БАКАЛАВРСЬКА РОБОТА

БР.КІ-11.00.00.000 ПЗ

Група КІ-22-1

Кулинич Сергій

2026

Івано-Франківський національний технічний університет нафти і газу
Факультет інформаційних технологій
Кафедра комп'ютерних систем і мереж

Кулинич Сергій Миколайович
(прізвище, ім'я, по батькові)

УДК 004.62

БАКАЛАВРСЬКА РОБОТА

Розробка web-додатку для планування та обліку особистих фінансів з інтеграцією банківських даних засобами JS, Fetch API, Python, mySql

Комп'ютерна інженерія
(назва освітньої програми)

123 - Комп'ютерна інженерія
(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня *С.М. Кулинич*
(підпис, ініціали та прізвище здобувача)

Науковий керівник *Гарасимів Т.Г. асист.*
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту

Завідувач кафедри *С. І. Мельничук*
(підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

(повне найменування вищого навчального закладу)

Факультет *інформаційних технологій*

Кафедра *комп'ютерних систем і мереж*

Освітній ступінь *бакалавр*

Спеціальність *123 – Комп'ютерна інженерія*

(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри КСМ
(С.І. Мельничук)
"*21*" *квітня 2026* року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Кулиничу Сергію Миколайовичу

(прізвище, ім'я, по батькові)

1. Тема роботи *"Розробка web-додатку для планування та обліку особистих фінансів з інтеграцією банківських даних засобами JS, Fetch API, Python, mySql"*

керівник проекту (роботи) *Гарасимів Т. Г. асист.*

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від "*21" квітня 2026 року №180/7*

2. Строк подання студентом роботи *11 червня 2026 р.*

3. Вихідні дані до роботи *Матеріали і результати отримані під час проходження переддипломної практики, методичні вказівки, технічна література.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) *1. Аналіз предметної області та постановка завдання. 2. Графове проєктування системи та транзакційного контуру. 3. Програмна реалізація та верифікація системи.*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів роботи

Розділ	Консультант	Підпис, дата

7. Дата видачі завдання *02 лютого 2026 р.*

№ з/п	Назва етапів дипломного проекту (роботи)	Термін виконання етапів проекту (роботи)	Примітка
1	*Аналіз предметної області та постановка завдання*	*Лютий, 2026*	
2	*Графове проєктування системи та транзакційного контуру*	*Березень, 2026*	
3	*Програмна реалізація та верифікація системи*	*Травень, 2026*	
4	*Оформлення пояснювальної записки*	*Червень, 2026*	
5			

Студент _____ *Кулинич С.М.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Гарасимів Т.Г.*
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Актуальність роботи зумовлена необхідністю автоматизації обліку особистих фінансів через стрімке зростання обсягів безготівкових операцій та потребу користувачів у консолідації даних з різних банківських джерел. Поставлено задачу розробити інформаційну систему, яка забезпечує автоматизований збір, нормалізацію та структурування фінансових транзакцій з API банків та файлів банківських виписок.

Шляхом вирішення задачі обрано проєктування архітектури на основі асинхронного фреймворку FastAPI, реляційної бази даних MySQL та впровадження алгоритмів ідемпотентної обробки даних для виключення дублювання записів. Реалізовано механізми взаємодії з банківськими сервісами, розроблено алгоритми парсингу табличних даних та структуру бази даних для зберігання фінансової інформації користувачів.

У результаті створено програмний комплекс, що автоматично інтегрує транзакції, класифікує їх за MCC-кодами та синхронізує з плановими показниками бюджету користувача. Висновки підтверджують ефективність обраних методів обробки даних у середовищі MySQL, що забезпечує цілісність фінансової звітності та мінімізує витрати часу користувача на ручне введення інформації.

Ключові слова: інформаційна система, фінансовий облік, fastapi, mysql, monobank api, обробка даних, парсинг виписок, автоматизація, транзакції, ідемпотентність, mcc-коди, категоризація.

SUMMARY

The relevance of the work is driven by the need to automate personal finance accounting due to the rapid growth of cashless transactions and the demand for consolidating data from various banking sources. The objective of the project is to develop an information system that provides automated collection, normalization, and structuring of financial transactions from bank APIs and statement files.

To achieve this objective, an architecture based on the asynchronous FastAPI framework and the MySQL relational database management system was designed. Algorithms for idempotent data processing were implemented to prevent the duplication of transaction records. Mechanisms for interaction with banking services, parsing algorithms for tabular data, and a database schema for storing user financial information were developed.

As a result, a software system was created that automatically integrates transactions, classifies them using MCC codes, and synchronizes them with the user's budget plans. The conclusions confirm the efficiency of the chosen data processing methods within the MySQL environment, ensuring the integrity of financial reporting and minimizing the time required for manual data entry.

Keywords: information system, financial accounting, fastapi, mysql, monobank api, data processing, statement parsing, automation, transactions, idempotency, mcc codes, categorization.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасних економічних умовах питання особистого фінансового планування набуває стратегічного значення як на індивідуальному, так і на макрорівні. Глобальна діджиталізація фінансового сектору призвела до появи великої кількості банківських продуктів, електронних гаманців та інвестиційних інструментів, що потребують комплексного обліку. Згідно зі звітами міжнародних аналітичних центрів, відсутність інструментів систематизації фінансових даних є однією з головних причин зниження фінансової грамотності населення та прийняття неефективних споживчих рішень. Ручний облік доходів та витрат у табличних редакторах втрачає свою актуальність через високу ймовірність помилок, низьку швидкість обробки даних та складність візуалізації фінансових тенденцій.

Для автоматизації фінансового обліку виникає інженерна задача розробки цифрових рішень, здатних агрегувати дані з різних джерел. В Україні розвиток інституту Open Banking, зокрема запровадження Monobank Open API, відкрило можливості для автоматизованого обміну даними між банками та клієнтськими застосунками. Попри це, існуючі програмні продукти часто є або надто спрощеними, або орієнтованими на закриті корпоративні екосистеми, що обмежує користувачів у виборі інструментарію. Сучасний стан цього інженерного завдання вимагає переходу до систем типу Personal Finance Management (PFM), які забезпечують синхронізацію даних, безпеку транзакцій та автоматичну класифікацію фінансових операцій.

Аналіз сучасного стану проєктування фінансових систем свідчить про доцільність впровадження гібридної архітектури, що поєднує високу швидкість обробки даних на стороні сервера з безпечним шифруванням чутливої інформації. Дослідження у галузі фінансової інформатики вказують на необхідність використання сучасних методів безсесійної автентифікації та механізмів запобігання дублюванню даних при обробці банківських виписок. Проєктування таких систем із використанням високопродуктивних фреймворків, таких як

FastAPI, та інструментів асинхронної обробки забезпечує масштабованість, необхідну для стабільної роботи з великими масивами транзакційних даних.

Об’єкт дослідження – процес автоматизованого збору, обробки, структурування та аналізу фінансових даних користувача в умовах інтеграції з банківськими сервісами.

Предмет дослідження – методи, алгоритми, архітектурні рішення та програмні компоненти інформаційної системи «Finance Planner», побудовані на базі клієнт-серверної моделі з використанням інтегрованих API та модулів парсингу даних.

Мета роботи – проєктування, програмна реалізація, тестування та розгортання автономного вебдодатка «Finance Planner» для централізованого обліку особистих фінансів, який повністю адаптований під специфіку українського фінтех-ринку.

Для досягнення поставленої мети необхідно вирішити такі завдання:

1. Проаналізувати стан ринку програмного забезпечення для особистого фінансового менеджменту та обґрунтувати вибір технологічного стеку.

2. Спроектувати структуру бази даних, здатну підтримувати багатокористувацький режим та цілісність фінансових операцій.

3. Розробити модуль безпеки, що забезпечує хешування паролів та симетричне шифрування банківських токенів доступу.

4. Реалізувати програмний інтерфейс для інтеграції з Monobank Open API з механізмами захисту від дублювання.

5. Розробити алгоритми парсингу та нормалізації неструктурованих файлових виписок (CSV, XLSX) для уніфікації даних з різних банківських установ.

6. Впровадити систему автоматичної категоризації транзакцій на основі MCC-кодів та аналізу опису операцій.

7. Розробити клієнтську частину веб-додатка з функціоналом візуалізації бюджетів та фінансових цілей.

Методи дослідження:

- Методи системного аналізу – для формування функціональних вимог до

системи та проєктування її загальної архітектури.

- Методи об'єктно-орієнтованого проєктування – для побудови ієрархії моделей даних та взаємодії між сервісними компонентами.
- Методи криптографічного захисту (стандарт Fernet) – для забезпечення конфіденційності токенів доступу до банківських API.
- Алгоритмічні методи парсингу – для обробки та нормалізації даних з різнорідних банківських файлів.

Практичне значення отриманих результатів полягає у створенні готового програмного продукту «Finance Planner», який дозволяє користувачам автоматизувати облік витрат, мінімізувати помилки при внесенні даних та отримувати деталізовану аналітику фінансового стану. Отримані архітектурні напрацювання щодо інтеграції з банківськими API та парсингу даних можуть бути використані при розробці аналогічних інтелектуальних систем підтримки прийняття фінансових рішень.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної області автоматизації обліку особистих фінансів

Сучасний етап розвитку глобального фінансового сектору характеризується інтенсивною діджиталізацією, поступовим витісненням готівкових розрахунків та стрімкою диверсифікацією користувацьких активів [17]. На відміну від попередніх десятиліть, коли управління капіталом індивіда обмежувалося фіксацією готівкових коштів у межах єдиного фізичного гаманця, сучасний споживач фінансових послуг взаємодіє з децентралізованою екосистемою. Вона охоплює дебетові та кредитні картки кількох банківських установ, накопичувальні й депозитні рахунки, цифрові гаманці платіжних систем, автоматизовані сервіси регулярних підписок та транскордонні транзакційні платформи .

Основна аналітична проблема такої децентралізації полягає в інформаційній ізольованості даних: кожна банківська установа надає інструменти моніторингу витрат виключно всередині власного програмного продукту. Як наслідок, користувач позбавлений можливості бачити консолідовану картину свого фінансового стану, оперативно оцінювати сумарні витрати за конкретними категоріями в усіх банках одночасно, а також контролювати кумулятивні ліміти та прогрес досягнення фінансових цілей. Це зумовлює об’єктивну інженерну потребу в розробці та впровадженні інтегрованих систем класу Personal Finance Management, які виконують функції централізованого хабу для збору, обробки, нормалізації та візуалізації гетерогенних фінансових потоків [17].

Інформаційний процес функціонування систем управління особистими фінансами підпорядковується вимогам чинного законодавства та галузевим стандартам безпеки. На території України ключовим нормативно-правовим документом, що регулює цей інфраструктурний простір, є Закон України «Про платіжні послуги» [1], який нормативно закріплює концепцію Відкритого банкінгу[11]. Даний закон гармонізований із європейською директовою Payment Services Directive 2 [18], що зобов'язує банківські установи надавати авторизованим

стороннім постачальникам послуг або індивідуальним сервісам користувачів безпечний доступ до інформації про рахунки через відкриті програмні інтерфейси. Оскільки фінансова інформація має статус банківської таємниці та містить персональні дані, архітектура сучасних фінансових вебдодатків повинна суворо відповідати Закону України «Про захист персональних даних» [2], а також міжнародному регламенту General Data Protection Regulation [19] в частині криптографічного захисту токенів доступу та забезпечення права користувача на видалення інформації.

З погляду системного аналізу, досліджуваний інформаційний процес у межах автоматизованого обліку особистих фінансів складається з кількох послідовних та взаємопов'язаних етапів обробки даних.

Першим базовим етапом є контур агрегації та нормалізації вхідних потоків. Він забезпечує імпорт відомостей із використанням двох ключових каналів: автоматизованого синхронного обміну через програмні інтерфейси банків у реальному часі та файлового імпорту структурованих виписок користувача. Автоматизація дозволяє повністю нівелювати вплив людського фактора, коли суб'єкт забуває або лінується зафіксувати операцію в ручному режимі. Проте підтримка файлового імпорту залишається критично важливою через відсутність прямого програмного доступу до деяких фінансових установ. Головним технологічним викликом на цьому етапі стає нормалізація даних. Різні банки застосовують власні стандарти запису: відмінності у відображенні знаку суми, використання окремих колонок для типів операцій, розбіжності у форматах дати та структурі текстових описів вимагають приведення різнорідних даних до єдиного реляційного стандарту всередині бази даних [12,13].

Другим етапом є класифікація та категоризація транзакцій, оскільки простий хронологічний список фінансових операцій має низьку аналітичну цінність для планування. Для автоматичного групування витрат використовуються два основні інструменти. Перший базується на ідентифікації кодів категорій торговця, які присвоюються банком-еквайєром кожній торговій точці для визначення типу бізнесу. Використання цих кодів дозволяє з високою точністю автоматично

визначати категорію витрат одразу після синхронізації. Другий інструмент передбачає семантичний аналіз текстових описів для транзакцій, де відсутній числовий код. У такому випадку категорія визначається за ключовими словами в описі платежу або назві продавця. Разом з тим, алгоритми автоматичної класифікації потребують наявності інтерфейсу для ручного редагування та створення персоналізованих правил перепризначення категорій.

Третій етап охоплює процеси проактивного бюджетування та планування. Цей контур забезпечує перехід від пасивного моніторингу минулих подій до активного управління капіталом. Інструменти бюджетування дозволяють встановлювати граничні суми витрат на розрахунковий період для конкретних категорій. Програмна реалізація цього процесу вимагає постійного динамічного перерахунку фінансових операцій у реальному часі для демонстрації стеження за лімітами. Це допомагає контролювати споживацькі звички та уникати дефіциту коштів.

Четвертим етапом є моделювання та відстеження довгострокових фінансових цілей для накопичення капіталу. Оскільки користувачі застосовують різні стратегії заощадження, функціонал роботи з цілями має підтримувати три основні режими розрахунку. Ручний режим передбачає самостійну фіксацію внесків. Режим на основі транзакцій аналізує потік операцій та автоматично зараховує внески при виконанні заданих тригерних умов. Режим прив’язки до балансу автоматично прирівнює суму накопиченого до поточного залишку на конкретному банківському рахунку.

Важливим технологічним аспектом проєктування фінансових інформаційних систем є забезпечення конфіденційності. Оскільки фінансова інформація є критично чутливою, використання публічних хмарних сервісів часто викликає занепокоєння щодо збереження таємниці. У зв'язку з цим актуальним є перехід до архітектурних рішень приватного хостингу, де вся база даних перебуває під повним контролем користувача [16]. Безпека взаємодії клієнтської та серверної частин у таких системах забезпечується через механізми безсесійної автентифікації на основі токенів, а збереження банківських токенів доступу у базі даних вимагає

обов'язкового застосування методів симетричного криптографічного шифрування. Це гарантує, що навіть у разі несанкціонованого доступу до сховища даних, зловмисники не зможуть скористатися ключами для доступу до банківських API.

Таким чином, проведений аналіз предметної області, нормативно-правового забезпечення та структури інформаційних потоків показує, що сучасна система обліку особистих фінансів є складним програмним комплексом. Вона здійснює повний цикл обробки даних – від автоматизованого збору інформації та її нормалізації до розумної класифікації, динамічного контролю бюджетних лімітів і безпечного збереження конфіденційних фінансових відомостей.

1.2 Порівняльний аналіз існуючих рішень та аналогів

Для вибору оптимальних архітектурних рішень необхідно проаналізувати наявні програмні продукти, які використовуються для обліку особистих фінансів. Сьогодні на ринку домінують три різні підходи: універсальні електронні таблиці, мобільні додатки конкретних банків та спеціалізовані комерційні сервіси. Щоб зрозуміти їхні переваги та обмеження, розглянемо три реальні продукти, які найчастіше обирають користувачі в Україні: Microsoft Excel, Monobank та закордонний PFM-сервіс Wallet від BudgetBakers [14,15].

Microsoft Excel є класичним інструментом завдяки своїй доступності та гнучкості [13]. Його головна перевага – користувач може самостійно створити будь-яку структуру бази даних, налаштувати власні категорії витрат, написати потрібні формули та побудувати кастомні графіки. Така свобода дій дозволяє масштабувати систему від простого щоденника витрат до складної моделі корпоративного фінансового обліку. Проте Excel не має автоматизації. Дані доводиться вносити вручну або постійно копіювати з банківських виписок, що займає багато часу та призводить до помилок через неуважність. Спроби автоматизувати процес за допомогою скриптів чи зовнішніх запитів вимагають значних навичок програмування, а сам інтерфейс таблиць незручний для щоденного використання з мобільних пристроїв. Через це користувачі часто

закидають ведення бюджету вже за кілька тижнів після старту, не витримуючи рутинного навантаження.

Застосунок Monobank є прикладом протилежного підходу – максимальної автоматизації [15]. Тут усі операції за карткою фіксуються миттєво в момент оплати, а система сама визначає категорію витрат на основі кодів торгових точок (MCC). Користувачеві взагалі не потрібно витрачати час на ручне введення. Це робить поріг входження мінімальним, адже аналітика формується у фоновому режимі без жодних зусиль з боку клієнта. Головний недолік Monobank – це його закритість. Додаток відображає лише ті гроші, які проходять через цей конкретний банк. Якщо у користувача є картки інших банків, готівка чи рахунки в іноземних сервісах, додаток Monobank не зможе побудувати загальну фінансову картину. Крім того, інструменти планування бюджету та створення цілей тут досить обмежені й не підлягають гнучкому налаштуванню під індивідуальні правила. У результаті банківський додаток залишається зручним трекером щоденних витрат, але не повноцінною системою управління капіталом.

Кросплатформний сервіс Wallet від BudgetBakers намагається об'єднати переваги обох попередніх варіантів [14]. Він пропонує зручний інтерфейс, синхронізацію між різними пристроями та детальні аналітичні звіти. Завдяки хмарній архітектурі користувач може вносити дані як з телефона на ходу, так і детально аналізувати графіки з великого екрана комп'ютера. Проте Wallet розроблявся під ринки США та ЄС, тому в Україні його автоматична синхронізація з банками працює нестабільно через використання іноземних сервісів-посередників. Будь-які зміни в API українських банків призводять до того, що синхронізація зупиняється на тривалий час. Функція імпорту CSV-файлів також часто видає помилки через невідповідність українських форматів дат та числових розділювачів копійок. Це змушує користувачів постійно «перепідключати» рахунки або виправляти помилки вручну, що нівелює ідею автоматизації. До того ж, повний функціонал Wallet платний, а необхідність передавати свої банківські токени на сторонні сервери створює серйозні ризики для безпеки даних, на які готові піти далеко не всі користувачі в питанні захисту власних заощаджень.

Для наочності проведеного аналізу основні технічні та функціональні характеристики цих продуктів зведено у загальну таблицю (табл. 1.1).

Таблиця 1.1. Порівняльна характеристика існуючих програмних засобів фінансового обліку

Критерій порівняння	Microsoft Excel	Мобільний додаток Monobank	Wallet від BudgetBakers
Спосіб отримання даних	Повністю ручне введення або імпорт через скрипти.	Автоматично (тільки для операцій всередині банку).	Автоматично через посередників або ручний імпорт файлів.
Консолідація кількох джерел	Можлива, але потребує ручного створення таблиць.	Відсутня (обмежено лише рахунками Monobank).	Часткова (підключення сторонніх банків працює нестабільно).
Категоризація витрат	Повністю ручна (або через складні формули).	Автоматична (жорстко зафіксована за MCC-кодами банку).	Автоматична (потребує постійного ручного коригування).
Гнучкість налаштувань	Максимальна (користувач сам будує всю логіку).	Низька (змінити логіку інтерфейсу чи звітів неможливо).	Середня (обмежена шаблонами та налаштуваннями сервісу).
Безпека та конфіденційність	Висока при локальному зберіганні файлу.	Висока (захищено банківськими стандартами безпеки).	Низька (фінансові токени та дані зберігаються на сторонніх серверах).
Адаптація до ринку України	Універсальна (залежить від налаштувань користувача).	Повна (продукт створено в Україні та для України).	Низька (орієнтовано на закордонні стандарти та формати даних).
Фінансова доступність	Безкоштовно (або вартість офісного пакету).	Повністю безкоштовно для клієнтів банку.	Платна підписка для доступу до автоматизації та звітів.

Аналіз показує, що жоден із продуктів не є ідеальним. Електронні таблиці забирають багато часу, банківські додатки замикають користувача в межах однієї картки, а закордонні сервіси є платними та створюють ризики для збереження персональних даних. Це обґрунтовує необхідність створення автономної веб-орієнтованої системи, яка поєднає пряму безкоштовну інтеграцію з локальними банківськими API, автоматичний парсинг українських виписок та локальне шифрування даних на власному сервері користувача.

1.3. Постановка задачі

На основі детального аналізу предметної області та критичного огляду існуючих аналогів (як закордонних комерційних платформ, так і стандартних інструментів автоматизації), було виявлено відсутність на українському ринку гнучкого, безкоштовного та безпечного рішення для консолідації особистих фінансів. Більшість існуючих програм або прив'язують користувача до екосистеми одного банку, або змушують передавати чутливі фінансові дані іноземним компаніям без гарантій їхньої конфіденційності.

З огляду на це, метою бакалаврської роботи є проєктування, програмна реалізація, тестування та розгортання автономного вебдодатка «Finance Planner» для централізованого обліку особистих фінансів, який повністю адаптований під специфіку українського фінтех-ринку та забезпечує максимальний рівень приватності користувача за допомогою архітектури self-hosted [16].

Для досягнення поставленої мети у роботі необхідно дотримуватись таких вимог:

1. Функціональні вимоги до розроблюваної системи

Програмний продукт має забезпечувати виконання таких ключових бізнес-функцій:

– Управління профілем та безпека: реалізація захищеної реєстрації та автентифікації користувачів з ізоляцією даних на рівні ідентифікатора акаунта, щоб унеможливити доступ до чужої фінансової інформації.

