефективності є прозорість діяльності перед благодійниками. Усі ключові дії фіксуються в журналі аудиту й залишаються простежуваними, а зв'язок «надходження – закупівля» дає змогу відстежити цільове використання кожного внеску. Публічна звітність формується як знімок показників за обраний період і стає доступною за відкритим посиланням без потреби входити в систему, що дозволяє благодійникам самостійно пересвідчитися в належному використанні наданих коштів. На відміну від розрізненого підходу, за якого звіти збираються вручну й не піддаються незалежній перевірці, єдина модель даних забезпечує узгодженість і достовірність оприлюднених відомостей.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі оглянуто реалізований функціонал розробленої системи управління операційною діяльністю волонтерських фондів з боку користувача. Послідовно розглянуто ключові робочі області – від автентифікації та керування доступом до формування звітності й ведення журналу аудиту.

Реалізовано вхід за обліковими даними з обов'язковим двофакторним підтвердженням, активацію облікових записів за запрошенням і відновлення пароля, а розмежування доступу за чотирма ролями забезпечує надання кожному користувачеві лише потрібного для його функцій обсягу повноважень. Зведена панель фонду дає швидкий огляд поточного стану діяльності, причому склад поданих на ній показників змінюється відповідно до ролі користувача.

Показано, що система охоплює повний операційний цикл фонду: ведення єдиної бази контрагентів, супроводження заявок військових підрозділів, облік грошових і натуральних надходжень, формування закупівель із прив'язкою джерел фінансування, приймання матеріальних цінностей на склад та їх видачу за заявками з автоматичним оновленням залишків і перерахунком статусів. Накопичені в такий спосіб дані є основою для формування внутрішньої та публічної звітності фонду.

Проведено оцінку ефективності розробленої системи відносно наявного способу ведення діяльності з використанням розрізнених засобів загального призначення. Установлено, що типова наскрізна операція забезпечення підрозділу за заявкою скорочується з дев'яти кроків до чотирьох за одноразового введення даних, типові процеси – перерахунок статусів заявок, оновлення складських залишків і формування звітності – автоматизовано, а прозорість діяльності перед благодійниками

забезпечується журналом аудиту, простежуваністю цільового використання коштів і публічною звітністю.

Наскрізне фіксування дій у журналі аудиту разом зі зв'язками між надходженнями, закупівлями та заявками забезпечують простежуваність і прозорість операцій – визначальні вимоги до діяльності волонтерських фондів. Оглянутий функціонал підтверджує, що розроблена система реалізує поставлені в технічному завданні задачі та придатна до практичного застосування.

ВИСНОВКИ

У дипломному проєкті спроєктовано та реалізовано інформаційну систему управління операційною діяльністю волонтерських фондів у військовій сфері – веб-застосунок, що забезпечує централізований облік контрагентів, супроводження запитів військових підрозділів, облік благодійних надходжень, управління закупівлями і складськими запасами та формування звітності.

У першому розділі проаналізовано предметну область і обґрунтовано актуальність роботи. Сформульовано проблему: відсутність цілісної інформаційної системи змушує фонди використовувати розрізнені програмні засоби загального призначення, що зумовлює відсутність єдиної моделі даних, інтеграційні розриви та неможливість автоматизованого формування звітності. Систематизовано теоретичну базу управління взаємовідносинами з контрагентами та проведено порівняльний аналіз чотирьох провідних систем – Salesforce, HubSpot, Bitrix24 і Zoho CRM. Встановлено, що жодна з них не задовольняє повного переліку вимог, унаслідок чого сформульовано мету проєкту – підвищення ефективності операційної діяльності волонтерських фондів військового спрямування шляхом розроблення спеціалізованого веб-застосунку, – і визначено перелік завдань для її досягнення.

У другому розділі обґрунтовано архітектурні та технологічні рішення: клієнт-серверну архітектуру у вигляді односторінкового веб-застосунку з прикладним програмним інтерфейсом на основі REST та модульний моноліт для серверної частини. Як технології серверної частини обрано Node.js із мовою TypeScript, каркас NestJS, систему керування базами даних PostgreSQL та засіб об'єктно-реляційного відображення Prisma; для клієнтської частини – бібліотеку React із супутніми засобами маршрутизації та керування станом.

У третьому розділі здійснено програмну реалізацію системи. Проєкт побудовано як монорепозиторій із поділом серверної частини на рівні контролерів, сервісів і репозиторіїв. Описано модель даних із наскрізною мультиорендністю, реалізовано двофакторну автентифікацію, розмежування доступу за ролями та ізоляцію даних між фондами, транзакційність групових операцій із журналом аудиту та автоматичне формування документації прикладного програмного інтерфейсу. Реалізовано основний операційний процес – від реєстрації заявки підрозділу до видачі матеріальних цінностей за нею, – а також підсистеми звітності, пошуку та файлових вкладень.

