Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики

**Кваліфікаційна робота**

освітній ступінь бакалавр

на тему: **" Розробка мобільного або веб-застосування для розподілу витрат "**

Текстова частина до курсової роботи

Виконала: студентка 3-го року навчання,
Освітньої програми «Комп'ютерні науки»,
122
Долбня Наталія Сергіївна
Керівник Гречко А.В.
кандидат фіз.-мат. наук

Рецензент _____
(прізвище та ініціали)

Кваліфікаційна робота залишена
з оцінкою _____

Секретар ЕК_____

“_____”_____20_____ Р

Київ-2025

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання курсової роботи

Тема: Розробка мобільного або вебзастосування для розподілу витрат

| № п/п | Назва етапу курсового проекту (роботи) | Термін виконання етапу | Примітка |
|---|---|---|---|
| 1. | Отримання завдання на курсову роботу | 25.10.2024 | |
| 2. | Ознайомлення з предметною областю | Грудень 2024 | |
| 3. | Огляд і аналіз аналогів | Січень 2025 | |
| 4. | Пошук технологій для розробки | Лютий 2025 | |
| 5. | Аналіз обраних технологій | Березень 2025 | |
| 6. | Початок створення практичної частини | 20.03.2025 | |
| 7. | Початок написання теоретичної частини | 20.04.2025 | |
| 8. | Корегування теоретичної частини | 01.05.2025 | |
| 9. | Остаточне завершення практичної та теоретичної частини | 04.05.2025 | |
| 12. | Створення презентації | 06.05.2025 | |
| 13. | Захист курсової роботи | 16.05.2025 | |

Студентка Долбня Н.С.

Керівник Гречко А. В.

“_____”_____

Зміст

Перелік умовних позначень

1. MoneyTrecker - назва розробленого вебзастосування.
2. API - інтерфейс програмування застосунків (Application Programming Interface)
3. JWT - формат токена для передачі автентифікаційної інформації (JSON Web Token)
4. DTO - об’єкт передачі даних між шарами системи (Data Transfer Object)
5. IoC - принцип інверсії керування для керування залежностями (Inversion of Control)
6. DI - механізм впровадження залежностей (Dependency Injection)
7. MVC - архітектурний шаблон «Модель-Подання-Контролер» (Model–View-Controller)
8. REST - архітектурний стиль побудови клієнт-серверної взаємодії (Representational State Transfer)
9. HTML - мова розмітки гіпертексту для структурування веб-сторінок (HyperText Markup Language)
10. CSS - каскадні таблиці стилів для оформлення інтерфейсу (Cascading Style Sheets)
11. JS / JavaScript - мова програмування для забезпечення інтерактивності веб-застосунку
12. SQL - мова структурованих запитів до реляційних баз даних (Structured Query Language)
13. ORM - механізм об'єктно-реляційного відображення (Object-Relational Mapping), наприклад, Hibernate
14. BCrypt - алгоритм хешування паролів із підвищеним рівнем безпеки
15. CrudRepository / JpaRepository - інтерфейси Spring Data JPA для роботи з базою даних

Вступ

Доволі часто у подорожах або при веденні спільного сімейного бюджету люди зіштовхуються з потребою фіксувати та ділити витрати. Ручний облік, наприклад, у таблицях Microsoft Excel не рідко призводить до плутанини, втрати даних або нерозуміння, "хто кому винен і скільки".

Хоча вже існують певні цифрові рішення, такі як Splitwise, вони не завжди враховують усі потреби користувачів, зокрема повторювані витрати, часткові чи відсоткові поділи, спрощення боргових зобов'язань та інші корисні функції. Тому розробка програми для управління спільними витратами є актуальним завданням.

Метою даної курсової роботи є створення веб-застосування для зручного розподілу витрат між користувачами, яке спрощуватиме контроль над спільним бюджетом.

Завдання курсової роботи:

1. Провести аналіз систем-аналогів, виявити їх переваги та недоліки.
2. Сформулювати функціональні вимоги до майбутнього застосування.
3. Розробити архітектуру вебзастосування та структуру бази даних;
4. Реалізувати основні функціональні можливості системи розподілу витрат.
5. Провести тестування.

Об’єктом дослідження можна вважати процес спільного обліку витрат та боргів у побутових і групових ситуаціях.

Предметом дослідження є програмні засоби автоматизації ведення спільного фінансового обліку в межах веб-застосування.

Практичне значення одержаних результатів полягає в розробці застосування, що зможе використовуватись у повсякденному житті - як у сім’ях, так і у групах друзів, студентів, колег. Це дозволить спростити контроль витрат, зменшити непорозуміння у фінансових питаннях та зробити процес ведення бюджету прозорим і зручним.

Для реалізації проєкту планується використовувати:

- Java 23 як основну мову програмування для серверної логіки;
- IntelliJ IDEA, система збирання Maven та GitHub як віддалений репозиторій;
- Spring Boot, Spring Security, Spring Scheduler, Spring Data JPA (Hibernate) для побудови backend-архітектури;
- JWT для авторизації;
- PostgreSQL як реляційна база даних;
- MapStruct і DTO для організації передачі даних;
- REST API як модель взаємодії між клієнтом і сервером.

# РОЗДІЛ 1 Аналіз предметної області. Постановка завдання курсової роботи

## 1.1 Аналіз сучасного стану питання та обґрунтування теми

В сучасному світі люди часто стикаються з необхідністю спільно вести облік коштів - від сімейного бюджету, до групових поїздок, дружніх зустрічей, бюджетів студентських груп, тощо. Така співпраця зазвичай передбачає нерівномірні та несинхронізовані грошові витрати з боку членів цих груп, що призводить до складної або навіть хаотичної системи боргів. Спроби вручну відстежувати ці дані за допомогою паперових записників або листувань у месенджерах є незручними, неефективними та іноді помилковими.

Це обумовило появу цифрових інструментів, що автоматизують облік витрат, розрахунок боргів та балансів. Одним із найвідоміших сервісів подібного типу є Splitwise. Проте навіть цей сервіс має ряд значних обмежень. Багато наявних рішень:

1. не підтримують зручний розподіл витрат за відсотками або частками;
2. не підтримують періодично повторювані витрати (наприклад, місячну плату за оренду житла);
3. не підтримують прогнозування або планування витрат;
4. не виконують автоматичного спрощення боргів між членами групи;
5. на практиці є обмеженими версіями платних сервісів;
6. доступні лише користувачам певних мов.

Це вказує на необхідність спроби створити власне веб-застосування, призначене для вирішення вищезазначених проблем і та полегшення процесу спільного обліку коштів.

## 1.2 Огляд існуючих аналогів розробки

На сучасному цифровому ринку існує чимало популярних програм для обліку спільних витрат, які пропонують функції розподілу витрат, ведення

боргових балансів та фінансової взаємодії між користувачами. Найбільш поширеними рішеннями в цій сфері є такі застосування, як Splitwise, Settle Up, Kittysplit, а також менш відомі варіанти як Billzer.

### 1.2.1 Splitwise

Splitwise [1] - один із найбільш відомих сервісів для управління спільними витратами у світі, доступний у веб версії та в GooglePlay, AppStore. Його основною перевагою є гнучкі варіанти розподілу витрат, що дають змогу рівномірно, вручну або відсотковим методом розподілити фінансові зобов'язання. Застосування також оперативно показує історію транзакцій та надає інформацію про суму боргу між учасниками. Підтримка мультивалютності дозволяє зручно зазначати валюту для кожного конкретного платежу. Серед додаткових функцій - можливість прикріплення фотографій чеків, додавання коментарів та push-сповіщення про нові витрати. Серед недоліків варто відзначити відсутність опцій для налаштування регулярних витрат, а також інтеграції з банківськими системами, що ускладнює реалізацію реальних фінансових переказів. Окрім цього, локалізація для україномовних користувачів залишається недосконалою: частина інтерфейсу перекладена неналежним чином або зовсім не адаптована під українську мову.

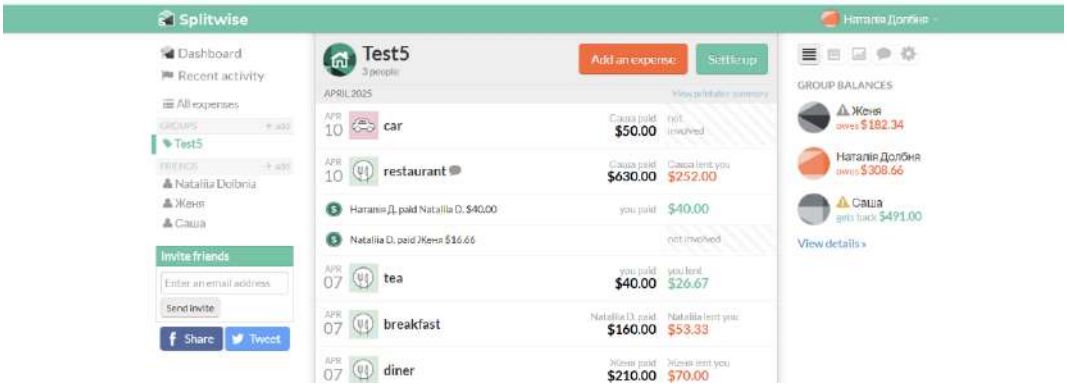


Рисунок 1.2.1. - Інтерфейс головної сторінки веб-сайту « Splitwise»

### 1.2.2. Settle Up

Settle Up [2] - застосування, призначене насамперед для використання у подорожах і для короткострокового обліку спільних витрат. Орієнтоване переважно на мобільні платформи. Перевагами цієї системи є проста багатомовна локалізація, швидке додавання витрат із можливістю прикріплення чеків та нотатків, а також базова статистика, що має розширення у платній версії. Інтерфейс сторінки веб-сайту можна побачити на рис. 1.2.2. Недоліками системи є відсутність підтримки повторюваних витрат, а також функція редагування транзакцій, що існує без жодної позначки “змінено”. Це може привести до недоброчесного використання програми.

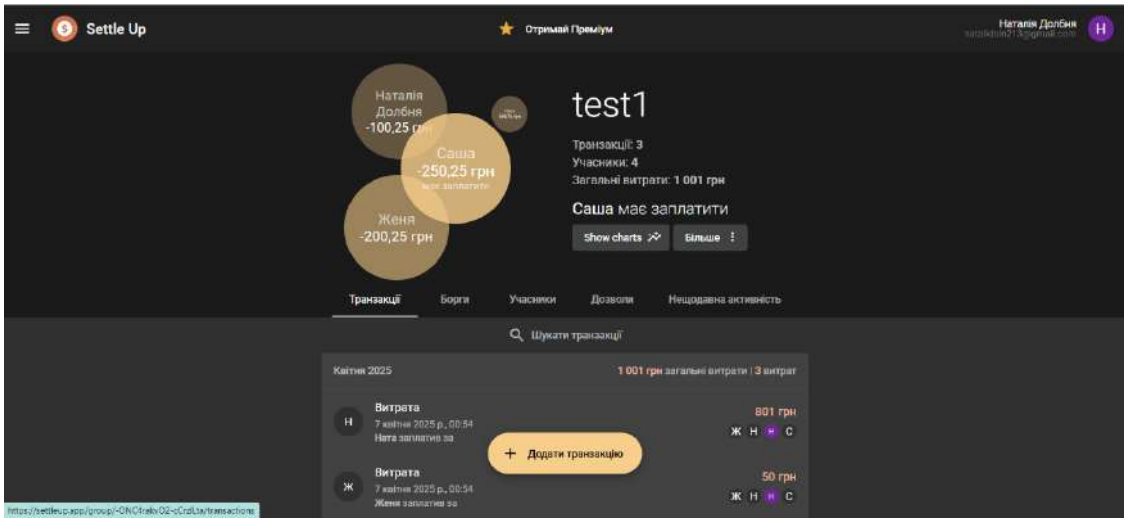


Рисунок 1.2.2. - Інтерфейс головної сторінки веб-сайту «Settle Up»

### 1.2.3. Kittysplit

Kittysplit [3] - максимально спрощений веб-сервіс, що не потребує реєстрації. Орієнтований переважно на короткочасні події (вечірки, поїздки), через це має протий інтерфейс та обмежений набір функцій, тобто немає регулярних витрат, детальної аналітики та прогнозування. Проте сплату боргу можна не просто записати у застосування, а й оплатити через сам kittysplit. Станом на зараз є можливість оплати за допомогою PayPal. Слід зауважити, що в цій системі повністю відсутня підтримка української мови.

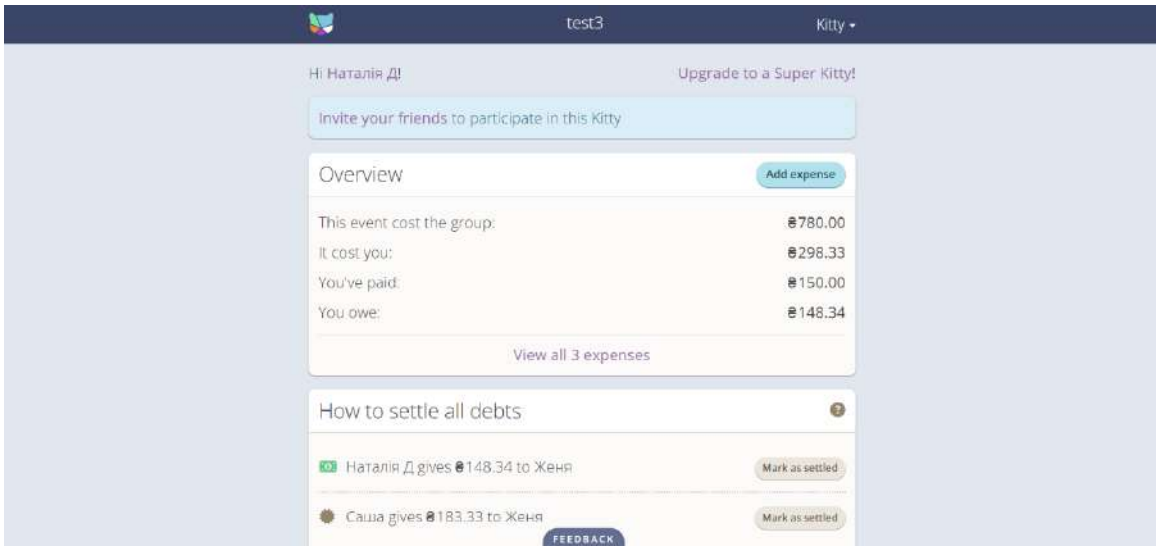


Рисунок 1.2.3. - Інтерфейс головної сторінки веб-сайту « Kittysplit»

Отже виходячи з наведеного порівняння, у сервісах, що вже існують, відсутні окремі функції, жоден продукт не поєднує простоту, локалізацію та аналітичну складову. Порівняння систем аналогів наведено в таблиці 1.2.1.

| Критерій | Splitwise | SettleUp | Kittysplit |
|---|---|---|---|
| Рівномірний, нерівномірний та відсотковий розподіл | наявно | наявний | наявний |
| Аналітика витрат | наявно у платній версії | наявно у платній версії | наявно у платній версії |
| Повторювані витрати | відсутньо | наявно у платній версії | відсутньо |
| Спрощення боргів | наявно | наявно | наявно |
| Прогнозування витрат | наявно у платній версії | наявно у платній версії | відсутньо |

| | | | |
|---|---|---|---|
| Локалізація (українська мова) | частково наявно | частково наявно | відсутньо |
| Доступ без реєстрації | відсутньо | відсутньо | наявно |

Таблиця 1.2.1 - Таблиця порівняння аналогів розроблюваного веб-сайту

1.3 Постановка задачі

Після огляду існуючих аналогів розробки та аналізу потреб користувачів було окреслено завдання створити зручне, багатофункціональне веб-застосування для розподілу та контролю спільних витрат, яке повинно надавати користувачам такі основні функціональні можливості:

1. Створення груп учасників.

Застосування повинно дозволяти користувачам створювати необмежену кількість груп із визначенням їх назв.

2. Додавання витрат із різними варіантами розподілу. Користувачам доступне додавання витрат по розподілам:

- рівномірний (сума транзакції ділиться автоматично порівно);
- нерівномірний (сума транзакції вручну визначається для кожного учасника);
- відсотковий (сума транзакції ділиться між учасниками за введеними відсотковими вкладами).

3. Автоматичне обчислення боргів та залишків, як особистих так і групи.

Система автоматично розраховує і миттєво відображає, хто кому і яку суму винен після кожної доданої витрати.

4. Перегляд індивідуального балансу учасника.

Кожен учасник повинен бачити власний баланс: скільки він витратив, скільки йому винні інші та скільки він винен іншим учасникам конкретної групи.

5. Спрощення боргів.

Застосування повинно автоматично мінімізувати кількість необхідних переказів між учасниками.

6. Підтримка повторюваних витрат.

Застосування має дозволяти користувачам задавати регулярні витрати з певним періодом (рік, місяць, тиждень), що автоматично додаються до транзакцій.

7. Аналітика та статистика витрат.

Користувачі повинні мати можливість переглядати історію та аналітику власних витрат.

8. Прогнозування майбутніх витрат.

При додаванні витрат, застосування має пропонувати певні категорії на основі аналізу попередніх платежів.

9. Реєстрація користувачів

Застосування повинно мати змогу запам’ятовувати користувачів та захищати інформацію за допомогою аутентифікації, хешування паролів.

# РОЗДІЛ 2 Теоретичні відомості

## 2.1 Опис та обгрунтування архітектури розроблюваної системи

Проєктування веб-застосування “MoneyTrecker” спирається на клієнт-серверну архітектуру (рис. 3.2), яка поділяє систему на три логічні рівні: клієнт (інтерфейс), сервер (обробка логіки) та база даних (персистентність). Такий підхід є стандартом для веб-систем і дозволяє розподілити відповідальність, масштабувати компоненти та ізолювати ризики. Клієнтська частина відповідає за безпосередню взаємодію з користувачем, відображення інформації та обробку введених даних. У вебзастосунках вона зазвичай реалізується засобами HTML, CSS, JavaScript та бібліотеками/фреймворками для створення інтерфейсів (наприклад, React, Vue, Angular, Bootstrap тощо). Інтерфейс не повинен містити бізнес-логіки, а лише відправляти запити на сервер і реагувати на отримані відповіді. Серверна частина ж забезпечує безпосереднє виконання бізнес-логіки (розрахунки, перевірки, агрегації даних), взаємодію з базою даних та генерацію відповідей у форматі JSON, XML тощо. Для реалізації серверної частини зазвичай використовують мови програмування загального призначення (Java, Python, JavaScript (Node.js), PHP тощо) та відповідні фреймворки (Spring, Django, Express, Laravel). За допомогою протоколу взаємодії - REST API (GET, POST, PUT / PATCH, DELETE...) йде взаємодія клієнта та сервера, що ґрунтується на використанні HTTP-запитів і статичних маршрутів.

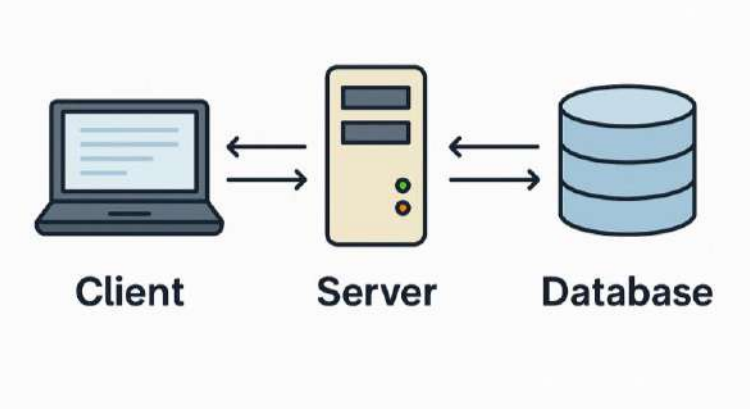


Рисунок 3.2. - Клієнт-серверна архітектура

На рівні реалізації застосунку логічна структура базується на шаблоні MVC (Model-View-Controller). За допомогою MVC можна чітко розділити обов'язки: моделі оперують даними, представлення відповідає за інтерфейс, контролери - за маршрутизацію запитів. Наприклад, модель реалізується у вигляді сутностей User, Group тощо, і вони пов’язані з базою даних через Spring Data JPA. За представлення відповідають динамічні HTML-сторінки з JavaScript та Fetch API. Контролер обробляє HTTP-запити користувача, викликає відповідні методи сервісного шару та повертає результат.

Така структура дозволяє підтримувати масштабованість і спрощує тестування. Проте це накладає додаткові витрати на комунікацію між шарами та на підтримку застосунку, бо з часом буле зростати складність логіки, кількість функцій. Альтернативні підходи, зокрема монолітна структура у вигляді єдиного PHP-додатку не дозволяє винести клієнт на мобільну платформу в майбутньому. Крім того, важче впровадити незалежне масштабування фронту й бекенду. З іншої сторони, прямий доступ клієнта до бази даних через Firebase або Supabase не забезпечує належного контролю за валідацією, безпекою й бізнес-логікою.

Таким чином, обрана архітектура є компромісом між простотою реалізації та гнучкістю при масштабуванні системи в майбутньому.

## 2.2 Алгоритми спрощення боргів між учасниками

Система передбачає визначення боргових зобов’язань між учасниками на основі здійснених витрат та їх часток у загальних витратах. Баланс кожного користувача визначається як різниця між сумою фактично витраченого і сумою, яку він мав би внести відповідно до обраного способу розподілу. Завдання спрощення боргів полягає в тому, щоб зменшити кількість переказів та уникнути кругових переказів.