– Консолідація рахунків: забезпечення можливості створення та керування рахунками у різних валютах як вручну, так і в автоматичному режимі під час синхронізації з банками.

– Автоматизований імпорт через Open API: розробка підсистеми асинхронної взаємодії з Monobank Open API для завантаження історії транзакцій та актуальних залишків за рахунками за визначений період без створення дублікатів записів [3].

– Гнучкий імпорт файлів виписок: створення універсального парсера документів у форматах CSV, TXT та XLSX для обробки даних з інших фінансових установ (зокрема тих, що не надають відкритого API для фізичних осіб)[12,13].

– Розумна категоризація: реалізація механізму автоматичного присвоєння категорій витрат на основі чотиризначних кодів торгових точок (MCC) та семантичного аналізу текстових полів призначення платежу, з можливістю подальшого ручного коригування користувачем.

– Моніторинг та аналітика: розробка інтерактивного дашборду для відображення ключових фінансових метрик (загальний дохід, витрати, чисте сальдо) та візуалізації структури витрат за допомогою динамічних графіків.

– Бюджетування: реалізація системи контролю помісячних лімітів витрат у розрізі окремих категорій із відстеженням прогресу в реальному часі.

– Управління фінансовими цілями: проєктування та реалізація підсистеми накопичень із підтримкою трьох режимів обчислення залишків та можливістю створення кастомних правил автоматичного відбору транзакцій.

2. Технічні та архітектурні вимоги до системи

Програмне забезпечення має відповідати таким інженерним стандартам та обмеженням:

– Архітектура рішення: система має розроблятися у межах єдиного монолітного репозиторію, де бекенд функціонує як REST API, а фронтенд – як легка статична сторінка в браузері (патерн Single Page Application) без додаткових етапів компіляції на стороні клієнта.

– Стек технологій backend: використання мови програмування Python 3.10+, асинхронного вебфреймворка FastAPI [4], сучасного ORM SQLAlchemy 2 [5] для взаємодії з базою даних та бібліотеки Pydantic v2 для суворої валідації вхідних і вихідних схем даних.

– Сховище даних: використання реляційної системи управління базами даних MySQL 8 з обов'язковим логуванням та керуванням структурою таблиць через систему міграцій Alembic [6].

– Стек технологій frontend: побудова інтерфейсу за допомогою чистого HTML, CSS та ванільного JavaScript (ES modules) з використанням бібліотеки Chart.js (через CDN) [10] для побудови графіків, що мінімізує оверхед на клієнтській стороні.

– Криптографічний захист: забезпечення безпечного зберігання токенів інтеграції з банками за допомогою симетричного шифрування алгоритмом Fernet [9].

3. Вимоги до забезпечення якості та розгортання

В рамках роботи необхідно виконати інженерні завдання з верифікації та підтримки життєвого циклу ПЗ:

– Тестування системи: розробка покриття коду автоматизованими unit тестами за допомогою фреймворку pytest [8] та інструменту TestClient (із використанням ізольованої бази даних SQL in-memory для тестового середовища). Кількість тестів має забезпечувати стабільну перевірку основних бізнес-сценаріїв (авторизація, логіка цілей та правил).

– Розгортання та портативність: створення чіткої інструкції для локального запуску системи у віртуальному середовищі (venv) та конфігурування параметрів додатку (база даних, JWT-секрети, ключі шифрування) за допомогою файлів змінних (.env).

Висновок до розділу

У розділі проведено аналіз предметної області та досліджено існуючі методи автоматизації персонального фінансового обліку. Сформовано набір функціональних вимог до перспективної інформаційної системи та обґрунтовано вибір технологічного стека, що дозволить ефективно вирішувати задачі консолідації та аналізу банківських даних. Отримані результати стали базою для подальшого архітектурного проєктування системи

2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРНІ РІШЕННЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Функції та специфікація вимог до системи

У цьому підрозділі наведено детальний опис практичної реалізації функціональних можливостей інформаційної системи «Finance Planner». Розроблений вебдодаток забезпечує повний цикл обробки, автоматизації та візуалізації особистих фінансових потоків. У програмі реалізовано чіткий поділ функцій за напрямками їхньої роботи, що дозволяє користувачеві ефективно взаємодіяти з системою.

На рисунку 2.1. наведено графічну схему у вигляді діаграми, яка відображає структуру реалізованих у програмі функцій.

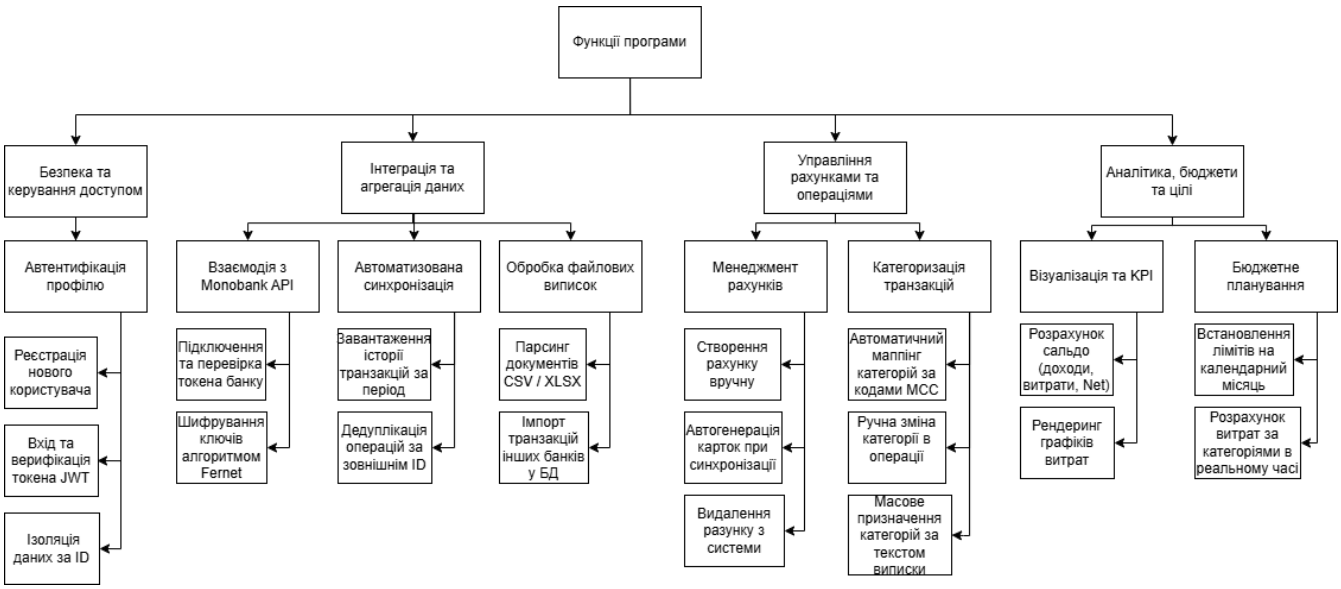

Рисунок 2.1 – Діаграма функціональної декомпозиції ІС

Далі наведено детальний опис практичної роботи та внутрішньої логіки кожної функції, яка представлена на схемі.

1. Безпека та керування доступом

Ця група функцій забезпечує повну ізоляцію фінансових даних користувачів

та захищає систему від несанкціонованого доступу до бази даних [2,19].

– Реєстрація нового користувача: Функція забезпечує створення нового облікового запису в системі. Вона приймає введену електронну пошту та пароль, виконує обов'язкову перевірку на наявність дублікатів email у базі даних, а потім шифрує пароль у захищений хеш за допомогою алгоритму pbkdf2_sha256 [9]. У відкритому вигляді паролі користувачів у системі не зберігаються.

– Вхід та верифікація токена JWT: Функція відповідає за перевірку облікових даних під час авторизації. Якщо введений логін і пароль збігаються з хешем у базі, програма генерує цифровий ключ – сесійний токен JWT. Браузер отримує цей ключ, зберігає його в localStorage і автоматично додає до кожного HTTP-запиту, підтверджуючи права користувача на виконання операцій.

– Ізоляція даних за ID користувача: Функція контролює кожен запит до бази даних. Усі рахунки, бюджети та транзакції мають прив'язку до унікального ідентифікатора власника (user_id). Програма автоматично перевіряє цей маркер при кожній дії та відображає або змінює тільки ті записи, які належать конкретному авторизованому користувачу.

2. Інтеграція та імпорт даних

Блок функцій, який відповідає за автоматичне наповнення системи первинними даними про рух коштів, повністю звільняючи користувача від ручного введення операцій.

– Підключення Monobank токена: Функція реалізує пряму інтеграцію з банком [15]. Користувач додає свій індивідуальний ключ доступу (токен) у налаштуваннях програми. Система робить тестовий запит до API банку для перевірки його валідності, після чого надійно зашифровує цей токен алгоритмом Fernet і зберігає в базі даних.

– Синхронізація транзакцій без дублікатів: Функція виконує асинхронне завантаження історії покупок і доходів користувача через інтернет. Вона звертається до серверів банку, отримує список транзакцій за останні 30 днів та перевіряє унікальний ID кожної операції. Якщо така покупка вже записана в базу, функція її пропускає, що гарантує точність балансу без подвоєння сум.

– Парсинг файлів виписок CSV/XLSX: Функція розроблена для імпорту даних з установ, які не мають відкритого API. Користувач завантажує файл виписки, а програма автоматично розбирає його структуру, знаходить колонки з датою, сумою та текстовим описом операцій, після чого переносить їх у загальну таблицю транзакцій [12,13].

3. Управління рахунками та операціями

Ці функції призначені для поточної обробки, редагування та структурування всіх фінансових рахунків і категорій всередині додатка.

– Менеджмент рахунків (CRUD): Набір інструментів, що дозволяє користувачеві створювати рахунки вручну (наприклад, для обліку готівки) або автоматично генерувати карткові рахунки під час імпорту даних із банку. Функція також дозволяє змінювати назви рахунків та видаляти їх разом із пов'язаною історією.

– Автоматичний маппінг категорій за MCC: Функція автоматично сортує витрати за категоріями одразу під час їх потрапляння в систему. Вона зчитує чотиризначний код MCC (тип торговельної точки), який додає банк до кожної операції, та за внутрішнім довідником автоматично присвоює категорію (наприклад, код 5411 перетворюється на «Продукти», а 5541 – на «АЗС»).

– Ручне та масове коригування категорій: Функція дає можливість користувачеві самостійно змінити категорію будь-якої транзакції через швидкий PATCH-запит прямо з інтерфейсу таблиці. Також реалізовано алгоритм масового оновлення за текстовим фільтром – можна створити правило, яке знаходитиме певне слово в описі (наприклад, "Uber") і автоматично присвоїть усім цим витратам категорію "Транспорт".

4. Аналітика, бюджети та цілі

Група функцій, яка здійснює математичну обробку фінансової історії, виводить звіти та допомагає керувати накопиченнями.

– Розрахунок фінансового сальдо (KPI): Функція агрегує фінансові показники за обраний період (тиждень, місяць або квартал). Вона сумує всі надходження, віднімає витрати та вираховує чистий залишок (сальдо). Усі

математичні операції з грошима проводяться на бекенді виключно в цілих числах (у копійках) для захисту від помилок округлення.

– Рендеринг аналітичних графіків Chart.js: Ця функція відповідає за візуальну частину аналітики. Вона бере готові цифри з бекенду і будує на головному екрані інтерактивні кругові діаграми розподілу витрат за категоріями у відсотках, а також лінійні графіки зміни загального балансу по днях [10].

– Контроль лімітів місячних бюджетів: Функція забезпечує роботу системи лімітів. Користувач встановлює планову суму витрат на місяць для певної категорії. Програма автоматично підраховує поточні витрати за цією категорією за календарний місяць і виводить смугу прогресу, яка наочно показує відсоток використання бюджету.

– Обчислення накопичень за правилами цілей: Функція управляє процесом збору коштів на фінансові цілі. Вона підтримує три алгоритми роботи: ручне додавання сум користувачем, автоматичне зчитування поточного залишку з реального рахунку (баланс картки) або автоматичне відкладання за правилами – коли система сама знаходить доходи за фільтрами (наприклад, слово «Зарплата» в описі) і автоматично зараховує відсоток від них у прогрес виконання цілі.

2.2 Поведінкове моделювання взаємодії користувача з інформаційною системою

Для забезпечення коректного функціонування інформаційної системи та автоматизації обробки фінансових даних було розроблено моделі поведінки користувача. Ці моделі відображають послідовність кроків, які виконує система під час взаємодії, включаючи етапи валідації, обробки та збереження даних.

Нижче наведено опис ключових процесів, візуалізованих у вигляді Use case діаграм.

Сценарій описує процес автентифікації зареєстрованого користувача та додавання API банку, імпорту банківських виписок, створення лімітів та керування цілями. Логіку цього покрокового алгоритму відображено на рисунку 2.2.

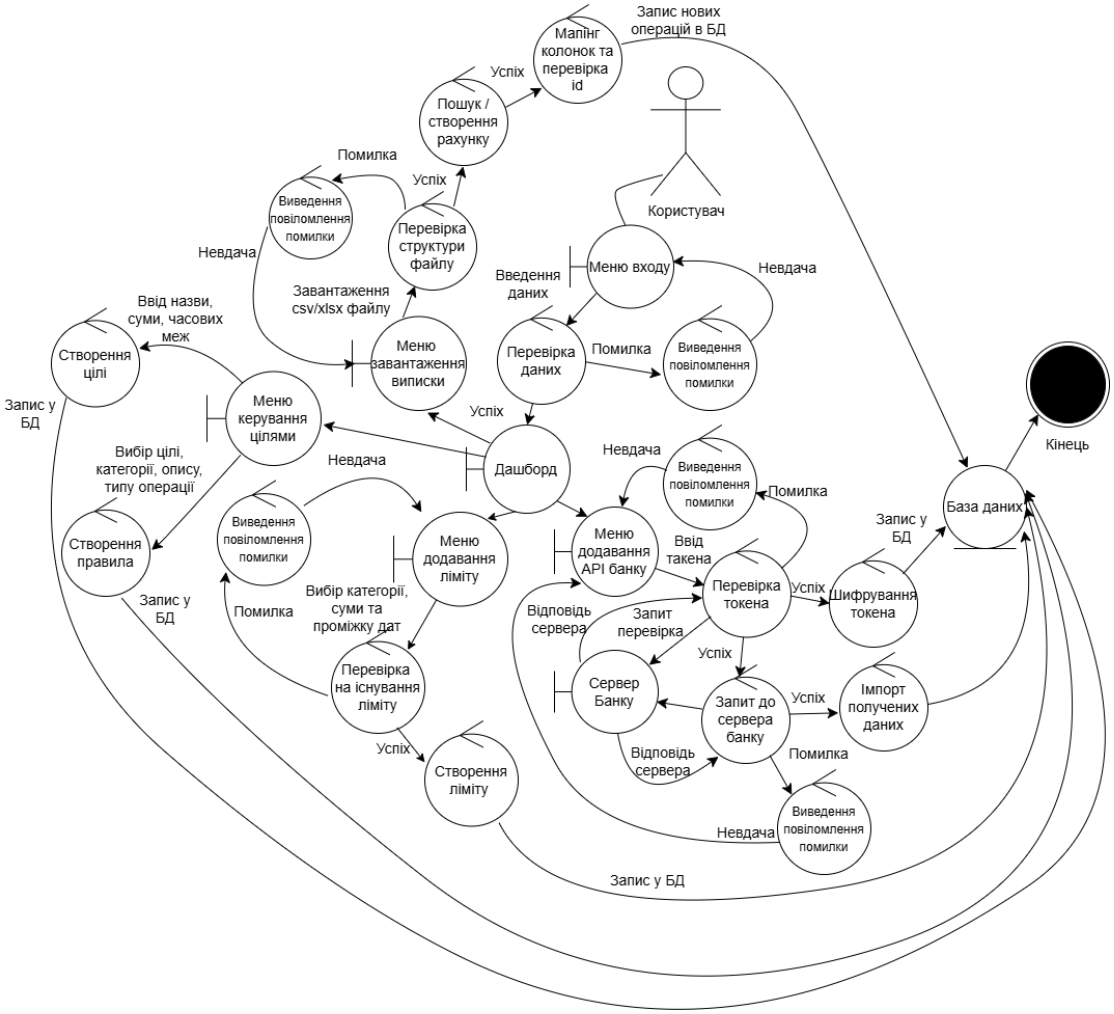

Рисунок 2.2 – Use-Case діаграма процесу керування додатком

Процес входу користувача в систему, верифікації його повноважень та безпечного збереження банківських реквізитів виконується за таким чітким покроковим алгоритмом:

1. Ініціалізація сесії користувача: Користувач взаємодіє з об'єктом грані «Форма входу (Boundary)», де вводить свої персональні авторизаційні дані (Email та пароль) і надсилає запит на сервер.

2. Автентифікація на бекенді: Введені дані у вигляді POST-запиту передаються на об'єкт керування «Контролер автентифікації (Control)» сервера FastAPI [4]. Контролер виконує перевірку наявності запису, надсилаючи запит до сутності «Таблиця users (Entity)» у базі даних MySQL.

3. Обробка розгалуження авторизації:

Увипадку помилки (неправильний логін або пароль) потік керування перенаправляється на «Обробник помилок (Control)», який повертає користувача на початковий екран із повідомленням про помилку автентифікації (HTTP 401).

У випадку успіху система активує «Генератор токенів (Control)», створює сесійний JWT-токен і перенаправляє користувача на об'єкт грані «Дашборд / Головна панель (Boundary)».

4. Перехід до інтеграції: З головної панелі користувач через навігаційне меню переходить до об'єкта грані «Налаштування інтеграцій (Boundary)», де вставляє свій персональний API-токен Monobank для активації автоматичного обліку.

5. Валідація та верифікація токена: Бекенд-сервер приймає рядок токена через об'єкт керування «Валідатор API-токена (Control)». Для перевірки працездатності ключа система здійснює прямий зовнішній HTTP-запит до стороннього актора – «Сервера API Monobank» [3].

6. Отримання метаданих: Сервер банку верифікує ключ і повертає JSON-відповідь із даними клієнта та переліком його відкритих карток. Об'єкт керування «Модуль парсингу (Control)» обробляє ці дані й автоматично створює відповідні записи (дзеркала карток) у сутності «Таблиця accounts (Entity)» бази даних MySQL, фіксуючи поточні баланси, назви банків та валюту рахунків.

7. Криптографічний захист: Після успішного створення рахунків чистий рядок токена передається на «Модуль шифрування Fernet (Control)», який виконує симетричне заплутування даних для забезпечення високого рівня безпеки зберігання приватних ключів [9].

8. Фіксація інтеграції: Зашифрований токен записується у сутність «Таблиця bank_integrations (Entity)» із прив'язкою до унікального ідентифікатора поточного користувача. Об'єкт грані «Налаштування інтеграцій» оновлює свій стан і виводить користувачеві повідомлення про успішне підключення банку та старт первинного імпорту фінансової історії.

Процес імпорту банківської виписки.

Механізм імпорту даних з файлів дозволяє користувачу завантажувати виписки у форматах CSV або XLSX. Система послідовно виконує такі етапи:

1. Валідація: перевірка структури файлу на відповідність очікуваному формату.

2. Аналіз рахунку: пошук або автоматичне створення відповідного рахунку в системі.

3. Мапінг та перевірка: зіставлення колонок файлу з атрибутами системи та перевірка ідентифікаторів транзакцій для запобігання дублюванню. Після проходження всіх перевірок дані записуються до бази даних, що завершує цикл обробки виписки.

Процес налаштування фінансового планування поділяється на два основні потоки:

Встановлення лімітів: користувач обирає категорію, суму та часовий проміжок. Система перевіряє наявність активних лімітів для цього періоду, щоб уникнути конфліктів, після чого зберігає налаштування в базі даних.

Створення фінансових цілей: користувач задає параметри цілі, а також формує правила автоматизації, що визначають критерії врахування транзакцій. Система записує як саму ціль, так і асоційовані з нею правила, забезпечуючи подальшу автоматичну обробку фінансових надходжень згідно з визначеною логікою.

2.3 Часове та хронологічне моделювання компонентів системи

Для дослідження динаміки функціонування інформаційної системи «Finance Planner» у часі,

визначення хронологічного порядку обміну повідомленнями між компонентами програми та аналізу життєвого циклу процесів застосовано поведінкове моделювання на рівні взаємодії об'єктів. Центральним інструментом для відображення часової послідовності виконання операцій обрано UML-діаграму послідовностей (Sequence Diagram).

З огляду на архітектурну розгалуженість системи та високу детальність взаємодії між клієнтською частиною (Frontend), серверною логікою (FastAPI) та СКБД MySQL, хронологічне моделювання наскрізного бізнес-процесу було розділено на три послідовні логічні фази:

1. Фаза автентифікації користувача та генерації сесії.
2. Фаза підключення банку та імпорту метаданих рахунків.
3. Фаза криптографічного захисту токена та фіксації інтеграції.

Фаза автентифікації користувача

Перша фаза моделювання відображає часовий інтервал від моменту надсилання облікових даних користувачем до перенаправлення його в робочий простір системи після успішної перевірки повноважень. Графічну модель цього етапу представлено на рисунку 2.3.

