МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

ім. Ю.М. Потебні

ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)

(рівень вищої освіти)

на тему **Розробка веб-додатку для управління особистими фінансами**

Виконав: студент 4 курсу, групи 6.1212-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
А.Г. Біліба
(ініціали та прізвище)
Керівник доцент, І.А. Скрипник

(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П. О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ім. Ю.М. Потебні ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА

“ 01 ” березня 2026 року

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

Білібі Анні Геннадіївні
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка веб-додатку для управління особистими фінансами

керівник роботи Скрипник Ірина Анатоліївна, доцент,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджена наказом ЗНУ від 23.12.2025 р. № 1830-с
2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   - огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - огляд та аналіз існуючих рішень та аналогів;
   - дослідження технологій, необхідних для реалізації;
   - створення програмного продукту та його опис;
   - перелік вимог для роботи програми;
   - розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) слайдів презентації

6. Консультанти розділів бакалаврської роботи

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

7. Дата видачі завдання ____01.03.2026 р._____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів магістерської роботи	Примітка
1.	Аналіз предметної області.	20.03-24.03	виконано
2.	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	25.03	виконано
3.	Аналіз програм-аналогів.	26.03-29.04	виконано
4.	Вибір технологій для реалізації.	01.04-05.04	виконано
5.	Проєктування архітектури системи, опис сценаріїв UML та побудова ER-діаграми бази даних	05.04-22.04	виконано
6.	Розробка серверної частини веб-додатку, підключення до БД та реалізація захищеного REST API	22.04-05.05	виконано
7.	Розробка клієнтської частини додатка (React SPA), верстка адаптивного інтерфейсу користувача	05.05-08.05	виконано
8.	Реалізація аналітичного модуля, системи SVG-візуалізації та автоматичної перевірки лімітів	08.05-16.05	виконано
9.	Впровадження заходів безпеки (JWT, bcrypt, CORS), мережеве налаштування та оптимізація запитів	16.05-22.05	виконано
10.	Хмарне розгортання БД (Neon DB) та прикладних модулів (Vercel), тестування та налагодження	22.05-25.05	виконано
11.	Оформлення пояснювальної записки, підготовка презентації та графічного матеріалу до захисту	25.05-02.06	виконано

Студент _____ _____А.Г. Біліба

Керівник роботи _____ _____І. А. Скрипник

Нормоконтроль пройдено

Нормоконтролер _____ _____І.А. Скрипник

АНОТАЦІЯ

Сторінок – 72.

Рисунків – 12.

Джерел – 15.

Біліба А.Г. Розробка веб-додатку для управління особистими фінансами: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник І. А. Скрипник. Запоріжжя : ЗНУ, 2026, 72 с.

Мета роботи – проєктування, реалізація та хмарне розгортання адаптивного веб-додатку «FinanceFlow» для автоматизованого обліку фінансів, контролю бюджетів і візуалізації витрат.

Методи дослідження – системний і порівняльний аналіз, об’єктно-орієнтоване проєктування та UML-моделювання, проєктування реляційних баз даних, компонентний підхід у веб-розробці, ручне тестування.

У ході виконання роботи було спроєктовано та реалізовано адаптивний веб-додаток «FinanceFlow». Бекенд-частину побудовано на базі платформи Node.js та фреймворку Express.js з використанням Sequelize ORM для роботи з реляційною СУБД PostgreSQL. Клієнтську частину (SPA) розроблено на React + TypeScript із застосуванням збирача Vite та Vanilla CSS для стилізації.

Реалізовано систему безпечної автентифікації користувачів на основі JWT-токенів із хешуванням паролів алгоритмом bcryptjs. Створено аналітичний модуль із динамічною візуалізацією витрат за допомогою інтерактивних SVG-діаграм, систему автоматичної перевірки перевищення встановлених бюджетів та фонову генерацію сповіщень. Розгортання бази даних проведено на серверлес-платформі Neon DB, а серверного та клієнтського модулів — на платформі Vercel з налаштуванням автоматичного CI/CD конвеєра через GitHub.

Ключові слова: веб-додаток, управління фінансами, Node.js, Express, React, TypeScript, PostgreSQL, Sequelize ORM, JWT, SVG-діаграми, хмарне розгортання, Vercel, Neon DB.

ABSTRACT

Pages – 72.

Drawings – 12.

Sources – 15.

Biliba A.H. Development of a web application for personal finance management»: bachelor’s qualification thesis in specialty 121 “Software Engineering” / academic supervisor I. A. Skrypnyk. Zaporizhzhia : ZNU, 2026, 72 p.

Goal of the work – design, software implementation, and practical cloud deployment of a modern adaptive web application «FinanceFlow» for automated finance tracking, budget limits monitoring, and expense statistics visualization.

Methods of research – system and comparative analysis, object-oriented design and UML-modeling, relational database design, component-oriented web development approach, manual verification testing.

In the course of the work, the adaptive web application «FinanceFlow» was designed and implemented. The backend part is built on the Node.js platform and Express.js framework using Sequelize ORM to interact with the PostgreSQL relational DBMS. The client part (SPA) is developed on React + TypeScript using Vite bundler and Vanilla CSS for styling.

A secure user authentication system based on JWT tokens with password hashing by bcryptjs algorithm has been implemented. An analytical module with dynamic expense visualization using interactive SVG charts, a system for automated monitoring of active budget limits, and background notification generation have been created. The database was deployed on the Neon DB serverless platform, while server and client modules were deployed on Vercel with an automated CI/CD pipeline set up via GitHub.

Keywords: web application, finance management, node.js, express, react, typescript, postgresql, sequelize orm, json web token, svg charts, cloud deployment, vercel, neon db.

ЗМІСТ

ВСТУП 10
1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ 14
1.1 Сучасний стан та тенденції розвитку систем управління фінансами 14
1.2 Порівняльний аналіз існуючих аналогів 15
1.3. Визначення вимог до веб-додатку 18
1.4. Обґрунтування вибору технологій розробки 20
1.5 Аналіз вимог безпеки та захисту персональних даних у фінансових системах 22
2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА БАЗИ ДАНИХ 25
2.1. Проєктування поведінки системи 25
2.1.1 Сценарій прецеденту «Реєстрація нового користувача» 27
2.1.2 Сценарій прецеденту «Додавання фінансової транзакції» 29
2.1.3 Сценарій прецеденту «Контроль та перевищення ліміту бюджету» 31
2.2 Архітектурне проєктування веб-додатку 33
2.3 Проектування бази даних 35
2.4. Розробка специфікації REST API ендпоінтів 38
3. ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ 41
3.1 Розробка серверної частини (Backend) 41
3.1.1. Налаштування середовища та підключення до бази даних 41
3.1.2. Реалізація механізму автентифікації користувачів 43
3.1.3. Логіка автоматичного перерахунку лімітів та контролю бюджетів 45
3.2. Розробка клієнтської частини 47
3.2.1. Збирання проєкту та конфігурація клієнтського API 47
3.2.2. Керування навігацією та станом веб-додатку 50
3.2.3. Стилізація та візуалізація аналітичних звітів 51

3.3 Реалізація системи сповіщень та аналітичного модуля 54
3.3.1. Функціонування системи сповіщень на стороні клієнта 54
3.3.2. Агрегація даних на сервері та візуалізація аналітики 55
3.4 Заходи з безпеки даних та оптимізації продуктивності 56
3.4.1. Криптографічний захист паролів 56
3.4.2. Авторизація запитів на базі Stateless JWT 57
3.4.3. Захист від SQL-ін'єкцій та ізоляція даних 58
3.4.4. Мережева безпека та CORS 58
3.4.5. Запобігання XSS-атакам на стороні клієнта 58
3.4.6. Оптимізація продуктивності бази даних 59
3.5 Оптимізація продуктивності клієнтської та серверної частин веб-додатку 59
3.5.1. Оптимізація клієнтської частини (Frontend) 60
3.5.2. Оптимізація серверної частини (Backend) та бази даних 61
4. РОЗГОРТАННЯ, ТЕСТУВАННЯ ТА ІНСТРУКЦІЯ КОРИСТУВАЧА 62
4.1 Тестування додатку 62
4.2 Розгортання системи 62
4.2.1 Хмарна база даних (Neon DB) 63
4.2.2 Розгортання серверної частини (Vercel Serverless Functions) 64
4.2.3. Розгортання клієнтської частини та CI/CD конвеєр 65
4.3 Інструкція користувача 65
4.3.1. Екран автентифікації та реєстрації 66
4.3.2. Головна панель (Dashboard) 67
4.3.3. Історія транзакцій 68
4.3.4. Керування категоріями та лімітами бюджетів 69
4.3.5. Модуль детальної аналітики 69
ВИСНОВКИ 71
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 72

ВСТУП

Актуальність теми

Сьогодні стежити за своїми фінансами стає дедалі складніше. Більшість розрахунків перейшли в онлайн, у людей з'явилося по кілька банківських карток, а доходи часто надходять із різних джерел. У таких умовах легко втратити контроль над витратами, що заважає створювати заощадження та планувати великі покупки.

Багато хто досі намагається записувати витрати в блокнот або вести таблиці в Excel. Проте ручний облік вимагає забагато часу та дисципліни. Крім того, звичайні таблиці не можуть автоматично надсилати сповіщення на телефон або будувати інтерактивні графіки одразу після купівлі товару. Спеціалізовані веб-додатки вирішують цю проблему, роблячи облік швидким і непомітним.

Об'єкт дослідження

Процес обліку, аналізу та планування особистих грошових потоків користувача.

Предмет дослідження

Методи розробки клієнт-серверних додатків для моніторингу та обмеження витрат особистого бюджету.

Мета роботи

Проектування та розробка веб-додатка, який дозволить користувачам вести облік транзакцій, встановлювати ліміти на категорії витрат, бачити наочну аналітику та автоматично отримувати сповіщення про перевищення бюджету.

Для реалізації цієї мети потрібно виконати такі завдання:

- Проаналізувати аналоги та виділити їхні основні недоліки.
- Сформулювати вимоги до майбутнього додатку.
- Обґрунтувати вибір стеку технологій (React, TypeScript, Node.js, Express, PostgreSQL).
- Спроектувати структуру бази даних та намалювати ER-діаграму.
- Описати маршрути REST API для обміну даними між клієнтом та сервером.
- Реалізувати серверну частину (використовуючи JWT авторизацію).
- Створити адаптивний інтерфейс кліенстької частини.
- Розгорнути систему на сторонньому сервері (БД на Neon DB, а сам додаток – на Vercel).
- Протестувати роботу всієї системи та зробити висновки.

Методи дослідження

У роботі використовувалися аналіз аналогів, моделювання баз даних, компонентний підхід до побудови інтерфейсів

Практичне значення

Практичне значення отриманих результатів полягає у створенні повністю готового додатку, який уже працює в мережі. Використання хмарних платформ Vercel та Neon.tech дозволяє проекту працювати безкоштовно та стабільно.

Глосарій

API (Application Programming Interface) – програмний інтерфейс застосунку; набір стандартизованих правил, протоколів та інструментів, що дозволяють різним програмним компонентам взаємодіяти між собою.

ACID (Atomicity, Consistency, Isolation, Durability) – набір властивостей транзакцій бази даних (атомарність, узгодженість, ізольованість, довговічність), що гарантують надійність виконання операцій.

CI/CD (Continuous Integration / Continuous Deployment) – безперервна інтеграція та безперервне розгортання; методологія розробки ПЗ, яка передбачає автоматизацію збирання, тестування та деплою коду при змінах у репозиторії.

CORS (Cross-Origin Resource Sharing) – спільне використання ресурсів різних джерел; механізм безпеки веб-браузерів, що обмежує або дозволяє веб-сторінкам здійснювати крос-доменні HTTP-запити.

CRUD (Create, Read, Update, Delete) – базові операції роботи з даними (створення, читання, оновлення, видалення), які підтримуються більшістю інформаційних систем.

CSS (Cascading Style Sheets) – каскадні таблиці стилів; мова опису зовнішнього вигляду веб-сторінок та інтерфейсів користувача.

DOM (Document Object Model) – об'єктна модель документа; структуроване представлення HTML-елементів веб-сторінки у вигляді дерева об'єктів, яким можна керувати за допомогою JavaScript коду.

ERD (Entity-Relationship Diagram) – діаграма «сутність-зв'язок»; графічна модель даних, яка відображає основні сутності інформаційної системи та логічні зв'язки між ними.

HTTP / HTTPS (HyperText Transfer Protocol / Secure) – протокол передачі гіпертексту (захищений); мережевий протокол прикладного рівня для обміну даними між клієнтом та сервером.

JSON (JavaScript Object Notation) – текстовий формат обміну даними, заснований на синтаксисі JavaScript; використовується для передачі структурованої інформації в REST API запитах.

JWT (JSON Web Token) – стандартизований формат токенів для безпечної автентифікації користувачів без збереження стану сесії на сервері застосунків.

ORM (Object-Relational Mapping) – об'єктно-реляційне відображення; технологія розробки ПЗ, яка пов'язує бази даних з концепціями об'єктно-

орієнтованого програмування, дозволяючи працювати з таблицями як з об'єктами.

REST (Representational State Transfer) – архітектурний стиль взаємодії клієнта та сервера через мережу на основі протоколу HTTP та використання стандартизованих методів (GET, POST, PUT, DELETE).

SPA (Single Page Application) – односторінковий веб-додаток; тип веб-застосунку, який завантажує єдиний HTML-документ і динамічно оновлює його вміст без перезавантаження сторінки під час взаємодії з користувачем.

SQL (Structured Query Language) – мова структурованих запитів; декларативна мова програмування для створення, модифікації та керування даними в реляційних СУБД.

SVG (Scalable Vector Graphics) – масштабована векторна графіка; мова розмітки на основі XML для опису двовимірної векторної графіки, що дозволяє рендерити масштабовані малюнки та графіки без втрати якості.

UI/UX (User Interface / User Experience) – інтерфейс користувача та досвід користування; проектування графічного оформлення додатка та оптимізація зручності його використання кінцевим споживачем.

UML (Unified Modeling Language) – уніфікована мова моделювання; стандартизована мова графічного опису для моделювання, проектування та документування архітектури та поведінки програмних систем.

UUID (Universally Unique Identifier) – універсальний унікальний ідентифікатор; 128-бітне число, що використовується в IT для унікальної ідентифікації записів у базах даних без необхідності централізованого керування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ

1.1 Сучасний стан та тенденції розвитку систем управління фінансами

Облік особистих фінансів з технічного погляду є класичною інформаційною системою з високою частотою операцій створення, читання, оновлення та видалення (CRUD), а також потребою в оперативній аналітичній обробці даних (OLAP). Фінансові концепції ведення бюджету - це правила та обмеження, які розробник повинен перекласти на мову програмної логіки, обмежень бази даних (database constraints) та тригерів.

Для реалізації бізнес-логіки додатків зазвичай використовують три основні концепції, кожна з яких ставить свої вимоги до структури даних та алгоритмів:

1. Метод лімітування (“50/30/20”). З погляду програмування, цей підхід вимагає динамічного агрегування транзакцій та порівняння отриманих сум із встановленими відсотковими частками від загального доходу. Програма повинна на кожну фінансову подію (додавання транзакції) оновлювати стан (state) системи, перераховувати загальний баланс та співвідношення категорій, не перевантажуючи при цьому процесор клієнта зайвими перемальовуваннями UI.

