МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Західноукраїнський національний університет

Факультет комп'ютерних інформаційних технологій

Кафедра інформаційно-обчислювальних систем і управління

ШАБАТЬКО Артур Вадимович

Веб-застосунок для аналізу та візуалізації особистих фінансів/Web application for analyzing and visualizing personal finances

спеціальність: 122 - Комп'ютерні науки

освітньо-професійна програма - Комп’ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-43

А.В. Шабатько

_____

Науковий керівник:

к.т.н., доцент, І.В. Турченко

_____

Кваліфікаційну роботу

допущено до захисту:

"___" _____ 20___ р.

Завідувач кафедри

_____Н. В. Дзюбановська

ТЕРНОПІЛЬ – 2026

Факультет комп'ютерних інформаційних технологій

Кафедра інформаційно-обчислювальних систем і управління

Освітній ступінь «бакалавр»

спеціальність: 122 – Комп'ютерні науки

освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ Н.В. Дзюбановська

«____» _____20__р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

ШАБАТЬКУ Артурові Вадимовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Веб-застосунок для аналізу та візуалізації особистих фінансів/Web application for analyzing and visualizing personal finances

керівник роботи к.т.н., доцент, І.В. Турченко

затверджений наказом по університету від 01 грудня 2025 року № 894.

2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.

4. Основні питання, які потрібно розробити

- Провести аналіз предметної області та огляд існуючих рішень для управління особистими фінансами, виявити їхні переваги та обмеження
- Обґрунтувати вибір методів та алгоритмів для категоризації витрат, аналізу фінансових потоків та прогнозування бюджету
- Спроектувати клієнт-серверну архітектуру веб-застосунку з використанням шаблону проектування модель-вигляд-контроллер, розробити структуру бази даних та кінцевих точок прикладного програмного інтерфейсу
- Програмна реалізація спеціалізованого веб-застосунку для автоматизованого збору, нормалізації, обліку та глибокого аналізу фінансових операцій користувача.
- Провести тестування веб-застосунку та зручність користувацького інтерфейсу для візуалізації фінансової інформації

5. Перелік графічного матеріалу у роботі

- Схема архітектури веб-застосунку
- Схема алгоритму автоматизованої категоризації та відображення фінансової операції
- Даталогічна модель даних

6. Консультанти розділів кваліфікаційної роботи

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

7. Дата видачі завдання_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Назва етапів кваліфікаційної роботи	до 01.01.2026 р.	
2	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.02.2026 р.	
3	Написання 1 розділу кваліфікаційної роботи	до 05.02.2026 р.	
4	Написання 2 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Написання 3 розділу кваліфікаційної роботи	до 15.05.2026 р.	
6	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 25.05.2026 р.	
7	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 30.05.2026 р.	
8	Перевірка кваліфікаційної роботи на оригінальність тексту	до 10.06.2026 р.	
9	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	

Студент_____А.В. Шабатько
підпис

Керівник роботи _____к.т.н., доцент І.В. Турченко
підпис

АНОТАЦІЯ

Кваліфікаційна робота на тему «Веб-застосунок для аналізу та візуалізації особистих фінансів» на здобуття освітнього ступеня «бакалавр» зі спеціальності 122 «Комп'ютерні науки» освітньої програми «Комп'ютерні науки» написана обсягом в 70 сторінок і містить 19 ілюстрацій, 6 додатків та 26 використаних джерел.

Метою кваліфікаційної роботи є розробка, дослідження, проектування та програмна реалізація спеціалізованого веб-застосунку для автоматизованого збору, нормалізації, обліку та глибокого аналізу фінансових операцій користувача.

Методами розроблення обрано метод аналізу (для вивчення існуючих фінансових трекерів та можливостей інтеграції з API банків), метод об'єктно-орієнтованого проектування (для побудови гнучкої архітектури додатку на базі фреймворку Laravel), методи моделювання баз даних (для проектування надійної реляційної структури збереження транзакцій) та метод тестування (для перевірки безпеки й коректності обробки фінансових даних).

Внаслідок виконання роботи обґрунтовано вибір архітектурних рішень для створення фінансових агрегаторів та розроблено веб-додаток, який дає змогу централізовано збирати, категоризувати та аналізувати історію витрат користувача з різних банківських рахунків у єдиному інтерфейсі.

Результати дослідження можуть бути використані для подальшого вдосконалення інструментів управління особистим бюджетом, масштабування системи, а також слугувати програмною основою для розробки комерційних фінтех-продуктів.

Ключові слова: ОСОБИСТІ ФІНАНСИ, ВІЗУАЛІЗАЦІЯ ДАНИХ, АНАЛІЗ ДАНИХ, УПРАВЛІННЯ БЮДЖЕТОМ, ДАШБОРД, ПРИКЛАДНИЙ ПРОГРАМНИЙ ІНТЕРФЕЙС

ANNOTATION

The work on the topic "Web application for analyzing and visualizing personal finances" for the degree of Bachelor in specialty 122 "Computer Science", educational program "Computer Science", consists of 70 pages and contains 19 figures, 6 appendices, and 26 references.

The objective of the work is the research, design, and software implementation of a specialized web application for the automated collection, normalization, accounting, and in-depth analysis of user financial transactions.

The chosen development methods include the analytical method (to study existing financial trackers and bank API integration capabilities), object-oriented design (to build a flexible application architecture based on the Laravel framework), database modeling (to design a robust relational structure for transaction storage), and testing (to verify the security and accuracy of financial data processing).

As a result of the work, the choice of architectural solutions for creating financial aggregators was substantiated, and a web application was developed. This application allows users to centrally collect, categorize, and analyze their expense history from various bank accounts within a single interface.

The research results can be used to further improve personal budget management tools, scale the system, and serve as a software foundation for the development of commercial fintech products.

Keywords: PERSONAL FINANCE, DATA VISUALIZATION, DATA ANALYSIS, BUDGET MANAGEMENT, DASHBOARD, APPLICATION PROGRAMMING INTERFACE

ЗМІСТ

ВСТУП

У сучасну епоху, що характеризується стрімкою цифровізацією фінансового сектору та глобальним переходом до безготівкової економіки, ефективне управління особистими та корпоративними фінансами стало критично важливим фактором повсякденного життя. Зростання рівня фінансової грамотності населення стимулює попит на зручні інструменти для персональної аналітики. Контроль і аналіз витрат займають центральне місце в цьому процесі, адже вони є необхідними для забезпечення фінансової стабільності, оптимізації бюджету, виявлення неочевидних патернів витрат та досягнення довгострокових фінансових цілей.

Незважаючи на наявність на ринку різноманітних фінансових додатків, значною проблемою залишається фрагментованість даних. Сучасний користувач часто обслуговується одразу в кількох фінансових установах, маючи різні рахунки для отримання доходу, заощаджень, кредитування чи повсякденних витрат. Традиційний підхід, який полягає у ручному зведенні балансу, веденні електронних таблиць або постійному перемиканні між окремими банківськими застосунками, є вкрай неефективним. Це не лише займає багато часу та підвищує когнітивне навантаження на користувача, але й неминуче призводить до помилок в обліку. Більше того, розрізнені банківські додатки надають аналітику лише в межах своєї екосистеми, що унеможливлює отримання цілісної картини фінансового стану.

Особливого загострення описана проблема фрагментованості набуває в умовах мультивалютності фінансових активів. Користувачі дедалі частіше оперують коштами у декількох валютах одночасно. Ручний підрахунок консолідованого балансу за таких обставин стає надзвичайно складною задачею. Спроби самостійного ведення мультивалютного обліку призводять до серйозних аналітичних спотворень. Статичні інструменти ручного введення принципово не здатні алгоритмізувати такий рівень динамічності даних. У результаті, без автоматичної синхронізації та приведення транзакцій до єдиної базової валюти,

користувач повністю втрачає можливість об'єктивно оцінювати свій чистий капітал та купівельну спроможність.

Вирішенням цієї проблеми є створення консолідованих платформ з використанням концепції відкритого банкінгу та сучасних технологій веб-розробки. Інтеграція з прикладними програмними інтерфейсами різних банківських систем дозволяє автоматизувати збір фінансової інформації в єдиному середовищі. Такий підхід усуває необхідність ручного введення даних, забезпечує оновлення балансу в режимі реального часу та відкриває широкі можливості для глибокого аналізу грошових потоків.

Актуальність теми зумовлена об'єктивною необхідністю створення зручного, швидкого та безпечного інструменту для агрегації фінансових даних в умовах мульти-банківського обслуговування.

Метою кваліфікаційної роботи є розробка веб-застосунку для аналізу та візуалізації особистих фінансів.

Для досягнення поставленої мети передбачається виконання наступних завдань:

1) дослідження предметної області систем управління особистими фінансами з системно-технологічної точки зору для структурування процесів збору, нормалізації та обчислення грошових потоків як цілісного бізнес-процесу;

2) аналіз існуючих програмних рішень для ведення фінансового обліку з метою виявлення проблеми фрагментації даних та обґрунтування доцільності використання технологій відкритого банкінгу;

3) постановка задачі та формування ключових функціональних і нефункціональних вимог до розроблюваного програмного продукту;

4) проектування загальної архітектури системи за шаблоном MVC, а також розробка її алгоритмічного та інформаційного забезпечення (включаючи концептуальну, даталогічну та фізичну моделі бази даних);

5) реалізація клієнт-серверної системи з використанням сучасних технологій веб-розробки (фреймворк Laravel, СУБД MySQL) та створення адаптивного інтерфейсу з інтерактивними аналітичними панелями;

6) інтеграція розробленого програмного забезпечення із зовнішніми прикладними програмними інтерфейсами (платформою відкритого банкінгу та сервісом обмінних курсів) для автоматизованого імпорту й мультивалютної нормалізації транзакцій;

7) проведення модульного та функціонального тестування компонентів системи для перевірки її стабільності, цілісності бази даних та коректності бізнес-логіки.

Результати роботи опубліковано в матеріалах міжнародної науково-практичної конференції студентів і молодих вчених «ICSPM 2026: Інтелектуальні обчислювальні системи та управління проєктами», м. Тернопіль, 21–22 травня 2026 р. (додаток Е).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧІ ПРОЕКТУВАННЯ

1.1 Опис систем управління особистими фінансами

Предметна область розроблюваного програмного продукту належить до динамічної та стрімко зростаючої сфери фінансових технологій, а саме - до систем управління особистими фінансами. Вона охоплює комплексні процеси збору, нормалізації, безпечного зберігання, математичної обробки та візуального аналізу інформації про грошові потоки кінцевого користувача [1, 2]. У сучасних умовах інформатизації суспільства особисті фінанси перестали бути просто сумою готівки; вони перетворилися на безперервний потік цифрових даних, який потребує систематизації.

Специфіка сучасної фінансової поведінки полягає в тому, що користувачі активно диверсифікують свої фінансові ризики та оптимізують вигоди, користуючись послугами одразу кількох банківських чи кредитних установ [3]. Наприклад, заробітна плата може надходити на картку одного банку, для повсякденних розрахунків та використання кредитного ліміту чи кешбеку використовується інший, а для довгострокових накопичень або валютних депозитів - третій. У такій розгалуженій екосистемі головною проблемою предметної області стає критична фрагментація даних. Користувач позбавлений єдиної точки доступу, де б він міг побачити свій реальний сукупний баланс, проаналізувати загальну структуру витрат та оцінити загальний фінансовий стан без виконання рутинних математичних операцій [4].

Історично ця проблема вирішувалася шляхом ручного ведення обліку в електронних таблицях (Excel, Google Sheets) або використання мобільних додатків першого покоління, які покладалися на перехоплення та парсинг SMS-повідомлень чи push-сповіщень від банків. Проте такі архітектурні підходи сьогодні вважаються застарілими та вкрай ненадійними [5]. Ручне введення вимагає високого рівня самодисципліни, забирає багато часу і неминуче призводить до втрати даних або людських помилок. З іншого боку, парсинг

текстових сповіщень є нестабільним: він ламається при будь-якій, навіть мінімальній, зміні текстового шаблону з боку банку, залежить від мовних налаштувань пристрою та не працює без стабільного підключення до мобільної мережі в момент здійснення транзакції.

Сучасний етап розвитку предметної області фундаментально спирається на концепцію відкритого банкінгу та стандартизоване використання прикладних програмних інтерфейсів. Цей підхід дозволяє на програмному рівні встановлювати безпечне, зашифроване з'єднання безпосередньо з серверами фінансових установ (наприклад, Monobank, ПриватБанк тощо) [6]. Завдяки прикладним програмним інтерфейсам система отримує фінансові виписки у чітко структурованому форматі (переважно JSON або XML), що гарантує точність даних та незалежність від клієнтського пристрою.

Робота в цій сфері вимагає глибокого розуміння фінансової бізнес-логіки та впровадження суворих протоколів роботи з чутливими даними. Розробка програмного забезпечення в межах цієї предметної області базується на об'єктно-орієнтованому підході та вимагає вирішення низки специфічних архітектурних завдань [7]:

1) нормалізація та уніфікація даних: оскільки кожен банк проектує власну архітектуру API і повертає масиви даних у різних форматах (різні назви полів, формати дат UNIX timestamp проти ISO 8601, представлення сум у копійках чи гривнях), бекенд розроблюваної системи повинен реалізовувати патерни проектування (наприклад, Adapter або Factory) [8]. Це дозволить приводити різнорідні JSON-відповіді до єдиного об'єктного представлення (Data Transfer Objects) перед їх збереженням у реляційну базу даних;

2) алгоритмічна категоризація та мапінг транзакцій: для забезпечення автоматичного розподілу витрат система інтегрується з аналітичними рушіями платформ відкритого банкінгу. Замість локальної обробки банківських кодів (MCC), програмний додаток отримує попередньо класифіковані транзакційні масиви. Завданням бекенду є реалізація механізмів мапінгу (відображення) зовнішніх категорій на внутрішню нормалізовану структуру об’єктів (наприклад,

за допомогою структур Enum у PHP) [9]. Це дозволяє групувати десятки дрібних банківських міток у глобальні категорії (наприклад, "Auto & Transport", "Health & Fitness"), залишаючи користувачу можливість ручного перевизначення категорії для окремих транзакцій;

3) мультивалютна нормалізація: у зв'язку з тим, що користувач може мати рахунки у різних валютах, система реалізує алгоритм приведення транзакцій до єдиної базової валюти. Для цього бекенд взаємодіє зі сторонніми API актуальних валютних курсів, кешуючи результати запитів для оптимізації продуктивності [10]. Конвертація відбувається на етапі збереження транзакції в базу даних, що гарантує цілісність аналітичного масиву та дозволяє уникати математичних колізій при агрегації загального балансу;

4) обробка специфічних фінансових подій. Предметна область вимагає реалізації чітких алгоритмів для уникнення дублювання коштів або викривлення статистики. Система повинна вміти програмно ідентифікувати перекази між власними рахунками користувача (щоб не рахувати їх як дохід і витрату одночасно), правильно обробляти банківські комісії, конвертацію валют за крос-курсом, а також скасування платежів та повернення коштів;

5) криптографічна безпека та конфіденційність. Оскільки система маніпулює фінансовою інформацією, критичним аспектом є захист даних. Архітектура передбачає надійне симетричне або асиметричне шифрування клієнтських токенів доступу (API-ключів) на рівні бази даних та додатку [11]. Це гарантує, що у випадку потенційної компрометації серверної інфраструктури, зловмисники отримають лише зашифровані хеші, що унеможливить несанкціонований доступ до банківських виписок користувачів.