У четвертому розділі оглянуто реалізований функціонал з боку користувача відповідно до операційного циклу фонду та проведено оцінку ефективності розробленої системи відносно наявного підходу з використанням розрізнених засобів загального призначення. Показано, що наскрізне фіксування дій у журналі аудиту разом зі зв'язками між надходженнями, закупівлями та заявками забезпечують простежуваність і прозорість операцій – визначальні вимоги до діяльності волонтерських фондів.

Таким чином, поставлену мету – підвищення ефективності операційної діяльності волонтерських фондів військового спрямування – досягнуто, що підтверджується скороченням кількості кроків типової наскрізної операції з дев'яти до чотирьох, автоматизацією перерахунку статусів заявок, складських залишків і звітності та забезпеченням прозорості перед благодійниками. Визначені завдання виконано в повному обсязі, а розроблена система придатна до практичного застосування. Подальший розвиток вбачається у створенні мобільного застосунку та інтеграції з бухгалтерськими програмами фондів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Опендатабот [Електронний ресурс]. – Режим доступу: https://opendatabot.ua/analytics/donates-in-war-2024 (дата звернення: 25.05.2026)
2. Payne A., Frow P. A Strategic Framework for Customer Relationship Management. Journal of Marketing. 2005. Vol. 69, No. 4. P. 167–176.
3. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures: PhD dissertation. University of California, Irvine, 2000. URL: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата звернення: 25.05.2026)
4. What Are the 4 Types of CRM? Salesforce [Електронний ресурс]. – Режим доступу: https://www.salesforce.com/crm/types-of-crm/ (дата звернення: 25.05.2026)
5. The 4 Types of CRM Software (and How to Choose the Right One). Insightly [Електронний ресурс]. – Режим доступу: https://www.insightly.com/blog/blog-crm-types/ (дата звернення: 25.05.2026)
6. Peppers D., Rogers M. Managing Customer Relationships: A Strategic Framework. Hoboken: John Wiley & Sons, 2004. 528 p.
7. Buttle F. The CRM Value Chain. Marketing Business. 2001. February. P. 52–55.
8. Woodcock N., Starkey M., Stone M. et al. State of the Nation III: 2002. An Ongoing Global Study of How Organisations Manage Their Customers. London: QCi Ltd, 2002.
9. Quality Competitive Index (QCi) Model. CIO Wiki [Електронний ресурс]. – Режим доступу: https://cio-wiki.org/wiki/Quality_Competitive_Index_(QCi)_Model (дата звернення: 25.05.2026)

10. Salesforce [Електронний ресурс]. – Режим доступу: https://www.salesforce.com (дата звернення: 25.05.2026)
11. How We Refreshed the Lightning UI Design. Salesforce Blog [Електронний ресурс]. – Режим доступу: https://www.salesforce.com/blog/salesforce-cosmos-slds-2/ (дата звернення: 25.05.2026)
12. HubSpot [Електронний ресурс]. – Режим доступу: https://www.hubspot.com (дата звернення: 25.05.2026)
13. Top Product Updates for January 2026. HubSpot Community [Електронний ресурс]. – Режим доступу: https://community.hubspot.com/t5/Releases-and-Updates/Top-Product-Updates-for-January-2026/ba-p/1247546 (дата звернення: 25.05.2026)
14. Bitrix24 [Електронний ресурс]. – Режим доступу: https://www.bitrix24.com (дата звернення: 25.05.2026)
15. Bitrix24 Interface Overview. Bitrix24 Helpdesk [Електронний ресурс]. – Режим доступу: https://helpdesk.bitrix24.com/open/25746895/ (дата звернення: 25.05.2026)
16. Zoho CRM, хмарна система управління взаємовідносинами з клієнтами [Електронний ресурс]. – Режим доступу: https://www.zoho.com/crm/ (дата звернення: 25.05.2026)
17. Navigating the Zoho CRM Nextgen UI. Zoho Help [Електронний ресурс]. – Режим доступу: https://help.zoho.com/portal/en/kb/crm/using-crm-for-everyone/zoho-crm-next-gen-ui/articles/navigating-the-zoho-crm-interface#The_Main_Pane (дата звернення: 25.05.2026)
18. Hahn E. Express in Action: Writing, Building, and Testing Node.js Applications. Shelter Island, NY: Manning Publications, 2016.
19. Fastify Documentation [Електронний ресурс]. – Режим доступу: https://fastify.dev/docs/latest/ (дата звернення: 25.05.2026)