Існують наступні підходи [4-6] (порівняння наведено в табл. 2.2.1):

1. Жадібний алгоритм - ітеративно знаходиться найбільший боржник та найбільший кредитор, між ними здійснюється переказ на мінімум із абсолютних значень їхніх балансів. Цикл повторюється, поки всі залишки не стануть нульовими або близькими до нуля (з урахуванням похибки округлення). Приклад: припустимо A: -100 B: -50 C: +150, отже: A → C: 100, B → C: 50 маємо 2 транзакції
2. Повний перебір - перебираються всі можливі варіанти спрощення боргів, вибирається той, що мінімізує кількість транзакцій.
3. Графовий підхід - боргові відносини подаються у вигляді орієнтованого графа, де вузли - учасники, ребра - борги. Таким чином знаходяться цикли, а отже в прикладі: A → B (50), B → C (50), C → A (50) цикли дозволяють анулювати борги.
4. LP-оптимізація - формалізація задачі як лінійної моделі з ціллю мінімізувати суму кількості переказів або їх загальну суму при дотриманні балансових обмежень.

| Алгоритм | Оптимальність | Швидкодія | Складність | Практичність |
|---|---|---|---|---|
| Жадібний (Greedy) | Не завжди дає мін. кількість транзакцій | $O(n^2)$ | Проста у реалізації | Для невеликих сервісів |
| Перебір (Exhaustive search) | Мін. кількість транзакцій | $O(n!)$ | Складна, потребує математичної моделі | Не практичний |
| Графовий (Cyclic reduction) | Не завжди дає мін. кількість транзакцій | $O(n^3)$ | Середня у реалізації | Для невеликих сервісів |

| | | | | |
|---|---|---|---|---|
| LP-оптимізація | Мін. кількість транзакцій при правильній постановці задачі | $O(n^4)$ | Складна, потребує математичної моделі | Для складних фінансових систем |

2.2.1 Таблиця - Порівняння алгоритмів спрощення боргів

2.3 Класифікація витрат і способів розподілу

Модель розподілу безпосередньо впливає на обчислення боргу та балансів.

1. Рівномірний розподіл - стандартна модель, де сума витрати ділиться порівну між усіма учасниками. Її перевага - простота, недолік - відсутність гнучкості.
2. Розподіл за відсотками - учасники сплачують у пропорціях, заданих відсотками. Проблема таких моделей полягає в необхідності механізму перевірки коректності введених часток.
3. Нерівномірний розподіл - користувачи вручну вводяти сумму, що сплатив той чи інший учасник. Потрібна автоматична перевірка на рівність суми введених частин до загальної витрати.

У деяких випадках також використовуються:

1. Фіксована сума - один учасник бере фіксовану частину, решта - рівномірно;
2. Ієрархічна модель - у сім'ї батьки беруть більшу частину;
3. Зовнішній спонсор - один учасник повністю покриває витрату.

В додатку реалізовані лише три моделі (рівномірний, розподіл за відсотками, нерівномірний) через один універсальний інтерфейс створення транзакцій, де користувач може вибрати тип розподілу через випадаючий список.

# РОЗДІЛ 3 Опис реалізації програмного продукта

## 3.1 Аналіз технічного завдання

Спираючись на аналіз розглянутих аналогів у розділі 1.2 та певні потреби користувачів можна сформувати технічне завдання на створення веб-застосування MoneyTrecker.

Цільовою авдиторією є неформальні групи друзів, колег, студентів та родини. Маємо різних за статтю та віком користувачів, що єдиним спільним мають ведення спільного бюджету вдома або під час подорожей.

Тому до застосування логічно висунути такі нефункціональні вимоги:

1. Максимально інтуїтивний інтерфейс без нагромадження функцій;

Інтерфейс користувача має бути інтуїтивно зрозумілим та адаптованим під користувачів з різним досвідом користування техніки.

2. Стійкість до помилок вводу користувача.

Всі місця для вводу користувача повинні мати строгу фільтрацію за форматом.

3. Захист персональних даних

Обов’язкове шифрування паролів і обмеження видимості груп/витрат лише для учасників. Підтримка CORS, CSRF-захисту і обмежень доступу до API .

4. Вимоги до клієнта

Сторінка має завантажуватись менше ніж за 3 секунди на 3G-з’єднанні. Максимальний розмір JS-файлів - не більше 300-400 кб.

Далі конкретизуємо основні функціональні вимоги для реалізації застосування:

1. Створення груп та управління учасниками;

Користувач має змогу створити фінансову групу, задати їй назву, запрошувати або видаляти учасників. Кожна витрата буде належати конкретній групі.

2. Додавання витрат із вказанням платника, суми та способу поділу;

Користувач може додавати транзакції вказуючи, хто здійснив оплату, причину, час, суму та тип розподілу витрати.

3. Підтримка рівного, відсоткового й нерівномірного поділу витрат;

Три типи розподілу боргу забезпечить гнучність системи до різних сценаріїв використання.

4. Обчислення боргів, формування індивідуального балансу та спрощення боргів;

Має бути реалізований механізм автоматичного запису боргу, його спрощення та відображення особистого балансу користувача.

5. Повторювані витрати;

Встановлення проведення певного платежу з періодом, щоб користувач не вводив один і той самий запит декілька разів.

6. Перегляд історії витрат;

Можливість бачити список усіх витрат у групі з фільтрацією за датою, платником, назвою.

Для розширеної версії веб застосунку у перспективі можливе додавання email-сповіщення та інтеграція з сервісами, на приклад, Google Calendar та платіжними інструментами, на приклад, Google Pay, Apple Pay. У перспективі може плануватися розширення функціональності у вигляді мобільного додатку або PWA-версії, що зумовлює необхідність підтримки масштабованої архітектури.

### 3.2 Обґрунтування алгоритму й структури програми

Розробка застосування MoneyTrecker реалізується відповідно до архітектурної моделі MVC, розгляднутої в розділі 2.1 роботи. Алгоритм роботи системи визначається основними сценаріями використання, такими як: створення групи, додавання витрати, розподіл витрати, обчислення боргів,

спрощення боргів та погашення заборгованості. Схема роботи веб-застосування зображена на рис. 3.2.1.

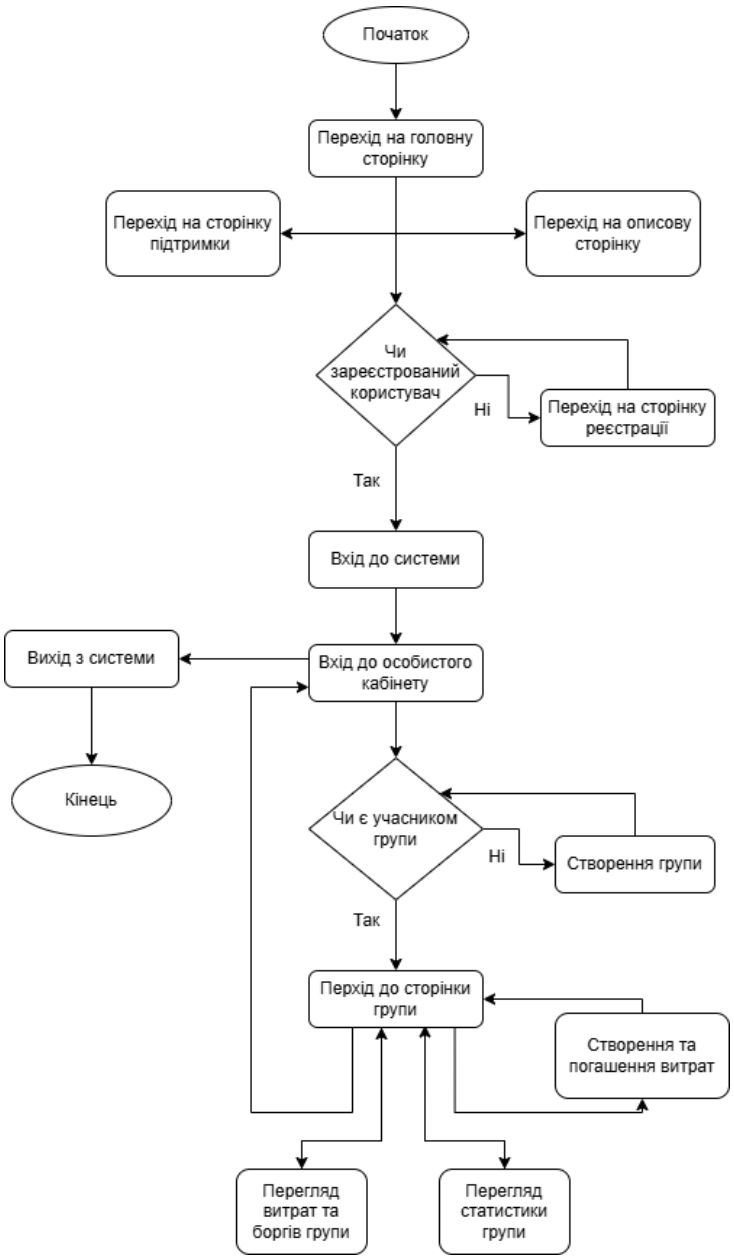


Рисунок 3.2.1 - Схема роботи веб-застосування

На рівні клієнта реалізовано інтерфейс взаємодії користувача із системою з використанням HTML, CSS і JavaScript, а також бібліотек Fetch API для асинхронної взаємодії з сервером. Шаблонні HTML-файли зберігаються у директорії *resources/templates,* стилі - у *resources/static/css*, а скрипти - у *resources/static/js.* і відображаються за допомогою MVC-контролерів. Двомовність забезпечується шаблонами, що знаходяться у *messages.properties*

На рівні сервера логіка проєкту організована у вигляді окремих компонентів: *controller/*, що містить REST-контролери для взаємодії через API у форматі JSON; *service/*, що зберігає інструкції щодо бізнес-логіки та обробки запитів; *repository/,* що забезпечує доступ до бази даних на основі Spring Data JPA; *entity/* - містить Java-класи, що відповідають таблицям бази даних. Для зберігання даних використовується реляційна база даних PostgreSQL, з'єднання з якою налаштовується через файл *application.properties, р*обота з нею відбувається через фреймворк Spring і вбудовану ORM фреймворк Hibernate.

У веб-застосуванні також реалізовані основні механізми захисту: хешування паролів із використанням алгоритму BCrypt, обмеження доступу до ресурсів за допомогою Spring Security та JWT-автентифікації [7], що реалізується в *security/*. Для захисту від зовнішніх атак застосовано політики CORS, перевірку CSRF-токенів та валідацію вхідних даних на стороні сервера.

Тож система MoneyTrecker побудована на принципах модульності, розділення обов'язків та незалежності компонентів, через це маємо гнучку основу для розширення функціоналу, повторного використання коду та підтримки в майбутньому.

## 3.3 Обгрунтування вибору засобів розробки

Для організації процесу розробки застосунку "MoneyTrecker" було використано набір інструментів, що забезпечують структуровану побудову проєкту, автоматизацію збірки, контроль версій та зручність у щоденній роботі з кодом.

### 3.3.1 Інструменти розробки. Інтегроване середовище розробки

IntelliJ IDEA (JetBrains) [8] - основне середовище для розробки серверної частини на Java. Підтримка Spring Boot, вбудовані інструменти для роботи з базами даних, тестування, дебагінгу та інтеграція з системою контролю версій дозволили ефективно керувати проєктом.

Система збирання - Maven - інструмент для керування залежностями та конфігурації збірки. Дозволяє централізовано визначити бібліотеки, плагіни, профілі середовища та автоматизувати компіляцію. Проте при зростанні проєкту Maven можливо буде ускладнюватись і це відобразиться на збільшенні часу збирання.

Git - система керування версіями, що забезпечує історію змін в проєкті.

GitHub - використовується як віддалене сховище. Це дало змогу безпечно зберігати код, організовувати резервне копіювання

### 3.3.2 Вибір технологій для серверної частини (Backend)

Серверну частину вебзастосунку “MoneyTrecker” реалізовано на основі фреймворку Spring Boot [9], який дозволяє швидко будувати модульні та масштабовані застосунки за допомогою попередньо налаштованих компонентів. Основним критерієм вибору стало прагнення до зменшення кількості шаблонного коду, централізації конфігурацій та автоматизації створення REST API.

Spring Boot забезпечує інверсію керування, автоматичне створення конфігурацій та модульний розподіл застосунку. Його використання дало змогу реалізувати чітку layered-архітектуру: контролери, сервіси, DTO, моделі даних. Spring Security обрано для реалізації автентифікації та авторизації. Доступ до захищених ендпоїнтів реалізовано за допомогою JWT-токенів. Spring Scheduler використано для реалізації повторюваних витрат - автоматично створюваних транзакцій із заданою періодичністю. Це спрощує введення циклічних витрат, таких як оренда чи підписки. Spring Data JPA [10] (Hibernate) застосовується як ORM-рівень для роботи з базою даних. Взаємодія з PostgreSQL[11] відбувається через репозиторії, що знижує кількість SQL-коду та підвищує узгодженість бізнес-логіки з моделлю даних. Хоча й JPA з Hibernate приховує частину SQL-процесів, що може призвести до неочікуваних запитів або проблем із продуктивністю при складній агрегації.

Розглянуті альтернативи:

- Node.js (Express) має нижчий поріг входу, але відсутність строгого контракту між шарами і більша кількість ручної конфігурації (без підтримки IoC) робить його менш придатним для масштабного фінансового сервісу.
- Firebase - хмарне рішення з вбудованою автентифікацією та базою даних, але не підходить для реалізації складної кастомної логіки та повністю залежить від інфраструктури Google.
- Nest.js (TypeScript): ближчий до Spring за архітектурою, але має нижчу зрілість екосистеми та потребує більше часу на налаштування.

### 3.3.3 Вибір технологій для клієнтської частини (Frontend)

Клієнтська частина веб-застосування реалізована з використанням стеку - HTML5, CSS3 та JavaScript (у вигляді модульного Vanilla JS без використання фреймворків). Такий підхід задовольняє потребу створити просту та керовану архітектуру MVP-рівня без перевантаження додатковими залежностями. HTML та CSS використовуються для побудови структури інтерфейсу та стилізації відповідно до функціонального контексту (групи, профіль тощо). JavaScript (Vanilla) відповідає за інтерактивність: обробку подій, валідацію форм, оновлення DOM і взаємодію з сервером. Для надсилання HTTP-запитів до REST API застосовується Fetch API . Запити формуються динамічно та обробляють JSON-відповіді без використання зовнішніх бібліотек(наприклад Axios). Адаптивність забезпечується за допомогою Grid-розмітки та CSS media-запитів. Інтерфейс адаптований до перегляду на різних пристроях.

Такі альтернативми як React / Vue / Angular не використовувалися з міркувань спрощення структури MVP, уникнення початкових накладних витрат на конфігурацію та збереження повного контролю над кодом. Також Bootstrap / Tailwind CSS були відкинуті для уникнення шаблонності у стилізації та надлишкових залежностей у верстці.

Таким чином, використання чистих HTML, CSS і JavaScript забезпечило контрольовану, легку та цілком достатню для проєкту реалізацію функціоналу та інтерфейсу користувача. Це спростило налагодження взаємодії з бекендом та зменшив загальну складність структури без шкоди для користувацького досвіду.

### 3.3.4 Вибір технологій для роботи з базою данних

Для побудови бази даних було обрано PostgreSQL. Такий вибір зумовлений характером задачі, найзручніше побудувати основну модель в реляціях, наприклад, кожен користувач може входити до кількох груп, кожна група містить множину витрат, кожна витрата має одного платника і декілька учасників, кожен з яких має свою частку у витраті. Тобто вимагається суворо структуроване зберігання даних, підтримка складних зв'язків між сутностями, а також високого рівня узгодженості та цілісності інформації. Ключовою перевагою PostgreSQL є підтримка ACID-транзакцій (атомарність, узгодженість, ізольованість, непорушність), що критично важливо для фінансових застосунків.

Під час роботи також були розглянуті альтернативні рішення: MySQL, MongoDB та Firebase.

1. MySQL хоча й має схожу реляційну природу, поступається PostgreSQL у гнучкості роботи з JSON, уніфікованій підтримці транзакцій та можливості використання складних SQL-конструкцій.
2. MongoDB (NoSQL) не використана через відсутність нормальної підтримки транзакцій між пов'язаними сутностями та складність у забезпеченні цілісності даних та ручну реалізацію зв'язків .
3. Firebase (хмарне NoSQL) дає зручну синхронізацію в реальному часі, однак є непридатною для складної обробки даних, зокрема для операцій над взаємопов'язаними структурами, таких як спрощення боргів, обчислення балансів та агрегація історії витрат.

Загалом PostgreSQL є оптимальним вибором хоча не позбавлена недоліків в складності створення та налаштуваннята програшу в часі на простих SELECT'ax [12].

## 3.4 Опис розробки програми

Розробка веб-застосування відбувалась за об'єктно-орієнтованим підходом із дотриманням принципів клієнт-серверної архітектури, розглянутої в розділі 2.1 роботи. У зв’язку з цим структура коду була логічно розділена на декілька модулів, що реалізують окремі функціональні частини. Структура проєкту наведена на рис. 3.4.1.

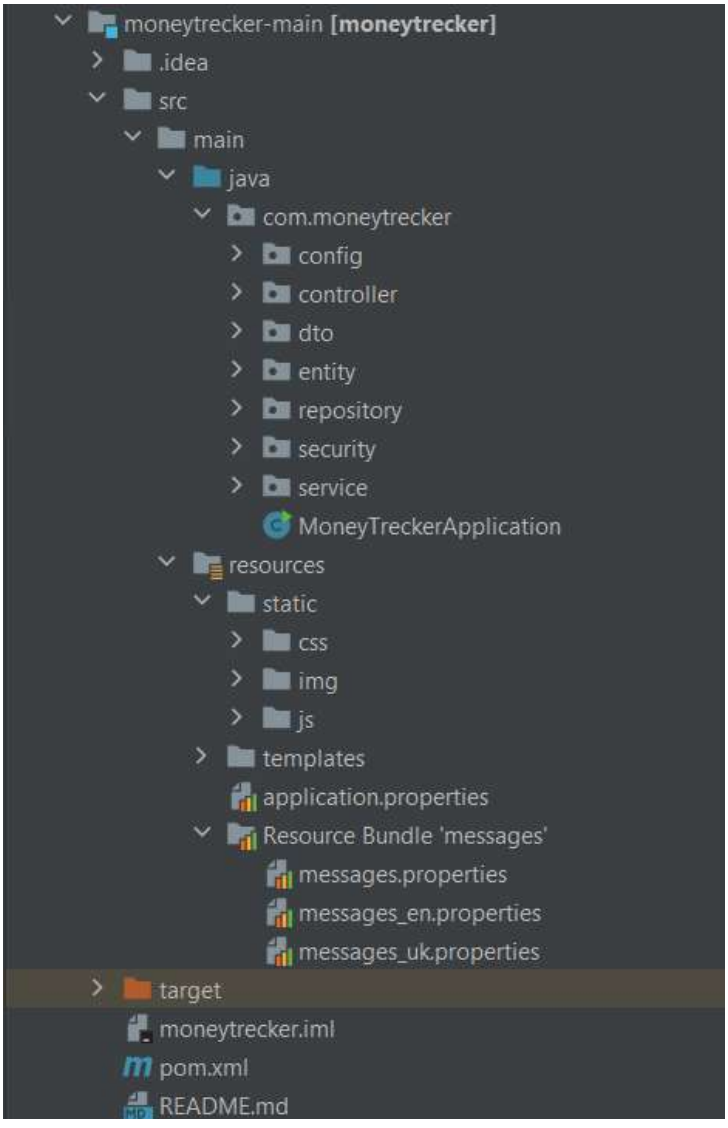


Рисунок 3.4.1 - Структура проєкту

3.4.1 Опис розробки серверної частини

*MoneyTreckerApplication* - це точка входу в застосунок, позначена анотацією @SpringBootApplication. Ініціалізує Spring-контекст, налаштовує всі компоненти та запускає сервер (рис. 3.4.1.1).

```
package com.moneytrecker;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

1 usage  NataliyaSerhiivna
@SpringBootApplication
public class MoneyTreckerApplication {

    NataliyaSerhiivna
    public static void main(String[] args) {
        SpringApplication.run(MoneyTreckerApplication.class, args);
    }
}
```

Рисунок 3.4.1.1 Код класу MoneyTreckerApplication

*Controller/ - містить* як REST, позначені анотацією *@RestController* (рис. 3.4.1.2), так і MVC-контролери - *@Controller* (рис. 3.4.1.3). Вони відповідають за маршрутизацію HTTP-запитів, виклик сервісів і формування відповіді у форматі JSON або HTML-шаблону.

```
@RestController
@RequestMapping("/api/expenses")
@RequiredArgsConstructor
public class ExpenseController {
    new *
    @GetMapping("/my/statistics")
    public ResponseEntity<Map<String, Object>>
    getMyExpenseStats(@AuthenticationPrincipal CustomUserDetails userDetails) {
        Long userId = userDetails.getUser().getId();
        return ResponseEntity.ok(expenseService.getExpenseStatsByUser(userId));
    }
```

Рисунок 3.4.1.2 Частина коду класу ExpenseController

```
@Controller
public class ProfileController {

    NataliyaSerhiivna
    @GetMapping("/profile")
    public String profilePage() {
        return "profile";
    }
}
```

Рисунок 3.4.1.3 Частина коду класу ProfileController

*Entity/* - класи, що відображають структуру таблиць у базі даних. *@Entity* - вказує на те, що клас є сутністю та відповідає таблиці у базі даних. *@Table(name = "...")* - задає назву таблиці. *@Data* автоматично генерує всі стандартні методи (getters, setters, toString(), equals(), hashCode()), необхідні для роботи з об'єктом. Далі *@NoArgsConstructor* та *@AllArgsConstructor* створюють конструктори без параметрів (вимагається JPA) та з усіма параметрами відповідно. *@Builder* дозволяє створювати об'єкти з використанням шаблону Builder (рис. 3.4.1.6). *@Id* та *@GeneratedValue(...)* - визначають первинний ключ і стратегія автогенерації. Відношення реалізовано через JPA-анімації: *@OneToMany, @ManyToOne, @JoinColumn.*

```
@Entity
@Table(name = "expenses")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Expense {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "group_id", nullable = false)
    private Group group;

    @Column(nullable = false)
    private BigDecimal amount;
```

Рисунок 3.4.1.4 Частина коду класу Expense

*Repository/* - інтерфейси, що наслідують JpaRepository для доступу до БД. В них використовуються як стандартні методи, так і кастомні через *@Query*. Завдяки можливостям, що дає Spring Boot та Hibernate, можна не писати деякі тривіальні запити, наприклад, Optional findById(ID var1), < S extends T> S save(S var1) , Iterable findAll() і тому подібні. На рис. 3.4.1.5 продемонстровано використання таких запитів.

```java
public interface ExpenseRepository extends JpaRepository<Expense, Long> {

    6 usages  NataliyaSerhiivna
    List<Expense> findByGroupId(Long groupId);

    1 usage  new *
    @Query("SELECT COALESCE(SUM(e.amount), 0) FROM Expense e WHERE e.group.id = :groupId")
    BigDecimal sumAmountByGroupId(@Param("groupId") Long groupId);
```

Рисунок 3.4.1.5 Частина коду класу ExpenseRepository

*Service/* - зберігає в собі бізнес-логіку застосунку, позначаються через *@Service. @RequiredArgsConstructor* - це використання анотації Lombok, що дозволяє автоматично згенерувати конструктор із параметрами для всіх фінальних (final) полів. На рис. 3.4.1.6 бачимо частину коду створення витрати.

```java
@Service
@RequiredArgsConstructor
public class ExpenseService {
    1 usage
    private final GroupRepository groupRepository;
    2 usages
    private final UserRepository userRepository;

    2 usages  NataliyaSerhiivna
    public Expense createExpense(User payer, CreateExpenseRequest request) {
        Group group = groupRepository.findById(request.getGroupId())
                .orElseThrow(() -> new IllegalArgumentException("Group not found"));
        User payerFromRequest = userRepository.findById(request.getPayerId())
                .orElseThrow(() -> new IllegalArgumentException("Payer not found"));

        Expense expense = Expense.builder()
                .group(group)
                .payer(payerFromRequest)
                .amount(request.getAmount())
                .title(request.getTitle())
                .build();
```

Рисунок 3.4.1.6 Частина коду класу ExpenseService

*Dto/* - зберігає об’єкти передачі даних між фронтендом і бекендом. Знову бачимо *@Data* для генерації get/set методів та *@AllArgsConstructor* для створення повного конструктору. Також на прикладі *рис.* 3.4.1.7 в класі ExpenseDetailsDTO як аргумент використовується список аргументів іншого класу з dto - List<SplitDetailDTO>. Це все дозволяє приховати внутрішню структуру сутностей, наприклад, Expense та уникнути надлишкових полів.