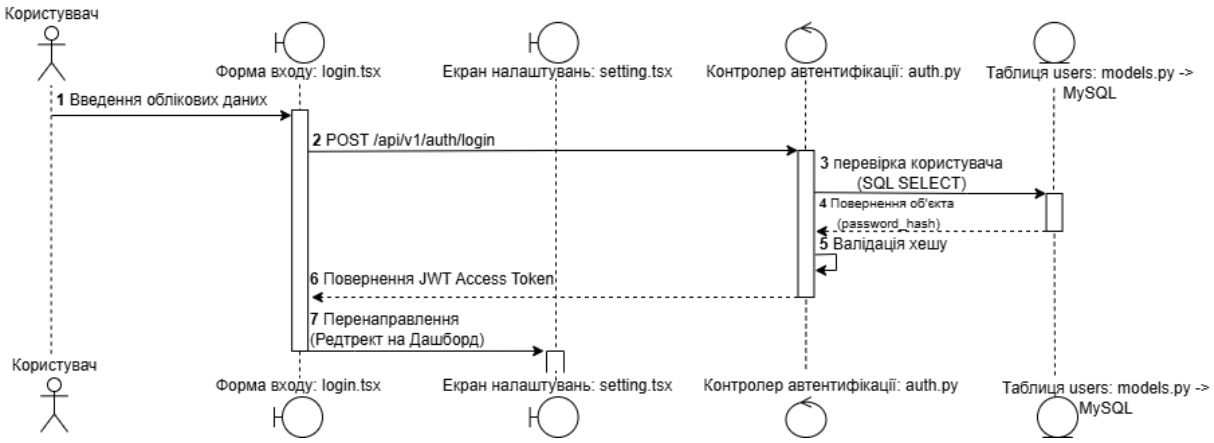

Рисунок 2.3 – UML-діаграма послідовності автентифікації користувача

Опис функціонування фази автентифікації:

1. Процес починається з того, що Користувач вводить свої ідентифікаційні дані у вебформу, активуючи життєвий цикл компонента login.tsx.

2. Інтерфейсний рівень формує асинхронний HTTP-запит POST /api/v1/auth/login, передаючи тіло запиту на бекенд-роутер auth.py.

3. Контролер автентифікації ініціює запит до реляційної бази даних MySQL, виконуючи селекцію запису з Таблиці users за унікальним атрибутом Email.

4. База даних повертає знайдений об'єкт користувача, який містить зашифрований рядок password_hash.

5. На стороні бекенду виконується внутрішня процедура верифікації відповідності відкритого пароля та знятого хешу за допомогою бібліотеки pwd_context.

6. При успішній валідації сервер генерує сесійний JWT-токен доступу (Access Token) і відправляє його у JSON-відповіді клієнту.

7. Компонент Forma входу зберігає отриманий токен у локальному сховищі браузера (localStorage) та ініціює редирект, викликаючи життєвий цикл екрана налаштувань (settings.tsx).

Фаза підключення банку та імпорту рахунків

Друга фаза описує хронологію взаємодії вебаплікації із зовнішнім середовищем (API Monobank) для перевірки працездатності доданого токена та автоматичного формування структури рахунків користувача. Графічну модель процесу представлено на рисунку 2.4.

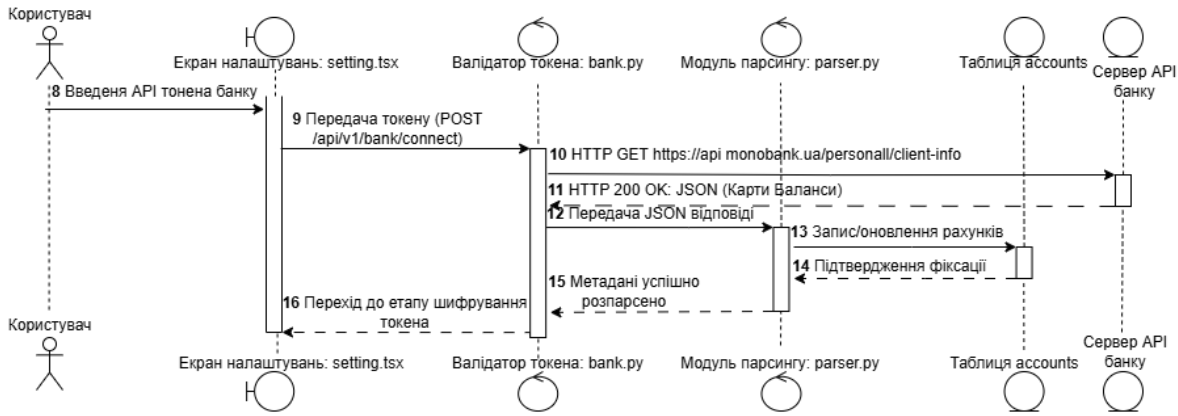

Рисунок 2.4 – UML-діаграма послідовності імпорту рахунків

Опис функціонування фази імпорту рахунків:

1. Перебуваючи в особистому кабінеті, Користувач вводить індивідуальний рядок API-ключа у відповідне поле settings.tsx.

2. Інтерфейс відправляє токен на серверний сервіс bank.py, викликаючи ендпоінт /api/v1/bank/connect.

3. Об'єкт керування Валідатор токена здійснює зовнішній синхронний HTTP-запит до віддаленого сервера для перевірки активності наданого ключа.

4. Сервер банку верифікує токен і повертає структурований JSON-пакет, що містить ідентифікатори карток, типи валют та поточні залишки на рахунках клієнта.

5. Валідатор токена делегує обробку отриманого пакета Модулю парсингу (parser.py).

6. Модуль парсингу ітерує масив карток і виконує операції INSERT / UPDATE у Таблицю accounts СКБД MySQL, створюючи локальні дзеркала банківських рахунків із приведенням балансів до цілочисельного формату (у копійках).

7. Після підтвердження транзакції базою даних Модуль парсингу звітує про успішне завершення первинного імпорту структур даних, і система переходить до фази криптографічного захисту сесії.

Фаза криптографічного захисту та фіксації інтеграції.

Третя фаза хронологічного моделювання відображає ізольовані процеси забезпечення інформаційної безпеки проєкту – шифрування відкритого значення API-ключа алгоритмом симетричного заплутування та фінальний запис параметрів у базу даних. Графічну модель представлено на рисунку 2.5.

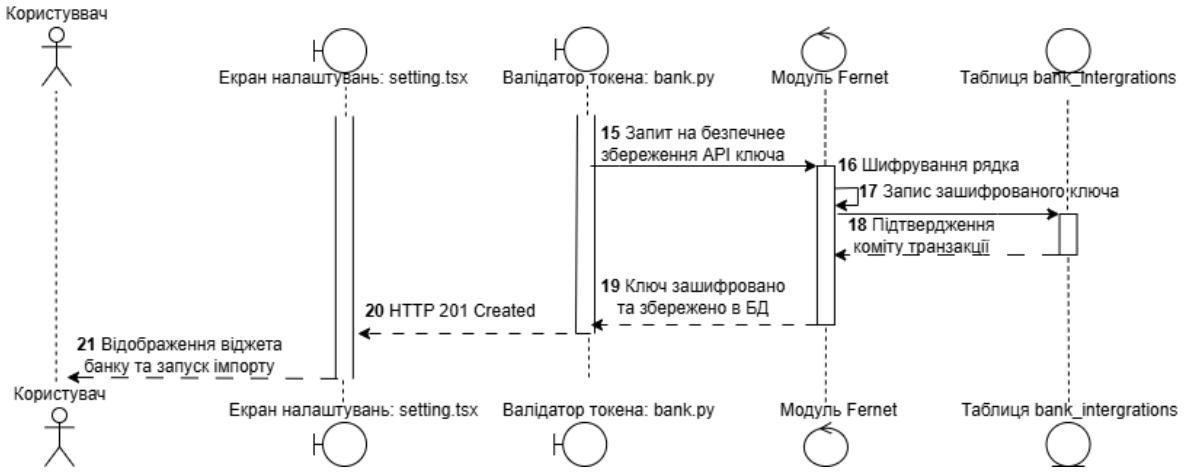

Рисунок 2.5 – UML-діаграма послідовності захисту та інтеграції даних

Опис функціонування фази криптографічного захисту:

1. Отримавши підтвердження про створення структури рахунків, валідатор токена передає відкритий рядок ключа до внутрішнього системного компонента безпеки security.py.

2. Об'єкт керування модуль Fernet за допомогою криптографічних бібліотек виконує процедуру симетричного шифрування токена, генеруючи безпечний рядок шифротексту secret_enc.

3. Обчислений шифротекст передається в СКБД MySQL, де виконується операція INSERT у Таблицю bank_integrations із обов'язковим зв'язуванням запису за зовнішнім ключем user_id.

4. База даних фіксує транзакцію та повертає підтвердження успішного комміту.

5. Модуль Fernet закриває свій життєвий цикл, повертаючи потік керування на головний сервіс [9] .

6. Валідатор токена відправляє на клієнтську частину фінальний статус успішної обробки (HTTP 201 Created).

7. Компонент інтерфейсу settings.tsx деактивує стан очікування, динамічно відмальовує підключений віджет Monobank на екрані та сповіщає Користувача про успішний старт фонового імпорту транзакцій, завершуючи роботу всього бізнес-сценарію.

2.4 Проєктування логічної та фізичної структури баз даних

Для збереження та забезпечення цілісності інформації у вебдодатку «Finance Planner» спроєктовано реляційну базу даних на базі СКБД MySQL з повною підтримкою транзакційного механізму ACID [7].

Взаємодія додатка із СУБД реалізована через об'єктно-реляційне відображення (ORM) SQLAlchemy [5] за допомогою декларативних класів мови Python. Для коректної обробки кирилиці підключення здійснюється через драйвер

PyMySQL із кодуванням utf8mb4, а параметри з'єднання винесені у змінні середовища файлу .env.

Завдяки концепції self-hosted, сервер FastAPI та база даних MySQL функціонують локально на ресурсах користувача, що гарантує приватність фінансових даних. При цьому система є багатокористувацькою: дані зберігаються в єдиній базі, а суворе розмежування доступу між обліковими записами реалізовано на рівні програмної логіки через ідентифікатори користувачів.

Логічна структура розробленої бази даних, яка відображає набір таблиць, їхні внутрішні поля та типи даних, представлена на рисунку 2.6.

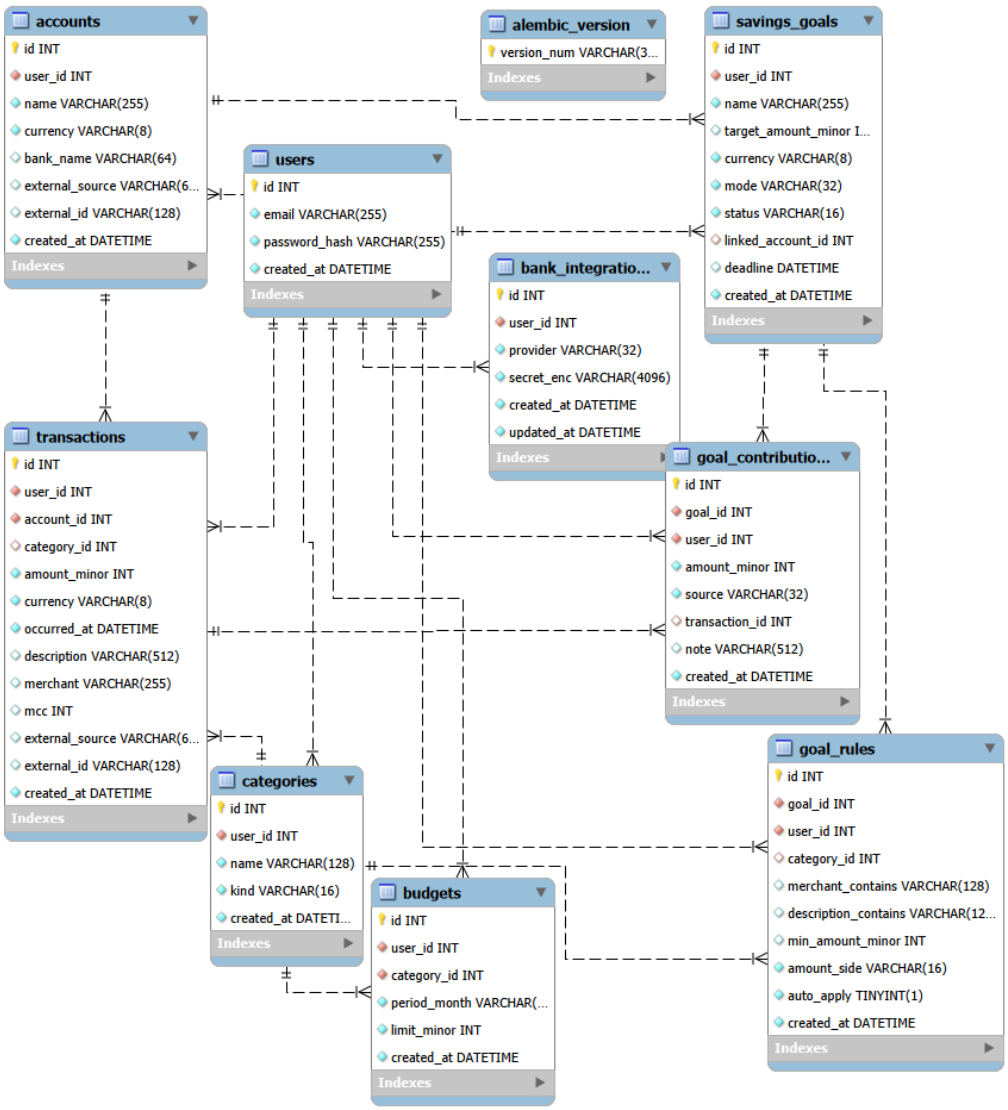

Рисунок 2.6 – Структура бази даних

Специфікація та детальний опис таблиць бази даних

1. Як показано в таблиці 2.1 users є кореневим елементом схеми даних, призначеним для реєстрації, автентифікації та розмежування доступу користувачів у системі.

Таблиця 2.1 – Users

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT	так	Первинний ключ (PK). Унікальний цифровий маркер користувача.
email	VARCHAR(255)	так	Електронна пошта для авторизації. Має прапорець UNIQUE.
password_hash	VARCHAR(255)	так	Хеш-код пароля користувача
created_at	DATETIME	так	створення облікового запису.

Ключі та індекси: Поле id виступає в ролі первинного ключа (Primary Key). На стовпець email накладено суворе обмеження унікальності (Unique Constraint), що технічно унеможливлює створення дублікатів профілів. Також для поля email згенеровано базу індексів (Index) для забезпечення максимальної швидкості виконання операцій пошуку запису при авторизації (вхід у систему).

Логіка зв'язків: Від первинного ключа users.id відходять зв'язки типу «Один-до-багатьох» (One-to-Many) до всіх основних операційних таблиць системи: accounts, categories, transactions, budgets та savings_goals. Конфігурація бази даних передбачає налаштування обмеження зовнішнього ключа з умовою ON DELETE CASCADE. Це означає, що при ліквідації облікового запису користувача СКБД MySQL автоматично і безповоротно очистить усі пов'язані з ним фінансові дані, рахунки та історію дій.

2. У таблиці 2.2 наведено структуру accounts, яка призначена для обліку доступних користувачеві платіжних інструментів, включаючи платіжні картки банківських установ, імпортовані виписки та ручні гаманці.

Ключі та індекси: Поле id – первинний ключ. Поле user_id функціонує як зовнішній ключ (Foreign Key), що зв'язує таблицю з users.id через каскадне видалення ON DELETE CASCADE.

Таблиця 2.2 – Accounts

Колонка	Тип	Обов’язковість	Опис та технічне призначення поля
id	INT PK AI	так	Первинний ключ (PK) платіжного рахунку в системі.
user_id	INT	так	Зовнішній ключ (FK). Посилання на власника рахунку.
name	VARCHAR(255)	так	Кастомне текстове найменування для відображення в інтерфейсі.
currency	VARCHAR(8)	так	Міжнародний код валюти рахунку (UAH, USD, EUR).
bank_name	VARCHAR(64)	ні	Текстова назва банку-емітента (наприклад, Monobank).
external_source	VARCHAR(64)	ні	Джерело походження (monobank, csv_import, manual).
external_id	VARCHAR(128)	ні	Унікальний ідентифікатор рахунку, наданий зовнішнім API банку.
created_at	DATETIME(tz)	так	Час інтеграції або створення рахунку в системі.

Логіка зв'язків та обмежень: У таблиці налаштовано складене унікальне обмеження uq_accounts_external, що охоплює комбінацію стовпців (user_id, external_source, external_id). Дане обмеження виконує захисну функцію: під час повторної автоматичної синхронізації з банківськими API система не створює нові рядки-дублікати, а оновлює баланс наявного рахунку. Сутність пов'язана відношенням «Один-до-багатьох» з таблицею transactions (один рахунок містить безліч операцій) та опціонально може посилатися на фінансову ціль через зовнішній ключ у таблиці savings_goals.

3. Для безпечного збереження параметрів підключення та авторизаційних маркерів доступу до відкритих API банківських установ використовується bank_integrations, структура якої наведена нижче в таблиці 2.3.

Ключі та індекси: Поле id – первинний ключ. Зв'язок із таблицею користувачів реалізовано через зовнішній ключ user_id з умовою ON DELETE CASCADE.

Таблиця 2.3 – Bank_integrations

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT PK AI	так	Первинний ключ (PK) запису інтеграції.
user_id	INT FK CASCADE	так	Зовнішній ключ (FK) для ідентифікації власника токена.
provider	VARCHAR(32)	так	Назва банку-провайдера
secret_enc	VARCHAR(4096)	так	Токен після Fernet-шифрування
created_at	DATETIME(tz)	так	Перше підключення
updated_at	DATETIME(tz)	так	Дата останнього оновлення або реактивації токена.

Логіка зв'язків та обмежень: На рівні СКБД MySQL накладено складене обмеження унікальності на поля (user_id, provider). Це реалізує логіку відношення «Один-до-одного» (One-to-One) у межах конкретного банку: у користувача може бути лише один активний профіль інтеграції з Monobank. Будь-які повторні запити на підключення не продукують нових записів, а оновлюють часову мітку updated_at та шифрований ключ secret_enc. Прямих зв'язків із таблицею транзакцій сутність не має.

4. Наведена нижче таблиця 2.4, містить структуру categories, яка призначена для систематизації та класифікації доходів і витрат користувачів.

Ключі та індекси: Поле id – первинний ключ. Поле user_id є зовнішнім ключем до users.id (ON DELETE CASCADE).

Таблиця 2.4 – Categories

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT	так	Первинний ключ (PK) фінансової категорії.
user_id	INT	так	Зовнішній ключ (FK) для прив'язки до профілю.
name	VARCHAR(128)	так	Текстова назва категорії (наприклад, «Продукти»)
kind	VARCHAR(16)	так	Напрямок операцій: expense (витрата) або income (дохід)
created_at	DATETIME(tz)	так	Час створення категорії в базі даних

Логіка зв'язків та обмежень: Комбінація полів (user_id, name) захищена обмеженням унікальності, що унеможливлює створення дублікатів з однаковими назвами для одного користувача. Таблиця пов'язана відношенням «Один-до-багатьох» з транзакціями (transactions.category_id) , місячними лімітами (budgets.category_id) та правилами автоматизації цілей (goal_rules.category_id). При видаленні категорії для транзакцій застосовується правило ON DELETE SET NULL, а пов'язані планові ліміти видаляються каскадом.

5. Наведена нижче таблиця 2.5 містить структуру transactions, яка є центральним сховищем фінансових записів, де акумулюється весь масив даних про рух грошей із різних джерел.

Таблиця 2.5 – Transactions

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT	так	Первинний ключ (PK) фінансової категорії.
user_id	INT	так	Зовнішній ключ (FK) для прив'язки до профілю.
account_id	INT	так	Зовнішній ключ (FK). Зв'язок із платіжним рахунком.
category_id	INT	ні	Зовнішній ключ (FK). Категорія операції.
amount_minor	INT	так	Сума операції в копійках (знак визначає вектор).
currency	VARCHAR(8)	так	Валюта, в якій проведено транзакцію.
occurred_at	DATETIME(tz), INDEX	так	Фактичний час здійснення транзакції банком.
description	VARCHAR(512)	ні	Текстове призначення платежу або коментар
merchant	VARCHAR(255)	ні	Назва торговельної точки чи контрагента
mcc	INT	ні	Чотиризначний код категорії торговця (MCC).
external_source	VARCHAR(64)	ні	Маркер походження записа (monobank, csv_import)
external_id	VARCHAR(128)	ні	Унікальний ідентифікатор транзакції від банку.
created_at	DATETIME(tz)	так	Час реєстрації запису в базі даних проєкту

Ключі та індекси: Поле id – первинний ключ. Поля user_id та account_id реалізовані як зовнішні ключі з умовою ON DELETE CASCADE (видалення картки чи профілю повністю видаляє історію пов'язаних дій). Поле category_id є зовнішнім ключем до categories.id із правилом ON DELETE SET NULL. Для оптимізації аналітичних запитів створено індекси на поля user_id, account_id, category_id та occurred_at.

Логіка зв'язків: Складене обмеження унікальності на стовпці (user_id, external_source, external_id) гарантує відсутність дублювання записів при повторних викликах синхронізації з банком. Таблиця перебуває у відношенні «Багато-до-одного» до користувачів, рахунків та категорій. Опціонально транзакція може ініціювати створення внесків у фінансові цілі.

6. Для збереження планових обсягів фінансових витрат, які встановлюються користувачами на певні часові інтервали, розроблено budgets, опис якої представлено нижче в таблиці 2.6.

Таблиця 2.6 – Budgets

Колонка	Тип	Обов’язковість	Опис та технічне призначення поля
id	INT PK AI	так	Первинний ключ (PK) запису ліміту.
user_id	INT FK CASCADE	так	Зовнішній ключ (FK) для ідентифікації власника.
category_id	INT FK CASCADE	так	Зовнішній ключ (FK). Категорія для лімітування.
period_month	VARCHAR(7), INDEX	так	Цільовий місяць планування у форматі YYYY-MM.
limit_minor	INT	так	Граничний розмір бюджету в копійках.
created_at	DATETIME(tz)	так	Дата та час формування запису ліміту

Ключі та індекси: Поле id – первинний ключ. Поля user_id та category_id є зовнішніми ключами, які підтримують каскадне видалення ON DELETE CASCADE. Для прискорення генерації звітів на стовпець period_month накладено пошуковий індекс.