Оскільки ядром розроблюваної системи є агрегація великих масивів транзакційних даних, ключову цінність становить їх подальший математичний та статистичний аналіз [12]. Варто зазначити, що класичні корпоративні методи фінансового аналізу (наприклад, розрахунок рентабельності активів чи EBITDA) не є релевантними для даної предметної області. Цільовим користувачем є фізична

особа, яка прагне оптимізувати побутові витрати. Тому в межах платформи реалізуються такі адаптовані методи аналізу:

1) аналіз ліквідності та щоденної динаміки (Daily Spend Trend). Алгоритм агрегує транзакції за допомогою SQL-функцій у розрізі кожного дня та обчислює середньоденний показник витрат (Average Daily Spend) за обраний період. Візуалізація цієї метрики у вигляді гістограми дозволяє користувачу відстежувати швидкість виснаження фінансових ресурсів (Burn Rate) та оперативно виявляти аномальні сплески витрат [13];

2) обчислення рівня заощаджень (Savings Rate). Це фундаментальна метрика персональних фінансів, яка визначає відсоткове співвідношення між чистим прибутком (різницею між загальними надходженнями та витратами) і сумарним доходом. Серверна частина системи динамічно розраховує цей показник залежно від застосовуваних часових фільтрів. Результат виводиться на клієнтську частину за допомогою інтерактивного радіального індикатора, що слугує головним ключовим показником ефективності фінансової стабільності користувача [14];

3) структурний аналіз грошових потоків (Expenses Distribution). Система виконує комплексні запити до бази даних із застосуванням умов агрегації для підсумовування витрат за кожною макро-категорією [15]. На основі цих обчислень формується структурна кільцева діаграма, яка дозволяє користувачу миттєво оцінити пропорції розподілу власного бюджету та виявити найбільш ресурсоємні статті витрат для їх подальшої оптимізації [16].

1.2 Огляд і аналіз існуючих рішень

Глобальний та локальний ринки програмного забезпечення для управління особистими фінансами є висококонкурентними. Доступні програмні продукти використовують різні парадигми взаємодії з користувачем, архітектурні рішення та підходи до збору даних. Для обґрунтування актуальності розроблюваного застосунку необхідно провести порівняльний аналіз існуючих аналогів. Їх можна

класифікувати на три основні макрогрупи залежно від рівня автоматизації та архітектури:

1) автономні додатки з ручним введенням даних. До цієї класичної категорії належать такі відомі на ринку системи, як базові версії CoinKeeper, Spendee, YNAB (You Need A Budget) та Money Manager & Expenses (рисунок 1.1).

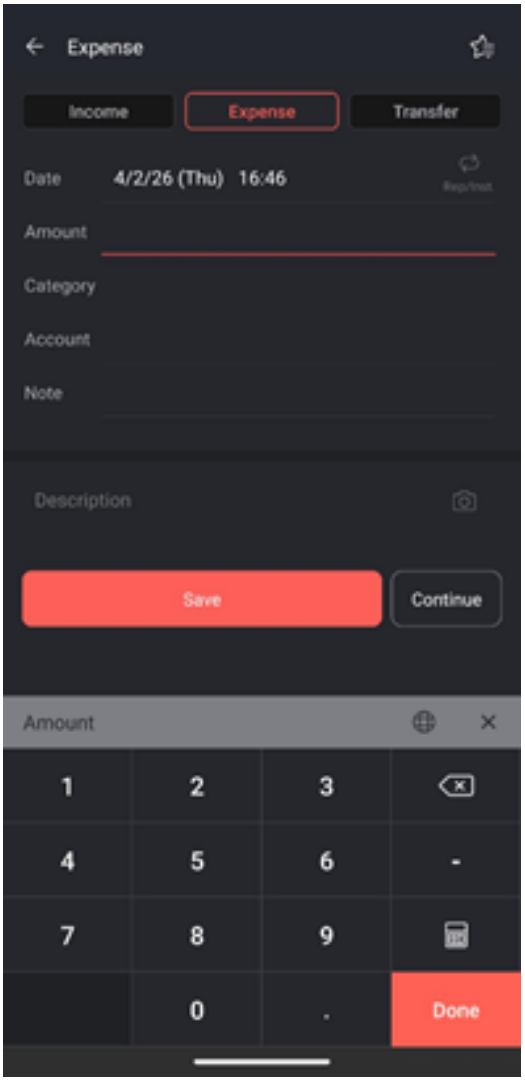

Рисунок 1.1 – Екран створення запису про транзакцію в Money manager & expenses

Архітектура таких додатків передбачає, що користувач виступає єдиним джерелом даних. Він повинен самостійно створювати віртуальні рахунки та після кожної фінансової операції відкривати додаток, щоб внести суму, обрати категорію та вказати дату:

– переваги: високий рівень ізоляції та конфіденційності (додаток не має зв'язку з реальними банківськими рахунками). Можливість вести облік готівки, екзотичних валют або криптоактивів. Деякі додатки (наприклад, YNAB) пропонують потужну методологію планування бюджету за методом конвертів;

– недоліки: головна проблема - критична залежність від людського фактора. Внесення транзакцій перетворюється на рутину, що створює високе когнітивне навантаження. З часом користувач забуває вносити дрібні витрати, що призводить до розбіжності між віртуальним балансом у додатку та реальним станом рахунку. Як наслідок, статистика стає нерелевантною, і сенс використання системи втрачається. До прикладу, користуючись додатком Money Manager & Expenses (рисунок 1.1), стабільною проблемою є втрата цілісності даних через нерегулярне внесення записів.

2) Вбудовані аналітичні модулі банківських клієнтів. У відповідь на запит користувачів більшість сучасних необанків та традиційних установ (Monobank, Sense Bank, ПриватБанк) інтегрували модулі аналітики безпосередньо у свої мобільні додатки. Дані в таких системах формуються на стороні сервера банку і відображаються клієнту у вигляді готових графіків.

– переваги: нульове когнітивне навантаження для користувача - транзакції категоризуються та з'являються в статистиці миттєво і повністю автоматично. Абсолютна точність даних та відповідність балансу;

– недоліки: архітектурна ізольованість. Ця аналітика обмежена екосистемою лише одного банку. Якщо користувач купує продукти з картки ПриватБанку, а пальне оплачує карткою Monobank, він ніколи не побачить загальної картини своїх витрат. Крім того, банківські додатки зазвичай мають жорстко зафіксоване дерево категорій без можливості створення власних кастомних розділів або обліку готівкових коштів. На рисунку 1.2 представлено приклад такої внутрішньої статистики в додатку Приват24.

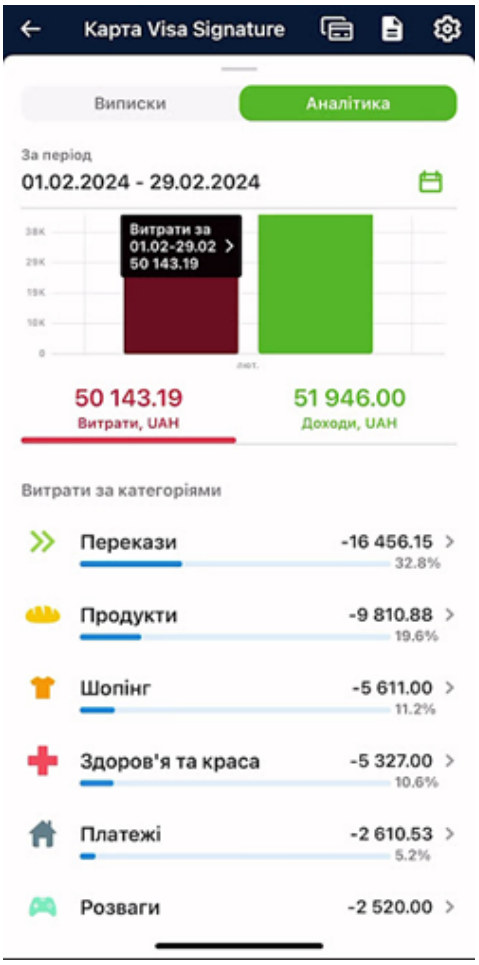

Рисунок 1.2 – Приклад статистики про витрати в додатку Приват24

3) універсальні фінансові агрегатори (Open Banking додатки). Це найбільш технологічно складний тип інформаційних систем, до якого належать Wallet by BudgetBakers, Toshl Finance, Zenmoney. Вони розробляються з метою об'єднання виписок з безлічі різних джерел у єдиному стандартизованому інтерфейсі. Такі додатки використовують або прямі інтеграції через банківські API, або послуги посередників-агрегаторів фінансових даних (наприклад, Salt Edge).

– переваги: надають найбільш повну і об'єктивну картину фінансового стану. Автоматизують процес збору даних, при цьому дозволяючи гнучко налаштовувати категорії, теги, створювати спільні бюджети для сім'ї;

– недоліки: головною проблемою для українського сегменту користувачів є обмежена підтримка локальних банків. Багато глобальних сервісів повільно оновлюють інтеграції або вимагають дорогої щомісячної підписки за функцію синхронізації (Premium feature). Крім того, деякі з них

досі використовують застарілі та небезпечні методи отримання даних (наприклад, запит логіна та пароля від інтернет-банкінгу клієнта для екранного скрейпінгу). На рисунку 1.3 наведено інтерфейс Wallet by BudgetBakers.

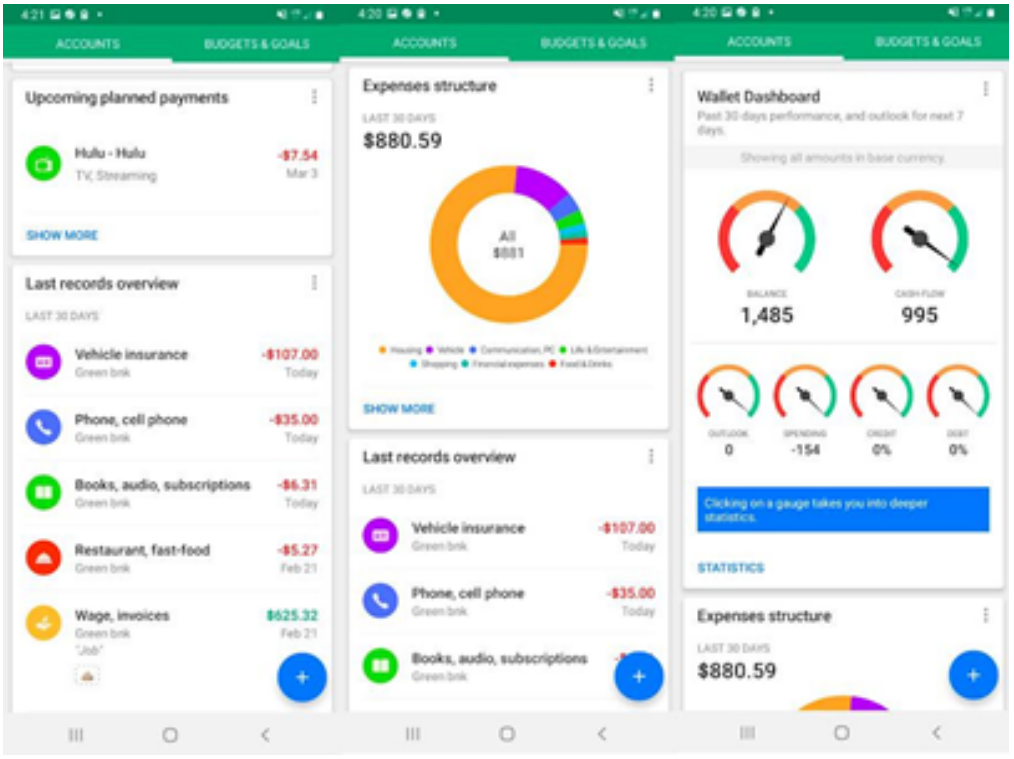

Рисунок 1.3 – Приклад статистики та історії транзакцій в Wallet by BudgetBakers

Висновки з огляду ринку. Проведений системний аналіз виявив наявність чіткої проблеми: користувачі змушені вибирати між точними, але розрізненими банківськими програмами, застосунками ручного обліку, що вимагають надмірних зусиль, та громіздкими закордонними платформами збору даних із високою вартістю передплати і обмеженою підтримкою вітчизняних установ. Відповідно, на ринку існує об'єктивна потреба у розробці альтернативного програмного рішення, яке б ефективно заповнило цю прогалину. Проектований застосунок має стати збалансованою екосистемою: він повинен поєднати переваги безшовної автоматизованої синхронізації через протоколи відкритого банкінгу із гнучкістю ручного керування фізичними готівковими активами. Такий підхід дозволить

ліквідувати фрагментацію фінансової інформації, звести до мінімуму рутину введення даних та надати користувачеві єдину, цілісну картину його бюджету.

