МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВЕБДОДАТКУ ДЛЯ ОБЛІКУ ФІНАНСОВИХ ОПЕРАЦІЙ»**

Виконав: студент 4 курсу, групи 6.1212-пі-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
В.М. Плахотний
(ініціали та прізвище)

Керівник завідувач кафедри програмної інженерії, доцент, к.ф.-м.н. Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Плахотному Вадиму Максимовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебдодатку для обліку фінансових операцій

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1741-с

2. Строк подання студентом роботи 01.06.2026 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі, аналіз предметної області.
2. Проєктування.
3. Реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 09.12.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	12.01.2026	
2.	Збір вихідних даних.	19.01.2026	
3.	Обробка методичних та теоретичних	02.02.2026	
	джерел.		
4.	Розробка першого та другого розділу.	16.03.2026	
5.	Розробка третього розділу.	18.05.2026	
6.	Оформлення та нормоконтроль	25.05.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	16.06.2026	

Студент _____ В.М. Плахотний
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка вебдодатку для обліку фінансових операцій»: 100 с., 43 рис., 1 табл., 19 джерел, 5 додатків.

ANGULAR, EXPRESS, NODE.JS, POSTGRESQL, PDF, PRISMA ORM, REST API, JWT, TAILWIND CSS, PUPPETEER, UML.

Об’єкт дослідження – вебдодатки персонального фінансового обліку.

Мета роботи – розробити вебдодаток для обліку фінансових операцій.

Методи дослідження – аналітичний, моделювання, проєктування, програмний.

У роботі розглянуто проєктування та реалізацію вебдодатку для персонального обліку фінансових операцій. Клієнтська частина виконана засобами Angular. Серверна частина реалізована на Node.js та Express.js і забезпечує REST API для роботи з користувачами, рахунками, категоріями, фінансовими операціями, звітами та аналітикою. Для збереження даних використано PostgreSQL, а взаємодія з базою даних виконується через Prisma ORM.

У системі реалізовано реєстрацію користувача з підтвердженням email, вхід до облікового запису, відновлення пароля, створення рахунків, категорій і фінансових операцій. Передбачено автоматичне оновлення балансу рахунків після створення доходів, витрат і переказів. Також реалізовано дашборд, фінансову звітність, експорт звітів у PDF та аналітику фінансових звичок користувача.

Побудовано ER-модель бази даних та UML-діаграми: варіантів використання, діяльності, класів, послідовності та розгортання.

Таким чином, розроблений вебдодаток забезпечує зручний інтерфейс для ведення персонального фінансового обліку, дозволяє користувачу контролювати доходи й витрати, формувати звіти та отримувати короткі аналітичні висновки щодо фінансової поведінки.

SUMMARY

Bachelor’s qualifying paper “Development of a Web Application for Financial Transaction Management”: 100 pages, 43 figures, 1 table, 19 references, 5 supplements.

ANGULAR, EXPRESS, NODE.JS, POSTGRESQL, PDF, PRISMA ORM, REST API, JWT, TAILWIND CSS, PUPPETEER, UML.

The object of the study is web applications for personal financial accounting.

The aim of the study is to to develop a web application for financial operations accounting.

The methods of research are analytical, modeling, design, and programmin.

The paper considers the design and implementation of a web application for personal financial operations accounting. The client side is implemented using Angular. The server side is implemented using Node.js and Express.js and provides a REST API for working with users, accounts, categories, financial operations, reports, and analytics. PostgreSQL is used for data storage, while interaction with the database is performed through Prisma ORM.

The system implements user registration with email confirmation, account login, password recovery, and the creation of accounts, categories, and financial operations. Automatic updating of account balances after creating income, expenses, and transfers is provided. The system also includes a dashboard, financial reporting, PDF report export, and analytics of the user’s financial habits.

An ER model of the database and UML diagrams were developed: use case diagram, activity diagram, class diagram, sequence diagram, and deployment diagram.

Thus, the developed web application provides a convenient interface for personal financial accounting, allows the user to control income and expenses, generate reports, and receive brief analytical conclusions about financial behavior.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Технічне завдання 10
1.1 Перелік умовних позначень, скорочень, термінів 10
1.2 Функціональні вимоги 11
1.2.1 Призначення і цілі створення системи 11
1.2.2 Загальні функціональні можливості системи 12
1.3 Нефункціональні вимоги 12
1.4 Опис предметної області 14
1.5 Опис системи 16
1.6 Огляд та порівняння аналогів 17
2 Проєктування 20
2.1 ER-модель бази даних та опис сутностей системи 20
2.2 Діаграма варіантів використання 22
2.2.1 Опис варіантів використання 24
2.3 Діаграма діяльності основного бізнес-процесу 34
2.4 Діаграма класів основного бізнес-процесу 36
2.5 Діаграма послідовності основного бізнес-процесу 38
2.6 Діаграма розгортання 39
3 Програмна реалізація та тестування 41
3.1 Опис інструментів розробки 41
3.2 Основні класи, інтерфейси та функції системи 42
3.3 Реалізація клієнтської частини 45
3.4 Реалізація серверної частини 47
3.5 Тестування проєкту 51

3.6 Керівництво користувача 53
3.6.1 Вхід та реєстрація 53
3.6.2 Робота з рахунками 57
3.6.3 Робота з категоріями 59
3.6.4 Робота з фінансовими операціями 60
3.6.5 Перегляд дашборду 63
3.6.6 Формування звітів та PDF-експорт 63
3.6.7 Перегляд аналітики 65
Висновки 67
Перелік посилань 68
Додаток А Angular TransactionsComponent 70
Додаток Б Angular TransactionService 83
Додаток В Express transaction.routes.js 86
Додаток Г Express transaction.controller.js 87
Додаток Д Express transaction.service.js 89

ВСТУП

У сучасних умовах зростає потреба у простих і доступних інструментах для контролю особистих фінансів. Користувачі щоденно здійснюють різні фінансові операції: отримують доходи, оплачують товари й послуги, переказують кошти між рахунками. Без систематичного обліку складно зрозуміти, на що саме витрачаються кошти, які категорії займають найбільшу частку витрат і як змінюється фінансова поведінка протягом певного періоду.

На ринку існує багато застосунків для персонального фінансового обліку, однак частина з них має надмірний функціонал, орієнтована переважно на мобільні пристрої або є комерційною. Крім того, готові рішення не завжди дозволяють простежити внутрішню логіку роботи системи, структуру бази даних, взаємодію клієнтської та серверної частин, а також механізми формування звітності й аналітики.

Актуальність роботи полягає у створенні вебдодатку, який забезпечує повний цикл персонального фінансового обліку: від реєстрації користувача та створення рахунків до внесення фінансових операцій, перегляду статистики, формування звітів і аналізу фінансових звичок. Така система має бути зручною, зрозумілою для користувача та побудованою на сучасних вебтехнологіях.

З огляду на це, сформульовано мету та задачі дослідження.

Мета: розробити вебдодаток для обліку фінансових операцій.

Задачі:

- сформулювати функціональні та нефункціональні вимоги до системи;
- проаналізувати предметну область персонального фінансового обліку;
- розглянути та порівняти аналогічні програмні рішення;
- спроєктувати структуру бази даних і побудувати ER-модель;
- побудувати UML-діаграми для основних процесів системи;
- реалізувати вебдодаток для обліку фінансових операцій;
- виконати тестування основних сценаріїв роботи системи;
- підготувати керівництво користувача.

Об’єкт дослідження: вебдодатки персонального фінансового обліку.

Предмет дослідження: засоби та методи розробки клієнт-серверного вебдодатку для обліку доходів, витрат, рахунків, категорій, звітності та аналітики фінансових операцій.

Методи дослідження: аналітичний, моделювання, проєктування, програмний.

У першому розділі описано функціональні та нефункціональні вимоги до системи, розглянуто предметну область персонального фінансового обліку, подано опис системи та огляд аналогів із порівняльною таблицею.

У другому розділі виконано проєктування вебдодатку: побудовано ER-модель бази даних, діаграму варіантів використання, діаграму діяльності, діаграми класів клієнтської та серверної частин, діаграму послідовності та діаграму розгортання.

У третьому розділі описано інструменти розробки, основні класи системи для прецеденту створення фінансової операції, реалізацію клієнтської та серверної частин, бази даних, тестування системи та керівництво користувача.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Перелік умовних позначень, скорочень, термінів

Витрата – фінансова операція, яка зменшує баланс обраного рахунку користувача.

Дохід – фінансова операція, яка збільшує баланс обраного рахунку користувача.

Категорія фінансової операції – ознака, за допомогою якої користувач групує доходи та витрати.

Облік витрат – процес фіксації та групування витрат користувача з метою подальшого аналізу фінансової поведінки [3].

Переказ – фінансова операція, що відображає переміщення коштів між двома рахунками одного користувача.

Рахунок – сутність системи, яка використовується для обліку коштів користувача.

Система – вебдодаток обліку фінансових операцій.

Фінансова грамотність – сукупність знань, навичок, ставлень і поведінкових моделей, які дають змогу людині приймати обґрунтовані фінансові рішення [13].

Фінансова операція – запис у системі, що відображає рух коштів користувача.

Angular – фреймворк і платформа для створення клієнтських вебзастосунків на базі TypeScript [1].

Express.js – мінімалістичний і гнучкий вебфреймворк для Node.js, який використовується для створення серверних маршрутів, обробки HTTP-запитів і підключення middleware [4].

Node.js – середовище виконання JavaScript на сервері, яке дає змогу створювати серверні застосунки та REST API [11].

PostgreSQL – об’єктно-реляційна система керування базами даних [14].

Prisma ORM – інструмент для роботи з базою даних, який дозволяє описувати моделі даних у Prisma Schema, виконувати міграції та взаємодіяти з PostgreSQL через Prisma Client [15].

REST API – архітектурний підхід до організації взаємодії між клієнтською та серверною частинами через HTTP-запити [4].

JWT – JSON Web Token, відкритий стандарт RFC 7519 для компактного й URL-безпечного передавання claims між двома сторонами [12].

1.2 Функціональні вимоги

Функціональні вимоги визначають перелік основних можливостей, які має надавати вебдодаток користувачу [7, 18].

Для системи обліку фінансових операцій ці вимоги пов’язані з реєстрацією користувача, керуванням рахунками, категоріями, фінансовими операціями, переглядом узагальнених даних, формуванням звітів та аналізом фінансової поведінки.

1.2.1 Призначення і цілі створення системи

Функціональне призначення системи – надати користувачу зручний інструмент для обліку особистих фінансових операцій, контролю доходів і витрат, аналізу фінансового стану та формування звітності.

Експлуатаційне призначення – система використовується користувачами для виконання та аналізу особистих фінансових операцій.

Мета створення системи – розробити вебдодаток, який дозволить користувачу швидко й зрозуміло вести облік фінансових операцій.

1.2.2 Загальні функціональні можливості системи

У системі передбачено два основні ролі: гість і авторизований користувач. Гість має доступ лише до сторінок входу, реєстрації та відновлення пароля. Авторизований користувач отримує доступ до основних функціональних модулів системи.

Система має надавати гостям наступні можливості:

– створити обліковий запис;
– підтвердити обліковий запис;
– увійти до облікового запису;
– відновити пароль.

Авторизованому користувачу система має надавати наступні можливості:

– створити/редагувати/видалити/переглянути список рахунок/рахунків;
– створити/редагувати/видалити/переглянути список категорію фінансової операції/категорій фінансових операцій;
– створити/редагувати/видалити/переглянути список фінансову операцію/фінансових операцій;
– переглянути/експортувати до PDF звіт фінансових операцій;
– переглянути аналітику фінансових операцій;
– фільтрувати рахунки/категорії/фінансові операції;
– виконувати пошук рахунків/категорій/фінансових операцій;
– переглянути дані облікового запису;
– вийти з облікового запису.

1.3 Нефункціональні вимоги

Нефункціональні вимоги визначають не окремі дії користувача, а загальні характеристики системи: зручність використання, безпеку, продуктивність, надійність, доступність, сумісність із браузерами та можливість подальшої

підтримки [7, 18]. Для вебдодатку обліку фінансових операцій ці вимоги мають критичне значення, оскільки система працює з персональними фінансовими даними користувача.

Вимоги до інтерфейсу користувача. Інтерфейс системи має бути простим, інтуїтивно зрозумілим і не вимагати від користувача спеціальних бухгалтерських або технічних знань.

Вимоги до адаптивності. Вебдодаток має коректно відображатися та бути зручним у використанні на різних типах пристроїв.

Вимоги до доступності. Інтерфейс системи має бути доступним і зрозумілим для широкого кола користувачів. Поля форм повинні мати підписи, повідомлення про помилки мають бути чіткими, а кнопки зрозумілими за призначенням.

Вимоги до продуктивності. Система повинна швидко реагувати на дії користувача. Завантаження основних сторінок після авторизації не повинно створювати відчутної затримки. Отримання списку рахунків, категорій і фінансових операцій має виконуватися через API-запити з використанням фільтрації та пагінації. Серверна частина повинна оптимізовувати обробку запитів, розділення логіки на маршрути, контролери та сервіси, а також виконувати операції з базою даних лише в необхідному обсязі.

Вимоги до безпеки. Доступ до основних модулів має бути обмежений лише для авторизованих осіб. Авторизація реалізується з використанням JWT-токена для підтвердження особи користувача під час звернення до захищених API-маршрутів. Сервер має перевіряти не лише наявність JWT-токена, а й належність запитуваних даних поточному користувачу. Паролі користувачів не повинні зберігатися у відкритому вигляді. Для їх збереження система повинна використовувати криптографічне хешування.

Також система повинна валідувати вхідні дані не тільки на клієнтській, а і на серверній частині для захисту API від некоректних або небезпечних запитів.

Вимоги до цілісності даних. Дані системи мають залишатися узгодженими після створення, редагування або видалення фінансових операцій.

Вимоги до надійності. Система повинна коректно обробляти типові помилки: та відображати зрозуміле повідомлення про причину помилки.

У разі недійсного JWT-токена клієнтська частина повинна автоматично очищувати локальну сесію та перенаправляти користувача на сторінку входу.

Вимоги до сумісності. Система повинна коректно працювати в актуальних версіях основних браузерів: Google Chrome, Mozilla Firefox, Microsoft Edge, Opera та Safari.

Вимоги до масштабованості. Система має передбачати можливість подальшого розширення без повної перебудови системи.

Вимоги до збереження конфіденційних даних. Конфіденційні налаштування повинні зберігатися у файлі .env і не потрапляти до системи контролю версій.

Вимоги до експорту. Назва PDF-файлу під час експорту звітності повинна формуватися латиницею, щоб уникнути проблем із кодуванням під час завантаження файлу браузером.

Вимоги до аналітики. Аналітика фінансових звичок повинна формуватися на основі конкретних та прозорих правил, а не прихованих алгоритмів.

1.4 Опис предметної області

Предметною областю розроблюваної системи є персональний облік фінансових операцій користувача. До неї належать процеси фіксації доходів, витрат, переказів між рахунками, групування операцій за категоріями, перегляд поточного фінансового стану, формування звітів та аналіз фінансових звичок.

Персональний фінансовий облік є важливою складовою фінансової грамотності. У матеріалах OECD фінансова грамотність розглядається як поєднання знань, навичок, ставлень і поведінки, які дають змогу людині приймати обґрунтовані фінансові рішення та підтримувати фінансовий добробут [13]. Тому система обліку фінансових операцій має не лише зберігати

записи про доходи й витрати, а й допомагати користувачу краще розуміти власну фінансову поведінку.

Основною сутністю предметної області є фінансова операція. У межах системи вона може бути трьох типів: дохід, витрата або переказ. Дохід збільшує баланс рахунку, витрата зменшує баланс рахунку, а переказ змінює баланс двох рахунків одночасно, не змінюючи загальну суму коштів користувача.

Важливою складовою системи є рахунок. Він відображає місце зберігання коштів користувача: готівку, банківську картку, банківський рахунок, накопичувальний рахунок або інший фінансовий ресурс. Кожен рахунок має назву, тип, валюту, початковий і поточний баланс.

Для аналізу фінансових операцій використовується категоризація. Категорії дозволяють групувати доходи й витрати за змістом. Саме категорії дають змогу надалі визначати основні напрями витрат і формувати звітність.

Consumer Financial Protection Bureau зазначає, що відстеження витрат допомагає користувачу краще зрозуміти власні витратні звички та приймати рішення відповідно до фінансових цілей [3]. Отже, система повинна не лише фіксувати операції, а й надавати користувачу узагальнену інформацію про його фінансову поведінку. Звітність дозволяє користувачу переглянути фінансові показники за певний період: суму доходів, витрат, переказів, чистий результат, групування за категоріями та рахунками, а також динаміку доходів і витрат.

Крім звітності, система передбачає аналітику фінансових звичок. Вона дозволяє визначити найбільшу категорію витрат, найбільшу разову витрату, середню щоденну витрату, частку витрат від доходів і зміну витрат порівняно з попереднім періодом. На основі цих показників система формує короткі висновки та рекомендації.

Основними проблемами предметної області, які має вирішити система, є відсутність єдиного місця для обліку доходів і витрат, складність аналізу структури витрат, необхідність ручного підрахунку фінансових показників і потреба у зручному формуванні звітів.

Таким чином, предметна область системи охоплює повний цикл персонального фінансового обліку: від внесення окремих операцій до

формування звітів і аналітичних висновків. Розроблюваний вебдодаток має забезпечити зручне ведення фінансових записів, контроль рахунків, аналіз витрат і підтримку прийняття більш усвідомлених фінансових рішень.

1.5 Опис системи

Розроблювана система є вебдодатком для обліку фінансових операцій користувача. Вона призначена для фіксації доходів, витрат і переказів між рахунками, перегляду поточного фінансового стану, формування звітів та аналізу фінансових звичок.

Система реалізується як клієнт-серверний вебзастосунок. Клієнтська частина розробляється з використанням Angular, який є фреймворком для створення швидких і масштабованих вебзастосунків [1]. Серверна частина створюється на основі Node.js та Express.js. Express.js є мінімалістичним і гнучким вебфреймворком для створення вебзастосунків та API [4, 11].

Для збереження даних використовується PostgreSQL, що виділяється підтримкою цілісності даних, ACID-властивостей і широкого набору можливостей [14]. Для взаємодії серверної частини з базою даних використовується Prisma ORM, що забезпечує роботу з PostgreSQL через опис моделей, міграції та Prisma Client [15].

У системі передбачено дві основні ролі: гість і авторизований користувач. Гість має доступ лише до сторінок реєстрації, входу, підтвердження email та відновлення пароля. Авторизований користувач отримує доступ до основних модулів системи: дашборду, рахунків, категорій, фінансових операцій, звітності, аналітики та профілю.

Основними модулями системи є:

– модуль авторизації – забезпечує реєстрацію, вхід, підтвердження email, відновлення пароля та захист доступу до системи;
– модуль рахунків – дозволяє створювати, переглядати, редагувати та видаляти рахунки користувача;

- модуль категорій – забезпечує створення категорій доходів і витрат;
- модуль фінансових операцій – дозволяє додавати доходи, витрати й перекази між рахунками;
- дашборд – відображає коротке фінансове зведення користувача;
- модуль звітності – формує деталізовані фінансові звіти за обраний період;
- модуль PDF-експорту – дозволяє зберегти фінансовий звіт у форматі PDF;
- модуль аналітики – формує висновки й рекомендації щодо фінансових звичок користувача.

Основною сутністю системи є фінансова операція. Вона може мати тип доходу, витрати або переказу. При створенні доходу система збільшує баланс відповідного рахунку, при створенні витрати – зменшує його, а при переказі зменшує баланс одного рахунку та збільшує баланс іншого. Така логіка дозволяє підтримувати актуальний стан рахунків користувача.

Доступ до персональних фінансових даних захищається за допомогою JWT-токена [12]. Після входу користувача система зберігає токен на клієнтській стороні та додає його до запитів на захищені API-маршрути. Сервер перевіряє токен і дозволяє користувачу працювати лише з власними рахунками, категоріями та операціями.

Загальна логіка роботи системи полягає в тому, що користувач створює рахунки, додає категорії, після чого вносить фінансові операції. На основі цих даних система автоматично формує статистику, звіти та аналітичні висновки. Таким чином, вебдодаток забезпечує повний цикл персонального фінансового обліку: від введення операцій до аналізу фінансової поведінки.

1.6 Огляд та порівняння аналогів

На ринку застосунків для обліку фінансів представлено багато готових рішень.

Розглянемо найбільш наближені до нашої системи аналоги на різних платформах.

BudgetBakers – застосунок для персонального й сімейного фінансового менеджменту, що підтримує відстеження витрат, створення бюджетів, синхронізацію з банківськими рахунками, планові платежі, аналіз грошових потоків і роботу з кількома валютами. Надає функціонал як для особистого користування на безоплатній основі, так і для бізнесу за індивідуально узгодженою ціною [2].

Spendee – сервіс для контролю готівкових витрат, банківських рахунків, електронних гаманців і криптогаманців. Надає можливість швидко переглядати доходи й витрати, працювати з бюджетами та бачити фінанси в одному місці. За аналогом з попереднім застосунком надає різний функціонал відповідно до підписки [17].

Monefy – застосунок для швидкого ручного обліку витрат і бюджету. Основною перевагою є швидкість та зручність для клієнта без складного налаштування [8].

Money Manager -– застосунок для ведення особистих фінансів, перегляду щотижневої та щомісячної статистики, роботи з бюджетами, фільтрами, календарем, графіками та збереженням фото чеків. Недоліком є наявність реклами [9].

У таблиці 1.1 наведено порівняння аналогів з нашою системою за базовими критеріями.

Таблиця 1.1 – Порівняння аналогів

Критерій	**BudgetBakers**	**Spendee**	**Monefy**	**Money Manager**	**Наша система**
Облік доходів і витрат	Так	Так	Так	Так	Так
Рахунки / гаманці	Так	Так	Так	Так	Так
Категорії операцій	Так	Так	Так	Так	Так
Перекази між рахунками	Так	Так	Обмежено / залежить від версії	Так	Так

Продовження таблиці 1.1

Критерій	**BudgetBakers**	**Spendee**	**Monefy**	**Money Manager**	**Наша система**
Дашборд	Так	Так	Так	Так	Так
Звіти	Так	Так	Так	Так	Так
Експорт звітів	Так / залежить від версії	Так / залежить від версії	Так / залежить від версії	Так / залежить від версії	Так
Аналітика фінансових звичок	Частково	Частково	Частково	Частково	Так
Платні функції	Так	Так	Ні	Ні, але є реклама	Ні
Призначення	Персональні та сімейні фінанси	Особисті та спільні бюджети	Простий ручний облік	Детальний облік і статистика	Повний цикл обліку

Аналіз аналогів показав, що більшість наявних рішень орієнтовані на швидке внесення витрат, побудову бюджетів і перегляд статистики. Головною перевагою є готовий інтерфейс, широкий набір функцій і зручність для щоденного використання. Із мінусів можна виділити наявність платних функцій або реклами.