Рисунок 3.4.1.7 Частина коду класу ExpenseDetailsDTO

*Security/* - модуль безпеки. В застосуванні MoneyTrecker безпека реалізована на основі Spring Security із використанням JWT (JSON Web Token) для автентифікації та авторизації. На рис. 3.4.1.8 клас SecurityConfig визначається як конфігураційний (*@Configuration*), *@EnableWebSecurity* активує Web Security. *@EnableMethodSecurity* у свою чергу дозволяє використовувати *@PreAuthorize, @Secured* на рівні методів. Тут же securityFilterChain(HttpSecurity http) конфігурує правила доступу до ендпоінтів: метод .permitAll() у конфігурації безпеки означає, що доступ до зазначених маршрутів дозволено без автентифікації. Відповідно, .authenticated() обмежує доступ лише для авторизованих користувачів.

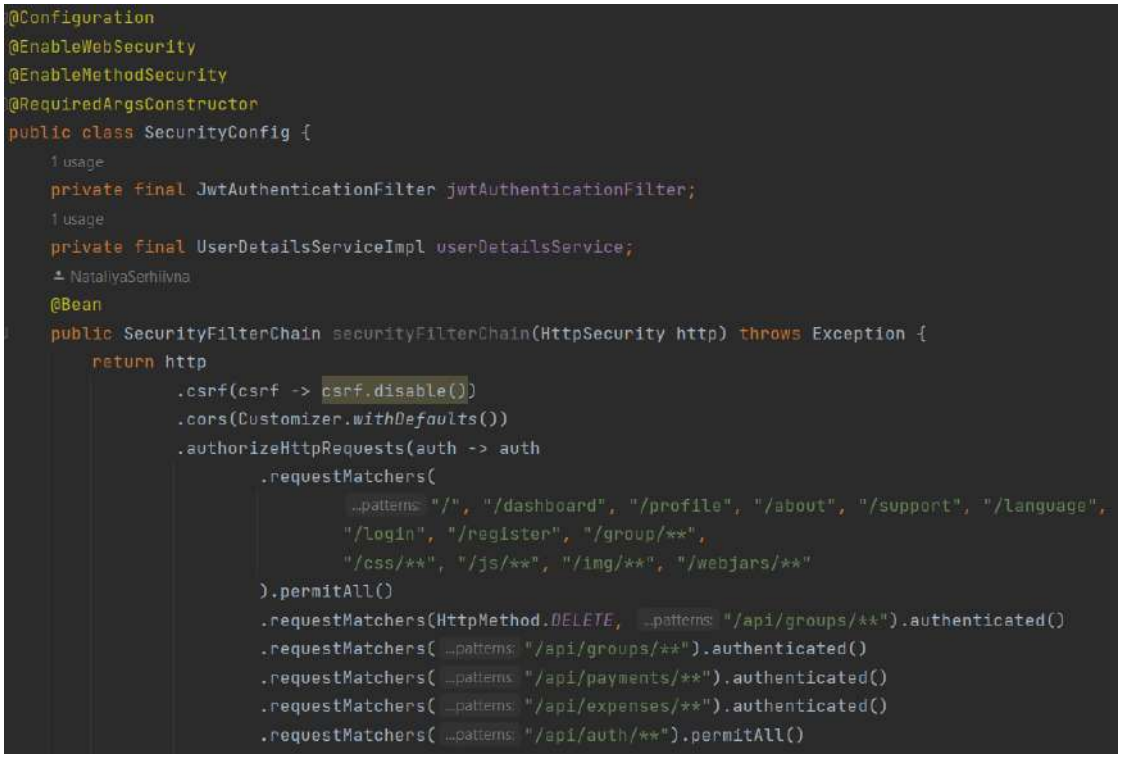


Рисунок 3.4.1.8 Частина коду класу SecurityConfig

До бази данних система підключається через налаштування в *application.properties* (рис. 3.4.1.9)

```
spring.datasource.url=jdbc:postgresql://localhost:5432/expense-tracker
spring.datasource.username=postgres
spring.datasource.password=postgresql!12
spring.jpa.hibernate.ddl-auto=update
server.port=8081
```

Рисунок 3.4.1.9 Частина коду файлу application.properties

3.4.2 Опис розробки клієнтської частини

*Resources/templates* - містить основні HTML-шаблони (login.html, register.html, group.html, ...) з використанням Thymeleaf, компонента, що створює однакову навігвційну панель зверху знаходиться в fragments/navbar.html.

*Resources/static/css* - зберігає стилі відображення HTML-сторінок. Використовуються Grid/Flexbox + медіа-запити для адаптивності.

Resources/static/js - має в собі JS-модулі для клієнтської логіки. api.js як універсальний fetchJSON та розділені за функціоналом одноіменні файли .js для відповідних HTML-сторінок.

*Messages.properties, messages_en.properties, messages_uk.properties* - є реалізацією двомовності інтерфейсу за допомогою поєднання th:text="#{register.header}", що розміщюється в HTML-шаблоні та відповідних register.header=Register і register.header=Реєстрація.

3.5 Створення об'єктів і розробка головної програми

Запуск та виконання застосунку MoneyTrecker реалізовано з використанням середовища Spring Boot, яке забезпечує автоматичне створення та ініціалізацію об’єктів через механізм інверсії керування (IoC) та впровадження залежностей (DI - Dependency Injection). Точкою входу є клас MoneyTreckerApplication (рис. 3.4.1.1). Усі компоненти застосунку (контролери, сервіси, ...) автоматично створюються Spring-контейнером, якщо вони позначені відповідними

анотаціями згаданими в попередньому пункті. Також об’єкти, які представляють сутності БД, створюються під час обробки запитів від користувача. Описаний код наведений в додатку В.

Усі основні функціональні вимоги, визначені в розділі 1.3, реалізовані у веб-застосуванні *MoneyTrecker* через створення відповідних об’єктів.

**1.Реєстрація користувачів і захист даних**

Використані об'єкти: User, LoginRequest, RegisterRequest.

Задіяні класи: AuthController, UserService, SecurityConfig, JwtAuthenticationFilter.

Процес:

1. При реєстрації POST /api/auth/register - створюється об’єкт User, пароль хешується через BCryptPasswordEncoder і в хешованому вигляді зберігається в БД.
2. При вході - токен створюється (JwtService) і зберігається у клієнта.
3. При логіні (/api/auth/login) створюється JWT-токен, який передається клієнту для подальшої аутентифікації.
4. Захищені API доступні лише при наявності дійсного JWT, який перевіряється у фільтрі JwtAuthenticationFilter.

**2.Створення груп учасників.**

Використані об'єкти: Group, User, GroupMember.

Задіяні класи: GroupService, GroupMemberService, GroupController.

Процес:

1. Користувач правильно заповнює форму створення групи та натискає зберегти, тим самим надсилає POST /api/groups.
2. Створюється об’єкт Group, який зберігається через GroupRepository.
3. Користувач автоматично додається до створеної групи як учасник (GroupMember), також заповнюється як власник групи, що має змогу видалити її.

4. Всі об’єкти зберігаються в базу через GroupRepository та GroupMemberRepository.

**3.Додавання витрат із різними варіантами розподілу**

Використані об'єкти: Expense, ExpenseSplit, CreateExpenseRequest, SplitDetailDTO.

Задіяні класи: ExpenseService, ExpenseController.

Процес:

1. Користувач правильно заповнює форму створення витрати та натискає зберегти, надсилає POST /api/expenses
2. При запиті з фронтенду (через transactionModal.js) передається вся інформація, в тому числі й тип розподілу (equal, manual, percent).
3. Сервіс ExpenseService динамічно створює екземпляр Expense та список ExpenseSplit з відповідними частками.

**4.Підтримка повторюваних витрат**

Використані об'єкти: RecurringExpense, RecurringExpenseSplit, RecurringExpenseService.

Задіяні класи: RecurringExpenseService.

Процес:

1. Користувач правильно заповнює форму створення витрати з прапорцем recurring (в полі recurring = true) і додатковими полями дат, натискає зберегти, надсилає POST /api/expenses.
2. Створюється екземпляр RecurringExpense та список RecurringExpenseSplit з відповідними частками.
3. Планувальник (@Scheduled) щодня перевіряє дату nextOccurrence і створює Expense та список ExpenseSplit.

**5.Автоматичне обчислення боргів та залишків**

Використані об'єкти: ExpenseSplit, DebtDTO, внутрішні map-структури.

Задіяні класи: DebtService, DebtSimplifierService, DebtController.

Процес:

1. Метод simplifyDebts(List<ExpenseSplit> splits, List<Expense> expenses, List<Payment> payments) викликається на бекенді. Де проходимося по всім витратам групи: у платника додаємо до балансу повну суму, а в учасника віднімаємо його частку (split).
2. Обчислюється, хто кому скільки винен: якщо баланс від'ємний - він винен, якщо позитивний - йому винні.
3. Будується список об'єктів DebtDTO
4. Отримавши виклик GET /api/groups/{groupId}/debts/raw, сервер виконує debtService.calculateRawDebts(groupId) і повертає список боргів.

**6.Перегляд індивідуального балансу учасника**

Використані об'єкти: User, Expense, ExpenseSplit, UserBalanceDTO.

Задіяні класи: UserService, UserController.

Процес:

1. Користувач відкриває сторінку групи і цим надсилає запит GET /api/users/me/balance, який повертає скільки користувач витратив, скільки винен і скільки йому винні.
2. Баланс розраховується в UserController.

**7.Спрощення боргів**

Використані об'єкти: Debt, Payment, UserBalanceDTO / DebtDTO.

Задіяні класи: DebtSimplifierService, DebtController.

Процес:

1. Метод simplifyDebts() застосовує жадібний алгоритм (див пункт 2.2)
2. Створюється об'єкт DebtDTO(from, to, amount)
3. Повертається мінімізована кількість переказів на сторінку групи.

**8.Аналітика та статистика витрат**

8.1. Групова аналітика

Використані об'єкти: Expense, User, Group, GroupStatsDTO.

Задіяні класи: GroupService, GroupController.

Процес:

1. Користувач відкриває сторінку групи і цим надсилає запит GET /api/groups/{groupId}/stats
2. Методом getGroupStats(Long groupId) створюється об'єкт GroupStatsDTO зі статистикою.
3. Повертається як JSON-відповідь на сторінку групи.

8.2. Персональна аналітика

Використані об'єкти: Expense, ExpenseSplit, Payment, UserStatsDTO.

Задіяні класи: UserService, ExpenseService, UserController.

Процес:

1. Користувач відкриває сторінку профілю і цим надсилає запит GET api/expenses/my/statistics.
2. Методом getExpenseStatsByUser(Long userId) створюємо Map<String, Object> та методом getMyExpenses створюємо ExpenseSummaryDTO.
3. Повертається як JSON-відповідь на сторінку профілю користувача.

3.6 Опис файлів даних та інтерфейсу програми

3.6.1 Опис бази даних.

Розроблюваний вебзастосунок MoneyTrecker використовує для зберігання даних реляційну базу даних PostgreSQL (pgadmin4-9.0-x64). Повна схема наведена в Додатоку Б. Схема бази даних.

Основні таблиці бази даних:

1. users - таблиця зберігає основні відомості про зареєстрованих користувачів.
   Первинний ключ: id.

– id - унікальний ідентифікатор користувача
– first_name - ім’я користувача
– last_name - прізвище користувача
– email - унікальний логін/пошта
– password_hash - хешований пароль
– created_at - дата та час реєстрації
– photo_url - URL до аватарки (необов’язкове поле)

2. groups - таблиця, що містить інформацію про створені групи користувачів.
   Первинний ключ: id.

– id - унікальний ідентифікатор групи
– name - назва групи
– created_by - зовнішній ключ на users.id (власник групи)
– is_deleted - прапорець лагідного видалення
– created_at - дата створення групи

3. group_members - таблиця зв'язку "багато-до-багатьох" між користувачами та групами.
   Первинний ключ: id.

– id - унікальний ідентифікатор членства
– group_id - зовнішній ключ на groups.id
– user_id - зовнішній ключ на users.id

4. expenses - таблиця, яка містить інформацію про окремі витрати.
   Первинний ключ: id.

– id - унікальний ідентифікатор витрати
– group_id - зовнішній ключ на groups.id
– payer_id - зовнішній ключ на users.id
– title - назва витрати
– amount - загальна сума витрати
– created_at - дата створення витрати

5. expense_splits - таблиця з частками витрат між учасниками. Первинний ключ: id.
– id - унікальний ідентифікатор частки
– expense_id - зовнішній ключ на expenses.id
– user_id - зовнішній ключ на users.id
– share_type - тип розподілу: equal, manual, percent
– share_value - числове значення частки

6. recurring_expenses - таблиця з інформацією про повторювані витрати. Первинний ключ: id.
– id - унікальний ідентифікатор
– group_id - зовнішній ключ на groups.id
– payer_id - зовнішній ключ на users.id
– title - назва витрати
– amount - сума
– recurrence_period - період повторення: day, week, month
– start_date / end_date - діапазон дії
– next_occurrence - наступна дата генерації витрати
– last_generated - дата останньої генерації
– created_at - дата створення

7. recurring_expense_splits - таблиця з частками повторюваних витрат. Первинний ключ: id.

– id - унікальний ідентифікатор
– recurring_expense_id - зовнішній ключ на recurring_expenses.id
– user_id - зовнішній ключ на users.id
– share_type - тип частки
– share_value - числове значення частки

8. payments - таблиця, що зберігає інформацію про здійснені перекази між користувачами для погашення боргів. Первинний ключ: id.

– id - унікальний ідентифікатор платежу
– group_id - зовнішній ключ на groups.id
– payer_id - зовнішній ключ на users.id
– receiver_id - зовнішній ключ на users.id
– amount - сума
– created_at - дата й час переказу

Опис зв’язків в базі данних:

1. users (користувачі)

– 1:N → expenses: один користувач може бути платником для багатьох витрат (payer_id).
– 1:N → payments (як payer): користувач може здійснювати багато платежів.
– 1:N → payments (як receiver): користувач може отримувати багато платежів.
– 1:N → expense_splits: один користувач може бути учасником багатьох витрат.

– 1:N → recurring_expenses (payer): один користувач може бути платником для багатьох повторюваних витрат.
– 1:N → recurring_expense_splits: один користувач може бути учасником багатьох повторюваних витрат.
– N:M через group_members → groups: користувач може входити до багатьох груп.

2. groups (групи)
– 1:N → expenses: одна група може мати багато витрат.
– 1:N → recurring_expenses: одна група може мати багато повторюваних витрат.
– 1:N → payments: одна група може мати багато платежів.
– 1:N → group_members: одна група може мати багато учасників.
– N:M через group_members → users: одна група включає більше одного користувача.

3. expenses (витрати)
– N:1 → groups: кожна витрата належить до однієї групи (group_id).
– N:1 → users (payer): кожна витрата має одного платника (payer_id).
– 1:N → expense_splits: одна витрата розподіляється між багатьма учасниками.

4. expense_splits (частки витрати)
– N:1 → expenses: кожна частка належить до конкретної витрати.
– N:1 → users: кожна частка прив’язана до певного користувача.

5. recurring_expenses (повторювані витрати)
– N:1 → groups: кожна повторювана витрата належить до певної групи.

– N:1 → users (payer): кожна повторювана витрата має одного платника (payer_id).
– 1:N → recurring_expense_splits: одна повторювана витрата розподіляється між багатьма учасниками.

6. recurring_expense_splits (частки регулярних витрат)
– N:1 → recurring_expenses: кожна частка прив’язана до певної регулярної витрати.
– N:1 → users: кожна частка належить конкретному користувачу.

7. payments (оплати між користувачами)
– N:1 → groups: кожен платіж належить до певної групи.
– N:1 → users (payer): кожен платіж має одного платника.
– N:1 → users (receiver): кожен платіж має одного отримувача.

8. group_members (учасники групи)
– N:1 → groups: кожен запис належить до певної групи.
– N:1 → users: кожен запис відповідає одному користувачу.

Доступ до таблиць бази даних у застосунку реалізовано через репозиторії, що розширюють інтерфейси JpaRepository та CrudRepository, які надає фреймворк Spring Data JPA. Це дозволяє уникати написання SQL-запитів вручну у тривіальних випадках (findById(), save(), findAll(), ...). У разі потреби написання складних запитів або приєднання кількох таблиць застосовується анотація *@Query*, яка дозволяє вказувати SQL безпосередньо в методах репозиторію.

Перед надсиланням даних до бази на боці клієнта реалізовано перевірку введених значень. Використовуються регулярні вирази для валідації полів, зокрема:

– ^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$ - валідація електронної пошти.
– ^(?=.*\\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[a-zA-Z]).{8,}$ - перевірка паролю на складність.

3.6.2 Опис інтерфейсу програми

Інтерфейс веб-застосування MoneyTrecker реалізовано як адаптивну клієнтську частину, що забезпечує зручну взаємодію з користувачем у браузері. Його структура побудована за модульним підходом: кожна сторінка має свій HTML-шаблон (Thymeleaf), окремі стилі CSS та логіку на JavaScript. Передача даних до і від сервера відбувається через REST API, з використанням асинхронних запитів fetch, а виведення - через рендеринг DOM-елементів.

1. HTML-шаблони:

– navbar.html - спільний компонент (фрагмент), який вставляється у всі сторінки як верхнє меню навігації.
– dashboard.html - загальна сторінка входу в систему.
– group.html - головна сторінка групи, що містить загальну інформацію про групу,з переліком транзакцій та боргів.
– login.html / register.html - сторінки автентифікації.
– profile.html - персональна сторінка користувача з відображенням особистої інформації, його груп, статистики.
– support.html, about.html - статичні сторінки з описом сервісу та зворотним зв’язком.

2. Кожна HTML-сторінка має власний CSS-файл, який забезпечує адаптивність, тобто масштабування під різні розміри екрану, зберігаючи читабельність (Grid, media queries) та єдиний стиль:

– aboutStyles.css
– dashboardStyles.css
– groupStyles.css
– navbar.css

- profileStyles.css
- registerStyles.css
- supportStyles.css

3. JavaScript-модулі

- api.js - універсальна функція fetchJSON для надсилання GET/POST/DELETE-запитів з обробкою токена авторизації та JSON-відповідей. Використовується іншими модулями.
- navbar.js - керування меню.
- login.js - логіка авторизації: збирання форми, відправка POST-запиту на /api/auth/login.
- register.js - валідація даних реєстрації, запит POST /api/auth/register.
- group.js - відповідає за отримання транзакцій і боргів, виклик модального вікна та обробку введених витрат, показ групової статистики.
- profile.js - відображення інформації прокористувача, його статистику по витратах.

4. Інтерфейс реалізовано двомовним: українська та англійська мови. Вся текстова інформація винесена у файли ресурсів: messages.properties, messages_uk.properties, messages_en.properties. У шаблонах використовується синтаксис th:text="#{ключ}" для автоматичної локалізації залежно від мови.

### 3.7 Тестування програми і результати її виконання

Тестування вебзастосунку MoneyTrecker здійснювалося вручну у браузері Google Chrome шляхом взаємодії з інтерфейсом у звичайному користувацькому режимі. Основна мета тестування - перевірити коректність реалізації функціональних вимог, зазначених у пункті 1.3, зокрема облік витрат, управління групами, реєстрацію користувачів, роботу повторюваних

транзакцій, спрощення боргів, а також загальну стабільність та зручність роботи інтерфейсу.