Розроблюваний веб-застосунок проектується як комбіноване рішення, що поєднує найкращі архітектурні підходи досліджених систем і нівелює їхні недоліки. На відміну від наявних аналогів, розроблена система пропонує такі конкурентні переваги:

1) комбінована модель збору даних: застосунок об'єднує автоматизоване отримання інформації через європейську платформу відкритого банкінгу для цифрових карток із можливістю створення рахунків ручного керування для обліку фізичної готівки в межах єдиного інтерфейсу;

2) вбудована мультивалютність: інтегровано модуль автоматичного перерахунку фінансових операцій через зовнішні програмні інтерфейси. Це дозволяє зводити залишки з різних банків та готівкових рахунків у різних валютах до єдиного базового еквівалента;

3) адаптивна аналітика: на противагу статичним графікам банківських додатків, система генерує інтерактивні інформаційні панелі з можливістю гнучкого відбору даних. Вони візуалізують не лише витрати, а й стратегічні показники: швидкість використання коштів та загальний рівень заощаджень.

Такий комплексний підхід робить розроблений застосунок швидким, безпечним та незалежним інструментом, що забезпечує користувача єдиним достовірним джерелом інформації щодо його особистих фінансів. Крім того, надання користувачеві повного контролю над власними грошовими потоками сприяє підвищенню рівня його фінансової грамотності та дисципліни. Завдяки інтуїтивно зрозумілому інтерфейсу поріг входження зводиться до мінімуму, усуваючи необхідність вивчення складних бухгалтерських концепцій для щоденного обліку. Водночас використання сучасних механізмів криптографічного захисту гарантує абсолютну конфіденційність чутливої інформації на всіх етапах її обробки. Запропонована архітектура програмного рішення закладає надійний фундамент для його подальшого масштабування та вдосконалення.

1.3 Постановка задачі на розробку програмного продукту

Метою кваліфікаційної роботи є розробка веб-застосунку для аналізу та візуалізації особистих фінансів. Рішення базується на принципах взаємодії з платформами відкритого банкінгу та використанні комбінованої моделі збору даних. Кінцевий продукт має розв'язати проблему розрізненості фінансової інформації, усунути рутинні процеси ручного введення та надати користувачеві інтуїтивно зрозумілу аналітичну панель для моніторингу метрик його особистого бюджету. Для успішного досягнення цієї мети проект розбивається на логічні етапи, що відповідають життєвому циклу розробки програмного забезпечення. Необхідно виконати такий комплекс взаємопов'язаних теоретичних та інженерно-практичних завдань:

1) проектування архітектури системи та реляційної бази даних:

– розробити концептуальну, логічну та фізичну моделі бази даних із дотриманням принципів третьої нормальної форми для уникнення надмірності інформації;

– спроектувати структури для керування основними сутностями системи: користувачами, гібридними рахунками для цифрових та готівкових активів, ієрархічним деревом категорій та записами про фінансові операції;

– забезпечити вибір оптимальних типів даних, критично важливих для фінансових систем, зокрема використання цілочисельних форматів для зберігання грошових сум з метою уникнення математичних похибок під час операцій з рухомою комою.

2) розробка серверної частини та інтеграція з API:

– побудувати надійне об'єктно-орієнтоване ядро системи на базі архітектурного шаблону «Модель-Вигляд-Контролер», використовуючи сучасний каркас веб-розробки мовою PHP;

– провести аналіз технічної документації та налаштувати безпечну взаємодію з прикладним програмним інтерфейсом відкритого банкінгу для автоматичного імпорту банківських виписок;

– розробити сервісні класи для встановлення захищених мережевих з'єднань із сервісом валютних курсів для забезпечення мультивалютності системи;

– впровадити криптографічні механізми захисту для безпечного зберігання токенів доступу користувачів, унеможливлюючи їх компрометацію.

3) розробка бізнес-логіки обробки та агрегації даних:

– написати алгоритми адаптації, які перетворюватимуть зовнішні формати даних від платформи відкритого банкінгу в єдиний стандартизований об'єкт транзакції всередині системи;

– реалізувати автоматизований модуль категоризації, який програмно зіставлятиме зовнішні мітки фінансових операцій із внутрішнім нормалізованим переліком категорій, залишаючи можливість їх ручного редагування;

– впровадити алгоритм автоматичного зведення транзакцій до єдиної базової валюти на етапі їх збереження, використовуючи кешовані дані про актуальні обмінні курси.

4) розробка клієнтської частини та візуалізація аналітики:

– спроектувати сучасний адаптивний інтерфейс користувача із застосуванням каскадних таблиць стилів та реактивних компонентів керування станом;

– реалізувати підсистему автентифікації, сторінки управління гібридними рахунками та інтерактивні таблиці для перегляду історії операцій із можливістю гнучкого фільтрування;

– створити деталізовані сторінки для кожного рахунку з розширеним відображенням фінансових реквізитів, поточних залишків та інтерактивною історією операцій із можливістю гнучкого фільтрування;

– інтегрувати графічні бібліотеки для побудови аналітичної панелі.

2 АРХІТЕКТУРА, АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ВЕБ-ЗАСТОСУНКУ ДЛЯ АНАЛІЗУ ТА ВІЗУАЛІЗАЦІЇ ОСОБИСТИХ ФІНАНСІВ

2.1 Архітектура проектованого веб-застосунку

Проектований веб-застосунок побудований за класичною трирівневою клієнт-серверною архітектурою та базується на фундаментальному програмному шаблоні «Модель-Вигляд-Контролер» (MVC) [17]. Такий архітектурний підхід забезпечує чітке розділення відповідальності між компонентами: відокремлює бізнес-логіку від інтерфейсу користувача та механізмів взаємодії з базою даних. Це гарантує високу масштабованість веб-застосунку, спрощує його подальшу підтримку та тестування, а також дозволяє незалежно модифікувати окремі модулі без впливу на інші частини архітектури веб-застосунку. На рисунку 2.1 зображено загальну архітектуру, яка демонструє взаємодію цих ключових рівнів.

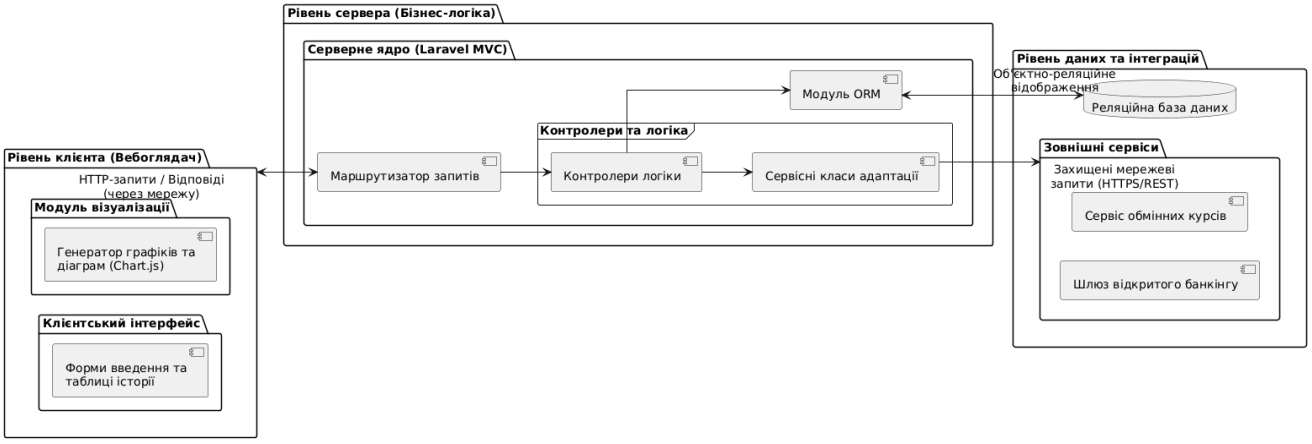

Рисунок 2.1 – Архітектура веб-застосунку

Першим компонентом архітектури є клієнтський рівень (рівень представлення), який відповідає за візуальну взаємодію з кінцевим користувачем. Він реалізований за допомогою сучасних мов розмітки, каскадних таблиць стилів та сценаріїв, що виконуються безпосередньо у веб-браузері. Цей рівень забезпечує відображення інтерактивних аналітичних панелей, форм введення даних для створення рахунків чи ручного додавання фінансових операцій, а також таблиць із

розширеною історією переказів. Для оптимізації швидкодії та зменшення навантаження на мережу використовуються технології асинхронного обміну даними [18]. Візуалізація статистичної інформації, зокрема структурного розподілу витрат та динаміки щоденного балансу, виконується на боці клієнта за допомогою спеціалізованих графічних бібліотек, що перетворюють масиви чисел у наочні діаграми [19].

Центральною ланкою архітектури веб-застосунку є серверний рівень (рівень бізнес-логіки), розроблений мовою PHP з використанням сучасного об'єктно-орієнтованого каркаса веб-розробки [20]. Цей рівень виконує функції контролера: приймає мережеві запити від клієнтської частини, здійснює перевірку прав доступу (автентифікацію та авторизацію), перевіряє коректність вхідних даних та керує потоками інформації. Саме тут реалізовано ключові алгоритми застосунку: автоматичний розподіл транзакцій за категоріями, розрахунок загального рівня заощаджень та алгоритмічна нормалізація даних. Крім того, серверний рівень містить відокремлені сервісні модулі для встановлення захищених з'єднань із зовнішніми системами. Зокрема, реалізовано модуль взаємодії із зовнішнім програмним інтерфейсом обмінних курсів для забезпечення вбудованої мультивалютності.

Третім, фундаментальним компонентом є рівень доступу до даних, який базується на реляційній системі управління базами даних. Взаємодія серверного ядра з фізичними таблицями відбувається не через прямі мовні запити, а за допомогою механізму об'єктно-реляційного відображення (ORM) [21]. Це дозволяє працювати з інформацією бази даних як з об'єктами мови програмування (Моделями), що значно підвищує рівень криптографічної безпеки застосунку, автоматично нівелюючи ризики впровадження шкідливого коду. Моделі також інкапсулюють правила формування зв'язків між сутностями системи (наприклад, зв'язок між користувачем, його гібридними рахунками та відповідними витратами). Такий підхід забезпечує підтримку цілісності, консистентності та нормалізації фінансової інформації на етапі її довгострокового збереження чи агрегованої вибірки для побудови звітів.

2.2 Алгоритмічне забезпечення веб-застосунку

Алгоритмічне забезпечення розробленого веб-застосунку становить комплекс математичних, логічних та сервісних процедур, спрямованих на автоматизацію процесів збору, перетворення та аналізу фінансової інформації. Головною метою цих алгоритмів є забезпечення цілісності даних, уникнення дублювання записів та підготовка масивів інформації для подальшої візуалізації на клієнтському рівні. У межах архітектури реалізовано кілька ключових обчислювальних блоків.

Першим є алгоритм безпечної інтеграції та нормалізації даних. Він ініціюється під час синхронізації з платформою відкритого банкінгу та виконує послідовну обробку вхідних масивів. Алгоритм перевіряє чинність криптографічних ключів доступу, здійснює синтаксичний аналіз отриманих структур даних та перетворює зовнішні атрибути фінансових операцій у стандартизовані об'єкти [22]. На цьому етапі діють алгоритми фільтрації та перевірки унікальності записів (за ідентифікаторами транзакцій) для запобігання їх повторному збереженню в базу даних у випадку переривання мережевого з'єднання.

Другим виступає алгоритм автоматизованої категоризації та відображення фінансової операції (рисунок 2.2). Під час надходження нового запису про витрату система аналізує метадані операції. Алгоритм виконує пошук відповідностей між зовнішніми мітками призначення платежу та внутрішнім ієрархічним довідником, реалізованим на базі перелічуваних типів даних. У разі успішного збігу транзакції автоматично присвоюється відповідна макро-категорія. Якщо збіг відсутній, алгоритм призначає базову категорію за замовчуванням, зберігаючи при цьому початковий опис транзакції для можливості подальшого ручного редагування користувачем.

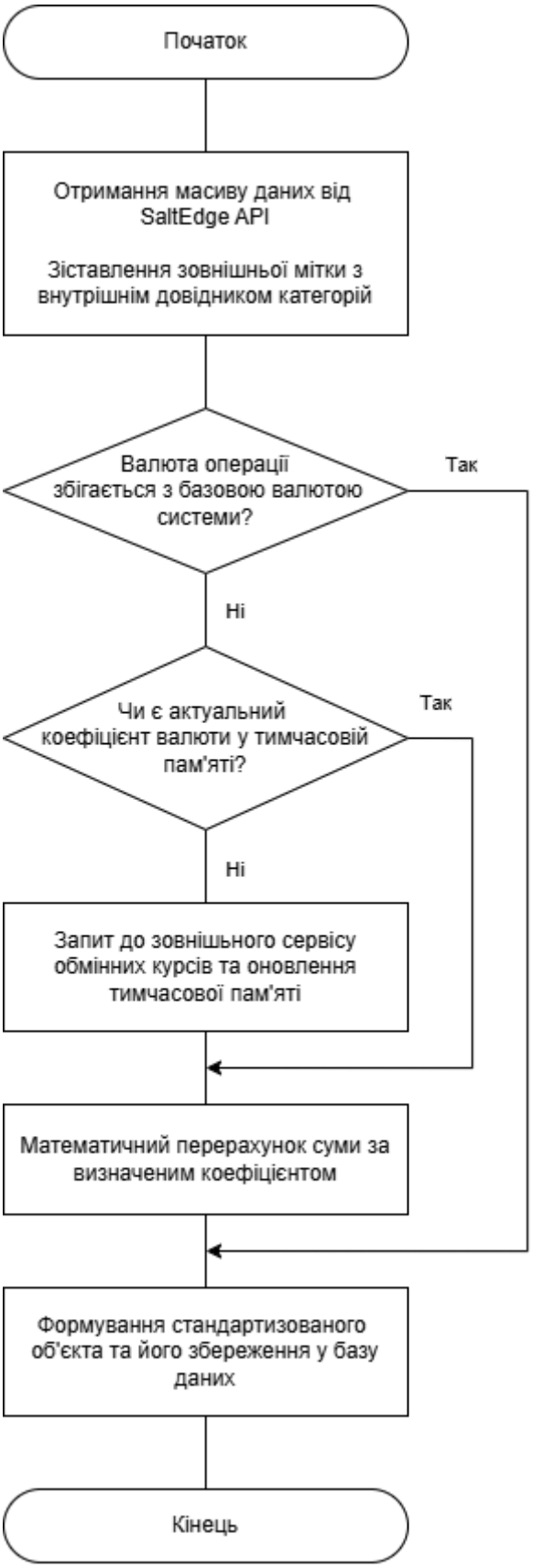

Рисунок 2.2 – Схема алгоритму автоматизованої категоризації та відображення фінансової операції

Третім критично важливим компонентом є алгоритм мультивалютної нормалізації. Оскільки застосунок підтримує ведення гібридних рахунків у різних валютах, система зобов'язана зводити всі розрахунки до єдиного базового еквівалента. Під час збереження фінансової операції алгоритм порівнює валюту транзакції з базовою валютою системи. У разі розбіжності ініціюється звернення до зовнішнього програмного інтерфейсу обмінних курсів. Для оптимізації швидкодії та зменшення кількості мережевих запитів використовується механізм тимчасового збереження (кешування) поточних курсів. Після отримання актуального коефіцієнта алгоритм виконує математичний перерахунок та фіксує обидва значення (оригінальне та конвертоване) для забезпечення математичної точності аналітики.