Головною перевагою нашої системи є реалізація основного циклу персонального фінансового обліку у вигляді зрозумілого вебдодатку: реєстрація користувача, рахунки, категорії, доходи, витрати, перекази, дашборд, звітність, PDF-експорт та аналітика фінансових звичок.

2 ПРОЄКТУВАННЯ

У другому розділі розглядається проєктування вебдодатку обліку фінансових операцій. На данному етапі визначається структура бази даних, основні сутності системи, зв’язки між ними, а також моделюються ключові процеси взаємодії користувача із системою.

Для проєктування системи використано ER-модель бази даних та UML-діаграми. ER-модель дозволяє відобразити логічну структуру даних, а UML-діаграми – показати функціональні можливості системи, взаємодію користувача з модулями вебдодатку та послідовність виконання основного бізнес-процесу [5, 10, 14].

2.1 ER-модель бази даних та опис сутностей системи

База даних вебдодатку призначена для збереження інформації про користувачів, рахунки, категорії та фінансові операції. Основна логіка системи побудована навколо фінансової операції, яка може бути доходом, витратою або переказом між рахунками.

ER-модель системи наведено на рисунку 2.1. У базі даних виділено чотири основні сутності: User, Account, Category та Transaction.

User – сутність, що зберігає дані користувача системи. Вона містить ім’я, email, хеш пароля, роль, статус підтвердження електронної пошти та службові дати створення й оновлення запису. Користувач є власником рахунків, категорій і фінансових операцій.

Account – сутність, що описує рахунок користувача. Рахунок може відповідати готівці, банківській картці, банківському рахунку, накопичувальному рахунку або іншому джерелу зберігання коштів. Для кожного рахунку зберігаються назва, тип, валюта, початковий баланс, поточний баланс та опис.

Category – сутність, що використовується для групування фінансових операцій. Категорія може належати до доходів або витрат. Вона має назву, тип, колір та умовне позначення іконки. Категорії потрібні для подальшого формування звітів і аналітики фінансових звичок.

Transaction – основна сутність системи, яка описує фінансову операцію користувача. Операція може мати тип INCOME, EXPENSE або TRANSFER. Для кожної операції зберігаються сума, дата, коментар, основний рахунок, користувач, а також категорія або цільовий рахунок залежно від типу операції.

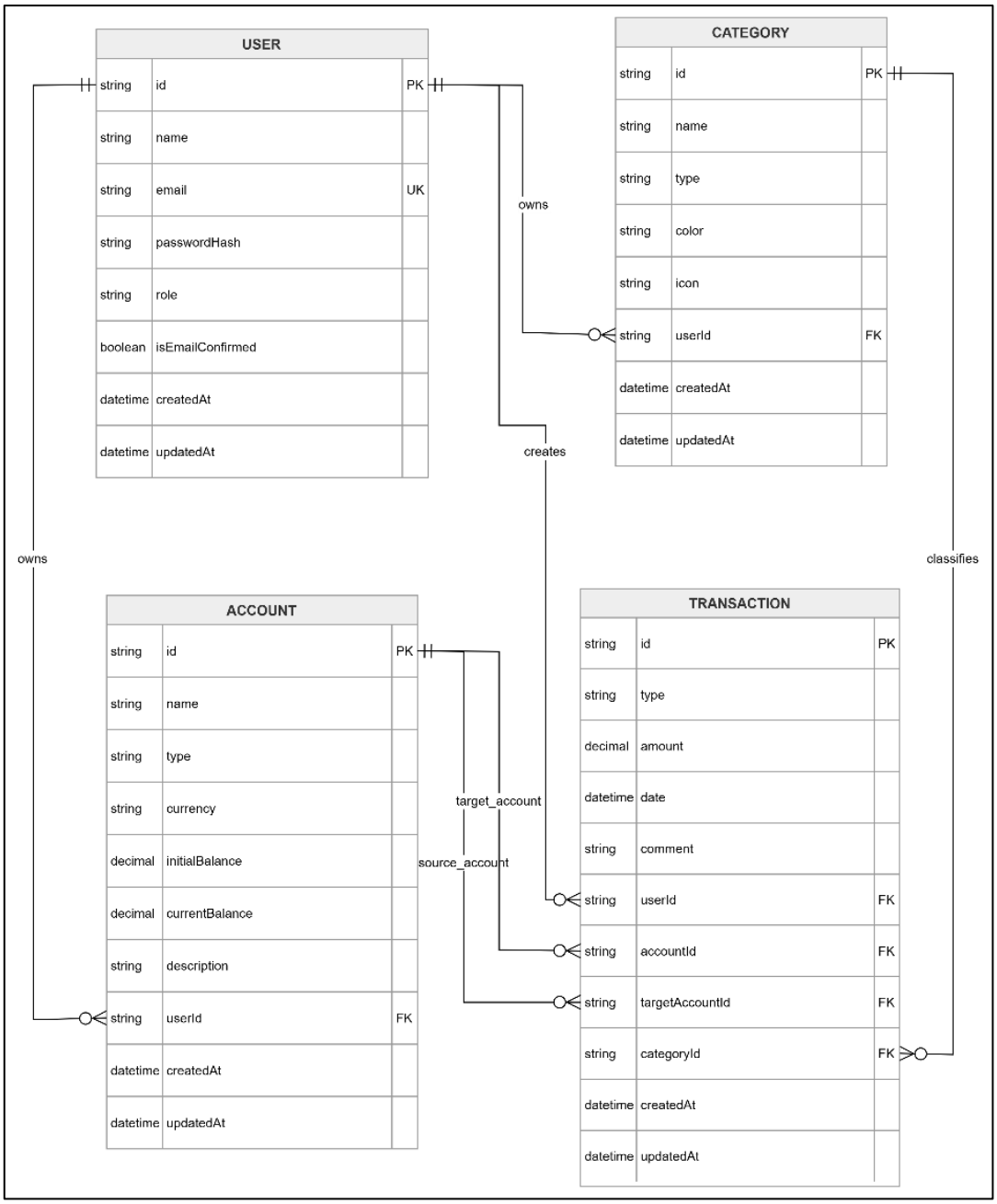


Рисунок 2.1 – ER-діаграма

Між сутностями встановлено наступні зв’язки:

– один користувач може мати багато рахунків;
– один користувач може мати багато категорій;
– один користувач може створювати багато фінансових операцій;
– одна фінансова операція обов’язково пов’язана з основним рахунком;
– фінансова операція типу «переказ» може мати цільовий рахунок;
– фінансові операції типу «дохід» або «витрата» можуть бути пов’язані з категорією.

Особливістю моделі є наявність двох зв’язків між рахунком і фінансовою операцією. Поле accountId використовується для основного рахунку операції. Для доходу це рахунок, на який надходять кошти, для витрати – рахунок, з якого списуються кошти, а для переказу – рахунок, з якого кошти переводяться. Поле targetAccountId використовується тільки для переказів і визначає рахунок, на який надходять кошти.

Для фінансових сум у системі використовується тип Decimal, оскільки фінансові значення потребують точного збереження. Використання чисел із плаваючою комою може призводити до похибок під час обчислень, тому для сум операцій і балансів рахунків застосовується окремий числовий тип із фіксованою точністю.

Запропонована ER-модель забезпечує цілісність фінансових даних, розмежування інформації між користувачами та правильне оновлення балансів рахунків після створення, редагування або видалення фінансових операцій. Модель є основою для реалізації серверної логіки, звітності, дашборду та аналітики фінансових звичок.

2.2 Діаграма варіантів використання

Діаграма варіантів використання відображає основні можливості системи з погляду користувача [5, 10].

Діаграму варіантів використання системи наведено на рисунку 2.2. У системі виділено два актори: гість та авторизований користувач.

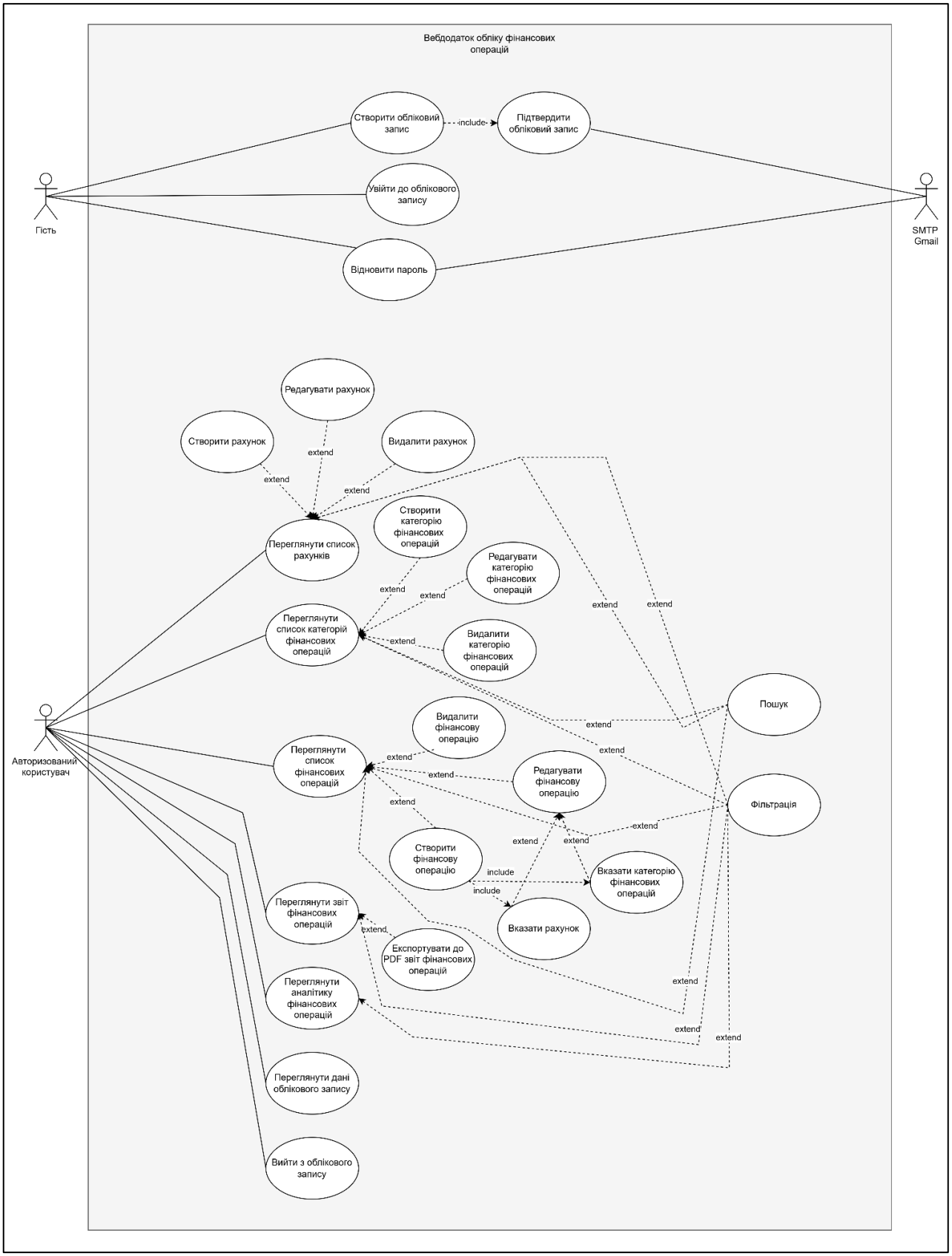


Рисунок 2.2 – Діаграма варіантів використання

Гість має можливість створити обліковий запис, підтвердити email, увійти до системи або відновити пароль. Авторизований користувач отримує доступ до основних модулів системи: рахунків, категорій, фінансових операцій, дашборду, звітності, аналітики та профілю.

Основним прецедентом системи є «Створити фінансову операцію», оскільки саме він реалізує ключову функцію вебдодатку – облік доходів, витрат і переказів між рахунками користувача. У межах цього сценарію користувач обирає тип операції, вводить суму, дату, рахунок, категорію або цільовий рахунок, після чого система перевіряє коректність даних, зберігає операцію в базі даних і автоматично оновлює баланс відповідних рахунків. Надалі ці дані використовуються для формування дашборду, звітності, PDF-експорту та аналітики фінансових звичок.

2.2.1 Опис варіантів використання

Прецедент «Створити обліковий запис».

Призначення: даний варіант використання надає можливість гостю створити обліковий запис у системі.

Передумови: користувач знаходиться на сторінці реєстрації.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить ім’я, email, пароль і натискає кнопку реєстрації. Система перевіряє коректність введених даних, створює обліковий запис, хешує пароль і надсилає лист для підтвердження email.

Виняткова ситуація 1: невалідні дані – система виводить повідомлення про помилку та не створює обліковий запис.

Виняткова ситуація 2: email уже використовується – система повідомляє користувача, що обліковий запис із таким email уже існує.

Прецедент «Підтвердити обліковий запис».

Призначення: даний варіант використання надає можливість користувачу підтвердити email після реєстрації.

Передумови: користувач отримав лист із посиланням для підтвердження email.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач переходить за посиланням із листа. Система перевіряє токен підтвердження, змінює статус email на підтверджений і повідомляє користувача про успішне підтвердження.

Виняткова ситуація 1: токен підтвердження недійсний або прострочений – система виводить повідомлення про помилку.

Виняткова ситуація 2: обліковий запис уже підтверджено – система повідомляє користувача, що повторне підтвердження не потрібне.

Прецедент «Увійти до облікового запису».

Призначення: даний варіант використання надає можливість користувачу увійти до системи.

Передумови: користувач має створений і підтверджений обліковий запис.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить email, пароль і натискає кнопку входу. Система перевіряє облікові дані, формує JWT-токен і надає доступ до захищених сторінок вебдодатку.

Виняткова ситуація 1: email або пароль введено неправильно – система виводить повідомлення про помилку.

Виняткова ситуація 2: email не підтверджено – система блокує вхід і повідомляє користувача про необхідність підтвердження облікового запису.

Прецедент «Відновити пароль».

Призначення: даний варіант використання надає можливість користувачу відновити доступ до облікового запису.

Передумови: користувач знаходиться на сторінці відновлення пароля.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить email і надсилає запит на відновлення пароля. Система формує токен відновлення, надсилає лист із посиланням і дозволяє користувачу встановити новий пароль.

Виняткова ситуація 1: введено некоректний email – система виводить повідомлення про помилку.

Виняткова ситуація 2: токен відновлення недійсний або прострочений – система не змінює пароль і повідомляє користувача про помилку.

Прецедент «Переглянути список рахунків».

Призначення: даний варіант використання надає можливість авторизованому користувачу переглянути власні рахунки.

Передумови: користувач авторизований і знаходиться на сторінці «Рахунки».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку рахунків. Система надсилає запит до сервера, отримує список рахунків поточного користувача та відображає їх у таблиці.

Виняткова ситуація 1: відсутні рахунки – система відображає порожній стан і пропонує створити перший рахунок.

Виняткова ситуація 2: помилка завантаження даних – система виводить повідомлення про технічну проблему.

Прецедент «Створити рахунок».

Призначення: даний варіант використання надає можливість авторизованому користувачу створити новий рахунок.

Передумови: користувач авторизований і знаходиться на сторінці «Рахунки».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить назву рахунку, обирає тип, валюту, початковий баланс, за потреби додає опис і натискає кнопку створення. Система перевіряє введені дані, створює рахунок і додає його до списку рахунків користувача.

Виняткова ситуація 1: невалідні дані – система виводить повідомлення про помилку та не створює рахунок.

Виняткова ситуація 2: помилка збереження – система виводить повідомлення про технічну проблему.

Прецедент «Редагувати рахунок».

Призначення: даний варіант використання надає можливість авторизованому користувачу змінити дані власного рахунку.

Передумови: користувач авторизований, має створений рахунок і знаходиться на сторінці «Рахунки».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає рахунок для редагування, змінює назву, тип, валюту або опис і натискає кнопку збереження. Система перевіряє дані та оновлює рахунок.

Виняткова ситуація 1: невалідні дані – система виводить повідомлення про помилку.

Виняткова ситуація 2: користувач намагається редагувати рахунок, який йому не належить – система блокує дію.

Прецедент «Видалити рахунок».

Призначення: даний варіант використання надає можливість авторизованому користувачу видалити власний рахунок.

Передумови: користувач авторизований, має створений рахунок і знаходиться на сторінці «Рахунки».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку видалення біля обраного рахунку та підтверджує дію. Система перевіряє, чи не використовується рахунок у фінансових операціях, після чого видаляє його.

Виняткова ситуація 1: рахунок уже використовується у фінансових операціях – система блокує видалення.

Виняткова ситуація 2: рахунок не знайдено або він не належить користувачу – система виводить повідомлення про помилку.

Прецедент «Переглянути список категорій фінансових операцій».

Призначення: даний варіант використання надає можливість авторизованому користувачу переглянути власні категорії доходів і витрат.

Передумови: користувач авторизований і знаходиться на сторінці «Категорії».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку категорій. Система завантажує список категорій поточного користувача та відображає їх у таблиці.

Виняткова ситуація 1: відсутні категорії – система відображає порожній стан.

Виняткова ситуація 2: помилка завантаження даних – система виводить повідомлення про технічну проблему.

Прецедент «Створити категорію фінансової операції».

Призначення: даний варіант використання надає можливість авторизованому користувачу створити категорію доходу або витрати.

Передумови: користувач авторизований і знаходиться на сторінці «Категорії».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить назву категорії, обирає її тип, колір, іконку та натискає кнопку створення. Система перевіряє дані, створює категорію та додає її до списку.

Виняткова ситуація 1: невалідні дані – система виводить повідомлення про помилку.

Виняткова ситуація 2: категорія з такою назвою і типом уже існує для користувача – система блокує створення дубліката.

Прецедент «Редагувати категорію фінансової операції».

Призначення: даний варіант використання надає можливість авторизованому користувачу змінити дані власної категорії.

Передумови: користувач авторизований, має створену категорію і знаходиться на сторінці «Категорії».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає категорію для редагування, змінює її назву, тип, колір або іконку та натискає кнопку збереження. Система перевіряє дані та оновлює категорію.

Виняткова ситуація 1: невалідні дані – система виводить повідомлення про помилку.

Виняткова ситуація 2: користувач намагається створити дубль категорії – система повідомляє про помилку.

Прецедент «Видалити категорію фінансової операції».

Призначення: даний варіант використання надає можливість авторизованому користувачу видалити власну категорію.

Передумови: користувач авторизований, має створену категорію і знаходиться на сторінці «Категорії».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку видалення біля категорії та підтверджує дію. Система перевіряє, чи не використовується категорія у фінансових операціях, після чого видаляє її.

Виняткова ситуація 1: категорія вже використовується у фінансових операціях – система блокує видалення.

Виняткова ситуація 2: категорію не знайдено або вона не належить користувачу – система виводить повідомлення про помилку.

Прецедент «Переглянути список фінансових операцій».

Призначення: даний варіант використання надає можливість авторизованому користувачу переглянути список власних фінансових операцій.

Передумови: користувач авторизований і знаходиться на сторінці «Фінансові операції».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку фінансових операцій. Система завантажує список операцій поточного користувача, а також пов’язані рахунки й категорії. Операції відображаються у таблиці з використанням пагінації.

Виняткова ситуація 1: відсутні фінансові операції – система відображає порожній стан.

Виняткова ситуація 2: помилка завантаження даних – система виводить повідомлення про технічну проблему.

Прецедент «Створити фінансову операцію».

Призначення: даний варіант використання надає можливість

авторизованому користувачу створити дохід, витрату або переказ між рахунками.

Передумови: користувач авторизований, має хоча б один рахунок і відповідні категорії для доходів або витрат.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку «Фінансові операції», обирає тип операції, вводить суму, дату, рахунок, категорію або цільовий рахунок і натискає кнопку створення. Система перевіряє введені дані, перевіряє належність рахунку й категорії поточному користувачу, створює фінансову операцію та автоматично оновлює баланс відповідного рахунку або рахунків.

Виняткова ситуація 1: невалідні дані – система виводить повідомлення про помилку та не створює операцію.

Виняткова ситуація 2: користувач обрав рахунок або категорію, які йому не належать – система блокує створення операції.

Виняткова ситуація 3: під час переказу обрано однаковий основний і цільовий рахунок – система повідомляє про помилку.

Виняткова ситуація 4: для доходу або витрати не обрано категорію – система повідомляє користувача про необхідність вибору категорії.

Прецедент «Редагувати фінансову операцію».

Призначення: даний варіант використання надає можливість авторизованому користувачу змінити дані раніше створеної фінансової операції.

Передумови: користувач авторизований, має створену фінансову операцію і знаходиться на сторінці «Фінансові операції».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає операцію для редагування, змінює тип, суму, дату, рахунок, категорію або коментар і натискає кнопку збереження. Система відкочує попередній вплив операції на баланс, оновлює операцію та застосовує новий вплив на баланс рахунку або рахунків.

Виняткова ситуація 1: невалідні дані – система не зберігає зміни та виводить повідомлення про помилку.

Виняткова ситуація 2: операцію не знайдено або вона не належить користувачу – система блокує редагування.

Виняткова ситуація 3: для переказу обрано однакові рахунки – система повідомляє про помилку.

Прецедент «Видалити фінансову операцію».

Призначення: даний варіант використання надає можливість авторизованому користувачу видалити фінансову операцію.

Передумови: користувач авторизований, має створену фінансову операцію і знаходиться на сторінці «Фінансові операції».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку видалення біля фінансової операції та підтверджує дію. Система оновлює баланс рахунку або рахунків, після чого видаляє запис операції.

Виняткова ситуація 1: операцію не знайдено або вона не належить користувачу – система блокує видалення.

Виняткова ситуація 2: виникла технічна помилка під час оновлення балансу або видалення – система виводить повідомлення про помилку.

Прецедент «Фільтрувати дані».

Призначення: даний варіант використання надає можливість авторизованому користувачу фільтрувати рахунки, категорії та фінансові операції.

Передумови: користувач авторизований і знаходиться на сторінці відповідного модуля.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає параметри фільтрації. Система надсилає запит до сервера та відображає записи, що відповідають вибраним параметрам.

Виняткова ситуація 1: за заданими параметрами записи не знайдено – система відображає порожній стан.