1. Головна сторінка та навігація

Сценарій: Перехід на головну сторінку як авторизованим так і неавторизованим користувачем.

Очікування: Відображення головної сторінки авторизованому та неавторизованому користувачу сторінок “Головна”, “Про нас”, “Підтримка”.

Результат: Навігація доступна, сторінки відображаються (рисунки А1-А6).

2. Реєстрація та вхід нового користувача

Сценарій: Заповнення форми реєстрації з усіма правильними полями.

Очікування: Створення нового облікового запису, вхід до особистого кабінету.

Результат: Користувача зареєстровано, перенаправлення на сторінку входу, після входу користувача направлено до сторінки особистого кабінету (рисунки А.7-А.9).

Сценарій: Введення неправильного email або короткого пароля.

Очікування: Вивід повідомлень про помилку.

Результат: Працює клієнтська валідація та підказки (рисунки А.10-А.11).

3. Створення та видалення групи

Сценарій: Створення нової групи через форму.

Очікування: Група додається, користувач позначається як власник (наявна плашка видалення).

Результат: Створення працює, група відображається у списку (рисунки А.12 - А14).

Сценарій: Видалення групи.

Очікування: Група видаляється, якщо в ній немає боргів.

Результат:  Видалення працює, група зникає зі списку(рисунки А.15-А.16).

4. Додавання транзакції

Сценарій: Створення витрати з типом поділу "equal", "manual" або "percent".

Очікування: Витрата ділиться рівномірно між усіма учасниками.

Результат: Сума розрахована правильно, транзакція відображається (рисунки А.17-А.20).

5. Повторювані витрати

Сценарій: Створення витрати з позначкою "повторювана", заповнення дати, періоду.

Очікування: Збереження регулярної витрати, автоматичне створення нової транзакції.

Результат: Витрата створена, генерація працює згідно планувальника (рисунки А.21-А.22).

6. Пропонування витрат

Сценарій: Наявність списку пропозицій користувачу по назві витрати.

Очікування: Відображення списку пропозицій.

Результат: Відображається список пропозицій. (рисунок А.23).

7. Спрощення боргів

Сценарій: Наявність відображення всіх боргів у групі та спрощеної схеми.

Очікування: Зменшення кількості транзакцій між учасниками.

Результат: кількості транзакцій для боргів зменшуються(рисунок А.24).

8. Оплата боргу

Сценарій: Оплата боргу.

Очікування: Зникнення боргу користувача.

Результат: Погашення боргу цього користувача (рисунок А.25).

9. Аналітика

Сценарій: Перехід до сторінки групи та профілю.

Очікування: Відображення аналітики по витратах, графіки, статистика.

Результат: Дані відображаються правильно, графіки завантажуються (рисунки А.26-А.27).

10. Двомовність

Сценарій: Зміна мови інтерфейсу користувачем.

Очікування: Відображення вмісту сайту іншою мовою (українською або англійською).

Результат: Дані відображаються згідно перекладу (рисунки А.28-А.31).

Висновки

Під час виконання курсової роботи було проаналізовано проблему обліку спільних витрат у малих соціальних групах. Вивчено функціональність наявних рішень на кшталт Splitwise, визначено їхні переваги та обмеження.

З урахуванням виявлених вимог було реалізовано повноцінний вебзастосунок MoneyTrecker, побудований за принципами клієнт-серверної архітектури з використанням стеку Spring Boot (бекенд), PostgreSQL (БД), HTML/CSS/JavaScript (фронтенд). З погляду користувача, система має інтуїтивно зрозумілий інтерфейс, підтримує українську мову, забезпечує адаптивність до різних розмірів екранів і забезпечує зручну взаємодію навіть для людей без технічної підготовки. Всі дані захищено за допомогою JWT-аутентифікації, паролі зберігаються у хешованому вигляді, а доступ до функцій контролюється через Spring Security.

Практична реалізація підтвердила, що розроблений застосунок здатен вирішити поставлені завдання й може бути використаний як реальний інструмент для фінансового планування в побуті.

У майбутньому можливе розширення системи - інтеграція з платіжними сервісами, мобільна версія, інтелектуальний аналіз витрат, автоматичні рекомендації щодо оптимізації бюджету.

Таким чином, розроблений веб-застосування не лише відповідає поставленим вимогам, а й демонструє можливість створення практичного, зручного та корисного ІТ-рішення, орієнтованого на реальні потреби людей.

Список використаної літератури


1. «Splitwise». Офіційний вебсайт сервісу для розподілу витрат [Електронний ресурс] - https://www.splitwise.com/
2. «Settle Up». Застосунок для обліку витрат [Електронний ресурс] - https://settleup.io/
3. «Kittysplit». Вебсервіс для поділу витрат [Електронний ресурс] - https://www.kittysplit.com/
4. Algorithm Behind Splitwise's Debt Simplification Feature [Електронний ресурс] - https://medium.com/@mithunmk93/algorithm-behind-splitwises-debt-simplification-feature-8ac485e97688
5. Verhoeff, T. Settling multiple debts efficiently: an invitation to computing science // Informatics in Education. - 2004. - Т. 3, №1. - С. 105–126.
6. Kumar, R., & Kleinberg, J. Minimizing Cash Flow in Payment Networks // Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. - 2006. - С. 389–398.
7. Understanding JSON Web Tokens [Електронний ресурс] - https://supertokens.com/blog/what-is-jwt
8. IntelliJ IDEA. IntelliJ IDEA - середовище розробки [Електронний ресурс] - Режим доступу: https://www.jetbrains.com/idea/
9. Spring. Spring Boot Reference Documentation [Електронний ресурс] - Режим доступу: https://docs.spring.io/spring-boot/docs/current/reference/html/
10. Spring Data JPA Repository [Електронний ресурс] - https://jcs.ep.jhu.edu/legacy-ejava-springboot/coursedocs/content/html_single/jparepo-notes.html
11. PostgreSQL Global Development Group. PostgreSQL 15 Documentation [Електронний ресурс] - Режим доступу: https://www.postgresql.org/docs/

12. «Web Framework Benchmarks». Порівняння продуктивності фреймворків [Електронний ресурс] - https://www.techempower.com/benchmarks