Четвертий блок становить алгоритмічне забезпечення аналітичної панелі, яке відповідає за агрегацію оброблених даних. Він включає алгоритм обчислення щоденної динаміки (групування транзакцій за датами та розрахунок середньоденного показника витрат), алгоритм структурного аналізу (підсумовування обсягів витрат у розрізі кожної категорії для побудови пропорційних діаграм) та алгоритм розрахунку рівня заощаджень (визначення відсоткового співвідношення між чистим залишком та сукупним доходом за обраний період). Ці обчислювальні процедури виконуються на стороні сервера з використанням оптимізованих агрегатних запитів до бази даних, після чого результати передаються на рівень представлення для побудови інтерактивних графіків.

Для програмної реалізації визначених алгоритмів та побудови архітектури було застосовано комплекс наукових та інженерних методів. Їх використання дозволило забезпечити високу надійність, продуктивність та захищеність розробленого застосунку.

Метод системного аналізу застосовано на етапі дослідження предметної області та вивчення технічної документації зовнішніх програмних інтерфейсів. Це дозволило декомпозивати складний процес синхронізації банківських даних на

окремі логічні підзадачі та спроектувати оптимальну схему обміну інформацією між сервером та шлюзом відкритого банкінгу.

Об'єктно-орієнтований метод проектування покладено в основу розробки серверного ядра веб-застосунку. Використання принципів інкапсуляції, успадкування та поліморфізму дозволило створити гнучку архітектуру класів, де кожна сутність (рахунок, фінансова операція, категорія) представлена окремим програмним об'єктом із власним набором властивостей та методів обробки.

Методи математичної статистики та агрегації використано для реалізації аналітичної підсистеми. Завдяки застосуванню методів зведення часових рядів, розрахунку середньоарифметичних величин та обчислення відносних частот, система здатна генерувати динамічні показники швидкості витрат та формувати структурні пропорції для побудови графічних діаграм.

Методи криптографічного захисту інформації застосовано для забезпечення конфіденційності даних користувачів. Використання алгоритмів симетричного шифрування на рівні бази даних гарантує безпечне зберігання ключів доступу та унеможливлює витік чутливої фінансової інформації навіть у випадку несанкціонованого доступу до фізичних носіїв сервера.

2.3 Інформаційне забезпечення веб-застосунку

2.3.1 Загальна структура інформаційного забезпечення та інтеграція даних

Інформаційне забезпечення розробленого веб-застосунку являє собою сукупність рішень щодо обсягів, розміщення і форм організації інформації, що циркулює в системі. Основу інформаційного забезпечення становить реляційна база даних, яка функціонує під управлінням системи керування базами даних. Вибір реляційної моделі зумовлений необхідністю забезпечення суворої структурованості фінансових записів та підтримки транзакційної цілісності даних під час довгострокового зберігання.

Логічна структура бази даних проектується навколо кількох базових сутностей, пов'язаних між собою реляційними відношеннями. Головною сутністю

є «Користувач», яка містить ідентифікаційні дані для авторизації. З нею зв'язком «один-до-багатьох» поєднана сутність «Рахунок», що зберігає інформацію про гібридні фінансові активи (як створені вручну, так і синхронізовані через банк), їхні поточні баланси, валюти та зашифровані маркери доступу. Сутність «Фінансова операція» є центральним елементом накопичення даних, яка пов'язана з конкретним рахунком та сутністю «Категорія» (нормалізованим довідником для класифікації витрат). З метою уникнення похибок заокруглення під час математичних обчислень, усі грошові суми зберігаються в базі даних виключно у вигляді цілих чисел (у найменших одиницях валюти), що є строгим галузевим стандартом проектування фінансових інформаційних систем.

Важливою складовою інформаційного забезпечення є зовнішні джерела даних, взаємодія з якими відбувається через захищені мережеві протоколи. Для автоматизованого наповнення бази даних використовуються структуровані відповіді від європейської платформи відкритого банкінгу. Обмін інформацією здійснюється у стандартизованому текстовому форматі обміну даними (JSON), що дозволяє серверному ядру ефективно розбирати масиви транзакцій, вилучати метадані (опис, суму, дату, ідентифікатор отримувача) та зберігати їх у відповідні таблиці. Додатковим інформаційним джерелом виступає зовнішній сервіс валютних курсів, який постачає актуальні коефіцієнти для функціонування модуля мультивалютної нормалізації.

Для гарантування цілісності та узгодженості інформації на рівні бази даних застосовуються механізми зовнішніх ключів та каскадних обмежень. Це забезпечує неможливість існування фінансової операції без прив'язки до дійсного рахунку, а видалення облікового запису користувача призводить до коректного очищення всіх пов'язаних із ним фінансових даних. Крім того, на рівні інформаційного забезпечення реалізовано алгоритми безпеки: конфіденційні ключі доступу до банківських установ зберігаються виключно у зашифрованому вигляді, що відповідає сучасним вимогам захисту персональної інформації.

2.3.2 Опис вхідної та вихідної інформації, яка обробляється в межах функцій предметної області, що автоматизується

Функціонування розробленої архітектури базується на безперервному циклі отримання, перетворення та видачі даних. Всю інформацію, що циркулює в межах автоматизованих функцій предметної області, можна розділити на два основні потоки: вхідний (первинні дані) та вихідний (результатна аналітична інформація).

Вхідна інформація формується з двох незалежних джерел: ручного введення та автоматизованого імпорту. До першої категорії належать аутентифікаційні дані користувача (електронна пошта, зашифрований пароль), параметри створюваних рахунків (назва, тип активу, базова валюта) та реквізити фінансових операцій ручного обліку (сума, дата, категорія, текстовий опис). Друга категорія вхідних даних - це машинозчитувані масиви інформації, що надходять від зовнішніх програмних інтерфейсів у текстовому форматі обміну даними (JSON). Від платформи відкритого банкінгу система отримує первинні банківські виписки, які включають ідентифікатори транзакцій, оригінальні суми, валюти розрахунку, призначення платежів та мітки категорій. Від сервісу валютних курсів надходять актуальні числові коефіцієнти для конвертації іноземних валют. Уся ця інформація є первинною і потребує обов'язкової алгоритмічної обробки.

Вихідна інформація є результатом роботи серверного ядра і становить собою нормалізовані, математично оброблені та агреговані дані, готові для сприйняття кінцевим користувачем. До неї належать: уніфіковані списки історії транзакцій (де всі операції приведені до єдиного стандарту незалежно від рахунку), розраховані поточні залишки на гібридних рахунках та конвертовані суми у єдиній базовій валюті. Найважливішим типом вихідної інформації є структуровані набори даних для генерації інтерактивної аналітичної панелі. Це масиви розрахованих показників (середньоденні витрати, швидкість використання коштів, відсоток заощаджень) та згруповані пропорції витрат за макро-категоріями, які система передає на рівень представлення для побудови кільцевих діаграм та гістограм.

2.3.3 Концептуальне інфологічне проектування

Метою концептуального інфологічного проектування є створення семантичної моделі предметної області, яка відображає інформаційні потреби веб-застосунку незалежно від обраної фізичної системи управління базами даних. Процес проектування передбачає виокремлення ключових інформаційних об'єктів (сутностей), їхніх властивостей (атрибутів) та встановлення логічних зв'язків між ними. Цей етап реалізується шляхом побудови локальних інфологічних моделей із їх подальшим об'єднанням у єдину глобальну модель.

Локальні інфологічні моделі відображають уявлення про дані з погляду окремих функціональних складових елементів застосунку. У межах розроблюваної архітектури виокремлено три базові локальні моделі:

– локальна модель модуля ідентифікації. Базовою сутністю є «Користувач». Вона інкапсулює атрибути, необхідні для автентифікації та персоналізації сеансу в застосунку: унікальний ідентифікатор, адресу електронної пошти, криптографічний хеш пароля та часові мітки створення облікового запису;

– локальна модель модуля фінансових активів. Описує сутність «Рахунок», яка є абстракцією реальних місць зберігання коштів. Атрибутивний склад включає: ідентифікатор рахунку, назву, тип активу (готівковий або цифровий/банківський), трилітерний код базової валюти, поточний фінансовий залишок та зашифрований маркер доступу до зовнішнього банківського інтерфейсу (за наявності).;

– локальна модель модуля обліку та категоризації. Охоплює дві сутності: «Категорія» та «Фінансова операція». Сутність «Категорія» виконує роль довідника і містить ідентифікатор, назву та візуальні параметри (колір, іконка). Сутність «Фінансова операція» фіксує сам факт руху коштів та містить розширений набір атрибутів: суму (у цілочисельному форматі), дату виконання, текстовий опис призначення платежу та мітку напрямку (витрата або надходження).

Глобальна інфологічна модель утворюється шляхом інтеграції перелічених локальних моделей і розв'язання можливих семантичних конфліктів чи

дублювання даних. У глобальній моделі встановлюються структурні зв'язки, що забезпечують цілісність фінансової аналітики.

Основою глобальної моделі є ієрархія підпорядкування. Сутність «Користувач» виступає головним вузлом, з яким сутність «Рахунок» пов'язана відношенням «один-до-багатьох». Своєю чергою, сутність «Рахунок» вступає у зв'язок «один-до-багатьох» із сутністю «Фінансова операція».

2.3.4 Проектування даталогічної моделі даних

Перехід від концептуального опису предметної області до її практичної реалізації вимагає розробки даталогічної (логічної) моделі даних. Цей етап передбачає трансформацію абстрактних сутностей у конкретні реляційні таблиці, визначення типів полів, встановлення первинних і зовнішніх ключів, а також налаштування обмежень цілісності інформації. На рисунку 2.3 зображено даталогічну модель даних у вигляді діаграми зв'язків між таблицями реляційної бази даних.

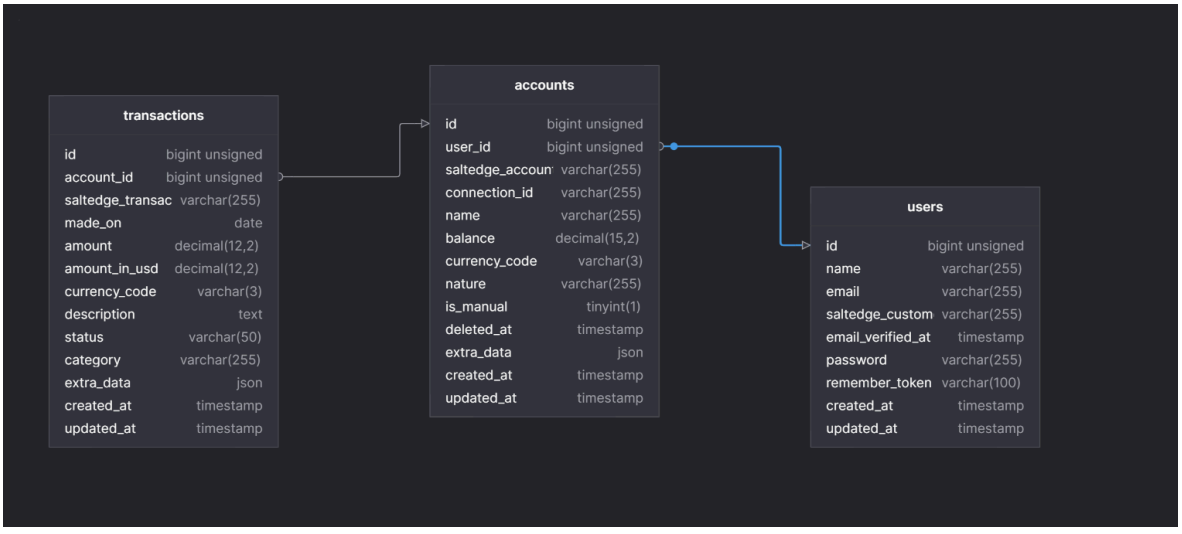

Рисунок 2.3 – Даталогічна модель даних

Першою базовою структурою даталогічної моделі є таблиця користувачів, яка забезпечує зберігання ідентифікаційних даних для доступу до системи. Вона містить унікальний первинний ключ для ідентифікації кожного облікового запису, текстові поля для збереження адреси електронної пошти та зашифрованого

пароля, а також службові мітки часу для фіксації моментів створення й оновлення запису. Для запобігання дублюванню профілів на рівні бази даних для поля електронної пошти встановлено обмеження унікальності.

Другою структурою є таблиця фінансових рахунків, яка логічно підпорядкована таблиці користувачів. Вона містить власний первинний ключ, а також зовнішній ключ, який посилається на ідентифікатор користувача, реалізуючи відношення «один-до-багатьох» (один користувач може володіти множиною рахунків). До складу полів цієї таблиці входять текстові атрибути для назви рахунку, типу фінансового активу (цифровий або готівковий) та міжнародного трилітерного коду валюти. Поточний баланс представлений цілочисельним типом даних для точного збереження грошових сум у копійках. Також передбачено окреме поле для збереження зашифрованого маркера доступу до банківських систем.

Центральною та найбільш динамічною структурою є таблиця фінансових операцій, яка підпорядкована таблиці рахунків зв'язком «один-до-багатьох» (до одного рахунку прив'язано безліч транзакцій). Вона містить первинний ключ операції та зовнішній ключ для зв'язку з конкретним рахунком. Поля суми та дати фіксують точний обсяг витрати або надходження та час проведення транзакції. Відповідно до обраної архітектури, аналітичні параметри інтегровані безпосередньо в структуру цієї таблиці: текстове поле опису призначення платежу, логічне поле напрямку руху коштів та текстове поле категорії операції. Категорія зберігається як фіксований рядок, що відповідає внутрішньому перелічувальному типу даних програмного коду, що дозволяє уникнути надлишкових операцій об'єднання таблиць під час формування аналітичних звітів. Для забезпечення цілісності інформації для всіх зовнішніх ключів налаштовано обмеження каскадного видалення записів.