20. Mysliwiec K. NestJS Documentation [Електронний ресурс]. – Режим доступу: https://docs.nestjs.com/ (дата звернення: 25.05.2026)
21. PostgreSQL Documentation. The PostgreSQL Global Development Group [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/ (дата звернення: 25.05.2026)
22. Knex.js Documentation [Електронний ресурс]. – Режим доступу: https://knexjs.org/guide/ (дата звернення: 25.05.2026)
23. Prisma Documentation [Електронний ресурс]. – Режим доступу: https://www.prisma.io/docs (дата звернення: 25.05.2026)
24. Drizzle ORM Documentation [Електронний ресурс]. – Режим доступу: https://orm.drizzle.team/docs/overview (дата звернення: 25.05.2026)
25. React Documentation [Електронний ресурс]. – Режим доступу: https://react.dev/ (дата звернення: 25.05.2026)
26. Vue.js Documentation [Електронний ресурс]. – Режим доступу: https://vuejs.org/guide/introduction.html (дата звернення: 25.05.2026)
27. Svelte Documentation [Електронний ресурс]. – Режим доступу: https://svelte.dev/docs (дата звернення: 25.05.2026)
28. Webpack Documentation [Електронний ресурс]. – Режим доступу: https://webpack.js.org/concepts/ (дата звернення: 25.05.2026)
29. Vite Documentation [Електронний ресурс]. – Режим доступу: https://vite.dev/guide/ (дата звернення: 25.05.2026)
30. Turbopack Documentation. Next.js [Електронний ресурс]. – Режим доступу: https://nextjs.org/docs/app/api-reference/turbopack (дата звернення: 25.05.2026)
31. React Router Documentation [Електронний ресурс]. – Режим доступу: https://reactrouter.com/ (дата звернення: 25.05.2026)
32. TanStack Router Documentation [Електронний ресурс]. – Режим доступу: https://tanstack.com/router/latest (дата звернення: 25.05.2026)

ДОДАТОК А

Система управління операційною діяльністю волонтерських фондів у військовій сфері

Структура системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2026 р

					ІАЛЦ.467200.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата	**Система управління операційною діяльністю волонтерських фондів у військовій сфері**	Літ.	Маса / Масш.
Розробив		Басараб С. А.					
Перевір.		Міщенко Л. Д.				Аркуш 1	Аркушів 1
Т. контр.							
Реценз.		Олійник Ю. О.			Структура системи	КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-22	
Н. контр.		Міщенко Л. Д.					
Затвердив		Волокита А. М.					

ДОДАТОК Б

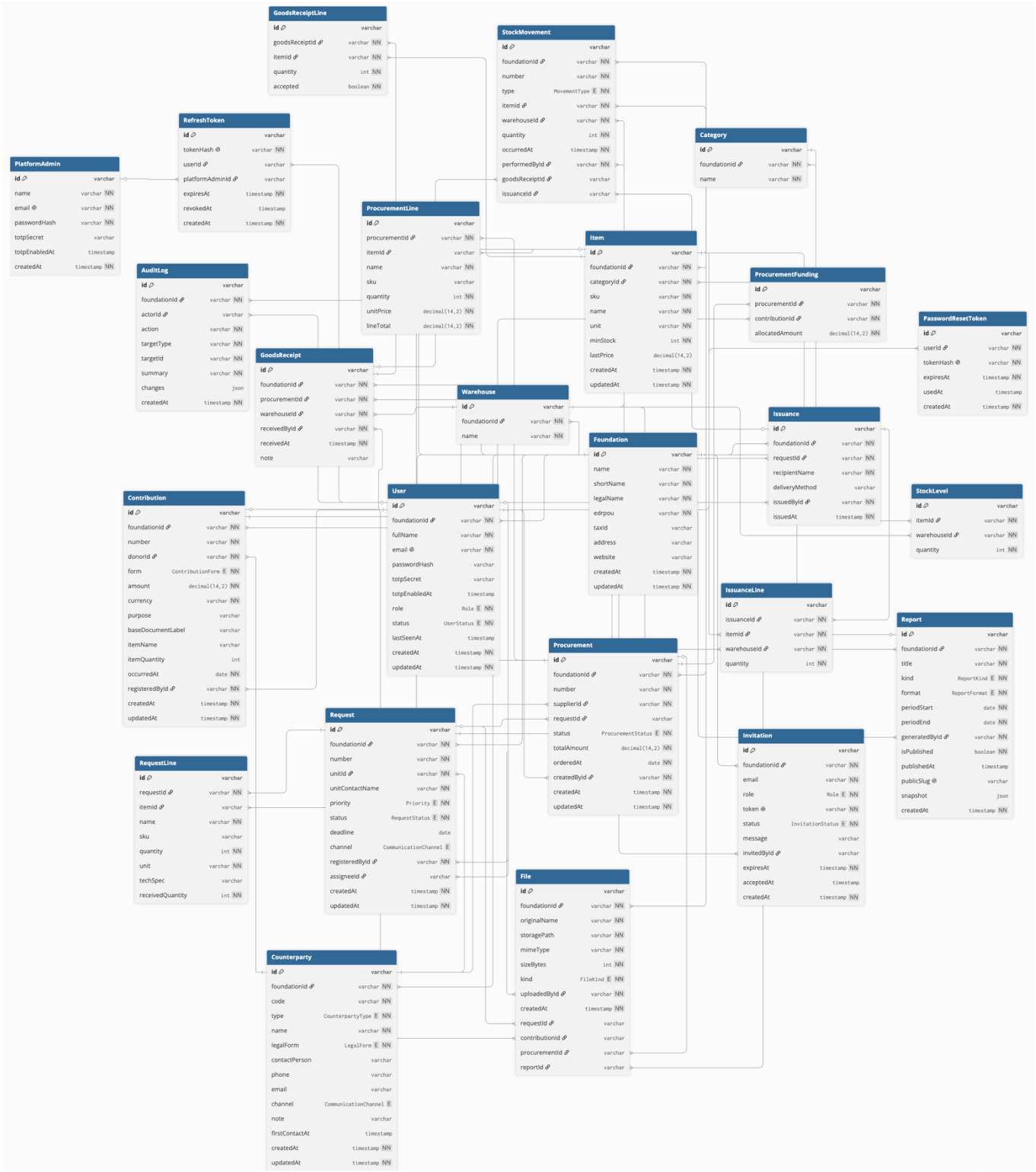
Система управління операційною діяльністю волонтерських фондів у військовій сфері