Логіка зв'язків та обмежень: Складена унікальність полів (user_id, category_id, period_month) технічно забороняє створення декількох паралельних

лімітів на одну й ту саму категорію в межах одного розрахункового місяця. Таблиця не зберігає поточний обсяг витрачених коштів – це значення розраховується динамічно шляхом агрегації сум із таблиці transactions за збігом категорій та часових меж.

7. Наведена нижче таблиця 2.7 описує структуру savings_goals, яка акумулює дані щодо довгострокових або короткострокових планів накопичення грошових коштів користувачами системи.

Таблиця 2.7 – Savings_goals

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT PK AI	так	Первинний ключ (PK) цілі накопичення.
user_id	INT FK CASCADE	так	Зовнішній ключ (FK) власника цілі.
name	VARCHAR(255)	так	Текстове найменування фінансової цілі.
target_amount_minor	INT	ні	Цільова фінішна сума збору в копійках.
currency	VARCHAR(8)	так	Валюта накопичення цілі
mode	VARCHAR(32)	так	Як рахувати прогрес (див. нижче)
status	VARCHAR(16)	так	Поточний стан цілі
linked_account_id	INT FK SET NULL	ні	Зовнішній ключ (FK). Прив'язаний платіжний рахунок.
deadline	DATETIME(tz)	ні	Кінцева календарна дата виконання плану
created_at	DATETIME(tz)	так	Час ініціалізації цілі в системі.

Ключі та індекси: Поле id – первинний ключ. Поле user_id – зовнішній ключ до users.id (ON DELETE CASCADE). Поле linked_account_id є опціональним зовнішнім ключем до таблиці accounts.id із правилом ON DELETE SET NULL.

Логіка зв'язків: Таблиця виступає батьківським об'єктом відношення «Один-до-багатьох» для сутностей goal_contributions (історія внесків) та goal_rules (критерії автоматичного перехоплення транзакцій). Показники відсоткового

виконання цілі не дублюються в базі даних, а розраховуються на рівні сервісів при кожному зверненні користувача.

8. Роль журналу транзиту грошових коштів, який фіксує кожне поповнення або списання з рахунку конкретної фінансової цілі, виконує goal_contributions, структура якої наведена нижче в таблиці 2.8.

Таблиця 2.8 – Goal_contributions

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT PK AI	так	Первинний ключ (PK) запису внеску.
goal_id	INT FK CASCADE	так	Зовнішній ключ (FK). Ціль, куди зараховано кошти.
user_id	INT FK CASCADE	так	Зовнішній ключ (FK). Дублюючий індекс власника.
amount_minor	INT	так	Розмір грошового внеску в копійках.
source	VARCHAR(32)	так	Джерело походження: manual, transaction, rule
transaction_id	INT FK CASCADE	ні	Зовнішній ключ (FK). Зв'язок із базовою транзакцією.
note	VARCHAR(512)	ні	Текстова примітка до внеску.
created_at	DATETIME(tz)	так	Час проведення операції.

Ключі та індекси: Поле id – первинний ключ. Поля goal_id, user_id та transaction_id спроєктовані як зовнішні ключі з каскадним видаленням ON DELETE CASCADE.

Логіка зв'язків та обмежень: Таблиця goal_contributions технічно забезпечує реалізацію концепції зв'язку «Багато-до-багатьох» (Many-to-Many) між фінансовими операціями (transactions) та планами накопичення (savings_goals). На комбінацію стовпців (goal_id, transaction_id) накладено обмеження унікальності uq_goal_contribution_goal_tx, що виключає ризик повторного нарахування однієї банківської операції на ту саму ціль.

9. Для збереження користувацьких шаблонів та логічних фільтрів, за якими система аналізує банківську виписку для автоматичного відрахування коштів на фінансові цілі, призначена goal_rules, опис якої представлено нижче в таблиці 2.9.

Таблиця 2.9 – Goal_rules

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
id	INT PK AI	так	Первинний ключ (PK) логічного правила.
goal_id	INT FK CASCADE	так	Зовнішній ключ (FK). Ціль, для якої створено правило.
user_id	INT FK CASCADE	так	Зовнішній ключ (FK). Ідентифікатор розробника правила.
category_id	INT FK SET NULL	ні	Зовнішній ключ (FK). Обмеження за фінансовою категорією.
merchant_contains	VARCHAR(128)	ні	Фільтр за входженням підрядка в назву торговця.
description_contains	VARCHAR(128)	ні	Фільтр за входженням підрядка в опис операції.
min_amount_minor	INT	ні	Мінімальний модуль суми операції
amount_side	VARCHAR(16)	так	Вектор операцій для аналізу:
auto_apply	BOOLEAN	так	Прапорець фонового виконання правила після синхронізації.
created_at	DATETIME(tz)	так	Дата реєстрації правила в базі даних.

Ключі та індекси: Поле id – первинний ключ. Стовпці goal_id та user_id є зовнішніми ключами з каскадним типом видалення (ON DELETE CASCADE). Стовпець category_id посилається на categories.id з інструкцією ON DELETE SET NULL.

Логіка зв'язків: Таблиця перебуває у відношенні «Багато-до-одного» до сутностей цілей та категорій. Важливо зазначити, що ця структура не є тригером на рівні сервера MySQL. Правила інтерпретуються програмними сервісами Python під час імпорту чи синхронізації даних: у разі успішного збігу параметрів виписки із шаблоном правила, бекенд генерує операцію вставки (INSERT) нового рядка в таблицю внесків goal_contributions.

10. Для автоматичного контролю актуальності схеми даних та фіксації поточного стану бази даних використовується службова сутність alembic_version, опис якої наведено нижче в таблиці 2.10.

Таблиця 2.10 – Alembic_version

Колонка	Тип	Обов'язковість	Опис та технічне призначення поля
version_num	VARCHAR(32)	так	Первинний ключ (PK). Унікальний буквено-цифровий хеш поточної версії схеми бази даних.

Призначення інструменту Alembic та сутність міграцій. У процесі життєвого циклу розробки структура Python-моделей постійно змінюється, що потребує відповідного оновлення схем у СКБД MySQL. Для вирішення цієї проблеми використано інструмент Alembic, який виконує функцію системи контролю версій для схем баз даних.

Alembic автоматично зіставляє стан моделей SQLAlchemy з реальною архітектурою таблиць на сервері MySQL і генерує файли міграцій – інструкції для оновлення (upgrade) або відкату (downgrade) схеми. Під час кожного запуску вебдодатка FastAPI автоматично виконується команда alembic upgrade head. Вона зчитує хеш поточної версії з таблиці alembic_version і послідовно застосовує нові міграційні скрипти. Такий підхід забезпечує еволюцію структури бази даних без ризику втрати чи спотворення накопиченої фінансової історії користувачів.

1.5 Алгоритмічне забезпечення системи

Алгоритмічне забезпечення інформаційної системи «Finance Planner» спрямоване на автоматизацію збору, обробки та структурування фінансових даних. Основна логіка системи реалізована через два паралельні алгоритмічні шляхи отримання операцій: автоматизовану синхронізацію через банківські API та імпорт офлайн-виписок (CSV/XLSX) рисунок 2.7.

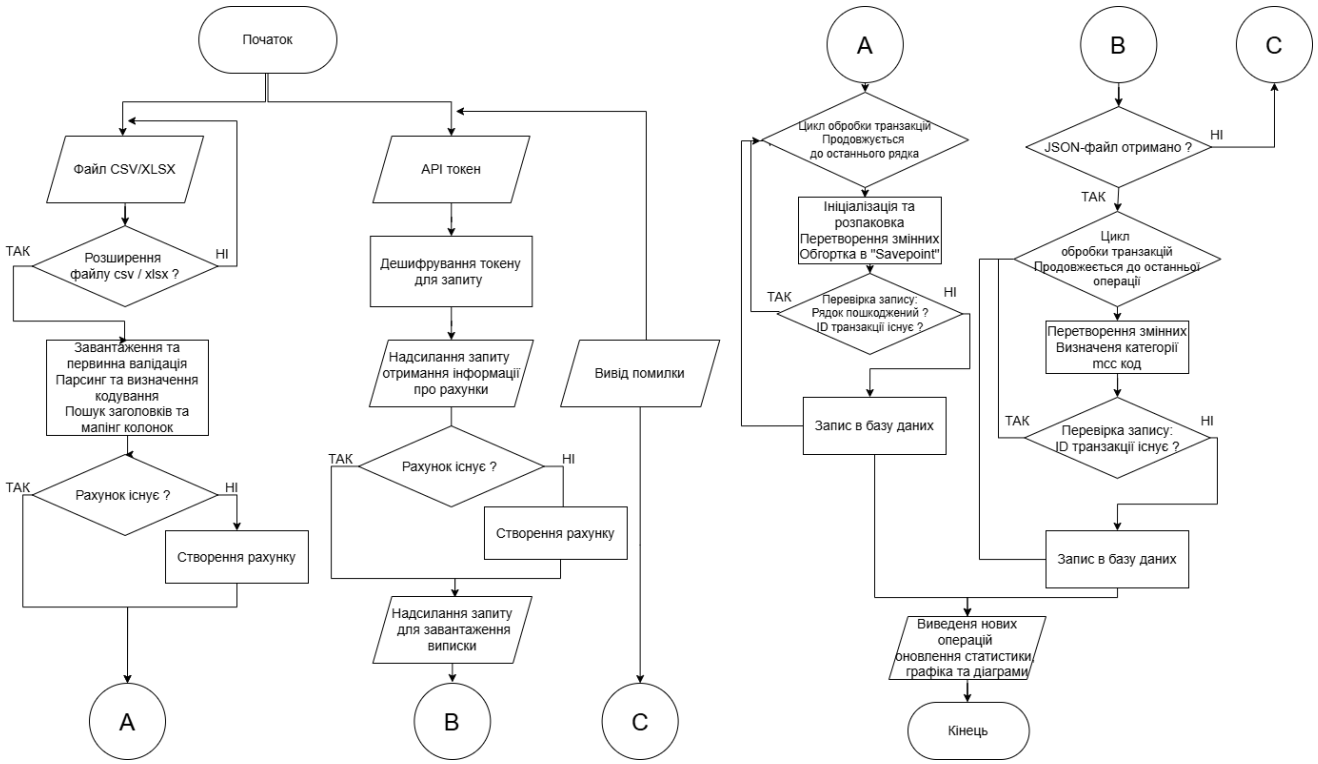

Рисунок 2.7 – Блок-схема алгоритму процесу збору даних

Алгоритм інтеграції з банківськими сервісами

Процес отримання даних через API (на прикладі Monobank) є автоматизованим і базується на використанні захищених токенів. Алгоритм включає наступні етапи:

1. Авторизація та дешифрування: Система отримує зашифрований API-токен із бази даних для конкретного користувача, дешифрує його та використовує для формування автентифікованих запитів.

2. Верифікація рахунків: Перед завантаженням транзакцій алгоритм перевіряє наявність активних рахунків у банку. Якщо рахунок у системі відсутній, здійснюється його автоматичне створення (прив’язка через external_id).

3. Ітеративна синхронізація: Система завантажує виписку за вказаний період. Кожен рядок обробляється в циклі, де ключовим етапом є перевірка унікальності транзакції за її ідентифікатором, що запобігає дублюванню даних у БД.

Алгоритм імпорту та парсингу файлових виписок

Для користувачів, які завантажують дані у вигляді файлів CSV або Excel, реалізовано алгоритм гнучкого парсингу:

1. Первинна валідація: Перевірка формату файлу та його розміру.

2. Структурний аналіз: Алгоритм автоматично визначає кодування та роздільники. Важливим етапом є функція пошуку заголовків, яка через систему псевдонімів (mappings) зіставляє довільні назви колонок у файлі з внутрішніми полями системи (дата, сума, опис).

3. Трансформація та хешування: Оскільки файли не мають нативних ID транзакцій, система генерує їх шляхом хешування (SHA-256) ключових параметрів операції. Це дозволяє запобігти дублюванню транзакцій при повторному завантаженні того самого файлу.

2.6. Архітектура модулів інформаційної системи

Для забезпечення високої масштабованості та підтримки цілісності даних у системі «Finance Planner» було розроблено об'єктно-орієнтовану модель даних. Діаграма класів слугує основним інструментом для візуалізації структури бази даних та логічних зв’язків між сутностями, що дозволяє формалізувати бізнес-правила системи на етапі проєктування.

Статична структура спроєктованої інформаційної системи наочно відображена на діаграмі класів, наведеній на рисунку 2.8

Ця діаграма демонструє архітектурну організацію системи, виокремлюючи ключові сутності, їхні атрибути та характерні взаємозв’язки між ними. Використання даної моделі дозволяє структурувати логіку обробки фінансових даних, визначаючи ієрархію та правила взаємодії компонентів, що є необхідним для коректної реалізації бази даних та подальшої програмної розробки.

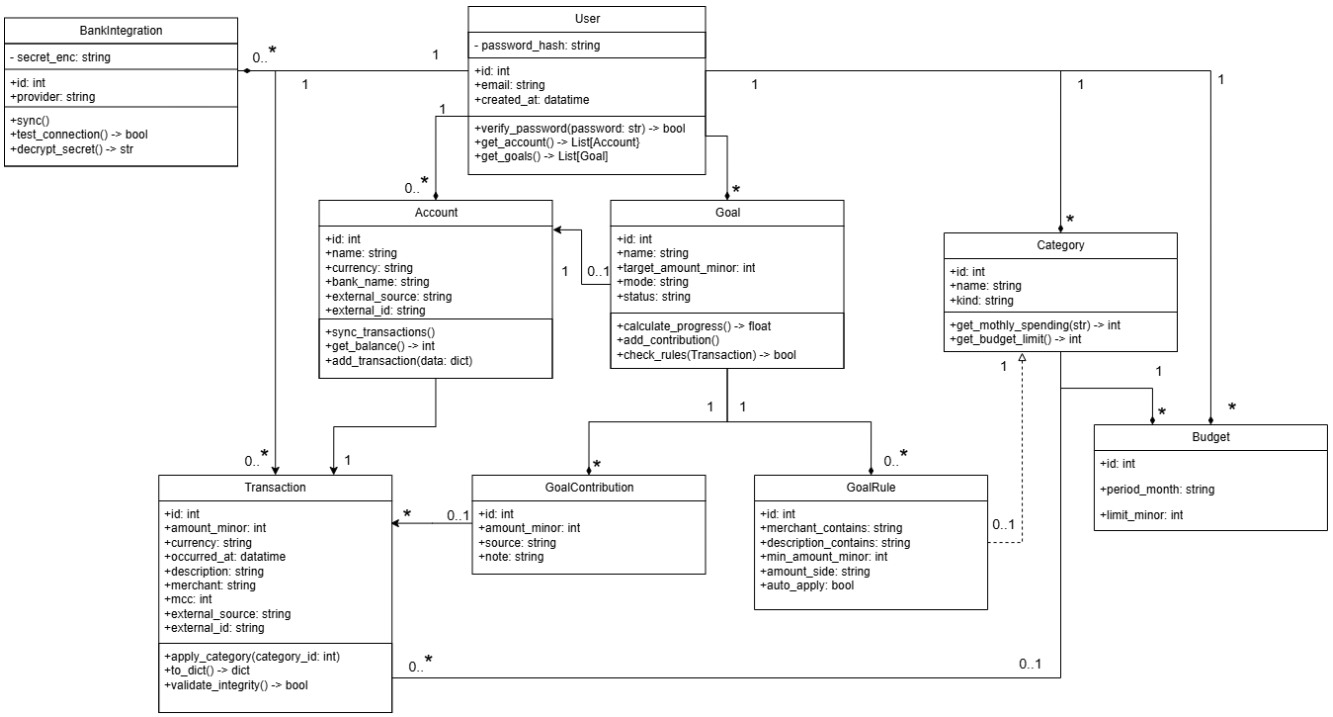

Рисунок 2.8 – Діаграма класів

Архітектура системи базується на взаємодії кількох модулів. Модуль ідентифікації (User) є кореневим і забезпечує ізоляцію даних користувача. За зв'язок із зовнішніми системами відповідає модуль банківської інтеграції (BankIntegration, Account), де перший об’єкт зберігає токени доступу, а другий групує транзакції за рахунками. Модуль фінансових операцій (Transaction, Category) акумулює дані про доходи й витрати, класифікує їх та взаємодіє з місячним бюджетом. Модуль планування (Goal, GoalContribution, GoalRule) забезпечує накопичення коштів, де клас GoalRule автоматично трансформує транзакції у внески на цілі.

Клас BankIntegration керує доступом до API банків: метод decrypt_secret розшифровує токени перед запитами, test_connection перевіряє зв'язок із сервером, а sync запускає процес імпорту даних.

Клас Account оперує рахунками: метод sync_transactions замовляє виписки для нормалізації, get_balance розраховує залишок у копійках, а add_transaction створює операції із прив'язкою до рахунку.

Клас Transaction обробляє фінансові операції: метод apply_category призначає категорію за MCC-кодом або вручну, validate_integrity перевіряє SHA-256 хеш для запобігання дублюванню при CSV/XLSX-імпорті, а to_dict серіалізує об'єкт у словник для фронтенду.

Клас Goal контролює накопичення: метод calculate_progress вираховує відсоток досягнення мети, add_contribution створює новий внесок, а check_rules аналізує транзакції на відповідність критеріям автоналаштування.

Клас Category відповідає за аналітику: метод get_monthly_spending підраховує витрати за вибраний місяць, а get_budget_limit повертає встановлені обмеження бюджету.

2.7. Обґрунтування вибору програмно-апаратного забезпечення

При проєктуванні інформаційної системи «Finance Planner» вибір програмного та апаратного забезпечення зумовлений вимогами до швидкодії, безпеки обробки даних та мінімізації витрат на розгортання інфраструктури. Основна мета полягала у підборі такого технологічного стеку, який забезпечує стабільну роботу системи в умовах постійного обміну даними із зовнішніми банківськими сервісами.

Для реалізації серверної частини системи було обрано мову програмування Python. Основними чинниками цього вибору є високий рівень стандартизації мови для фінансових додатків, наявність розвинених бібліотек для роботи з базами даних та підтримка сучасних інструментів асинхронного програмування.

Як базовий вебфреймворк було обрано FastAPI. Вибір цієї технології замість класичних рішень обґрунтований такими архітектурними перевагами:

– Підтримка асинхронної архітектури: FastAPI працює на базі специфікації ASGI, що дозволяє ефективно обробляти велику кількість одночасних з'єднань. Для даного проєкту це є критично важливим, оскільки взаємодія з API Monobank потребує виконання мережевих запитів через інтернет. Асинхронність дозволяє серверу не блокувати інші процеси під час очікування відповіді від стороннього

банківського шлюзу.

– Автоматична валідація вхідних даних: Завдяки інтеграції з бібліотекою Pydantic, фреймворк забезпечує строгу типізацію. Це дозволяє автоматично перевіряти коректність структури JSON-пакетів від клієнта ще до їх передачі в логічні модулі сервера, що знижує ризик виникнення внутрішніх помилок.

Оскільки система оперує конфіденційними даними, особливу увагу приділено захисту банківських токенів доступу користувачів. Для реалізації криптографічного захисту обрано модуль Fernet з пакета Cryptography. Цей інструмент реалізує алгоритм симетричного шифрування, що дозволяє зберігати банківські ключі у базі даних виключно у вигляді шифротексту. Це унеможливлює використання токенів зловмисниками у випадку компрометації самої бази даних, оскільки ключ дешифрування зберігається ізольовано у середовищі сервера.

Для організації збереження інформації про користувачів, їхні рахунки та транзакції обрано реляційну систему керування базами даних MySQL. Вибір реляційної моделі зумовлений необхідністю чіткого структурування фінансових даних та забезпечення вимог транзакційності. MySQL обрано через її високу надійність під час виконання масових операцій запису та оновлення даних, а також завдяки ефективним механізмам індексації, які дозволяють оперативно формувати вибірки фінансової історії за певні проміжки часу.

Управління структурою бази даних та версіонування змін у таблицях реалізовано за допомогою інструменту міграцій Alembic. Він дозволяє автоматизувати процес оновлення схеми бази даних при перенесенні проєкту між різними середовищами розробки.

Локальне розгортання сервера розробки та СКБД MySQL здійснювалося на базі операційної системи Windows 11 за допомогою стандартного інструментарію командного рядка для керування віртуальним оточенням Python.

У другому розділі було проведено етап системного проєктування, результатом якого стала повна архітектурна модель майбутнього програмного продукту. Ми перейшли від теоретичних вимог до створення конкретних технічних рішень, що визначають структуру системи та логіку її роботи.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1 Архітектура програмного забезпечення та структура проєкту

Ієрархічну структуру каталогів та файлів розробленого програмного комплексу представлено на рисунку 3.1.

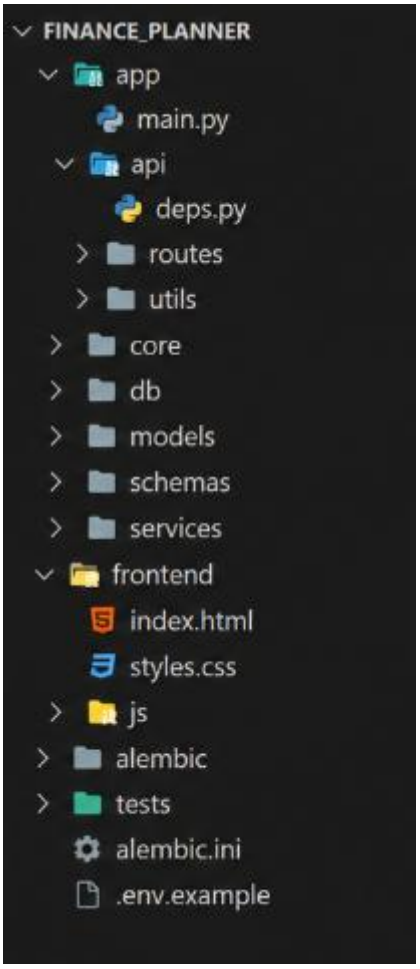

Рисунок 3.1 – Структурна схема розташування файлів та каталогів проєкту

Репозиторій Finance Planner організований як один веб-продукт без окремих папок backend/ і frontend/ на верхньому рівні. У корені лежать конфігурація (pyproject.toml, alembic.ini, .env.example), тести (tests/) і дві головні частини програми: app/ (сервер) та frontend/ (інтерфейс).

app/ – Python-пакет із точкою входу main.py. Тут збирається застосунок FastAPI, підключаються роутери, на старті виконуються міграції БД, монтується статика інтерфейсу на /ui/. Усередині app/ код розкладений за ролями:

- api/ – HTTP-шар: routes/ (ендпоінти), deps.py (сесія БД і користувач з JWT),
- utils/ – допоміжні функції, наприклад межі календарного місяця для бюджетів);
- services/ – бізнес-логіка без прив'язки до HTTP (Monobank, імпорт XLSX, категорії, цілі);
- models/ – ORM-моделі SQLAlchemy (таблиці users, accounts, transactions тощо);
- schemas/ – Pydantic-моделі запитів і відповідей API;
- core/ – конфіг з .env, безпека (паролі, JWT), шифрування токена банку;
- db/ – підключення до MySQL, Base, запуск Alembic.