2.3.5 Фізичне проектування бази даних

У межах даної кваліфікаційної роботи фізичну реалізацію бази даних передбачається здійснити на базі реляційної СУБД MySQL. Цей вибір

зумовлюється її високою продуктивністю, надійністю та повною підтримкою стандартів мови структурованих запитів (SQL).

Фізична модель визначатиме точні типи даних для кожного поля, механізми індексації та правила забезпечення цілісності. Для ідентифікаторів усіх таблиць (полів «id») планується використати тип «BIGINT UNSIGNED» із параметром автоматичного збільшення (AUTO_INCREMENT), що забезпечить унікальність записів та високу швидкість пошуку. Текстові дані обмеженої довжини, такі як імена, адреси електронної пошти та назви рахунків, зберігатимуться у полях типу «VARCHAR(255)», що дозволить економити дисковий простір. Для зберігання криптографічних маркерів доступу (токенів) буде застосовано тип «TEXT», оскільки їхня довжина може перевищувати стандартні ліміти рядкових типів.

Особливу увагу під час проектування приділено фізичному збереженню фінансових показників. Для полів балансу рахунків та сум транзакцій передбачається свідома відмова від використання типів із рухомою комою («FLOAT» чи «DOUBLE») через властиві їм проблеми втрати точності під час математичних операцій. Замість них буде застосовано цілочисельний тип «BIGINT», у якому всі суми зберігатимуться в найменших одиницях валюти (наприклад, у копійках чи центах).

Для оптимізації виконання аналітичних запитів та забезпечення цілісності на рівні бази даних проектується система індексів та обмежень. На поле електронної пошти в таблиці користувачів буде накладено обмеження «UNIQUE», що унеможливить реєстрацію дублікатів. Усі зовнішні ключі (ідентифікатор користувача в рахунках та ідентифікатор рахунку в транзакціях) будуть проіндексовані та пов'язані з батьківськими таблицями правилом «ON DELETE CASCADE». Це означатиме, що фізичне видалення профілю користувача ініціюватиме автоматичне та незворотне видалення всіх його рахунків і відповідних фінансових операцій на рівні ядра бази даних.

Розгортання фізичної моделі планується автоматизувати за допомогою вбудованого механізму програмних міграцій обраного каркаса веб-розробки. Це дозволить відмовитися від ручного виконання SQL-скриптів на користь

об'єктно-орієнтованого опису структури таблиць, забезпечивши суворий контроль версій схеми бази даних протягом усього життєвого циклу розробки програмного продукту [23].

2.3.6 Програмна реалізація бази даних

Програмна реалізація спроектованої фізичної моделі бази даних у розробленій архітектурі виконана за допомогою вбудованих інструментів сучасного об'єктно-орієнтованого фреймворка Laravel. Замість традиційного підходу, що передбачає пряме ручне написання SQL-запитів до СУБД MySQL, у проекті застосовано вбудовану систему об'єктно-реляційного відображення Eloquent ORM. Це дозволяє взаємодіяти з базою даних через об'єкти мови програмування PHP, що значно підвищує безпеку, швидкість розробки та читабельність коду.

Основним інструментом розгортання структури бази даних виступає механізм програмних міграцій Laravel. Він дозволяє описувати схему реляційних таблиць (колонки, типи даних, обмеження) у вигляді спеціалізованих класів, використовуючи компонент Schema Builder. Кожен файл міграції містить два базові методи: перший відповідає за безпосереднє створення таблиці та налаштування первинних і зовнішніх ключів, а другий – за зворотне скасування цих дій. Такий підхід забезпечує повноцінний контроль версій бази даних, уможливлює швидке розгортання проекту на нових серверах і гарантує абсолютну ідентичність структур у середовищах розробки та експлуатації.

Для програмної маніпуляції фінансовою інформацією (збереження, оновлення, видалення та пошук записів) ключову роль відіграє Eloquent ORM, яка базується на архітектурному шаблоні Active Record. Відповідно до цього шаблону, кожна таблиця фізичної моделі («users», «accounts», «transactions») програмно представлена окремим класом-моделлю Eloquent. Ці класи інкапсулюють у собі всю логіку взаємодії з даними. Використання Eloquent ORM автоматично забезпечує високий рівень захисту від SQL-ін'єкцій, оскільки фреймворк за

замовчуванням використовує підготовлені запити (PDO parameter binding) для всіх операцій вибірки та збереження.

Окрім базових операцій, класи-моделі Eloquent містять програмні декларації реляційних зв'язків між сутностями предметної області. Зокрема, між моделями Користувача та Рахунку налаштовано зв'язок «hasMany» (один-до-багатьох), а модель Фінансової операції пов'язана з конкретним рахунком за допомогою зворотного зв'язку «belongsTo». Це дозволяє серверному ядру системи автоматично формувати оптимальні SQL-запити з об'єднанням таблиць під час звернення до властивостей об'єктів, значно скорочуючи обсяг вихідного коду програми.

З міркувань безпеки та модульності архітектури, параметри підключення до фізичного сервера бази даних MySQL (адреса хоста, порт, ім'я бази, користувач та пароль) повністю ізольовані від коду моделей і винесені у захищений конфігураційний файл змінних середовища. Це гарантує неможливість компрометації облікових даних СУБД під час публікації сирцевого коду системи у відкритих репозиторіях.

3 РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ ДЛЯ АНАЛІЗУ ТА ВІЗУАЛІЗАЦІЇ ОСОБИСТИХ ФІНАНСІВ

3.1 Реалізація програмного забезпечення

Веб-застосунок було спроектовано за допомогою архітектурного шаблону «Модель-Вигляд-Контролер» (MVC) [24]. Моделями, на яких будується уся взаємодія веб-застосунку з інформацією, є - користувач, рахунок, транзакція. На рисунку 3.1 можна побачити взаємодію класів між моделями, контролерами та сервісами.

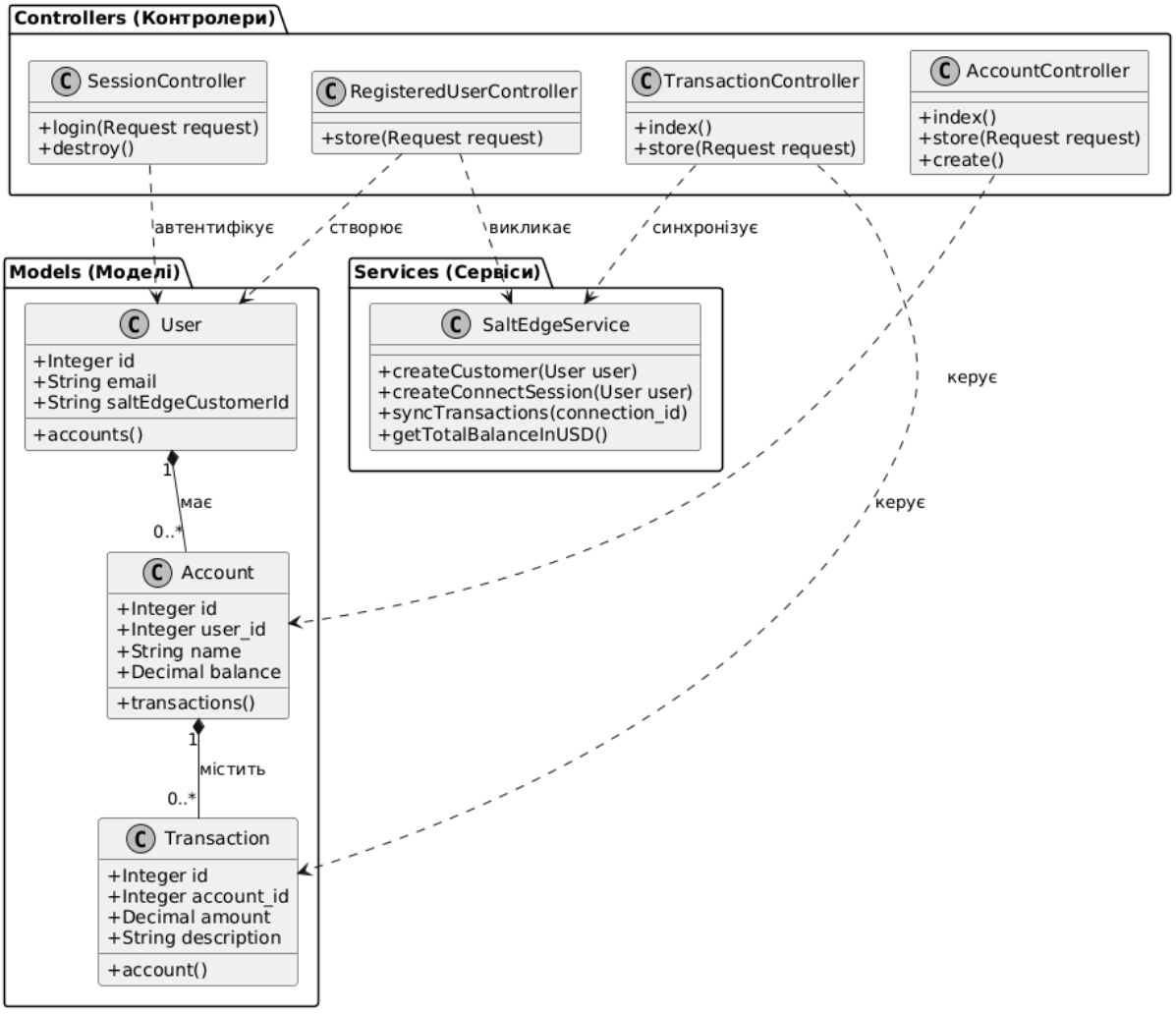

Рисунок 3.1 – Діаграма класів

Коли не залогінений користувач заходить на сайт, єдиний доступний для нього функціонал – це вхід через існуючий акаунт та реєстрація. На відміну від

моделей рахунок та транзакція, за взаємодію з котрими відповідають по одному контроллеру, логіка взаємодії з користувачем поділена на два контроллери - SessionController для взаємодії з зареєстрованим користувачем та RegisteredUserController для взаємодії з незареєстрованим користувачем. Якщо користувач вирішить зареєструватись, то після введення усіх необхідних даних, виконається POST-запит до сервера. Дані з цього запиту отримає маршрутизатор та передає у RegisteredUserController, який за допомогою відповідного методу створить користувача. Процес створення користувача в цьому методі складається з - валідації даних отриманих від користувача (перевірка чи усі необхідні поля заповнені правильно) та створення в БД веб-застосунку та на стороні SaltEdge API (функція для створення користувача в SaltEdge API знаходиться у сервісі SaltEdgeService, там написані усі функції які пов’язані з роботою з прикладним програмним інтерфейсом SaltEdge) нового користувача. SessionController у свою чергу містить метод для перевірки введених користувачем даних, якщо все вірно, то користувач успішно переходить на головну сторінку, якщо ні, то отримує відповідну помилку.

Коли користувача переносить на головну сторінку, що містить історію усіх транзакцій з усіх його рахунків, а також загальний баланс, веб-застосунок робить спочатку GET-запит для виведення історії транзакцій, якщо такі присутні, а потім робить ще один GET-запит вже до Exchange Rate API для отримання актуальних курсів. Саме використовуючи дані отримані від Exchange Rate API, обчислюється загальний баланс користувача з усіх його рахунків. У випадку, якщо користувач лише щойно зареєструвався та немає підключених рахунків, в нього є можливість додати рахунки з онлайн-банків або створити власний готівковий рахунок. Якщо користувач обирає онлайн-банкінг, то під час натискання на кнопку для додавання рахунку, веб-застосунок виконає POST-запит на створення посилання. Отримане у відповідь тимчасове посилання слугує для безпечного перенаправлення користувача на віджет авторизації SaltEdge, де відбувається безпосередній вибір банку та підтвердження надання доступу до балансів. Після завершення авторизації система фіксує унікальний маркер доступу, який зберігається у БД.

Наявність цього токена розблоковує можливість виконання подальших автоматизованих запитів на отримання історії транзакцій та перегляду інших даних своїх рахунків. Клієнт ніколи не передає облікові дані свого онлайн-банкінгу безпосередньо в веб-застосунок, уся необхідна інформація зберігається на стороні SaltEdge.

Програмний код SaltEdgeService представлено в додатку А. У додатках Б, В, Г та Д представлено програмний код основних контролерів веб-застосунку, а саме – RegisteredUserController, SessionController, AccountController, TransactionController.

3.2 Інтерфейс користувача

Розробка інтерфейсу користувача велася з урахуванням принципів інтуїтивності, мінімалізму та високої інформативності. Оскільки основна мета застосунку - надання користувачеві наочної фінансової аналітики, пріоритет було віддано створенню чистого робочого простору, де візуалізація даних переважає над зайвими елементами управління.

На рисунку 3.2 можна побачити сторінку реєстрації користувача.

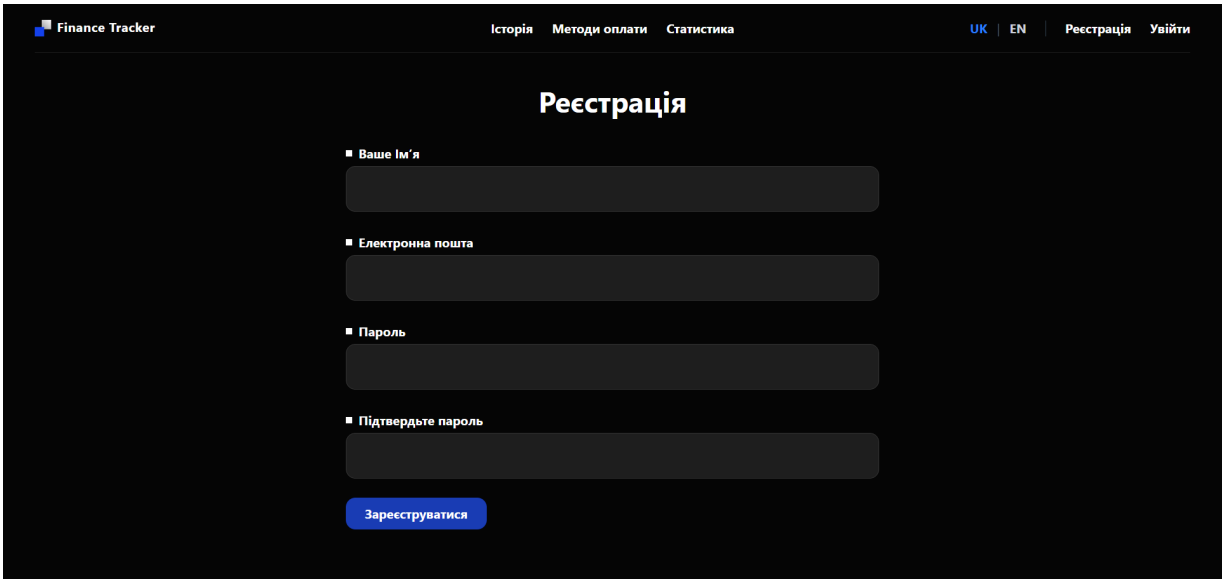

Рисунок 3.2 - Сторінка реєстрації користувача

Над полями введення для реєстрації знаходиться навігаційна панель, яка складається з трьох частин - ліва частина це логотип, натиснувши на який користувача перекине на головну сторінку, центральна частина містить посилання на усі основні сторінки сайту, права частина показує обрану мову та можливість увійти чи вийти з сайту, залежно від статусу користувача.