2. Метод віртуальних конвертів. Ця бізнес-логіка вимагає створення в базі даних зв'язків “один-до-багатьох” між сутністю користувача та лімітами категорій. Технічно це реалізується через порівняння суми поля `amount` таблиці `Transactions` із полем `limit_amount` таблиці `Budgets` за певний часовий інтервал (наприклад, місяць). Алгоритм перевірки лімітів має спрацьовувати автоматично як тригер (на рівні СУБД або на рівні middleware сервера бэкенду) при кожному запиті `POST` на створення транзакції.

3. Бюджетування з нульовим балансом. Цей підхід накладає суворе обмеження цілісності даних (integrity constraint) на рівні бізнес-логіки. Система повинна перевіряти, щоб сума всіх зарезервованих лімітів та вільних коштів дорівнювала загальному обсягу доходів користувача. Будь-яка невідповідність має блокувати транзакцію на рівні сервісного шару додатку (transaction rollback) для запобігання втрати консистентності даних.

Спроби реалізації цих алгоритмів через локальні таблиці Excel або ручні записи є неефективними через відсутність автоматичної перевірки обмежень цілісності та неможливість асинхронної обробки подій. Роль автоматизації в даному контексті полягає в перенесенні обчислювальної складності з користувача на програмне забезпечення. Автоматизовані системи дозволяють реалізувати асинхронний обмін даними між клієнтом та сервером, оптимізувати SQL-запити для швидкої побудови аналітики (наприклад, через агрегатні функції `SUM` та групування `GROUP BY`), а також забезпечити автоматичне сповіщення користувача на основі обробки подій (event-driven architecture). Таким чином, автоматизація особистих фінансів є типовим прикладом розробки високонавантаженого SPA-додатку з динамічним управлінням станом та роботою з реляційними базами даних.

1.2 Порівняльний аналіз існуючих аналогів

Перед початком проектування власного програмного забезпечення необхідно провести детальний технічний аналіз існуючих рішень на ринку. Для порівняння було обрано чотири популярні системи обліку фінансів: Monefy, Wallet by BudgetBakers, YNAB (You Need A Budget) та CoinKeeper. Оцінювати ці додатки ми будемо не з точки зору їхньої фінансової користі, а через призму архітектури, методів рендерингу інтерфейсу, способів синхронізації та механізмів безпеки даних.

Більшість мобільних додатків на кшталт Monefy розроблені як нативні або гібридні клієнти, що використовують локальні бази даних (наприклад,

SQLite). Головною технічною проблемою таких систем є синхронізація даних між різними пристроями одного користувача. Синхронізація через хмари (наприклад, Google Drive або Dropbox) часто призводить до конфліктів версій бази даних (split-brain) та втрати транзакцій при роботі в офлайн-режимі. Крім того, нативні додатки важко масштабувати на веб-платформи без повного переписування кодової бази.

Складніші веб-сервіси, такі як YNAB, використовують клієнт-серверну архітектуру з використанням SPA на фронтенді та REST API на бекенді. Проте з точки зору продуктивності вони є дуже важкими для клієнтських пристроїв. Вони завантажують великі JavaScript-бандли і використовують сторонні бібліотеки для побудови графіків, що уповільнює перше завантаження сторінки (Time to Interactive). Що стосується безпеки, то більшість комерційних рішень використовують закриті протоколи обміну даними та зберігають чутливу інформацію користувачів на власних серверах без наскрізного шифрування, що створює ризики витоку персональних даних.

Для наочності порівняємо технічні характеристики аналогів у таблиці 1.1.

Таблиця *1.1 – Порівнялний аналіх аналогів*

Назва додатку	Архітектур-ний шаблон	Зберігання	Технологія рендерингу UI	Метод автентифіка-ції	Синхроніза-ція
Monefy	Локальний клієнт	SQLite / Google Drive	Native SDK (Android/ iOS)	Локальний PIN-код	Файлова хмара
Wallet	Клієнт-серверна	PostgreSQL / Cloud	Hybrid (React Native / Web)	OAuth 2.0 / Email	REST API (Real-time)
YNAB	Клієнт-серверна	NoSQL (Cloud)	SPA (React) + сторонні SVG	JWT / Cookie session	REST API
CoinKeeper	Клієнт-серверна	MongoDB (Cloud)	Canvas / Custom 2D	OAuth 2.0	REST API

Аналіз технічних характеристик показує кілька загальних недоліків існуючих систем. По-перше, використання важких фреймворків та сторонніх графічних бібліотек (на кшталт Chart.js чи D3) на клієнтській частині збільшує

час рендерингу графіків на слабких мобільних процесорах. По-друге, реляційні бази даних у хмарах комерційних сервісів часто мають надмірну нормалізацію, що призводить до повільного виконання аналітичних запитів типу `GROUP BY` при великій кількості транзакцій.

Перевагою нашого розроблюваного веб-додатку буде використання триланкової архітектури на базі легкого React-клієнта з TypeScript та бекенду на Node.js/Express. Для мінімізації навантаження на процесор клієнта рендеринг графіків буде реалізовано на чистих SVG-елементах за допомогою математичних формул у коді React, без підключення сторонніх JS-модулів. Впровадження авторизації на основі JWT-токенів з шифруванням паролів через bcryptjs забезпечить надійний рівень безпеки без використання застарілих механізмів сесій на основі Cookie. Крім того, розгортання реляційної бази даних на хмарній платформі Neon DB (PostgreSQL) дозволить скористатися перевагами автоматичного масштабування СУБД під поточне навантаження.

1.3 Визначення вимог до веб-додатку

Процес проектування будь-якої інформаційної системи вимагає чіткого структурування вимог до програмного забезпечення згідно з інженерними стандартами. Визначення специфікацій дозволяє закласти правильну архітектуру бази даних та обрати оптимальні алгоритми обробки даних на сервері. Вимоги до розроблюваного веб-додатку «FinanceFlow» поділяються на дві основні категорії: функціональні, які описують поведінку та сервіси системи, та нефункціональні, що визначають якісні характеристики та технічні обмеження програмного продукту.

Функціональні вимоги описують взаємодію користувача з додатком та внутрішню логіку обробки інформації сервером. До системи висуваються такі вимоги:

- Модуль автентифікації та авторизації. Додаток має підтримувати реєстрацію нових користувачів із валідацією полів (унікальність email, довжина пароля не менше 6 символів) та вхід за допомогою пари email/пароль. Сервер повинен генерувати сесійний токен доступу JWT з терміном дії 7 днів і повертати його клієнту для збереження в локальному сховищі (localStorage). Доступ до фінансових маршрутів API має бути заблокований для неавторизованих запитів.

- Управління фінансовими транзакціями (CRUD). Користувач повинен мати можливість створювати, переглядати, редагувати та видаляти операції доходів та витрат. Кожна транзакція обов'язково повинна містити числове поле суми (додатне значення, два знаки після коми), тип операції (income/expense), унікальний ідентифікатор категорії (categoryId), дату (формат YYYY-MM-DD) та необов'язкове текстове поле опису.

- Управління категоріями. Система має розділяти категорії на системні (доступні для всіх користувачів, створені за замовчуванням і захищені від видалення) та користувацькі (створені конкретним користувачем з можливістю їх редагування та видалення). Кожна категорія повинна містити унікальне ім'я, тип операції, назву іконки та HEX-код кольору для UI.

- Моніторинг та лімітування бюджетів. Користувач повинен мати можливість встановлювати граничні суми витрат для окремих категорій або загальний ліміт на певний проміжок часу (тиждень/місяць). Система має автоматично розраховувати суму фактичних витрат за вибраний період та порівнювати її з лімітом при додаванні кожної нової транзакції.

- Генерація сповіщень. У разі перевищення ліміту бюджету сервер повинен автоматично створювати запис у таблиці сповіщень із описом перевищення (сума та категорія) та прив'язкою до ідентифікатора користувача. Клієнтська частина повинна відображати кількість непрочитаних повідомлень та надавати можливість позначити їх як прочитані.

- Аналітичний модуль. Програма повинна виконувати агрегацію транзакцій користувача для побудови аналітичних звітів за категоріями витрат та динамікою доходів/витрат по місяцях.

Нефункціональні вимоги визначають технічні обмеження системи, її продуктивність та безпеку:

- Продуктивність та швидкодія. Час відгуку сервера для більшості запитів API не повинен перевищувати 200 мс у нормальних умовах навантаження. Швидкість першого рендерингу інтерфейсу на клієнті має бути меншою за 1,5 секунди на середніх мобільних пристроях.

- Безпека та конфіденційність. Всі паролі користувачів перед збереженням у базу даних повинні хешуватися на рівні сервера за допомогою алгоритму `bcryptjs` із сіллю (salt factor 10). Прямий обмін даними між клієнтом та сервером має здійснюватися виключно через захищений протокол HTTPS. Для захисту від SQL-ін'єкцій всі запити до бази даних повинні виконуватися через типізовані моделі ORM Sequelize з параметризацією.

- Масштабованість та сумісність. Програмна архітектура додатку має підтримувати розгортання в безсерверному хмарному середовищі (Serverless) з автоматичним виділенням ресурсів під навантаження. База даних повинна підтримувати пул підключень (connection pooling) для оптимізації роботи з хмарною СУБД.

- Інтерфейс та адаптивність. Користувацький інтерфейс має бути повністю адаптивним

1.4 Обґрунтування вибору технологій розробки

Вибір технологічного стеку є базовим етапом проектування системи, який визначає її архітектурну гнучкість, швидкість роботи та складність розробки. Для веб-додатку було обрано клієнт-серверну архітектуру на основі REST API. Такий підхід дозволяє повністю розділити інтерфейс користувача (фронтенд) та сервер із бізнес-логікою (бекенд).

Головна перевага REST API полягає в його слабкій зв'язності (decoupling). Завдяки цьому ми можемо оновлювати фронтенд або навіть написати нативний мобільний додаток у майбутньому, не змінюючи логіку роботи сервера. Крім того, REST працює за принципом відсутності збереження стану сесії на сервері (stateless). Усі дані про авторизацію зберігаються на клієнті у вигляді токена, що суттєво полегшує масштабування та знижує навантаження на серверну пам'ять.

Обґрунтування фронтенд-стеку: React, TypeScript, Vite

При виборі клієнтського фреймворку порівнювалися три найпопулярніші рішення: Angular, Vue.js та React.js.

Angular є занадто складним інструментом для проектів такого рівня. Він орієнтований на великі корпоративні системи, вимагає суворого дотримання своєї структури і завантажує важкі JS-бандли на пристрій користувача, що негативно впливає на швидкість роботи інтерфейсу.

Vue.js є легшим, але його екосистема менш стандартизована порівняно з React.

React.js було обрано через його компонентний підхід, який дозволяє розбити інтерфейс на незалежні блоки (картки, таблиці, графіки) та використовувати їх повторно. Декларативний підхід та використання Virtual DOM мінімізують кількість прямих звернень до реального DOM дерева браузера, що прискорює оновлення інтерфейсу при зміні фінансового стану.

Для підвищення надійності коду було вирішено використовувати TypeScript замість чистого JavaScript. У фінансових додатках помилка в типі даних (наприклад, додавання рядка замість числа) може повністю зламати розрахунок балансу. TypeScript надає строгу типізацію на етапі компіляції, підказує структуру об'єктів (інтерфейси транзакцій та категорій) та допомагає уникати помилок undefined під час роботи додатка.

Для збірки проекту замість класичного Webpack обрано інструмент Vite. Він використовує сучасні ES-модулі браузера та швидкодіючий компілятор Esbuild на мові Go. Це забезпечує миттєвий запуск сервера розробки та гарячу

заміну модулів (HMR) при зміні коду, що значно прискорює процес програмування.

Обґрунтування бекенд-стеку: Node.js, Express, Sequelize

Серверна частина реалізована на платформі Node.js. Використання JavaScript на бекенді та фронтенді спрощує розробку, оскільки дозволяє використовувати однакові типи даних та моделі на обох ланках системи. Асинхронна модель обробки подій у Node.js (Event Loop) ідеально підходить для веб-додатків із великою кількістю дрібних I/O запитів (запис витрат, перевірка лімітів), оскільки сервер не блокує потік при очікуванні відповіді від бази даних.

Express.js обрано як мінімалістичний та швидкий фреймворк для створення REST API. Він не нав'язує жорсткої архітектури, дозволяє легко налаштовувати маршрутизацію запитів та підключати middleware (наприклад, для перевірки JWT-токенів перед доступом до маршрутів).

Для взаємодії з базою даних використовується Sequelize ORM Написання прямих SQL-запитів у коді бэкенду часто призводить до помилок та вразливостей (SQL-ін'єкцій). Sequelize дозволяє описати схему бази даних у вигляді JavaScript-класів (моделей) та працювати з даними через готові методи. Також ORM надає зручну систему міграцій для відстеження та оновлення структури таблиць у процесі розробки.

Обґрунтування вибору СУБД PostgreSQL

Для збереження даних додатку обрано реляційну систему керування базами даних PostgreSQL. На відміну від NoSQL рішень (наприклад, MongoDB), PostgreSQL гарантує повну підтримку транзакційності та вимог ACID (атомарність, узгодженість, ізольованість, довговічність). Це критично важливо для фінансових додатків: якщо під час створення транзакції або перерахунку ліміту станеться збій, система скасує всю операцію (rollback), не допустивши збереження некоректних або часткових даних.

Крім того, PostgreSQL має високу продуктивність при виконанні складних аналітичних запитів із групуванням (SUM, AVERAGE, COUNT з

використанням GROUP BY), які використовуються для формування звітів. Також СУБД підтримує тип даних UUID на рівні ядра, що спрощує генерацію унікальних ключів, та надає можливість роботи з неструктурованими даними у форматі JSONB, якщо у майбутньому виникне потреба розширити профіль користувача.

1.5 Аналіз вимог безпеки та захисту персональних даних у фінансових системах

Розробка веб-додатку для управління особистими фінансами вимагає особливої уваги до безпеки даних, оскільки користувачі довіряють системі конфіденційну інформацію про свої доходи, витрати, фінансові цілі та структуру споживання. Компрометація цих даних може призвести до порушення приватності, фінансового шантажу або цільового фішингу. Тому на етапі аналізу та проєктування системи необхідно чітко визначити нормативні та технічні вимоги до захисту інформації.

З юридичної точки зору, веб-додаток повинен відповідати вимогам законодавства у сфері захисту персональних даних. В Україні базовим документом є Закон України «Про захист персональних даних», який регулює правові відносини, пов’язані з обробкою та захистом персональних даних, і спрямований на захист основоположних прав і свобод людини. Також, оскільки веб-додаток є доступним у глобальній мережі, доцільно враховувати вимоги Загального регламенту про захист даних Європейського Союзу (GDPR – General Data Protection Regulation), який встановлює суворі правила щодо згоди на обробку даних, права бути забутим (видалення облікового запису) та мінімізації збору даних.

Для побудови надійного захисту розробники веб-додатків орієнтуються на стандарти консорціуму OWASP [13] (Open Web Application Security Project). Зокрема, документ OWASP Top 10 описує найбільш критичні загрози безпеці

веб-додатків. До основних ризиків, що мають пряме відношення до нашого додатка, належать:

Недостатня автентифікація та авторизація (Broken Authentication): виникає при слабкому захисті сесій користувачів. Зловмисник може перехопити ідентифікатор сесії або виконати атаку методом перебору паролів. Для запобігання цьому необхідно використовувати криптографічно стійкі алгоритми авторизації та шифрування сесійних токенів.