Виняткова ситуація 2: параметри фільтрації некоректні -– система виводить повідомлення про помилку.

Прецедент «Виконати пошук даних».

Призначення: даний варіант використання надає можливість авторизованому користувачу виконувати пошук рахунків, категорій або фінансових операцій.

Передумови: користувач авторизований і знаходиться на сторінці відповідного модуля.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач вводить пошуковий запит. Система виконує пошук та відображає знайдені записи.

Виняткова ситуація 1: записи за пошуковим запитом не знайдено – система повідомляє користувача про відсутність результатів.

Виняткова ситуація 2: пошуковий запит некоректний або занадто довгий – система виводить повідомлення про помилку.

Прецедент «Переглянути звіт фінансових операцій».

Призначення: даний варіант використання надає можливість авторизованому користувачу переглянути фінансовий звіт за обраний період.

Передумови: користувач авторизований і має створені фінансові операції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку «Звіти». Система формує звіт за поточний або обраний період і відображає загальні доходи, витрати, перекази, чистий результат, найбільшу витрату, групування за категоріями й рахунками, динаміку доходів і витрат та список операцій.

Виняткова ситуація 1: за обраний період немає фінансових операцій – система відображає порожній стан.

Виняткова ситуація 2: виникла помилка формування звіту – система виводить повідомлення про технічну проблему.

Прецедент «Експортувати звіт до PDF».

Призначення: даний варіант використання надає можливість авторизованому користувачу зберегти фінансовий звіт у форматі PDF.

Передумови: користувач авторизований і знаходиться на сторінці «Звіти».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Експорт PDF». Система формує звіт за вибраними параметрами, створює PDF-файл і передає його користувачу для завантаження.

Виняткова ситуація 1: дані для звіту відсутні – система повідомляє користувача, що PDF не може бути сформований або буде сформований із порожніми даними.

Виняткова ситуація 2: помилка генерації PDF – система виводить повідомлення про технічну проблему та пропонує повторити спробу.

Прецедент «Переглянути аналітику фінансових операцій».

Призначення: даний варіант використання надає можливість авторизованому користувачу переглянути аналітику фінансових звичок.

Передумови: користувач авторизований і має внесені фінансові операції.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку «Аналітика». Система розраховує суму доходів і витрат, середню щоденну витрату, частку витрат від доходів, найбільшу категорію витрат, найбільшу разову витрату та зміну витрат порівняно з попереднім періодом. На основі цих даних система формує висновки та рекомендації.

Виняткова ситуація 1: даних для аналітики недостатньо – система відображає повідомлення про відсутність фінансових операцій.

Виняткова ситуація 2: помилка розрахунку аналітики – система виводить повідомлення про технічну проблему.

Прецедент «Вийти з облікового запису».

Призначення: даний варіант використання надає можливість авторизованому користувачу завершити роботу із системою.

Передумови: користувач авторизований у системі.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку виходу. Система видаляє JWT-токен і дані користувача з локального сховища та перенаправляє користувача на сторінку входу.

Виняткова ситуація 1: локальні дані сесії вже відсутні – система все одно перенаправляє користувача на сторінку входу.

Виняткова ситуація 2: сталася помилка під час переходу між сторінками – користувач може оновити сторінку та повторити дію.

Прецедент «Переглянути дані облікового запису».

Призначення: даний варіант використання надає можливість авторизованому користувачу переглянути дані власного облікового запису.

Передумови: користувач авторизований і знаходиться на сторінці «Профіль».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку профілю. Система відображає ім’я користувача, email, роль, статус підтвердження email та дату створення облікового запису.

Виняткова ситуація 1: дані користувача не вдалося отримати – система виводить повідомлення про помилку.

Виняткова ситуація 2: JWT-токен недійсний – система видаляє локальну сесію та перенаправляє користувача на сторінку входу.

2.3 Діаграма діяльності основного бізнес-процесу

Діаграма діяльності використовується для відображення послідовності дій під час виконання основного бізнес-процесу системи [5, 10].

Для розроблюваного вебдодатку основним процесом є створення фінансової операції, оскільки саме він забезпечує фіксацію доходів, витрат і переказів між рахунками користувача.

Діаграму діяльності для прецеденту «Створити фінансову операцію» наведено на рисунку 2.3.

Процес створення фінансової операції починається з того, що авторизований користувач відкриває сторінку фінансових операцій. Система завантажує список рахунків, категорій та раніше створених операцій.

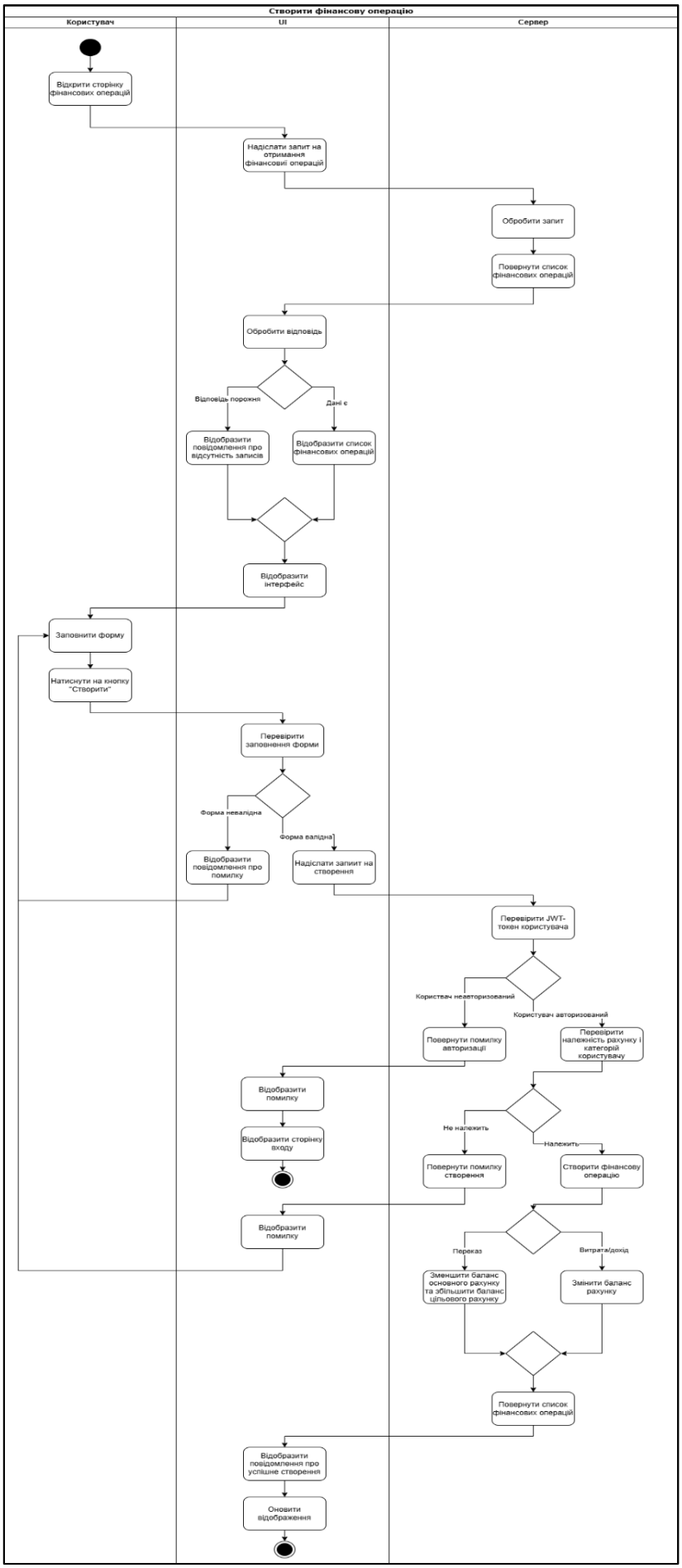

Створити фінансову операцію
Користувач
UI
Сервер
Відкрити сторінку фінансових операцій
Надіслати запит на отримання фінансові операцій
Обробити запит
Повернути список фінансових операцій
Обробити відповідь
Відповідь порожня
Дані є
Відобразити повідомлення про відсутність записів
Відобразити список фінансових операцій
Відобразити інтерфейс
Заповнити форму
Натиснути на кнопку "Створити"
Перевірити заповнення форми
Форма невалідна
Форма валідна
Відобразити повідомлення про помилку
Надіслати запиит на створення
Перевірити JWT-токен користувача
Користувач неавторизований
Користувач авторизований
Повернути помилку авторизації
Перевірити належність рахунку і категорій користувачу
Відобразити помилку
Відобразити сторінку входу
Не належить
Належить
Повернути помилку створення
Створити фінансову операцію
Відобразити помилку
Переказ
Витрата/дохід
Зменшити баланс основного рахунку та збільшити баланс цільового рахунку
Змінити баланс рахунку
Повернути список фінансових операцій
Відобразити повідомлення про успішне створення
Оновити відображення

Рисунок 2.3

Після цього користувач заповнює форму, обирає тип операції, вводить суму, дату, рахунок, а також категорію або цільовий рахунок залежно від типу операції.

2.4 Діаграма класів основного бізнес-процесу

Діаграма класів використовується для відображення основних структурних елементів системи, їхніх атрибутів, методів та зв’язків між ними [5, 10]. Оскільки розроблювана система має клієнт-серверну архітектуру, то доцільніше розділити діаграму класів на дві частини: клієнтську та серверну.

На рисунках 2.4 та 2.5 зображено діаграми класів зі сторони UI та сервера для прецедента «Створити фінансову операцію».

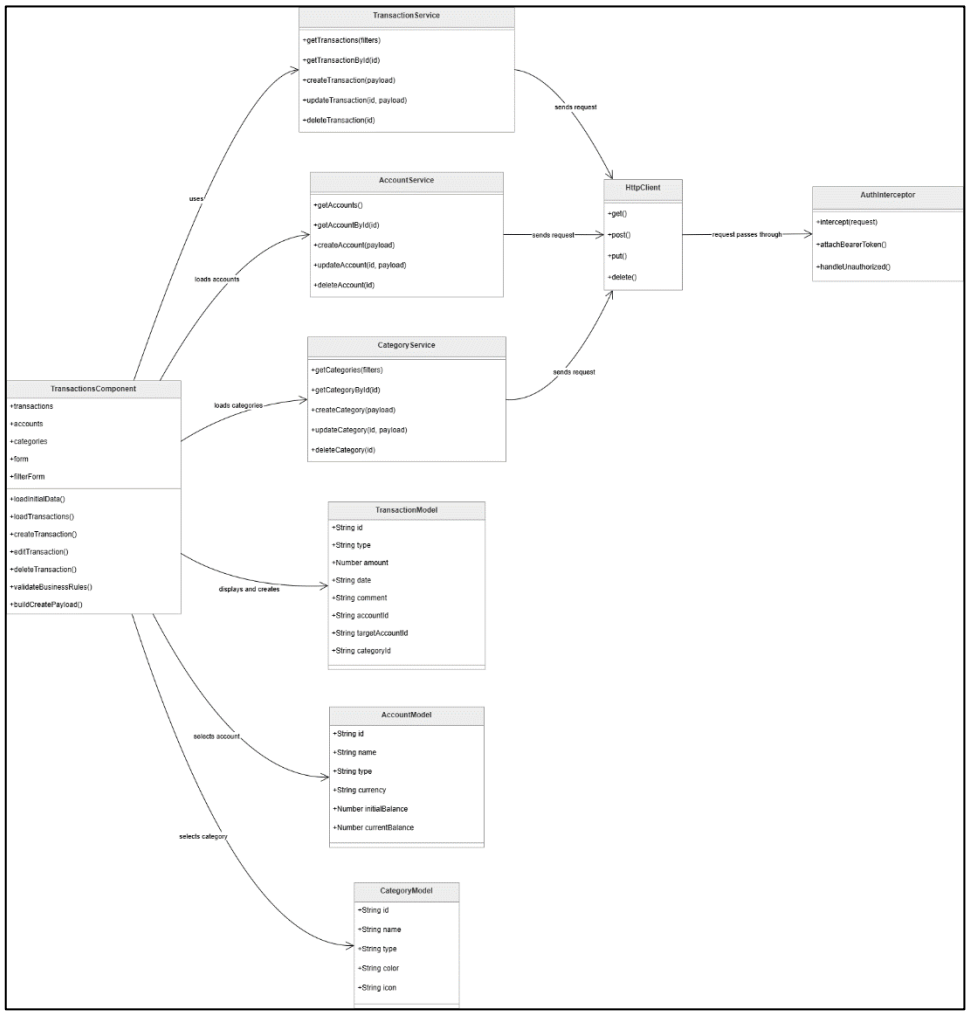


Рисунок 2.4 – Діаграма класів зі сторони клієнта

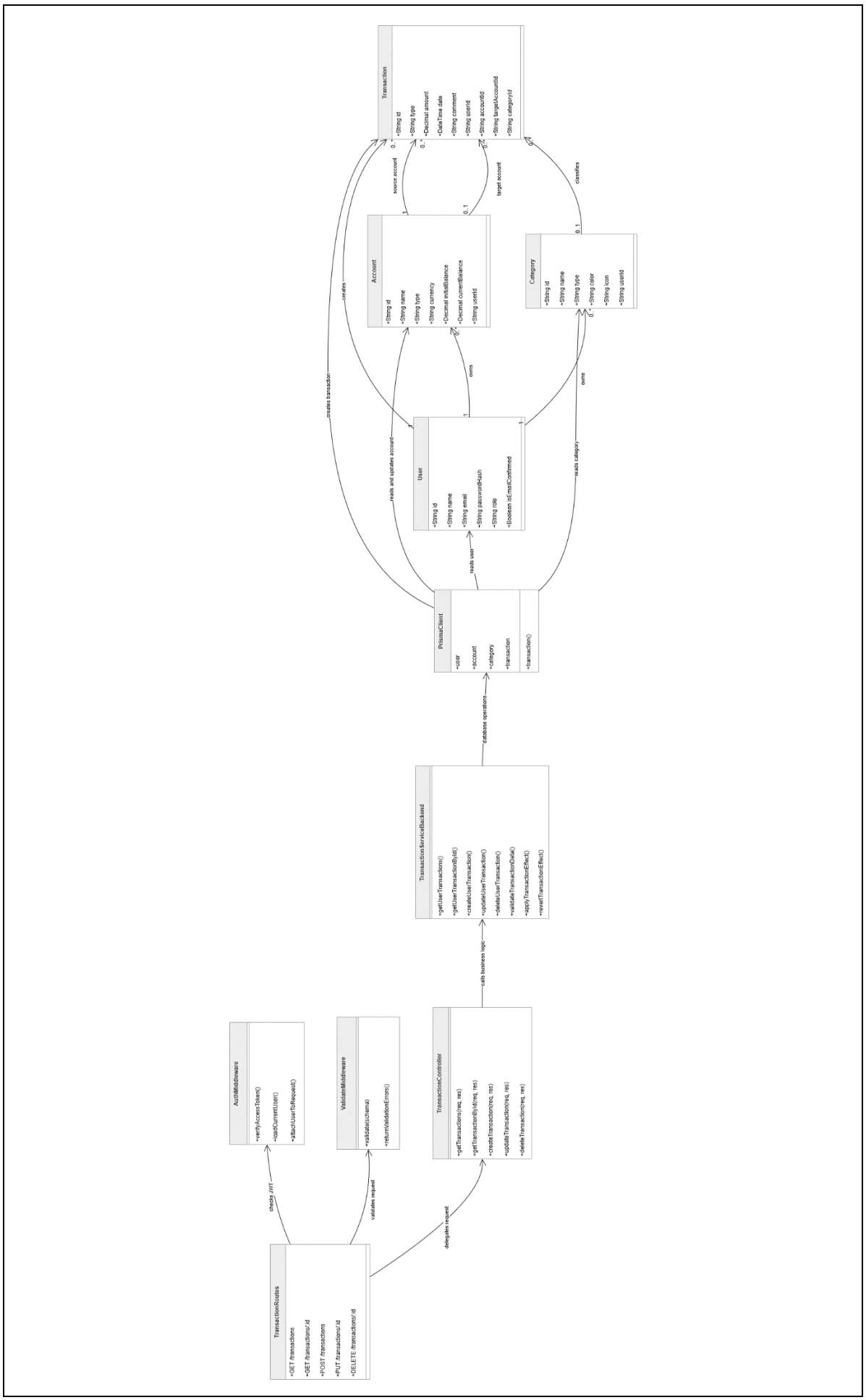


Рисунок 2.5 – Діаграма класів зі сторони сервера

2.5 Діаграма послідовності основного бізнес-процесу

Діаграма послідовності використовується для відображення взаємодії між користувачем, клієнтською частиною, серверною частиною та базою даних у межах конкретного сценарію роботи системи [5 ,10].

Діаграму послідовності для прецеденту «Створити фінансову операцію» наведено на рисунку 2.6.

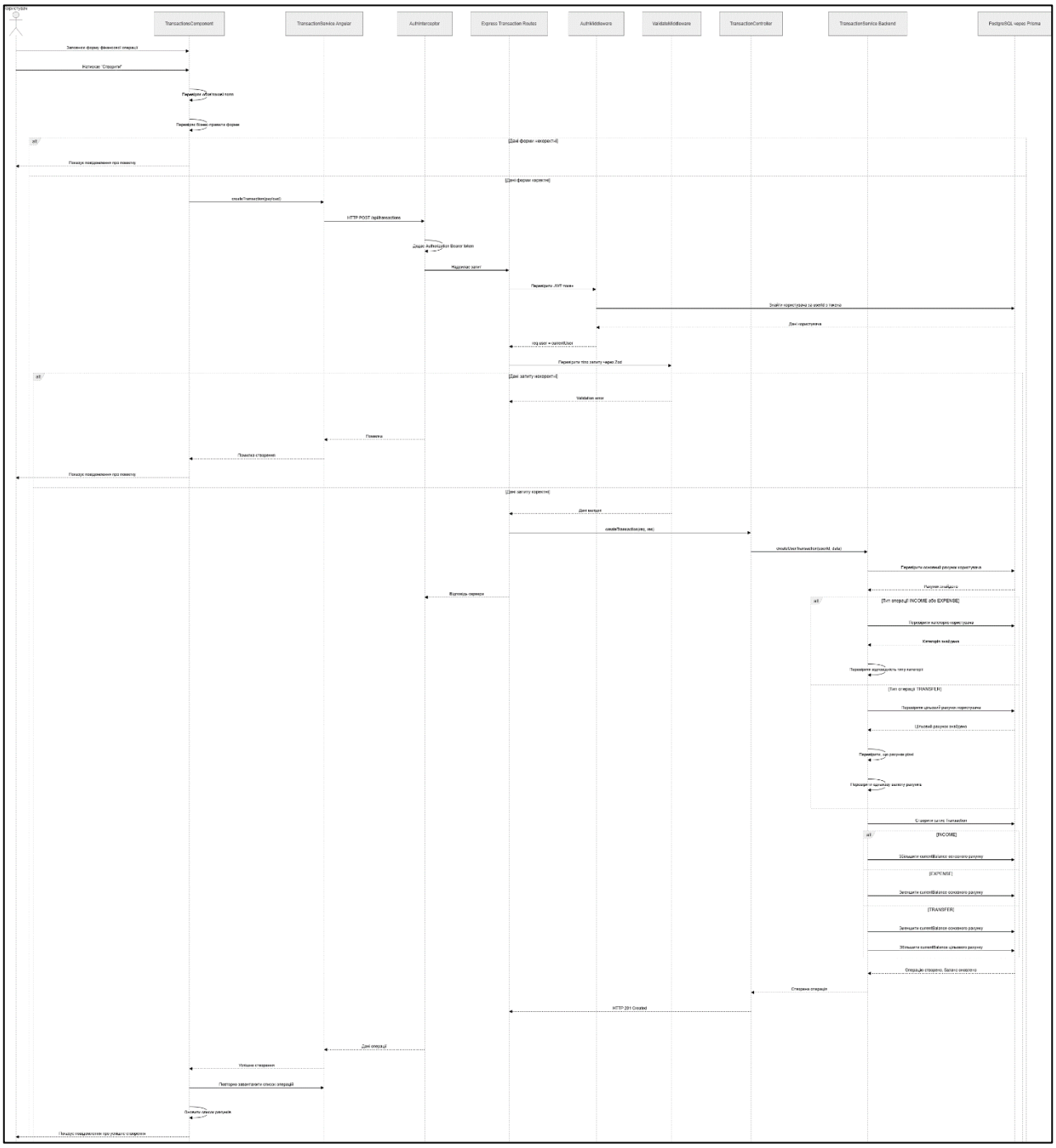


Рисунок 2.6

Під час виконання цього сценарію користувач заповнює форму створення операції на клієнтській частині. Angular-компонент перевіряє введені дані та через сервіс надсилає HTTP-запит до Express API. Далі сервер перевіряє JWT-токен, виконує валідацію запиту, перевіряє належність рахунків і категорій поточному користувачу, створює запис фінансової операції та оновлює баланс відповідного рахунку або рахунків.

2.6 Діаграма розгортання

Діаграма розгортання відображає фізичну структуру системи та показує, на яких вузлах розміщуються її основні компоненти [5, 10]. Русунок 2.7 демонструє діаграму розгортання системи.