Діаграма сутність-зв'язок

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2026 р



					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата	**Система управління операційною діяльністю волонтерських фондів у військовій сфері**	Літ.	Маса / Масш.
Розробив		Басараб С. А.					
Перевір.		Міщенко Л. Д.				Аркуш 1	Аркушів 1
Т. контр.							
Реценз.		Олійник Ю. О.			Діаграма сутність-зв'язок	КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-22	
Н. контр.		Міщенко Л. Д.					
Затвердив		Волокита А. М.					

ДОДАТОК В

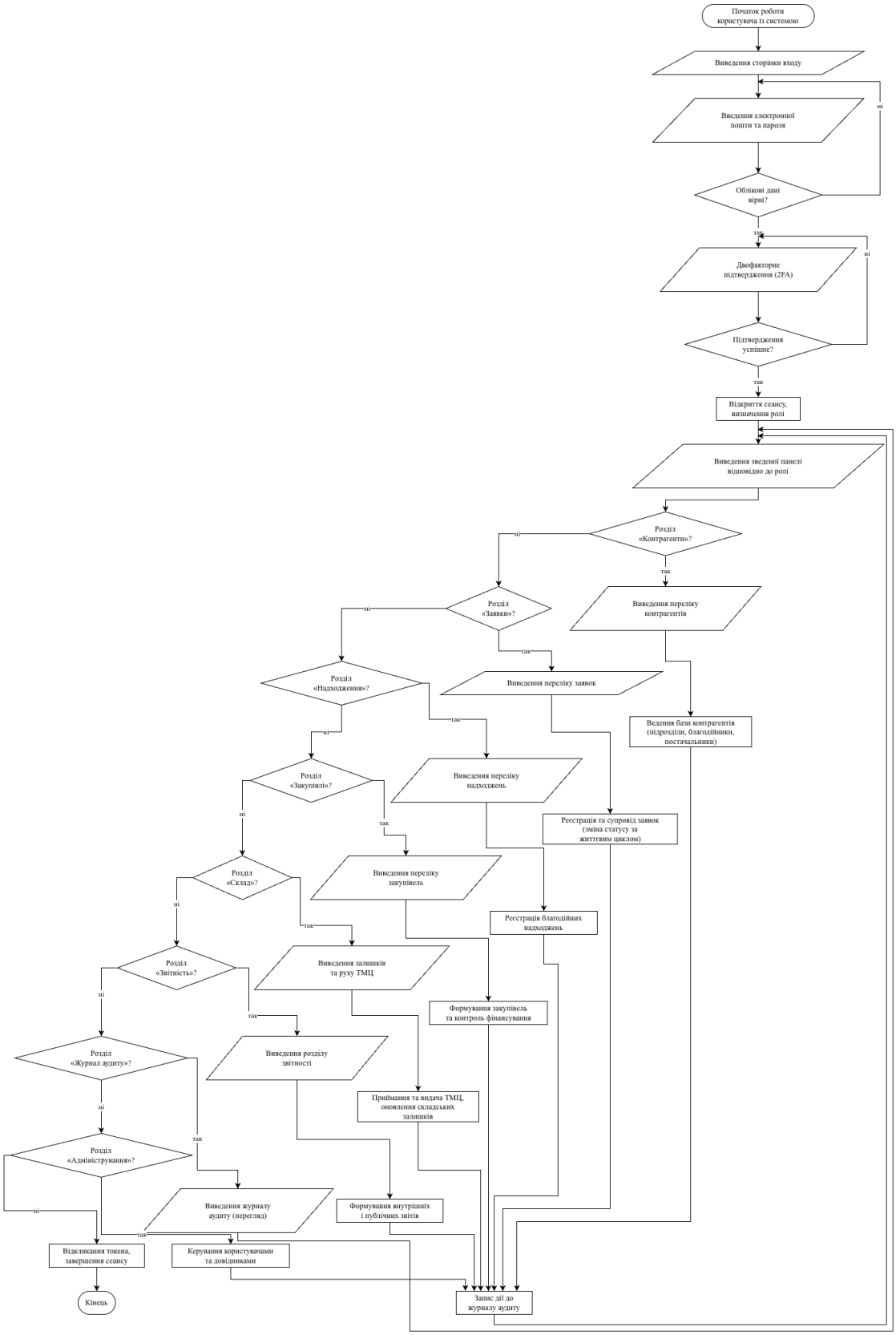
Система управління операційною діяльністю волонтерських фондів у військовій сфері