Небезпечна пряма посилання на об'єкти (IDOR – Insecure Direct Object Reference): ситуація, коли користувач може отримати доступ до даних іншого користувача, просто змінивши ідентифікатор (наприклад, ID транзакції чи категорію) у запиті або URL. Запобіганням є обов'язкова перевірка прав власності на рівні бізнес-логіки сервера для кожної операції з даними.

Впровадження шкідливого коду (Injection): включає SQL-ін'єкції, коли зловмисник впроваджує шкідливі SQL-команди у форми введення з метою зчитування чи знищення бази даних. Захистом є використання параметризованих запитів (Prepared Statements) та ORM-технологій.

Міжсайтовий скриптинг (XSS): впровадження шкідливого Javascript-коду в сторінки, які відображаються іншим користувачам. Для захисту необхідно проводити обов'язкове екранування всіх вхідних даних на клієнтській та серверній сторонах.

На основі проведеного аналізу було сформульовано такі технічні вимоги до безпеки додатка:

1. Використання протоколу HTTPS з шифруванням TLS для захисту трафіку від перехоплення в мережі.
2. Зберігання паролів у базі даних виключно в зашифрованому (хешованому) вигляді з використанням випадкової солі.
3. Реалізація авторизації запитів на основі Stateless-токенів JWT, підписаних секретним ключем сервера.
4. Забезпечення логічного розподілу даних у базі даних (фільтрація всіх транзакцій за ID авторизованого користувача).

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА БАЗИ ДАНИХ

2.1 Проєктування поведінки системи

Для формалізації вимог до веб-додатку та детального опису його взаємодії з користувачем на етапі проектування застосовано методологію об'єктно-орієнтованого аналізу. Основним інструментом моделювання поведінки системи обрано уніфіковану мову моделювання UML (Unified Modeling Language), зокрема діаграму прецедентів (Use Case Diagram). Це дозволяє чітко розмежувати межі проектованої системи, визначити склад її користувачів та встановити зв'язки між функціональними блоками.

Головними суб'єктами (акторами), що взаємодіють із системою, є:

- **Користувач** — фізична особа, яка пройшла процедуру автентифікації та використовує веб-додаток для контролю власних коштів. Він ініціює більшість прецедентів: реєструється в системі, створює та редагує категорії, додає інформацію про доходи та витрати, встановлює бюджетні обмеження та переглядає згенеровані системою аналітичні звіти.
- **Серверна система** (або бекенд-модуль автентифікації та бізнес-логіки) — допоміжний актор, який виконує автоматичні фонові операції. Він відповідає за перевірку валідності токенів безпеки, автоматичний перерахунок залишків лімітів під час додавання кожної транзакції та негайне генерування сповіщень у разі перевищення встановленого порогу витрат.

Діаграма прецедентів використання системи (Рис.2.1) відображає основні сценарії взаємодії акторів із веб-додатком. Вона демонструє взаємозв'язок між функціями управління обліковим записом, транзакціями, категоріями, бюджетами та модулем сповіщень.

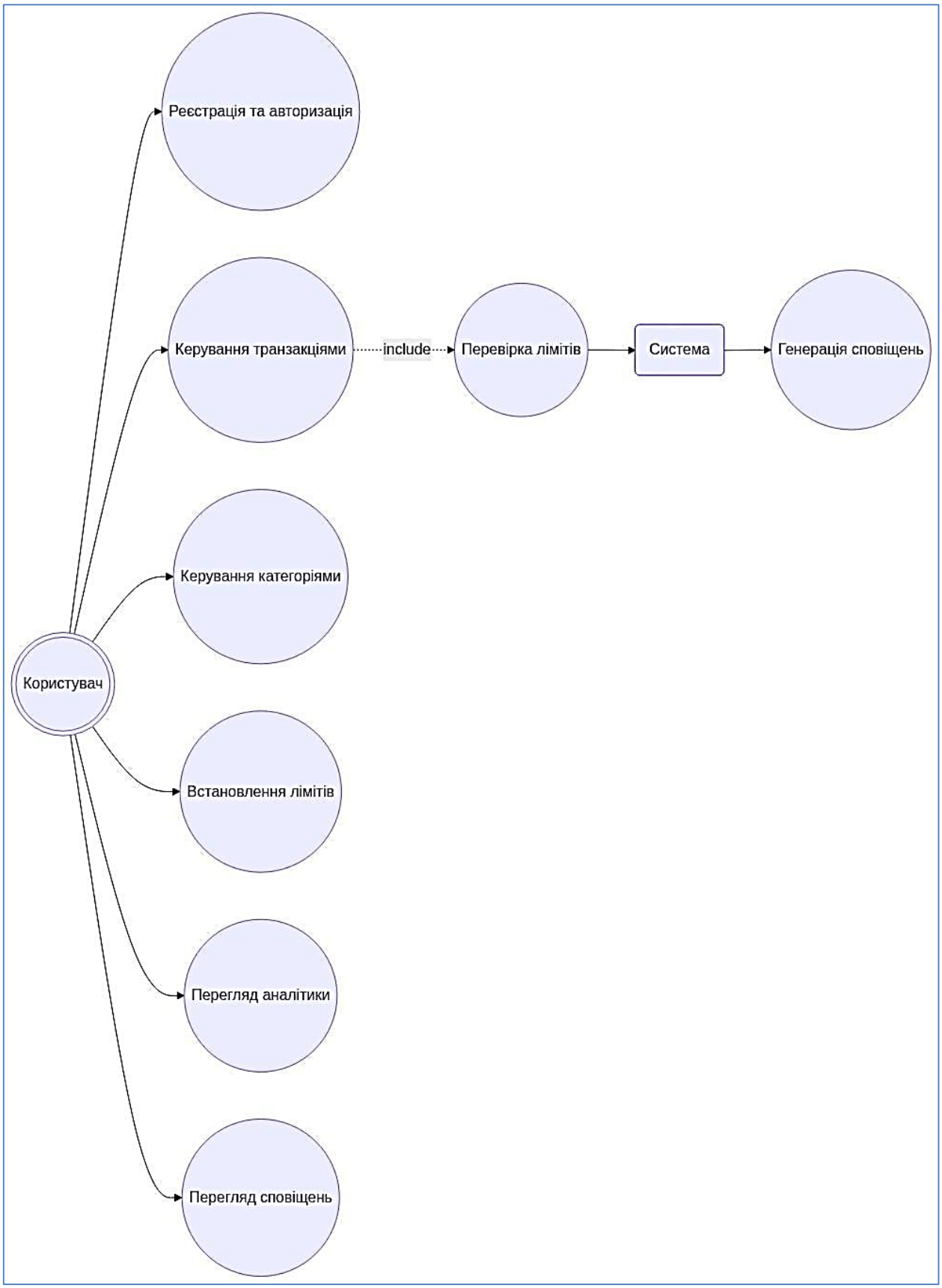


Рисунок *2.1 — Діаграма прецедентів використання системи*

Для детального розуміння логіки роботи системи необхідно описати сценарії виконання основних прецедентів, оскільки вони визначають алгоритмічну основу для подальшої програмної реалізації додатку.

2.1.1 Сценарій прецеденту «Реєстрація нового користувача»

Актори: Користувач, Серверна система.

Мета: створення нового облікового запису користувача у базі даних для надання доступу до функціоналу додатку.

Передумови: користувач відкрив сторінку реєстрації у веб-додатку, його пристрій має підключення до мережі Інтернет.

Тригер: натискання кнопки «Зареєструватися» після заповнення форми.

Схематично процес взаємодії компонентів системи під час реєстрації нового користувача представлено на діаграмі послідовності на рисунку 2.2 (UML Sequence Diagram).

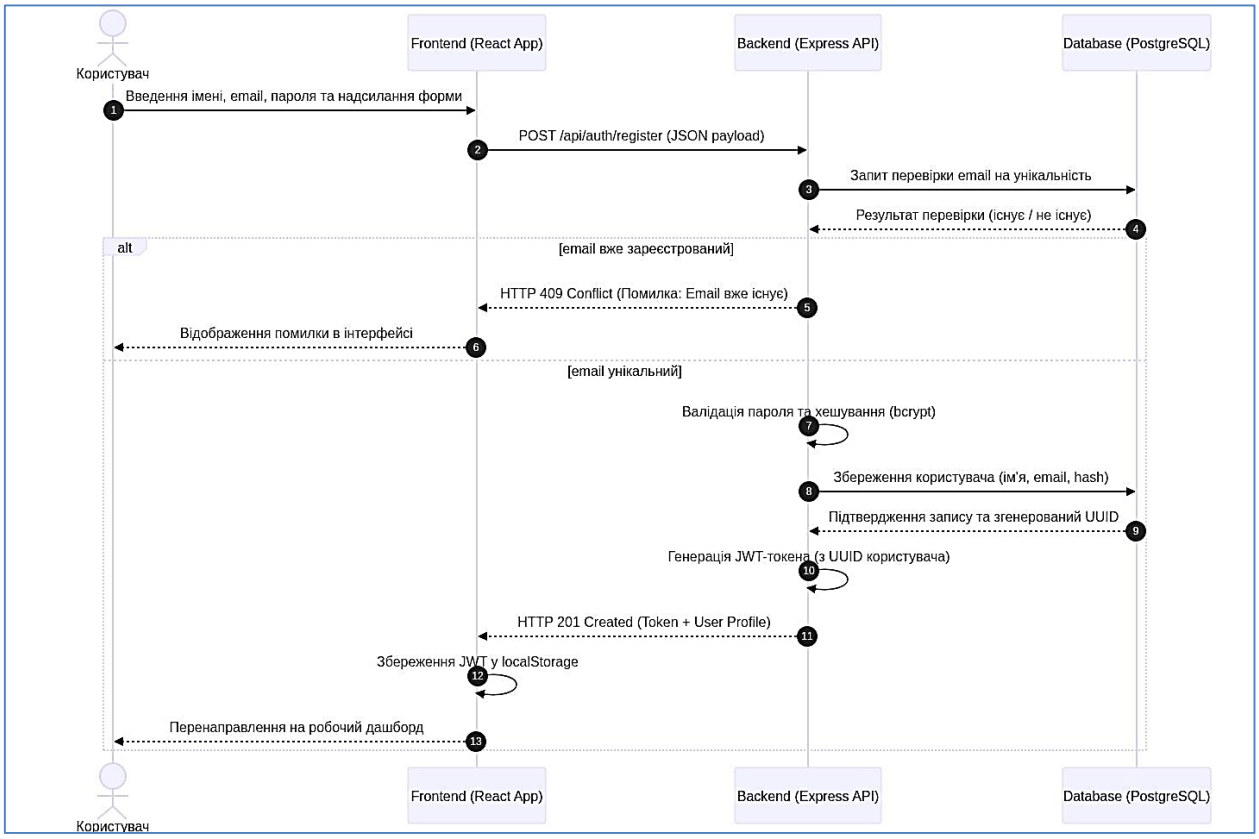


Рисунок 2.2 — *Діаграма послідовності процесу реєстрації користувача*

Основний сценарій:

1. Користувач вводить своє ім'я, унікальну адресу електронної пошти та пароль у відповідні поля форми реєстрації.
2. Користувач надсилає запит на сервер.
3. Серверна система перевіряє унікальність адреси електронної пошти у базі даних.
4. Серверна система валідує пароль на відповідність вимогам безпеки (мінімальна довжина 8 символів, наявність літер та цифр).
5. Серверна система здійснює одностороннє хешування пароля за допомогою алгоритму bcrypt із використанням випадкової солі.
6. Серверна система створює новий запис у таблиці користувачів базі даних із присвоєнням унікального ідентифікатора UUID.
7. Серверна система генерує для користувача набір стандартних (системних) категорій витрат та доходів (наприклад, «Продукти», «Транспорт», «Комунальні послуги», «Заробітна плата»).
8. Система повертає клієнту успішний статус-код та перенаправляє його на головну сторінку авторизації.

Альтернативні сценарії:

- Адреса електронної пошти вже існує: system перериває процес реєстрації, повертає помилку «Користувач із такою електронною поштою вже зареєстрований» та підсвічує відповідне поле інтерфейсу червоним кольором.
- Введені дані не пройшли валідацію: якщо поле пошти порожнє або пароль занадто простий, сервер повертає помилку 400 Bad Request, запис у базу даних не вноситься, користувачеві відображається повідомлення про помилку валідації.

Післяумови: у базі даних успішно створено новий об'єкт користувача із захешованим паролем та згенеровано базові категорії для початку роботи.

2.1.2 Сценарій прецеденту «Додавання фінансової транзакції»

Актори: Користувач, Серверна система.

Мета: фіксація нової фінансової операції (доходу або витрати) в історії обліку з прив'язкою до конкретної категорії.

Передумови: користувач успішно авторизований у системі, має активний JWT-токен у заголовках запитів.

Тригер: натискання кнопки «Зберегти транзакцію» у формі додавання операції.

Взаємодія компонентів та послідовність викликів при створенні нової операції представлена на відповідній діаграмі послідовності (Рис.2.3).

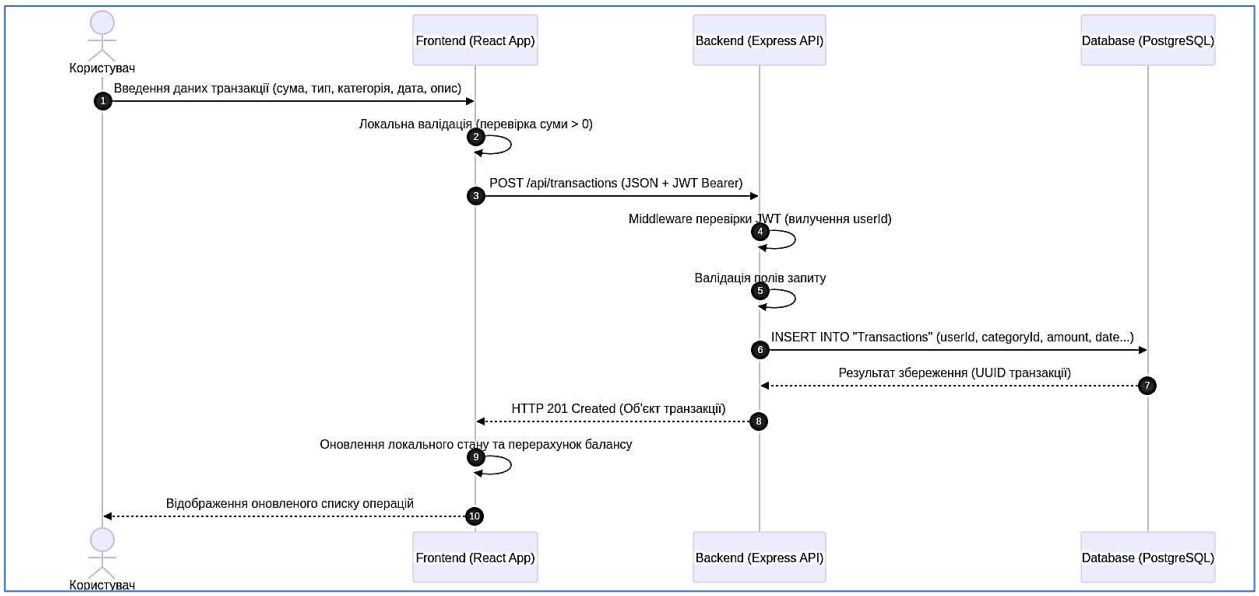


Рисунок *2.3 — Діаграма послідовності процесу додавання фінансової транзакції*

Основний сценарій:

1. Користувач відкриває форму додавання транзакції на дашборді або у списку операцій.