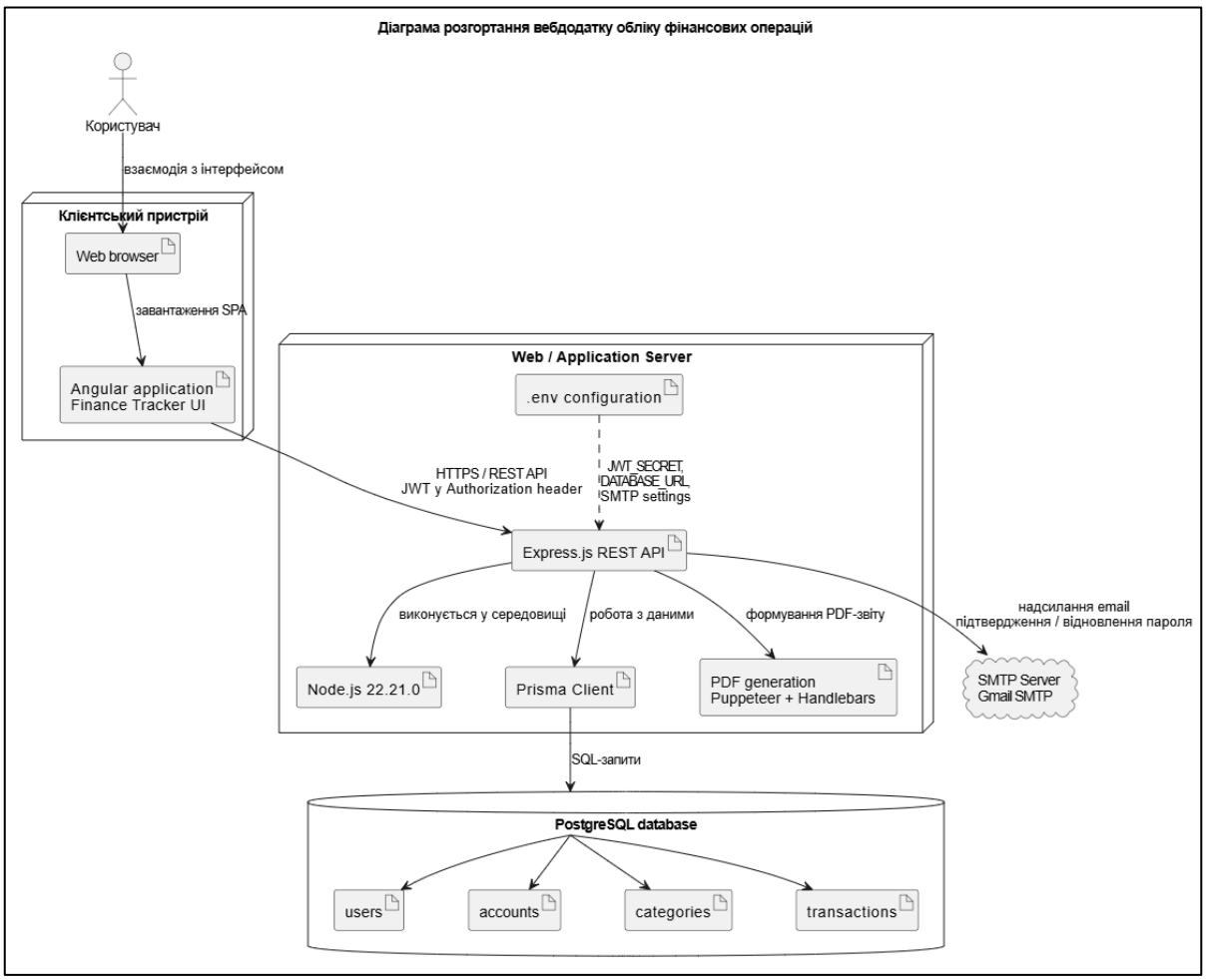


Рисунок 2.7 – Діаграма розгортання

Розроблювана система має клієнт-серверну архітектуру: користувач працює з Angular-застосунком у браузері, клієнтська частина надсилає HTTP-запити до Express API, серверна частина взаємодіє з PostgreSQL через Prisma ORM, а для підтвердження email і відновлення пароля використовується SMTP-сервер. Така структура відповідає опису системи як клієнт-серверного вебзастосунку з Angular, Express.js, PostgreSQL і Prisma ORM.

У другому розділі було виконано проєктування вебдодатку обліку фінансових операцій. На основі функціональних вимог визначено структуру бази даних, основні сутності системи та зв’язки між ними. ER-модель показала, що ключовими сутностями є користувач, рахунок, категорія та фінансова операція.

Також було розглянуто основні варіанти використання системи для гостя та авторизованого користувача. Виділено прецедент «Створити фінансову операцію», оскільки саме він реалізує основну бізнес-логіку системи: створення доходу, витрати або переказу між рахунками з автоматичним оновленням балансу.

Для детальнішого опису роботи системи побудовано діаграму діяльності, діаграми класів клієнтської та серверної частин, діаграму послідовності та діаграму розгортання. Вони дозволяють показати, як користувач взаємодіє з інтерфейсом, як Angular-клієнт надсилає запити до Express API, як сервер перевіряє дані та як інформація зберігається у PostgreSQL.

Таким чином, у другому розділі сформовано проєктну основу системи, яка надалі використовується для програмної реалізації вебдодатку.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Опис інструментів розробки

Для реалізації вебдодатку обліку фінансових операцій використано сучасний стек технологій, що охоплює клієнтську та серверну частини, базу даних, авторизацію, стилізацію інтерфейсу й формування PDF-звітів.

Клієнтську частину системи реалізовано на основі фреймворка Angular, який надає засоби для побудови компонентів, маршрутизації, роботи з формами та HTTP-запитами [1]. У проєкті він використовується для реалізації сторінок авторизації, дашборду, рахунків, категорій, фінансових операцій, звітів, аналітики та профілю користувача.

Для стилізації інтерфейсу застосовано utility-first CSS-фреймворк Tailwind CSS, який дозволяє будувати інтерфейс за допомогою готових службових класів без необхідності створювати окремі CSS-правила [19]. У проєкті він використовується для реалізації світлої теми, адаптивної сітки, карток, таблиць, форм, кнопок і повідомлень про помилки.

Серверну частину реалізовано на платформі Node.js версії 22.21.0 – відкритому кросплатформному середовищі виконання JavaScript, що забезпечує запуск серверного коду та Express API. Для побудови самого API обрано фреймворк Express.js завдяки його гнучкості та широкому набору можливостей: він використовується для опису маршрутів, підключення middleware, обробки HTTP-запитів, авторизації, валідації даних і передавання запитів до сервісів [4, 11].

Для збереження даних обрано PostgreSQL – об’єктно-реляційну систему керування базами даних, що підтримує цілісність даних та ACID-властивості. Це критично важливо для системи фінансового обліку, де операції зі створення, редагування та видалення записів повинні зберігати узгодженість балансів рахунків. Взаємодію серверної частини з базою даних забезпечує Prisma ORM, яка надає типобезпечний доступ до бази даних, механізм міграцій і Prisma Client

для виконання запитів [14, 15]. У проєкті Prisma використовується для опису моделей User, Account, Category і Transaction та управління міграціями.

Формування PDF-звітів реалізовано за допомогою Handlebars і Puppeteer. Спочатку Handlebars як шаблонізатор поєднує HTML-шаблон із даними звіту й формує готовий HTML-документ, який потім Puppeteer перетворює на PDF-файл, керуючи браузером Chrome через DevTools Protocol [6 ,16].

Авторизацію користувачів реалізовано на основі стандарту JSON Web Token [12]. Після входу користувач отримує токен, який додається до кожного запиту на сервер. Сервер перевіряє токен і надає доступ лише до власних фінансових даних користувача, забезпечуючи таким чином захист API-маршрутів.

Отже, обраний стек технологій забезпечує повний цикл роботи застосунку: від побудови клієнтського інтерфейсу та обробки запитів на сервері до збереження даних у базі, розмежування доступу до персональної інформації й генерації звітів у форматі PDF.

3.2 Основні класи, інтерфейси та функції системи

Основним бізнес-процесом є взаємодія з фінансовими операціями. Отже, основними класами є ті, що забезпечують повний цикл цього процесу.

Зі сторони клієнту можна виділити наступні класи, інтерфейси та функції.

TransactionsComponent – Angular-компонент, що відповідає за сторінку фінансових операцій. Він забезпечує відображення форми створення операції, списку операцій, фільтрів і пагінації. Також компонент виконує первинну перевірку введених даних перед надсиланням запиту на сервер.

TransactionService – Angular-сервіс, що відповідає за взаємодію з серверною частиною під час роботи з фінансовими операціями. Через нього виконуються запити на отримання списку операцій, створення, редагування та видалення фінансової операції.

AccountService – Angular-сервіс, що використовується для отримання списку рахунків користувача. Дані рахунків потрібні під час створення операції, оскільки користувач має обрати основний рахунок, а для переказу також цільовий рахунок.

CategoryService – Angular-сервіс, що відповідає за отримання списку категорій доходів і витрат. Категорії використовуються під час створення операцій типу «Дохід» або «Витрата».

AuthInterceptor – функціональний HTTP-interceptor, що додає JWT-токен до запитів на захищені API-маршрути. Він забезпечує передавання даних авторизації під час створення фінансової операції.

TransactionModel – модель даних фінансової операції на клієнтській частині. Вона описує тип операції, суму, дату, коментар, рахунок, цільовий рахунок і категорію.

AccountModel – модель рахунку, що містить дані про назву рахунку, тип, валюту, початковий і поточний баланс.

CategoryModel – модель категорії фінансової операції, що містить назву, тип, колір та умовну іконку категорії.

Зі сторони серверу можна виділити наступні елементи.

Transaction.routes.js – файл маршрутів, що визначає API-шляхи для роботи з фінансовими операціями. Для створення операції використовується маршрут POST /api/transactions.

Auth.middleware.js – middleware, що перевіряє JWT-токен користувача та додає дані поточного користувача до запиту. Це потрібно для того, щоб система створювала фінансові операції лише для авторизованого користувача.

Validate.middleware.js – middleware, що перевіряє коректність вхідних даних перед передаванням запиту до контролера. Завдяки цьому сервер не обробляє некоректні або неповні дані.

Transaction.controller.js – контролер, що приймає запит на створення фінансової операції та передає його до сервісного шару. Контролер відповідає за обробку HTTP-запиту і формування відповіді клієнту.

Transaction.service.js – сервіс, що містить основну бізнес-логіку роботи з фінансовими операціями. Саме тут перевіряється належність рахунків і категорій користувачу, створюється операція та оновлюється баланс рахунку або рахунків.

PrismaClient – інструмент доступу до бази даних PostgreSQL. Він використовується для створення запису фінансової операції, отримання рахунків і категорій, а також оновлення балансів рахунків.

Transaction – модель фінансової операції у базі даних. Вона зберігає тип операції, суму, дату, коментар, користувача, основний рахунок, цільовий рахунок і категорію.

Account – модель рахунку у базі даних. Вона використовується для збереження початкового та поточного балансу. Під час створення фінансової операції значення поточного балансу змінюється.

Category – модель категорії у базі даних. Вона використовується для класифікації доходів і витрат.

User – модель користувача системи. Вона пов’язана з рахунками, категоріями та фінансовими операціями, що дозволяє розмежувати дані між різними користувачами.

На рисунку 3.1 наведено фрагмент реалізації створення фінансової операції на серверній частині, який демонструє основну бізнес-логіку: перевірку даних, створення операції та оновлення балансу рахунку або рахунків.

```
export async function createUserTransaction({ userId, data }) : Promise<any> { Show usages
    return prisma.$transaction(async (tx) : Promise<...> => {
        const preparedData : ... | ... = await validateTransactionData(tx, {userId, data}: {
            userId,
            data
        });

        const transaction = await tx.transaction.create({
            data: {
                type: preparedData.type,
                amount: toDecimal(preparedData.amount),
                date: preparedData.date,
                comment: preparedData.comment?.trim() || null,

                userId,

                accountId: preparedData.accountId,
                targetAccountId: preparedData.targetAccountId || null,
                categoryId: preparedData.categoryId || null
            },
            include: transactionInclude
```

Рисунок 3.1 – Функція створення фінансової операції

З повними реалізаціями основних компонентів та сервісів клієнтської частини можна ознайомитися у додатках А, Б, а з серверними обробниками та моделями – у додатках В, Г, Д.

3.3 Реалізація клієнтської частини

Клієнтська частина вебдодатку реалізована за допомогою Angular. Вона відповідає за відображення інтерфейсу користувача, роботу з формами, первинну перевірку введених даних і надсилання запитів до серверної частини.

Основним компонентом для роботи з фінансовими операціями є TransactionsComponent. Він забезпечує завантаження рахунків, категорій і списку фінансових операцій, а також містить форму створення доходу, витрати або переказу між рахунками. Перед створенням фінансової операції клієнтська частина перевіряє основні бізнес-правила. Наприклад, для доходу або витрати користувач повинен обрати категорію, а для переказу – цільовий рахунок. Також система перевіряє, щоб основний і цільовий рахунок під час переказу не збігалися. Фрагмент методу перевірки бізнес-правил наведено на рисунку 3.2.

```
private validateBusinessRules(): boolean {  Show usages
  const rawValue : { type: TransactionType; amount: number; ... } = this.form.getRawValue();

  this.errorMessage.set(null);

  if (!rawValue.accountId) {
    this.errorMessage.set('Оберіть основний рахунок.');
    return false;
  }

  if (rawValue.type === 'INCOME' || rawValue.type === 'EXPENSE') {
    if (!rawValue.categoryId) {
      this.errorMessage.set('Для доходу або витрати потрібно обрати категорію.');
      return false;
    }
  }

  if (rawValue.type === 'TRANSFER') {
    if (!rawValue.targetAccountId) {
      this.errorMessage.set('Для переказу потрібно обрати цільовий рахунок.');
      return false;
    }
```

Рисунок 3.2 – Фрагмент перевірки бізнес-правил під час створення фінансової операції

Після проходження перевірки компонент формує об’єкт запиту. Для операцій типу INCOME і EXPENSE передається категорія, а для операції типу TRANSFER передається цільовий рахунок. Це дозволяє використовувати одну форму для різних типів фінансових операцій. Фрагмент формування запиту наведено на рисунку 3.3.

```
private buildCreatePayload(): CreateTransactionRequest {  Show usages
  const rawValue : {type: TransactionType; amount: number; ... } = this.form.getRawValue();

  return {
    type: rawValue.type,
    amount: Number(rawValue.amount),
    date: rawValue.date,
    accountId: rawValue.accountId,
    targetAccountId:
      rawValue.type === 'TRANSFER' ? rawValue.targetAccountId : null,
    categoryId:
      rawValue.type === 'TRANSFER' ? null : rawValue.categoryId,
    comment: rawValue.comment.trim() || null
  };
}
```

Рисунок 3.3 – Формування об’єкта запиту для створення фінансової операції

Для взаємодії з серверною частиною використовується TransactionService. Він інкапсулює HTTP-запити до API та дозволяє компоненту не працювати напряму з адресами серверних маршрутів.

Фрагмент сервісу для створення фінансової операції наведено на рисунку 3.4.

```
createTransaction(payload: CreateTransactionRequest) {  Show usages
  return this.http.post<ApiResponse<{ transaction: Transaction }>>(
    '/api/transactions',
    payload
  );
}
```

Рисунок 3.4 – Метод сервісу для створення фінансової операції

Після успішного створення операції клієнтська частина оновлює список фінансових операцій і рахунків. Це потрібно для того, щоб користувач одразу

бачив нову операцію та актуальний баланс рахунку. Фрагмент методу створення операції наведено на рисунку 3.5.

```
createTransaction(): void {  Show usages
  const payload = this.buildCreatePayload();

  this.transactionService.createTransaction(payload).subscribe({
    next: () => {
      this.successMessage.set('Операцію успішно створено.');
      this.resetForm();
      this.isSaving.set(false);
      this.loadTransactions(1);
      this.loadAccounts();
    },
    error: (error) => {
      this.errorMessage.set(
        error.error?.message || 'Не вдалося створити операцію'
      );
      this.isSaving.set(false);
    }
  });
}
```

Рисунок 3.5 – Метод створення фінансової операції на клієнтській частині

Отже, клієнтська частина забезпечує введення даних, первинну перевірку, формування запиту та оновлення інтерфейсу після створення фінансової операції. Основна бізнес-логіка зміни балансу рахунків виконується на серверній частині, що дозволяє зберігати цілісність фінансових даних і не покладати критичні обчислення лише на клієнтський інтерфейс.

3.4 Реалізація серверної частини

Серверна частина вебдодатку реалізована за допомогою Node.js та Express.js. Вона відповідає за обробку HTTP-запитів, авторизацію користувача, валідацію вхідних даних, взаємодію з базою даних та виконання основної бізнес-логіки системи.

Для роботи з фінансовими операціями використовується окремий набір маршрутів, контролерів і сервісів. Такий підхід дозволяє розділити

відповідальність між частинами серверного коду: маршрути приймають запити, middleware перевіряють авторизацію та валідність даних, контролер передає запит до сервісу, а сервіс виконує основну логіку створення операції. Фрагмент маршруту для створення фінансової операції наведено на рисунку 3.6.

```
router.post( path: '/', validate(createTransactionSchema), createTransaction);
```

Рисунок 3.6 – Маршрут створення фінансової операції

Перед виконанням контролера сервер перевіряє тіло запиту за допомогою middleware validate. Це дозволяє не передавати до бізнес-логіки некоректні або неповні дані. Фрагмент middleware для валідації наведено на рисунку 3.7.

```
export function validate(schema) {  Show usages
    return (req, res, next) : any | undefined  => {
        const result = schema.safeParse({
            body: req.body ?? {},
            params: req.params ?? {},
            query: req.query ?? {}
        });

        if (!result.success) {
            const errors = result.error.issues.map((issue) : {message: any, path: any}  => ({
                path: issue.path.join('.'),
                message: issue.message
            }));

            return res.status(400).json({
                status: 'error',
                message: 'Validation error',
                errors
            });
        }
```

Рисунок 3.7 – Middleware для перевірки вхідних даних

Контролер transaction.controller.js приймає запит від маршруту та передає дані до сервісного шару. Він не містить складної бізнес-логіки, а лише викликає відповідний метод сервісу та формує відповідь клієнту. Фрагмент контролера створення операції наведено на рисунку 3.8.

```
export const createTransaction = asyncHandler( controller: async (req, res) : Promise<void> => { Show usages
    const transaction = await createUserTransaction( {userId, data}: {
        userId: req.user.id,
        data: req.validated.body
    });

    res.status(201).json({
        status: 'success',
        message: 'Transaction has been created successfully',
        data: {
            transaction
        }
    });
});
```

Рисунок 3.8 – Контролер створення фінансової операції

Основна логіка створення фінансової операції знаходиться у сервісі transaction.service.js. У ньому система перевіряє, чи належать рахунки та категорії поточному користувачу, створює запис операції та змінює баланс відповідного рахунку або рахунків. Фрагмент функції створення фінансової операції наведено на рисунку 3.9.

```
export async function createUserTransaction({ userId, data }) : Promise<any> { Show usages
    return prisma.$transaction(async (tx) : Promise<...> => {
        const preparedData : ... | ... = await validateTransactionData(tx, {userId, data}: {
            userId,
            data
        });

        const transaction = await tx.transaction.create({
            data: {
                type: preparedData.type,
                amount: toDecimal(preparedData.amount),
                date: preparedData.date,
                comment: preparedData.comment?.trim() || null,

                userId,

                accountId: preparedData.accountId,
                targetAccountId: preparedData.targetAccountId || null,
```

Рисунок 3.9 – Фрагмент функції створення фінансової операції

У наданому фрагменті використовується транзакція Prisma. Вона відповідає за те, щоб створення фінансової операції та оновлення балансу рахунку виконувалися як єдина цілісна дія. Якщо одна з операцій завершиться помилкою, зміни не повинні бути частково збережені в базі даних.

Після створення операції викликається функція applyTransactionEffect, яка змінює баланс рахунку залежно від типу операції. Фрагмент оновлення балансу наведено на рисунку 3.10.

```
async function applyTransactionEffect(tx, transaction) : Promise<void>  {  Show usages
    const amount = toDecimal(transaction.amount);

    if (transaction.type === 'INCOME') {
        await tx.account.update({
            where: {
                id: transaction.accountId
            },
            data: {
                currentBalance: {
                    increment: amount
                }
            }
        });

        return;
    }

    if (transaction.type === 'EXPENSE') {
        await tx.account.update({
            where: {
                id: transaction.accountId
            },
            data: {
                currentBalance: {
                    decrement: amount
                }
            }
        });

        return;
    }
```

Рисунок 3.10 – Зміна балансу рахунків залежно від типу операції

Наведений фрагмент демонструє, що для доходу баланс основного рахунку збільшується, для витрати – зменшується, а для переказу змінюються два рахунки: з одного кошти списуються, а на інший зараховуються.

Також присутня перевірка належності даних користувачу. Користувач не повинен мати можливості створити операцію для чужого рахунку або використати чужу категорію. Тому перед створенням операції сервер перевіряє userId у рахунках і категоріях. Фрагмент перевірки рахунку користувача наведено на рисунку 3.11.

```
export async function getUserAccounts(userId) : Promise<any> { Show usages
    const accounts = await prisma.account.findMany({
        where: {
            userId
        },
        orderBy: {
            createdAt: 'desc'
        }
    });

    return accounts.map(formatAccount);
}
```

Рисунок 3.11 – Перевірка належності рахунку користувачу

Таким чином, серверна частина забезпечує безпечне створення фінансової операції, перевірку прав доступу, валідацію даних і коректне оновлення балансів рахунків. Основні фінансові обчислення виконуються на сервері, що дозволяє зберігати цілісність даних і не залежати від клієнтської частини під час зміни балансу.

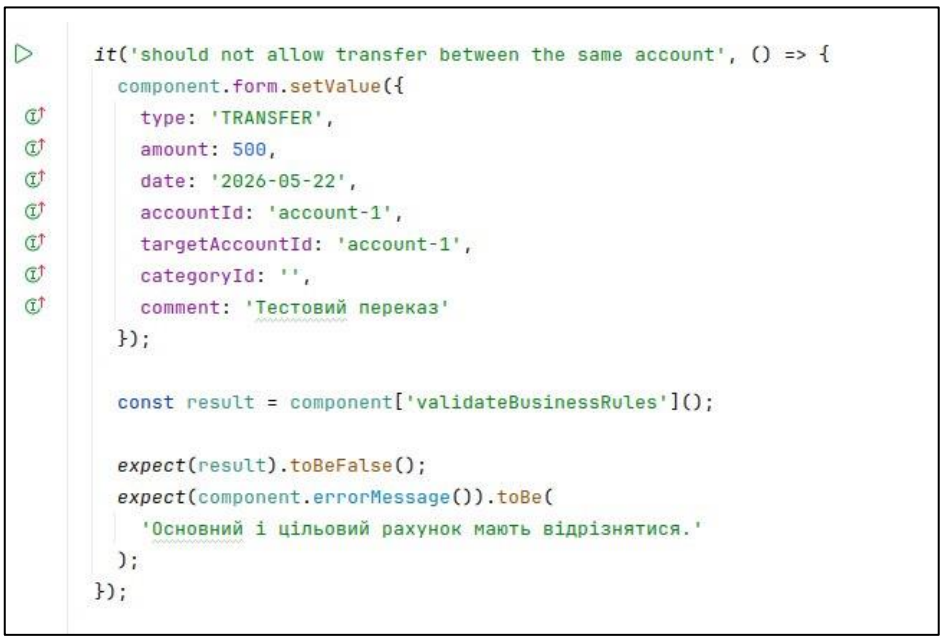
3.5 Тестування проєкту

У процесі розробки системи було проведено unit-тестування для перевірки коректності основної бізнес-логіки, роботи форм та взаємодії клієнтської частини з API. Оскільки головним бізнес-процесом системи є створення фінансової операції, тестування було зосереджено на компоненті TransactionsComponent.