Архітектура серверної частини

Сервер розроблено як багатошаровий REST API на базі фреймворку FastAPI. Запити обробляються послідовно зверху вниз через ізольовані архітектурні рівні:

1. Точка входу (app/main.py) – ініціалізує додаток і керує його життєвим циклом. При старті автоматично запускаються міграції бази даних (Alembic). Тут налаштовується CORS, реєструються роутери та підключається роздача статичних файлів інтерфейсу. Один процес Uvicorn одночасно обслуговує і API, і клієнтську частину.

2. Шар HTTP-маршрутизації (app/api/routes/) – містить контролери, розділені за функціональними доменами (автентифікація, рахунки, інтеграції, транзакції, аналітика, бюджети та цілі). Кожен роутер приймає HTTP-запит, валідує вхідні JSON-дані за допомогою Pydantic-схем, перевіряє права доступу користувача, викликає відповідний сервіс і повертає кінцевий JSON-результат із кодом статусу. Складна бізнес-логіка на цьому рівні відсутня.

3. Залежності запитів (app/api/deps.py) – керують контекстом виконання. Функція get_db відкриває одну сесію SQLAlchemy на кожен HTTP-запит і закриває її після видачі відповіді (з відкатом транзакції у разі помилки). Залежність

get_current_user декодує та перевіряє JWT-токен із заголовка Authorization, завантажуючи дані поточного користувача. Це ізолює дані та унеможливлює доступ до чужих записів.

4. Шар бізнес-логіки (app/services/) – містить чисті алгоритми обробки даних, які не залежать від вебфреймворку, що дозволяє тестувати їх окремо. До цього рівня входять модулі для взаємодії з Monobank API, синхронізації рахунків, парсингу файлів XLSX, автоматичної категоризації транзакцій за MCC-кодами та розрахунку прогресу фінансових цілей. Сервіси приймають сесію бази даних та ідентифікатор користувача як параметри.

5. Шар даних (app/models/, app/db/) – описує реляційну структуру таблиць MySQL за допомогою ORM-моделей. На цьому рівні зафіксовані зв'язки між сутностями, правила каскадного видалення та обмеження унікальності для забезпечення ідемпотентності імпорту. Підключення до бази даних конфігурується через окремий фабричний модуль, який зчитує змінні оточення.

6. Інфраструктурний шар (app/core/) – забезпечує системні функції додатку. Модуль конфігурації зчитує та валідує параметри з .env, модуль безпеки відповідає за хешування паролів та генерацію токенів, а криптографічний модуль реалізує симетричне шифрування Fernet для безпечного збереження банківських токенів доступу.

Архітектура клієнтської частини

Клієнтську частину реалізовано як односторінковий додаток (SPA, Single Page Application) без використання сторонніх компонентних фреймворків (React, Vue тощо). Система функціонує за принципом «тонкого клієнта»: основна аналітична та обчислювальна логіка, розрахунок бюджетів та перевірка правил накопичення перенесені на серверний рівень. Браузер виконує функції інтерфейсу відображення, збору даних із форм та надсилання команд. Взаємодія між компонентами інтерфейсу забезпечується перемиканням видимості секцій через маніпуляції з DOM та CSS-класами.

Інтерфейс користувача розділено на три базові рівні:

1. Рівень розмітки (index.html) – визначає статичну структуру сторінки. Глобальний стан екрана керується двома основними блоками: секцією авторизації (#auth-section), яка активується для неавторизованих відвідувачів, та робочою областю (#dashboard-section), доступною після перевірки прав доступу. Робочий простір містить модулі навігації (огляд, рахунки, бюджети, цілі, операції), форми введення, табличні компоненти та графічні елементи canvas для динамічної візуалізації фінансових метрик за допомогою бібліотеки Chart.js.

2. Рівень стилізації (styles.css) – відповідає за візуальне оформлення додатка. Стилі описують адаптивну сітку інтерфейсу, дизайн карток ключових показників ефективності (KPI), оформлення таблиць транзакцій, поведінку прихованих бічних панелей та відображення станів помилок валідації. Стилізація побудована на базі чистого CSS без застосування технологій CSS-in-JS.

3. Рівень логіки інтерфейсу (js/) – реалізований за допомогою модулів JavaScript (ES Modules). Архітектура коду на клієнтській стороні дублює доменну структуру сервера, розподіляючи логіку за функціональними компонентами (керування сесіями, обробка рахунків, операції імпорту, відображення графіків аналітики). Модулі відповідають за асинхронний обмін даними з сервером через Fetch API, локальну валідацію полів введення та динамічне оновлення елементів сторінки на основі отриманих JSON-відповідей.

3.2. Реалізація структури бази даних та моделей даних

Проєктування архітектури даних інформаційної системи «Finance Planner» базується на використанні реляційної моделі даних та СКБД MySQL. Взаємодія з базою даних реалізована через технологію об'єктно-реляційного відображення (ORM) SQLAlchemy. Цей підхід дозволяє перетворити структуру таблиць на класи мови Python, що забезпечує високу надійність транзакцій та спрощує підтримку коду (повний код створення бази даних наведено в додатку А).

Архітектура моделей та базовий шар

В основу всіх сутностей системи покладено базовий клас Base, успадкований від DeclarativeBase. Він є спільним для всіх моделей і дозволяє SQLAlchemy автоматично реєструвати кожну таблицю в метаданих системи.

```
from sqlalchemy.orm import DeclarativeBase
class Base(DeclarativeBase):
    pass
```

Кожна сутність предметної області – користувач, рахунок, транзакція – описується окремим класом, що наслідує Base. Це дозволяє керувати структурою бази даних безпосередньо з коду, забезпечуючи типізацію та автоматичну валідацію.

Реалізація сутності користувача

Модель користувача (User) у файлі app/models/user.py відображає таблицю users. Вона містить мінімально необхідний набір полів для автентифікації та ідентифікації:

```
        id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
        email: Mapped[str] = mapped_column(String(255), unique=True, index=True,
nullable=False)
        password_hash: Mapped[str] = mapped_column(String(255), nullable=False)
        created_at: Mapped[datetime] = mapped_column(DateTime(timezone=True),
server_default=func.now())
```

Використання індексу (index=True) для поля email суттєво прискорює процес пошуку запису під час входу користувача в систему, що є критично важливим при великій кількості зареєстрованих акаунтів.

Організація зв'язків та цілісність даних

Система транзакцій вимагає високої узгодженості даних. У моделі Transaction реалізовані зовнішні ключі (ForeignKey), які пов’язують фінансові операції з конкретним користувачем та рахунком.

```
class Transaction(Base):
    __tablename__ = "transactions"
        id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)
        user_id: Mapped[int] = mapped_column(ForeignKey("users.id",
ondelete="CASCADE"), index=True)
```

```
        account_id:    Mapped[int]    =    mapped_column(ForeignKey("accounts.id",
ondelete="CASCADE"), index=True)
        amount_minor: Mapped[int] = mapped_column(Integer, nullable=False)
        occurred_at:  Mapped[datetime]  =  mapped_column(DateTime(timezone=True),
index=True)
```

Застосування ondelete="CASCADE" забезпечує автоматичну очистку бази даних: при видаленні користувача система автоматично видаляє всі його транзакції. Поле amount_minor зберігає суму в копійках цілим числом, що є стандартною практикою для фінансових систем, оскільки це унеможливлює похибки округлення, притаманні типам з плаваючою комою.

Механізм міграцій та сесій

Для версіонування схеми бази даних використовується інструмент Alembic. Кожна зміна в коді моделей автоматично трансформується у скрипт міграції, що дозволяє розгорнути структуру бази даних на будь-якому середовищі.

Робота з даними в реальному часі виконується через фабрику сесій SessionLocal. Кожен HTTP-запит отримує окремий контекст транзакції:

```
engine = create_engine(settings.database_url, pool_pre_ping=True)
SessionLocal = sessionmaker(bind=engine, autoflush=False, autocommit=False)
```

При виконанні операцій запису код використовує стандартний життєвий цикл: об’єкт моделі додається до сесії (db.add), після чого виклик db.commit() ініціює запис у MySQL. Якщо на етапі запису виникає помилка, транзакція автоматично скасовується (rollback), що запобігає пошкодженню даних або появі «сміттєвих» записів.

Таким чином, реалізована структура бази даних забезпечує не лише надійне зберігання інформації, а й ефективну підтримку складних аналітичних запитів, необхідних для роботи модуля візуалізації фінансових показників.

3.3 Програмна реалізація модулів автентифікації та безпеки даних

Реалізація модуля безпеки базується на принципах багаторівневого захисту конфіденційних даних та платіжних токенів користувачів Для відокремлення

конфігураційних секретів від програмного коду використано бібліотеку pydantic-settings, яка зчитує чутливі змінні середовища безпосередньо з ізольованого файлу .env за допомогою класу Settings:

```
class Settings(BaseSettings):
    model_config = SettingsConfigDict(env_file=".env",env_file_encoding="utf-8", extra="ignore")
    db_host: str
    db_password: str
    jwt_secret: str
    jwt_alg: str = "HS256"
    mono_token_enc_key: str # Ключ для шифрування токенів Monobank
```

Обробка та верифікація паролів здійснюється в модулі app/core/security.py за допомогою адаптивного алгоритму хешування pbkdf2_sha256 бібліотеки passlib, що унеможливлює відновлення оригінальних символів навіть у разі повного витоку дампу бази даних:

```
pwd_context=CryptContext(schemes=["pbkdf2_sha256","bcrypt"],deprecated="auto")
def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(password: str, password_hash: str) -> bool:
    return pwd_context.verify(password, password_hash)
```

Для авторизації клієнтських запитів без повторного введення пароля впроваджено безсесійний протокол JWT. Функція create_access_token генерує підписаний токен із обмеженим терміном дії, що містить ідентифікатор суб'єкта у корисній структурі даних payload:

```
def create_access_token(*, subject: str) -> str:
    now = datetime.now(tz=timezone.utc)
    exp = now + timedelta(minutes=settings.jwt_access_ttl_min)
    payload = {"sub": subject, "iat": int(now.timestamp()), "exp": int(exp.timestamp())}
    return jwt.encode(payload, settings.jwt_secret, algorithm=settings.jwt_alg)
```

Система перевіряє exp (час дії) та sub (ідентифікатор користувача), що дозволяє анулювати доступ без зміни пароля, просто видаливши токен на стороні клієнта.

Захист банківських інтеграцій за допомогою Fernet

Найбільш вразливим елементом системи є токени доступу до API Monobank. Для їх захисту в модулі app/core/crypto.py реалізовано симетричне шифрування Fernet.

Шифрування відбувається перед записом даних у базу:

```
def encrypt_text(value: str) -> str:
    return _fernet().encrypt(value.encode("utf-8")).decode("utf-8")

def decrypt_text(value_enc: str) -> str:
    try:
        data = _fernet().decrypt(value_enc.encode("utf-8"))
        return data.decode("utf-8")
    except InvalidToken:
        raise ValueError("Invalid encrypted value")
```

Таким чином, у базі даних зберігається не сам токен, а його зашифрована копія (secret_enc). При виконанні запиту до банківського API система зчитує цей рядок, розшифровує його виключно в оперативній пам'яті та відправляє запит.

Реалізація клієнтської безпеки

На стороні фронтенду збережений у localStorage токен автоматично додається до кожного HTTP-запиту в заголовок Authorization: Bearer за допомогою модульної функції apiFetch:

```
export async function apiFetch(path, options = {}) {
    const token = getToken();
    const headers = new Headers(options.headers || {});
    if (token) headers.set("Authorization", `Bearer ${token}`);
    return await fetch(`${apiBase()}${path}`, { ...options, headers });
}
```

Результат роботи алгоритмів шихування Fernet та збереження паролів користувачів у захищеному вигляді всередині СКБД продемонстровано на рис. 3.2.

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	user_id	provider	secret_enc	created_at	updated_at
4	8	monobank	gAAAAABqJPa4u443LVUppjvQaUD8LHcuzdXvN...	2026-06-07 07:42:32	2026-06-07 07:42:32
5	5	monobank	gAAAAABqJjLAtV70SN6kDbA3zzflRCnqWFnkLa...	2026-06-08 06:10:56	2026-06-08 06:10:56
6	7	monobank	gAAAAABqJjNQ4vuB7bYRFBXJqbSrKoUXPnetse...	2026-06-08 06:11:24	2026-06-08 06:13:20
7	6	monobank	gAAAAABqJjMxpWiQGgldzdJq3pbkNx1RETGlmp...	2026-06-08 06:12:49	2026-06-08 06:12:49
NULL	NULL	NULL	NULL	NULL	NULL

Рисунок 3.2 – Хеш зашифрованих API токенів

Комплексна реалізація цих модулів – від хешування паролів до шифрування токенів та перевірки JWT на кожному запиті – створює надійний захисний бар'єр, який мінімізує ризики витоку персональних та фінансових даних користувачів системи.

3.4 Реалізація модуля інтеграції з банківськими сервісами

Модуль інтеграції забезпечує автоматизацію обліку фінансових операцій шляхом уніфікації вхідних даних. Незалежно від джерела отримання інформації – безпосередньо через Monobank Open API чи за допомогою імпорту локальних файлів виписок (CSV, XLSX) – система трансформує відомості у єдиний формат для збереження в таблицях accounts та transactions (повний код імпорту даних наведено в додатку Б). Розподіл джерел виконується за допомогою констант, визначених у модулі app/core/banking.py:

```
PROVIDER_MONOBANK = "monobank"
PROVIDER_PRIVAT24 = "privat24"  # зарезервовано, API ще немає
TX_SOURCE_MONOBANK = "monobank"
TX_SOURCE_CSV = "csv_import"
```

Користувацький запит на отримання каталогу доступних способів підключень обробляється за допомогою HTTP-ендпоінта GET /integrations/catalog. Роутер перевіряє наявність активних записів у таблиці bank_integrations та повертає статус підключень для клієнтської частини :

```
@router.get("/catalog")
def integrations_catalog(current_user: User = Depends(get_current_user), db:
Session = Depends(get_db)):
    mono = db.scalar(
        select(BankIntegration).where(
            BankIntegration.user_id == current_user.id,
            BankIntegration.provider == PROVIDER_MONOBANK,
        )
    )
    return {
        "banks": [
            {"id": PROVIDER_MONOBANK, "label": "Monobank", "flow": "token",
"connected": mono is not None, ...},
            {"id": "csv", "label": "Інший банк (CSV)", "flow": "csv", ...},
            {"id": PROVIDER_PRIVAT24, "label": "ПриватБанк", "flow": "soon",
"connected": False, ...},
```

```
            ]
        }
```

Клієнт Monobank API – це тонкий шар над httpx у mono_client.py. Базова адреса – https://api.monobank.ua. Токен передається в заголовку X-Token, як вимагає документація Monobank.

Перевірка токена і отримання списку карток:

```
async def mono_get_client_info(token: str) -> MonoClientInfo:
    async with httpx.AsyncClient(base_url=MONO_BASE_URL, timeout=20.0) as client:
        resp = await client.get("/personal/client-info", headers={"X-Token": token})
    if resp.status_code != 200:
        raise MonoApiError(f"mono client-info failed: {resp.status_code} {resp.text}")
    return MonoClientInfo(raw=resp.json())
```

Виписка за період (unix-час from_ts … to_ts):

```
async def mono_get_statement(token: str, *, account_id: str, from_ts: int, to_ts: int) -> list[MonoStatementItem]:
    async with httpx.AsyncClient(base_url=MONO_BASE_URL, timeout=30.0) as client:
        resp = await client.get(
            f"/personal/statement/{account_id}/{from_ts}/{to_ts}",
            headers={"X-Token": token},
        )
    if resp.status_code != 200:
        raise MonoApiError(f"mono statement failed: {resp.status_code} {resp.text}")
    data = resp.json()
    return [MonoStatementItem(raw=item) for item in data]
```

Модуль не знає про MySQL – він лише ходить в Monobank і повертає JSON у вигляді Python-об’єктів. Помилки API обгортаються в MonoApiError, щоб вище по стеку показати користувачу зрозуміле повідомлення.

Підключення Monobank реалізовано в app/api/routes/mono.py. Користувач має бути залогінений (get_current_user). Він надсилає токен у тілі запиту; сервер спочатку перевіряє його через mono_get_client_info, потім шифрує і зберігає в bank_integrations:

```
@router.post("/connect", response_model=MonoConnectOut)
async def connect(payload: MonoConnectIn, current_user: User = Depends(get_current_user), db: Session = Depends(get_db)):
    await mono_get_client_info(payload.token)
    token_enc = encrypt_text(payload.token)
    existing = db.scalar(select(BankIntegration).where(...))
    if existing:
```

```
            existing.secret_enc = token_enc
        else:
            db.add(BankIntegration(user_id=current_user.id,
provider=PROVIDER_MONOBANK, secret_enc=token_enc))
        db.commit()
```

Якщо токен невалідний – 400 до того, як щось потрапить у базу. Якщо валідний – у таблиці з'являється або оновлюється один рядок інтеграції на користувача.

Синхронізація викликається кнопкою «Синхронізувати» на фронті (POST /integrations/monobank/sync). Роут задає період за замовчуванням – останні 30 днів – і викликає головну функцію імпорту:

```
    @router.post("/sync")
    async def sync(from_ts: int | None = Query(None), to_ts: int | None =
Query(None), current_user: User = Depends(get_current_user), db: Session =
Depends(get_db)):
        if to_ts is None:
            to_ts = int(datetime.now(tz=timezone.utc).timestamp())
        if from_ts is None:
            from_ts = int((datetime.now(tz=timezone.utc) -
timedelta(days=30)).timestamp())
        result = await monobank_sync(db, user_id=current_user.id, from_ts=from_ts,
to_ts=to_ts)
        rules_applied = apply_auto_rules_for_user(db, user_id=current_user.id)
        return {**result, "goal_contributions_from_rules": rules_applied}
```

Функція monobank_sync розшифровує токен, ітерує отриманий список рахунків та зіставляє кожну операцію з ORM-моделлю Transaction . Для видаткових операцій викликається підсистема автоматичного визначення внутрішньої категорії за кодами MCC. Ідемпотентність повторної синхронізації забезпечується перехопленням виключення IntegrityError завдяки унікальному складеному ключу:

```
    for item in statement:
        raw = item.raw
        ext_id = raw.get("id")
        occurred_at = datetime.fromtimestamp(int(raw.get("time", 0)),
tz=timezone.utc)
        amount_minor = int(raw.get("amount", 0))  # у Monobank вже в копійках
        description = raw.get("description")
        mcc_val = int(mcc_raw) if mcc_raw is not None else None
        if amount_minor < 0:
            cat_id = resolve_category_id_for_mcc(db, user_id=user_id,
mcc=mcc_val)
        else:
            cat_id = None  # доходи без автокатегорії
        tx = Transaction(
            user_id=user_id,
```

```
            account_id=account_id,
            amount_minor=amount_minor,
            occurred_at=occurred_at,
            description=...,
            merchant=...,
            mcc=mcc_val,
            category_id=cat_id,
            external_source=MONO_SOURCE,
            external_id=str(ext_id),
        )
```

Візуальне відображення каталогу підключень та успішний результат виконання онлайн-синхронізації рахунків Monobank через користувацький інтерфейс представлено на рис. 3.3.

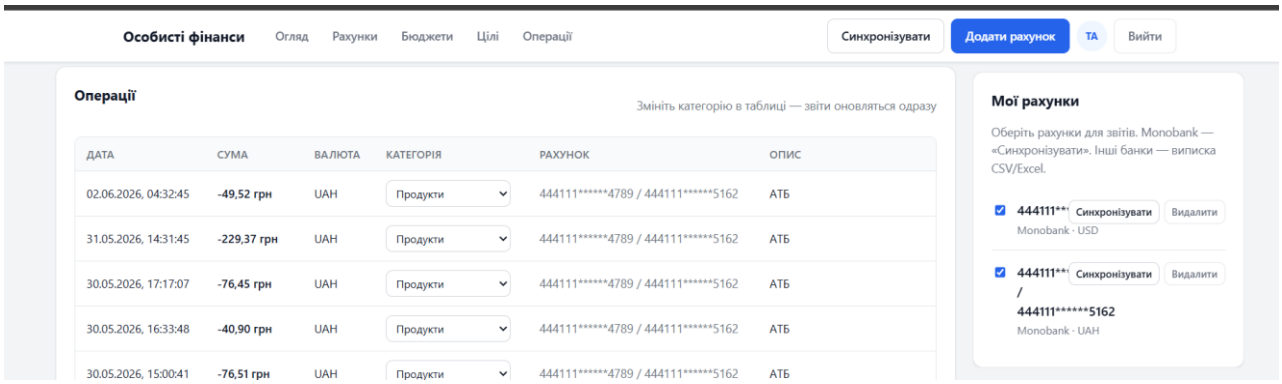

Рисунок 3.3 – Синхронізація рахунку monobank

Другий архітектурний шлях отримання даних – офлайн-імпорт виписок у форматах CSV/XLSX розміром до 10 МБ через маршрут POST /csv/import. Сервіс зчитує байти файлу, визначає кодування та автоматично підбирає розділювачі за допомогою csv.Sniffer :

```
def _rows_from_csv_bytes(file_bytes: bytes) -> list[list[str]]:
    text = _decode_bytes(file_bytes)
    dialect = csv.Sniffer().sniff(sample, delimiters=",;\t")
    reader = csv.reader(io.StringIO(text), dialect)
    return [[_cell_to_str(c) for c in row] for row in reader if ...]
```

Для обробки документів різних банків реалізовано пошук рядка заголовків та маппінг колонок за допомогою множин мовних псевдонімів (DATE_ALIASES, AMOUNT_ALIASES тощо).