2. Користувач обирає тип операції (дохід чи витрата), категорію зі списку доступних, вводить суму, дату здійснення операції та необов'язковий текстовий опис.
3. Користувач натискає кнопку збереження, надсилаючи HTTP-запит POST на ендпоінт сервера.
4. Серверна система перевіряє JWT-токен у заголовку Authorization, вилучає з нього ідентифікатор користувача та перевіряє його права доступу.
5. Серверна система перевіряє суму транзакції на коректність (числове значення має бути більшим за нуль).
6. Серверна система записує нову транзакцію до таблиці бази даних із автоматичним присвоєнням UUID.
7. Серверна система запускає асинхронний процес перевірки лімітів бюджету для категорії, до якої належить додана транзакція (якщо тип операції - витрата).
8. Система оновлює загальний баланс користувача на клієнтській частині додатку та відображає додану транзакцію на початку списку останніх операцій.

Альтернативні сценарії:

- Недійсний токен авторизації: сервер повертає статус-код 401 Unauthorized, користувач автоматично розлогінюється, а інтерфейс перенаправляє його на сторінку входу.
- Помилка валідації суми: якщо введено нечислове значення або від'ємне число, сервер повертає помилку 400 Bad Request, операція відхиляється, користувач бачить попередження про некоректний формат суми.

Післяумови: транзакція збережена в базі даних, баланс користувача перераховано, запущено тригер перевірки лімітів.

2.1.3 Сценарій прецеденту «Контроль та перевищення ліміту бюджету»

Актори: Серверна система, Користувач.

Мета: автоматичне виявлення перевищення встановленого користувачем ліміту витрат на категорію та створення попереджувального сповіщення.

Передумови: у системі додано нову транзакцію витрати, користувач раніше встановив ліміт бюджету для цієї категорії на поточний період.

Тригер: успішне додавання нової транзакції типу «витрата».

Алгоритм та логіка перевірки поточного стану витрат користувача відносно встановленого ліміту бюджету схематично представлена на діаграмі послідовності (Рис.2.4).

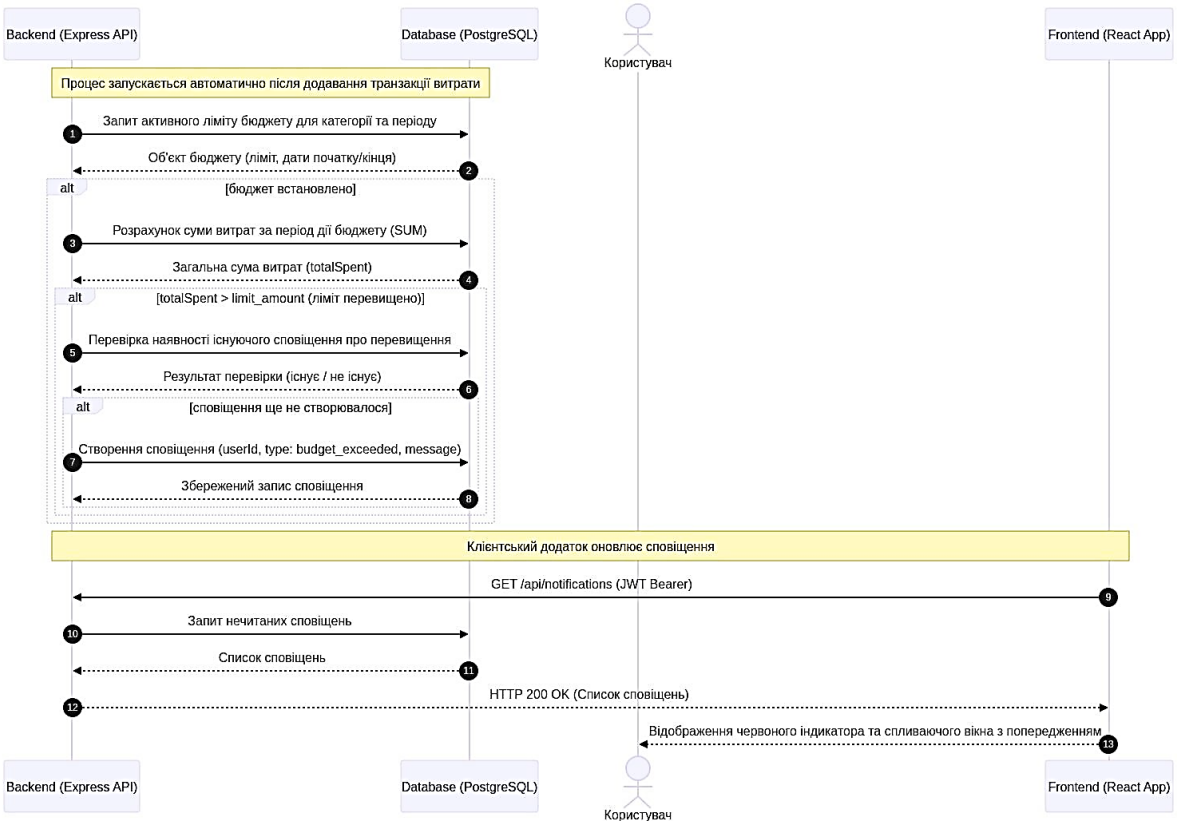


Рисунок 2.4 — *Діаграма послідовності процесу контролю лімітів бюджету*

Основний сценарій:

1. Серверна система шукає у базі даних активний запис ліміту бюджету для цього користувача та категорії, дата транзакції має входити в часовий діапазон дії бюджету.

Якщо бюджет знайдено, сервер виконує агрегаційний SQL-запит для розрахунку суми всіх витрат користувача у цій категорії за період дії бюджету.

2. Серверна система порівнює отриману суму витрат із лімітом, встановленим у налаштуваннях бюджету.

Якщо сума витрат перевищує встановлений ліміт, сервер розраховує точну суму перевищення.

3. Серверна система перевіряє, чи не було раніше створено аналогічне сповіщення для цього періоду бюджету, щоб уникнету спаму та дублювання інформації.

Якщо сповіщення немає, сервер створює новий запис у таблиці сповіщень із прапорцем «не прочитано» та додає опис перевищення.

4. Під час наступного опитування або оновлення інтерфейсу користувачем клієнтський додаток отримує список сповіщень і відображає яскравий індикатор (badge) на іконці сповіщень дашборду.

Альтернативні сценарії:

- Бюджет не встановлено: якщо для цієї категорії немає обмежень, сервер завершує процес без виконання додаткових розрахунків та створення сповіщень.
- Витрати не перевищують ліміт: якщо розрахована сума менша або дорівнює ліміту, сервер фіксує безпечний стан бюджету та завершує сценарій.

Післяумови: у разі перевищення ліміту в базі даних з'являється нове неактивне сповіщення, яке стає доступним користувачеві на клієнтській стороні додатка.

Описані сценарії прецедентів повністю покривають життєвий цикл взаємодії користувача з додатком на рівні логіки процесів. Вони слугують

основою для розробки обробників маршрутів на сервері та організації стану інтерфейсу на фронтенді.

2.2 Архітектурне проєктування веб-додатку

Проєтування архітектури веб-додатку базується на принципах модульності, масштабованості, безпеки та слабкої зв'язності компонентів. Для забезпечення високої швидкості розробки, спрощення підтримки та можливості незалежного масштабування окремих частин системи було обрано класичну триланкову архітектуру. Цей архітектурний шаблон передбачає чіткий поділ системи на три взаємодіючі рівні: рівень представлення (Presentation Layer), рівень бізнес-логіки (Business Logic Layer) та рівень даних (Data Access Layer).

Кожен рівень виконує строго визначений набір функцій та ізольований від внутрішньої реалізації інших рівнів, взаємодіючи з ними лише через стандартизовані інтерфейси.

Рівень представлення реалізований у вигляді SPA додатку з використанням бібліотеки React та мови TypeScript. Головне завдання цього рівня — забезпечення інтерактивного та адаптивного користувацького інтерфейсу, валідація введених користувачем даних на клієнтській стороні, візуалізація аналітичних звітів за допомогою динамічних SVG-графіків, а також комунікація з сервером застосунків через HTTP-запити. Клієнтська частина повністю виконується у веб-браузері користувача на персональному комп'ютері або мобільному пристрої, що мінімізує навантаження на серверну інфраструктуру. Доступ до статичних файлів інтерфейсу надається через сервери веб-хостингу.

Рівень бізнес-логіки є ядром системи і реалізований як веб-сервер на базі програмної платформи Node.js та фреймворку Express.js. Серверна частина відповідає за обробку REST API запитів від клієнтського додатка, реалізацію

бізнес-правил (таких як розрахунок залишків лімітів, перевірка перевищення бюджетів, генерація сповіщень), автентифікацію та авторизацію користувачів за допомогою JWT-токенів, а також взаємодію з базою даних через Sequelize ORM. Сервер застосунків обробляє вхідні запити асинхронно, забезпечуючи високу швидкість виконання операцій та можливість горизонтального масштабування.

Рівень даних представлений реляційною системою керування базами даних PostgreSQL. Цей рівень забезпечує надійне, транзакційне та безпечне зберігання інформації про користувачів, транзакції, категорії, бюджети та сповіщення. Доступ до бази даних з боку сервера застосунків здійснюється через захищене з'єднання з автентифікацією.

Схематично логічна взаємодія між основними компонентами системи представлена на діаграмі компонентів (Рис. 2.5).



Рисунок 2.5 — *Діаграма компонентів веб-додатку*

Взаємодія між клієнтською та серверною частинами веб-додатку побудована на базі протоколу HTTP/HTTPS та архітектурного стилю REST. Бекенд функціонує у режимі stateless (без збереження стану сесії на сервері), що означає, що кожен запит від клієнта повинен містити всю необхідну інформацію для його ідентифікації та авторизації.

Для реалізації безпечної взаємодії використовується механізм авторизації на базі JSON Web Tokens (JWT). Процес авторизації та обміну даними складається з таких кроків:

1. Користувач вводить облікові дані (email та пароль) у форму входу на клієнті.

2. Клієнтський додаток надсилає POST-запит на серверний маршрут /api/auth/login.

3. Сервер перевіряє наявність користувача у базі даних та порівнює хеш пароля.

4. У разі успішної перевірки server генерує JWT-токен, підписаний секретним ключем сервера (JWT_SECRET). Токен містить унікальний ідентифікатор користувача (userId) та термін дії.

5. Сервер повертає токен клієнту у форматі JSON разом із базовим профілем користувача.

6. Клієнтська частина зберігає отриманий JWT-токен у локальному сховищі браузера (localStorage).

При кожному наступному запиті до захищених маршрутів API (наприклад, додавання транзакції чи отримання звітів) клієнт автоматично прикріплює токен до заголовка запиту у форматі Authorization: Bearer <token>. Серверний проміжний обробник (middleware) перехоплює запит, перевіряє підпис токена за допомогою секретного ключа та, у разі успіху, вилучає userId для виконання бізнес-операцій.

Така архітектурна модель забезпечує високий рівень безпеки даних, оскільки секретний ключ ніколи не передається клієнту, а сам токен захищений від підробки цифровим підписом.

2.3 Проектування бази даних

Проєтування бази даних є фундаментальним етапом розробки веб-додатку, оскільки від правильності моделювання даних залежить цілісність, консистентність, безпека та швидкодія всієї системи. Процес проєктування складається з розробки концептуальної схеми даних, створення ER-діаграми та детального опису фізичної структури таблиць реляційної бази даних.

Для побудови інформаційної моделі системи було обрано реляційну модель даних. Вона дозволяє представити сутності предметної області у

вигляді таблиць, пов’язаних між собою за допомогою первинних та зовнішніх ключів. Основною особливістю проєктування бази даних для нашого веб-додатку є використання унікальних ідентифікаторів типу UUID як первинних ключів для всіх сутностей. Це гарантує унікальність записів у масштабах будь-якої розподіленої системи, виключає можливість перебору ідентифікаторів зловмисниками через URL та запобігає колізіям під час синхронізації або імпорту даних.

На основі аналізу функціональних вимог до веб-додатку виділено п’ять основних сутностей:

Користувачі (Users) - містить облікові дані осіб, які зареєстровані у веб-додатку.

Категорії (Categories) - класифікатор фінансових операцій користувача. Категорії можуть бути системними (доступними всім за замовчуванням) або персональними (створеними конкретним користувачем).

Транзакції (Transactions) - детальні записи про кожну фінансову операцію (дохід або витрату).

Бюджети (Budgets) - планові обмеження витрат користувача на певний період часу (тиждень, місяць) для конкретної категорії або загального бюджету.

Сповіщення (Notifications) - системні повідомлення користувачу про перевищення лімітів бюджету або інші важливі події.

Логічна структура бази даних та зв'язки між сутностями представлені на діаграмі «сутність-зв'язок» на рисунку 2.6 (ERD).

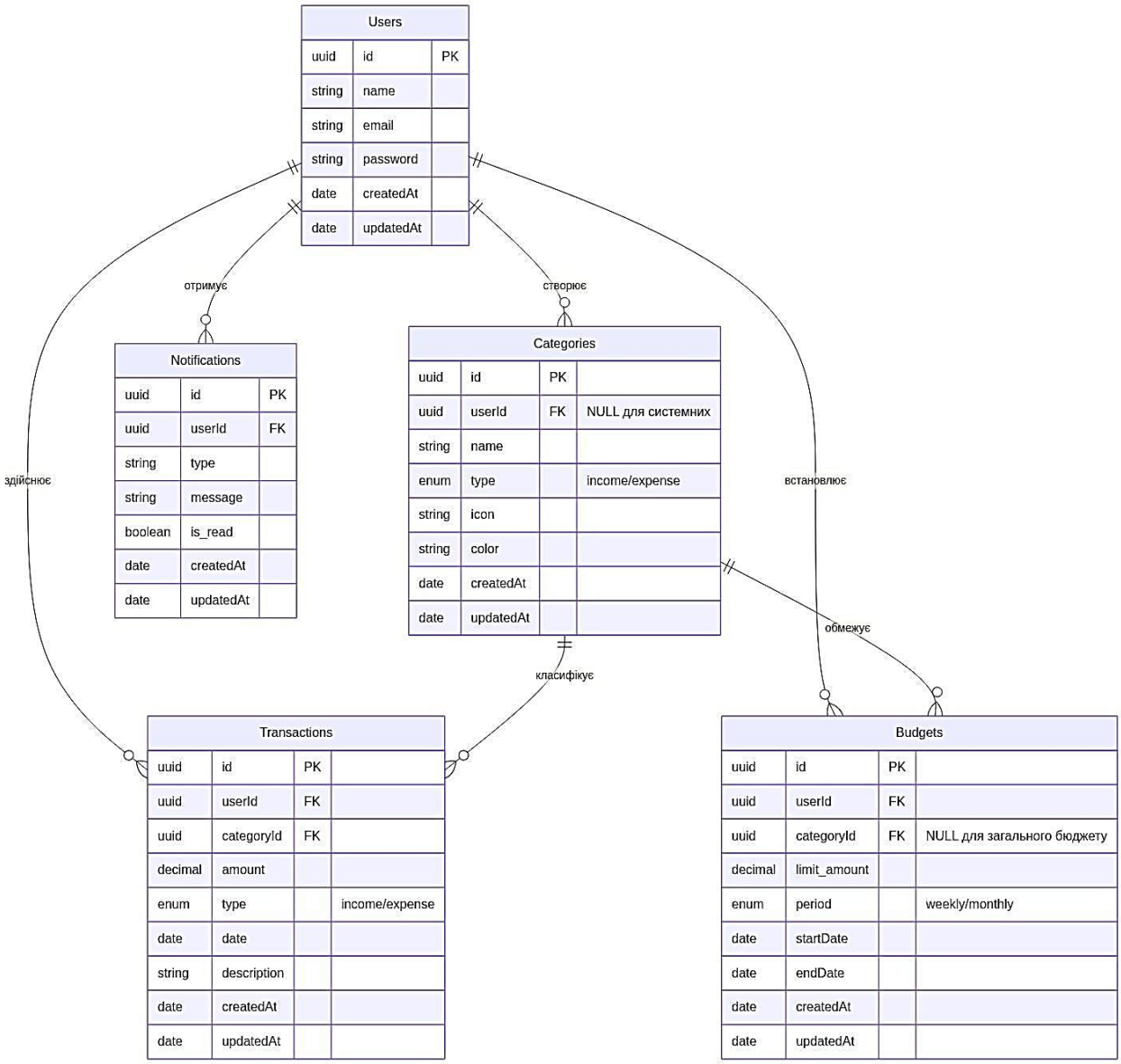


Рисунок 2.6 — *ER-діаграма бази даних веб-додатку FinanceFlow*

Зв’язки між сутностями характеризуються такими правилами цілісності:

1. Один користувач може створювати багато категорій (зв'язок «один-до-багатьох» між Users та Categories). При цьому поле userId для системних категорій має значення NULL.
2. Один користувач здійснює багато транзакцій (зв'язок «один-до-багатьох» між Users та Transactions).
3. Один користувач встановлює багато бюджетів (зв'язок «один-до-багатьох» між Users та Budgets).