Для тестування використовувалися засоби Angular CLI, а саме вбудовані бібліотеки Jasmine та Karma [1].

Тестування мало дві основні цілі: по-перше, перевірити коректність роботи форми створення фінансової операції, а по-друге – перевірити, чи правильно компонент реагує на невалідні бізнес-ситуації. Наприклад, для операції типу TRANSFER користувач не повинен мати можливості обрати однаковий основний і цільовий рахунок.

На рисунку 3.12 наведено приклад unit-тесту компонента TransactionsComponent, який перевіряє бізнес-правило для переказу між рахунками.



```
it('should not allow transfer between the same account', () => {
  component.form.setValue({
    type: 'TRANSFER',
    amount: 500,
    date: '2026-05-22',
    accountId: 'account-1',
    targetAccountId: 'account-1',
    categoryId: '',
    comment: 'Тестовий переказ'
  });

  const result = component['validateBusinessRules']();

  expect(result).toBeFalse();
  expect(component.errorMessage()).toBe(
    'Основний і цільовий рахунок мають відрізнятися.'
  );
});
```

Рисунок 3.12 – Фрагмент unit-тест компонента TransactionsComponent

У наведеному тесті продемонстровано ситуацію, коли користувач намагається виконати переказ між одним і тим самим рахунком. Метод validateBusinessRules() повинен повернути false, а компонент має встановити повідомлення про помилку. Подібні перевірки є критично важливими, бо не дозволяють створити некоректну фінансову операцію ще на клієнтській частині.

Тести запускаються за допомогою консольної команди ng test або npm run test [1].

Після запуску Angular CLI виконує тести у тестовому середовищі. У випадку використання Karma результати відображаються у браузері та в консолі [1]. Приклад успішного результату тестування наведено на рисунку 3.13.

Проведене тестування дозволяє перевірити коректність важливих бізнес-правил і зменшити ризик появи помилок під час створення фінансових операцій. Зокрема, unit-тест підтверджує, що система не дозволяє виконати переказ між однаковими рахунками, що забезпечує логічну цілісність фінансових даних.

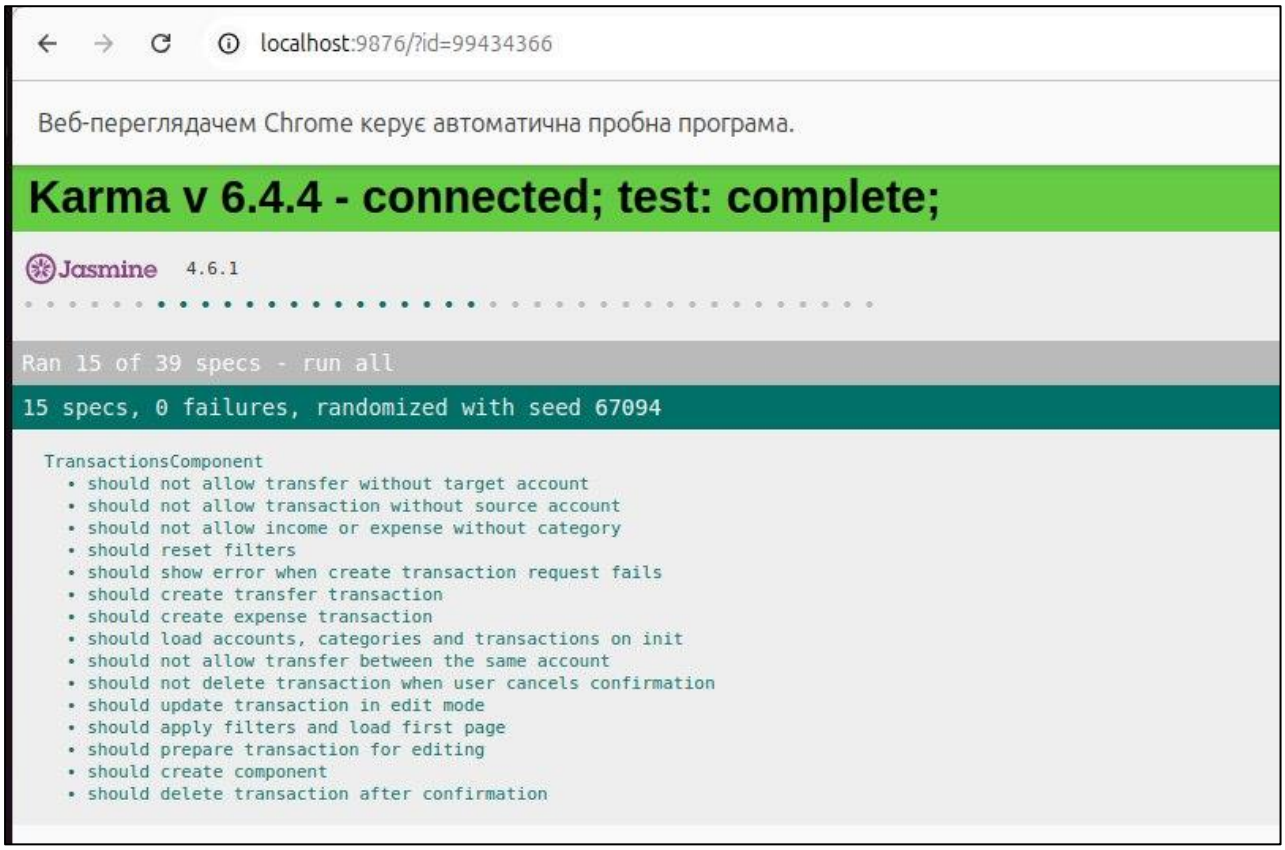


Рисунок 3.13 – Результат виконання unit-тесту TransactionsComponent

3.6 Керівництво користувача

3.6.1 Вхід та реєстрація

Для початку роботи з вебдодатком користувачу необхідно перейти за адресою системи у браузері. Після завантаження сторінки автоматично відкривається форма входу до системи (див. рис. 3.14).

Якщо користувач ще не має облікового запису, необхідно натиснути на посилання «Створити акаунт». Після цього система відкриває форму реєстрації (див. рис. 3.15).

У формі реєстрації користувач вводить ім’я, email та пароль. Після заповнення полів необхідно натиснути кнопку «Зареєструватися». Якщо дані введено коректно, система створює обліковий запис і надсилає лист для підтвердження email (див. рис. 3.16).

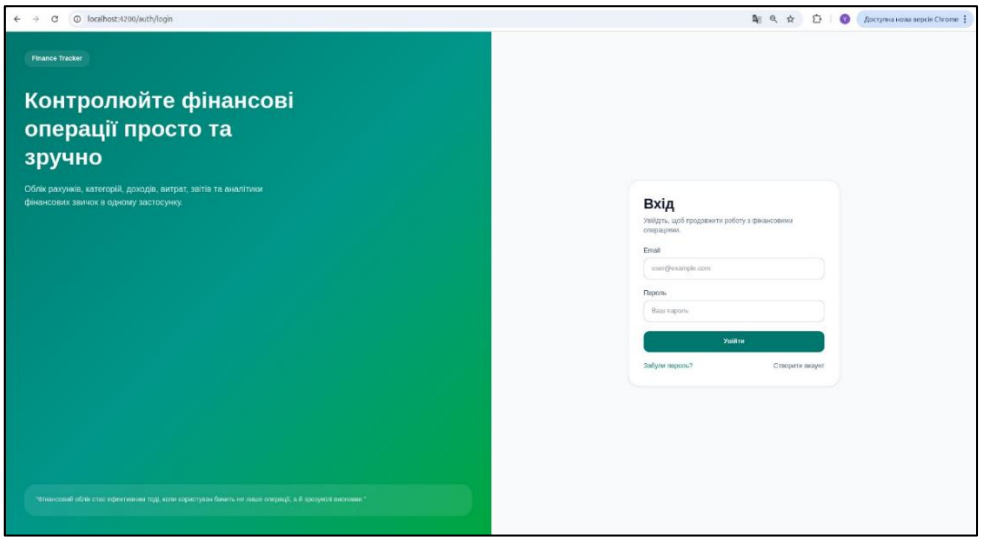


Рисунок 3.14 – Форма входу до системи

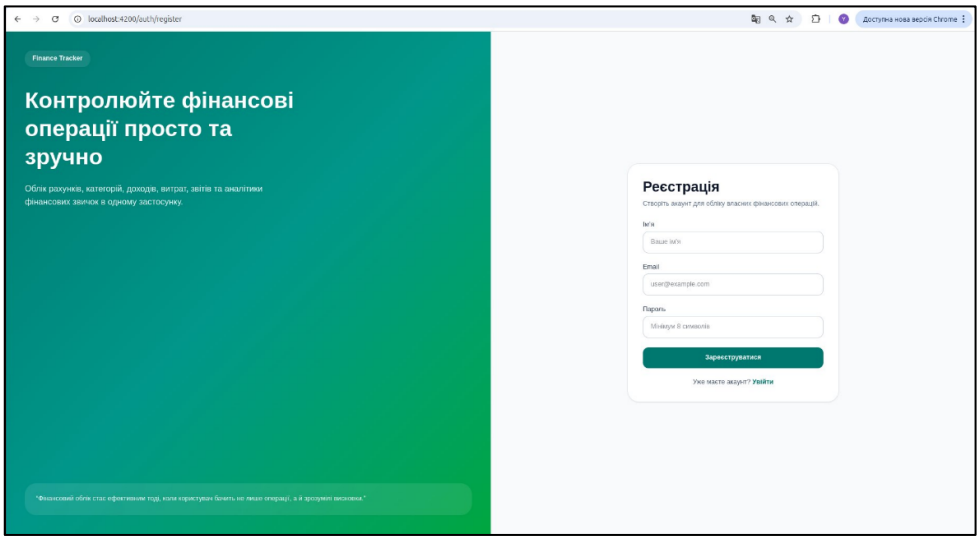


Рисунок 3.15 – Форма реєстрації

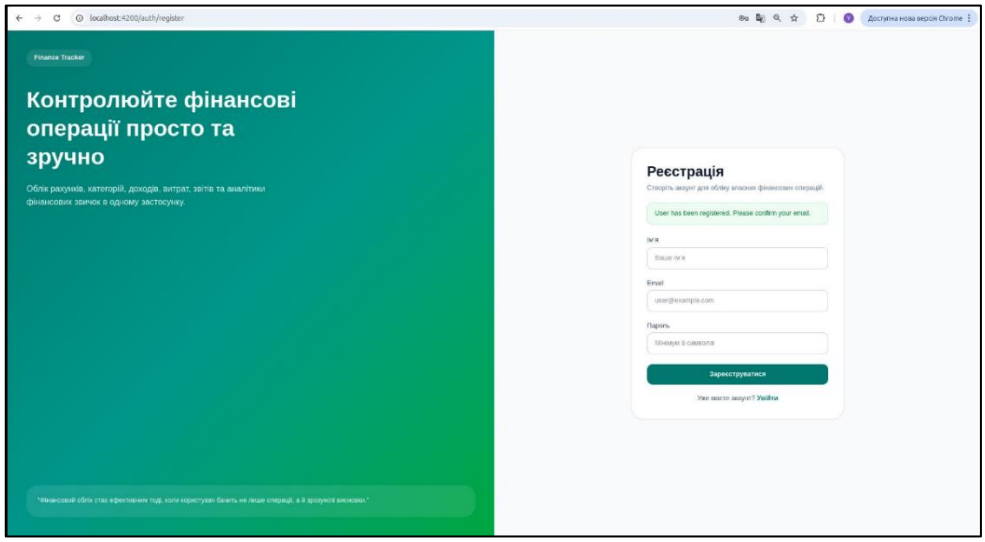


Рисунок 3.16 – Повідомлення про успішну реєстрацію

Якщо користувач вводить некоректні дані або email уже використовується, система відображає відповідне повідомлення про помилку (рис. 3.17).

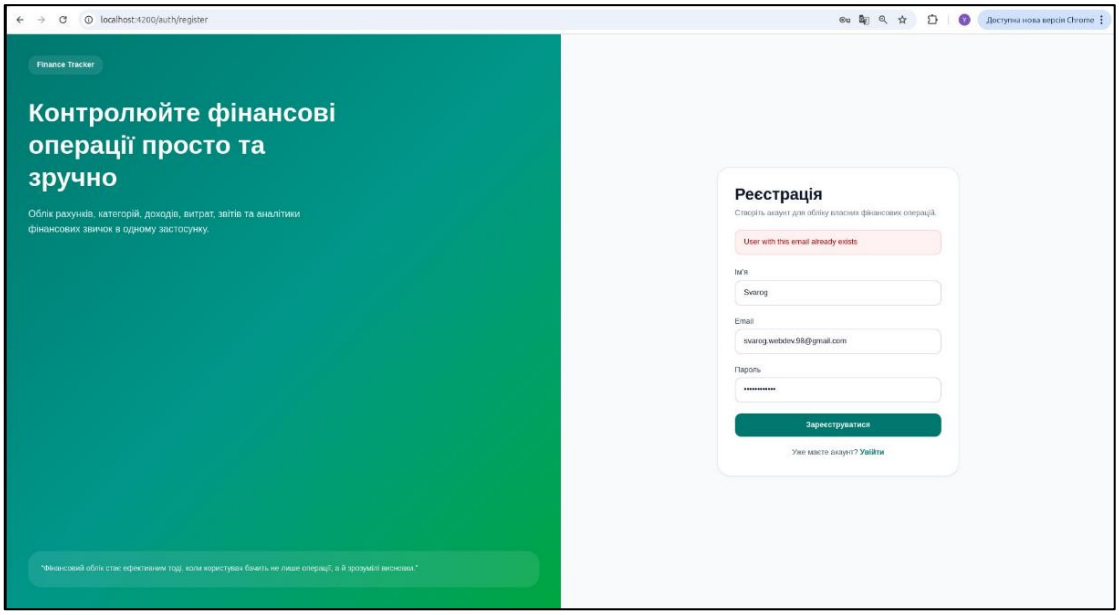


Рисунок 3.17 – Повідомлення про помилку під час реєстрації

Для підтвердження облікового запису користувачу необхідно перейти за посиланням, яке було надіслано на email. Після успішного підтвердження система повідомляє, що email підтверджено, і користувач може перейти до сторінки входу (рис. 3.18).

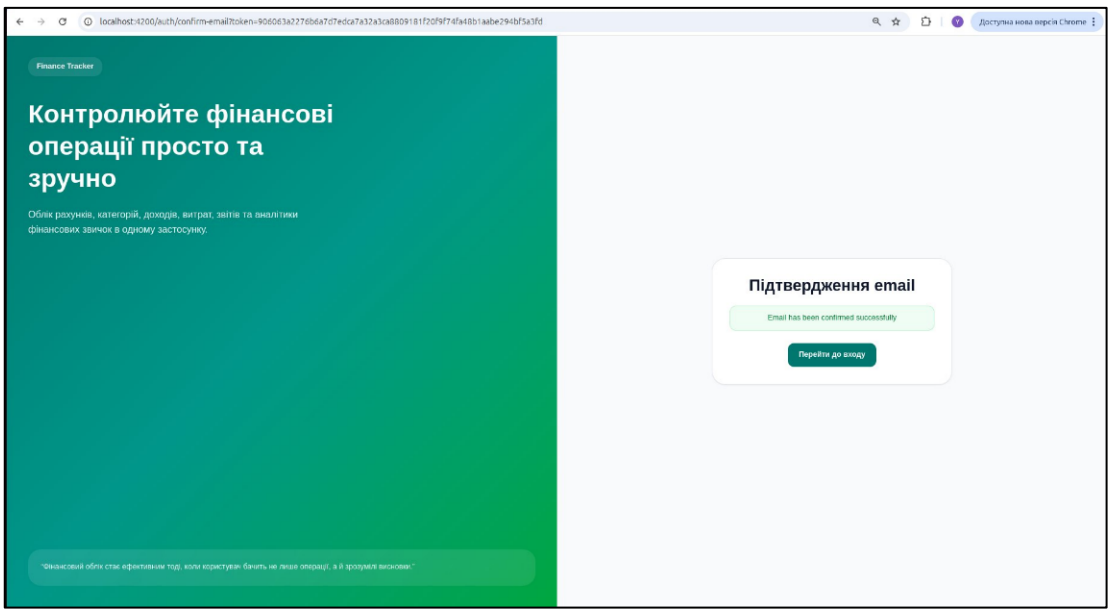


Рисунок 3.18 – Підтвердження email

Для входу до системи користувач вводить email і пароль, вказані під час реєстрації, та натискає кнопку «Увійти». Якщо дані введено правильно, система відкриває головну сторінку вебдодатку (рис. 3.19).

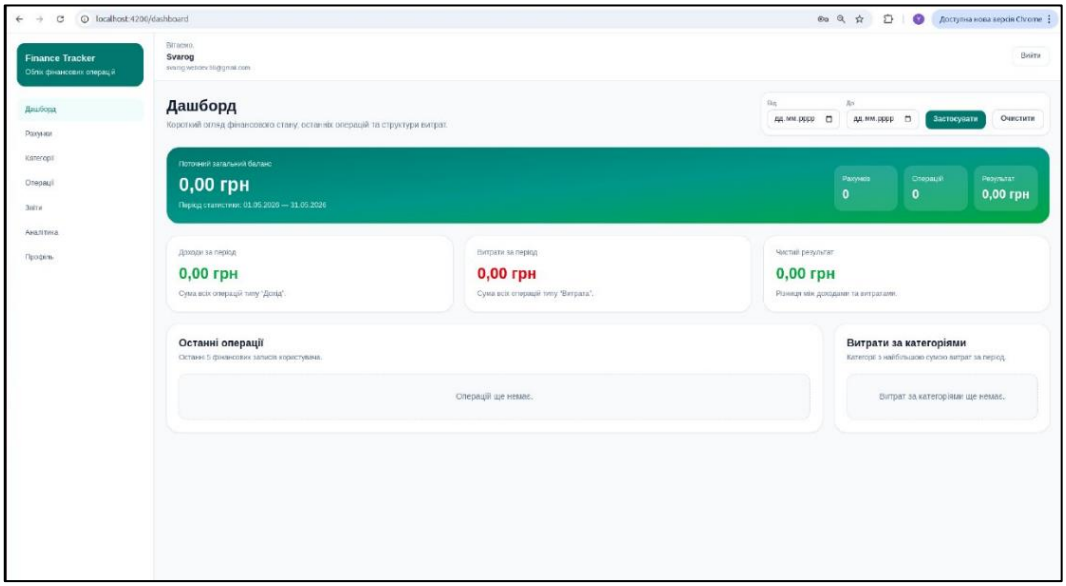


Рисунок 3.19 – Головна сторінка після входу

Якщо користувач забув пароль, він може перейти на сторінку відновлення пароля. Для цього необхідно натиснути «Забули пароль?», ввести email і надіслати запит. Система надішле лист із посиланням для встановлення нового пароля (рис. 3.20).

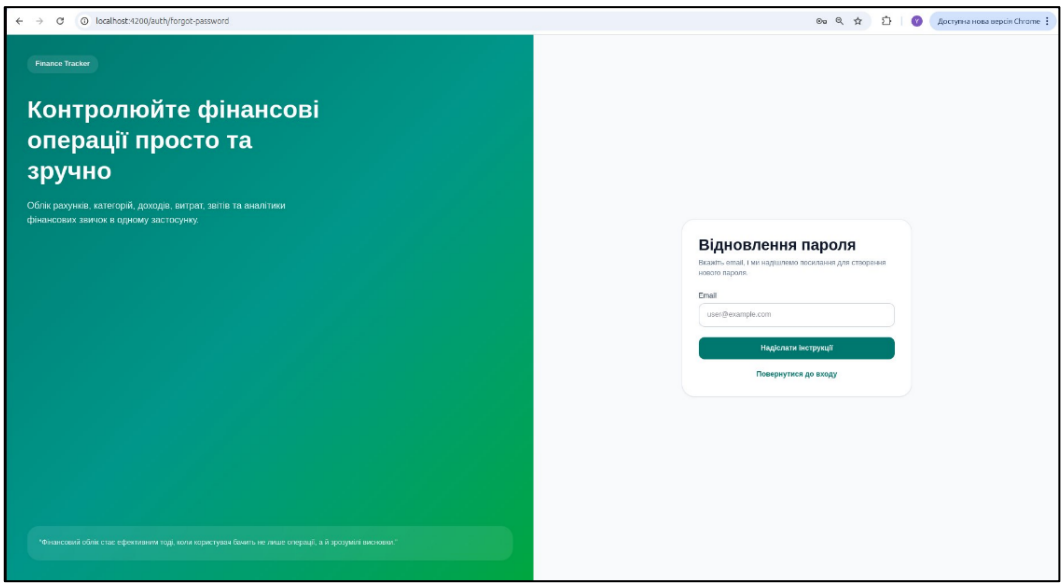


Рисунок 3.20 – Форма відновлення пароля

3.6.2 Робота з рахунками

Після входу до системи користувач може перейти до розділу «Рахунки» через бокове меню. На цій сторінці відображається список створених рахунків, їхній тип, валюта, початковий та поточний баланс (рис. 3.21).