# Додаток А. Скріншоти тестування системи

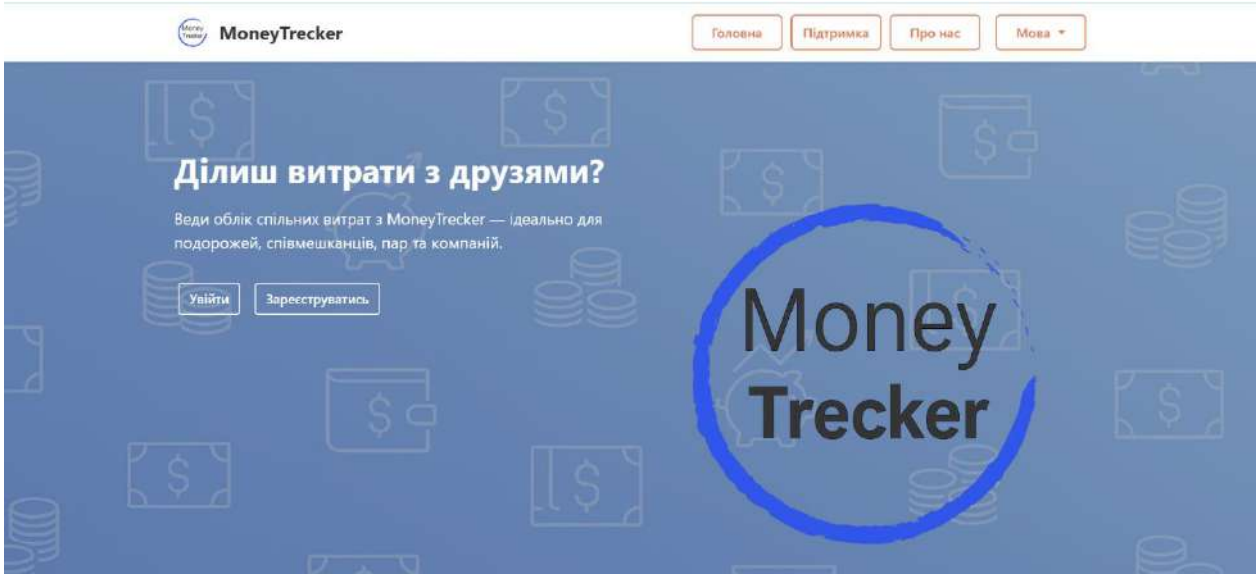


Рисунок А.1 - Головна сторінка для неавторизованого користувача

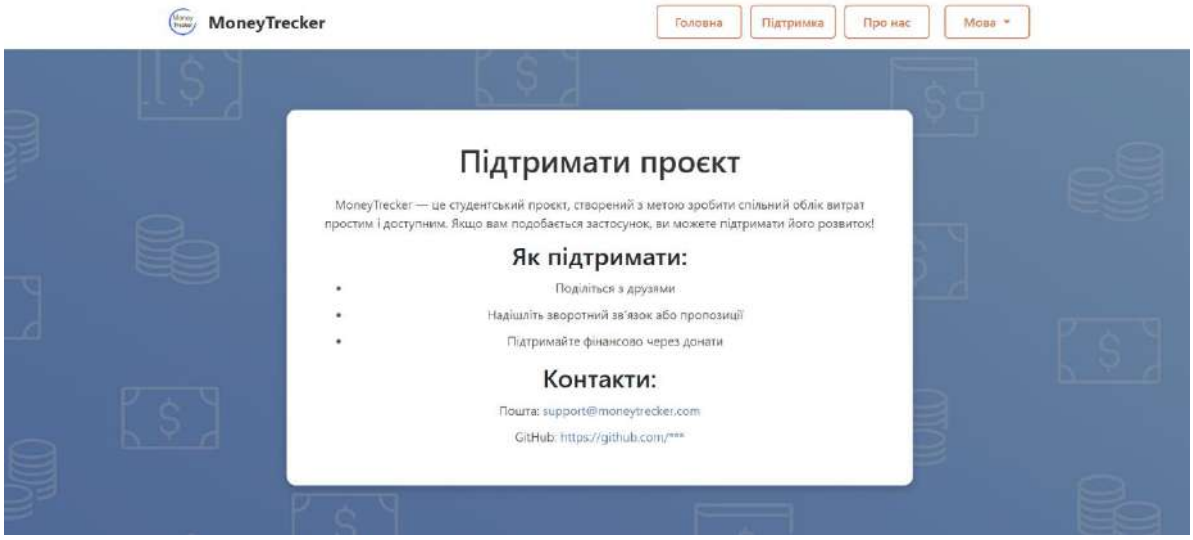


Рисунок А.2 -Сторінка ”Підтримати” для неавторизованого користувача

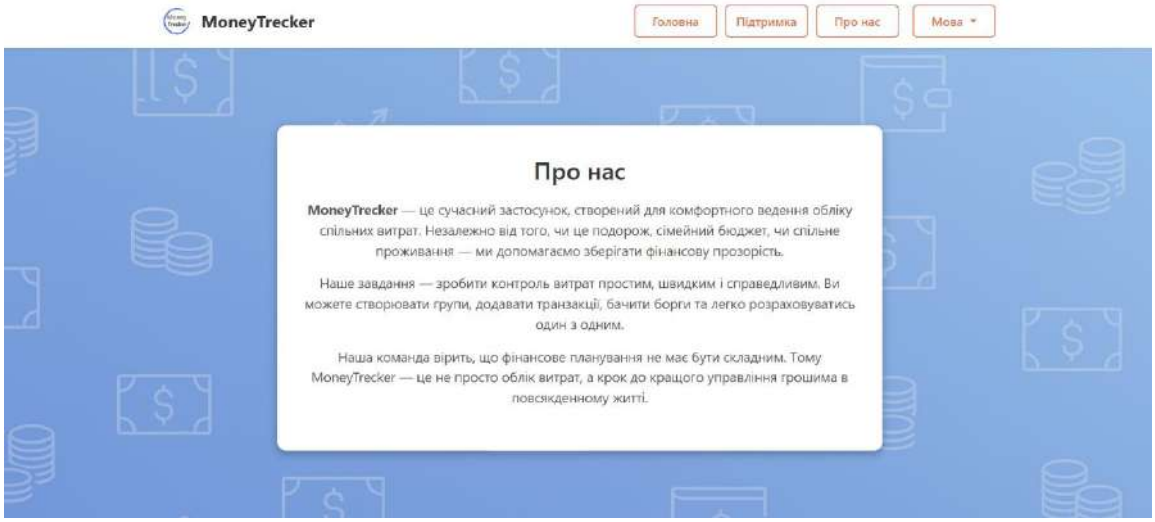


Рисунок А.3 - Сторінка “Про нас” для неавторизованого користувача

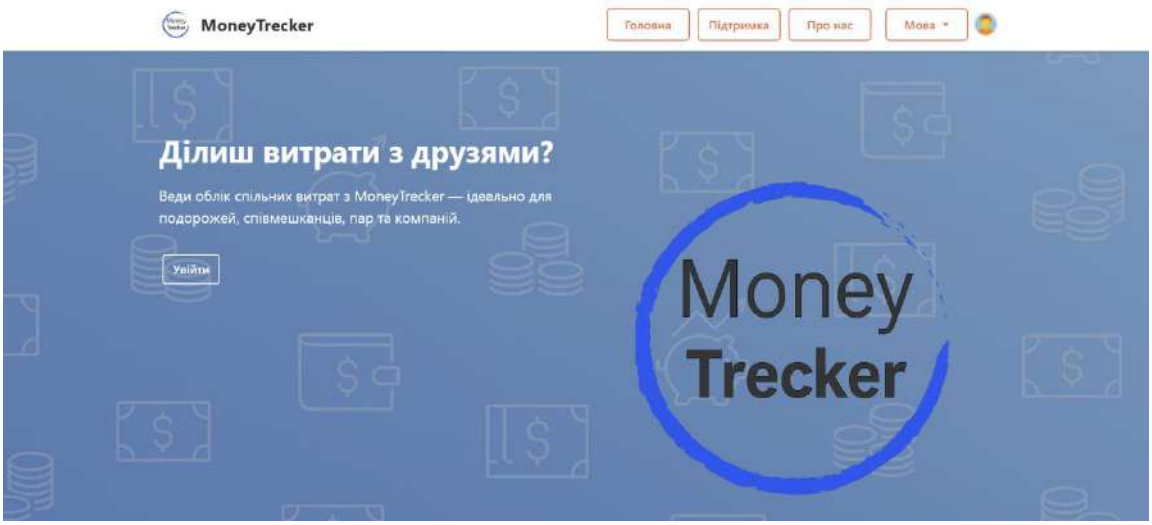


Рисунок А.4 - Головна сторінка для авторизованого користувача

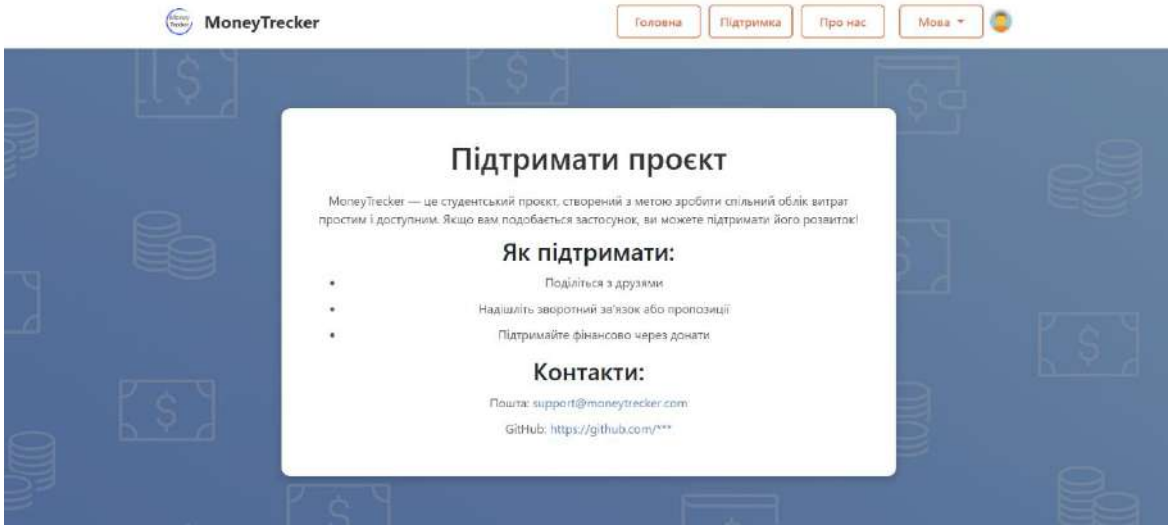


Рисунок А.5 - Сторінка ”Підтримати” для авторизованого користувача

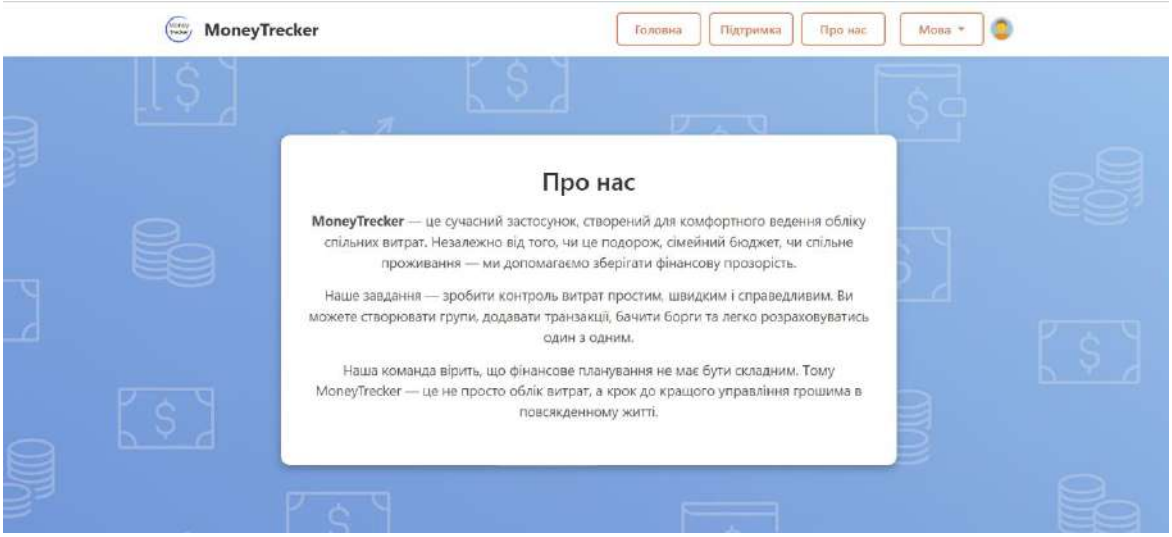


Рисунок А.6 - Сторінка “Про нас” для авторизованого користувача

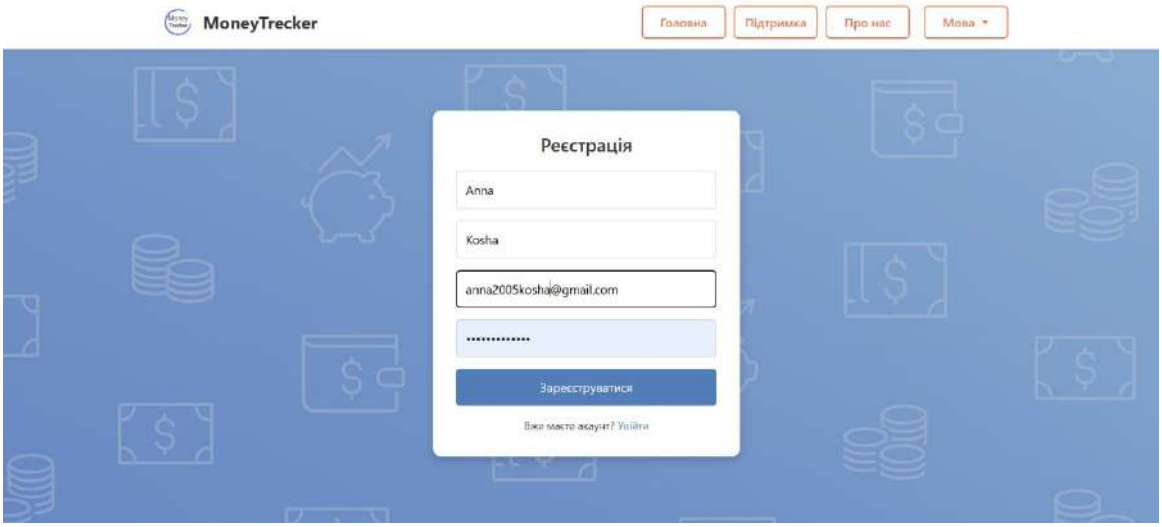


Рисунок А.7 - Сторінка реєстрації


Рисунок А.8 - Сторінка входу

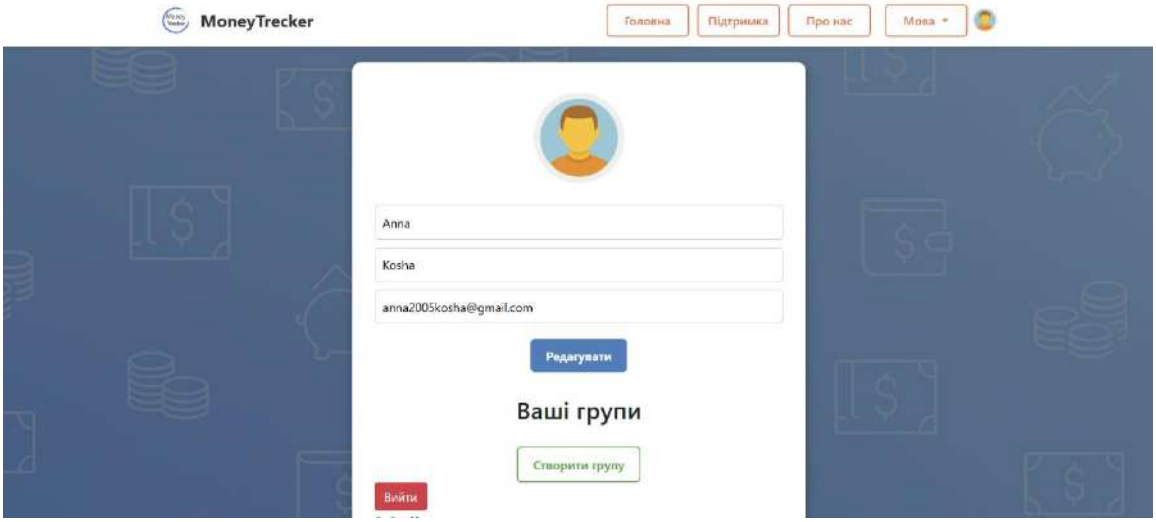


Рисунок А.9 - Сторінка особистого кабінету користувача, що увійшов у систему

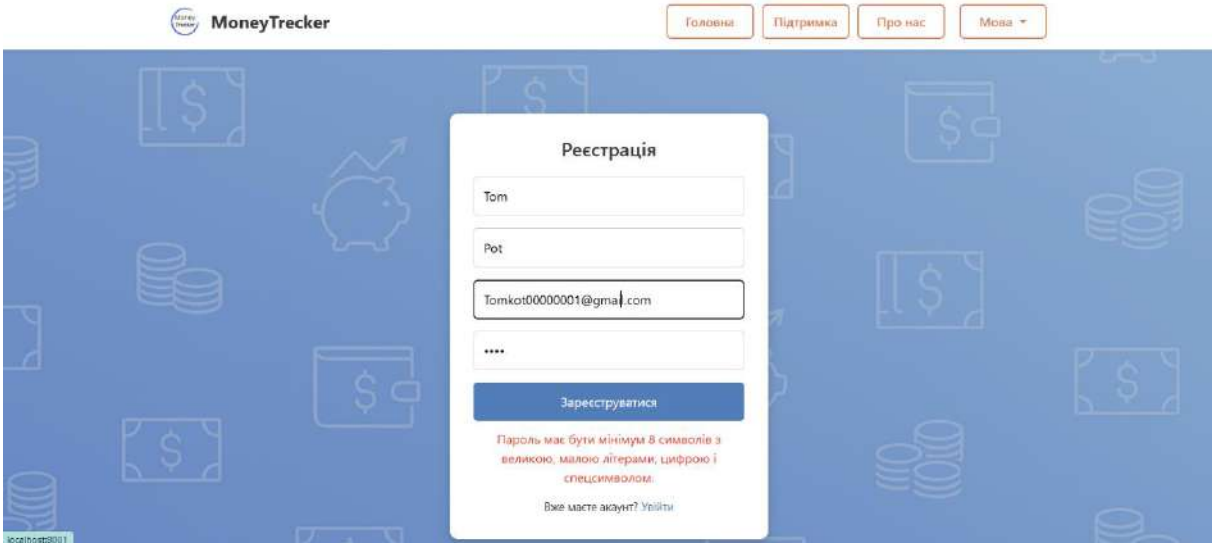


Рисунок А.10 - Сторінка реєстрації з паролем “0000”


Рисунок А.11 - Сторінка входу з неправильними логіном та паролем

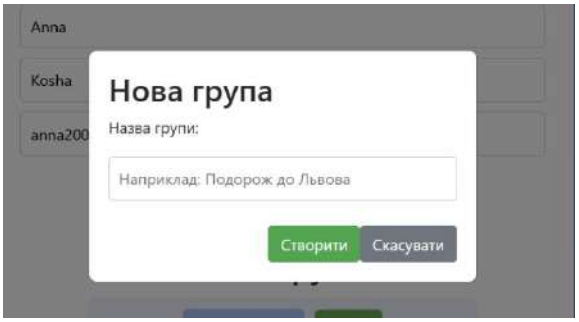


Рисунок А.12 - Форма створення групи

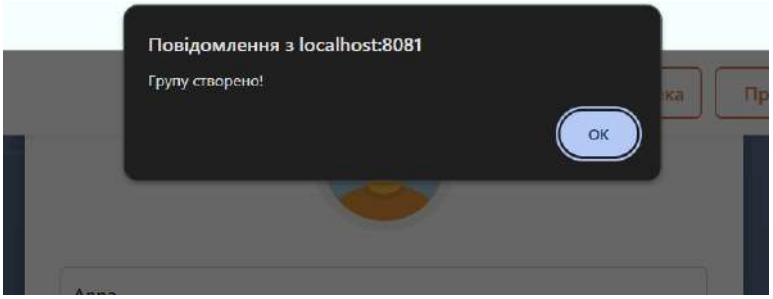


Рисунок А.13 - Повідомлення про створення групи

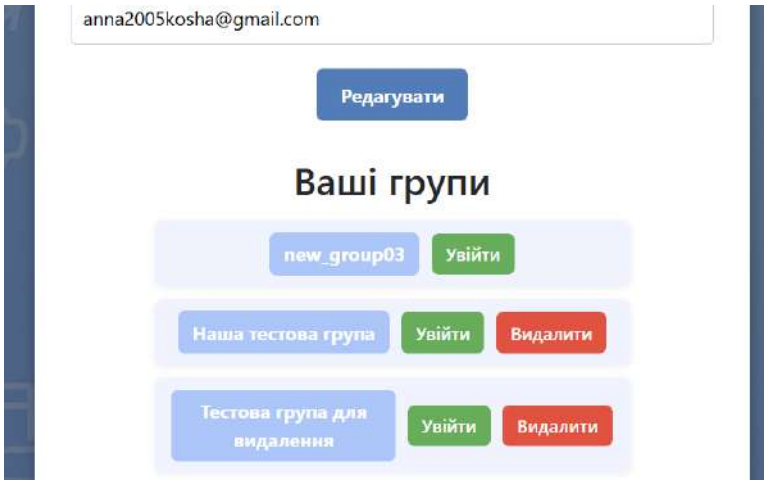


Рисунок А.14- Відображення списку груп

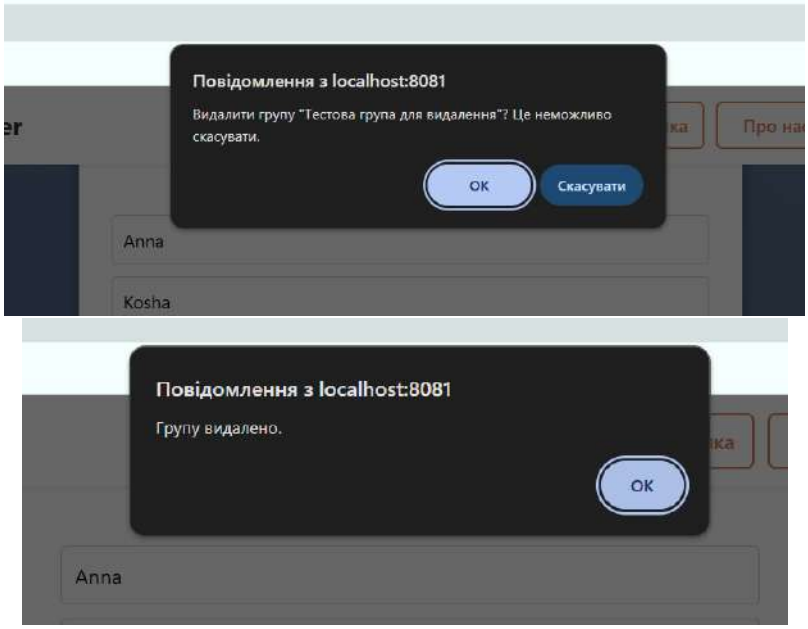


Рисунок А.15 -Попередження та повідомлення про видалення групи

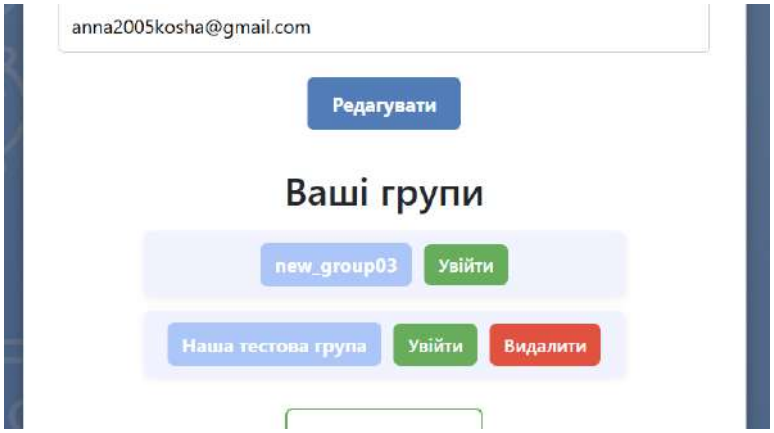


Рисунок А.16 - Список груп після видалення

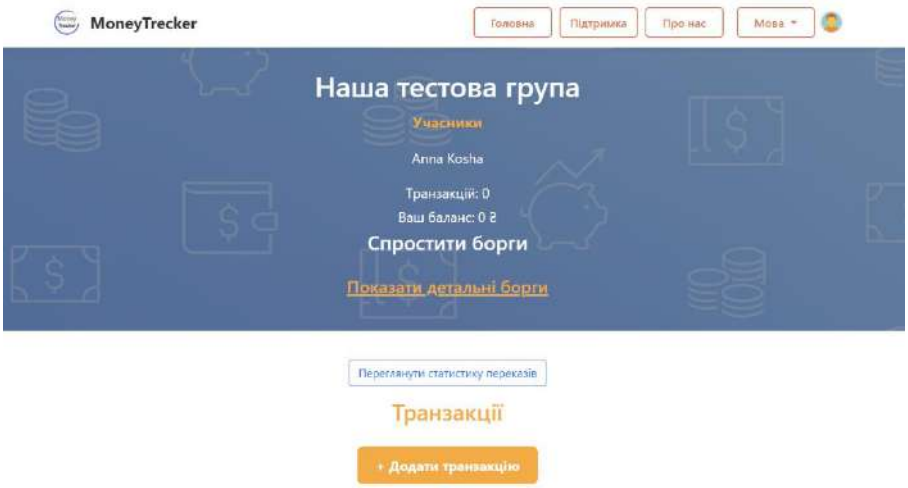


Рисунок А.17 - Сторінка групи

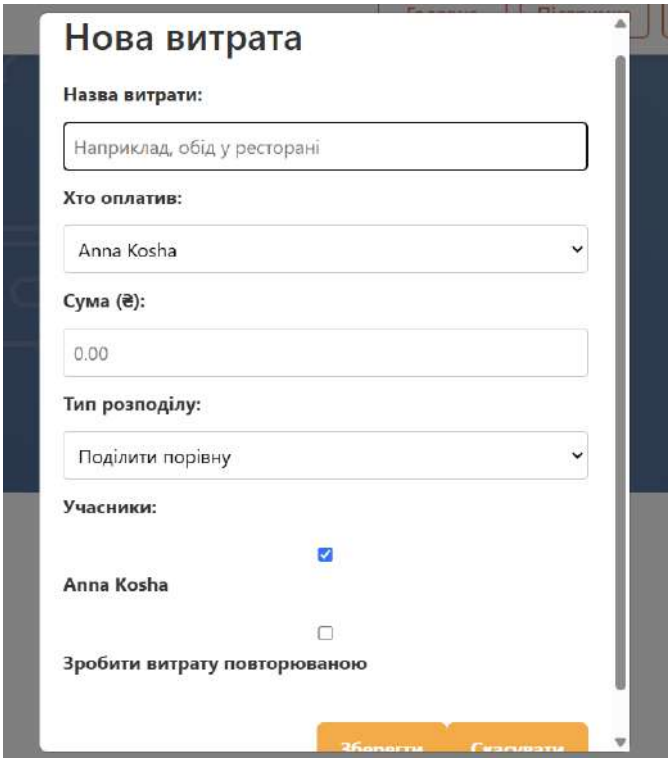


Рисунок А.18 - Форма створення транзакції

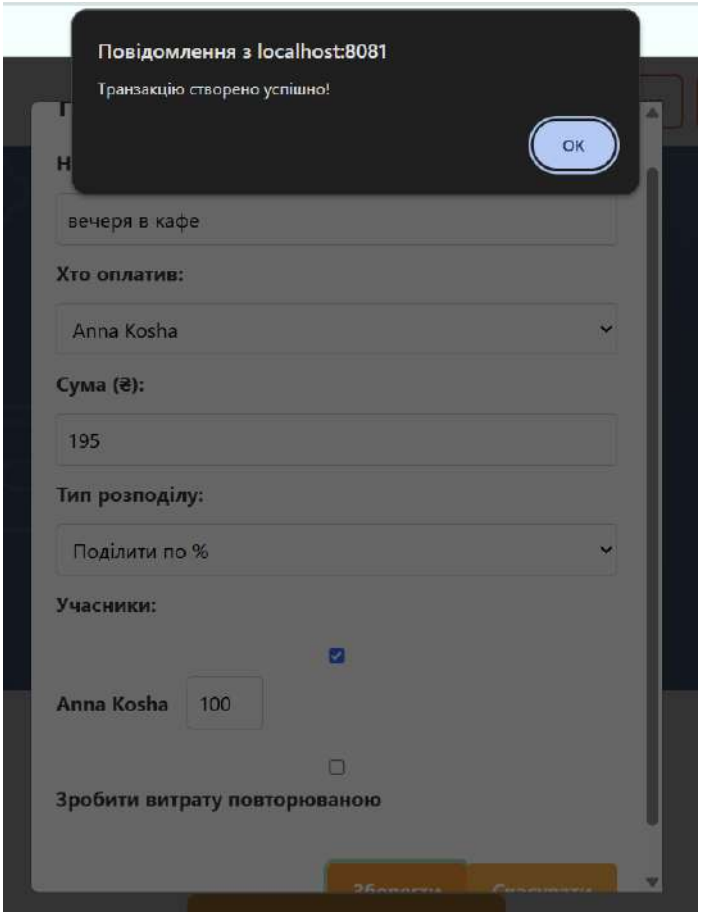


Рисунок А.19 - Створення витрати

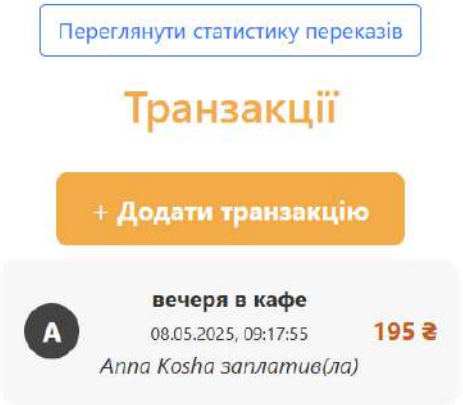


Рисунок А.20 - Список витрат після додавання

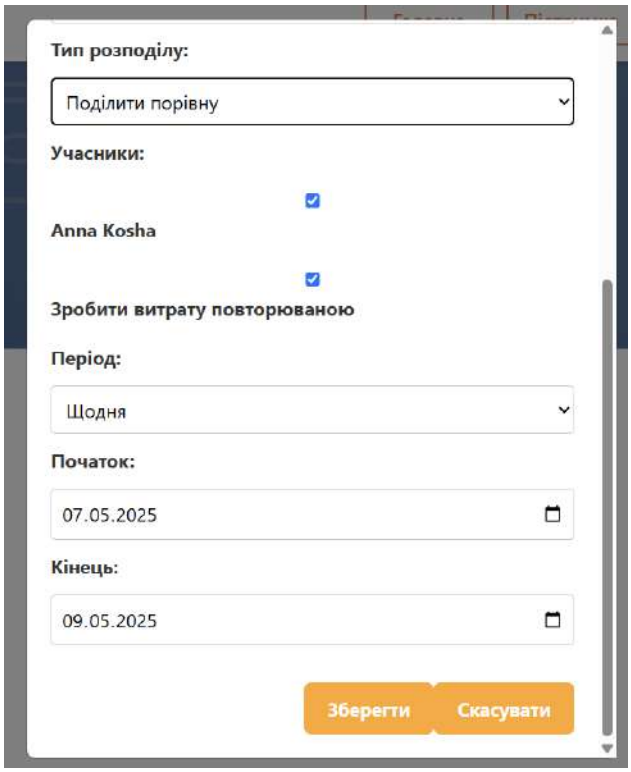


Рисунок А.21 - Створення повторювальної витрати

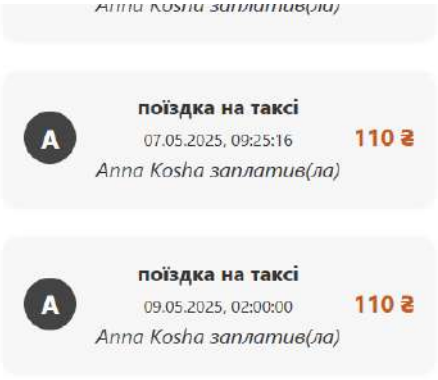


Рисунок А.22 - Відображення повторюваних витрат

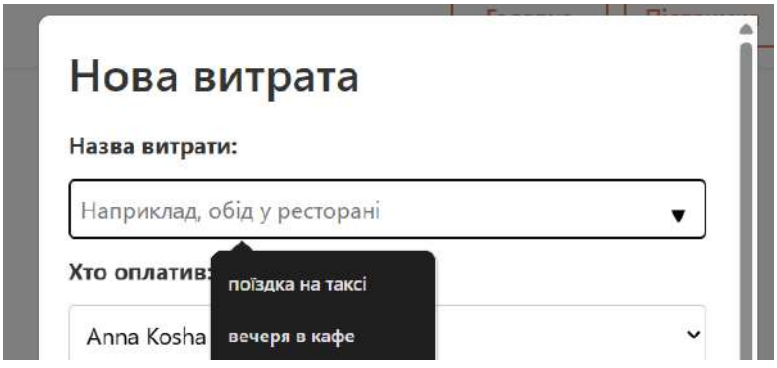


Рисунок А.23 - Список пропозицій по назві витрати


Рисунок А.24 - Зменшений список транзакцій та список боргів

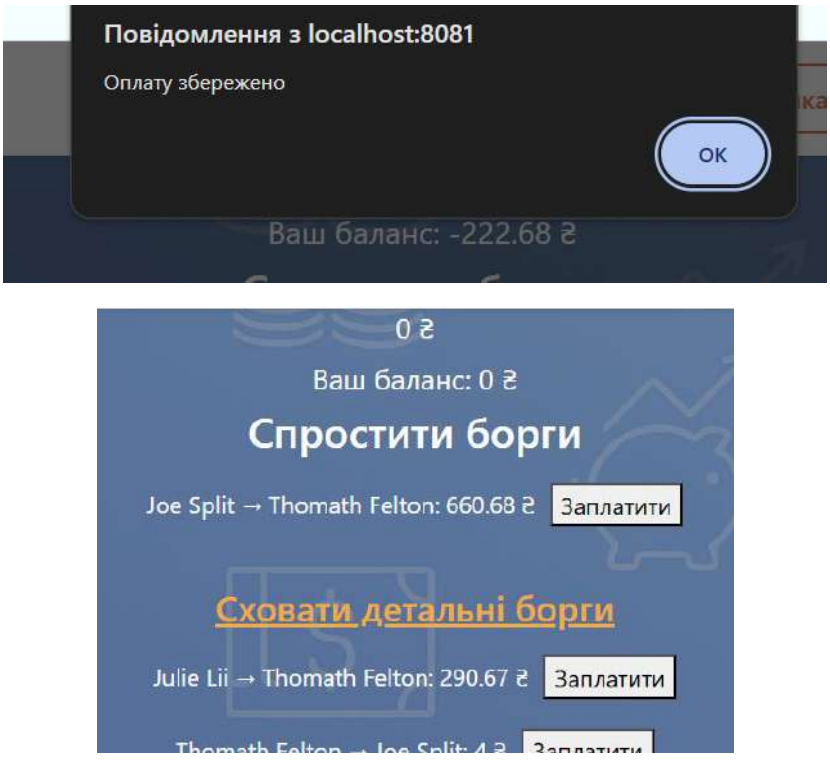


Рисунок А.25 - Сповіщення про оплату боргу користувача Julie Lii та зникнення її зі списку боржників.