4. Один користувач отримує багато сповіщень (зв'язок «один-до-багатьох» між Users та Notifications).
5. Кожна транзакція обов’язково належить до однієї категорії (зв’язок «один-до-багатьох» між Categories та Transactions).
6. Кожне бюджетне обмеження може стосуватися певної категорії (зв’язок «один-до-багатьох» між Categories та Budgets). Якщо бюджет є загальним, поле categoryId приймає значення NULL.

Фізична структура бази даних реалізована за допомогою СУБД PostgreSQL. Така схема бази даних відповідає вимогам третьої нормальної форми (3NF), оскільки всі неключові атрибути таблиць залежать виключно від первинних ключів і не містять транзитивних залежностей. Це запобігає аномаліям вставки, оновлення та видалення даних.

2.4. Розробка специфікації REST API ендпоінтів

Для організації взаємодії між клієнтською частиною (React SPA) та сервером застосунків (Express.js) розроблено програмний інтерфейс REST API. Вся взаємодія будується на обміні даними у форматі JSON через стандартні HTTP-методи (GET, POST, PUT, DELETE), що відповідають операціям CRUD

Всі захищені маршрути API вимагають передачі валідного JWT-токена в HTTP-заголовку запиту: Authorization: Bearer <JWT_TOKEN>. Якщо токен відсутній або термін його дії закінчився, server повертає статус HTTP 401 Unauthorized. Базова URL-адреса всіх маршрутів: /api.

Нижче наведено структуру розроблених URL-маршрутів REST API:

1. **Автентифікація та авторизація (/api/auth)**
   - POST /register - реєстрація нового облікового запису користувача.
   - POST /login - авторизація користувача з поверненням JWT-токена та профілю.

- GET /me - перевірка поточної сесії та отримання даних профілю користувача.

2. Категорії фінансових операцій (/api/categories)
   - GET / - отримання списку системних та власних категорій користувача.
   - POST / - створення нової персональної категорії користувача.
   - DELETE /:id - видалення персональної категорії користувача за її ідентифікатором.
3. Фінансові транзакції (/api/transactions)
   - GET / - отримання списку транзакцій користувача з можливістю фільтрації.
   - POST / - створення нової транзакції (витрати/доходу) з автоматичною ініціацією перевірки бюджетів.
   - PUT /:id - редагування деталей існуючої транзакції.
   - DELETE /:id - видалення транзакції за її ідентифікатором.
4. Планові ліміти бюджетів (/api/budgets)
   - GET / - отримання лімітів бюджетів із розрахованими сумами фактичних витрат.
   - POST / - встановлення нового ліміту бюджету для обраної категорії або загального бюджету.
   - DELETE /:id - видалення ліміту бюджету за його ідентифікатором.
5. Аналітичні звіти (/api/analytics)
   - GET /category-expenses - отримання агрегованих сум витрат по категоріях для кругової діаграми.
   - GET /monthly-trend - отримання трендів доходів та витрат за останні 6 місяців для стовпчикового графіка.
6. Системні сповіщення (/api/notifications)
   - GET / - отримання списку сповіщень про перевищення бюджету поточного користувача.
   - PUT /:id/read - позначення сповіщення як прочитаного.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ

3.1 Розробка серверної частини (Backend)

Серверна частина веб-додатку «FinanceFlow» виконує роль ядра бізнес-логіки, обробляє REST API запити, взаємодіє з базою даних PostgreSQL та реалізує заходи безпеки. Бекенд побудовано на платформі Node.js із використанням фреймворку Express.js та Sequelize ORM (Object-Relational Mapping).

3.1.1 Налаштування середовища та підключення до бази даних

Для організації доступу до бази даних через Sequelize ORM конфігураційні параметри для підключення винесено у файл config/config.js (Лістинг3.1). Це дозволяє гнучко перемикатися між локальним середовищем розробки (development) та хмарною версією (production) через системні змінні оточення (файл .env).

Лістинг 3.1 — *config.js*

```
// config/config.js
require('dotenv').config();

module.exports = {
  development: {
    username: process.env.DB_USER || 'postgres',
    password: process.env.DB_PASSWORD || 'postgres',
    database: process.env.DB_NAME ||
'personal_finance_app',
    host: process.env.DB_HOST || '127.0.0.1',
    port: process.env.DB_PORT || 5432,
    dialect: 'postgres',
```

```
    logging: false
  },
  production: {
    use_env_variable: 'DATABASE_URL',
    dialect: 'postgres',
    dialectOptions: {
      ssl: {
        require: true,
        rejectUnauthorized: false
      }
    }
  }
};
```

Для ініціалізації Sequelize та підключення моделей у проекті створено єдину точку входу models/index.js (Лістинг3.2). З метою сумісності із сервером безсерверних функцій (Serverless) платформи Vercel, яка вимагає статичного аналізу залежностей, динамічне сканування каталогу замінено на явний імпорт моделей:

Лістинг 3.2 — *index.js*

```
// models/index.js (Фрагмент ініціалізації)
const Sequelize = require('sequelize');
const env = process.env.NODE_ENV || 'development';
const config = require(__dirname +
'/../config/config.js')[env];
const db = {};

let sequelize;
if (config.use_env_variable) {
```

```
  sequelize = new
Sequelize(process.env[config.use_env_variable], config);
} else {
  sequelize = new Sequelize(config.database,
config.username, config.password, config);
}

const models = [
  require('./user'),
  require('./category'),
  require('./transaction'),
  require('./budget'),
  require('./notification')
];

models.forEach(modelDef => {
  const model = modelDef(sequelize, Sequelize.DataTypes);
  db[model.name] = model;
});

Object.keys(db).forEach(modelName => {
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});

db.sequelize = sequelize;
db.Sequelize = Sequelize;

module.exports = db;
```

3.1.2 Реалізація механізму автентифікації користувачів

Безпека особистих фінансових даних користувачів забезпечується хешуванням паролів за допомогою алгоритму bcryptjs та авторизацією запитів на основі токенів JSON Web Token (JWT).

При реєстрації користувача його пароль хешується перед збереженням у базі даних (у поле password_hash таблиці Users). Це показується в лістингу 3.3:

Лістинг 3.3 — *Хешування пароля при реєстрації*

```
// controllers/authController.js (Фрагмент створення
користувача)
const bcrypt = require('bcryptjs');
const { User } = require('../models');

const salt = await bcrypt.genSalt(10);
const password_hash = await bcrypt.hash(password, salt);

const user = await User.create({
  name,
  email,
  password_hash
});
```

У разі успішної автентифікації (після звірки введеного пароля із збереженим хешем за допомогою функції bcrypt.compare) генерується JWT-токен із терміном дії 7 днів.

Для обмеження доступу неавторизованих користувачів до API-маршрутів розроблено проміжний обробник (middleware) middlewares/auth.js (Лістинг3.4), який декодує токен, перевіряє його валідність та підвантажує дані користувача
з бази даних:

Лістинг 3.4 — *auth.js*

```
// middlewares/auth.js
const jwt = require('jsonwebtoken');
const { User } = require('../models');

module.exports = async (req, res, next) => {
  try {
    const authHeader = req.headers.authorization;
    if (!authHeader || !authHeader.startsWith('Bearer ')) {
      return res.status(401).json({ message: 'Авторизація
обов\'язкова (відсутній токен)' });
    }

    const token = authHeader.split(' ')[1];
    if (!token) {
      return res.status(401).json({ message: 'Невірний
формат токена' });
    }

    const decoded = jwt.verify(token,
process.env.JWT_SECRET ||
'supersecretjwtkeyforpersonalfinanceapp2026');

    const user = await User.findByPk(decoded.id);
    if (!user) {
      return res.status(401).json({ message: 'Користувача
не знайдено або токен недійсний' });
    }

    req.user = user; // Збереження об'єкта користувача у
запиті
    next();
```

```
  } catch (error) {
    if (error.name === 'TokenExpiredError') {
      return res.status(401).json({ message: 'Термін дії
токена закінчився' });
    }
    return res.status(401).json({ message: 'Невірний або
зіпсований токен авторизації' });
  }
};
```

3.1.3 Логіка автоматичного перерахунку лімітів та контролю бюджетів

Однією з ключових функцій бекенду є автоматичний моніторинг фактичних витрат відносно встановлених лімітів бюджету. Ця логіка виконується як асинхронна операція відразу після збереження нової транзакції з типом expense (витрата).

Для реалізації перевірки лімітів у контролері транзакцій controllers/transactionController.js створено допоміжну функцію checkBudgetExceeded (Лістинг3.5), яка перевіряє як ліміти по конкретній категорії, так і загальний ліміт витрат користувача за вказаний період:

Лістинг 3.5 — *Функція checkBudgetExceeded*

```
// Допоміжна функція перевірки перевищення бюджету const
checkBudgetExceeded = async (userId, categoryId,
transactionDate) => { try { const now = new Date();
// 1. Перевірка бюджету по конкретній категорії
if (categoryId) {
  const budget = await Budget.findOne({
   where: {
      userId,
```

```
      categoryId,
      startDate: { [Op.lte]: transactionDate },
      endDate: { [Op.gte]: transactionDate }
    },
    include: [{ model: Category, as: 'category' }]
  });

  if (budget) {
    const transactionsSum = await Transaction.sum('amount',
{
      where: {
        userId,
        categoryId,
        type: 'expense',
        date: {
          [Op.between]: [budget.startDate, budget.endDate]
        }
      }
    }) || 0;

    if (parseFloat(transactionsSum) >
parseFloat(budget.limit_amount)) {
      const diff = (parseFloat(transactionsSum) -
parseFloat(budget.limit_amount)).toFixed(2);
      const categoryName = budget.category ?
budget.category.name : 'Категорію';

      await Notification.create({
        userId,
        type: 'budget_exceeded',
        message: `Увага! Бюджет на категорію
"${categoryName}" перевищено на ${diff} грн. Ліміт:
${budget.limit_amount} грн.`
```

```
      });
    }
  }
}

// 2. Перевірка загального бюджету (де categoryId: null)
const totalBudget = await Budget.findOne({ ...
});

if (totalBudget) { ...
}

} catch (err) {
console.error('Помилка перевірки лімітів бюджету:', err);
}
};
```

Ця допоміжна функція автоматично викликається в методах створення та оновлення транзакцій, якщо операція належить до категорії витрат, що забезпечує миттєву реакцію системи та створення відповідного сповіщення для відображення в інтерфейсі користувача.

3.2 Розробка клієнтської частини

Клієнтська частина веб-додатку спроектована як SPA веб-додаток на базі бібліотеки React, мови TypeScript та інструменту збирання проєктів Vite. Основне завдання фронтенду - забезпечення інтерактивного інтерфейсу користувача, виконання локального стану авторизації, обробка фінансових операцій та відображення кастомних графічних звітів без використання важких зовнішніх бібліотек візуалізації.

3.2.1 Збирання проєкту та конфігурація клієнтського API

Збирання додатку налаштовано у файлі vite.config.ts (Лістинг3.6) з використанням офіційного плагіна @vitejs/plugin-react для підтримки швидкого оновлення модулів (HMR):

Лістинг 3.6 — *vite.config.ts*

```
// vite.config.ts
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vite.dev/config/
export default defineConfig({
  plugins: [react()],
})
```

Для комунікації з серверною частиною розроблено модуль src/utils/api.ts (Лістинг3.7). Він інкапсулює базову адресу ендпоінтів (import.meta.env.VITE_API_URL з фолбеком на порт 5001) та автоматично підставляє JWT-токен із локального сховища localStorage в HTTP-заголовок Authorization:

Лістинг 3.7 — *Модуль комунікації з серверною частиною*

```
const BASE_URL = import.meta.env.VITE_API_URL ||
'http://localhost:5001/api';

const getHeaders = () => {
  const token = localStorage.getItem('token');
  const headers: HeadersInit = {
    'Content-Type': 'application/json'
```

```
  };
  if (token) {
    headers['Authorization'] = `Bearer ${token}`;
  }
  return headers;
};

export const api = {
  get: async (endpoint: string) => {
    const response = await fetch(`${BASE_URL}${endpoint}`,
{
      method: 'GET',
      headers: getHeaders()
    });
    if (!response.ok) {
      const err = await response.json().catch(() => ({}));
      throw new Error(err.message || 'Помилка при
завантаженні даних');
    }
    return response.json();
  },
  post: async (endpoint: string, data: any) => {
    const response = await fetch(`${BASE_URL}${endpoint}`,
{
      method: 'POST',
      headers: getHeaders(),
      body: JSON.stringify(data)
    });
    if (!response.ok) {
      const err = await response.json().catch(() => ({}));
      throw new Error(err.message || 'Помилка при
збереженні даних');
    }
```

```
    return response.json();
  },
  put: async (endpoint: string, data: any) => {
    const response = await fetch(`${BASE_URL}${endpoint}`,
{
      method: 'PUT',
      headers: getHeaders(),
      body: JSON.stringify(data)
    });
    if (!response.ok) {
      const err = await response.json().catch(() => ({}));
      throw new Error(err.message || 'Помилка при оновленні
даних');
    }
    return response.json();
  },
  delete: async (endpoint: string) => {
    const response = await fetch(`${BASE_URL}${endpoint}`,
{
      method: 'DELETE',
      headers: getHeaders()
    });
    if (!response.ok) {
      const err = await response.json().catch(() => ({}));
      throw new Error(err.message || 'Помилка при видаленні
даних');
    }
    return response.json();
  }
};
```