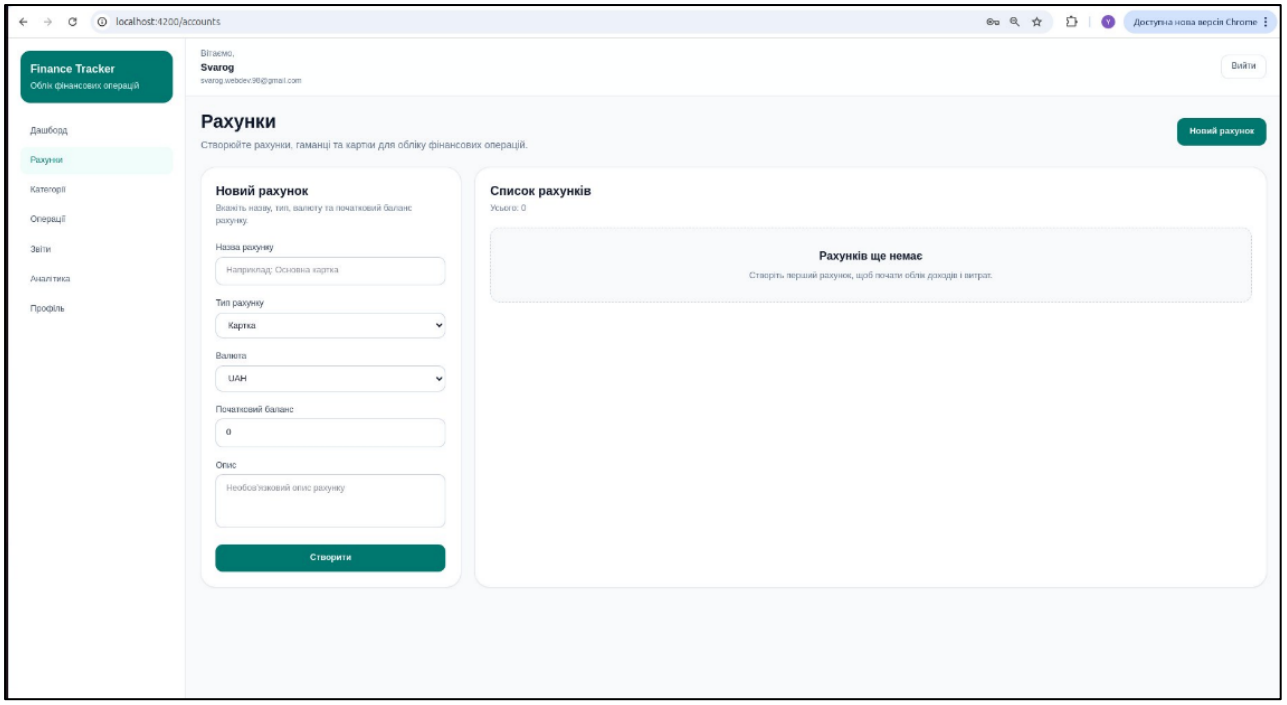


Рисунок 3.21 – Сторінка рахунків

Для створення нового рахунку користувач заповнює форму, вказуючи назву рахунку, тип, валюту, початковий баланс і, за потреби, опис. Після натискання кнопки «Створити» рахунок додається до списку (рис. 3.22).

Для редагування рахунку необхідно натиснути кнопку «Змінити» біля відповідного запису. Дані рахунку будуть підставлені у форму, після чого користувач може змінити назву, тип, валюту або опис і натиснути кнопку «Оновити» (рис. 3.23).

Для видалення рахунку користувач натискає кнопку «Видалити». Якщо рахунок не використовується у фінансових операціях, система видаляє його зі списку. Якщо рахунок уже пов’язаний з операціями, система блокує видалення та показує повідомлення про помилку.

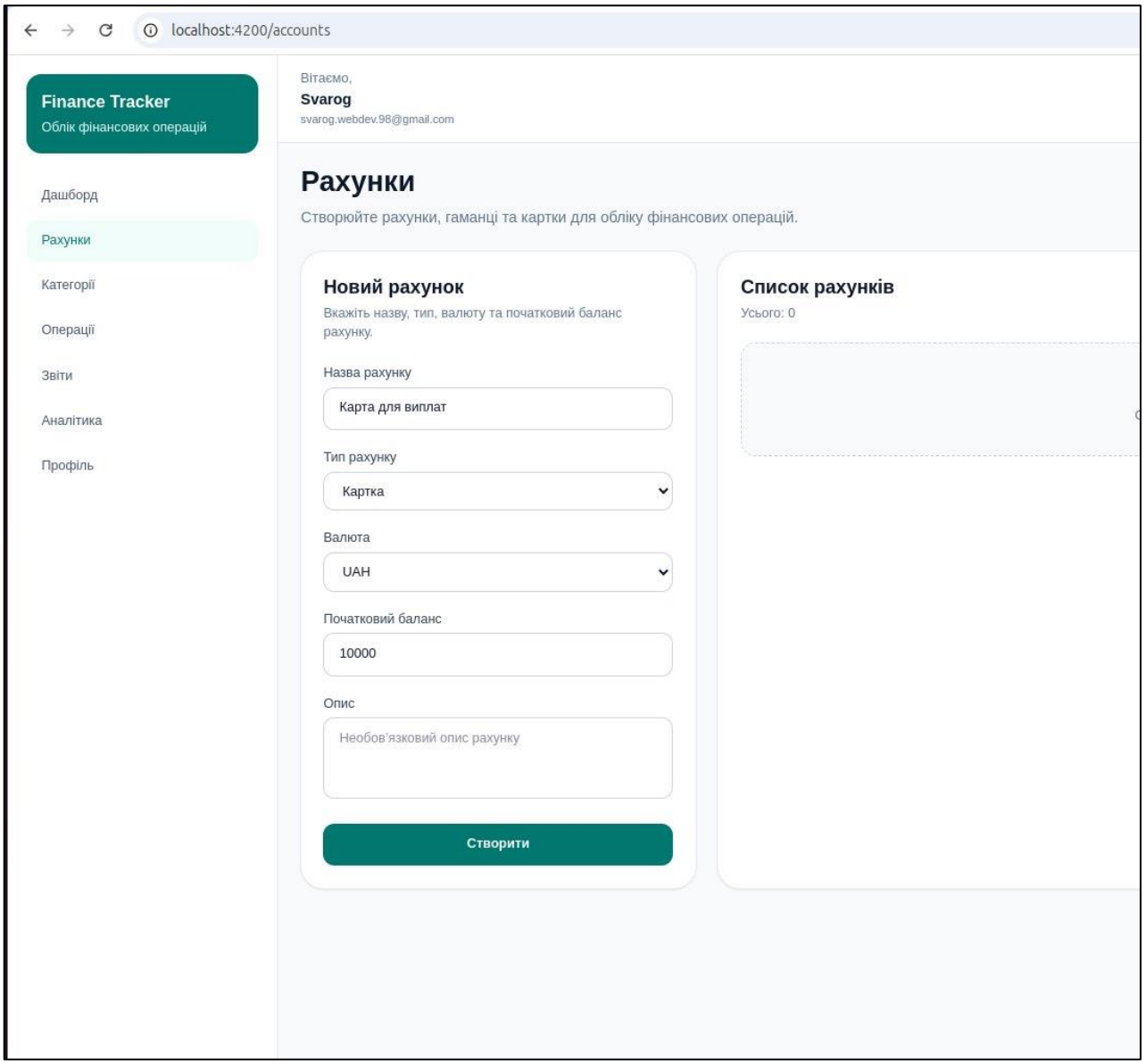


Рисунок 3.22 – Форма створення рахунку

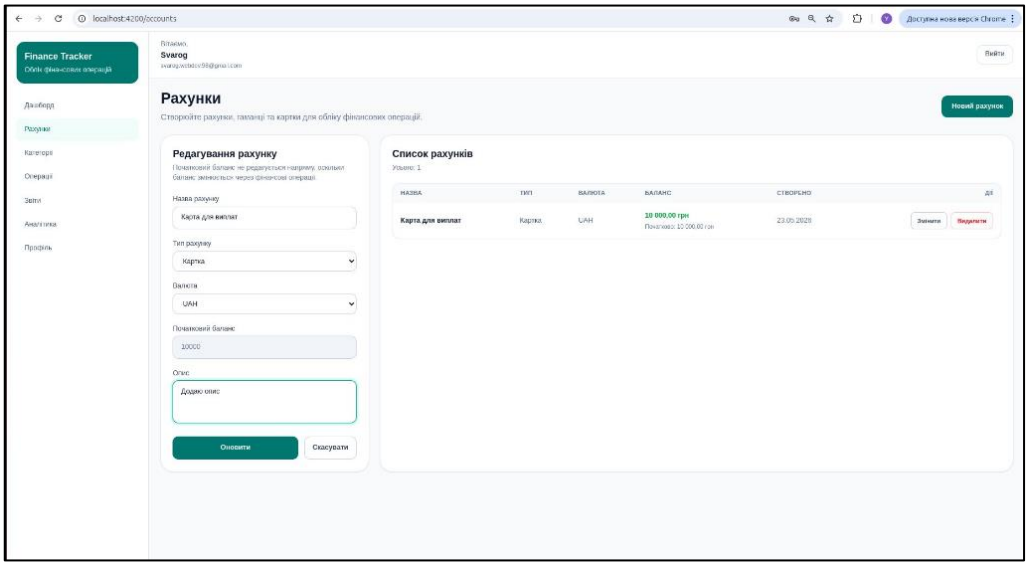


Рисунок 3.23 – Редагування рахунку

3.6.3 Робота з категоріями

Для керування категоріями користувач переходить до розділу «Категорії». На сторінці відображаються категорії доходів і витрат, їхній тип, колір та умовне позначення іконки (рис. 3.24).

Для створення нової категорії користувач вводить її назву, обирає тип («Дохід»/«Витрата»), задає колір та, за потреби, іконку. Після натискання кнопки «Створити» категорія додається до списку (рис. 3.25).

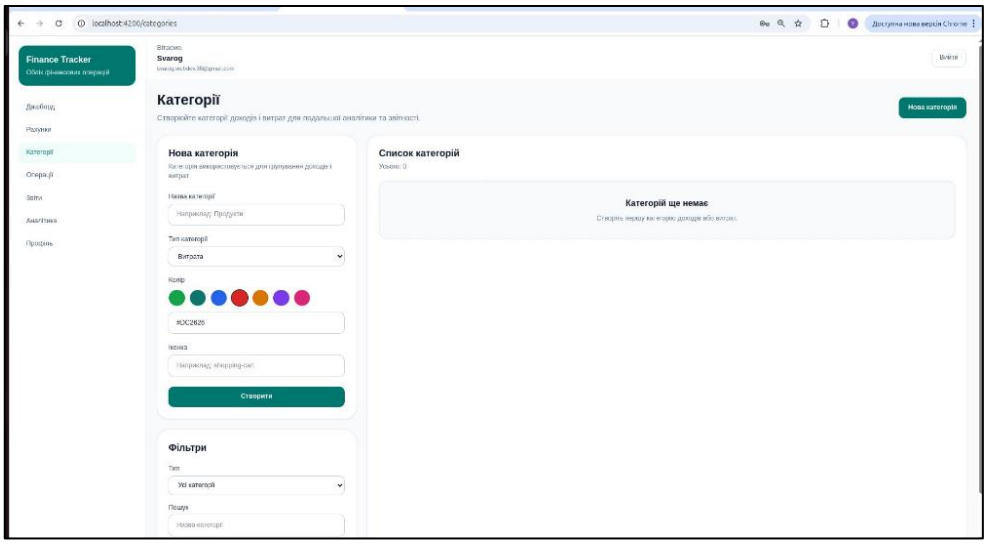


Рисунок 3.24 – Сторінка категорій

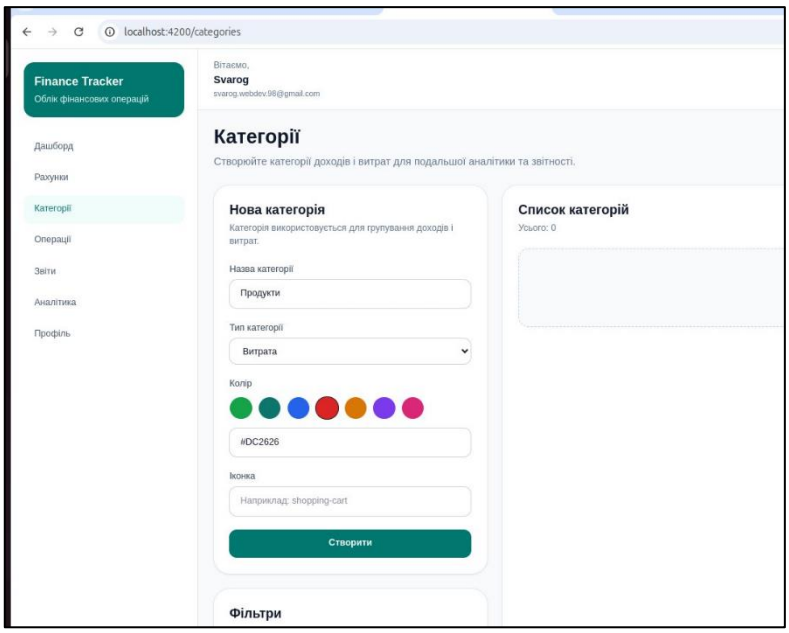


Рисунок 3.25 – Форма створення категорії

На сторінці категорій доступні фільтрація за типом та пошук за назвою. Це дозволяє швидко знайти потрібну категорію серед створених записів (рис. 3.26).

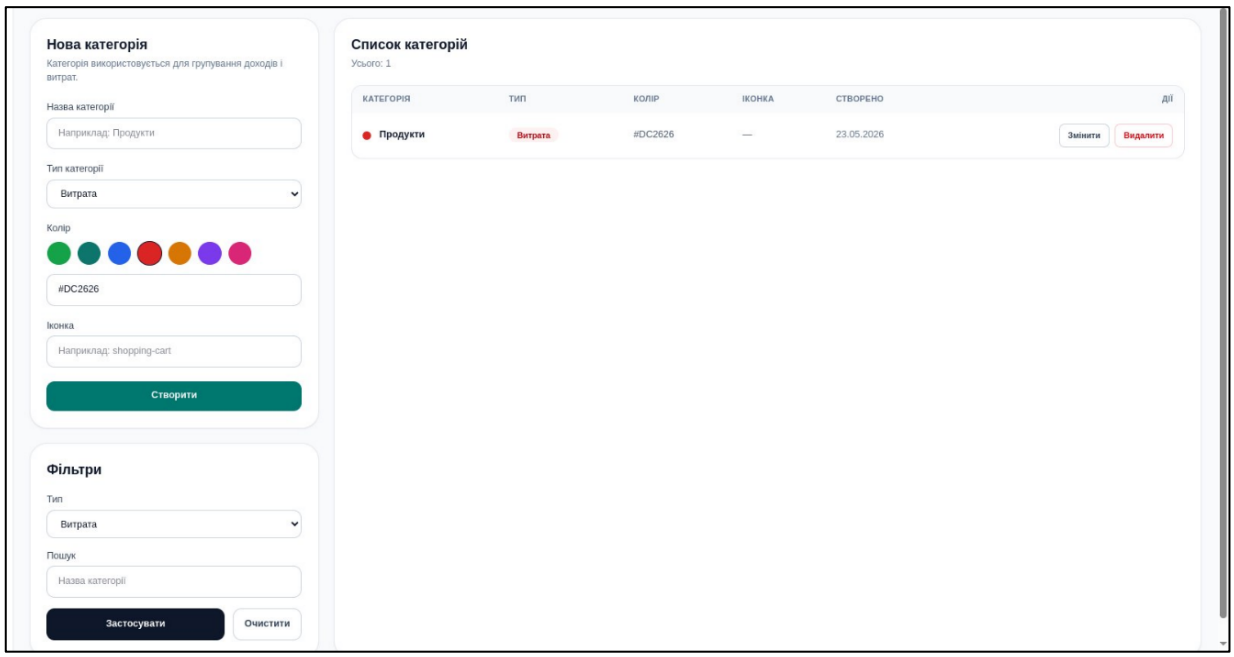


Рисунок 3.26 – Фільтрація та пошук категорій

Для редагування категорії користувач натискає кнопку «Змінити», після чого система підставляє дані категорії у форму. Для видалення категорії використовується кнопка «Видалити». Якщо категорія вже використовується у фінансових операціях, система не дозволяє її видалити.

3.6.4 Робота з фінансовими операціями

Основним розділом системи є «Операції». У ньому користувач може створювати доходи, витрати та перекази між рахунками. На сторінці відображається форма створення операції, фільтри та таблиця вже створених фінансових операцій (рис. 3.27).

Для створення операції користувач обирає її тип: «Дохід», «Витрата» або «Переказ». Далі необхідно вказати суму, дату, основний рахунок і коментар за потреби.

Якщо створюється дохід або витрата, користувач додатково обирає категорію операції (рис. 3.28).

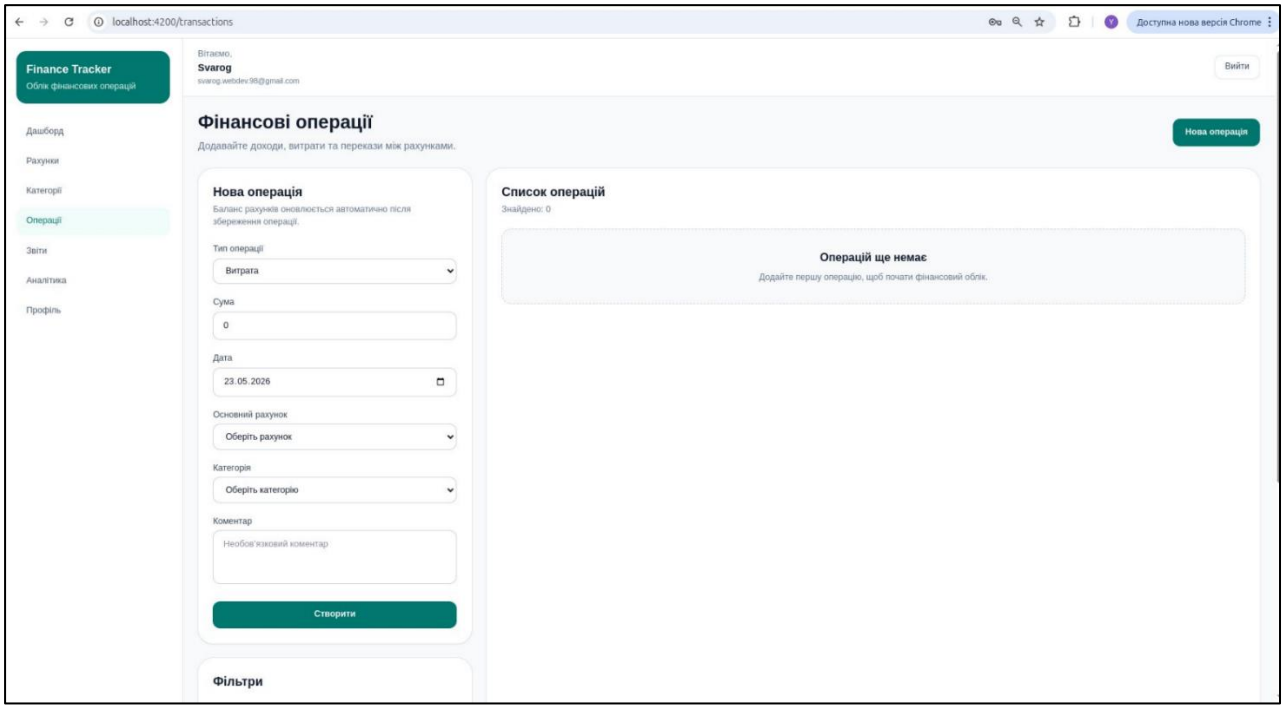


Рисунок 3.27 – Сторінка фінансових операцій

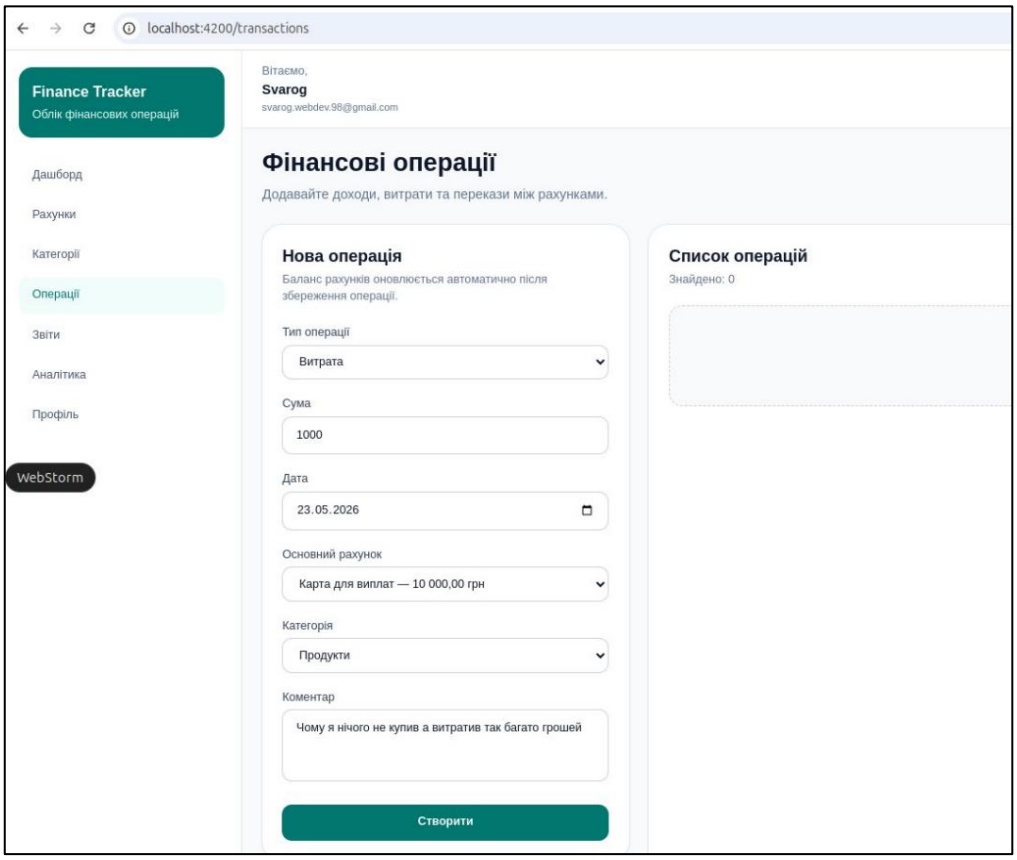


Рисунок 3.28 – Створення доходу або витрати

Якщо створюється переказ, замість категорії користувач обирає цільовий рахунок, на який будуть переказані кошти (рис. 3.29). Після натискання кнопки «Створити» система перевіряє введені дані. Якщо всі поля заповнені правильно, операція додається до списку, а баланс рахунку або рахунків автоматично оновлюється. Для доходу баланс збільшується, для витрати – зменшується, а для переказу кошти списуються з одного рахунку та зараховуються на інший.