Коли користувач зареєструється чи залогіниться, він опиниться на головній сторінці (рисунок 3.3). На головній сторінці користувач має доступ до свого загального балансу, усієї історії транзакцій з усіх рахунків та панелі з фільтром, яка дозволяє шукати потрібні транзакції за їх назвою, категорією та датою.

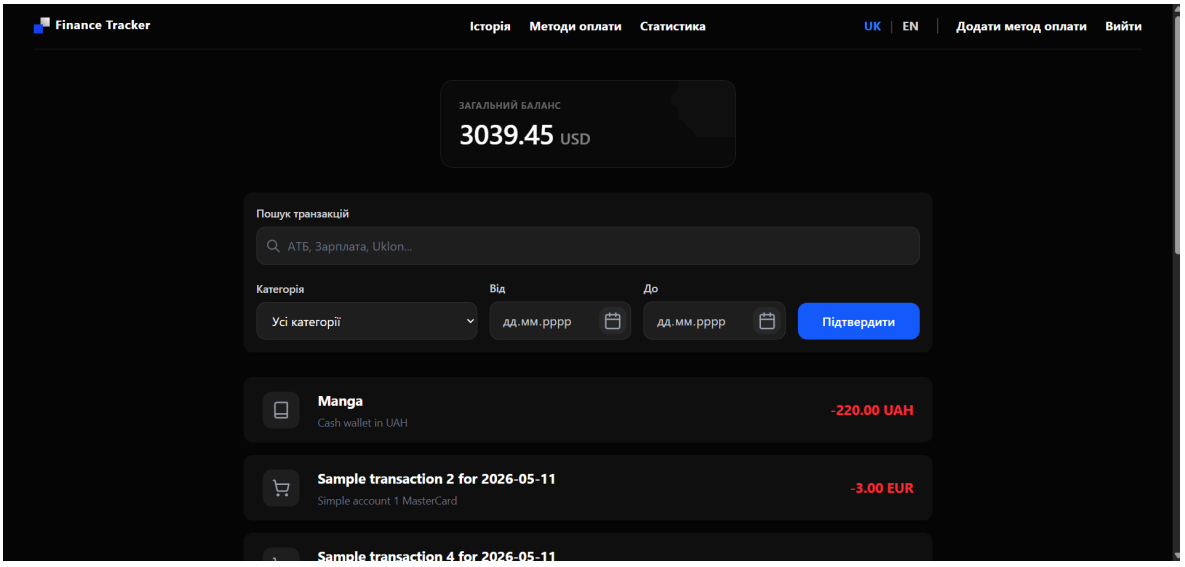

Рисунок 3.3 – Головна сторінка веб-застосунку

Веб-застосунок відображає лише десять транзакцій на сторінку, це зроблено в першу чергу заради продуктивності, адже при кожному переході на головну сторінку робити запит на отримання тисяч транзакцій - це буде довго, не ефективно та можливі ризики помилок через нестачу пам’яті на стороні сервера. Те, як оформлена пагінація та відображення додаткової інформації про транзакцію можна побачити на рисунку 3.4.

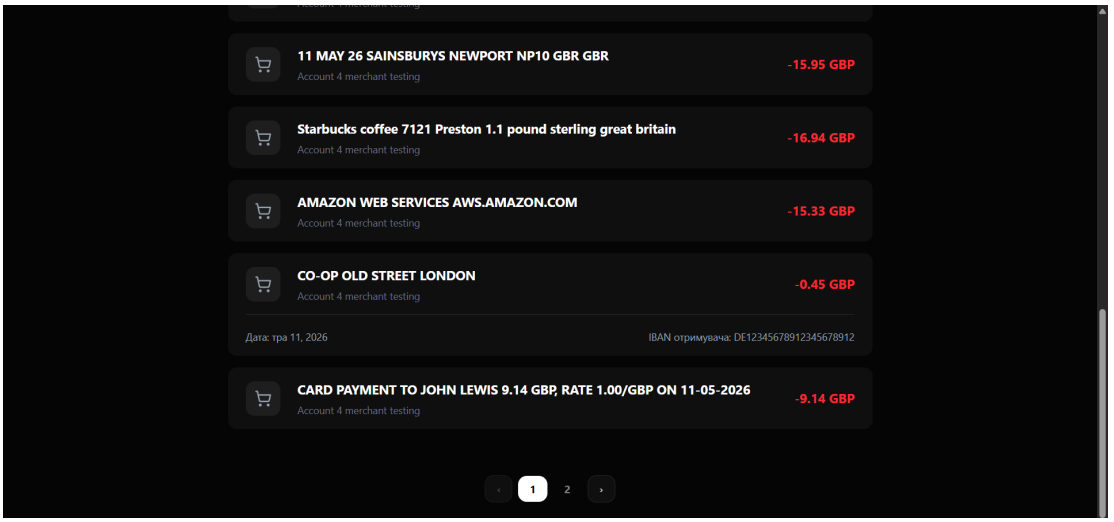

Рисунок 3.4 – Пагінація та додаткові дані про транзакцію

На рисунках 3.5 та 3.6 можна побачити сторінку “Методи оплати”, де відображаються усі рахунки користувача, та детальну інформацію конкретного рахунка. Дані про кожен рахунок виводяться в залежності від типу рахунку за допомогою функції isset(). Функція перевіряє чи містить рахунок ту чи іншу інформацію та, якщо вона є, то виводить її, а якщо ні, то пропускає. У рахунках які є готівковими, тобто створеними користувачем, під фільтром та рядком пошуку знаходиться кнопка для створення транзакції. Кожна картка рахунку містить на собі назву рахунку, ім’я власника рахунку, тип рахунку та баланс даного рахунку, в кінці списку рахунків знаходиться кнопка для створення готівкових рахунків.

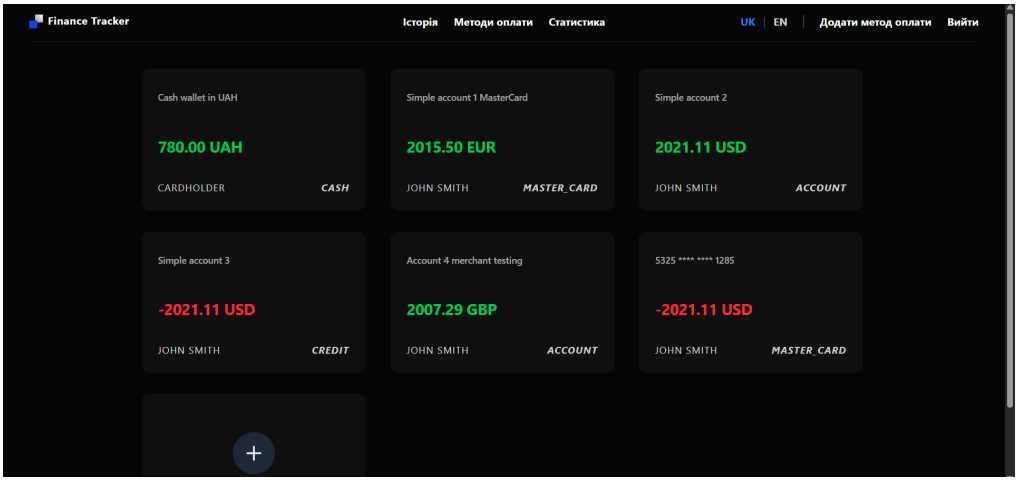

Рисунок 3.5 – Методи оплати

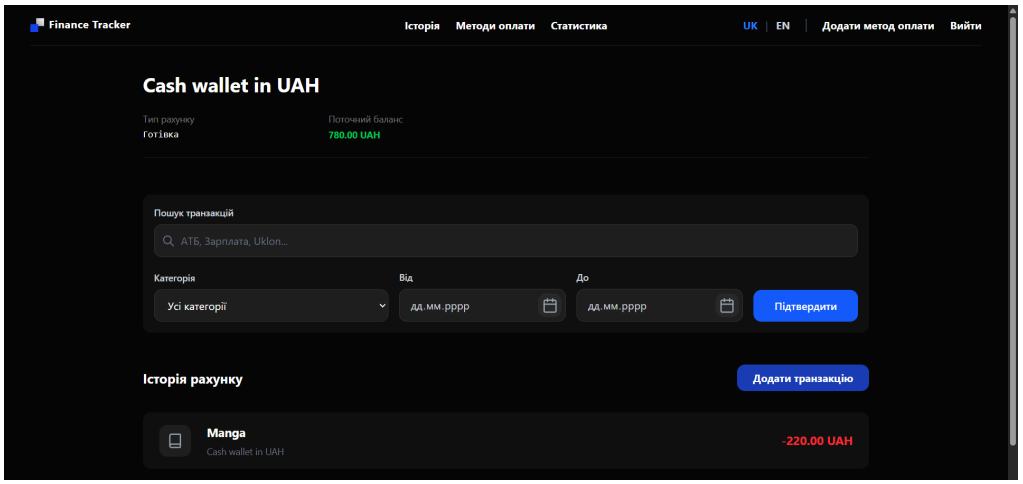

Рисунок 3.6 – Детальна інформація про рахунок

На сторінці “Методи оплати” можна побачити усі рахунки та кнопку для додавання нового рахунку. Навівши курсор на рахунок, можна побачити також кнопку для видалення рахунку (рисунок 3.7).

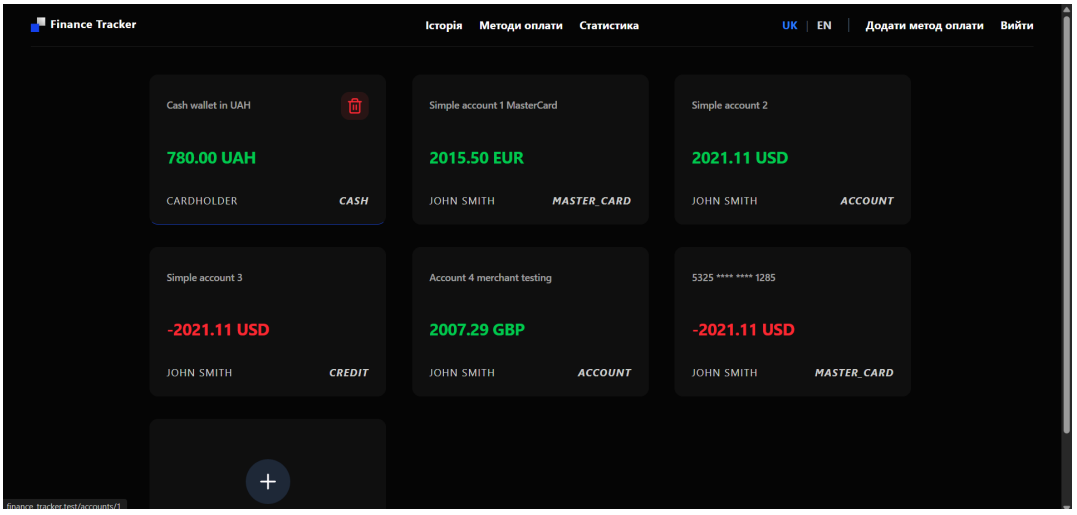

Рисунок 3.7 – Вікно керування рахунками

Натиснувши на рахунок, користувач переходить на сторінку з детальною інформацією про цей рахунок (див. рисунок 3.6). Там йому доступна історія транзакцій цього рахунку та фільтрація історії транзакцій. Якщо це готівковий рахунок, користувач має можливість додавати власноруч створені транзакції (рисунок 3.8).

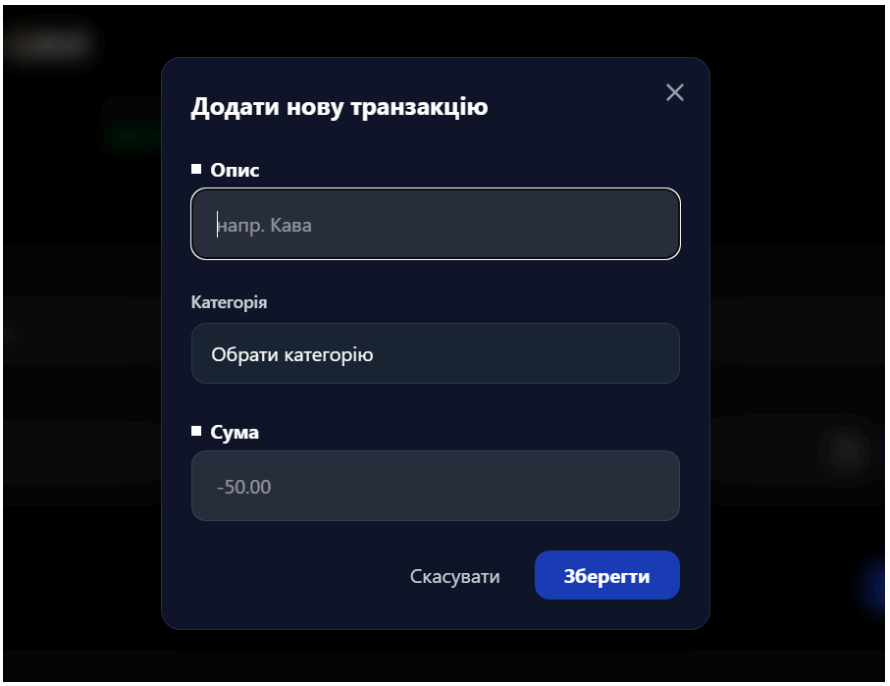

Рисунок 3.8 – Вікно створення транзакції

В залежності від даних, які отримуються від прикладного програмного інтерфейсу SaltEdge, веб-застосунок виводить різну додаткову інформацію про рахунок. На рисунку 3.9 можна побачити як виглядають дані про кредитний рахунок.