3.2.2 Керування навігацією та станом веб-додатку

У веб-додатку не використовуються сторонні маршрутизатори (такі як react-router-dom). Навігація між основними інтерфейсними екранами реалізована за допомогою умовного рендерингу (conditional rendering)(Лістинг3.8) на базі стану currentView у головному компоненті src/App.tsx:

Лістинг 3.8 — *Функція умовного рендерингу*

```
const renderActiveView = () => {
  switch (currentView) {
    case 'dashboard':
      return (
        <DashboardView
          setView={setView}
          onAddTransactionClick={handleQuickAddClick}
        />
      );
    case 'transactions':
      return (
        <TransactionsView
          showAddModal={showAddTransactionModal}
          setShowAddModal={setShowAddTransactionModal}
        />
      );
    case 'budgets':
      return <BudgetsView />;
    case 'categories':
      return <CategoriesView />;
    case 'analytics':
      return <AnalyticsView />;
    default:
      return <DashboardView setView={setView}
onAddTransactionClick={handleQuickAddClick} />;
  }};
```

Глобальний фінансовий стан користувача зберігається в контексті src/context/FinanceContext.tsx. Він виконує асинхронні запити при зміні стану авторизації, завантажуючи категорії, ліміти, сповіщення та аналітику за допомогою Promise.all. Фрагмент логіки фінансового контексту показано в лістингу 3.9:

Лістинг 3.9 — *Фрагмент логіки фінансового контексту*

```
// src/context/FinanceContext.tsx (Фрагмент ініціалізації
даних)
useEffect(() => {
  if (isAuthenticated) {
    setLoading(true);
    Promise.all([
      fetchCategories(),
      fetchBudgets(),
      fetchNotifications(),
      fetchTransactions(),
      fetchAnalytics()
    ]).finally(() => setLoading(false));
  } else {
    // Очищення стейту при логауті
    setTransactions([]);
    setCategories([]);
    setBudgets([]);
    setNotifications([]);
    setSummary(null);
    setCategoryExpenses([]);
    setMonthlyTrend([]);
  }
}, [isAuthenticated]);
```

3.2.3 Стилізація та візуалізація аналітичних звітів

Для забезпечення високої швидкості роботи та сучасного вигляду розроблено дизайн-систему на базі CSS Custom Properties (змінних) у файлі src/index.css. Фрагмент стилізації карток показано в лістингу 3.10:

Лістинг 3.10 — *Фрагмент стилізації карток*

```
/* src/index.css (Фрагмент стилізації карток) */
.glass-card {
  background: var(--card-bg);
  backdrop-filter: blur(16px);
  -webkit-backdrop-filter: blur(16px);
  border: 1px solid var(--card-border);
  border-radius: var(--border-radius-md);
  padding: 1.5rem;
  box-shadow: 0 8px 32px 0 rgba(0, 0, 0, 0.37);
  transition: border-color var(--transition-normal),
transform var(--transition-normal), box-shadow var(--
transition-normal);
}
```

Графічний аналіз витрат за категоріями у компоненті src/components/AnalyticsView.tsx реалізовано за допомогою векторного формату SVG. Кругова діаграма (Donut Chart) будується за допомогою математичного розрахунку зсуву секторів кола через властивості strokeDasharray та strokeDashoffset (Лістинг 3.11):

Лістинг 3.11 — *Розрахунок Donut Chart*

```
// src/components/AnalyticsView.tsx (Математичний
розрахунок Donut Chart)
```

```
const radius = 35;
const circumference = 2 * Math.PI * radius; // Довжина кола
~219.9
let currentOffset = 0;

// Рендеринг елементів SVG-кола для кожної категорії
витрат:
{categoryExpenses.map((item) => {
  const percent = item.total / totalExpensesSum;
  const strokeLength = percent * circumference;
  const strokeOffset = circumference - strokeLength +
currentOffset;
  currentOffset -= strokeLength;

  return (
    <circle
      key={item.categoryId}
      cx="50"
      cy="50"
      r={radius}
      fill="none"
      stroke={item.color}
      strokeWidth="9"
      strokeDasharray={`${strokeLength} ${circumference}`}
      strokeDashoffset={strokeOffset}
      transform="rotate(-90 50 50)"
      style={{ transition: 'stroke-dasharray 0.5s ease' }}
    />
  );
})}
```

Отримані значення динамічно підставляються в параметри SVG-елементів <circle>, формуючи плавний круговий графік розподілу фінансів. Стовпчиковий графік порівняння доходів та витрат (Monthly Trend) реалізований за допомогою гнучких Flexbox-контейнерів, де висота кожного стовпчика вираховується як відсоток від максимального значення у вибірці, а детальна інформація про транзакції відображається у спливаючих підказках при наведенні курсору.

3.3 Реалізація системи сповіщень та аналітичного модуля

Управління особистими фінансами потребує оперативної реакції на перевищення витрат та якісного візуального аналізу фінансового стану. Для цього в системі реалізовано модуль сповіщень у реальному часі та аналітичний інструментарій агрегації даних.

3.3.1 Функціонування системи сповіщень на стороні клієнта

Система сповіщень працює за принципом зворотного зв'язку на основі подій. Після додавання транзакцій сервер перевіряє ліміти та, у разі перевищення, створює запис у таблиці Notifications. Клієнтська частина через React Context (src/context/FinanceContext.tsx) завантажує цей список при старті або оновленні стану транзакцій (Лістинг 3.12):

Лістинг 3.12 — *Обробка сповіщень*

```
// src/context/FinanceContext.tsx (Клієнтська обробка
сповіщень)
const fetchNotifications = async () => {
  try {
    const data = await api.get('/notifications');
```

```
    setNotifications(data);
  } catch (error) {
    console.error('Помилка при завантаженні сповіщень:',
error);
  }
};
```

Непрочитані сповіщення відображаються на головній панелі (src/components/DashboardView.tsx) у вигляді попереджувальних карток з описом перевищення ліміту. При натисканні на кнопку закриття надсилається запит на оновлення статусу в БД. Фрагмент логіки сповіщень показано в лістингу 3.13:

Лістинг 3.13 — *Фрагмент логіки сповіщень*

```
const markNotificationRead = async (id: string) => {
  try {
    await api.put(`/notifications/${id}/read`, {});
    setNotifications(prev => prev.map(n => n.id === id ? {
...n, is_read: true } : n));
  } catch (err) {
    console.error('Помилка оновлення статусу сповіщення:',
err);
  }
};
```

3.3.2 Агрегація даних на сервері та візуалізація аналітики

Аналітичний модуль виконує групування та підрахунок фінансових показників. На сервері розрахунок витрат за категоріями реалізований за допомогою Sequelize ORM з використанням агрегаційної функції SUM та групування GROUP BY (Лістинг 3.14):

Лістинг 3.14 — *Агрегація витрат по категоріям*

```
const expenses = await Transaction.findAll({
  where: { userId, type: 'expense', date: { [Op.between]:
[startDate, endDate] } },
  attributes: ['categoryId', [sequelize.fn('SUM',
sequelize.col('amount')), 'total']],
  include: [{ model: Category, as: 'category', attributes:
['id', 'name', 'color', 'icon'] }],
  group: ['categoryId', 'category.id', 'category.name',
'category.color', 'category.icon']
});
```

На фронтенді уся аналітика (загальний баланс, витрати по категоріях та місячний тренд) оновлюється паралельно за допомогою асинхронного запиту Promise.all для мінімізації мережевих затримок. Паралельна обробка аналітичних запитів показана в лістингу 3.15:

Лістинг 3.15 — *Паралельна обробка аналітичних запитів*

```
const [sumData, catData, trendData] = await Promise.all([
  api.get(`/analytics/summary${query}`),
  api.get(`/analytics/categories${query}`),
  api.get('/analytics/trend')
]);
```

Візуалізація структури витрат здійснюється у компоненті src/components/AnalyticsView.tsx через інтерактивну кругову діаграму (Donut Chart), яка будується на чистому SVG без важких зовнішніх бібліотек. Сектори

круга формуються динамічним розрахунком значень strokeDasharray та strokeDashoffset на основі відсоткової частки кожної категорії у загальному бюджеті.

3.4 Заходи з безпеки даних та оптимізації продуктивності

Безпека збереження конфіденційної фінансової інформації користувачів є одним із ключових аспектів розробки веб-додатку. У проєкті реалізовано багаторівневий комплексний підхід до захисту даних на всіх етапах взаємодії клієнтської та серверної частин системи.

3.4.1 Криптографічний захист паролів

На рівні збереження даних паролі користувачів захищаються від компрометації за допомогою бібліотеки bcryptjs, яка реалізує алгоритм криптографічного хешування (Лістинг 3.16). Процедура реєстрації передбачає генерацію унікальної випадкової солі (salt) для кожного пароля індивідуально та виконання 10 раундів хешування, що захищає систему від brute-force та rainbow-table атак.

Лістинг 3.16 — *Хешування пароля*

```
// controllers/authController.js (Хешування пароля)
const salt = await bcrypt.genSalt(10);
const password_hash = await bcrypt.hash(password, salt);
```

У базі даних PostgreSQL зберігається виключно незворотний хеш у полі password_hash таблиці Users, а автентифікація під час входу здійснюється через порівняння хешу введеного пароля із записом у базі за допомогою bcrypt.compare.

3.4.2 Авторизація запитів на базі Stateless JWT

Для обмеження доступу неавторизованих користувачів до REST API використовується механізм Stateless JWT. Після успішного входу користувача сервер генерує токен, підписаний секретним ключем JWT_SECRET, який містить унікальний ідентифікатор користувача (id) та термін дії (7 днів). Клієнтська частина зберігає токен у localStorage та додає його до заголовка Authorization: Bearer <token> кожного запиту.

Проміжний обробник middlewares/auth.js на сервері Express перевіряє валідність та цілісність токена (Лістинг 3.17):

Лістинг 3.17 — *Фрагмент валідації токена*

```
const token = authHeader.split(' ')[1];
const decoded = jwt.verify(token, process.env.JWT_SECRET ||
'supersecretjwtkeyforpersonalfinanceapp2026');
const user = await User.findByPk(decoded.id);
if (!user) {
  return res.status(401).json({ message: 'Користувача не
знайдено або токен недійсний' });
}
req.user = user;
```

3.4.3 Захист від SQL-ін'єкцій та ізоляція даних

Захист від атак типу SQL-ін'єкцій (SQL Injection) реалізовано завдяки використанню Sequelize ORM. Об'єктно-реляційне відображення автоматично перетворює об'єктні запити Sequelize на параметризовані SQL-запити до СУБД (Prepared Statements), де значення, введені користувачем, передаються окремо від самого тіла команди.

Для забезпечення логічної ізоляції даних між різними користувачами (захист від атак IDOR), кожна операція вибірки, створення чи видалення

сутностей фільтрується за ідентифікатором авторизованого користувача. Приклад вибірки транзакцій поточного користувача показано в лістингу 3.18:

Лістинг 3.18 — *Приклад вибірки транзакцій поточного користувача*

```
const transactions = await Transaction.findAll({
  where: { userId: req.user.id }
});
```

3.4.4. Мережева безпека та CORS

На транспортному рівні захист даних реалізовано шляхом обов'язкового використання протоколу HTTPS з шифруванням TLS, а також захищеного SSL-підключення до бази даних Neon DB.

Для обмеження несанкціонованих крос-доменних запитів до API налаштовано CORS-політику за допомогою пакета cors, яка динамічно перевіряє походження запиту та дозволяє передачу авторизаційних заголовків (credentials: true). Для запобігання витоку службової інформації всі непередбачені системні помилки обробляються централізовано глобальним мідлваром, який повертає клієнту узагальнене повідомлення зі статус-кодом HTTP 500 без детального стеку помилок.

3.4.5. Запобігання XSS-атакам на стороні клієнта

Для захисту від XSS атаки клієнтський React-додаток використовує вбудовані механізми безпеки. Усі динамічні текстові дані (наприклад, опис транзакції чи назва категорії), що рендеряться в інтерфейсі через фігурні дужки {} у JSX, автоматично екрануються бібліотекою React перед вставкою в DOM. Небезпечні символи (такі як <, >, &, ", ') перетворюються на відповідні безпечні HTML-сутності, що робить неможливим виконання шкідливого Javascript-коду, введеного користувачем.

3.4.6. Оптимізація продуктивності бази даних

Для забезпечення високої швидкодії веб-додатку за умов зростання обсягів даних проведено оптимізацію реляційної СУБД PostgreSQL. Основним джерелом навантаження на базу даних є часті вибірки історії транзакцій користувача та агрегація даних для звітів, які фільтруються за ідентифікатором та датою.

Для запобігання повному скануванню таблиць та прискорення виконання запитів, до структури таблиць було додано індекси на поля, що найчастіше беруть участь у фільтраціях. Створення індексів у міграції бази даних показано в лістингу 3.19:

Лістинг 3.19 — *Створення індексів у міграції бази даних*

```
await queryInterface.addIndex('Transactions', ['userId']);
await queryInterface.addIndex('Transactions', ['userId',
'date']);
```

Індекс на поле userId значно оптимізує завантаження персональних даних користувача, а комбінований індекс ['userId', 'date'] оптимізує аналітичні запити групування доходів та витрат за часові інтервали.

3.5 Оптимізація продуктивності клієнтської та серверної частин веб-додатку

Висока швидкість відгуку інтерфейсу та мінімальний час обробки серверних запитів є критично важливими для утримання користувачів. Веб-додаток використовує хмарну Serverless-архітектуру Vercel та Neon DB, що накладає певні обмеження, зокрема проблему «холодного старту» (cold start) серверних функцій та ліміти безкоштовних тарифних планів. Для

забезпечення стабільної продуктивності було реалізовано комплекс заходів з оптимізації на обох рівнях системи.

3.5.1. Оптимізація клієнтської частини (Frontend)

Основною метою оптимізації React SPA є мінімізація кількості повторних рендерингів (re-renders) компонентів та зменшення розміру завантажуваної збірки додатка. Для цього впроваджено такі рішення:

- Оптимальне управління станом: глобальний фінансовий стан (транзакції, категорії, бюджети) винесено в окремий React Context (FinanceContext). Це дозволяє уникнути передачі пропсів через багато рівнів компонентів (prop drilling) та оновлювати лише ті елементи інтерфейсу, які безпосередньо залежать від змін у контексті.
- Ефективна робота з великими списками: під час рендерингу таблиці транзакцій у списку використовується унікальний атрибут key (UUID транзакції). Це дозволяє алгоритму React Virtual DOM точно визначати змінені елементи та перемальовувати лише додані чи відредаговані рядки, не зачіпаючи решту списку.
- Відмова від сторонніх бібліотек візуалізації: замість інтеграції важких бібліотек (наприклад, Chart.js або Recharts), які збільшують розмір JS-бандлу на 150-200 КБ, графіки аналітики розроблено на чистій розмітці SVG з обчисленням координат засобами React. Це дозволило зменшити обсяг передаваного коду та прискорити час першої отрисовки сторінки (FCP — First Contentful Paint).

3.5.2. Оптимізація серверної частини (Backend) та бази даних

Основне навантаження на сервер пов'язане із запитами до бази даних PostgreSQL для вибірки транзакцій та перевірки лімітів. Оптимізація бекенду включає такі кроки:

1. Параметризація та індексація запитів: у базі даних Neon DB створено комбіновані індекси на поля, які найчастіше використовуються для

фільтрації: ['userId'] та ['userId', 'date']. Це дозволило перевести пошук записів зі сканування всієї таблиці (Sequential Scan) на логарифмічний пошук по дереву індексу (Index Scan), що скоротило час виконання складних аналітичних запитів з 200-300 мс до 2-5 мс.