Алгоритм дій системи

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2026



					ІАЛЦ.467200.006 ДЗ	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Басараб С. А.			**Система управління операційною діяльністю волонтерських фондів у військовій сфері**	Літ. / Маса / Масш.
Перевір.		Міщенко Л. Д.				
Т. контр.						Аркуш 1 / Аркушів 1
Реценз.		Олійник Ю. О.			Алгоритм дій системи	КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-22
Н. контр.		Міщенко Л. Д.				
Затвердив		Волокита А. М.				

ДОДАТОК Г

Система управління операційною діяльністю волонтерських фондів у військовій сфері

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 11

Київ 2026 р

apps/server/src/main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import multipart from '@fastify/multipart';


async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter(),
  );


  await app.register(multipart, {
    limits: { fileSize: 26214400, files: 10 },
  });


  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
    }),
  );


  const swaggerConfig = new DocumentBuilder()
    .setTitle('Система управління операційною діяльністю фондів · API')
    .setDescription('Операційна платформа волонтерських фондів')
    .setVersion('1.0')
    .addBearerAuth()
    .addTag('Auth', 'Аутентифікація користувачів фонду')
    .addTag('Requests', 'Заявки від військових підрозділів')
    .addTag('Contributions', 'Благодійні внески: грошові та натуральні')
    .addTag('Procurements', 'Закупівлі та замовлення у постачальників')
    .addTag('Reports', 'Звітність: внутрішні та публічні звіти')
    .addTag('Audit', 'Аудит та історія: журнал усіх змін у фонді')
    .build();
  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('docs', app, document);


  await app.listen(process.env.PORT ?? 3000, '0.0.0.0');
}


void bootstrap();
```

apps/server/prisma/schema.prisma

```
generator client {
  provider     = "prisma-client"
  output       = "../src/generated/prisma"
  moduleFormat = "cjs"
}

datasource db {
  provider = "postgresql"
}

enum Role {
  ADMIN
  COORDINATOR
  ACCOUNTANT
  AUDITOR
}

enum UserStatus {
  ACTIVE
  INVITED
  BLOCKED
}

enum CounterpartyType {
  UNIT
  DONOR
  SUPPLIER
}

enum Priority {
  HIGH
  MEDIUM
  LOW
}

enum RequestStatus {
  NEW
  CONFIRMED
  IN_PROGRESS
  PARTIALLY_FULFILLED
  FULFILLED
  CLOSED
  REJECTED
}

enum ContributionForm {
  MONETARY
  IN_KIND
}

enum ProcurementStatus {
  DRAFT
  ORDERED
  PAID
  RECEIVED
  CLOSED
}

enum ReportFormat {
  XLSX
  PDF
  CSV
```

```
  HTML
}

model Foundation {
  id         String   @id @default(cuid())
  name       String
  shortName  String
  legalName  String
  edrpou     String
  taxId      String?
  address    String?
  website    String?
  createdAt  DateTime @default(now())
  updatedAt  DateTime @updatedAt

  users          User[]
  invitations    Invitation[]
  counterparties Counterparty[]
  categories     Category[]
  items          Item[]
  warehouses     Warehouse[]
  requests       Request[]
  contributions  Contribution[]
  procurements   Procurement[]
  goodsReceipts  GoodsReceipt[]
  issuances      Issuance[]
  movements      StockMovement[]
  files          File[]
  reports        Report[]
  auditLogs      AuditLog[]
}
```

apps/server/src/infrastructure/database/prisma.service.ts

```
import {
  Injectable,
  Logger,
  OnModuleDestroy,
  OnModuleInit,
} from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PrismaPg } from '@prisma/adapter-pg';
import { PrismaClient } from '../../generated/prisma/client';