Модуль _parse_amount конвертує текстові суми у копійки, очищуючи знаки валют та перетворюючи розділювачі.

Оскільки локальні файли виписок не завжди містять постійні унікальні ID транзакцій, унікальний ідентифікатор external_id для захисту від дублювання генерується штучно як SHA-256 хеш від комбінації стабільних параметрів рядка :

```
        for line_no, cells in enumerate(rows[start:], start=start + 1):
            occurred_at = _parse_datetime(cells[date_i])
            amount_minor   =   _amount_from_row(cells,   amount_i=...,   debit_i=...,
credit_i=...)
            cat_txt = cells[cat_i] if cat_i else None
            opis = cells[desc_i] if desc_i else None
            desc  =  _build_description(bank_display=eff_bank,  category_text=cat_txt,
opis=opis)
            cat_id     =     resolve_category_for_statement(db,     user_id=user_id,
category_text=cat_txt, amount_minor=amount_minor, mcc=mcc_val)
```

Категоризація операцій з файлів здійснюється в модулі statement_categories.py шляхом пошуку ключових слів у текстовому описі транзакцій.

Поведінку інтерфейсу системи під час завантаження зовнішнього файлу банківської виписки та відображення імпортованих даних наведено на рис. 3.4.

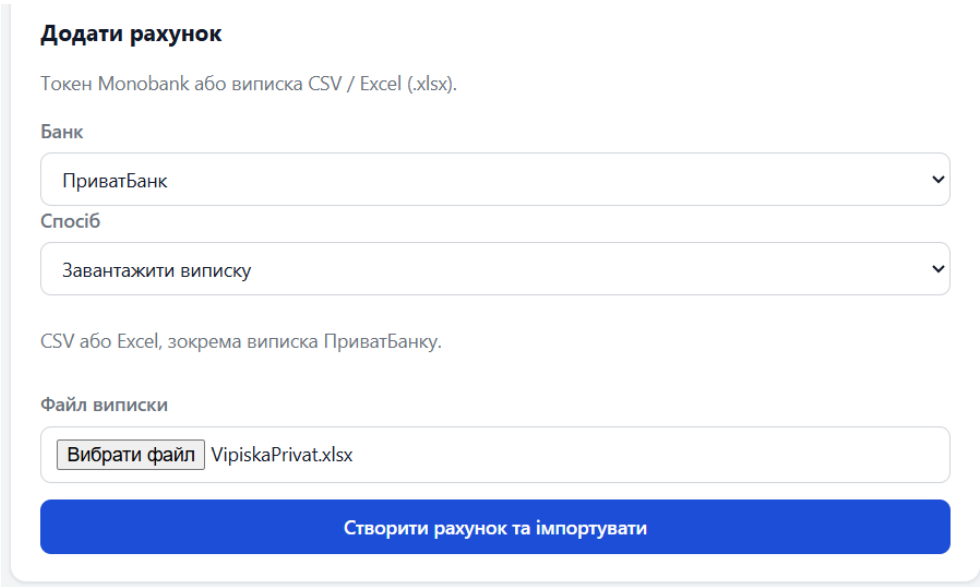

Рисунок 3.4 – Інтерфейс завантаження виписки

Таким чином, розроблений комбінований архітектурний підхід дозволив повністю автоматизувати процеси збору інформації.

3.5 Програмна реалізація блоку візуалізації фінансових даних

Блок візуалізації фінансових даних забезпечує інтерактивне відображення аналітичної інформації на головному дашборді користувача у розділі «Огляд». Для побудови динамічних графіків на стороні клієнта використовується бібліотека Chart.js, яка підключається через CDN-шлюз у файлі index.html (код візуалізації наведено в додатку Б). Розмітка полотен для графіків – у frontend/index.html, стилі висоти – у frontend/styles.css.

Chart.js підключається в index.html з CDN:

```
    <script
src="https://cdn.jsdelivr.net/npm/chart.js@4.4.1/dist/chart.umd.min.js"
defer></script>
```

На сторінці два елементи <canvas> – саме вони стають «полотном» для графіків:

```
<article class="panel chart-panel">
  <h2 class="panel-title">Рух коштів по днях</h2>
  <div class="chart-canvas-holder">
    <canvas id="chart-daily"></canvas>
  </div>
</article>
<article class="panel chart-panel">
  <h2 class="panel-title">Витрати за категоріями</h2>
  <div class="chart-canvas-holder chart-canvas-holder--mcc">
    <canvas id="chart-by-category"></canvas>
  </div>
</article>
```

Спочатку браузер визначає період і рахунки, за якими будувати звіт. У frontend/js/core/period.js:

```
export function getPeriodRange() {
  const days = parseInt(String(periodSelect?.value || "30"), 10) || 30;
  const to = new Date();
  const from = new Date(to.getTime() - days * 86400000);
  return { from, to };
}
```

```
export function accountIdsQueryParams() {
  return [...state.selectedAccountIds.values()].map((id) => ["account_ids",
String(id)]);
}
```

Користувач обирає 7, 30 або 90 днів у <select id="period-days">. Якщо в розділі рахунків відмічені не всі картки – у запит додаються лише account_ids обраних.

На сервері app/api/routes/analytics.py з таблиці transactions рахуються суми за фільтром користувача, періоду й рахунків. Дохід – сума додатних amount_minor, витрати – сума модулів від'ємних, сальдо – чиста сума всіх операцій:

```
base = [
    Transaction.user_id == uid,
    Transaction.occurred_at >= from_dt,
    Transaction.occurred_at <= to_dt,
]
if account_ids:
    base.append(Transaction.account_id.in_(account_ids))
income_minor = int(db.scalar(
    select(func.coalesce(func.sum(Transaction.amount_minor), 0)).where(
        *base, Transaction.amount_minor > 0,
    )
) or 0)
expense_minor = int(db.scalar(
    select(func.coalesce(func.sum(-Transaction.amount_minor), 0)).where(
        *base, Transaction.amount_minor < 0,
    )
) or 0)
net_minor = int(db.scalar(
    select(func.coalesce(func.sum(Transaction.amount_minor), 0)).where(*base)
) or 0)
```

Візуалізація сум рахунків за вказаний період зображено на рисунку 3.5

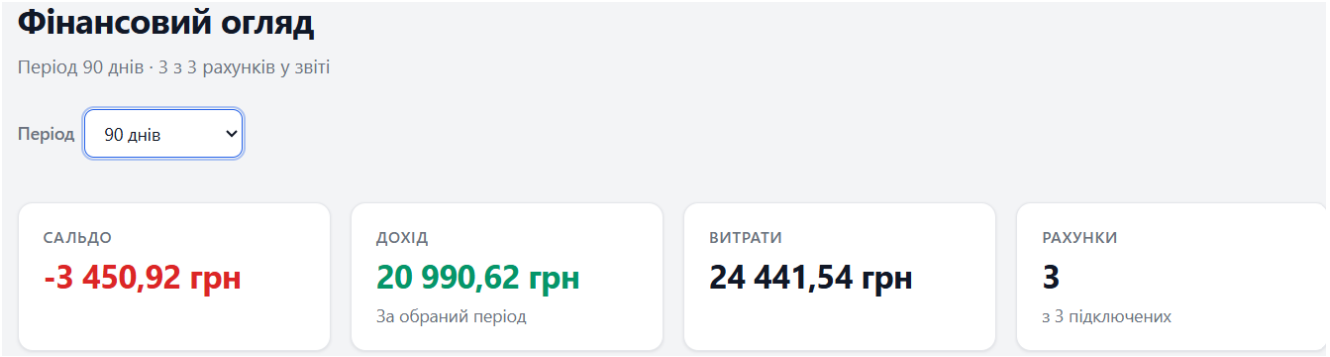

Рисунок 3.5 – Візуалізація сум рахунків за вказаний період

Дані для лінійного графіка – групування по календарному дню: для кожного дня сума всіх amount_minor (сальдо за день, не лише витрати):

```
day_col = cast(Transaction.occurred_at, Date)
daily_rows = db.execute(
    select(day_col, func.sum(Transaction.amount_minor))
    .where(*base)
    .group_by(day_col)
    .order_by(day_col)
).all()
daily = [DailyPointOut(day=r[0], net_minor=int(r[1])) for r in daily_rows]
```

Дані для кругової діаграми – витрати згруповані за назвою категорії (якщо категорії немає – «Без категорії»), топ-12:

```
cat_label = func.coalesce(Category.name, "Без категорії")
category_rows = db.execute(
    select(cat_label, func.sum(-Transaction.amount_minor))
    .select_from(Transaction)
    .outerjoin(Category, Category.id == Transaction.category_id)
    .where(*base, Transaction.amount_minor < 0)
    .group_by(cat_label)
    .order_by(func.sum(-Transaction.amount_minor).desc())
    .limit(12)
).all()
category_expenses = [
    CategoryExpenseOut(category_name=str(r[0]), expense_minor=int(r[1])) for
r in category_rows
]
```

Код лінійного графіка – у dashboard.js, після перевірки typeof Chart !== "undefined". Спочатку знищуються попередні екземпляри (destroyCharts), щоб не було витоку пам’яті при зміні періоду:

```
export function destroyCharts() {
  if (state.chartDaily) {
    state.chartDaily.destroy();
    state.chartDaily = null;
  }
  if (state.chartCategory) {
    state.chartCategory.destroy();
    state.chartCategory = null;
  }
}
```

Потім беруться canvas і дані з data.daily:

```
const ctxD = document.getElementById("chart-daily");
const daily = data.daily || [];
const labelsD = daily.map((d) => d.day);
const valsD = daily.map((d) => d.net_minor / 100);
state.chartDaily = new Chart(ctxD, {
  type: "line",
```

```
      data: {
        labels: labelsD.length ? labelsD : ["–"],
        datasets: [{
          label: "Сальдо за день",
          data: labelsD.length ? valsD : [0],
          borderColor: "#2563eb",
          backgroundColor: "rgba(37, 99, 235, 0.08)",
          fill: true,
          tension: 0.25,
        }],
      },
      options: {
        responsive: true,
        maintainAspectRatio: false,
        plugins: { legend: { display: false } },
        scales: {
          x: { ticks: { maxRotation: 45 } },
          y: { beginAtZero: false },
        },
      },
    });
```

Це лінійний графік (line chart) Chart.js: вісь X – дні, вісь Y – сальдо за день у гривнях (net_minor / 100). Заливка під лінією (fill: true), згладжування (tension: 0.25). Якщо за період немає операцій – одна мітка «–» і нуль. Приклад візуалізації лінійного графіка зображенно на рисунку 3.6

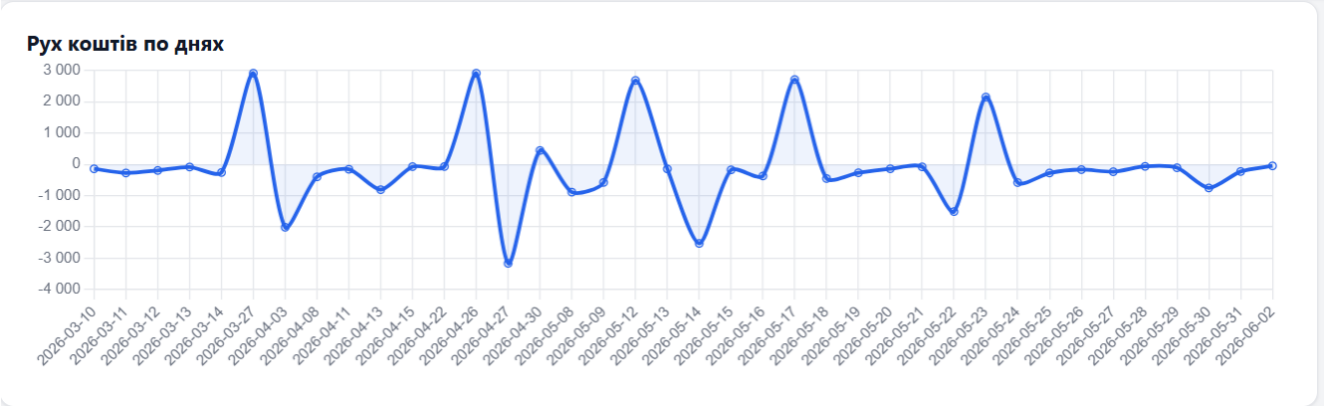

Рисунок 3.6 – Візуалізація лінійного графіка

Код кругової діаграми – той самий файл, canvas chart-by-category, тип doughnut (кільце з вирізом посередині):

```
    const ctxM = document.getElementById("chart-by-category");
    const byCat = data.category_expenses || [];
    const colors = ["#3d8bfd", "#34d399", "#fbbf24", "#a78bfa", "#f87171", ...];
    if (!byCat.length) {
      state.chartCategory = new Chart(ctxM, {
        type: "doughnut",
        data: { labels: ["Немає витрат за період"], datasets: [{ data: [1],
backgroundColor: ["#e5e7eb"] }] },
```

```
      options: {
        responsive: true,
        maintainAspectRatio: false,
        cutout: "58%",
        plugins: { legend: catLegend },
      },
    });
  } else {
    state.chartCategory = new Chart(ctxM, {
      type: "doughnut",
      data: {
        labels: byCat.map((x) => escapeHtml(x.category_name)),
        datasets: [{
          data: byCat.map((x) => x.expense_minor / 100),
          backgroundColor: colors.slice(0, byCat.length),
        }],
      },
      options: {
        responsive: true,
        maintainAspectRatio: false,
        cutout: "52%",
        plugins: { legend: catLegend },
      },
```

Кожен сектор – одна категорія витрат; розмір сектора пропорційний expense_minor. Легенда знизу (position: "bottom"). Якщо витрат немає – сірий «заглушковий» круг з підписом «Немає витрат за період». Назви категорій проходять через escapeHtml, щоб спецсимволи в HTML не зламали розмітку.

Приклад візуалізації кругової діаграми зображенно на рисунку 3.7

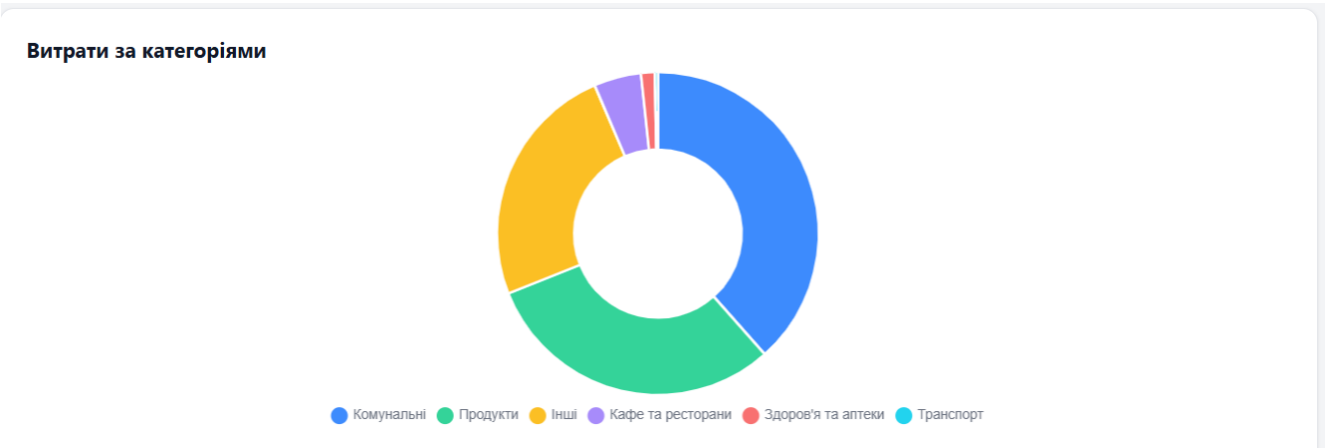

Рисунок 3.7 – Візуалізація кругової діаграми

Оновлення візуалізації відбувається при вході (session.js викликає loadDashboard), при зміні періоду і при натисканні «Синхронізувати»:

```
periodSelect?.addEventListener("change", async () => {
  updateDashboardMeta();
  await loadDashboard();
  const { loadTransactions } = await import("./transactions.js");
  await loadTransactions();
});
```

При виході з акаунта в auth.js викликається destroyCharts(), щоб прибрати графіки з DOM. Весь код наведений в додатках А, Б, В.

3.6 Організація тестування функціональних модулів системи

Автоматизоване тестування розробленої інформаційної системи реалізовано на базі фреймворку pytest з розподілом тестових сценаріїв на ізольовані модульні (unit) та комплексні (інтеграційні API) перевірки. Для ізоляції середовища та захисту продуктової бази даних MySQL, конфігурація оточення винесена у файл tests/conftest.py. Замість фізичного сервера під час тестів розгортається реляційна СУБД SQLite в оперативній пам'яті (:memory:), схема таблиць в якій автоматично (повний код тестування наведено в додатку В)

```
import os
import pytest
from sqlalchemy import create_engine
from sqlalchemy.orm import Session
from sqlalchemy.pool import StaticPool
from app.database import Base
os.environ.setdefault("JWT_SECRET",      "unit-test-jwt-secret-do-not-use-in-
prod")
os.environ.setdefault("MONO_TOKEN_ENC_KEY",
"bV9ZajV3bXF6ZGNidWloeXBnY3Z4bXp3cWthc2RjYmU=")

@pytest.fixture(name="db", scope="function")
def db_fixture():
    engine               =               create_engine("sqlite:///:memory:",
connect_args={"check_same_thread": False}, poolclass=StaticPool)
    Base.metadata.create_all(engine)
    with Session(engine) as session:
        yield sessionos
```

Перевірка ізольованої бізнес-логіки обробки транзакцій та банківських виписок без залучення HTTP-рівня реалізована в модулі tests/unit/ test_mono_sync.py. Тестовий сценарій test_monobank_sync_idempotency валідує алгоритм дедуплікації даних: функція імітує отримання сирих транзакцій від шлюзу Monobank API, здійснює послідовний подвійний запис тієї самої сутності в

ізольовану сесію та контролює, щоб повторна операція ігнорувалася завдяки перехопленню виключень унікальності:

```
import pytest
from sqlalchemy import select
from app.models.transaction import Transaction
from app.services.banking.sync import store_transactions
from app.schemas.banking import MonoStatementItem
@pytest.mark.asyncio
async def test_monobank_sync_idempotency(db):
    user_id, account_id = 1, 10
    stmt_item = MonoStatementItem(raw={
        "id": "tx_123_unique_id",
        "time": 1700000000,
        "amount": -15000,
        "mcc": 5411,
        "description": "АТБ-Маркет"
    })
    await store_transactions(db, user_id=user_id, account_id=account_id,
items=[stmt_item])
    await store_transactions(db, user_id=user_id, account_id=account_id,
items=[stmt_item])
    txs = db.scalars(select(Transaction).where(Transaction.account_id ==
account_id)).all()
    assert len(txs) == 1
```

Тестування накопичувальних підсистем та правил автоматичного розподілу коштів по цілях реалізовано всередині модуля tests/unit/test_goals.py. Створений модульний тест test_apply_auto_rules_income перевіряє точність тригерів: він ініціалізує сутність фінансової цілі, створює для неї правило відрахування у розмірі 10% від доходів, що містять у текстовому описі ключове слово "Premium", та ініціює обробку транзакції, контролюючи коректність розрахунку фінальної суми внеску:

```
from sqlalchemy import select
from app.models.goals import Goal, GoalRule, GoalContribution
from app.models.transaction import Transaction
from app.services.goals.rules import apply_rules_for_transaction
def test_apply_auto_rules_income(db):
    user_id = 1
    goal = Goal(user_id=user_id, name="Резерв", target_amount_minor=100000,
mode="from_transactions")
    db.add(goal); db.flush()
    rule = GoalRule(goal_id=goal.id, description_contains="Premium",
amount_side="income", percent=10)
```

Комплексний сценарій test_auth_flow повністю імітує поведінку користувача: надсилає POST-запити на реєстрацію облікового запису, перевіряє роботу валідатора для виявлення дублікатів електронної пошти та здійснює

контроль доступу до захищеного ендпоінта профілю /auth/me, перевіряючи повернення статус-коду неавторизованого запиту 401 Unauthorized у разі відсутності авторизаційного токена:

```
    from fastapi.testclient import TestClient
    from app.main import app
    client = TestClient(app)
    def test_auth_flow():
        reg_resp = client.post("/api/v1/auth/register", json={"email":
"test_user@example.com", "password": "securepassword123"})
        assert reg_resp.status_code == 201
        dup_resp = client.post("/api/v1/auth/register", json={"email":
"test_user@example.com", "password": "different_password"})
        assert dup_resp.status_code == 400
        me_resp = client.get("/api/v1/auth/me")
        assert me_resp.status_code == 401
```

Комплексний запуск створеної сукупності тестів дозволяє гарантувати високу стабільність взаємодії між архітектурними рівнями додатка, повну ізоляцію даних окремих користувачів та коректність обробки помилок. Результат успішного виконання всієї сукупності розроблених модульних та інтеграційних тестів у консолі середовища розробки з виведенням фінальних статусів перевірок представлено на рисунку 3.8.