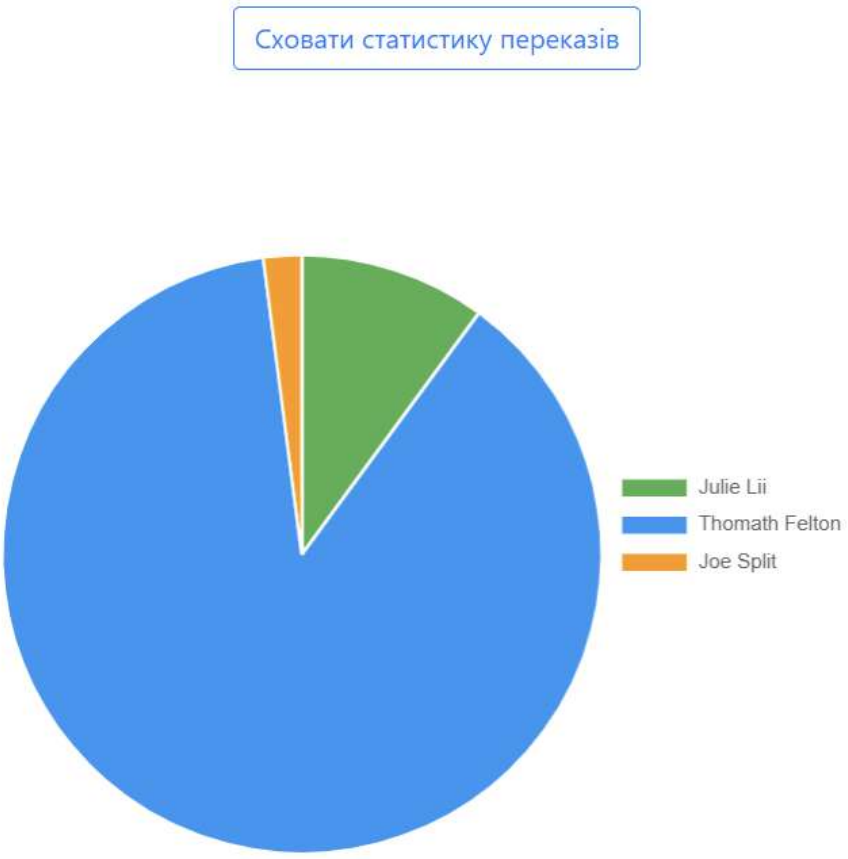


Рисунок А.26 - Статистика за витратами в групі

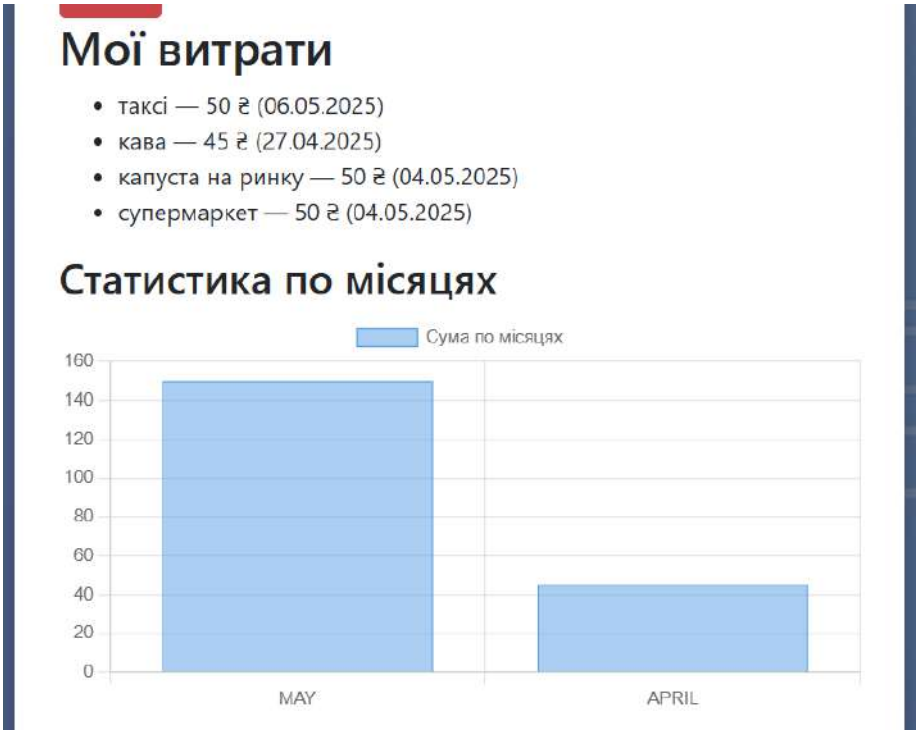


Рисунок А.27 - Особиста статистика витрат користувача

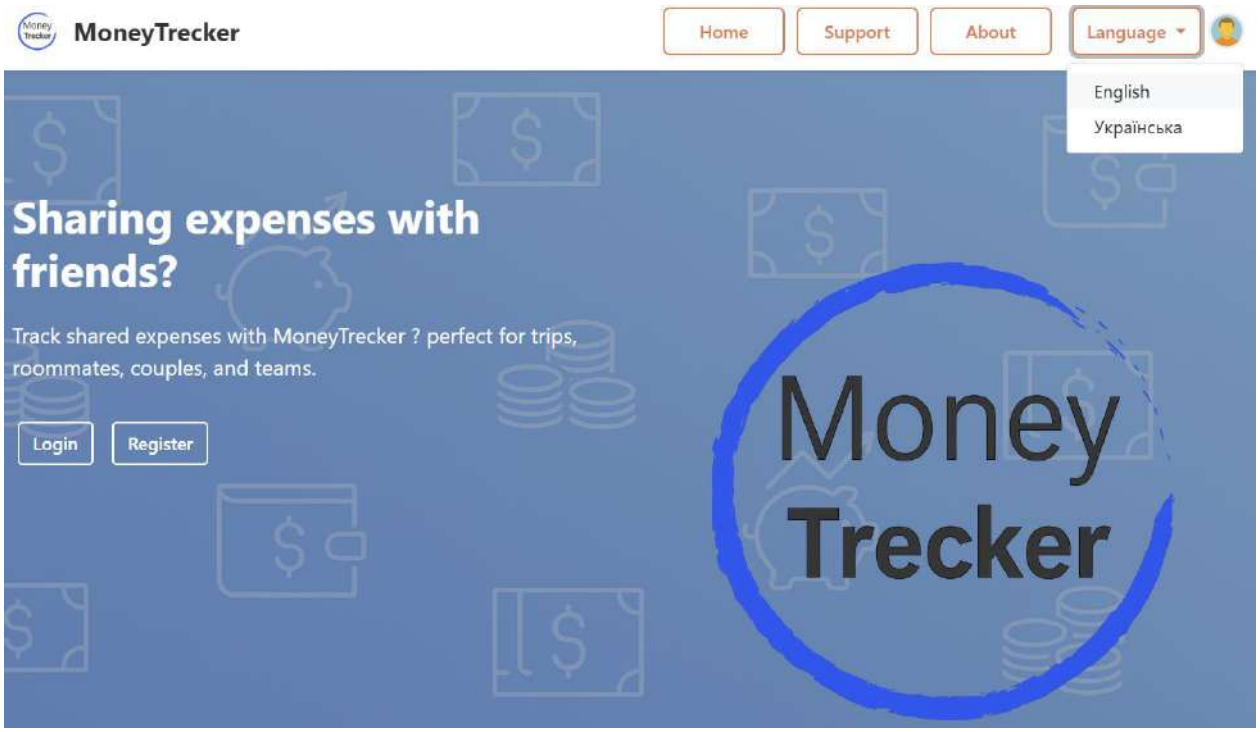


Рисунок А.28 - Головна сторінка англійською мовою

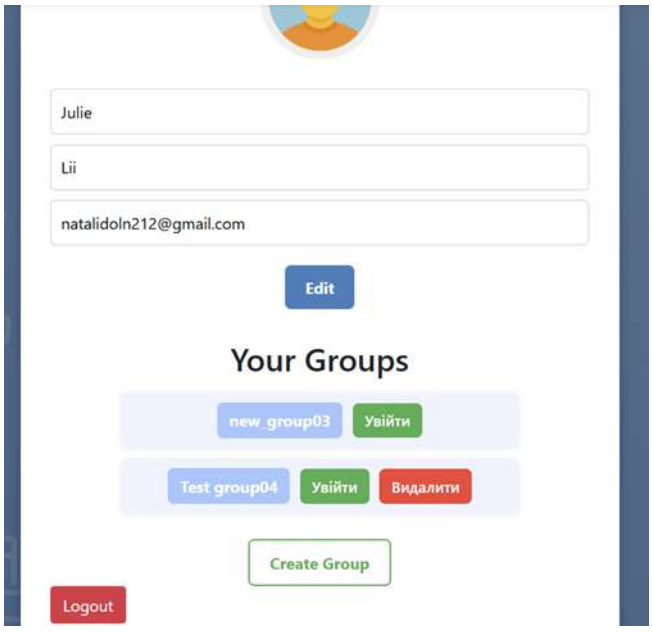


Рисунок А.29 - Частина особистого кабінету користувача англійською мовою

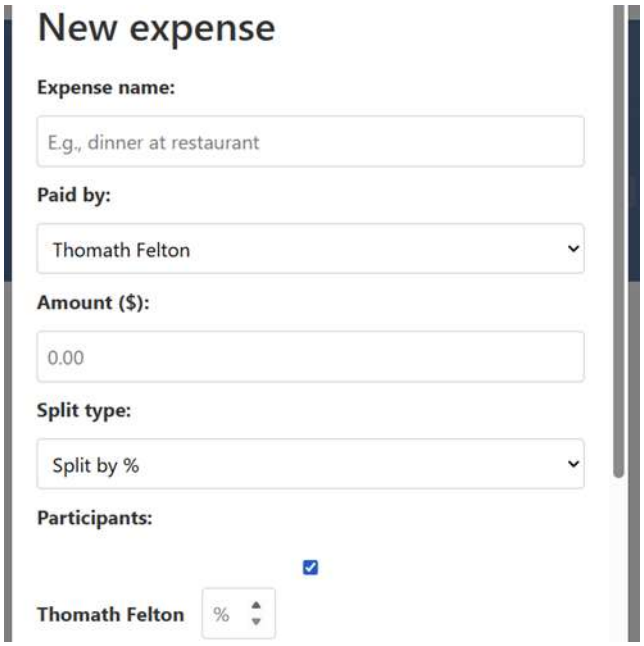


Рисунок А.30 - Частина форми створення витрати англійською мовою

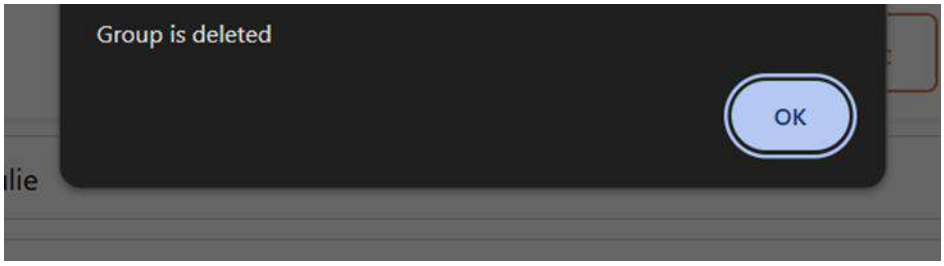


Рисунок А.31 - Приклад вспливаючого повідомлення англійською мовою

Додаток Б. Схема бази даних.

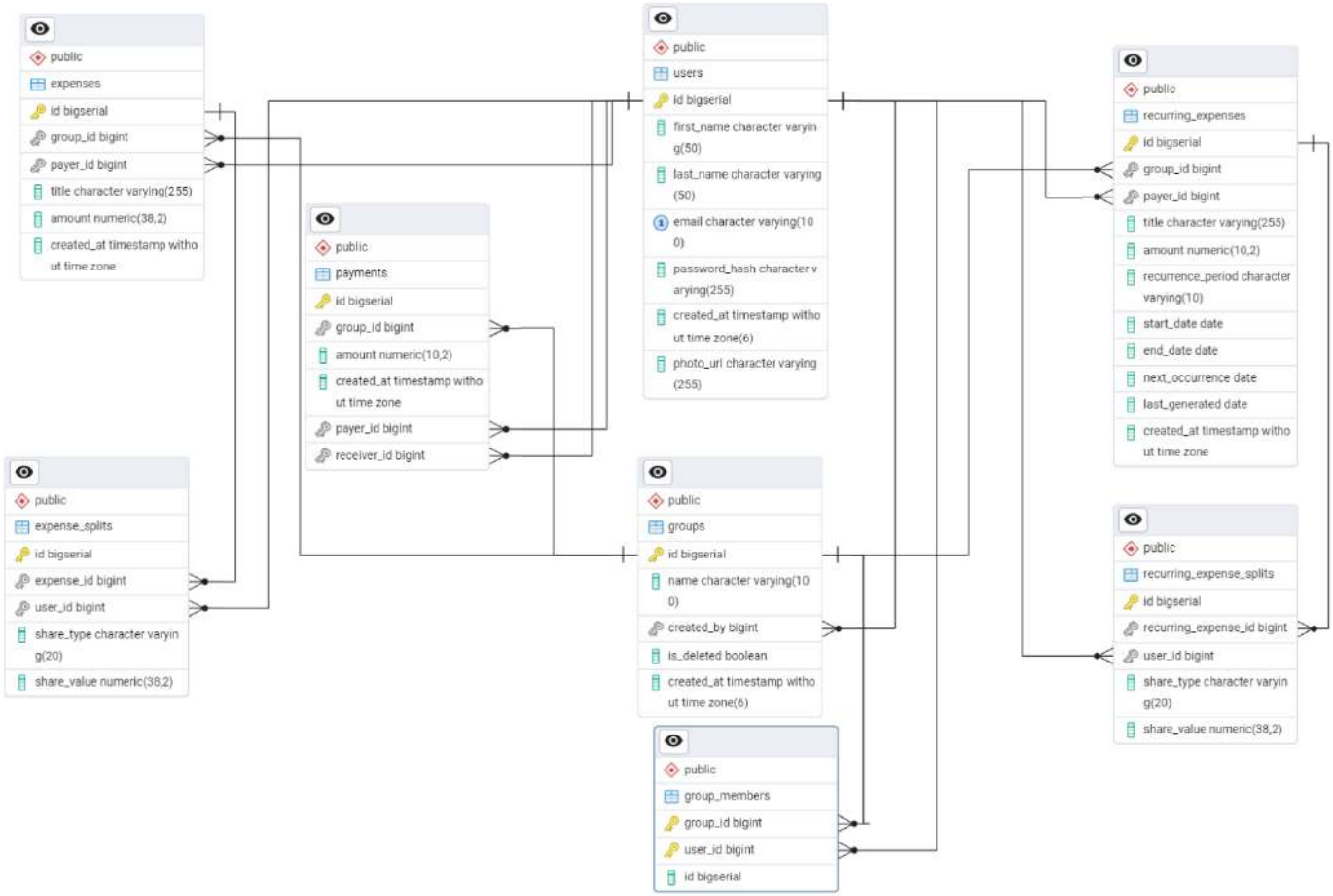


Рисунок Б. Схема бази даних системи

# Додаток В. Серверний код вебзастосунку

## MoneyTreckerApplication

```
@SpringBootApplication
public class MoneyTreckerApplication {

    public static void main(String[] args) {
        SpringApplication.run(MoneyTreckerApplication.class, args);
    }
}
```

## LocaleConfig

```
@Configuration
public class LocaleConfig implements WebMvcConfigurer {

    @Bean
    public LocaleResolver localeResolver() {
        CookieLocaleResolver resolver = new CookieLocaleResolver();
        resolver.setDefaultLocale(new Locale("uk"));
        return resolver;
    }

    @Bean
    public LocaleChangeInterceptor localeChangeInterceptor() {
        LocaleChangeInterceptor interceptor = new LocaleChangeInterceptor();
        interceptor.setParamName("lang");
        return interceptor;
    }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(localeChangeInterceptor());
    }
}
```

## AuthController

```
@RestController
@RequestMapping("/api/auth")
```

```
@RequiredArgsConstructor
public class AuthController {
    private final UserService userService;
    private final JwtService jwtService;
    private final AuthenticationManager authenticationManager;
    private String t(String ua, String en, String lang) {
        return "en".equalsIgnoreCase(lang) ? en : ua;
    }

    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody RegisterRequest request,
                                      @RequestHeader(value = "Accept-Language", defaultValue = "uk") String lang) {
        if (request.getFirstName() == null || request.getFirstName().trim().isEmpty()) {
            return ResponseEntity.badRequest().body(t("Ім'я обов'язкове.", "First name is required.", lang));
        }
        if (request.getEmail() == null || request.getEmail().trim().isEmpty()) {
            return ResponseEntity.badRequest().body(t("Пошта обов'язкова.", "Email is required.", lang));
        }
        if (request.getPassword() == null || request.getPassword().trim().isEmpty()) {
            return ResponseEntity.badRequest().body(t("Пароль обов'язковий.", "Password is required.",
lang));
        }
        if (userService.emailExists(request.getEmail())) {
            return ResponseEntity.badRequest().body(t("Така пошта вже використовується.", "Email is
already in use.", lang));
        }
        if (!isValidPassword(request.getPassword())) {
            return ResponseEntity.badRequest().body(
                    t("Пароль має містити мінімум 8 символів, велику літеру, малу літеру, цифру та
спецсимвол.",
                            "Password must be at least 8 characters, with uppercase, lowercase, number and special
character.",
                            lang)
            );
        }

        userService.registerUser(
                request.getFirstName(),
```

```
            request.getLastName(),
            request.getEmail(),
            request.getPassword()
        );

        return ResponseEntity.ok(t("Реєстрація успішна", "Registration successful", lang));
    }

    @PostMapping("/login")
    public ResponseEntity<AuthResponse> login(@RequestBody LoginRequest request) {
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(request.getEmail(), request.getPassword())
        );

        UserDetails user = (UserDetails) authentication.getPrincipal();
        String token = jwtService.generateToken(user);
        return ResponseEntity.ok(new AuthResponse(token));
    }

    private boolean isValidPassword(String password) {
        String passwordPattern = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-
z\\d@$!%*?&]{8,}$";
        return password.matches(passwordPattern);
    }
}
```

## DebtController

```
@RestController
@RequestMapping("/api/groups/{groupId}/debts")
@RequiredArgsConstructor
public class DebtController {

    private final ExpenseRepository expenseRepository;
    private final ExpenseSplitRepository expenseSplitRepository;
    private final PaymentRepository paymentRepository;
    private final GroupRepository groupRepository;
    private final GroupMemberRepository groupMemberRepository;
    private final DebtService debtService;
```

```
    private final DebtSimplifierService debtSimplifierService;
    private final UserService userService;

    private boolean isUserMember(Long groupId, String email) {
        User user = userService.findByEmail(email).orElseThrow();
        Group group = groupRepository.findById(groupId).orElseThrow();
        return groupMemberRepository.existsByGroupAndUser(group, user);
    }

    @GetMapping("/raw")
    public ResponseEntity<List<DebtDTO>> getDebts(
            @PathVariable Long groupId,
            @AuthenticationPrincipal UserDetails userDetails) {

        if (!isUserMember(groupId, userDetails.getUsername())) {
            return ResponseEntity.status(403).build();
        }

        List<DebtDTO> result = debtService.calculateDebtsByGroup(groupId);
        return ResponseEntity.ok(result);
    }

    @GetMapping("/simplified")
    public ResponseEntity<List<DebtDTO>> getSimplified(
            @PathVariable Long groupId,
            @AuthenticationPrincipal UserDetails userDetails) {

        if (!isUserMember(groupId, userDetails.getUsername())) {
            return ResponseEntity.status(403).build();
        }

        List<DebtDTO> raw = debtService.calculateDebtsByGroup(groupId);
        List<ExpenseSplit> splits = expenseSplitRepository.findByExpense_Group_Id(groupId);
        List<Expense> expenses = expenseRepository.findByGroupId(groupId);
        List<Payment> payments = paymentRepository.findByGroupId(groupId);

        List<DebtSimplifierService.DebtRecord> simplifiedRecords =
debtSimplifierService.simplifyDebts(splits, expenses, payments);
```

```
        List<DebtDTO> simplified = simplifiedRecords.stream()
                .map(r -> new DebtDTO(r.fromUserId(), r.toUserId(), r.amount()))
                .toList();
        return ResponseEntity.ok(simplified);
    }

    @GetMapping("/simplified-with-payments")
    public ResponseEntity<List<DebtSimplifierService.DebtRecord>> getSimplifiedDebts(
            @PathVariable Long groupId,
            @AuthenticationPrincipal UserDetails userDetails) {

        if (!isUserMember(groupId, userDetails.getUsername())) {
            return ResponseEntity.status(403).build();
        }

        List<Expense> expenses = expenseRepository.findByGroupId(groupId);
        List<ExpenseSplit> splits = expenseSplitRepository.findByExpense_Group_Id(groupId);
        List<Payment> payments = paymentRepository.findByGroupId(groupId);

        List<DebtSimplifierService.DebtRecord> simplified = debtSimplifierService.simplifyDebts(splits,
expenses, payments);
        return ResponseEntity.ok(simplified);
    }
}
```

## ExpenseController

```
@RestController
@RequestMapping("/api/expenses")
@RequiredArgsConstructor
public class ExpenseController {private final DebtService debtService;
    private final ExpenseService expenseService;
    private final UserService userService;
    private final RecurringExpenseService recurringExpenseService;

    @PostMapping
    public ResponseEntity<?> addExpense(@RequestBody CreateExpenseRequest request,
                                        @AuthenticationPrincipal UserDetails userDetails) {

        User payer = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException("Користувача не знайдено"));
```

```
        if (request.getRecurrencePeriod() != null && !request.getRecurrencePeriod().isBlank()) {
            recurringExpenseService.saveRecurring(payer, request);
            Expense expense = expenseService.createExpense(payer, request);
            return ResponseEntity.ok(expense);
        }

        Expense expense = expenseService.createExpense(payer, request);
        return ResponseEntity.ok(expense);
    }

    @GetMapping("/groups/{groupId}/expenses")
    public ResponseEntity<List<ExpenseSummaryDTO>> getGroupExpenses(
            @PathVariable Long groupId,
            @RequestParam(required = false) Long payerId,
            @RequestParam(required = false) @DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME)
LocalDateTime from,
            @RequestParam(required = false) @DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME)
LocalDateTime to,
            @RequestParam(defaultValue = "createdAt") String sortBy,
            @RequestParam(defaultValue = "desc") String order
    ) {
        List<ExpenseSummaryDTO> result = expenseService.getExpensesByGroup(groupId, payerId, from,
to, sortBy, order);
        return ResponseEntity.ok(result);
    }

    @GetMapping("/{expenseId}")
    public ResponseEntity<ExpenseDetailsDTO> getExpenseDetails(@PathVariable Long expenseId) {
        return ResponseEntity.ok(expenseService.getExpenseDetails(expenseId));
    }

    @GetMapping("/group/{groupId}/summary")
    public ResponseEntity<List<ExpenseSummaryDTO>> getExpenseSummaries(@PathVariable Long
groupId) {
        return ResponseEntity.ok(expenseService.getExpensesByGroup(groupId));
    }


    @GetMapping("/simplified/{groupId}")
    public ResponseEntity<List<DebtSimplifierService.DebtRecord>> getSimplifiedDebts(@PathVariable
Long groupId) {
        return ResponseEntity.ok(debtService.getSimplifiedDebts(groupId));
    }

    @GetMapping("/suggested-titles")
    public ResponseEntity<List<String>> getSuggestedTitles(@RequestParam Long userId) {
        List<String> suggestions = expenseService.getMostUsedTitles(userId);
        return ResponseEntity.ok(suggestions);
    }

    @GetMapping("/my")
    public ResponseEntity<List<ExpenseSummaryDTO>> getMyExpenses(@AuthenticationPrincipal
```

```
CustomUserDetails userDetails) {
        Long userId = userDetails.getUser().getId();
        return ResponseEntity.ok(expenseService.getExpensesByUser(userId));
    }
    @GetMapping("/my/statistics")
    public ResponseEntity<Map<String, Object>> getMyExpenseStats(@AuthenticationPrincipal
CustomUserDetails userDetails) {
        Long userId = userDetails.getUser().getId();
        return ResponseEntity.ok(expenseService.getExpenseStatsByUser(userId));
    }

}
```