@Injectable()
export class PrismaService
  extends PrismaClient
  implements OnModuleInit, OnModuleDestroy
{
  private readonly logger = new Logger(PrismaService.name);

  constructor(config: ConfigService) {
    super({
      adapter: new PrismaPg({
        connectionString: config.getOrThrow<string>('DATABASE_URL'),
      }),
    });
  }

  async onModuleInit(): Promise<void> {
    await this.$connect();
    this.logger.log('Connected to the database');
  }

  async onModuleDestroy(): Promise<void> {
```

```
    await this.$disconnect();
  }
}
```

apps/server/src/modules/auth/strategies/jwt.strategy.ts

```
import { Inject, Injectable } from '@nestjs/common';
import type { ConfigType } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { jwtConfig } from '../../../config/configuration';
import type { AuthenticatedPrincipal } from '../../../common/data/authenticated-principal';
import type { JwtPayload } from '../data/jwt-payload';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(@Inject(jwtConfig.KEY) config: ConfigType<typeof jwtConfig>) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: config.secret,
    });
  }

  validate(payload: JwtPayload): AuthenticatedPrincipal {
    if (payload.type === 'PLATFORM') {
      return { type: 'PLATFORM', sub: payload.sub };
    }
    return {
      type: 'USER',
      sub: payload.sub,
      foundationId: payload.foundationId as string,
      role: payload.role as NonNullable<JwtPayload['role']>,
    };
  }
}
```

apps/server/src/common/guards/roles.guard.ts

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import type { Role } from '../../generated/prisma/enums';
import type { AuthenticatedPrincipal } from '../data/authenticated-principal';
import { ROLES_KEY } from '../decorators/roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const required = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!required || required.length === 0) {
      return true;
    }
    const { user } = context
      .switchToHttp()
      .getRequest<{ user?: AuthenticatedPrincipal }>();
    return !!user && user.type === 'USER' && required.includes(user.role);
  }
}
```

apps/server/src/common/decorators/current-foundation-user.decorator.ts

```
import {
```

```
  createParamDecorator,
  ExecutionContext,
  ForbiddenException,
} from '@nestjs/common';
import type {
  AuthenticatedPrincipal,
  UserPrincipal,
} from '../data/authenticated-principal';

export const CurrentFoundationUser = createParamDecorator(
  (_data: unknown, ctx: ExecutionContext): UserPrincipal => {
    const { user } = ctx
      .switchToHttp()
      .getRequest<{ user?: AuthenticatedPrincipal }>();
    if (!user || user.type !== 'USER') {
      throw new ForbiddenException('Доступно лише користувачам фонду');
    }
    return user;
  },
);
```

apps/server/src/modules/auth/token.service.ts

```
import { Inject, Injectable, UnauthorizedException } from '@nestjs/common';
import type { ConfigType } from '@nestjs/config';
import { JwtService } from '@nestjs/jwt';
import type { JwtSignOptions } from '@nestjs/jwt';
import { jwtConfig } from '../../config/configuration';
import { generateToken, hashToken } from '../../common/crypto';
import type { Role } from '../../generated/prisma/enums';
import { RefreshTokenRepository } from
'../../infrastructure/database/repos/refresh-token.repo';
import type { IssuedTokens, RefreshOwner } from './data/issued-tokens';
import type { JwtPayload } from './data/jwt-payload';

const MS_PER_DAY = 24 * 60 * 60 * 1000;

@Injectable()
export class TokenService {
  constructor(
    private readonly jwt: JwtService,
    private readonly refreshTokens: RefreshTokenRepository,
    @Inject(jwtConfig.KEY)
    private readonly config: ConfigType<typeof jwtConfig>,
  ) {}

  async issueForUser(user: {
    id: string;
    foundationId: string;
    role: Role;
  }): Promise<IssuedTokens> {
    const accessToken = await this.signAccess({
      sub: user.id,
      type: 'USER',
      foundationId: user.foundationId,
      role: user.role,
    });
    const refreshToken = await this.createRefresh({ userId: user.id });
    return { accessToken, refreshToken };
  }

  async consume(rawRefresh: string): Promise<RefreshOwner> {
    const record = await
this.refreshTokens.findByHash(hashToken(rawRefresh));
    if (!record || record.revokedAt || record.expiresAt < new Date()) {
      throw new UnauthorizedException('Недійсний токен оновлення');
    }
```

```
          await this.refreshTokens.revoke(record.id);
          return      {      userId:      record.userId,      platformAdminId:
record.platformAdminId };
        }

        async revoke(rawRefresh: string): Promise<void> {
          const                  record                  =                  await
this.refreshTokens.findByHash(hashToken(rawRefresh));
          if (record && !record.revokedAt) {
            await this.refreshTokens.revoke(record.id);
          }
        }

        private signAccess(payload: JwtPayload): Promise<string> {
          const options: JwtSignOptions = {
            expiresIn: this.config.accessTtl as JwtSignOptions['expiresIn'],
          };
          return this.jwt.signAsync(payload, options);
        }

        private async createRefresh(
          owner: { userId: string } | { platformAdminId: string },
        ): Promise<string> {
          const raw = generateToken();
          const expiresAt = new Date(
            Date.now() + this.config.refreshTtlDays * MS_PER_DAY,
          );
          await this.refreshTokens.create({
            tokenHash: hashToken(raw),
            expiresAt,
            ...owner,
          });
          return raw;
        }
      }
```