2. Налаштування пулу з'єднань (Connection Pooling): безсерверні функції Vercel при кожному запиті можуть створювати нове підключення до бази даних, що призводить до швидкого вичерпання ліміту підключень (max connections) СУБД Neon DB. Для оптимізації Sequelize налаштовано на роботу через вбудований пул підключень із обмеженням максимальної кількості одночасних конектів:
3. Серверна агрегація даних: розрахунок звітів для дашборду та аналітики виконується безпосередньо на стороні СКБД за допомогою SQL-функцій агрегації (SUM, COUNT, GROUP BY). Бекенд отримує вже готові стислі результати (наприклад, масив об'єктів із сумами за категоріями), замість того, щоб завантажувати в пам'ять тисячі записів транзакцій та підраховувати їх засобами Javascript, що суттєво заощаджує оперативну пам'ять сервера застосунків.

4. РОЗГОРТАННЯ, ТЕСТУВАННЯ ТА ІНСТРУКЦІЯ КОРИСТУВАЧА

4.1 Тестування додатку

Для перевірки працездатності системи було проведено ручне верифікаційне тестування ключових модулів:

Автентифікація та авторизація: перевірено блокування спроб повторної реєстрації на існуючий email (HTTP 409) та реєстрації з некоректним паролем (HTTP 400). Спроби доступу до приватних маршрутів без валідного JWT-токена блокуються мідлваром із поверненням статусу HTTP 401.

Контроль лімітів: після створення бюджету з лімітом 1000 грн... та додавання витрати на 1200 грн у категорії «Транспорт» система успішно згенерувала сповіщення про перевищення ліміту на 200 грн. Локальний інтерфейс користувача миттєво відобразив червону попереджувальну картку.

Графічний інтерфейс та адаптивність: підтверджено відповідність розрахунків SVG Donut Chart та Flex-діаграм динаміці доходів/витрат з бекенду. Перевірено коректність відображення інтерфейсу на мобільних пристроях із шириною екрана менше 992px (автоматичне згортання бічної панелі навігації).

4.2 Розгортання системи

Для забезпечення постійної доступності та стабільної роботи веб-додатку у продуктивному середовищі (Production) було реалізовано хмарне розгортання компонентів системи. Для цього використано сучасні хмарні платформи та концепцію Serverless, що дозволяє мінімізувати витрати на адміністрування інфраструктури та автоматизувати процес оновлення коду.

Схематично архітектура розгортання системи в хмарній інфраструктурі та процес автоматичного оновлення (CI/CD) представлені на діаграмі розгортання (Рис.4.1).

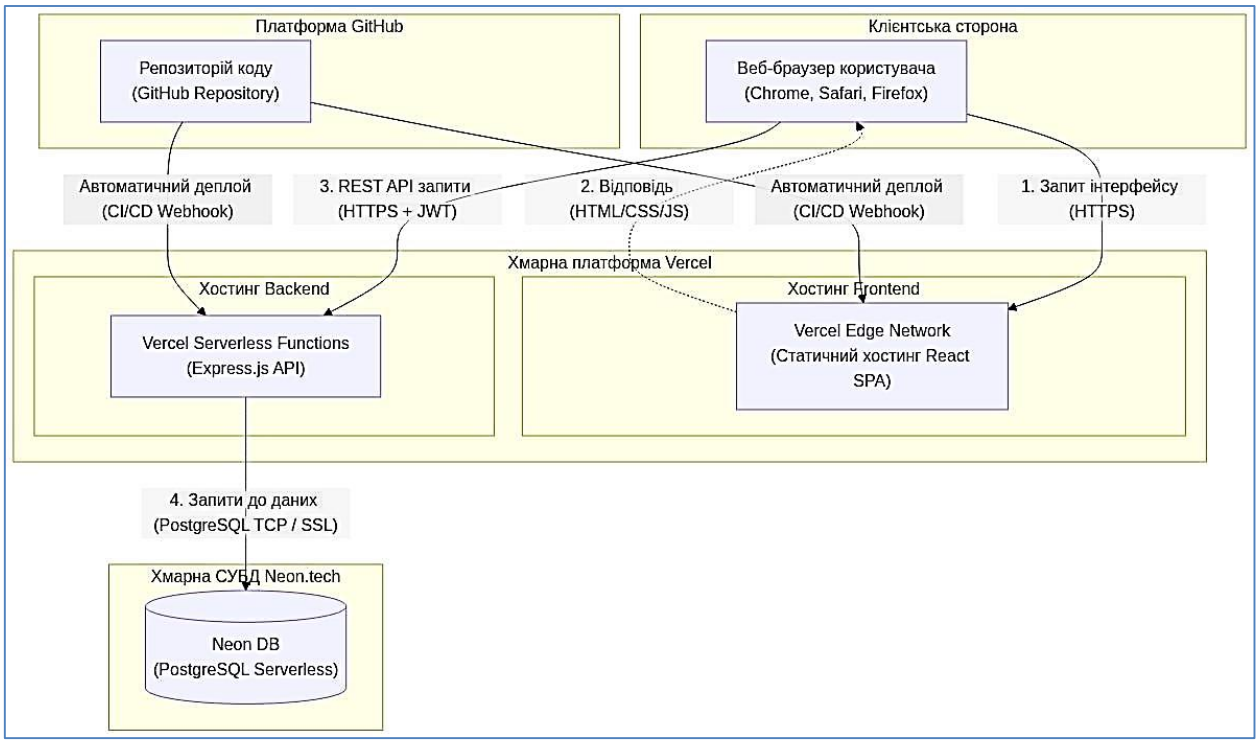


Рисунок 4.1 — *Архітектура розгортання системи*

4.2.1 Хмарна база даних (Neon DB)

Базу даних PostgreSQL розгорнуто на хмарній Serverless-платформі Neon DB (neon.tech). Вибір Neon DB зумовлений такими перевагами:

- Автоматичне масштабування обчислювальних ресурсів бази даних залежно від навантаження.
- Функція розгалуження бази даних (branching), що дозволяє миттєво створювати копії БД для тестування.
- Підтримка з'єднань виключно через захищений протокол SSL/TLS.
- Безкоштовність

Для підключення сервера застосунків до бази даних у панелі Neon DB було створено продуктивну базу, активовано режим обов'язкового шифрування трафіку та згенеровано рядок підключення (DATABASE_URL), який має такий формат:
postgres://[user]:[password]@[host]/[dbname]?sslmode=require

4.2.2 Розгортання серверної частини (Vercel Serverless Functions)

Серверний Express-додаток розгорнуто на хмарній платформі Vercel як набір безсерверних функцій. Для цього архітектура Express-сервера була адаптована під вимоги Vercel:

Файл index.js налаштований як єдиний експорт додатка (module.exports = app), що дозволяє Vercel динамічно імпортувати Express як обробник запитів.

У корені папки бекенду створено конфігураційний файл vercel.json (Лістинг 4.1), який описує збирання проєкту за допомогою @vercel/node та перенаправляє всі вхідні API-запити на точку входу index.js:

Лістинг 4.1 — *Файл vercel.json*

```
{
  "version": 2,
  "builds": [
    {
      "src": "index.js",
      "use": "@vercel/node"
    }
  ],
  "routes": [
    {
      "src": "/(.*)",
      "dest": "index.js"
    }
  ]
}
```

У налаштуваннях проєкту на Vercel було додано змінні оточення (Environment Variables) для production середовища:

- NODE_ENV = production - вимикає зайве логування та активує оптимізації Sequelize.
- DATABASE_URL - посилання на Neon DB з параметрами SSL.
- JWT_SECRET - криптографічний ключ для генерації та підпису JWT-токенів.

4.2.3. Розгортання клієнтської частини та CI/CD конвеєр

Клієнтський SPA-додаток (React + TypeScript) розгорнуто як статичний веб-ресурс на платформі Vercel. Процес розгортання повністю інтегровано з репозиторієм на GitHub, що дозволило побудувати автоматизований CI/CD конвеєр:

1. При кожному коміті та злитті (push/merge) гілок з основною гілкою main, GitHub автоматично надсилає вебхук на сервери Vercel.
2. Vercel ініціює нову збірку клієнтського додатка, запускаючи команду npm run build для компіляції TypeScript та мінімізації статичних файлів (HTML, CSS, JS) за допомогою збирача Vite.
3. Під час збирання Vite підставляє адресу production API зі змінної оточення VITE_API_URL (наприклад, https://finance-flow-api.vercel.app/api) в клієнтський код.
4. Після завершення збирання нові статичні файли розгортаються на мережі доставки контенту Vercel Edge Network, роблячи додаток доступним користувачам у всьому світі.

4.3 Інструкція користувача

Веб-додаток спроєктовано з урахуванням сучасних вимог до зручності UI/UX та адаптованості для роботи як на персональних комп'ютерах, так і на

мобільних пристроях. Основні елементи керування та екрани системи мають інтуїтивно зрозуміле призначення.

4.3.1. Екран автентифікації та реєстрації

При першому відкритті додатку неавторизований користувач потрапляє на форму входу (Рис.4.2). У разі відсутності облікового запису користувач обирає вкладку реєстрації, де вводить ім'я, електронну пошту та пароль. Після успішної реєстрації система автоматично авторизує користувача та перенаправляє його на головну сторінку.

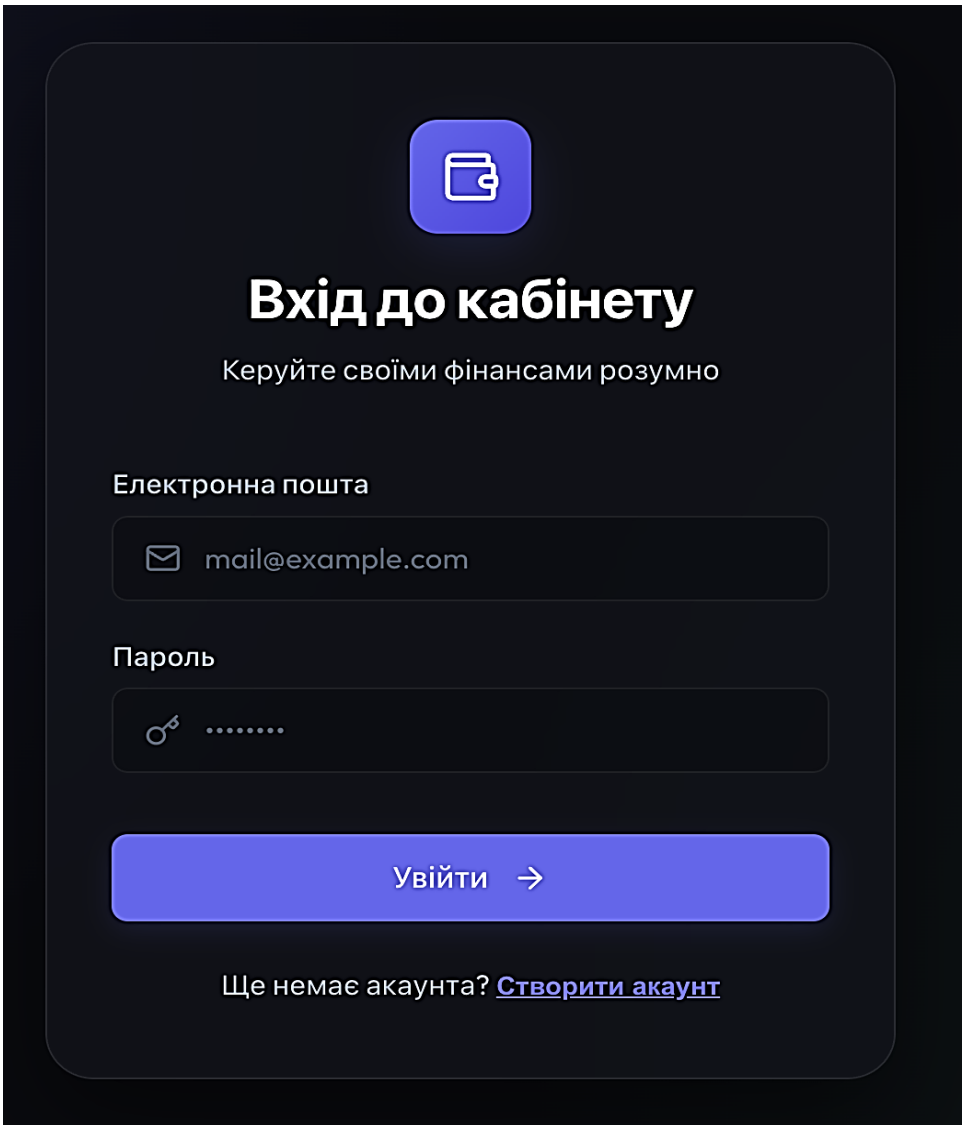


Рисунок 4.2 — *Форма логіну*

4.3.2. Головна панель (Dashboard)

Головна панель (Рис.4.3) є центральним робочим простором додатку і містить такі ключові елементи:

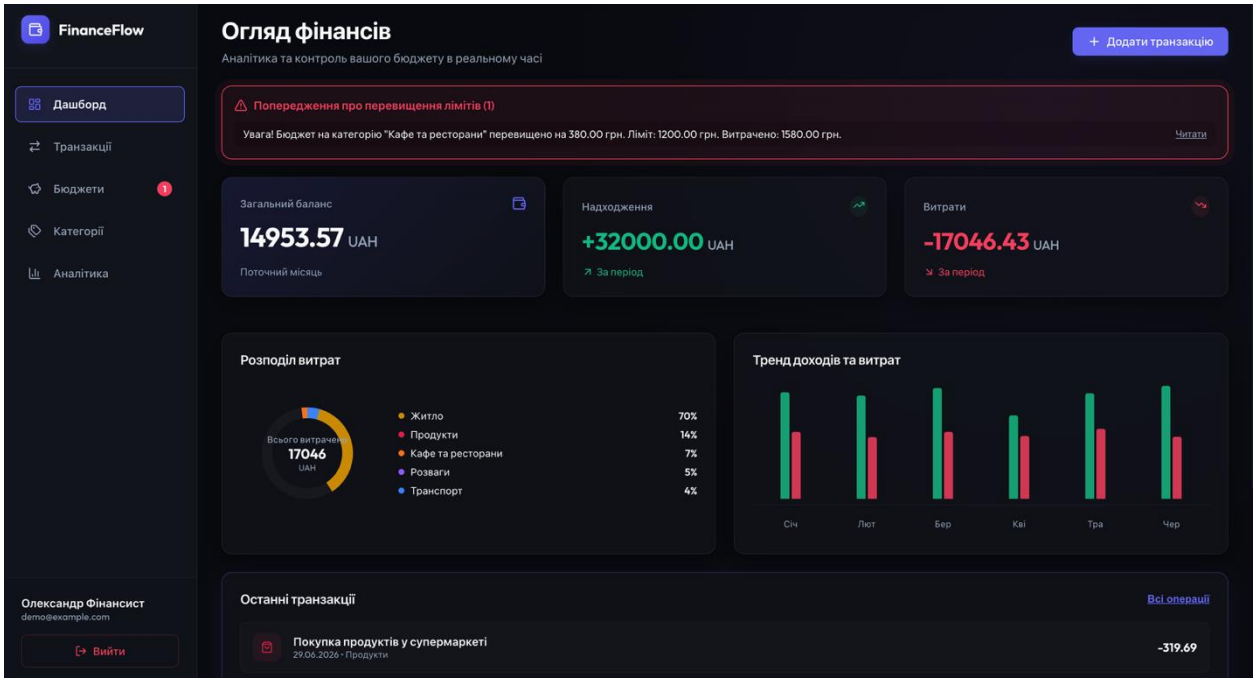


Рисунок 4.3 — *Головна панель*

- Базові аналітичні картки: відображають загальну суму надходжень, витрат, баланс та відсоток заощаджень користувача за поточний місяць.
- Блок швидкого введення транзакцій: дозволяє в один клік відкрити форму додавання витрати чи доходу, що спрощує щоденне фіксування операцій.
- Панель сповіщень про перевищення бюджету: якщо у будь-якій категорії витрати перевищили встановлений ліміт, у верхній частині панелі рендериться сповіщення із зазначенням категорії та суми перевищення. Користувач може позначити його як прочитане, натиснувши на відповідну іконку.
- Міні-графік та список останніх транзакцій для швидкого ознайомлення.