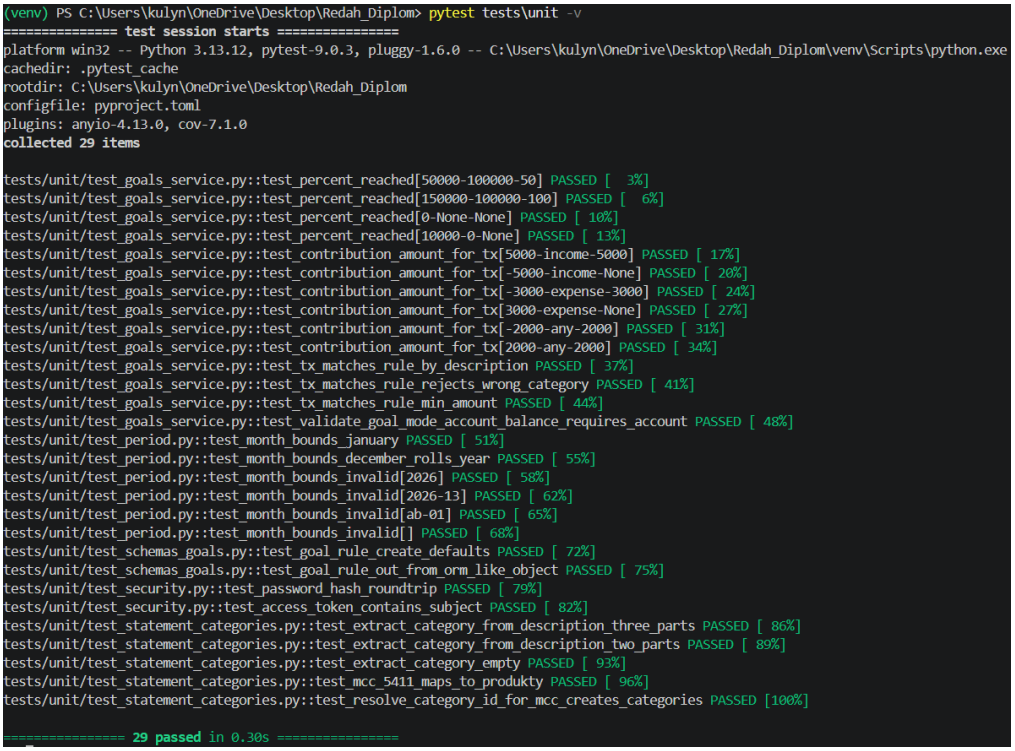

Рисунок 3.8 – Результат виконання тестів

Висновок до розділу

Практична реалізація підтвердила коректність обраних алгоритмів інтеграції з банківськими API та модулів парсингу файлів. Створений програмний комплекс забезпечує автоматизовану категоризацію операцій за MCC-кодами та синхронізацію з бюджетними планами, що мінімізує ручну працю користувача.

ВИСНОВКИ

У межах виконаної кваліфікаційної роботи успішно здійснено повний цикл проєктування, алгоритмізації, програмної реалізації та верифікації інформаційної системи для автоматизованого управління особистими фінансами «Finance Planner». Актуальність проведеного дослідження зумовлена необхідністю систематизації фінансових потоків користувача в умовах багатоджерельності банківських даних та потребою в інструментах аналітики, що дозволяють мінімізувати суб'єктивні помилки при веденні бюджету. Створений програмний комплекс вирішує важливу науково-практичну задачу агрегації транзакційних даних, їх нормалізації та автоматичної категоризації, надаючи користувачу інтерактивні інструменти візуалізації фінансового стану та планування цілей.

Під час виконання першого розділу роботи проведено детальний системний аналіз предметної області, досліджено специфіку функціонування банківських API та вивчено наявні світові технологічні аналоги у сфері персонального фінансового менеджменту (PFM). Обґрунтовано доцільність застосування клієнт-серверної архітектури з використанням асинхронних протоколів для забезпечення високої швидкості обробки запитів. На основі проведеного аналізу сформовано концептуальний стек технологій розробки, де ключовий акцент зміщено на використання сучасного асинхронного веб-фреймворку FastAPI, реляційного сховища даних PostgreSQL з використанням ORM SQLAlchemy та безпечних механізмів автентифікації.

У межах другого розділу розроблено архітектурний проєкт інформаційної системи, що базується на принципах цілісності та безпеки персональних даних. Побудовано концептуальні та логічні моделі даних, детально задокументовано атрибутивний склад таблиць транзакцій, рахунків і категорій, а також специфіковано алгоритмічне забезпечення для математичної обробки сальдо та бюджетних показників. Особливу увагу приділено теоретичному обґрунтуванню моделей транзакційного захисту, що включають безсесійний Stateless-доступ на базі JWT-токенів та безпечне шифрування банківських ключів доступу (API-

токенів) із використанням алгоритму Fernet, що гарантує захист конфіденційної інформації користувача навіть у разі порушення цілісності сховища.

Практична реалізація програмного комплексу, описана в третьому розділі, підтвердила високу ефективність обраних рішень та забезпечила створення повнофункціонального веб-додатка. Розроблено два ключові контури обробки даних: інтеграційний модуль для синхронізації з Monobank API з реалізацією механізмів для запобігання дублюванню транзакцій та модуль парсингу неструктурованих файлових виписок (CSV/XLSX), який використовує детерміноване хешування для ідентифікації записів. Впроваджені алгоритми автоматичної категоризації на базі аналізу MCC-кодів та семантичних ключів опису операцій дозволили значно підвищити якість аналітичних звітів, мінімізуючи частку некласифікованих транзакцій.

Комплексна верифікація та тестування розробленого програмного забезпечення експериментально довели повну відповідність системи критеріям надійності, швидкодії та безпеки. Функціональні випробування на реальних наборах фінансових даних підтвердили коректність математичних моделей сальдо та балансу, а також стабільність роботи системи при обробці пакетних імпортів. Успішне проходження регресійних тестів дозволяє рекомендувати створену інформаційну систему «Finance Planner» для практичного використання як надійного інструменту персонального фінансового менеджменту. Подальші дослідження можуть бути спрямовані на впровадження методів машинного навчання для прогнозування майбутніх витрат на основі історичних даних користувача та розширення інтеграційного модуля для підтримки міжнародних платіжних систем.

ПЕРЕЛІК ПОСИЛАНЬ НА ДЖЕРЕЛА

1 Про платіжні послуги : Закон України від 30 черв. 2021 р. № 1591-IX. Відомості Верховної Ради України. 2021. № 45. Ст. 362. URL: https://zakon .rada. gov.ua/laws/show/1591-20 (дата звернення: 07.06.2026).

2 Про захист персональних даних : Закон України від 01 черв. 2010 р. № 2297-VI. Відомості Верховної Ради України. 2010. № 34. Ст. 481. URL: https://zakon.rada.gov.ua/laws/show/2297-17 (дата звернення: 07.06.2026).

3 Документація Monobank Open API. Офіційний портал для розробників. URL: https://api.monobank.ua/docs/ (дата звернення: 07.06.2026).

4 FastAPI Framework documentation. Python-based high-performance web framework. URL: https://fastapi.tiangolo.com/ (дата звернення: 07.06.2026).

5 SQLAlchemy 2.0 Documentation. Object Relational Mapper for Python. URL: https://docs.sqlalchemy.org/en/20/ (дата звернення: 07.06.2026).

6 Alembic documentation. Database migration tool for SQLAlchemy. URL: https://alembic.sqlalchemy.org/en/latest/ (дата звернення: 07.06.2026).

7 MySQL 8.0 Reference Manual. Oracle Corporation official documentation. URL: https://dev.mysql.com/doc/refman/8.0/en/ (дата звернення: 07.06.2026).

8 Pytest Documentation. Full-featured Python testing tool. URL: https://docs.pytest.org/en/stable/ (дата звернення: 07.06.2026).

9 Cryptography Documentation. Fernet symmetric encryption standard recipe. URL: https://cryptography.io/en/latest/fernet/ (дата звернення: 07.06.2026).

10 Chart.js documentation. Flexible JavaScript charting for designers & developers. URL: https://www.chartjs.org/docs/latest/ (дата` звернення: 07.06.2026).

11 Концепція відкритого банкінгу (Open Banking) в Україні. Офіційне інтернет-представництво Національного банку України. URL: https://bank.gov.ua /ua/payments/open-banking (дата звернення: 07.06.2026).

12 RFC 4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files. Internet Engineering Task Force (IETF). URL: https://datatracker.ietf.org

/doc/html/rfc4180 (дата звернення: 07.06.2026).

13 Microsoft Excel Spreadsheet File Format (.xlsx) Specification. Microsoft Learn Developer Center. URL: https://learn.microsoft.com/en-us/openspecs/office_standards /ms-xlsx/ (дата звернення: 07.06.2026).

14 Офіційний сайт та документація Wallet by BudgetBakers. Personal Finance Management Ecosystem. URL: https://budgetbakers.com/ (дата звернення: 07.06.2026).

15 Monobank (АТ «Універсал Банк»). Офіційний сайт першого в Україні мобільного банку. URL: https://www.monobank.ua/ (дата звернення: 07.06.2026).

16 Awesome Self-Hosted: Directory of Free Software Solutions and Self-Hosted Architecture. Open Source Initiative. URL: https://awesome-selfhosted.net/ (дата звернення: 07.06.2026).

17 Кизима Т. Є. Фінансова грамотність населення: проблемні питання фінансової участі населення у цифровій економіці: контент-аналіз та перспективи Світ фінансів. 2018. Вип. 1. С. 118–127. URL: http://sf.wunu.edu.ua/ (дата звернення: 07.06.2026).

18 Directive (EU) 2015/2366 of the European Parliament and of the Council of 25 November 2015 on payment services in the internal market (PSD2). Official Journal of the European Union. 2015. L 337. P. 35–127. URL: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32015L2366 (дата звернення: 07.06.2026).

19 Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data (GDPR). Official Journal of the European Union. 2016. L 119. P. 1–88. URL: https:// eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32016R0679 (дата звернення: 07.06.2026).

ДОДАТКИ

ДОДАТОК А

Файл env.py:

```
from __future__ import annotations
from logging.config import fileConfig
from alembic import context
from sqlalchemy import engine_from_config, pool
from app.core.config import settings
from app.db.base import Base
from app import models  # noqa: F401
config = context.config
if config.config_file_name is not None:
    fileConfig(config.config_file_name)
target_metadata = Base.metadata
def get_url() -> str:
    return settings.database_url
def run_migrations_offline() -> None:
    url = get_url()
    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )
    with context.begin_transaction():
        context.run_migrations()
def run_migrations_online() -> None:
    configuration = config.get_section(config.config_ini_section) or {}
    configuration["sqlalchemy.url"] = get_url()
       connectable   =   engine_from_config(configuration,   prefix="sqlalchemy.",
poolclass=pool.NullPool)
    with connectable.connect() as connection:
context.configure(connection=connection, target_metadata=target_metadata)
        with context.begin_transaction():
            context.run_migrations()
if context.is_offline_mode():
    run_migrations_offline()
else:
    run_migrations_online()
```

Файл Transactions.py:

```
from datetime import datetime

from fastapi import APIRouter, Depends, HTTPException, Query, status
from sqlalchemy import or_, select
from sqlalchemy.orm import Session

from app.api.deps import get_current_user, get_db
from app.models.account import Account
from app.models.category import Category
from app.models.transaction import Transaction
from app.models.user import User
from  app.schemas.transactions  import  ApplyMccCategoriesOut,  TransactionOut,
TransactionPatchIn, TransactionsListOut
from app.core.banking import TX_SOURCE_CSV
from app.services.mcc_categories import resolve_category_id_for_mcc
from app.services.mono_sync import MONO_SOURCE
```

```
from app.services.statement_categories import extract_category_from_description,
resolve_category_for_statement

router = APIRouter(prefix="/transactions", tags=["transactions"])

@router.get("", response_model=TransactionsListOut)
def list_transactions(
    from_dt: datetime | None = Query(default=None, alias="from"),
    to_dt: datetime | None = Query(default=None, alias="to"),
    bank: str | None = Query(
        default=None,
        description="Фільтр за банком для списку транзакцій: monobank або csv",
    ),
    account_id: int | None = None,
    account_ids: list[int] | None = Query(default=None, description="Мультивибір
рахунків"),
    category_id: int | None = None,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    q = (
        select(Transaction, Account.bank_name, Account.name)
        .join(Account, Account.id == Transaction.account_id)
        .where(Transaction.user_id == current_user.id)
    )
    if from_dt is not None:
        q = q.where(Transaction.occurred_at >= from_dt)
    if to_dt is not None:
        q = q.where(Transaction.occurred_at <= to_dt)
    if bank:
        if bank == "monobank":
            q = q.where(Transaction.external_source == MONO_SOURCE)
        elif bank == "csv":

            q = q.where(Transaction.external_source == TX_SOURCE_CSV)
        else:
                    raise  HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail="Невідомий bank")
    if account_id is not None:
        q = q.where(Transaction.account_id == account_id)
    if account_ids:
        q = q.where(Transaction.account_id.in_(account_ids))
    if category_id is not None:
        q = q.where(Transaction.category_id == category_id)
    q = q.order_by(Transaction.occurred_at.desc()).limit(500)

    rows = db.execute(q).all()

    items: list[TransactionOut] = []
    for r in rows:
        tx, bnk, accnm = r[0], r[1], r[2]
        items.append(
            TransactionOut(
                id=tx.id,
                account_id=tx.account_id,
                category_id=tx.category_id,
                amount_minor=tx.amount_minor,
                currency=tx.currency,
                occurred_at=tx.occurred_at,
                description=tx.description,
                merchant=tx.merchant,
                mcc=tx.mcc,
```

```
                external_source=tx.external_source,
                external_id=tx.external_id,
                bank_name=str(bnk) if bnk else None,
                account_name=str(accnm) if accnm else None,
            )
        )
    return TransactionsListOut(items=items)

@router.patch("/{transaction_id}", response_model=TransactionOut)
def patch_transaction(
    transaction_id: int,
    payload: TransactionPatchIn,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    tx = db.get(Transaction, transaction_id)
    if not tx or tx.user_id != current_user.id:
                    raise    HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Транзакцію не знайдено")

    data = payload.model_dump(exclude_unset=True)
    if "category_id" in data:
        cid = data["category_id"]
        if cid is not None:
            cat = db.get(Category, cid)
            if not cat or cat.user_id != current_user.id:
                    raise  HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail="Невірна категорія")
        tx.category_id = cid
    db.add(tx)
    db.commit()
    db.refresh(tx)

    acc = db.get(Account, tx.account_id)
    return TransactionOut(
        id=tx.id,
        account_id=tx.account_id,
        category_id=tx.category_id,
        amount_minor=tx.amount_minor,
        currency=tx.currency,
        occurred_at=tx.occurred_at,
        description=tx.description,
        merchant=tx.merchant,
        mcc=tx.mcc,
        external_source=tx.external_source,
        external_id=tx.external_id,
        bank_name=acc.bank_name if acc else None,
        account_name=acc.name if acc else None,
    )

def _inshi_category_ids(db: Session, *, user_id: int) -> list[int]:
    rows = db.scalars(
                    select(Category.id).where(Category.user_id    ==    user_id,
Category.name.in_(("Інші", "Інше")))
    ).all()
    return [int(r) for r in rows]

@router.post("/apply-mcc-categories", response_model=ApplyMccCategoriesOut)
def apply_mcc_categories(
    patch_missing: bool = Query(
        True,
        description="Оновити витрати без категорії (category_id = null)",
    ),
```

```
    patch_inshi_bucket: bool = Query(
        True,
         description="Перерахувати витрати в «Інші»/«Інше» за MCC (корисно, якщо
раніше не було цільових категорій)",
    ),
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
     """Підставити категорії за MCC для витрат Monobank (за замовчуванням: без
категорії + «Інші»/«Інше»)."""
    if not patch_missing and not patch_inshi_bucket:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
                    detail="Обери  хоча  б  один  режим:  patch_missing  або
patch_inshi_bucket",
        )

    parts = []
    if patch_missing:
        parts.append(Transaction.category_id.is_(None))
    if patch_inshi_bucket:
        inshi_ids = _inshi_category_ids(db, user_id=current_user.id)
        if inshi_ids:
            parts.append(Transaction.category_id.in_(inshi_ids))

    if not parts:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Створи категорію «Інші» або натисни «Стандартні категорії»",
        )

    q = select(Transaction).where(
        Transaction.user_id == current_user.id,
        Transaction.external_source == MONO_SOURCE,
        Transaction.amount_minor < 0,
        or_(*parts),
    )

    rows = db.scalars(q).all()
    updated = 0
    for tx in rows:
              cid   =   resolve_category_id_for_mcc(db,   user_id=current_user.id,
mcc=tx.mcc)
        if tx.category_id != cid:
            tx.category_id = cid
            updated += 1
    if updated:
        db.commit()
    return ApplyMccCategoriesOut(updated=updated)

@router.post("/apply-statement-categories",
response_model=ApplyMccCategoriesOut)
def apply_statement_categories(
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
     """Підставити категорії для операцій з виписки (за текстом категорії в
описі)."""
    inshi_ids = _inshi_category_ids(db, user_id=current_user.id)
    q = select(Transaction).where(
        Transaction.user_id == current_user.id,
        Transaction.external_source == TX_SOURCE_CSV,
```

```
    )
    if inshi_ids:
                      q     =     q.where(or_(Transaction.category_id.is_(None),
Transaction.category_id.in_(inshi_ids)))
    else:
        q = q.where(Transaction.category_id.is_(None))

    rows = db.scalars(q).all()
    updated = 0
    for tx in rows:
        cat_txt = extract_category_from_description(tx.description)
        if not cat_txt:
            continue
        cid = resolve_category_for_statement(
            db,
            user_id=current_user.id,
            category_text=cat_txt,
            amount_minor=tx.amount_minor,
            mcc=tx.mcc,
        )
        if cid is not None and tx.category_id != cid:
            tx.category_id = cid
            updated += 1
    if updated:
        db.commit()
    return ApplyMccCategoriesOut(updated=updated)
```

Файл security.py:

```
from datetime import datetime, timedelta, timezone
from typing import Any
from jose import jwt
from passlib.context import CryptContext
from app.core.config import settings
# Нові паролі — pbkdf2_sha256 (стабільно на Python 3.13). bcrypt залишено для
перевірки старих записів.
pwd_context       =       CryptContext(schemes=["pbkdf2_sha256",       "bcrypt"],
deprecated="auto")
def hash_password(password: str) -> str:
    return pwd_context.hash(password)
def verify_password(password: str, password_hash: str) -> bool:
    return pwd_context.verify(password, password_hash)
def create_access_token(*, subject: str) -> str:
    now = datetime.now(tz=timezone.utc)
    exp = now + timedelta(minutes=settings.jwt_access_ttl_min)
     payload: dict[str, Any] = {"sub": subject, "iat": int(now.timestamp()),
"exp": int(exp.timestamp())}
    return jwt.encode(payload, settings.jwt_secret, algorithm=settings.jwt_alg)
```

Файл crypto.py:

```
from cryptography.fernet import Fernet, InvalidToken
from app.core.config import settings
def _fernet() -> Fernet:
    try:
        return Fernet(settings.mono_token_enc_key.encode("utf-8"))
    except (ValueError, TypeError) as e:
```

```
        raise ValueError(
            "MONO_TOKEN_ENC_KEY має бути ключем Fernet (44 символи). "
               "Згенеруй: python -c \"from cryptography.fernet import Fernet;
print(Fernet.generate_key().decode())\""
        ) from e

def encrypt_text(value: str) -> str:
    token = _fernet().encrypt(value.encode("utf-8"))
    return token.decode("utf-8")

def decrypt_text(value_enc: str) -> str:
    try:
        data = _fernet().decrypt(value_enc.encode("utf-8"))
    except InvalidToken as e:
        raise ValueError("Invalid encrypted value") from e
    return data.decode("utf-8")
```

Файл mono.py:

```
@router.post("/connect", response_model=MonoConnectOut)
async def connect(
    payload: MonoConnectIn,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    try:
        await mono_get_client_info(payload.token)
    except MonoApiError as e:
                raise   HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=str(e))
    token_enc = encrypt_text(payload.token)
    existing = db.scalar(
        select(BankIntegration).where(
            BankIntegration.user_id == current_user.id,
            BankIntegration.provider == PROVIDER_MONOBANK,
        )
    )
    if existing:
        existing.secret_enc = token_enc
           else:                db.add(BankIntegration(user_id=current_user.id,
provider=PROVIDER_MONOBANK, secret_enc=token_enc))
    db.commit()
    return MonoConnectOut()
@router.post("/sync")
async def sync(
    from_ts: int | None = Query(default=None, description="unix seconds"),
    to_ts: int | None = Query(default=None, description="unix seconds"),
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    if to_ts is None:
        to_ts = int(datetime.now(tz=timezone.utc).timestamp())
    if from_ts is None
          from_ts         =         int((datetime.now(tz=timezone.utc)         -
timedelta(days=30)).timestamp())
    try:
               result   =   await   monobank_sync(db,   user_id=current_user.id,
from_ts=from_ts, to_ts=to_ts)
        rules_applied = apply_auto_rules_for_user(db, user_id=current_user.id)
        return {**result, "goal_contributions_from_rules": rules_applied}
    except ValueError as e:
```

```
                raise    HTTPException(status_code=status.HTTP_400_BAD_REQUEST,
detail=str(e))
```