apps/server/src/modules/requests/requests.controller.ts

```
      import {
        Body, Controller, Delete, Get, HttpCode, HttpStatus,
        Param, Patch, Post, UseGuards,
      } from '@nestjs/common';
      import {
        ApiBearerAuth, ApiCreatedResponse, ApiOkResponse, ApiOperation, ApiTags,
      } from '@nestjs/swagger';
      import { RolesGuard } from '../../common/guards/roles.guard';
      import { Roles } from '../../common/decorators/roles.decorator';
      import { CurrentFoundationUser } from '../../common/decorators/current-
foundation-user.decorator';
      import type { UserPrincipal } from '../../common/data/authenticated-
principal';
      import { Role } from '../../generated/prisma/enums';
      import { RequestsService } from './requests.service';
      import { CreateRequestDto } from './body/create-request.dto';
      import { UpdateRequestDto } from './body/update-request.dto';
      import { RequestResponse } from './responses/request.response';
      import   {   RequestDetailResponse   }   from   './responses/request-
detail.response';

      @ApiTags('Requests')
      @ApiBearerAuth()
      @Controller('requests')
      export class RequestsController {
        constructor(private readonly requests: RequestsService) {}

        @Get()
        @ApiOperation({ summary: 'Перелік заявок фонду' })
```

```
  @ApiOkResponse({ type: RequestResponse, isArray: true })
  list(
    @CurrentFoundationUser() actor: UserPrincipal,
  ): Promise<RequestResponse[]> {
    return this.requests.list(actor);
  }

  @Get(':id')
  @ApiOperation({ summary: 'Картка заявки' })
  @ApiOkResponse({ type: RequestDetailResponse })
  get(
    @CurrentFoundationUser() actor: UserPrincipal,
    @Param('id') id: string,
  ): Promise<RequestDetailResponse> {
    return this.requests.get(actor, id);
  }

  @Post()
  @UseGuards(RolesGuard)
  @Roles(Role.ADMIN, Role.COORDINATOR)
  @ApiOperation({ summary: 'Створити заявку (адмін, координатор)' })
  @ApiCreatedResponse({ type: RequestDetailResponse })
  create(
    @CurrentFoundationUser() actor: UserPrincipal,
    @Body() dto: CreateRequestDto,
  ): Promise<RequestDetailResponse> {
    return this.requests.create(actor, dto);
  }

  @Patch(':id')
  @UseGuards(RolesGuard)
  @Roles(Role.ADMIN, Role.COORDINATOR)
  @ApiOperation({ summary: 'Оновити заявку (адмін, координатор)' })
  @ApiOkResponse({ type: RequestDetailResponse })
  update(
    @CurrentFoundationUser() actor: UserPrincipal,
    @Param('id') id: string,
    @Body() dto: UpdateRequestDto,
  ): Promise<RequestDetailResponse> {
    return this.requests.update(actor, id, dto);
  }

}
```

apps/server/src/modules/requests/requests.service.ts

```
@Injectable()
export class RequestsService {
  constructor(
    private readonly requests: RequestRepository,
    private readonly items: ItemRepository,
    private readonly counterparties: CounterpartyRepository,
    private readonly users: UserRepository,
    private readonly audits: AuditLogRepository,
    private readonly filesService: FilesService,
  ) {}

  async list(actor: UserPrincipal): Promise<RequestResponse[]> {
    const records = await this.requests.findManyByFoundation(
      actor.foundationId,
    );
    return records.map((record) => this.toListResponse(record));
  }

  async get(actor: UserPrincipal, id: string):
Promise<RequestDetailResponse> {
```

```
    const record = await this.requests.findByIdFull(id);
    if (!record || record.foundationId !== actor.foundationId) {
      throw new NotFoundException('Заявку не знайдено');
    }
    const history = await this.audits.findByTarget(
      actor.foundationId, 'REQUEST', id,
    );
    return this.toDetailResponse(record, history);
  }

  private async ensureOwned(actor: UserPrincipal, id: string) {
    const request = await this.requests.findById(id);
    if (!request || request.foundationId !== actor.foundationId) {
      throw new NotFoundException('Заявку не знайдено');
    }
    return request;
  }

  private async assertUnit(
    actor: UserPrincipal,
    unitId: string,
  ): Promise<void> {
    const unit = await this.counterparties.findById(unitId);
    if (
      !unit ||
      unit.foundationId !== actor.foundationId ||
      unit.type !== CounterpartyType.UNIT
    ) {
      throw new BadRequestException('Невідомий підрозділ');
    }
  }

  private audit(
    actor: UserPrincipal,
    requestId: string,
    action: string,
    summary: string,
  ) {
    return this.audits.create({
      foundationId: actor.foundationId,
      actorId: actor.sub,
      action,
      targetType: 'REQUEST',
      targetId: requestId,
      summary,
    });
  }
}
```