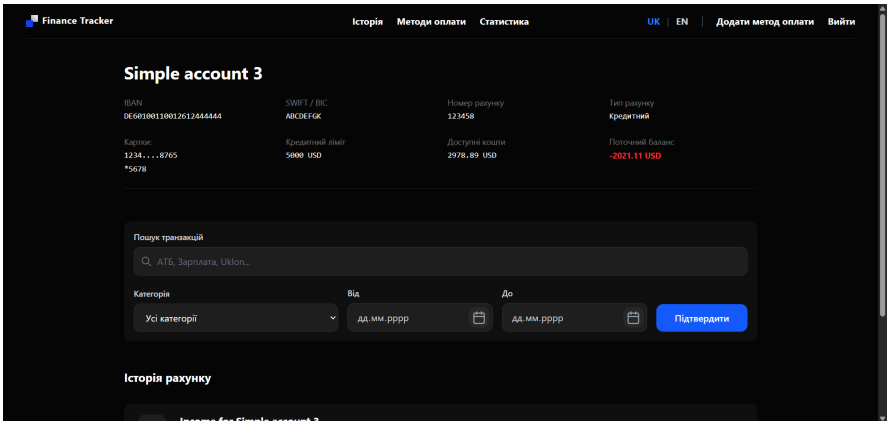

Рисунок 3.9 – Кредитний рахунок

На сторінці “Статистика” користувач має можливість переглянути - динаміку щоденних витрат, розподіл витрат по категоріям та рівень заощаджень (рисунок 3.10).

Рисунок 3.10 – Сторінка статистики користувача

Рисунок 3.11 демонструє діаграму розподілу витрат за категоріями. Даний візуальний інструмент призначений для миттєвої оцінки структури споживання користувача, що дозволяє реалізувати принцип Парето у фінансах та виявити найбільш обтяжливі для бюджету категорії.

Використання кільцевої діаграми ґрунтується на концепціях когнітивної ергономіки, зокрема на теорії мультимедійного навчання (Mayer, R. E.), яка стверджує, що поєднання візуальних образів із текстовими даними значно знижує когнітивне навантаження при обробці інформації.

З точки зору поведінкових фінансів, такий підхід забезпечує «підштовхування» (nudge) користувача до усвідомленого споживання через наочне відображення диспропорцій у бюджеті.

Технічно діаграма формується шляхом багатоступеневої агрегації даних: серверне ядро застосунку за допомогою Eloquent ORM виконує SQL-запит із групуванням витрат за категоріями (GROUP BY), після чого агреговані суми передаються на клієнтську частину у форматі JSON. Кожна категорія отримує на діаграмі відповідний колір. Витрати за категоріями, як і усі діаграми, можна відображати за вибраний користувачем період часу

Візуалізація виконується за допомогою бібліотеки Chart.js, яка через HTML5 Canvas забезпечує швидкий та інтерактивний рендеринг, перетворюючи стандартизовані дані на зрозумілу для користувача графічну модель.

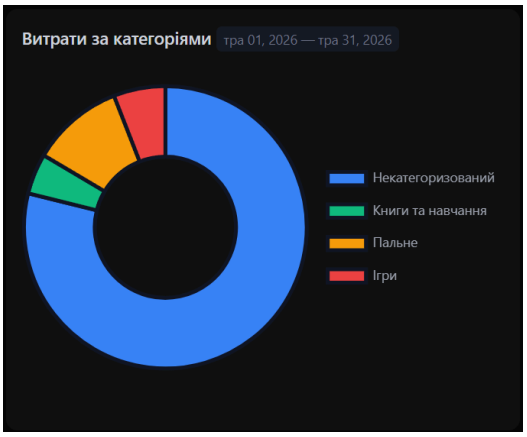

Рисунок 3.11 – Діаграма витрат за категоріями

На рисунку 3.12 зображено лінійну діаграму динаміки щоденних витрат. Ця діаграма виконує функцію інструмента фінансового моніторингу, дозволяючи користувачеві виявляти часові інтервали максимальної активності, аномалії у видатках та сезонні коливання грошових потоків.

З наукової точки зору, цей графік реалізує концепцію фідбеку в режимі реального часу, яка згідно зі звітами OECD/INFE щодо фінансової грамотності, є критично важливою для формування дисципліни накопичення капіталу та виявлення прихованих витрат.

Візуалізація часових рядів сприяє кращому розумінню поведінкових патернів користувача, що відповідає концепції «підштовхування» до більш раціонального фінансового планування.

Програмна реалізація базується на агрегації транзакцій у розрізі часових інтервалів на рівні бази даних MySQL, де для запобігання втраті точності всі фінансові показники зберігаються у цілочисельному форматі (копійках).

Отриманий масив упорядкованих за часом даних передається фронтенд-контролеру, який через клієнтські JavaScript-сценарії будує динамічну лінію тренду, забезпечуючи високу продуктивність без додаткового навантаження на сервер під час взаємодії з графіком.

Крім того, застосування сучасних інструментів візуалізації забезпечує високий рівень інтерактивності розробленого компонента. Завдяки вбудованим механізмам обробки подій, користувач має можливість деталізувати інформацію

безпосередньо на графіку: при наведенні курсору на специфічні точки тренду система миттєво відображає контекстні підказки з точними сумами та датами.

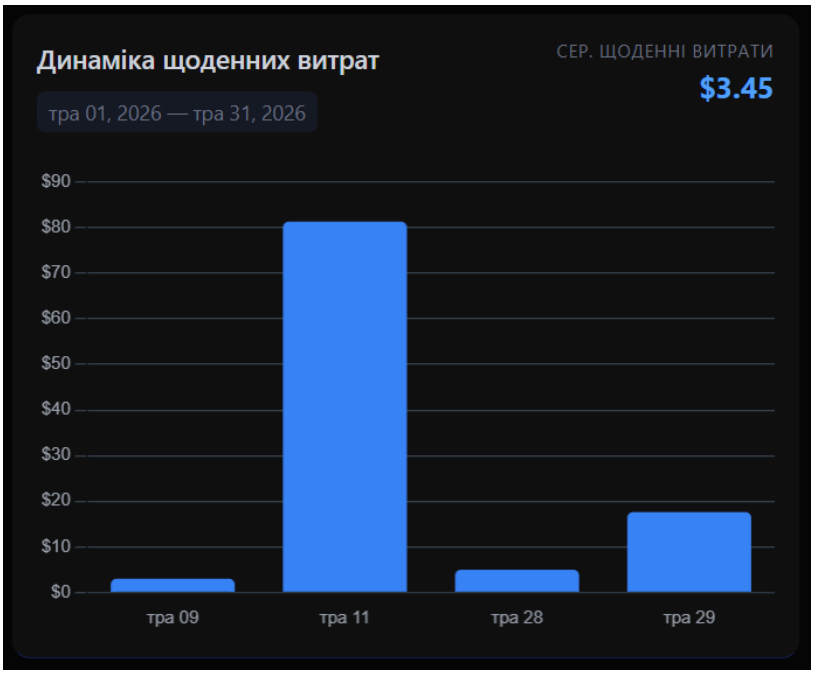

Рисунок 3.12 – Діаграма динаміки щоденних витрат

На рисунку 3.13 зображено аналітичний показник рівня заощаджень. Цей показник є фундаментальним індикатором фінансового здоров’я, що відображає відсоткову частку доходу, яку користувач спрямовує на формування капіталу, а не на споживання.

З позиції фінансової психології, систематичне відстеження рівня заощаджень є ключовим інструментом розвитку фінансової дисципліни та запобігання надмірному споживанню, що повністю узгоджується з дослідженнями OECD/INFE щодо фінансової грамотності, які ідентифікують відсутність контролю за залишком коштів як основну перешкоду для довгострокового планування.

Концептуально цей показник базується на методології «плати спочатку собі», яка ґрунтується на принципах поведінкових фінансів та стимулює користувача до свідомого формування заощаджувальних резервів ще до моменту здійснення поточних витрат.

Технічна реалізація показника у веб-застосунку передбачає динамічний

розрахунок на основі SQL-агрегації даних таблиці transactions. Серверне ядро у реальному часі фільтрує записи за типом операції («надходження» та «витрати»), обчислює чистий грошовий потік як різницю між цими показниками та ділить отриманий результат на загальну суму доходів за визначений період. У результаті користувач отримує візуалізований індикатор у відсотковому виразі, що дозволяє в режимі реального часу оцінювати ефективність управління власними грошовими потоками та коригувати фінансову стратегію.

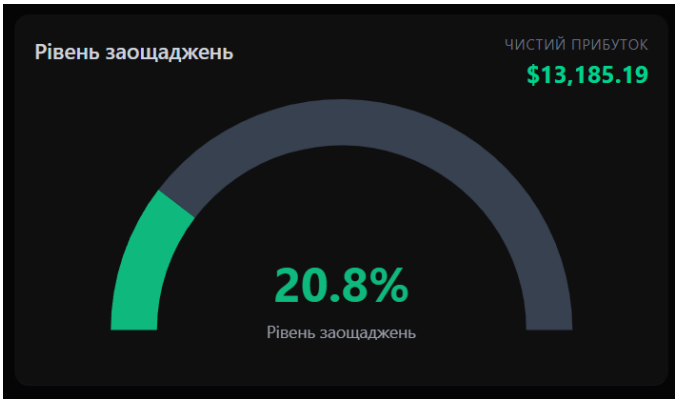

Рисунок 3.13 – Діаграма рівня заощаджень

Цей інструмент перетворює суху статистику на дієвий механізм стратегічного управління фінансами, забезпечуючи користувачеві зворотний зв’язок щодо результативності його особистої заощаджувальної політики та наближення до цільових фінансових орієнтирів.

3.3 Тестування розробленої програми

Тестування є невід'ємною частиною життєвого циклу розробки програмного забезпечення, особливо для систем, що працюють з фінансовими даними. Для забезпечення надійності, цілісності даних та коректності бізнес-логіки у веб-застосунку було застосовано тестування за допомогою вбудованого у фреймворк Laravel інструментарію PHPUnit. Методологія тестування включала два основні рівні: модульні (Unit) тести для перевірки окремих алгоритмів та функціональні (Feature) тести для валідації користувацьких сценаріїв (user flows).

Модульні тести спрямовані на ізольовану перевірку окремих класів або методів, що містять бізнес-логіку. Найбільш критичним об'єктом для тестування є сервісний шар, зокрема клас TransactionsService, що відповідає за нормалізацію категорій транзакцій. Це дозволяє гарантувати, що алгоритм мапінгу зовнішніх міток банку на внутрішні категорії працює безпомилково.

Окрему увагу під час тестування було приділено інтеграційній взаємодії із зовнішнім шлюзом відкритого банкінгу через клас SaltEdgeService. Оскільки здійснення реальних мережевих HTTP-запитів під час тестування є небажаним через залежність від стороннього сервера та ризик спотворення робочих даних, для перевірки методів цього сервісу було застосовано механізм імітації відповідей за допомогою вбудованого фасаду Http. Це дозволило симулювати різноманітні відповіді від API SaltEdge (успішне отримання підключень, повернення масивів транзакцій або виникнення помилок) та верифікувати, що сервіс коректно їх обробляє і безпечно оновлює стан бази даних.

Функціональні тести імітують поведінку реального користувача, взаємодіючи з HTTP-інтерфейсом додатка. Вони перевіряють, як веб-застосунок реагує на запити, чи правильно зберігаються дані в базі та чи здійснюються перенаправлення. Цей тип тестування був використаний для перевірки процесу створення рахунку, включаючи верифікацію того, що атрибут is_manual встановлюється коректно для нових рахунків.

Застосування автоматизованих тестів дозволило досягти високої стабільності. Основні переваги такого підходу - це регресійний захист, будь-які зміни в коді не призводять до випадкового пошкодження існуючої логіки, оскільки автоматизовані тести миттєво сигналізують про помилку, документування коду, тести слугують «живою документацією», що описує очікувану поведінку веб-застосунку для кожного окремого модуля, висока якість даних, тестування бази даних забезпечує впевненість у тому, що фінансова інформація обробляється та зберігається цілісно, без викривлень, що є критично важливим для довіри користувача до застосунку.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було:

1. Досліджено предметну область систем управління особистими фінансами з системно-технологічної точки зору, що дозволило структурувати процеси збору, нормалізації, зберігання та обчислення грошових потоків як цілісний інформаційний бізнес-процес.

2. Проаналізовано існуючі програмні рішення та сервіси для ведення фінансового обліку (автономні додатки, банківські клієнти, універсальні агрегатори). У результаті виявлено проблему фрагментації даних та великих часових витрат на ручне введення транзакцій, що обґрунтувало доцільність створення веб-застосунку на базі технологій відкритого банкінгу.

3. Проведено постановку задачі та сформульовано вимоги до програмного продукту, що дало змогу визначити ключові функціональні та нефункціональні характеристики проектованого рішення.

4. Спроектовано загальну архітектуру веб-застосунку за шаблоном «Модель-Вигляд-Контролер» (MVC) і розроблено її алгоритмічне та інформаційне забезпечення (включаючи концептуальну, даталогічну та фізичну моделі бази даних). Це забезпечило чітке розподілення компонентів архітектури та визначення принципів їхньої взаємодії, що стало основою для подальшої програмної реалізації.

5. Реалізовано веб-застосунок з клієнт-серверною архітектурою на основі попередньо спроектованої структури з використанням сучасних технологій веб-розробки. Серверне ядро написано мовою PHP із використанням об'єктно-орієнтованого фреймворка Laravel та СУБД MySQL. Клієнтська частина включає адаптивний інтерфейс та інтерактивні аналітичні панелі, побудовані за допомогою бібліотеки Chart.js.