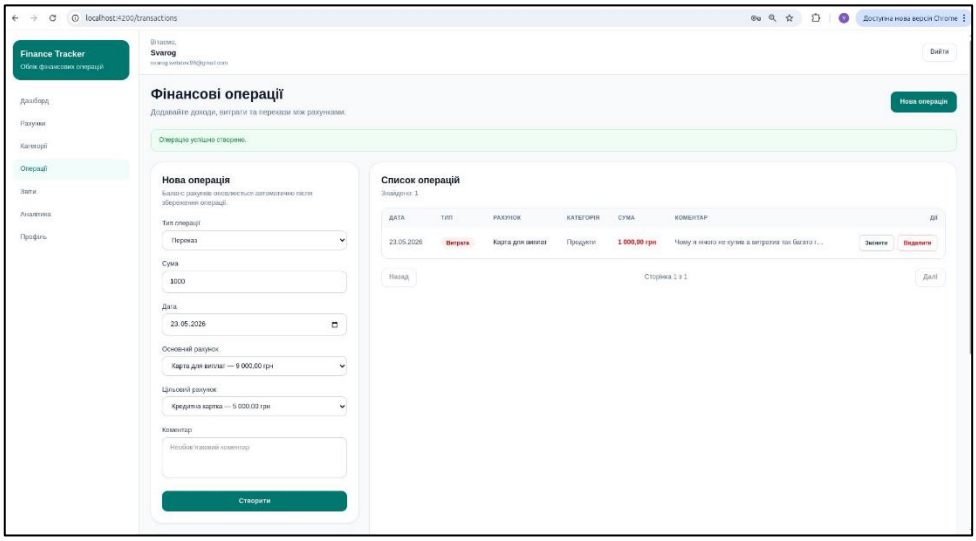


Рисунок 3.29 – Створення переказу між рахунками

На сторінці фінансових операцій також доступні фільтри за типом операції, рахунком, категорією, датами та коментарем (рис. 3.30).

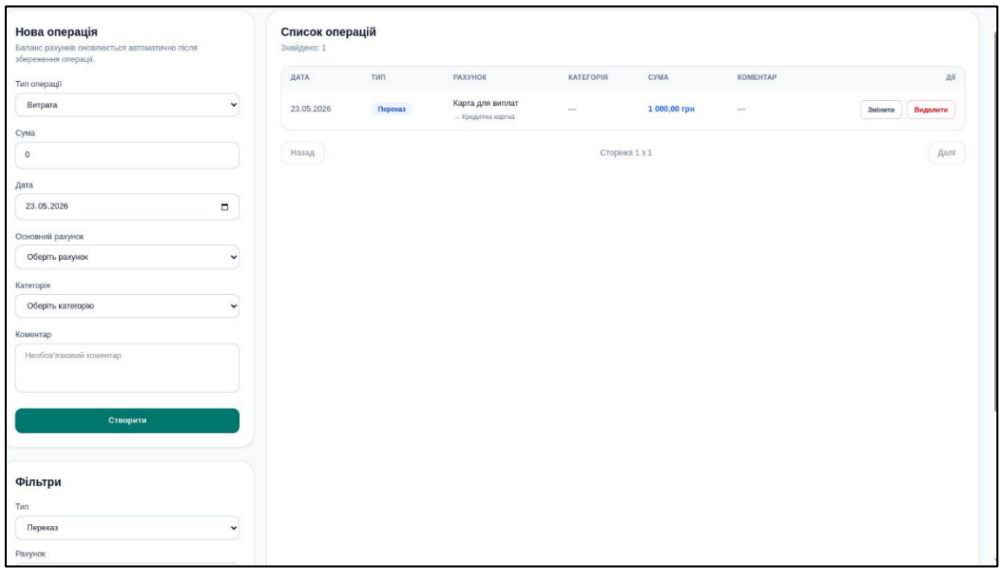


Рисунок 3.30 – Фільтрація фінансових операцій

Для редагування операції користувач натискає кнопку «Змінити» біля потрібного запису. Після внесення змін система перераховує баланс рахунків. Для видалення операції використовується кнопка «Видалити». Після видалення система також оновлює баланс відповідного рахунку.

3.6.5 Перегляд дашборду

Після входу до системи користувач потрапляє на сторінку «Дашборд». На ній відображається коротке фінансове зведення: загальний баланс, доходи, витрати, чистий результат, кількість рахунків і операцій (рис. 3.31).

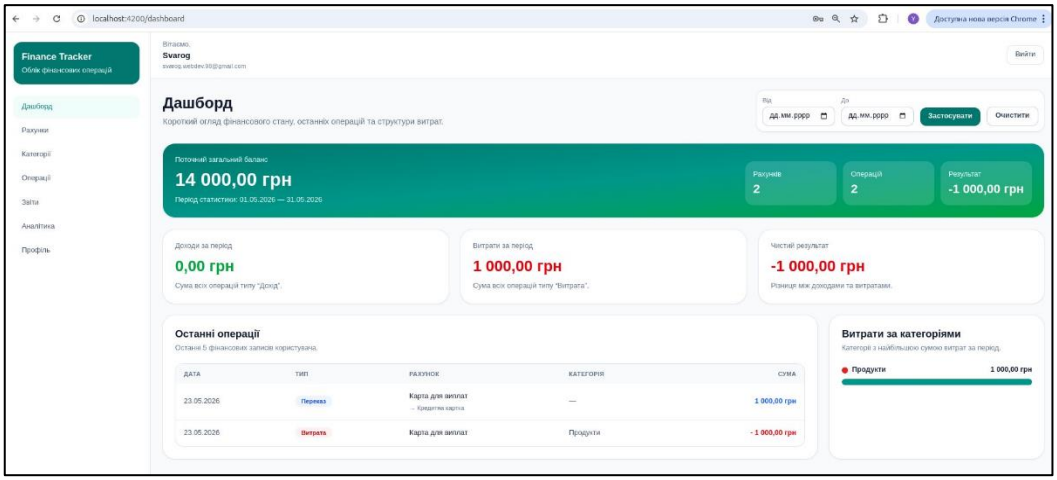


Рисунок 3.31 – Дашборд користувача

Також на дашборді відображаються останні фінансові операції та витрати за категоріями. Користувач може обрати період, за який потрібно переглянути статистику, вказавши дати у відповідних полях.

3.6.6 Формування звітів та PDF-експорт

Для перегляду детальнішої фінансової інформації користувач переходить до розділу «Звіти». На цій сторінці можна сформувати звіт за обраний період,

застосувати фільтри за типом операції, рахунком, категорією та коментарем (рис. 3.32).

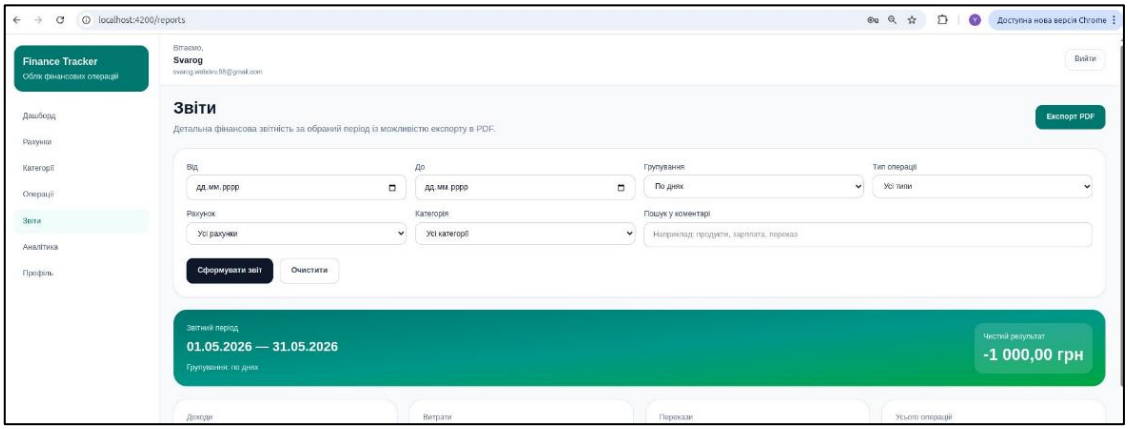


Рисунок 3.32 – Сторінка звітності

Після формування звіту система відображає загальні доходи, витрати, перекази, чистий результат, найбільшу витрату, групування за категоріями та рахунками, а також динаміку доходів і витрат (рис. 3.33).

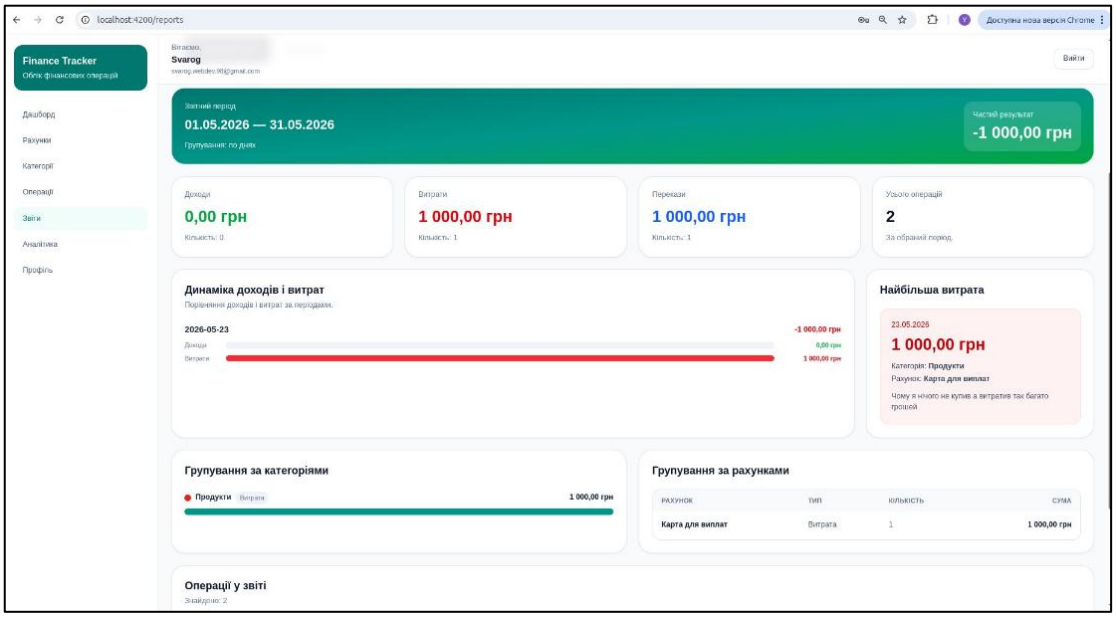


Рисунок 3.33 – Сформований фінансовий звіт

Для збереження звіту користувач натискає кнопку «Експорт PDF». Після цього система формує PDF-файл і завантажує його на пристрій користувача (рис. 3.34).

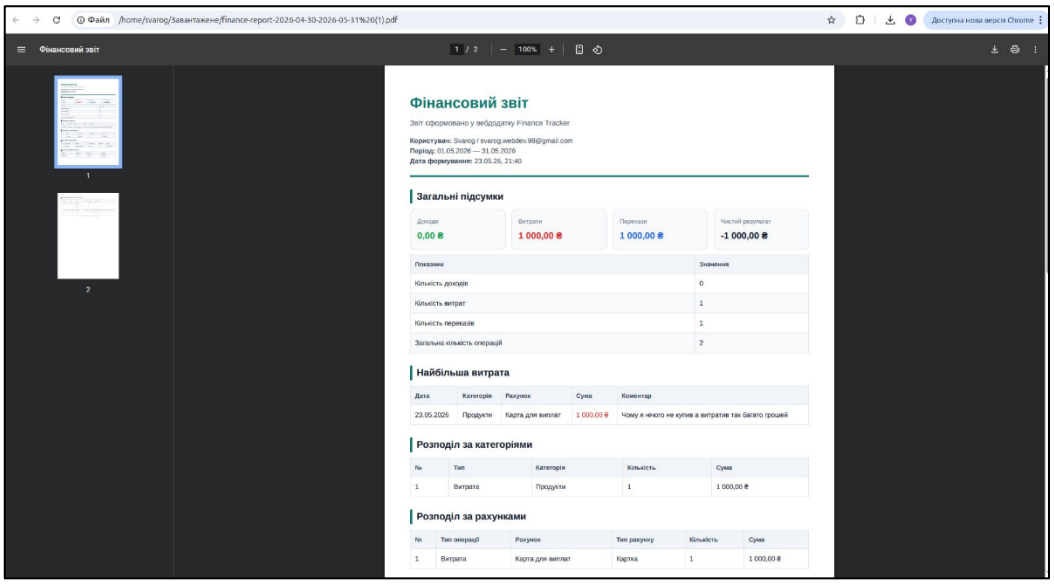


Рисунок 3.34 – Експорт звіту у PDF

3.6.7 Перегляд аналітики

Для аналізу фінансових звичок користувач переходить до розділу «Аналітика». На цій сторінці система відображає доходи, витрати, чистий результат, середню щоденну витрату, частку витрат від доходів і зміну витрат порівняно з попереднім періодом (рис. 3.35).

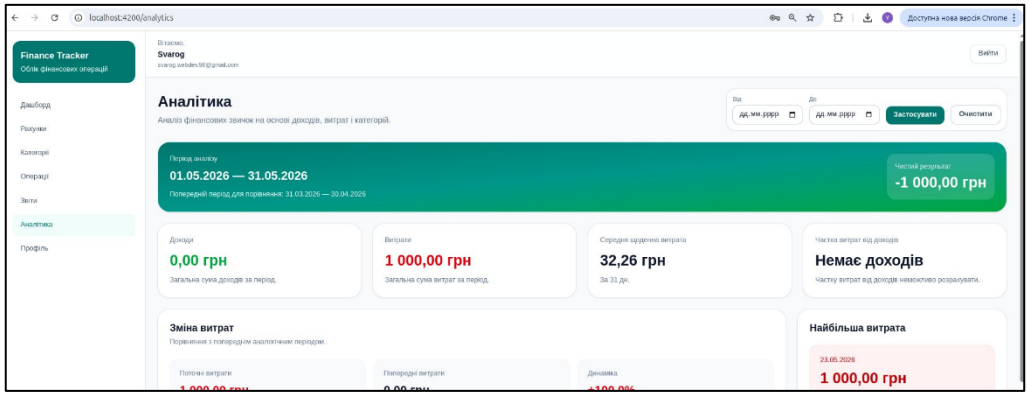


Рисунок 3.35 – Сторінка аналітики

Також система показує найбільшу категорію витрат, найбільшу разову витрату та розподіл витрат за категоріями. На основі цих даних формуються короткі висновки й рекомендації для користувача (рис. 3.36).

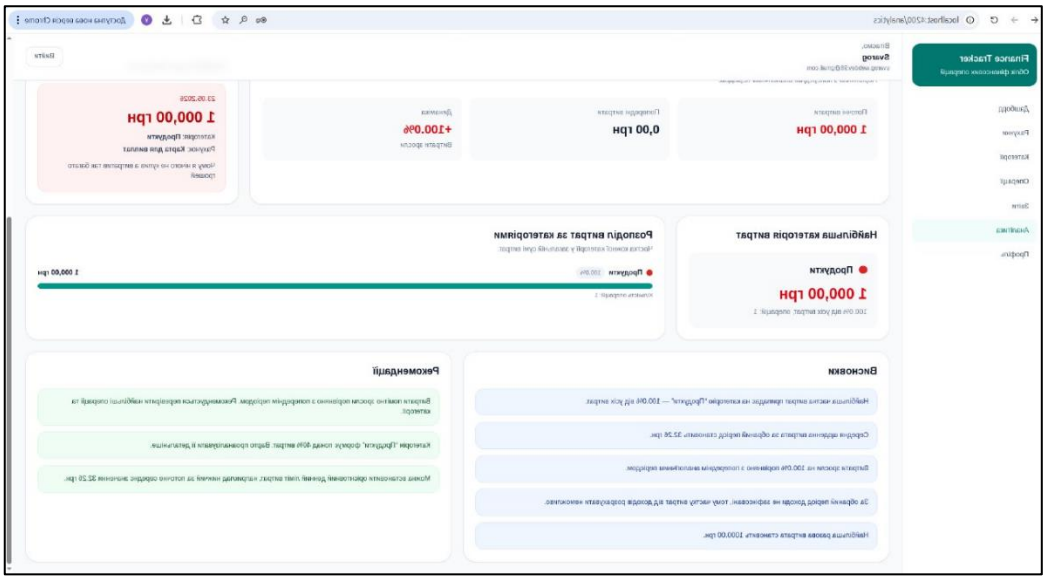


Рисунок 3.36 – Висновки та рекомендації аналітики

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено вебдодаток для обліку фінансових операцій. Система дозволяє користувачу створювати обліковий запис, підтверджувати email, входити до системи, створювати рахунки, категорії, доходи, витрати та перекази між рахунками. Також реалізовано перегляд дашборду, формування фінансових звітів, експорт звітів у PDF та аналітику фінансових звичок.

Відповідно до мети кваліфікаційної роботи реалізовано вебдодаток із використанням таких технологій:

- Angular та Tailwind CSS для клієнтської частини;
- Node.js та Express.js для серверної частини й REST API;
- PostgreSQL для збереження даних;
- Prisma ORM для взаємодії з базою даних;
- JWT для захисту API-маршрутів;
- Puppeteer та Handlebars для формування PDF-звітів.

За поставленими задачами виконано такі етапи:

- сформовано функціональні та нефункціональні вимоги до системи;
- проведено аналіз предметної області персонального фінансового обліку;
- розглянуто аналоги та виконано їх порівняння;
- спроєктовано базу даних і побудовано ER-модель;
- побудовано UML-діаграми варіантів використання, діяльності, класів, послідовності та розгортання;
- реалізовано вебдодаток для обліку фінансових операцій;
- виконано unit-тестування одного з ключових бізнес-правил;
- підготовлено керівництво користувача.

ПЕРЕЛІК ПОСИЛАНЬ

1. Angular Developer Documentation. URL: https://angular.dev/overview (дата звернення: 25.04.2026).

2. BudgetBakers. URL: https://budgetbakers.com/en/ (дата звернення: 07.03.2026).

3. Consumer Financial Protection Bureau. Track your spending with this easy tool. URL: https://www.consumerfinance.gov/about-us/blog/track-your-spending-with-this-easy-tool/ (дата звернення: 16.02.2026).

4. Express.js Guide. URL: https://expressjs.com/ (дата звернення: 15.04.2026).

5. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 2003. 208 p.

6. Handlebars. URL: https://handlebarsjs.com/ (дата звернення: 22.04.2026).

7. ISO/IEC/IEEE 29148:2018. Systems and software engineering – Life cycle processes – Requirements engineering. URL: https://www.iso.org/standard/72089.html (дата звернення: 16.03.2026).

8. Monefy. URL: https://www.monefy.com/ (дата звернення: 07.03.2026).

9. Money Manager. URL: https://www.realbyteapps.com/ (дата звернення: 09.03.2026).

10. Mrzyglocki F. UML Summarized: Key Concepts and Diagrams for Software Engineers, Architects, and Designers. Phillip Mrzyglocki, 2023. 59 p.

11. Node.js Documentation. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 15.04.2026).

12. Npm jsonwebtoken. URL: https://www.npmjs.com/package/jsonwebtoken (дата звернення: 21.04.2026).

13. Financial education. *OECD*. URL: https://www.oecd.org/en/topics/financial-education.html (дата звернення: 16.02.2026).

14. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 19.04.2026).

15. Prisma Docs. URL: https://www.postgresql.org/docs/ (дата звернення: 19.04.2026).

16. Puppeteer Docs. URL: https://pptr.dev/category/introduction (дата звернення: 22.04.2026).

17. Spendee. URL: https://www.spendee.com/ (дата звернення: 09.03.2026).

18. SWEBOK Guide V4.0. Guide to the Software Engineering Body of Knowledge. URL: https://www.computer.org/education/bodies-of-knowledge/software-engineering (дата звернення: 16.03.2026).