## GroupController

```
@RestController
@RequestMapping("/api/groups")

@RequiredArgsConstructor public class GroupController {
    private final GroupRepository groupRepository;
    private final GroupService groupService;
    private final ExpenseRepository expenseRepository;
    private final ExpenseService expenseService;
    private final GroupMemberRepository groupMemberRepository;
    private final UserService userService;

    @GetMapping("/my")
    public ResponseEntity<List<GroupResponseDto>> getMyGroups(
            @AuthenticationPrincipal UserDetails userDetails,
            @RequestHeader(value = "Accept-Language", defaultValue = "uk") String lang) {

        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));

        List<Group> groups = groupMemberRepository.findByUser(user)
                .stream()
                .map(GroupMember::getGroup)
                .toList();

        List<GroupResponseDto> dtos = groups.stream()
                .map(group -> new GroupResponseDto(
                        group.getId(),
                        group.getName(),
                        groupService.canDeleteGroup(group.getId(), user.getId())
                ))
                .toList();

        return ResponseEntity.ok(dtos);
    }

    @GetMapping("/{id}/details")
    public ResponseEntity<GroupDetailsDto> getGroupDetails(
```

```
            @PathVariable Long id,
            @AuthenticationPrincipal UserDetails userDetails,
            @RequestHeader(value = "Accept-Language", defaultValue = "uk") String lang
    ) {
        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));

        Group group = groupRepository.findById(id)
                .orElseThrow(() -> new RuntimeException(t("Групу не знайдено", "Group not found", lang)));

        boolean isMember = groupMemberRepository.existsByGroupAndUser(group, user);
        if (!isMember) {
            return ResponseEntity.status(403).build();
        }

        Double totalExpenses = 0.0;
        Long transactionCount = 0L;
        List<User> members = groupMemberRepository.findByGroup(group)
                .stream()
                .map(GroupMember::getUser)
                .toList();

        GroupDetailsDto dto = GroupDetailsDto.builder()
                .name(group.getName())
                .createdAt(group.getCreatedAt())
                .createdBy(group.getCreatedBy().getFirstName())
                .members(members)
                .totalExpenses(totalExpenses)
                .transactionCount(transactionCount)
                .build();

        return ResponseEntity.ok(dto);
    }

    @PostMapping
    public ResponseEntity<Group> createGroup(@AuthenticationPrincipal UserDetails userDetails,
                                             @RequestBody GroupRequest request,
                                             @RequestHeader(value = "Accept-Language", defaultValue = "uk") String
lang) {
        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));

        Group group = Group.builder()
                .name(request.getName())
                .createdBy(user)
                .deleted(false)
                .build();

        Group savedGroup = groupRepository.save(group);
```

```
        GroupMember member = GroupMember.builder()
                .group(savedGroup)
                .user(user)
                .build();
        groupMemberRepository.save(member);

        return ResponseEntity.ok(savedGroup);
    }

    @GetMapping("/{id}/members")
    public ResponseEntity<List<UserDto>> getGroupMembers(@PathVariable Long id,
                                                         @AuthenticationPrincipal UserDetails userDetails,
                                                         @RequestHeader(value = "Accept-Language", defaultValue = "uk")
String lang) {
        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));

        Group group = groupRepository.findById(id)
                .orElseThrow(() -> new RuntimeException(t("Групу не знайдено", "Group not found", lang)));

        boolean isMember = groupMemberRepository.existsByGroupAndUser(group, user);
        if (!isMember) {
            return ResponseEntity.status(403).build();
        }

        List<User> members = groupMemberRepository.findByGroup(group)
                .stream()
                .map(GroupMember::getUser)
                .toList();

        List<UserDto> memberDtos = members.stream()
                .map(u -> new UserDto(u.getId(), u.getFirstName(), u.getLastName()))
                .toList();

        return ResponseEntity.ok(memberDtos);
    }

    @GetMapping("/{groupId}/stats")
    public ResponseEntity<GroupStatsDTO> getGroupStats(
            @PathVariable Long groupId,
            @AuthenticationPrincipal UserDetails userDetails,
            @RequestHeader(value = "Accept-Language", defaultValue = "uk") String lang) {

        Group group = groupService.findById(groupId)
                .orElseThrow(() -> new RuntimeException(t("Групу не знайдено", "Group not found", lang)));

        Long count = expenseRepository.countByGroupId(groupId);
        BigDecimal sum = expenseRepository.sumAmountByGroupId(groupId);
        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));
```

```
        BigDecimal balance = expenseService.getUserBalanceInGroup(groupId, user);

        return ResponseEntity.ok(new GroupStatsDTO(
                group.getName(), count, sum, balance
        ));
    }
    @DeleteMapping("/{groupId}")
    public ResponseEntity<?> deleteGroup(@PathVariable Long groupId,
                                         @AuthenticationPrincipal UserDetails userDetails,
                                         @RequestHeader(value = "Accept-Language", defaultValue = "uk") String lang)
{
        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));

        boolean deleted = groupService.deleteGroupIfNoDebtsAndOwner(groupId, user.getId());

        return deleted
                ? ResponseEntity.ok().build()
                : ResponseEntity.status(HttpStatus.FORBIDDEN).body(t("Неможливо видалити групу.",
"Cannot delete group.", lang));
    }
    @GetMapping("/{groupId}/can-delete")
    public ResponseEntity<Boolean> canDeleteGroup(@PathVariable Long groupId,
                                                  Principal principal,
                                                  @RequestHeader(value = "Accept-Language", defaultValue = "uk") String
lang) {
        Optional<User> optionalUser = userService.findByEmail(principal.getName());
        if (optionalUser.isEmpty()) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();
        }

        Long userId = optionalUser.get().getId();
        boolean canDelete = groupService.canDeleteGroup(groupId, userId);
        return ResponseEntity.ok(canDelete);
    }

    @GetMapping("/{groupId}/analytics")
    public ResponseEntity<Map<String, BigDecimal>> getGroupAnalytics(
            @PathVariable Long groupId,
            @AuthenticationPrincipal UserDetails userDetails,
            @RequestHeader(value = "Accept-Language", defaultValue = "uk") String lang) {

        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException(t("Користувача не знайдено", "User not found",
lang)));

        Group group = groupRepository.findById(groupId)
                .orElseThrow(() -> new RuntimeException(t("Групу не знайдено", "Group not found", lang)));

        if (!groupMemberRepository.existsByGroupAndUser(group, user)) {
```

```
            return ResponseEntity.status(HttpStatus.FORBIDDEN).build();
        }

        List<Expense> expenses = expenseRepository.findByGroupId(groupId);
        Map<String, BigDecimal> totals = new HashMap<>();

        for (Expense expense : expenses) {
            String name = expense.getPayer().getFirstName() + " " + expense.getPayer().getLastName();
            totals.merge(name, expense.getAmount(), BigDecimal::add);
        }

        return ResponseEntity.ok(totals);
    }
    private String t(String uk, String en, String lang) {
        return "en".equalsIgnoreCase(lang) ? en : uk;
    }


}
```

## GroupMemberController

```
@RestController
@RequestMapping("/api/groups/{groupId}/members")
@RequiredArgsConstructor
public class GroupMemberController {

    private final GroupRepository groupRepository;
    private final GroupMemberService groupMemberService;
    private final UserService userService;

    @PostMapping
    public ResponseEntity<GroupMember> addMember(@PathVariable Long groupId,
                                                 @RequestBody AddMemberRequest request,
                                                 @AuthenticationPrincipal UserDetails userDetails) {
        Group group = groupRepository.findById(groupId).orElseThrow();
        User toAdd = userService.findByEmail(request.getEmail()).orElseThrow();
        GroupMember gm = groupMemberService.addMember(group, toAdd);
        return ResponseEntity.ok(gm);
    }

    @GetMapping("/list")
    public ResponseEntity<List<GroupMember>> getMembers(@PathVariable Long groupId) {
        Group group = groupRepository.findById(groupId).orElseThrow();
        return ResponseEntity.ok(groupMemberService.getMembers(group));
    }
}
```

## GroupPageController

```
@Controller
public class GroupPageController {

    @GetMapping("/group")
```

```
    public String groupPage() {
        return "group";
    }
}
```

## PaymentController

```
@RestController
@RequestMapping("/api/payments")
@RequiredArgsConstructor
public class PaymentController {

    private final PaymentRepository paymentRepository;
    private final GroupRepository groupRepository;
    private final UserRepository userRepository;

    @PostMapping
    public ResponseEntity<?> makePayment(@RequestBody PaymentRequest request,
@AuthenticationPrincipal UserDetailsImpl userDetails) {
        User payer = userRepository.findById(request.getToUserId()).orElseThrow();
        User receiver = userRepository.findById(request.getPayerId()).orElseThrow();
        Group group = groupRepository.findById(request.getGroupId()).orElseThrow();

        Payment payment = Payment.builder()
                .group(group)
                .payer(payer)
                .receiver(receiver)
                .amount(request.getAmount())
                .build();

        paymentRepository.save(payment);

        return ResponseEntity.ok().build();
    }
}
```

## ProfileController

```
@Controller
public class ProfileController {

    @GetMapping("/profile")
    public String profilePage() {
        return "profile";
    }
}
```

## UserController

```
@RestController
@RequestMapping("/api/users")
@RequiredArgsConstructor
public class UserController {
    private final JwtService jwtService;
```

```
    private final UserService userService;
    private final ExpenseService expenseService;

    @GetMapping("/me/balance")
    public ResponseEntity<UserBalanceDTO> getBalance(@AuthenticationPrincipal UserDetails
userDetails) {
        User user = userService.findByEmail(userDetails.getUsername()).orElseThrow();
        return ResponseEntity.ok(expenseService.getUserBalance(user));
    }

    @PutMapping("/me")
    public ResponseEntity<UserProfileDTO> updateProfile(
            @AuthenticationPrincipal UserDetails userDetails,
            @RequestBody UpdateProfileRequest request) {

        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException("Користувача не знайдено"));

        user.setFirstName(request.getFirstName());
        user.setLastName(request.getLastName());
        user.setEmail(request.getEmail());

        userService.save(user);

        UserProfileDTO dto = new UserProfileDTO(
                user.getFirstName(),
                user.getLastName(),
                user.getEmail()
        );

        return ResponseEntity.ok(dto);
    }
    @GetMapping("/me")
    public ResponseEntity<UserProfileDTO> getCurrentUser(@AuthenticationPrincipal UserDetails
userDetails) {
        User user = userService.findByEmail(userDetails.getUsername())
                .orElseThrow(() -> new RuntimeException("Користувача не знайдено"));

        UserProfileDTO dto = new UserProfileDTO(
                user.getFirstName(),
                user.getLastName(),
                user.getEmail()
        );

        return ResponseEntity.ok(dto);
    }
    @GetMapping("/me/id")
    public ResponseEntity<Long> getCurrentUserId(@RequestHeader("Authorization") String
authHeader) {
        String token = authHeader.replace("Bearer ", "");
        String email = jwtService.extractUsername(token);
        Long userId = userService.getUserIdByEmail(email);
```

```
        return ResponseEntity.ok(userId);
    }

}
```

## ViewController

```
@Controller
public class ViewController {

    @GetMapping("/")
    public String redirectToDashboard() {
        return "redirect:/dashboard";
    }

    @GetMapping("/dashboard")
    public String dashboard() {
        return "dashboard";
    }

    @GetMapping("/login")
    public String loginPage() {
        return "login";
    }

    @GetMapping("/register")
    public String registerPage() {
        return "register";
    }

    @GetMapping("/group/{id}")
    public String groupPage() {
        return "group";
    }

    @GetMapping("/support")
    public String supportPage() {
        return "support";
    }

    @GetMapping("/about")
    public String aboutPage() {
        return "about";
    }

}
```

## AddMemberRequest

```
@Data
public class AddMemberRequest {
    private String email;
}
```

## AuthResponse

```
@Data
@AllArgsConstructor
public class AuthResponse {
    private String token;
}
```

## CreateExpenseRequest

```
@Data
@Getter
@Setter
public class CreateExpenseRequest {
    private String title;
    private BigDecimal amount;
    private Long groupId;
    private Long payerId;
    private List<Long> participants;
    private List<SplitShareDTO> shares;
    private String recurrencePeriod;
    private LocalDate startDate;
    private LocalDate endDate;

    public List<SplitShareDTO> getShares() {
        return shares;
    }

    public void setShares(List<SplitShareDTO> shares) {
        this.shares = shares;
    }
}
```

## CreateRecurringExpenseRequest

```
@Data
public class CreateRecurringExpenseRequest {

    private Long groupId;
    private Long payerId;

    private String title;
    private BigDecimal amount;

    private String recurrencePeriod;

    private LocalDate startDate;
    private LocalDate endDate;
}
```

## DebtDTO

```
@Data
@AllArgsConstructor
```

```
public class DebtDTO {
    private Long fromUserId;
    private Long toUserId;
    private BigDecimal amount;

}
```

## ExpenseDetailsDTO

```
@Data
@AllArgsConstructor
public class ExpenseDetailsDTO {
    private Long id;
    private String title;
    private BigDecimal amount;
    private String payer;
    private LocalDateTime createdAt;
    private List<SplitDetailDTO> splits;
}
```

## ExpenseSummaryDTO

```
@Data
@AllArgsConstructor
@Builder
public class ExpenseSummaryDTO {
    private Long id;
    private String title;
    private BigDecimal amount;
    private LocalDateTime createdAt;
    private String payerFullName;
}
```

## GroupDetailsDto

```
@Data
@Builder
public class GroupDetailsDto {
    private String name;
    private LocalDateTime createdAt;
    private String createdBy;
    private List<User> members;
    private Double totalExpenses;
    private Long transactionCount;
}
```

## GroupRequest

```
@Data
public class GroupRequest {
    private String name;
}
```

## GroupResponseDto

```
@Data
@AllArgsConstructor
public class GroupResponseDto {
    private Long id;
    private String name;
    private boolean canDelete;
}
```

## GroupStatsDTO

```
@Data
@AllArgsConstructor
public class GroupStatsDTO {
    private String groupName;
    private Long transactionCount;
    private BigDecimal totalAmount;
    private BigDecimal userBalance;
}
```

## LoginRequest

```
@Data
public class LoginRequest {
    private String email;
    private String password;
}
```

## PaymentRequest

```
@Data
public class PaymentRequest {
    private Long groupId;
    private Long toUserId;
    private Long payerId;
    private BigDecimal amount;

}
```

## RegisterRequest

```
@Data
public class RegisterRequest {
    private String firstName;
    private String lastName;
    private String email;
    private String password;
}
```

## SplitDetailDTO

```
@Data
@AllArgsConstructor
public class SplitDetailDTO {
    private String user;
    private String shareType;
```

```
    private BigDecimal shareValue;
}
```

SplitShareDTO

```
@Data
public class SplitShareDTO {
    private Long userId;
    private String shareType;
    private BigDecimal shareValue;
}
```

UpdateProfileRequest

```
@Data
public class UpdateProfileRequest {
    private String firstName;
    private String lastName;
    private String email;
}
```

UserBalanceDTO

```
@Data
@AllArgsConstructor
public class UserBalanceDTO {
    private BigDecimal paid;
    private BigDecimal owed;
    private BigDecimal balance;
}
```

UserDto

```
@Data
@AllArgsConstructor
public class UserDto {
    private Long id;
    private String firstName;
    private String lastName;
}
```

UserProfileDTO

```
@Data
@AllArgsConstructor
public class UserProfileDTO {
    private String firstName;
    private String lastName;
    private String email;
}
```

Expense

```
@Entity
@Table(name = "expenses")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Expense {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "group_id", nullable = false)
    private Group group;

    @Column(nullable = false)
    private BigDecimal amount;

    @ManyToOne
    @JoinColumn(name = "payer_id", nullable = false)
    private User payer;

    @Column(nullable = false)
    private String title;

    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;

    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
    }
}
```

## ExpenseSplit

```
@Entity
@Table(name = "expense_splits")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ExpenseSplit {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "expense_id", nullable = false)
    private Expense expense;
```

```
    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @Column(name = "share_type", nullable = false)
    private String shareType;

    @Column(name = "share_value", nullable = false)
    private BigDecimal shareValue;
}
```

## Group

```
@Entity
@Table(name = "groups")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Group {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 100)
    private String name;

    @ManyToOne
    @JoinColumn(name = "created_by")
    private User createdBy;

    @Column(name = "is_deleted")
    private boolean deleted = false;

    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;

    @ManyToMany
    @JoinTable(
            name = "group_members",
            joinColumns = @JoinColumn(name = "group_id"),
            inverseJoinColumns = @JoinColumn(name = "user_id")
    )
    private Set<User> members = new HashSet<>();
    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
    }
}
```

## GroupMember

```
@Entity
@Table(name = "group_members",
        uniqueConstraints = {@UniqueConstraint(columnNames = {"group_id", "user_id"})})
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class GroupMember {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "group_id", nullable = false)
    private Group group;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false)
    private User user;
}
```

## Payment

```
@Entity
@Table(name = "payments")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(optional = false)
    @JoinColumn(name = "group_id")
    private Group group;

    @ManyToOne(optional = false)
    @JoinColumn(name = "payer_id")
    private User payer;

    @ManyToOne(optional = false)
    @JoinColumn(name = "receiver_id")
    private User receiver;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal amount;

    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;
```

```
    @PrePersist
    public void prePersist() {
        this.createdAt = LocalDateTime.now();
    }
}
```

## RecurringExpense

```
@Entity
@Table(name = "recurring_expenses")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class RecurringExpense {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "group_id", nullable = false)
    private Group group;

    @ManyToOne
    @JoinColumn(name = "payer_id", nullable = false)
    private User payer;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false, precision = 10, scale = 2)
    private BigDecimal amount;

    @Column(name = "recurrence_period", nullable = false, length = 10)
    private String recurrencePeriod;

    @Column(name = "start_date", nullable = false)
    private LocalDate startDate;

    @Column(name = "end_date")
    private LocalDate endDate;

    @Column(name = "next_occurrence")
    private LocalDate nextOccurrence;

    @Column(name = "last_generated")
    private LocalDate lastGenerated;

    @Column(name = "created_at", nullable = false)
    private LocalDateTime createdAt;
```

```
}
```

## RecurringExpenseSplit

```
@Entity
@Table(name = "recurring_expense_splits")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class RecurringExpenseSplit {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "recurring_expense_id", nullable = false)
    private RecurringExpense recurringExpense;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "user_id", nullable = false)
    private User user;

    @Column(name = "share_type", nullable = false)
    private String shareType;

    @Column(name = "share_value", nullable = false)
    private BigDecimal shareValue;
}
```

## User

```
@Entity
@Table(name = "users", uniqueConstraints = {
        @UniqueConstraint(columnNames = "email")
})
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "first_name", nullable = false, length = 50)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 50)
    private String lastName;
```

```
    @Column(nullable = false, length = 100)
    private String email;

    @Column(name = "password_hash", nullable = false)
    private String passwordHash;

    @Column(name = "created_at", updatable = false)
    private LocalDateTime createdAt;

    @PrePersist
    public void prePersist() {
        createdAt = LocalDateTime.now();
    }
}
```

## ExpenseRepository

```
public interface ExpenseRepository extends JpaRepository<Expense, Long> {

    List<Expense> findByGroupId(Long groupId);
    List<Expense> findByGroup(Group group);
    List<Expense> findAllByPayerId(Long payerId);
    Long countByGroup(Group group);

    @Query("SELECT COALESCE(SUM(e.amount), 0) FROM Expense e WHERE e.group = :group")
    BigDecimal calculateTotalAmountByGroup(Group group);

    @Query("SELECT COALESCE(SUM(e.amount), 0) FROM Expense e WHERE e.group.id =
:groupId")
    BigDecimal sumAmountByGroupId(@Param("groupId") Long groupId);
    @Query("SELECT COUNT(e) FROM Expense e WHERE e.group.id = :groupId")
    Long countByGroupId(@Param("groupId") Long groupId);

    @Query("SELECT e FROM Expense e WHERE e.group = :group" +
           " AND (:payer IS NULL OR e.payer = :payer)" +
           " AND (:from IS NULL OR e.createdAt >= :from)" +
           " AND (:to IS NULL OR e.createdAt <= :to)")
    List<Expense> findByGroupWithFilters(@Param("group") Group group,
                                         @Param("payer") User payer,
                                         @Param("from") LocalDateTime from,
                                         @Param("to") LocalDateTime to);
}
```