6. Здійснено інтеграцію розробленого програмного забезпечення із зовнішніми прикладними програмними інтерфейсами: платформою відкритого

банкінгу SaltEdge для автоматизованого безпечного імпорту банківських виписок та ExchangeRate-API для мультивалютної нормалізації транзакцій.

7. Проведено тестування (модульне та функціональне за допомогою PHPUnit) основних компонентів архітектури з метою оцінки стабільності, цілісності бази даних та коректності функціонування бізнес-логіки мапінгу фінансових категорій.

8. Отримані результати свідчать про досягнення поставленої мети, передбаченої темою роботи. Розроблене програмне рішення «Finance Tracker» реалізує автоматизований збір, нормалізацію, мультивалютний облік та глибокий візуальний аналіз фінансових операцій користувача в єдиному безпечному середовищі.

9. Практичне значення роботи полягає у можливості застосування розробленого веб-застосунку фізичними особами для ефективного управління особистим бюджетом. Веб-застосунок сприяє підвищенню рівня фінансової дисципліни, усуває необхідність рутинного ручного обліку витрат і забезпечує підтримку прийняття рішень щодо оптимізації фінансів на основі наочної візуалізації динаміки бюджету та автоматичного розрахунку рівня заощаджень.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. FinTech and Consumer Financial Outcomes: Evidence from Personal Finance Management Apps. The Review of Financial Studies. 2019. Vol. 32, No. 5. P. 1668–1701.
2. Epstein D. A., Ping A., Fogarty J., Munson S. A. A lived informatics model of personal informatics. Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp). 2015. P. 731–742.
3. Терещенко О. М. Фінансовий менеджмент: підручник. Київ : КНЕУ, 2019. 552 с.
4. OECD/INFE. International Survey of Adult Financial Literacy. OECD Publishing. 2020. 120 p.
5. Пушкар М. С., Кашуба І. М. Інформаційні технології в економіці : підручник. Тернопіль : Карт-бланш, 2018. 350 с.
6. Salt Edge API Reference. Open Banking Gateway. Salt Edge Inc. URL: https://docs.saltedge.com
7. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 533 p.
8. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994. 395 p.
9. PHP: Hypertext Preprocessor. URL: https://www.php.net/manual/en/
10. ExchangeRate-API Documentation. Free & Open Exchange Rate API. URL: https://www.exchangerate-api.com/docs
11. Іванов А. В. Кібербезпека веб-додатків: монографія. Київ: Наукова думка, 2021. 250 с.
12. Бази даних: підручник для студентів вищих навчальних закладів за ред. В. А. Павлиша. Київ: Видавнича група BHV, 2015. 480 с.
13. Few S. Show Me the Numbers: Designing Tables and Graphs to Enlighten. 2nd ed. Analytics Press. 2012. 464 p.

14. Thaler R. H. Nudge, not sludge. Science. 2018. Vol. 361, Issue 6401. P. 431.

15. Pratt P. J., Lastaszewski J. J. A Guide to SQL. 10th ed. Cengage Learning, 2019. 432 p.

16. Tufte E. R. The Visual Display of Quantitative Information. 2nd ed. Graphics Press. 2001. 200 p.

17. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner's Approach. 9th ed. McGraw-Hill Education, 2019. 800 p.

18. UX/UI дизайн: принципи та методології проектування інтерфейсів / О. С. Петренко. Львів: Магнолія, 2020. 220 с.

19. Chart.js Documentation. URL: https://www.chartjs.org/docs/latest/

20. Програмування на PHP: сучасні підходи та фреймворки / В. І. Смирнов [та ін.]. Харків : ХНУРЕ, 2022. 310 с.

21. Laravel Documentation. URL: https://laravel.com/docs

22. Hardt D. The OAuth 2.0 Authorization Framework. RFC 6749. 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749

23. Stauffer M. Laravel: Up & Running: A Framework for Building Modern PHP Apps. 2nd ed. O'Reilly Media, 2019. 540 p.

24. Sommerville I. Software Engineering. 10th ed. Pearson, 2015. 816 p.

25. Шабатько А.В., Турченко І.В. Веб-застосунок для аналізу та візуалізації особистих фінансів. CSPM 2026: Інтелектуальні обчислювальні системи та управління проєктами: збірник тез доповідей міжнародної науково-практичної конференції студентів і молодих вчених (Тернопіль, 21-22 травня 2026 р). Тернопіль: ЗУНУ, 2026. С. 168-171.

26. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

Додаток А

Програмний код сервісу для SaltEdge API

SaltEdgeService.php

```
namespace App\Services;

use App\Models\Account;
use App\Models\Transaction;
use App\Models\User;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Http;

class SaltEdgeService
{
    public static function getTransactions($connection_id)
    {
        $response = Http::saltedge()
            ->get("/transactions?connection_id=$connection_id")
            ->json();
        return $response['data'];
    }

    public static function syncTransactions($connection_id)
    {
        $apiTransactions = self::getTransactions($connection_id);
        foreach ($apiTransactions as $apiTransaction) {
                                        $localAccount = Account::where('saltedge_account_id',
$apiTransaction['account_id'])->first();
            if (!$localAccount) {
                continue;
            }
            $tempTransaction = new Transaction([
                'amount'        => $apiTransaction['amount'],
                'currency_code' => $apiTransaction['currency_code'],
```

```
        ]);
        $usdAmount = $tempTransaction->convertToUSD();

        Transaction::updateOrCreate(
            ['saltedge_transaction_id' => $apiTransaction['id']],
            [
                'account_id'    => $localAccount->id,
                'made_on'       => $apiTransaction['made_on'],
                'amount'        => $apiTransaction['amount'],
                'amount_in_usd' => $usdAmount,
                'currency_code' => $apiTransaction['currency_code'],
                'description'   => $apiTransaction['description'],
                'status'        => $apiTransaction['status'],
                'category'      => $apiTransaction['category'] ?? null,
                'extra_data'    => $apiTransaction['extra'] ?? null
            ]
        );
    }
}

public static function createCustomer(User $user)
{
    $response = Http::saltedge()
        ->post("/customers",
            ['data' => ['identifier' => "$user->email"]])
        ->json();

    if (isset($response['error'])) {
        throw new \Exception($response['error']['message']);
    }
    if ($response['data']['identifier'] == "$user->email") {
        $user->update(['saltedge_customer_id' => $response['data']['customer_id']]);
    }
}

public static function createConnectSession(User $user)
```

```
{
    $response = Http::saltedge()
        ->post("/connections/connect", ['data' =>
            [
                'customer_id' => "$user->saltedge_customer_id",
                'consent'     => ['scopes' => ['accounts', 'transactions']],
                'attempt'     => [
                    'fetch_scopes' => ['accounts', 'transactions'],
                    'return_to'    => 'http://finance_tracker.test'
                ],
            ]
        ])
        ->json();
    return $response['data']['connect_url'];
}

public static function getAccounts(User $user)
{
    $response = Http::saltedge()
        ->get("/accounts?customer_id=$user->saltedge_customer_id")->json();
    return $response['data'];
}

public static function syncAccounts(User $user)
{
    $apiAccounts = self::getAccounts($user);
    foreach ($apiAccounts as $apiAccount) {
        Account::updateOrCreate(
            ['saltedge_account_id' => $apiAccount['id']],
            [
                'user_id'       => $user->id,
                'connection_id' => $apiAccount['connection_id'],
                'name'          => $apiAccount['name'],
                'balance'       => $apiAccount['balance'],
                'currency_code' => $apiAccount['currency_code'],
                'nature'        => $apiAccount['nature'],
```

```
                    'extra_data'    => $apiAccount['extra'] ?? null
                ]
            );
        }
    }

    public static function getConnection(User $user)
    {
                                                                $response =
Http::saltedge()->get("/connections?customer_id=$user->saltedge_customer_id")->json();
        return $response['data'][0]['id'];
    }

    public static function getRates()
    {
        return Cache::remember('currency_rates', 3600, function () {
            $response = Http::get('https://open.er-api.com/v6/latest/USD')->json();
            return $response['rates'];
        });
    }

    public static function getTotalBalanceInUSD()
    {
        $accounts = Account::where('user_id', Auth::user()->id)->get();
        $rates = self::getRates();
        $total = 0;
        foreach ($accounts as $account) {
            $currency = $account['currency_code'];
            $balance = $account['balance'];
            $exchangeRate = $rates[$currency] ?? 1;
            $total += ($balance / $exchangeRate);
        }
        return round($total, 2);
    }
}
```

Додаток Б

Програмний код контролера для незареєстрованих користувачів

RegisteredUserController.php

```
<?php
namespace App\Http\Controllers;

use App\Models\User;
use App\Services\SaltEdgeService;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Log;
use Illuminate\Validation\Rules\File;
use Illuminate\Validation\Rules\Password;

class RegisteredUserController extends Controller
{
    public function create()
    {
        return view('auth/register');
    }

    public function store(Request $request)
    {
        $userAttributes = $request->validate([
            'name'     => 'required',
            'email'    => ['required', 'email', 'unique:users,email'],
            'password' => ['required', 'confirmed', Password::min(8)],
        ]);

        $user = User::create($userAttributes);
        try {
            SaltEdgeService::createCustomer($user);
        } catch (\Exception $e) {
            Log::error('Помилка створення клієнта Salt Edge: ' . $e->getMessage());
```

```
            $user->delete();
            return back()->withInput()->withErrors([
                'email' => 'Сервіс тимчасово недоступний. Спробуйте пізніше.'
            ]);
        }

        Auth::login($user);

        return redirect('/');
    }
}
```

Додаток В

Програмний код контролера для зареєстрованих користувачів

SessionController.php

```
<?php
    namespace App\Http\Controllers;
    use Illuminate\Http\Request;
    use Illuminate\Support\Facades\Auth;
    use Illuminate\Validation\ValidationException;
    class SessionController extends Controller
    {
        public function create()
        {
            return view('auth.login');
        }
        public function store()
        {
            $attributes = request()->validate([
                'email'    => ['required', 'email'],
                'password' => ['required']
            ]);
            if (!Auth::attempt($attributes)) {
                throw ValidationException::withMessages([
                    'email' => 'The provided credentials do not match our records.',
                ]);
            }
            request()->session()->regenerate();
            return redirect('/');
        }
        public function destroy()
        {
            Auth::logout();
            return redirect('/');
        }
    }
```

Додаток Г

Програмний код контролера для рахунків

AccountController.php

```
namespace App\Http\Controllers;

use App\Enums\TransactionCategory;
use App\Jobs\ProcessSyncAccounts;
use App\Models\Account;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\Facades\Http;

class AccountController extends Controller
{
    public function index()
    {
        $user = Auth::user();

        ProcessSyncAccounts::dispatch($user);
        $accounts = $user->accounts()->orderBy('created_at')->get();

        return view('accounts', [
            'accounts' => $accounts
        ]);
    }

    public function show(Account $account)
    {
        $transactions = $account->transactions()
            ->filter(request()->all())
            ->orderBy('made_on', 'desc')
            ->paginate(10)
            ->withQueryString();
```

```
    $groupedCategories = TransactionCategory::grouped();
    $categories = TransactionCategory::grouped();

    return view('account', [
        'account'           => $account,
        'transactions'      => $transactions,
        'groupedCategories' => $groupedCategories,
        'categories'        => $categories
    ]);
}

public function create()
{
    $currencies = Cache::remember('supported_currencies', 86400, function () {
        $response = Http::withoutVerifying()->get('https://open.er-api.com/v6/latest/USD');

        if ($response->successful() && $response->json('result') === 'success') {
            return array_keys($response->json('rates'));
        }
        return ['UAH', 'USD', 'EUR'];
    });

    return view('auth.create-account', [
        'currencies' => $currencies
    ]);
}

public function store(Request $request)
{
    $accountAttributes = $request->validate([
        'name'          => ['required', 'string', 'max:255'],
        'balance'       => ['required', 'numeric'],
        'currency_code' => ['required', 'string'],
    ]);
```

```
        $request->user()->accounts()->create([
            'name'          => $accountAttributes['name'],
            'balance'       => $accountAttributes['balance'],
            'currency_code' => $accountAttributes['currency_code'],
            'nature'        => 'cash',
            'is_manual'     => 1,
        ]);

        return redirect('/accounts');
    }

    public function destroy(Account $account)
    {
        $account->delete();
        return redirect('/accounts');
    }
}
```

Додаток Д

Програмний код контролера для транзакцій

TransactionController.php

```
namespace App\Http\Controllers;

use App\Enums\TransactionCategory;
use App\Jobs\ProcessSyncAccounts;
use App\Jobs\ProcessSyncTransactions;
use App\Models\Account;
use App\Models\Transaction;
use App\Services\SaltEdgeService;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class TransactionController extends Controller
{
    public function index()
    {
        $user = Auth::user();
        $connection_id = SaltEdgeService::getConnection($user);

        ProcessSyncAccounts::dispatch($user);
        ProcessSyncTransactions::dispatch($connection_id);

        $totalBalance = SaltEdgeService::getTotalBalanceInUSD();
        $transactions = Transaction::with('account')
            ->whereHas('account', function ($query) use ($user) {
                $query->where('user_id', $user->id);
            })
            ->filter(request()->all())
            ->orderBy('made_on', 'desc')
            ->paginate(10)
            ->withQueryString();
```

```
        $categories = TransactionCategory::grouped();

        return view('index', [
            'transactions' => $transactions,
            'totalBalance' => $totalBalance,
            'categories'   => $categories
        ]);
    }

    public function store(Request $request, Account $account)
    {
        $transactionsAttributes = $request->validate([
            'description' => ['required', 'string', 'max:255'],
            'category'    => ['required', 'string', 'max:255'],
            'amount'      => ['required', 'numeric']
        ]);

        $tempTransaction = new Transaction([
            'amount'        => $transactionsAttributes['amount'],
            'currency_code' => $account->currency_code,
        ]);

        $usdAmount = $tempTransaction->convertToUSD();

        $account->transactions()->create([
            'description'   => $transactionsAttributes['description'],
            'category'      => $transactionsAttributes['category'],
            'amount'        => $transactionsAttributes['amount'],
            'amount_in_usd' => $usdAmount,
            'currency_code' => $account->currency_code,
            'made_on'       => now()->toDateString(),
        ])
        $account->increment('balance', $transactionsAttributes['amount']);
        return redirect("/accounts/{$account->id}");
    }
}
```

Додаток Е

Копія опублікованих результатів