19. Tailwind CSS Docs. URL: https://tailwindcss.com/docs/installation/using-vite (дата звернення: 25.04.2026).

ДОДАТОК А

Angular TransactionsComponent

```
import { DatePipe } from '@angular/common';
import { Component, computed, inject, OnInit, signal } from '@angular/core';
import {
  FormControl,
  FormGroup,
  ReactiveFormsModule,
  Validators
} from '@angular/forms';
import { forkJoin } from 'rxjs';

import { AccountService } from '../../core/services/account.service';
import { CategoryService } from '../../core/services/category.service';
import { TransactionService } from '../../core/services/transaction.service';

import { Account, Currency } from '../../core/models/account.model';
import { Category } from '../../core/models/category.model';
import {
  CreateTransactionRequest,
  Transaction,
  TransactionFilters,
  TransactionPagination,
  TransactionType,
  UpdateTransactionRequest
} from '../../core/models/transaction.model';

interface TransactionForm {
  type: FormControl<TransactionType>;
  amount: FormControl<number>;
  date: FormControl<string>;
  accountId: FormControl<string>;
  targetAccountId: FormControl<string>;
  categoryId: FormControl<string>;
  comment: FormControl<string>;
}

interface FilterForm {
```

```
  type: FormControl<TransactionType | ''>;
  accountId: FormControl<string>;
  categoryId: FormControl<string>;
  from: FormControl<string>;
  to: FormControl<string>;
  search: FormControl<string>;
}

@Component({
  selector: 'app-transactions',
  imports: [ReactiveFormsModule, DatePipe],
  templateUrl: './transactions.component.html',
  styleUrl: './transactions.component.css'
})
export class TransactionsComponent implements OnInit {
  private readonly transactionService = inject(TransactionService);
  private readonly accountService = inject(AccountService);
  private readonly categoryService = inject(CategoryService);

  readonly transactions = signal<Transaction[]>([]);
  readonly accounts = signal<Account[]>([]);
  readonly categories = signal<Category[]>([]);

  readonly isLoading = signal(false);
  readonly isSaving = signal(false);

  readonly errorMessage = signal<string | null>(null);
  readonly successMessage = signal<string | null>(null);

  readonly selectedTransactionId = signal<string | null>(null);

  readonly pagination = signal<TransactionPagination>({
    page: 1,
    limit: 10,
    total: 0,
    totalPages: 1
  });

  readonly isEditMode = computed(() => Boolean(this.selectedTransactionId()));

  readonly incomeCategories = computed(() =>
    this.categories().filter((category) => category.type === 'INCOME')
```

```
);

readonly expenseCategories = computed(() =>
  this.categories().filter((category) => category.type === 'EXPENSE')
);

readonly transactionTypes: { value: TransactionType; label: string }[] = [
  {
    value: 'INCOME',
    label: 'Дохід'
  },
  {
    value: 'EXPENSE',
    label: 'Витрата'
  },
  {
    value: 'TRANSFER',
    label: 'Переказ'
  }
];

readonly form = new FormGroup<TransactionForm>({
  type: new FormControl('EXPENSE', {
    nonNullable: true,
    validators: [Validators.required]
  }),

  amount: new FormControl(0, {
    nonNullable: true,
    validators: [Validators.required, Validators.min(0.01)]
  }),

  date: new FormControl(this.getTodayInputValue(), {
    nonNullable: true,
    validators: [Validators.required]
  }),

  accountId: new FormControl('', {
    nonNullable: true,
    validators: [Validators.required]
  }),
```

```
  targetAccountId: new FormControl('', {
    nonNullable: true
  }),

  categoryId: new FormControl('', {
    nonNullable: true
  }),

  comment: new FormControl('', {
    nonNullable: true,
    validators: [Validators.maxLength(500)]
  })
});

readonly filterForm = new FormGroup<FilterForm>({
  type: new FormControl('', {
    nonNullable: true
  }),

  accountId: new FormControl('', {
    nonNullable: true
  }),

  categoryId: new FormControl('', {
    nonNullable: true
  }),

  from: new FormControl('', {
    nonNullable: true
  }),

  to: new FormControl('', {
    nonNullable: true
  }),

  search: new FormControl('', {
    nonNullable: true
  })
});

ngOnInit(): void {
  this.loadInitialData();
```

```
}

loadInitialData(): void {
  this.isLoading.set(true);
  this.errorMessage.set(null);

  forkJoin({
    accountsResponse: this.accountService.getAccounts(),
    categoriesResponse: this.categoryService.getCategories()
  }).subscribe({
    next: ({ accountsResponse, categoriesResponse }) => {
      this.accounts.set(accountsResponse.data?.accounts ?? []);
      this.categories.set(categoriesResponse.data?.categories ?? []);

      this.isLoading.set(false);
      this.loadTransactions();
    },
    error: (error) => {
      this.errorMessage.set(
        error.error?.message || 'Не вдалося завантажити початкові дані'
      );
      this.isLoading.set(false);
    }
  });
}

loadAccounts(): void {
  this.accountService.getAccounts().subscribe({
    next: (response) => {
      this.accounts.set(response.data?.accounts ?? []);
    }
  });
}

loadTransactions(page = this.pagination().page): void {
  this.isLoading.set(true);
  this.errorMessage.set(null);

  const rawFilters = this.filterForm.getRawValue();

  const filters: TransactionFilters = {
    page,
```

```
    limit: this.pagination().limit,
    type: rawFilters.type,
    accountId: rawFilters.accountId || undefined,
    categoryId: rawFilters.categoryId || undefined,
    from: rawFilters.from || undefined,
    to: rawFilters.to || undefined,
    search: rawFilters.search || undefined
   };

   this.transactionService.getTransactions(filters).subscribe({
    next: (response) => {
     this.transactions.set(response.data?.transactions ?? []);

     if (response.data?.pagination) {
      this.pagination.set(response.data.pagination);
     }

     this.isLoading.set(false);
    },
    error: (error) => {
     this.errorMessage.set(
      error.error?.message || 'Не вдалося завантажити операції'
     );
     this.isLoading.set(false);
    }
   });
  }

  applyFilters(): void {
   this.loadTransactions(1);
  }

  resetFilters(): void {
   this.filterForm.reset({
    type: '',
    accountId: '',
    categoryId: '',
    from: '',
    to: '',
    search: ''
   });
```

```
  this.loadTransactions(1);
}

submit(): void {
  if (!this.validateBusinessRules()) {
    return;
  }

  if (this.form.invalid) {
    this.form.markAllAsTouched();
    return;
  }

  this.isSaving.set(true);
  this.errorMessage.set(null);
  this.successMessage.set(null);

  if (this.isEditMode()) {
    this.updateTransaction();
    return;
  }

  this.createTransaction();
}

createTransaction(): void {
  const payload = this.buildCreatePayload();

  this.transactionService.createTransaction(payload).subscribe({
    next: () => {
      this.successMessage.set('Операцію успішно створено.');
      this.resetForm();
      this.isSaving.set(false);
      this.loadTransactions(1);
      this.loadAccounts();
    },
    error: (error) => {
      this.errorMessage.set(
        error.error?.message || 'Не вдалося створити операцію'
      );
      this.isSaving.set(false);
    }
```

```
  });
}

updateTransaction(): void {
  const transactionId = this.selectedTransactionId();

  if (!transactionId) {
    this.isSaving.set(false);
    return;
  }

  const payload = this.buildUpdatePayload();

  this.transactionService.updateTransaction(transactionId, payload).subscribe({
    next: () => {
      this.successMessage.set('Операцію успішно оновлено.');
      this.resetForm();
      this.isSaving.set(false);
      this.loadTransactions();
      this.loadAccounts();
    },
    error: (error) => {
      this.errorMessage.set(
        error.error?.message || 'Не вдалося оновити операцію'
      );
      this.isSaving.set(false);
    }
  });
}

editTransaction(transaction: Transaction): void {
  this.selectedTransactionId.set(transaction.id);
  this.successMessage.set(null);
  this.errorMessage.set(null);

  this.form.patchValue({
    type: transaction.type,
    amount: transaction.amount,
    date: this.toDateInputValue(transaction.date),
    accountId: transaction.accountId,
    targetAccountId: transaction.targetAccountId ?? '',
    categoryId: transaction.categoryId ?? '',
```

```
    comment: transaction.comment ?? ''
  });
}

deleteTransaction(transaction: Transaction): void {
  const confirmed = confirm('Видалити цю фінансову операцію?');

  if (!confirmed) {
    return;
  }

  this.errorMessage.set(null);
  this.successMessage.set(null);

  this.transactionService.deleteTransaction(transaction.id).subscribe({
    next: () => {
      this.successMessage.set('Операцію успішно видалено.');
      this.loadTransactions();
      this.loadAccounts();

      if (this.selectedTransactionId() === transaction.id) {
        this.resetForm();
      }
    },
    error: (error) => {
      this.errorMessage.set(
        error.error?.message || 'Не вдалося видалити операцію'
      );
    }
  });
}

resetForm(): void {
  this.selectedTransactionId.set(null);

  this.form.reset({
    type: 'EXPENSE',
    amount: 0,
    date: this.getTodayInputValue(),
    accountId: '',
    targetAccountId: '',
    categoryId: '',
```

```
    comment: ''
  });
}

onTransactionTypeChange(): void {
  const type = this.form.controls.type.value;

  if (type === 'TRANSFER') {
    this.form.controls.categoryId.setValue('');
    return;
  }

  this.form.controls.targetAccountId.setValue('');
  this.form.controls.categoryId.setValue('');
}

getAvailableCategories(): Category[] {
  const type = this.form.controls.type.value;

  if (type === 'INCOME') {
    return this.incomeCategories();
  }

  if (type === 'EXPENSE') {
    return this.expenseCategories();
  }

  return [];
}

getFilterCategories(): Category[] {
  const type = this.filterForm.controls.type.value;

  if (type === 'INCOME') {
    return this.incomeCategories();
  }

  if (type === 'EXPENSE') {
    return this.expenseCategories();
  }

  return this.categories().filter((category) => category.type !== undefined);
```

```
}

changePage(page: number): void {
  if (page < 1 || page > this.pagination().totalPages) {
    return;
  }

  this.loadTransactions(page);
}

private buildCreatePayload(): CreateTransactionRequest {
  const rawValue = this.form.getRawValue();

  return {
    type: rawValue.type,
    amount: Number(rawValue.amount),
    date: rawValue.date,
    accountId: rawValue.accountId,
    targetAccountId:
      rawValue.type === 'TRANSFER' ? rawValue.targetAccountId : null,
    categoryId:
      rawValue.type === 'TRANSFER' ? null : rawValue.categoryId,
    comment: rawValue.comment.trim() || null
  };
}

private buildUpdatePayload(): UpdateTransactionRequest {
  const rawValue = this.form.getRawValue();

  return {
    type: rawValue.type,
    amount: Number(rawValue.amount),
    date: rawValue.date,
    accountId: rawValue.accountId,
    targetAccountId:
      rawValue.type === 'TRANSFER' ? rawValue.targetAccountId : null,
    categoryId:
      rawValue.type === 'TRANSFER' ? null : rawValue.categoryId,
    comment: rawValue.comment.trim() || null
  };
}
```

```
private validateBusinessRules(): boolean {
  const rawValue = this.form.getRawValue();

  this.errorMessage.set(null);

  if (!rawValue.accountId) {
    this.errorMessage.set('Оберіть основний рахунок.');
    return false;
  }

  if (rawValue.type === 'INCOME' || rawValue.type === 'EXPENSE') {
    if (!rawValue.categoryId) {
      this.errorMessage.set('Для доходу або витрати потрібно обрати категорію.');
      return false;
    }
  }

  if (rawValue.type === 'TRANSFER') {
    if (!rawValue.targetAccountId) {
      this.errorMessage.set('Для переказу потрібно обрати цільовий рахунок.');
      return false;
    }

    if (rawValue.accountId === rawValue.targetAccountId) {
      this.errorMessage.set('Основний і цільовий рахунок мають відрізнятися.');
      return false;
    }
  }

  return true;
}

private getTodayInputValue(): string {
  return new Date().toISOString().slice(0, 10);
}

private toDateInputValue(value: string): string {
  return new Date(value).toISOString().slice(0, 10);
}

getTransactionTypeLabel(type: TransactionType): string {
  return this.transactionTypes.find((item) => item.value === type)?.label ?? type;
```

```
  }

  getCurrency(transaction: Transaction): Currency {
    return transaction.account?.currency ?? 'UAH';
  }

  formatMoney(value: number, currency: Currency = 'UAH'): string {
    return new Intl.NumberFormat('uk-UA', {
      style: 'currency',
      currency
    }).format(value);
  }
}
```

ДОДАТОК Б

Angular TransactionService

```
import { HttpClient, HttpParams } from '@angular/common/http';
import { inject, Injectable } from '@angular/core';

import { ApiResponse } from '../models/api-response.model';
import {
  CreateTransactionRequest,
  Transaction,
  TransactionFilters,
  TransactionPagination,
  UpdateTransactionRequest
} from '../models/transaction.model';

@Injectable({
  providedIn: 'root'
})
export class TransactionService {
  private readonly http = inject(HttpClient);

  getTransactions(filters: TransactionFilters = {}) {
    let params = new HttpParams();

    if (filters.page) {
      params = params.set('page', String(filters.page));
    }

    if (filters.limit) {
      params = params.set('limit', String(filters.limit));
    }

    if (filters.type) {
      params = params.set('type', filters.type);
    }

    if (filters.accountId) {
      params = params.set('accountId', filters.accountId);
    }
```

```
  if (filters.categoryId) {
    params = params.set('categoryId', filters.categoryId);
  }

  if (filters.from) {
    params = params.set('from', filters.from);
  }

  if (filters.to) {
    params = params.set('to', filters.to);
  }

  if (filters.search?.trim()) {
    params = params.set('search', filters.search.trim());
  }

  return this.http.get<
    ApiResponse<{
      transactions: Transaction[];
      pagination: TransactionPagination;
    }>
  >('/api/transactions', { params });
}

getTransactionById(id: string) {
  return this.http.get<ApiResponse<{ transaction: Transaction }>>(
    `/api/transactions/${id}`
  );
}

createTransaction(payload: CreateTransactionRequest) {
  return this.http.post<ApiResponse<{ transaction: Transaction }>>(
    '/api/transactions',
    payload
  );
}

updateTransaction(id: string, payload: UpdateTransactionRequest) {
  return this.http.put<ApiResponse<{ transaction: Transaction }>>(
    `/api/transactions/${id}`,
    payload
  );
```

```
  }

  deleteTransaction(id: string) {
    return this.http.delete<ApiResponse<null>>(`/api/transactions/${id}`);
  }
}
```

ДОДАТОК В

Express transaction.routes.js

```
import { Router } from 'express';

import {
  createTransaction,
  deleteTransaction,
  getTransactionById,
  getTransactions,
  updateTransaction
} from '../controllers/transaction.controller.js';

import { authMiddleware } from '../middlewares/auth.middleware.js';
import { validate } from '../middlewares/validate.middleware.js';

import {
  createTransactionSchema,
  getTransactionsSchema,
  transactionIdSchema,
  updateTransactionSchema
} from '../validations/transaction.validation.js';

const router = Router();

router.use(authMiddleware);

router.get('/', validate(getTransactionsSchema), getTransactions);

router.get('/:id', validate(transactionIdSchema), getTransactionById);

router.post('/', validate(createTransactionSchema), createTransaction);

router.put('/:id', validate(updateTransactionSchema), updateTransaction);

router.delete('/:id', validate(transactionIdSchema), deleteTransaction);

export default router;
```

ДОДАТОК Г

Express transaction.controller.js

```
import { asyncHandler } from '../utils/async-handler.js';

import {
  createUserTransaction,
  deleteUserTransaction,
  getUserTransactionById,
  getUserTransactions,
  updateUserTransaction
} from '../services/transaction.service.js';

export const getTransactions = asyncHandler(async (req, res) => {
  const result = await getUserTransactions({
    userId: req.user.id,
    filters: req.validated.query
  });
  res.json({
    status: 'success',
    data: result
  });
});

export const getTransactionById = asyncHandler(async (req, res) => {
  const transaction = await getUserTransactionById({
    transactionId: req.validated.params.id,
    userId: req.user.id
  });

  res.json({
    status: 'success',
    data: {
      transaction
    }
  });
});

export const createTransaction = asyncHandler(async (req, res) => {
```

```
  const transaction = await createUserTransaction({
    userId: req.user.id,
    data: req.validated.body
  });

  res.status(201).json({
    status: 'success',
    message: 'Transaction has been created successfully',
    data: {
      transaction
    }
  });
});

export const updateTransaction = asyncHandler(async (req, res) => {
  const transaction = await updateUserTransaction({
    transactionId: req.validated.params.id,
    userId: req.user.id,
    data: req.validated.body
  });

  res.json({
    status: 'success',
    message: 'Transaction has been updated successfully',
    data: {
      transaction
    }
  });
});

export const deleteTransaction = asyncHandler(async (req, res) => {
  const result = await deleteUserTransaction({
    transactionId: req.validated.params.id,
    userId: req.user.id
  });

  res.json({
    status: 'success',
    message: result.message
  });
});
```

ДОДАТОК Д

Express transaction.service.js

```
import { Prisma } from '@prisma/client';

import { prisma } from '../config/prisma.js';
import { AppError } from '../utils/app-error.js';

function toDecimal(value) {
  return new Prisma.Decimal(value);
}

function formatAccount(account) {
  if (!account) {
    return null;
  }

  return {
    ...account,
    initialBalance: Number(account.initialBalance),
    currentBalance: Number(account.currentBalance)
  };
}

function formatTransaction(transaction) {
  return {
    ...transaction,
    amount: Number(transaction.amount),
    account: formatAccount(transaction.account),
    targetAccount: formatAccount(transaction.targetAccount)
  };
}

const transactionInclude = {
  account: true,
  targetAccount: true,
  category: true
};
```

```
async function getAccountOrThrow(tx, { accountId, userId, label = 'Account' }) {
  const account = await tx.account.findFirst({
    where: {
      id: accountId,
      userId
    }
  });

  if (!account) {
    throw new AppError(`${label} not found`, 404);
  }

  return account;
}

async function getCategoryOrThrow(tx, { categoryId, userId, expectedType }) {
  const category = await tx.category.findFirst({
    where: {
      id: categoryId,
      userId
    }
  });

  if (!category) {
    throw new AppError('Category not found', 404);
  }

  if (category.type !== expectedType) {
    throw new AppError(`Category type must be ${expectedType}`, 400);
  }

  return category;
}

async function validateTransactionData(tx, { userId, data }) {
  const sourceAccount = await getAccountOrThrow(tx, {
    accountId: data.accountId,
    userId,
    label: 'Source account'
  });

  if (data.type === 'INCOME' || data.type === 'EXPENSE') {
```

```
    await getCategoryOrThrow(tx, {
      categoryId: data.categoryId,
      userId,
      expectedType: data.type
    });

    return {
      ...data,
      targetAccountId: null
    };
  }

  const targetAccount = await getAccountOrThrow(tx, {
    accountId: data.targetAccountId,
    userId,
    label: 'Target account'
  });

  if (sourceAccount.id === targetAccount.id) {
    throw new AppError('Source and target accounts must be different', 400);
  }

  if (sourceAccount.currency !== targetAccount.currency) {
    throw new AppError(
      'Transfer between accounts with different currencies is not supported yet',
      400
    );
  }

  return {
    ...data,
    categoryId: null
  };
}

async function applyTransactionEffect(tx, transaction) {
  const amount = toDecimal(transaction.amount);

  if (transaction.type === 'INCOME') {
    await tx.account.update({
      where: {
        id: transaction.accountId
```

```
    },
    data: {
      currentBalance: {
        increment: amount
      }
    }
  });

  return;
}

if (transaction.type === 'EXPENSE') {
  await tx.account.update({
    where: {
      id: transaction.accountId
    },
    data: {
      currentBalance: {
        decrement: amount
      }
    }
  });

  return;
}

if (transaction.type === 'TRANSFER') {
  await tx.account.update({
    where: {
      id: transaction.accountId
    },
    data: {
      currentBalance: {
        decrement: amount
      }
    }
  });

  await tx.account.update({
    where: {
      id: transaction.targetAccountId
    },
```

```
      data: {
        currentBalance: {
          increment: amount
        }
      }
    });
  }
}

async function revertTransactionEffect(tx, transaction) {
  const amount = toDecimal(transaction.amount);

  if (transaction.type === 'INCOME') {
    await tx.account.update({
      where: {
        id: transaction.accountId
      },
      data: {
        currentBalance: {
          decrement: amount
        }
      }
    });

    return;
  }

  if (transaction.type === 'EXPENSE') {
    await tx.account.update({
      where: {
        id: transaction.accountId
      },
      data: {
        currentBalance: {
          increment: amount
        }
      }
    });

    return;
  }
```

```
  if (transaction.type === 'TRANSFER') {
    await tx.account.update({
      where: {
        id: transaction.accountId
      },
      data: {
        currentBalance: {
          increment: amount
        }
      }
    });

    await tx.account.update({
      where: {
        id: transaction.targetAccountId
      },
      data: {
        currentBalance: {
          decrement: amount
        }
      }
    });
  }
}

function buildDateFilter({ from, to }) {
  if (!from && !to) {
    return undefined;
  }

  const filter = {};

  if (from) {
    filter.gte = new Date(from);
  }

  if (to) {
    const endDate = new Date(to);
    endDate.setHours(23, 59, 59, 999);
    filter.lte = endDate;
  }
```

```
  return filter;
}

export async function getUserTransactions({ userId, filters }) {
  const {
    page,
    limit,
    type,
    accountId,
    categoryId,
    from,
    to,
    search
  } = filters;

  const skip = (page - 1) * limit;

  const dateFilter = buildDateFilter({ from, to });

  const where = {
    userId,

    ...(type && {
      type
    }),

    ...(categoryId && {
      categoryId
    }),

    ...(accountId && {
      OR: [
        {
          accountId
        },
        {
          targetAccountId: accountId
        }
      ]
    }),

    ...(dateFilter && {
```

```
    date: dateFilter
  }),

  ...(search && {
    comment: {
      contains: search,
      mode: 'insensitive'
    }
  })
};

const [total, transactions] = await prisma.$transaction([
  prisma.transaction.count({
    where
  }),

  prisma.transaction.findMany({
    where,
    include: transactionInclude,
    orderBy: [
      {
        date: 'desc'
      },
      {
        createdAt: 'desc'
      }
    ],
    skip,
    take: limit
  })
]);

const totalPages = Math.ceil(total / limit);

return {
  transactions: transactions.map(formatTransaction),
  pagination: {
    page,
    limit,
    total,
    totalPages
  }
```

```
  };
}

export async function getUserTransactionById({ transactionId, userId }) {
  const transaction = await prisma.transaction.findFirst({
    where: {
      id: transactionId,
      userId
    },
    include: transactionInclude
  });

  if (!transaction) {
    throw new AppError('Transaction not found', 404);
  }

  return formatTransaction(transaction);
}

export async function createUserTransaction({ userId, data }) {
  return prisma.$transaction(async (tx) => {
    const preparedData = await validateTransactionData(tx, {
      userId,
      data
    });

    const transaction = await tx.transaction.create({
      data: {
        type: preparedData.type,
        amount: toDecimal(preparedData.amount),
        date: preparedData.date,
        comment: preparedData.comment?.trim() || null,

        userId,

        accountId: preparedData.accountId,
        targetAccountId: preparedData.targetAccountId || null,
        categoryId: preparedData.categoryId || null
      },
      include: transactionInclude
    });
```

```
    await applyTransactionEffect(tx, transaction);

    return formatTransaction(transaction);
  });
}

export async function updateUserTransaction({ transactionId, userId, data }) {
  return prisma.$transaction(async (tx) => {
    const existingTransaction = await tx.transaction.findFirst({
      where: {
        id: transactionId,
        userId
      }
    });

    if (!existingTransaction) {
      throw new AppError('Transaction not found', 404);
    }

    const nextType = data.type ?? existingTransaction.type;

    const nextData = {
      type: nextType,
      amount: data.amount ?? Number(existingTransaction.amount),
      date: data.date ?? existingTransaction.date,
      comment:
        data.comment !== undefined
          ? data.comment
          : existingTransaction.comment,
      accountId: data.accountId ?? existingTransaction.accountId,
      targetAccountId:
        data.targetAccountId !== undefined
          ? data.targetAccountId
          : existingTransaction.targetAccountId,
      categoryId:
        data.categoryId !== undefined
          ? data.categoryId
          : existingTransaction.categoryId
    };

    if (nextData.type === 'TRANSFER') {
      nextData.categoryId = null;
```

```
    }

    if (nextData.type === 'INCOME' || nextData.type === 'EXPENSE') {
      nextData.targetAccountId = null;
    }

    const preparedData = await validateTransactionData(tx, {
      userId,
      data: nextData
    });

    await revertTransactionEffect(tx, existingTransaction);

    const updatedTransaction = await tx.transaction.update({
      where: {
        id: transactionId
      },
      data: {
        type: preparedData.type,
        amount: toDecimal(preparedData.amount),
        date: preparedData.date,
        comment: preparedData.comment?.trim() || null,

        accountId: preparedData.accountId,
        targetAccountId: preparedData.targetAccountId || null,
        categoryId: preparedData.categoryId || null
      },
      include: transactionInclude
    });

    await applyTransactionEffect(tx, updatedTransaction);

    return formatTransaction(updatedTransaction);
  });
}

export async function deleteUserTransaction({ transactionId, userId }) {
  return prisma.$transaction(async (tx) => {
    const existingTransaction = await tx.transaction.findFirst({
      where: {
        id: transactionId,
        userId
```

```
      }
    });

    if (!existingTransaction) {
      throw new AppError('Transaction not found', 404);
    }

    await revertTransactionEffect(tx, existingTransaction);

    await tx.transaction.delete({
      where: {
        id: transactionId
      }
    });

    return {
      message: 'Transaction has been deleted successfully'
    };
  });
}
```