## ExpenseSplitRepository

```
public interface ExpenseSplitRepository extends JpaRepository<ExpenseSplit, Long> {
    List<ExpenseSplit> findByExpense_Group_Id(Long groupId);
}
```

## GroupMemberRepository

```
public interface GroupMemberRepository extends JpaRepository<GroupMember, Long> {
    List<GroupMember> findByGroup(Group group);
    List<GroupMember> findByUser(User user);
    boolean existsByGroupAndUser(Group group, User user);
    boolean existsByGroupIdAndUserId(Long groupId, Long userId);

}
```

## GroupRepository

```
public interface GroupRepository extends JpaRepository<Group, Long> {
    List<Group> findAllByCreatedByAndDeletedFalse(User user);
}
```

## PaymentRepository

```
public interface PaymentRepository extends JpaRepository<Payment, Long> {
    List<Payment> findByGroupId(Long groupId);
}
```

## RecurringExpenseRepository

```
@Repository
public interface RecurringExpenseRepository extends JpaRepository<RecurringExpense, Long> {
    List<RecurringExpense> findAllByNextOccurrence(LocalDate date);
}
```

## RecurringExpenseSplitRepository

```
public interface RecurringExpenseSplitRepository extends JpaRepository<RecurringExpenseSplit, Long>
{
    List<RecurringExpenseSplit> findByRecurringExpenseId(Long recurringExpenseId);
}
```

## UserRepository

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
    boolean existsByEmail(String email);
}
```

## CustomUserDetails

```
@Getter
public class CustomUserDetails implements UserDetails {

    private final User user;

    public CustomUserDetails(User user) {
        this.user = user;
    }
    public User getUser() {
        return user;
```

```
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.emptyList();
    }

    @Override
    public String getPassword() {
        return user.getPasswordHash();
    }

    @Override
    public String getUsername() {
        return user.getEmail();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

JwtAuthenticationFilter

```
@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final UserDetailsServiceImpl userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain) throws ServletException, IOException {

        final String authHeader = request.getHeader("Authorization");
        final String jwt;
```

```
        final String userEmail;

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        jwt = authHeader.substring(7);
        userEmail = jwtService.extractUsername(jwt);

        if (userEmail != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);

            if (jwtService.isTokenValid(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(
                        userDetails,
                        null,
                        userDetails.getAuthorities()
                );

                authToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }

        filterChain.doFilter(request, response);
    }
}
```

## JwtService

```
@Service
public class JwtService {

    @Value("${app.jwt.secret}")
    private String secretKeyString;

    private Key secretKey;

    @PostConstruct
    public void init() {
        this.secretKey = Keys.hmacShaKeyFor(secretKeyString.getBytes());
    }

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
```

```
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(secretKey)
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    public boolean isTokenValid(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()));
    }

    public String generateToken(UserDetails userDetails) {
        return Jwts.builder()
                .setClaims(Map.of())
                .setSubject(userDetails.getUsername())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 24))
                .signWith(secretKey, SignatureAlgorithm.HS256)
                .compact();
    }
}
```

## SecurityConfig

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {
    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final UserDetailsServiceImpl userDetailsService;
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(csrf -> csrf.disable())
                .cors(Customizer.withDefaults())
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers(
                                "/", "/dashboard", "/profile", "/about", "/support", "/language",
                                "/login", "/register", "/group/**",
                                "/css/**", "/js/**", "/img/**", "/webjars/**"
                        ).permitAll()
                        .requestMatchers(HttpMethod.DELETE, "/api/groups/**").authenticated()
                        .requestMatchers("/api/groups/**").authenticated()
                        .requestMatchers("/api/payments/**").authenticated()
                        .requestMatchers("/api/expenses/**").authenticated()
                        .requestMatchers("/api/auth/**").permitAll()
                        .anyRequest().authenticated()
```

```
            )
            .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authenticationProvider(daoAuthenticationProvider())
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)
            .build();
    }

    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder());
        return provider;
    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws
Exception {
        return config.getAuthenticationManager();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

## UserDetailsImpl

```
@AllArgsConstructor
@Getter
public class UserDetailsImpl implements UserDetails {
    private final User user;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.emptyList();
    }

    @Override
    public String getPassword() {
        return user.getPasswordHash();
    }

    @Override
    public String getUsername() {
        return user.getEmail();
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }
```

```
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }

    public Long getId() {
        return user.getId();
    }
}
```

## DebtService

```
@Service
@RequiredArgsConstructor
public class DebtService {
    private final ExpenseSplitRepository expenseSplitRepository;
    private final PaymentRepository paymentRepository;
    private final DebtSimplifierService debtSimplifierService;
    private final ExpenseRepository expenseRepository;
    private final ExpenseSplitRepository splitRepository;

    public List<DebtDTO> calculateDebtsByGroup(Long groupId) {
        List<Expense> expenses = expenseRepository.findAll();
        List<DebtDTO> debts = new ArrayList<>();

        for (Expense expense : expenses) {
            if (!expense.getGroup().getId().equals(groupId)) continue;

            Long payerId = expense.getPayer().getId();

            List<ExpenseSplit> splits = splitRepository.findAll();

            for (ExpenseSplit split : splits) {
                if (!split.getExpense().getId().equals(expense.getId())) continue;

                Long userId = split.getUser().getId();
                if (!userId.equals(payerId)) {
                    debts.add(new DebtDTO(
                            split.getUser().getId(),
                            payerId,
                            split.getShareValue()
                    ));
```

```
            }
        }
    }

    return mergeDebts(debts);
}

private List<DebtDTO> mergeDebts(List<DebtDTO> raw) {
    Map<String, BigDecimal> aggregated = new HashMap<>();

    for (DebtDTO debt : raw) {
        String key = debt.getFromUserId() + "->" + debt.getToUserId();
        aggregated.put(key, aggregated.getOrDefault(key, BigDecimal.ZERO).add(debt.getAmount()));
    }

    List<DebtDTO> result = new ArrayList<>();
    for (Map.Entry<String, BigDecimal> entry : aggregated.entrySet()) {
        String[] parts = entry.getKey().split("->");
        Long from = Long.parseLong(parts[0]);
        Long to = Long.parseLong(parts[1]);
        result.add(new DebtDTO(from, to, entry.getValue()));
    }

    return result;
}
public boolean allBalancesZero(Long groupId) {
    List<DebtDTO> debts = calculateDebtsByGroup(groupId);
    return debts.stream().allMatch(debt -> debt.getAmount().compareTo(BigDecimal.ZERO) == 0);
}
public List<DebtSimplifierService.DebtRecord> getSimplifiedDebts(Long groupId) {
    List<Expense> expenses = expenseRepository.findByGroupId(groupId);
    List<ExpenseSplit> splits = expenseSplitRepository.findByExpense_Group_Id(groupId);
    List<Payment> payments = paymentRepository.findByGroupId(groupId);

    return debtSimplifierService.simplifyDebts(splits, expenses, payments);
}
}
```

## DebtSimplifierService

```
@Service
public class DebtSimplifierService {
    @Data
    @AllArgsConstructor
    @NoArgsConstructor
    private static class UserBalance {
        private Long user;
        private BigDecimal amount;
    }

    public List<DebtDTO> simplify(List<DebtDTO> debts) {
        Map<Long, BigDecimal> netBalance = new HashMap<>();
```

```
        for (DebtDTO debt : debts) {
            netBalance.put(debt.getFromUserId(), netBalance.getOrDefault(debt.getFromUserId(),
BigDecimal.ZERO).subtract(debt.getAmount()));
            netBalance.put(debt.getToUserId(), netBalance.getOrDefault(debt.getToUserId(),
BigDecimal.ZERO).add(debt.getAmount()));
        }

        PriorityQueue<UserBalance> debtors = new PriorityQueue<>(Comparator.comparing(ub ->
ub.amount));
        PriorityQueue<UserBalance> creditors = new PriorityQueue<>((a, b) ->
b.amount.compareTo(a.amount));

        for (Map.Entry<Long, BigDecimal> entry : netBalance.entrySet()) {
            BigDecimal amount = entry.getValue();
            if (amount.compareTo(BigDecimal.ZERO) < 0) {
                debtors.add(new UserBalance(entry.getKey(), amount.abs()));
            } else if (amount.compareTo(BigDecimal.ZERO) > 0) {
                creditors.add(new UserBalance(entry.getKey(), amount));
            }
        }

        List<DebtDTO> simplified = new ArrayList<>();

        while (!debtors.isEmpty() && !creditors.isEmpty()) {
            UserBalance debtor = debtors.poll();
            UserBalance creditor = creditors.poll();

            BigDecimal amount = debtor.amount.min(creditor.amount);
            simplified.add(new DebtDTO(debtor.user, creditor.user, amount));

            if (debtor.amount.compareTo(creditor.amount) > 0) {
                debtors.add(new UserBalance(debtor.user, debtor.amount.subtract(amount)));
            } else if (creditor.amount.compareTo(debtor.amount) > 0) {
                creditors.add(new UserBalance(creditor.user, creditor.amount.subtract(amount)));
            }
        }

        return simplified;
    }

    public record DebtRecord(Long fromUserId, Long toUserId, BigDecimal amount) {}

    public List<DebtRecord> simplifyDebts(List<ExpenseSplit> splits, List<Expense> expenses,
List<Payment> payments) {
        Map<Long, BigDecimal> balanceMap = new HashMap<>();

        for (ExpenseSplit split : splits) {
            balanceMap.merge(split.getUser().getId(), split.getShareValue().negate(), BigDecimal::add);
        }

        for (Expense expense : expenses) {
            balanceMap.merge(expense.getPayer().getId(), expense.getAmount(), BigDecimal::add);
```

```
        }

        for (Payment payment : payments) {
            balanceMap.merge(payment.getPayer().getId(), payment.getAmount().negate(),
BigDecimal::add);
            balanceMap.merge(payment.getReceiver().getId(), payment.getAmount(), BigDecimal::add);
        }

        List<UserBalance> debtors = new ArrayList<>();
        List<UserBalance> creditors = new ArrayList<>();

        for (Map.Entry<Long, BigDecimal> entry : balanceMap.entrySet()) {
            BigDecimal balance = entry.getValue().setScale(2);
            if (balance.compareTo(BigDecimal.ZERO) < 0) {
                debtors.add(new UserBalance(entry.getKey(), balance));
            } else if (balance.compareTo(BigDecimal.ZERO) > 0) {
                creditors.add(new UserBalance(entry.getKey(), balance));
            }
        }

        List<DebtRecord> results = new ArrayList<>();
        int i = 0, j = 0;
        while (i < debtors.size() && j < creditors.size()) {
            var debtor = debtors.get(i);
            var creditor = creditors.get(j);
            BigDecimal amount = debtor.amount.abs().min(creditor.amount);

            if (amount.compareTo(BigDecimal.ZERO) > 0) {
                results.add(new DebtRecord(debtor.user, creditor.user, amount));
                debtor.amount = debtor.amount.add(amount);
                creditor.amount = creditor.amount.subtract(amount);
            }

            if (debtor.amount.compareTo(BigDecimal.ZERO) == 0) i++;
            if (creditor.amount.compareTo(BigDecimal.ZERO) == 0) j++;
        }

        return results;
    }
}
```

## ExpenseService

```
@Service
@RequiredArgsConstructor
public class ExpenseService {
    private final ExpenseRepository expenseRepository;
    private final ExpenseSplitRepository splitRepository;
    private final GroupRepository groupRepository;
    private final UserRepository userRepository;

    public Expense createExpense(User payer, CreateExpenseRequest request) {
        Group group = groupRepository.findById(request.getGroupId())
```

```
        .orElseThrow(() -> new IllegalArgumentException("Group not found"));
User payerFromRequest = userRepository.findById(request.getPayerId())
        .orElseThrow(() -> new IllegalArgumentException("Payer not found"));

Expense expense = Expense.builder()
        .group(group)
        .payer(payerFromRequest)
        .amount(request.getAmount())
        .title(request.getTitle())
        .build();

Expense savedExpense = expenseRepository.save(expense);

List<SplitShareDTO> shares = request.getShares();
BigDecimal totalCalculated = BigDecimal.ZERO;

int equalCount = (int) shares.stream()
        .filter(s -> s.getShareType().equalsIgnoreCase("equal"))
        .count();

for (SplitShareDTO dto : shares) {
    User user = userRepository.findById(dto.getUserId())
            .orElseThrow(() -> new IllegalArgumentException("User not found: " + dto.getUserId()));

    String type = dto.getShareType().toLowerCase();
    BigDecimal value;

    switch (type) {
        case "manual":
            value = dto.getShareValue();
            break;

        case "percentage":
            value = request.getAmount()
                    .multiply(dto.getShareValue())
                    .divide(BigDecimal.valueOf(100), 2, RoundingMode.HALF_UP);
            break;

        case "equal":
            value = request.getAmount()
                    .divide(BigDecimal.valueOf(equalCount), 2, RoundingMode.HALF_UP);
            break;

        default:
            throw new IllegalArgumentException("Unknown share type: " + dto.getShareType());
    }

    totalCalculated = totalCalculated.add(value);

    ExpenseSplit split = ExpenseSplit.builder()
            .expense(savedExpense)
            .user(user)
```

```
                .shareType(type)
                .shareValue(value)
                .build();

        splitRepository.save(split);
      }
      BigDecimal diff = totalCalculated.subtract(request.getAmount()).abs();
      if (diff.compareTo(BigDecimal.valueOf(0.01)) > 0) {
        throw new IllegalStateException("Total shares do not match expense amount (difference: " + diff
+ ")");
      }

      return savedExpense;
    }
    public List<ExpenseSummaryDTO> getExpensesByGroup(Long groupId, Long payerId,
LocalDateTime from, LocalDateTime to,  String sortBy, String order) {
      List<Expense> expenses = expenseRepository.findByGroupId(groupId);
      return expenses.stream()
            .map(exp -> new ExpenseSummaryDTO(
                  exp.getId(),
                  exp.getTitle(),
                  exp.getAmount(),
                  exp.getCreatedAt(),
                  exp.getPayer().getFirstName() + " " + exp.getPayer().getLastName()
            ))
            .toList();
    }

    public List<ExpenseSummaryDTO> getExpensesByGroup(Long groupId) {
      return getExpensesByGroup(groupId, null, null, null, null, null);
    }
    public ExpenseDetailsDTO getExpenseDetails(Long expenseId) {
      Expense expense = expenseRepository.findById(expenseId).orElseThrow();

      List<ExpenseSplit> splits = splitRepository.findAll()
            .stream()
            .filter(s -> s.getExpense().getId().equals(expenseId))
            .toList();

      List<SplitDetailDTO> details = splits.stream()
            .map(s -> new SplitDetailDTO(
                  s.getUser().getFirstName(),
                  s.getShareType(),
                  s.getShareValue()
            ))
            .toList();

      return new ExpenseDetailsDTO(
            expense.getId(),
            expense.getTitle(),
            expense.getAmount(),
            expense.getPayer().getFirstName(),
```

```
            expense.getCreatedAt(),
            details
    );
}
public BigDecimal getUserBalanceInGroup(Long groupId, User user) {
    BigDecimal paid = expenseRepository.findByGroupId(groupId).stream()
            .filter(e -> e.getPayer().getId().equals(user.getId()))
            .map(Expense::getAmount)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    BigDecimal owed = splitRepository.findAll().stream()
            .filter(s -> s.getExpense().getGroup().getId().equals(groupId))
            .filter(s -> s.getUser().getId().equals(user.getId()))
            .map(ExpenseSplit::getShareValue)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    return paid.subtract(owed);
}
public UserBalanceDTO getUserBalance(User user) {
    BigDecimal paid = expenseRepository.findAll().stream()
            .filter(e -> e.getPayer().getId().equals(user.getId()))
            .map(Expense::getAmount)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    BigDecimal owed = splitRepository.findAll().stream()
            .filter(s -> s.getUser().getId().equals(user.getId()))
            .map(s -> s.getShareValue())
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    BigDecimal balance = paid.subtract(owed);
    return new UserBalanceDTO(paid, owed, balance);
}
public Map<String, Long> getMostUsedCategoriesByUser(Long userId) {
    return expenseRepository.findAllByPayerId(userId).stream()
            .collect(Collectors.groupingBy(
                    Expense::getTitle,
                    Collectors.counting()
            ));
}

public List<String> getMostUsedTitles(Long userId) {
    List<Expense> expenses = expenseRepository.findAllByPayerId(userId);

    return expenses.stream()
            .filter(e -> e.getTitle() != null)
            .collect(Collectors.groupingBy(
                    Expense::getTitle,
                    Collectors.counting()
            ))
            .entrySet().stream()
            .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
            .limit(3)
```

```
            .map(Map.Entry::getKey)
            .collect(Collectors.toList());
    }
    public List<ExpenseSummaryDTO> getExpensesByUser(Long userId) {
        return expenseRepository.findAllByPayerId(userId).stream()
                .map(expense -> new ExpenseSummaryDTO(
                        expense.getId(),
                        expense.getTitle(),
                        expense.getAmount(),
                        expense.getCreatedAt(),
                        expense.getPayer().getFirstName() + " " + expense.getPayer().getLastName()
                ))
                .collect(Collectors.toList());
    }

    public Map<String, Object> getExpenseStatsByUser(Long userId) {
        List<Expense> expenses = expenseRepository.findAllByPayerId(userId);

        var byMonth = expenses.stream().collect(Collectors.groupingBy(
                e -> e.getCreatedAt().getMonth().toString(),
                Collectors.summingDouble(e -> e.getAmount().doubleValue())
        ));

        var total = expenses.stream()
                .mapToDouble(e -> e.getAmount().doubleValue())
                .sum();

        Map<String, Object> result = new HashMap<>();
        result.put("totalAmount", total);
        result.put("byMonth", byMonth);

        return result;
    }
```

## GroupMemberService

```
@Service
@RequiredArgsConstructor
public class GroupMemberService {

    private final GroupMemberRepository memberRepository;

    public GroupMember addMember(Group group, User user) {
        if (memberRepository.existsByGroupAndUser(group, user)) {
            throw new IllegalStateException("User already in group");
        }

        GroupMember member = GroupMember.builder()
                .group(group)
                .user(user)
                .build();
        return memberRepository.save(member);
    }
```

```
    public List<GroupMember> getMembers(Group group) {
        return memberRepository.findByGroup(group);
    }
}
```

## GroupService

```
@Service
@RequiredArgsConstructor
public class GroupService {
    private final DebtService debtService;
    private final GroupRepository groupRepository;

    public Group createGroup(User creator, String name) {
        Group group = Group.builder()
                .name(name)
                .createdBy(creator)
                .deleted(false)
                .build();
        return groupRepository.save(group);
    }
    public boolean deleteGroupIfNoDebtsAndOwner(Long groupId, Long userId) {
        Optional<Group> optional = groupRepository.findById(groupId);
        if (optional.isEmpty()) return false;
        Group group = optional.get();
        if (!group.getCreatedBy().getId().equals(userId)) return false;
        if (!debtService.allBalancesZero(groupId)) return false;

        groupRepository.delete(group);
        return true;
    }
    public boolean canDeleteGroup(Long groupId, Long userId) {
        Optional<Group> optional = groupRepository.findById(groupId);
        if (optional.isEmpty()) return false;
        Group group = optional.get();
        return group.getCreatedBy().getId().equals(userId) && debtService.allBalancesZero(groupId);
    }
    public Optional<Group> findById(Long id) {
        return groupRepository.findById(id);
    }
    public List<Group> getUserGroups(User user) {
        return groupRepository.findAllByCreatedByAndDeletedFalse(user);
    }
}
```

## RecurringExpenseService

```
@Service
@RequiredArgsConstructor
public class RecurringExpenseService {

    private final RecurringExpenseRepository recurringExpenseRepository;
```

```
    private final RecurringExpenseSplitRepository recurringExpenseSplitRepository;
    private final GroupRepository groupRepository;
    private final UserRepository userRepository;

    public void saveRecurring(User payer, CreateExpenseRequest request) {
        Group group = groupRepository.findById(request.getGroupId())
                .orElseThrow(() -> new IllegalArgumentException("Group not found"));

        RecurringExpense recurringExpense = RecurringExpense.builder()
                .group(group)
                .payer(payer)
                .title(request.getTitle())
                .amount(request.getAmount())
                .recurrencePeriod(request.getRecurrencePeriod())
                .startDate(request.getStartDate())
                .endDate(request.getEndDate())
                .nextOccurrence(request.getStartDate())
                .createdAt(LocalDateTime.now())
                .build();

        recurringExpenseRepository.save(recurringExpense);

        for (SplitShareDTO share : request.getShares()) {
            User user = userRepository.findById(share.getUserId())
                    .orElseThrow(() -> new IllegalArgumentException("User not found"));

            RecurringExpenseSplit split = RecurringExpenseSplit.builder()
                    .recurringExpense(recurringExpense)
                    .user(user)
                    .shareType(share.getShareType())
                    .shareValue(share.getShareValue())
                    .build();

            recurringExpenseSplitRepository.save(split);
        }
    }
}
```

## UserDetailsServiceImpl

```
@Service
@RequiredArgsConstructor
public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserRepository userRepository;


    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepository.findByEmail(email)
                .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        return new CustomUserDetails(user);
```

```
    }
}
```

UserService

```
@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public Optional<User> findByEmail(String email) {
        return userRepository.findByEmail(email);
    }

    public boolean emailExists(String email) {
        return userRepository.existsByEmail(email);
    }

    public User registerUser(String firstName, String lastName, String email, String rawPassword) {
        String hashed = passwordEncoder.encode(rawPassword);
        User user = User.builder()
                .firstName(firstName)
                .lastName(lastName)
                .email(email)
                .passwordHash(hashed)
                .build();
        return userRepository.save(user);
    }
    public Long getUserIdByEmail(String email) {
        return userRepository.findByEmail(email)
                .orElseThrow(() -> new RuntimeException("Користувача не знайдено"))
                .getId();
    }
    public void save(User user) {
        userRepository.save(user);
    }
}
```