4.3.3 Історія транзакцій

Екран «Транзакції» призначений для детального перегляду та адміністрування фінансових операцій. На рисунку 4.4 показана історія транзакцій:

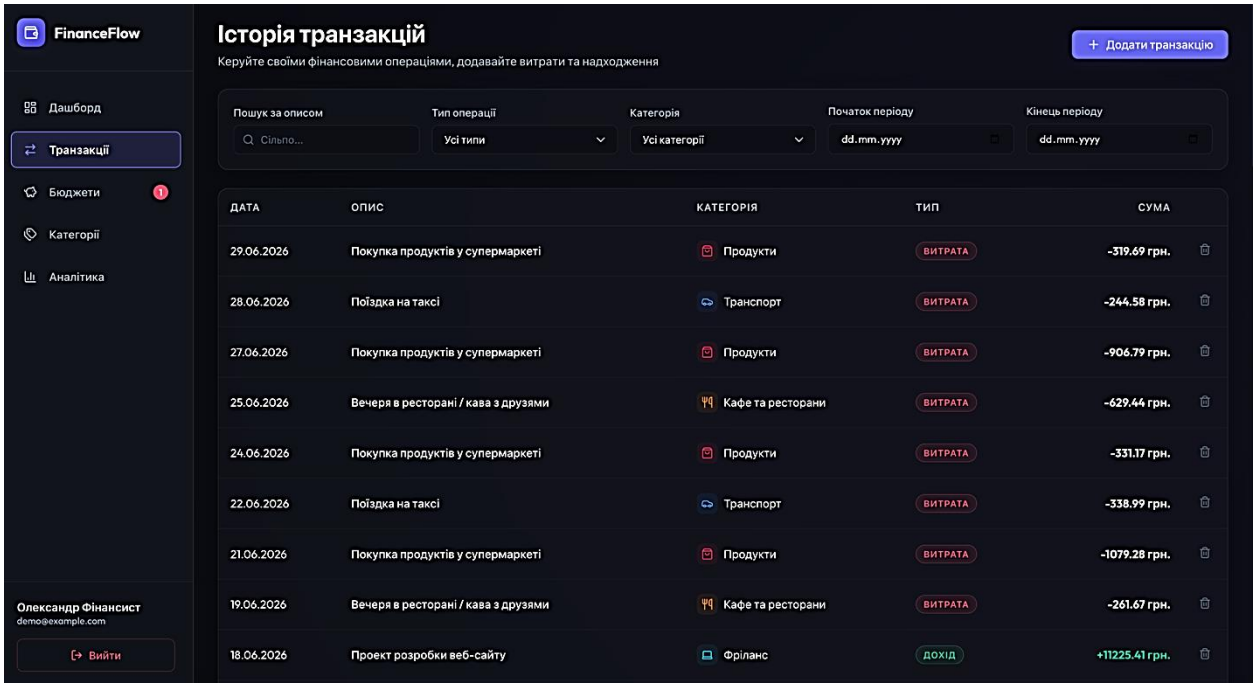


Рисунок 4.4 — *Історія транзакцій*

Таблиця транзакцій: містить дату, обрану категорію (із відповідною іконкою та кольором), суму, опис та тип транзакції.

Кнопки керування: користувач може відредагувати кожну транзакцію через спливаючу форму або повністю видалити її, після чого загальний баланс системи буде перераховано.

4.3.4 Керування категоріями та лімітами бюджетів

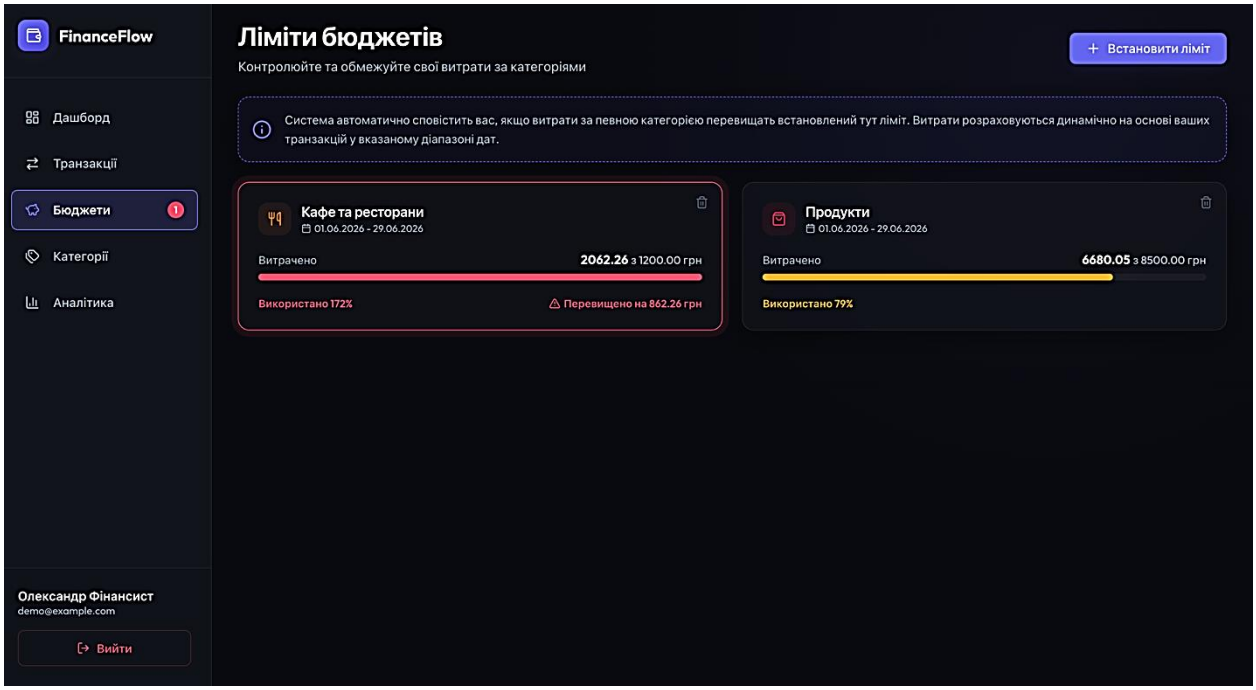


Рисунок 4.5 — *Єкран лімітів бюджету*

Розділ «Бюджети» (Рис.4.5) дозволяє планувати майбутні витрати:

Встановлення лімітів: користувач обирає категорію витрат, вказує граничну суму та період дії ліміту (тиждень, місяць).

Візуальний індикатор: кожен бюджет відображається у вигляді картки зі смугою прогресу. При наближенні фактичних витрат до встановленого ліміту колір індикатора динамічно змінюється, візуально попереджаючи користувача про загрозу перевищення.

4.3.5 Модуль детальної аналітики

Екран «Аналітика» (Рис.4.6) надає інструменти для глибинного ретроспективного аналізу:

se1212-26@stu.zsea.edu.ua)

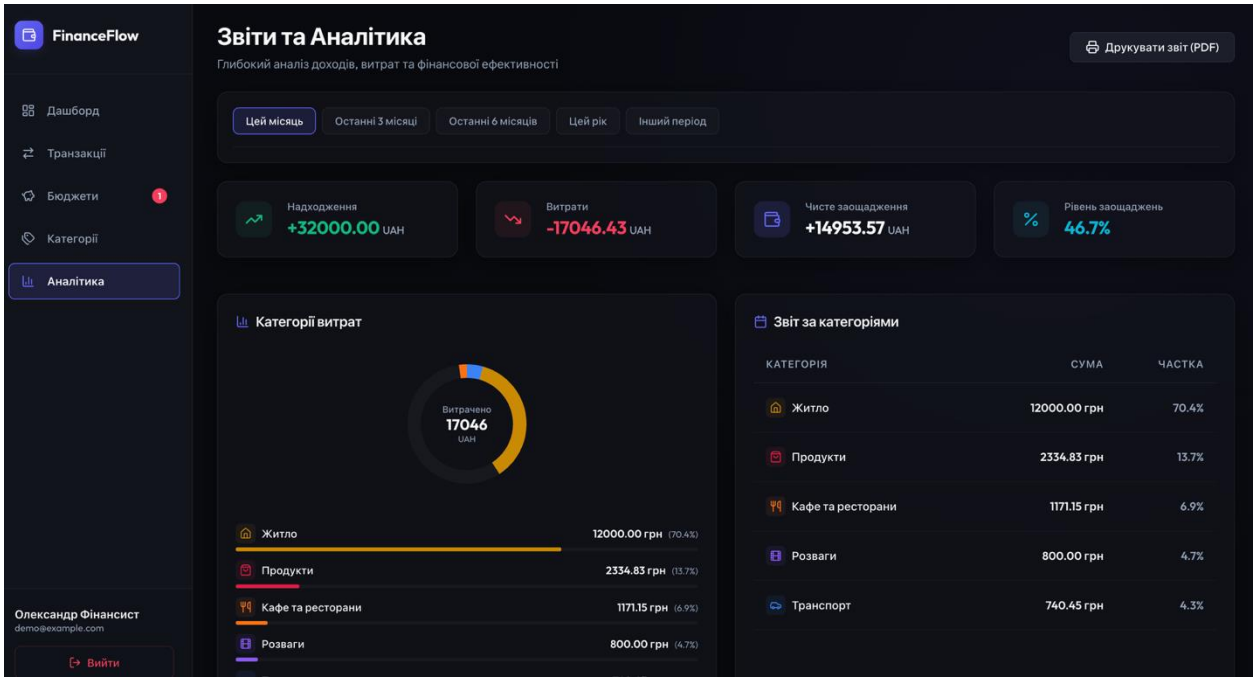


Рисунок 4.6 — *Звіти та аналітика*

Фільтрація за періодом: користувач може обрати стандартний період (поточний місяць, останні 3 чи 6 місяців, поточний рік) або задати довільний інтервал дат у календарі.

Діаграма розподілу витрат: SVG-кругова діаграма динамічно групує витрати за категоріями, відображаючи відсоткову частку та абсолютну суму по кожному напрямку.

Графік динаміки фінансів: стовпчиковий графік порівнює доходи та витрати за місяцями, допомагаючи відстежувати довгострокові фінансові тренди.

ВИСНОВКИ

1. В процесі роботи було проаналізовано предметну область ведення особистого бюджету, досліджено існуючі програмні аналоги та сформульовано вимоги до веб-додатку.

2. Обґрунтовано та обрано технологічний стек для реалізації клієнт-серверної архітектури: React та TypeScript на фронтенді, Node.js та Express на бекенді, СУБД PostgreSQL.

3. Розроблено архітектурну та інформаційну модель системи, побудовано UML-діаграми прецедентів і послідовностей взаємодії компонентів, а також спроектовано реляційну базу даних із використанням унікальних ідентифікаторів UUID v4.

4. Реалізовано серверну частину веб-додатку (Express API) із JWT-автентифікацією, алгоритмами автоматичного контролю лімітів бюджету та захистом від типових веб-вразливостей.

5. Розроблено адаптивний клієнтський SPA-застосунок з інтерактивною візуалізацією аналітичних звітів за допомогою кастомних SVG-діаграм без залучення важких сторонніх бібліотек.

6. Виконано хмарне розгортання бази даних на платформі Neon DB та прикладних модулів на хостингу Vercel, а також налаштовано автоматизований CI/CD конвеєр через GitHub.

7. Результати тестування підтвердили коректність роботи бізнес-логіки перевірки лімітів, захищеність API-ендпоінтів та стабільне відображення інтерфейсу на мобільних пристроях.

.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Фленеган Д. JavaScript: Детальний посібник. 7-ме вид. Пер. з англ. Київ : Діалектика, 2021. 720 с.
2. Документація Postgresql. URL: https://postgrespro.ru/docs/postgresql/14 (дата звернення - 22.04.2026)
3. Мартін Р. Чиста архітектура. Мистецтво розробки програмного забезпечення.2022. 368 с.
4. Фаулер М. Рефакторинг. Поліпшення дизайну існуючого коду. 2-ге вид. Пер. з англ. Київ : Діалектика, 2020. 448 с.
5. Документація Git. URL: https://git-scm.com/book/uk/v2/%D0%92%D1%81%D1%82%D1%83%D0%BF-%D0%9E%D1%81%D0%BD%D0%BE%D0%B2%D0%B8-Git (дата звернення - 17.04.2026)
6. Вейсфельд М. Об'єктно-орієнтований підхід до проектування. Київ : Діалектика, 2020. 320 с.
7. Sequelize Object-Relational Mapper Documentation. URL: https://sequelize.org (дата звернення: 15.05.2026).
8. React Official Documentation. URL: https://react.dev (дата звернення: 10.05.2026).
9. Node.js Runtime Environment Documentation. URL: https://nodejs.org/docs (дата звернення: 12.05.2026).
10. Express.js Web Framework Documentation. URL: https://expressjs.com (дата звернення: 14.05.2026).
11. JSON Web Token (JWT) Specification (RFC 7519). Internet Engineering Task Force (IETF), 2015. URL: https://tools.ietf.org/html/rfc7519 (дата звернення: 21.05.2026).
12. Vercel Cloud Platform Documentation. URL: https://vercel.com/docs (дата звернення: 18.05.2026).

13. OWASP Top 10 Web Application Security Risks. OWASP Foundation, 2021. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 20.05.2026).
14. Neon Serverless PostgreSQL Documentation. URL: https://neon.tech/docs (дата звернення: 19.05.2026).
15. Біліба А. РОЗРОБКА ВЕБ-ДОДАТКУ ДЛЯ УПРАВЛІННЯ ОСОБИСТИМИ ФІНАНСАМИ. *Збірник наукових праць студентів, здобувачів PhD, докторантів і молодих вчених, університетська конференція «МОЛОДА НАУКА-2026» Том III «Секції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету»* : м. Запоріжжя, 20–24 квітня 2026 р. Запоріжжя, 2026. С. 82–83. – .

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Біліба Анна Генадіївна, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми «Програмне забезпечення систем» — підтверджую, що написана мною кваліфікаційна робота на тему «Розробка програмного забезпечення для автоматизації бізнес-процесів» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

-згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 12.06.2026 _____ Біліба Анна Генадіївна
(студент)

Дата 12.06.2026 _____ Скрипник Ірина Анатоліївна
(керівник)