Файл budget.py:

```
@router.post
def _spent_for_category_month(

db: Session, *, user_id: int, category_id: int, period_month: str, account_ids:
list[int] | None
) -> int:
    start, end = month_bounds_utc(period_month)
    parts = [
        Transaction.user_id == user_id,
        Transaction.category_id == category_id,
        Transaction.occurred_at >= start,
        Transaction.occurred_at < end,
        Transaction.amount_minor < 0,
    ]
    if account_ids:
        parts.append(Transaction.account_id.in_(account_ids))
    val = db.scalar(
                        select(func.coalesce(func.sum(-Transaction.amount_minor),
0)).where(*parts)
    )
    return int(val or 0)
@router.get("", response_model=BudgetsListOut)
def list_budgets(
    month: str = Query(..., pattern=r"^\d{4}-\d{2}$", description="YYYY-MM"),
    account_ids: list[int] | None = Query(default=None, description="Мультивибір
рахунків"),
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    budgets = db.scalars(
                    select(Budget).where(Budget.user_id    ==    current_user.id,
Budget.period_month == month)
    ).all()
    items: list[BudgetRowOut] = []
    for b in budgets:
        cat = db.get(Category, b.category_id)
        name = cat.name if cat and cat.user_id == current_user.id else "?"
        spent = _spent_for_category_month(
            db,
            user_id=current_user.id,
            category_id=b.category_id,
            period_month=month,
            account_ids=account_ids,
        )
        remaining = b.limit_minor - spent
        pct = (spent / b.limit_minor * 100.0) if b.limit_minor > 0 else 0.0
        items.append(
            BudgetRowOut(
                id=b.id,
                category_id=b.category_id,
                category_name=name,
                period_month=b.period_month,
                limit_minor=b.limit_minor,
                spent_minor=spent,
                remaining_minor=remaining,
                percent_used=round(pct, 2),
            )
```

```
        )
    return BudgetsListOut(items=items)
@router.post("",                                        response_model=BudgetRowOut,
status_code=status.HTTP_201_CREATED)
def create_budget(
    payload: BudgetCreateIn,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),

):
    cat = db.get(Category, payload.category_id)
    if not cat or cat.user_id != current_user.id:
                    raise    HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Категорію не знайдено")
    b = Budget(
        user_id=current_user.id,
        category_id=payload.category_id,
        period_month=payload.period_month,
        limit_minor=payload.limit_minor,
    )
    db.add(b)
    try:
        db.commit()
        db.refresh(b)
    except IntegrityError:
        db.rollback()
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail="Бюджет для цієї категорії й місяця вже існує",
        )
    spent = _spent_for_category_month(
        db,
        user_id=current_user.id,
        category_id=b.category_id,
        period_month=b.period_month,
        account_ids=None,
    )
    remaining = b.limit_minor - spent
    pct = (spent / b.limit_minor * 100.0) if b.limit_minor > 0 else 0.0
    return BudgetRowOut(
        id=b.id,
        category_id=b.category_id,
        category_name=cat.name,
        period_month=b.period_month,
        limit_minor=b.limit_minor,
        spent_minor=spent,
        remaining_minor=remaining,
        percent_used=round(pct, 2),
    )

@router.delete("/{budget_id}", status_code=status.HTTP_204_NO_CONTENT)
def delete_budget(
    budget_id: int,
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    b = db.get(Budget, budget_id)
    if not b or b.user_id != current_user.id:
                    raise    HTTPException(status_code=status.HTTP_404_NOT_FOUND,
detail="Бюджет не знайдено")
    db.delete(b)
    db.commit()
```

```
    return None
```

Файл momo_sync.py:

```
from __future__ import annotations
from datetime import datetime, timezone
from sqlalchemy import select
from sqlalchemy.exc import IntegrityError
from sqlalchemy.orm import Session

from app.core.crypto import decrypt_text
from app.models.account import Account
from app.core.banking import PROVIDER_MONOBANK, TX_SOURCE_MONOBANK
from app.models.bank_integration import BankIntegration
from app.models.transaction import Transaction
from app.services.mcc_categories import resolve_category_id_for_mcc
from app.services.mono_client import MonoApiError, mono_get_client_info,
mono_get_statement

MONO_SOURCE = TX_SOURCE_MONOBANK

def _currency_code_to_str(code: int | None) -> str:
    # MVP mapping; extend as needed
    if code == 980:
        return "UAH"
    if code == 840:
        return "USD"
    if code == 978:
        return "EUR"
    return "UAH"

async def monobank_sync(db: Session, *, user_id: int, from_ts: int, to_ts: int)
-> dict:
    conn = db.scalar(
        select(BankIntegration).where(
            BankIntegration.user_id == user_id,
            BankIntegration.provider == PROVIDER_MONOBANK,
        )
    )
    if not conn:
        raise ValueError("Monobank не підключено. Збережіть токен Monobank.")

    token = decrypt_text(conn.secret_enc)

     # щоб ensure_categories_for_mcc_mapping відпрацював знову після змін у
категоріях у цьому ж процесі
    db.info.pop(("mcc_cat_ensure", user_id), None)

    info = await mono_get_client_info(token)
    accounts = info.raw.get("accounts") or []

    created_tx = 0
    created_accounts = 0

    for acc in accounts:
        mono_account_id = str(acc.get("id"))
        if not mono_account_id:
            continue

        currency = _currency_code_to_str(acc.get("currencyCode"))
        name = acc.get("maskedPan") or acc.get("type") or "Monobank"
        if isinstance(name, list):
```

```
        existing_account = db.scalar(
            select(Account).where(
                Account.user_id == user_id,
                Account.external_source == MONO_SOURCE,
                Account.external_id == mono_account_id,
            )
        )
        if existing_account:
            existing_account.currency = currency
            existing_account.name = str(name)
            existing_account.bank_name = "Monobank"
            account_id = existing_account.id
        else:
            new_acc = Account(
                user_id=user_id,
                name=str(name),
                currency=currency,
                bank_name="Monobank",
                external_source=MONO_SOURCE,
                external_id=mono_account_id,
            )
            db.add(new_acc)
            db.flush()
            account_id = new_acc.id
            created_accounts += 1

        try:
                                statement    =    await    mono_get_statement(token,
account_id=mono_account_id, from_ts=from_ts, to_ts=to_ts)
        except MonoApiError:
            # skip a single account on error; continue others
            continue

        for item in statement:
            raw = item.raw
            ext_id = raw.get("id")
            if not ext_id:
                continue

                 occurred_at  =  datetime.fromtimestamp(int(raw.get("time",  0)),
tz=timezone.utc)
            amount_minor = int(raw.get("amount", 0))
            description = raw.get("description")
            mcc_raw = raw.get("mcc")
            if mcc_raw is None:
                mcc_raw = raw.get("originalMcc")
            merchant = raw.get("merchant")
            try:
                           mcc_val  =  int(mcc_raw)  if  mcc_raw  is  not  None  and
str(mcc_raw).strip() != "" else None
            except (TypeError, ValueError):
                mcc_val = None
             # MCC-мапінг застосовуємо лише до витрат; надходження — без авто-
категорії
            if amount_minor < 0:
                     cat_id  =  resolve_category_id_for_mcc(db,  user_id=user_id,
mcc=mcc_val)
            else:
                cat_id = None

            tx = Transaction(
```

ДОДАТОК Б

Файл analytics.py:

```
from datetime import datetime, timedelta, timezone

from fastapi import APIRouter, Depends, Query
from sqlalchemy import Date, cast, func, select
from sqlalchemy.orm import Session

from app.api.deps import get_current_user, get_db
from app.models.category import Category
from app.models.transaction import Transaction
from app.models.user import User
from app.schemas.analytics import (
    AnalyticsOverviewOut,
    CategoryExpenseOut,
    DailyPointOut,
    MccExpenseOut,
)

router = APIRouter(prefix="/analytics", tags=["analytics"])

@router.get("/overview", response_model=AnalyticsOverviewOut)
def analytics_overview(
    from_dt: datetime | None = Query(default=None, alias="from"),
    to_dt: datetime | None = Query(default=None, alias="to"),
    account_ids: list[int] | None = Query(default=None, description="Мультивибір
рахунків"),
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db),
):
    now = datetime.now(tz=timezone.utc)
    if to_dt is None:
        to_dt = now
    if from_dt is None:
        from_dt = to_dt - timedelta(days=30)

    uid = current_user.id
    base = [
        Transaction.user_id == uid,
        Transaction.occurred_at >= from_dt,
        Transaction.occurred_at <= to_dt,
    ]
    if account_ids:
        base.append(Transaction.account_id.in_(account_ids))

    income_minor = int(
        db.scalar(
            select(func.coalesce(func.sum(Transaction.amount_minor), 0)).where(
                *base,
                Transaction.amount_minor > 0,
            )
        )
        or 0
    )

    expense_minor = int(
        db.scalar(
            select(func.coalesce(func.sum(-Transaction.amount_minor), 0)).where(
                *base,
                Transaction.amount_minor < 0,
```

```
 )

    net_minor = int(
              db.scalar(select(func.coalesce(func.sum(Transaction.amount_minor),
0)).where(*base)) or 0
    )

    cur_row = db.execute(
        select(Transaction.currency, func.count())
        .where(*base)
        .group_by(Transaction.currency)
        .order_by(func.count().desc())
        .limit(1)
    ).first()
    currency = cur_row[0] if cur_row else "UAH"

    day_col = cast(Transaction.occurred_at, Date)
    daily_rows = db.execute(
        select(day_col, func.sum(Transaction.amount_minor))
        .where(*base)
        .group_by(day_col)
        .order_by(day_col)
    ).all()
    daily = [DailyPointOut(day=r[0], net_minor=int(r[1])) for r in daily_rows]

    mcc_rows = db.execute(
        select(Transaction.mcc, func.sum(-Transaction.amount_minor))
        .where(*base, Transaction.amount_minor < 0)
        .group_by(Transaction.mcc)
        .order_by(func.sum(-Transaction.amount_minor).desc())
        .limit(10)
    ).all()
     mcc_expenses = [MccExpenseOut(mcc=r[0], expense_minor=int(r[1])) for r in
mcc_rows]

    cat_label = func.coalesce(Category.name, "Без категорії")
    category_rows = db.execute(
        select(cat_label, func.sum(-Transaction.amount_minor))
        .select_from(Transaction)
        .outerjoin(Category, Category.id == Transaction.category_id)
        .where(
            *base,
            Transaction.amount_minor < 0,
        )
        .group_by(cat_label)
        .order_by(func.sum(-Transaction.amount_minor).desc())
        .limit(12)
    ).all()
    category_expenses = [
        CategoryExpenseOut(category_name=str(r[0]), expense_minor=int(r[1])) for
r in category_rows
    ]

    return AnalyticsOverviewOut(
        currency=currency,
        period_from=from_dt,
        period_to=to_dt,
        income_minor=income_minor,
        expense_minor=expense_minor,
        net_minor=net_minor,
        daily=daily,
        mcc_expenses=mcc_expenses,
```

```
  category_expenses=category_expenses,
    )

def read_root():
    return FileResponse("static/index.html")
```

Файл dashboard.js:

```
import { apiFetch, getToken } from "./core/api.js";
import { state } from "./core/state.js";
import { escapeHtml, formatMoney, hideDashboardHint, showDashboardHint,
updateDashboardMeta } from "./core/ui.js";
import { accountIdsQueryParams, getPeriodRange } from "./core/period.js";
import { syncMonobank } from "./core/api.js";

const periodSelect = document.getElementById("period-days");
const btnRefreshDashboard = document.getElementById("btn-refresh-dashboard");

export function destroyCharts() {
  if (state.chartDaily) {
    state.chartDaily.destroy();
    state.chartDaily = null;
  }
  if (state.chartCategory) {
    state.chartCategory.destroy();
    state.chartCategory = null;
  }
}

export async function loadDashboard() {
  if (!getToken()) return;
  const { from, to } = getPeriodRange();
  const qs = new URLSearchParams();
  qs.set("from", from.toISOString());
  qs.set("to", to.toISOString());
  for (const [k, v] of accountIdsQueryParams()) qs.append(k, v);

  let data;
  try {
    data = await apiFetch(`/analytics/overview?${qs}`);
  } catch (err) {
      showDashboardHint(err instanceof Error ? err.message : "Не вдалося
завантажити аналітику", "error");
    return;
  }

  const elIncome = document.getElementById("kpi-income");
  const elExpense = document.getElementById("kpi-expense");
  const elNet = document.getElementById("kpi-net");
    if (elIncome) elIncome.textContent = formatMoney(data.income_minor,
data.currency);
   if (elExpense) elExpense.textContent = formatMoney(data.expense_minor,
data.currency);
  if (elNet) {
    elNet.textContent = formatMoney(data.net_minor, data.currency);
    elNet.classList.toggle("negative", data.net_minor < 0);
  }
  updateDashboardMeta();

  if (typeof Chart === "undefined") return;

  destroyCharts();
```

```
  const ctxD = document.getElementById("chart-daily");
  const ctxM = document.getElementById("chart-by-category");
  if (!ctxD || !ctxM) return;

  Chart.defaults.color = "#6b7280";
  Chart.defaults.borderColor = "#e5e7eb";

  const daily = data.daily || [];
  const labelsD = daily.map((d) => d.day);
  const valsD = daily.map((d) => d.net_minor / 100);

  state.chartDaily = new Chart(ctxD, {
    type: "line",
    data: {
      labels: labelsD.length ? labelsD : ["—"],
      datasets: [
        {
          label: "Сальдо за день",
          data: labelsD.length ? valsD : [0],
          borderColor: "#2563eb",
          backgroundColor: "rgba(37, 99, 235, 0.08)",
          fill: true,
          tension: 0.25,
        },
      ],
    },
    options: {
      responsive: true,
      maintainAspectRatio: false,
      layout: { padding: 6 },
      plugins: { legend: { display: false } },
      scales: {
        x: { ticks: { maxRotation: 45, minRotation: 0 } },
        y: { beginAtZero: false },
      },
    },
  });

  const byCat = data.category_expenses || [];
   const colors = ["#3d8bfd", "#34d399", "#fbbf24", "#a78bfa", "#f87171",
"#22d3ee", "#fb7185", "#94a3b8", "#64748b", "#475569"];
  const catLegend = {
    position: "bottom",
     labels: { boxWidth: 12, padding: 10, font: { size: 11 }, maxWidth: 240,
usePointStyle: true },
  };

  if (!byCat.length) {
    state.chartCategory = new Chart(ctxM, {
      type: "doughnut",
       data: { labels: ["Немає витрат за період"], datasets: [{ data: [1],
backgroundColor: ["#e5e7eb"] }] },
      options: {
        responsive: true,
        maintainAspectRatio: false,
        layout: { padding: 8 },
        cutout: "58%",
        plugins: { legend: catLegend },
      },
    });
  } else {
    state.chartCategory = new Chart(ctxM, {
```

```
     type: "doughnut",
      data: {
        labels: byCat.map((x) => escapeHtml(x.category_name)),
             datasets:  [{  data:  byCat.map((x)  =>  x.expense_minor  /  100),
backgroundColor: colors.slice(0, byCat.length) }],
      },
      options: {
        responsive: true,
        maintainAspectRatio: false,
        layout: { padding: { left: 8, right: 8, top: 4, bottom: 4 } },
        cutout: "52%",
        plugins: { legend: catLegend },
      },
    });
  }
}

export function initDashboard() {
  periodSelect?.addEventListener("change", async () => {
    updateDashboardMeta();
    await loadDashboard();
    const { loadTransactions } = await import("./transactions.js");
    await loadTransactions();
  });

  btnRefreshDashboard?.addEventListener("click", async () => {
    hideDashboardHint();
    try {
      const synced = await syncMonobank({ silent: true });
      const { refreshAllData } = await import("./refresh.js");
      await refreshAllData();
      if (synced) {
        showDashboardHint(
                    `Monobank:   +${synced.accounts_created   ??   0}   рахунків,
~${synced.transactions_created ?? 0} операцій.`,
          "ok",
        );
      } else {
        showDashboardHint("Дані оновлено.", "ok");
      }
    } catch (err) {
          showDashboardHint(err  instanceof  Error  ?  err.message  :  "Помилка
синхронізації", "error");
    }
  });
}
```

ДОДАТОК В

Файл test_goals_service.py:

```
from __future__ import annotations

from types import SimpleNamespace

import pytest

from app.models.goal import (
    GOAL_MODE_ACCOUNT_BALANCE,
    RULE_AMOUNT_ANY,
    RULE_AMOUNT_EXPENSE,
    RULE_AMOUNT_INCOME,
)
from app.services.goals import (
    contribution_amount_for_tx,
    percent_reached,
    tx_matches_rule,
    validate_goal_mode,
)

def _tx(**kwargs: object) -> SimpleNamespace:
    defaults = {
        "id": 1,
        "user_id": 1,
        "category_id": None,
        "amount_minor": 10_000,
        "merchant": "Monobank",
        "description": "Зарплата за березень",
    }
    defaults.update(kwargs)
    return SimpleNamespace(**defaults)

def _rule(**kwargs: object) -> SimpleNamespace:
    defaults = {
        "id": 1,
        "goal_id": 1,
        "user_id": 1,
        "category_id": None,
        "merchant_contains": None,
        "description_contains": None,
        "min_amount_minor": None,
        "amount_side": RULE_AMOUNT_INCOME,
        "auto_apply": False,
    }
    defaults.update(kwargs)
    return SimpleNamespace(**defaults)

@pytest.mark.parametrize(
    ("saved", "target", "expected"),
    [
        (50_000, 100_000, 50),
        (150_000, 100_000, 100),
        (0, None, None),
        (10_000, 0, None),
    ],
)
def test_percent_reached(saved: int, target: int | None, expected: int | None):
    assert percent_reached(saved, target) == expected
```

```
        (5000, RULE_AMOUNT_INCOME, 5000),
        (-5000, RULE_AMOUNT_INCOME, None),
        (-3000, RULE_AMOUNT_EXPENSE, 3000),
        (3000, RULE_AMOUNT_EXPENSE, None),
        (-2000, RULE_AMOUNT_ANY, 2000),
        (2000, RULE_AMOUNT_ANY, 2000),
    ],
)
def test_contribution_amount_for_tx(amount: int, side: str, expected: int |
None):
    assert contribution_amount_for_tx(amount, side) == expected

def test_tx_matches_rule_by_description():
    tx = _tx(description="Кафе центр", amount_minor=-15000)
    rule = _rule(description_contains="кафе", amount_side=RULE_AMOUNT_EXPENSE)
    assert tx_matches_rule(tx, rule)

def test_tx_matches_rule_rejects_wrong_category():
    tx = _tx(category_id=5, amount_minor=10000)
    rule = _rule(category_id=9, amount_side=RULE_AMOUNT_INCOME)
    assert not tx_matches_rule(tx, rule)

def test_tx_matches_rule_min_amount():
    tx = _tx(amount_minor=500)
    rule = _rule(min_amount_minor=1000, amount_side=RULE_AMOUNT_ANY)
    assert not tx_matches_rule(tx, rule)

def test_validate_goal_mode_account_balance_requires_account():
    validate_goal_mode("manual", None)
    with pytest.raises(ValueError, match="баланс"):
        validate_goal_mode(GOAL_MODE_ACCOUNT_BALANCE, None)
```

Файл test_period.py:

```
from datetime import datetime, timezone

import pytest

from app.api.utils.period import month_bounds_utc

def test_month_bounds_january():
    start, end = month_bounds_utc("2026-01")
    assert start == datetime(2026, 1, 1, tzinfo=timezone.utc)
    assert end == datetime(2026, 2, 1, tzinfo=timezone.utc)

def test_month_bounds_december_rolls_year():
    start, end = month_bounds_utc("2025-12")
    assert start == datetime(2025, 12, 1, tzinfo=timezone.utc)
    assert end == datetime(2026, 1, 1, tzinfo=timezone.utc)

@pytest.mark.parametrize("bad", ["2026", "2026-13", "ab-01", ""])
def test_month_bounds_invalid(bad: str):
    with pytest.raises(ValueError):
        month_bounds_utc(bad)
}
```

Файл test_schemas_goals.py:

```
from datetime import datetime, timezone

from app.schemas.goals import GoalRuleCreateIn, GoalRuleOut
```

```
def test_goal_rule_create_defaults():
    data = GoalRuleCreateIn()
    assert data.amount_side == «income»
    assert data.auto_apply is False

def test_goal_rule_out_from_orm_like_object():
    «»»Регресія: без from_attributes API повертав 500 після POST /rules.»»»

    Class OrmRule:
        id = 1
        goal_id = 2
        category_id = None
        merchant_contains = None
        description_contains = «зарплата»
        min_amount_minor = None
        amount_side = «income»
        auto_apply = True
        created_at = datetime(2026, 1, 15, 12, 0, tzinfo=timezone.utc)

    out = GoalRuleOut.model_validate(OrmRule())
    assert out.id == 1
    assert out.goal_id == 2
    assert out.description_contains == «зарплата»
    assert out.auto_apply is True
```

Файл test_security.py:

```
from jose import jwt

from app.core.config import settings
from     app.core.security     import     create_access_token,     hash_password,
verify_password

def test_password_hash_roundtrip():
    raw = «my-secure-password»
    hashed = hash_password(raw)
    assert hashed != raw
    assert verify_password(raw, hashed)
    assert not verify_password(«wrong», hashed)

def test_access_token_contains_subject():
    token = create_access_token(subject=»42»)
                payload       =       jwt.decode(token,       settings.jwt_secret,
algorithms=[settings.jwt_alg])
    assert payload[«sub»] == «42»
    assert «exp» in payload
   assert «iat» in payload
```

Протокол аналізу звіту подібності науковим керівником

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

Автор: Кулинич Сергій Миколайович

Співавтор:

Назва: Бакалаврська_Робота_Кулинич

Науковий керівник: Гарасимів Тарас Григорович

Підрозділ: Каф. КСМ

Коефіцієнт подібності 1:2.5%

Мікропробіли: 2

Заміна букв: 1

Інтервали: 0

Білі знаки: 35

Дата створення звіту: 2026-06-09 03:33:26.0

Після аналізу Звіту подібності констатую наступне:

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-06-09 Надія Ширмовська

Дата експерт

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: «Розробка web-додатку для планування та обліку особистих фінансів з інтеграцією банківських даних засобами JS, Fetch API, Python, mySql»

Обсяг пояснювальної записки 67 аркушів.

16 рисунків;

11 таблиць;

4 додатки.

Дата завершення роботи: *10 червня 2026 р*.

Підпис студента-дипломника _____ *Кулинич С.М.*