apps/server/src/infrastructure/database/repos/request.repo.ts

```
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma.service';
import type {
  CommunicationChannel, Priority, RequestStatus,
} from '../../../generated/prisma/enums';

export interface CreateRequestLineInput {
  itemId: string | null;
  name: string;
  sku: string | null;
  quantity: number;
  unit: string;
  techSpec: string | null;
}

export interface CreateRequestInput {
```

```
  foundationId: string;
  number: string;
  unitId: string;
  unitContactName: string;
  priority: Priority;
  status: RequestStatus;
  deadline: Date | null;
  channel: CommunicationChannel | null;
  registeredById: string;
  assigneeId: string | null;
  lines: CreateRequestLineInput[];
}

@Injectable()
export class RequestRepository {
  constructor(private readonly prisma: PrismaService) {}

  findManyByFoundation(foundationId: string) {
    return this.prisma.request.findMany({
      where: { foundationId },
      orderBy: { createdAt: 'desc' },
      include: {
        unit: { select: { name: true } },
        lines: {
          select: {
            name: true,
            quantity: true,
            item: { select: { lastPrice: true } },
          },
        },
      },
    });
  }

  create(input: CreateRequestInput) {
    const { lines, ...rest } = input;
    return this.prisma.request.create({
      data: { ...rest, lines: { create: lines } },
    });
  }

}
```

apps/web/src/lib/api/client.ts

```
export const BASE_URL = import.meta.env.VITE_API_URL ?? 'http://localhost:3000'

export class ApiError extends Error {
  readonly status: number
  constructor(status: number, message: string) {
    super(message)
    this.name = 'ApiError'
    this.status = status
  }
}

interface RequestOptions {
  method?: 'GET' | 'POST' | 'PATCH' | 'DELETE'
  body?: unknown
  token?: string | null
}

export interface AuthHooks {
  getRefreshToken: () => string | null
  onTokens: (accessToken: string, refreshToken: string) => void
```

```
  onAuthFailure: () => void
}

let authHooks: AuthHooks | null = null

export function registerAuthHooks(hooks: AuthHooks): void {
  authHooks = hooks
}

function send(path: string, options: RequestOptions, token: string | null
| undefined) {
  const hasBody = options.body !== undefined
  return fetch(`${BASE_URL}${path}`, {
    method: options.method ?? 'GET',
    headers: {
      ...(hasBody ? { 'Content-Type': 'application/json' } : {}),
      ...(token ? { Authorization: `Bearer ${token}` } : {}),
    },
    body: hasBody ? JSON.stringify(options.body) : undefined,
  })
}

async function callRefresh(
  refreshToken: string,
): Promise<{ accessToken: string; refreshToken: string } | null> {
  try {
    const response = await fetch(`${BASE_URL}/auth/refresh`, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ refreshToken }),
    })
    if (!response.ok) return null
    return (await response.json()) as { accessToken: string; refreshToken:
string }
  } catch {
    return null
  }
}

let refreshing: Promise<string | null> | null = null

function refreshAccessToken(): Promise<string | null> {
  if (refreshing) return refreshing
  const hooks = authHooks
  const refreshToken = hooks?.getRefreshToken() ?? null
  if (!hooks || !refreshToken) return Promise.resolve(null)
  refreshing = callRefresh(refreshToken)
    .then((tokens) => {
      if (!tokens) return null
      hooks.onTokens(tokens.accessToken, tokens.refreshToken)
      return tokens.accessToken
    })
    .finally(() => {
      refreshing = null
    })
  return refreshing
}

export async function apiFetch<T>(path: string, options: RequestOptions =
{}): Promise<T> {
  let response = await send(path, options, options.token)

  if (
    response.status === 401 &&
```

```
      options.token &&
      path !== '/auth/refresh' &&
      authHooks?.getRefreshToken()
    ) {
      const newToken = await refreshAccessToken()
      if (newToken) {
        response = await send(path, options, newToken)
      } else {
        authHooks.onAuthFailure()
      }
    }

    if (!response.ok) {
      throw new ApiError(response.status, await extractMessage(response))
    }
    if (response.status === 204) {
      return undefined as T
    }
    return (await response.json()) as T
  }

  async function extractMessage(response: Response): Promise<string> {
    try {
      const data: unknown = await response.json()
      if (data && typeof data === 'object' && 'message' in data) {
        const message = (data as { message: string | string[] }).message
        return Array.isArray(message) ? message.join(', ') : message
      }
    } catch {
      // вертаємось до тексту статусу
    }
    return response.statusText || 'Помилка запиту'
  }
```