

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

«До захисту допущено»

Завідувач кафедри

Наталія АУШЕВА
“ ” 2026 р.

**Дипломна робота
на здобуття ступеня бакалавр**

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Система управління фінансами з можливостями прогнозування”

Виконала: студентка 4 курсу, групи ТР-25

Голуб Тетяна Дмитрівна

(прізвище, ім’я, по батькові)

(підпис)

Керівник доцент, к.е.н, Ірина СЕГЕДА

(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ)

(підпис)

Рецензент В.о. директора НН ІАТЕ КПІ ім. Ігоря Сікорського, д.т.н.,
проф. Черноусенко О.Ю.

(посада, науковий ступінь, вчене звання, ім’я, ПРІЗВИЩЕ)

(підпис)

Н.контроль асистент Константин ЗДОР

(посада, ім’я, ПРІЗВИЩЕ)

(підпис)

Засвідчую, що у цій дипломній роботі немає
запозичень з праць інших авторів без
відповідних посилань.

Студентка _____

Київ - 2026

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

**НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ
ЕНЕРГЕТИКИ**

кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)

спеціальність 122 “Комп’ютерні науки”

Освітньо-професійна програма “Цифрові технології в енергетиці”

ЗАТВЕРДЖУЮ

Завідувач кафедри ЦТЕ

Наталія АУШЕВА

“ ” 2026 р.

**ЗАВДАННЯ
на дипломну роботу студентці**

ГОЛУБ Тетяні Дмитрівні

(прізвище, ім’я, по батькові)

1. Тема роботи: «Система управління фінансами з можливостями
прогнозування»

Науковий керівник Сегеда Ірина Василівна, к.е.н., доцент

(прізвище, ім’я, по батькові, ніковий ступінь, вчене звання)

затверджені наказом по університету від “28” травня 2026 р. №1992-с

2. Термін подання роботи студентом “01” червня 2026 року

3. Вихідні дані до роботи: мова програмування TypeScript, програмна платформа Node.js, фреймворки NestJS та React.js, система керування базами даних PostgreSQL, брокер повідомлень RabbitMQ, інструмент контейнеризації Docker.

4. Перелік питань, які потрібно розробити:

1) провести аналіз предметної області управління персональними фінансами та існуючих програмних рішень;

2) обґрунтувати вибір мікросервісної архітектури та технологічного стеку для забезпечення масштабованості системи;

- 3) спроектувати архітектуру баз даних та модель асинхронної міжсервісної взаємодії;
- 4) розробити математичне забезпечення модуля прогнозування витрат на основі методу зваженої лінійної регресії;
- 5) реалізувати серверну частину (мікросервіси користувачів, транзакцій, цілей, аналітики) та клієнтський веб-інтерфейс;
- 6) провести тестування працездатності системи в ізольованих контейнеризованих середовищах та представити сценарії її роботи.
5. Орієнтовний перелік ілюстративного матеріалу: структурна схема мікросервісу та потоків обробки даних, графік апроксимації фінансових витрат методом найменших квадратів, діаграма компонентів системи управління фінансами, діаграма класів об'єктної моделі бази даних, діаграми послідовності, скріншоти графічного інтерфейсу веб-застосунку (дашборд, управління цілями, аналітика).
6. Дата видачі завдання 10.12.2026 р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1.	Вибір теми роботи	04.11.2025	виконано
2.	Аналіз методів та засобів розв'язання задачі	20.04.2026 – 21.04.2026	виконано
3.	Розробка архітектури та загальної структури системи	22.04.2026 – 26.04.2026	виконано
4.	Розробка окремих підсистем	27.04.2026 – 28.04.2026	виконано
5.	Програмна реалізація системи	29.04.2026 – 13.05.2026	виконано
6.	Оформлення пояснювальної записки	14.05.2026 – 30.05.2026	виконано
7.	Захист програмного забезпечення	13.05.2026	виконано
8.	Передзахист	03.06.2026	виконано
9.	Захист	15.06.2026	виконано

Студентка

(підпис) Тетяна ГОЛУБ
(ім'я, ПРІЗВИЩЕ)

Керівник роботи

(підпис) Ірина СЕГЕДА
(ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана на 66 сторінках, містить 16 ілюстрацій, 2 таблиці, 2 додатки, 20 джерел в переліку посилань.

Метою роботи є проектування та розробка масштабованої мікросервісної системи управління особистими фінансами «FinanceLibre», що забезпечує автоматизований облік операцій, аналітичну обробку даних та інтелектуальне прогнозування майбутніх витрат.

Методи та засоби розробки: мікросервісна архітектура, контейнеризація Docker, брокер повідомлень RabbitMQ, фреймворк NestJS, бібліотека React.js, СУБД PostgreSQL, метод зваженої лінійної регресії для аналізу часових рядів та механізми JWT-автентифікації.

Основний зміст роботи полягає у створенні розподіленої системи, де кожен функціональний вузол (користувачі, транзакції, цілі, прогнози) функціонує як незалежний сервіс. У роботі детально описано реалізацію синхронної та асинхронної міжсервісної взаємодії, що дозволяє мінімізувати затримки та забезпечити високу відмовостійкість. Особливу увагу приділено математичному забезпеченню аналітичного модуля, який на основі історичних даних будує прогноз фінансового стану користувача.

Результатом впровадження розробки є дієздатна програмна система, що дозволяє користувачам ефективно планувати бюджет. Експериментально підтверджено, що використання зваженої моделі регресії підвищує точність ідентифікації фінансових трендів порівняно зі стандартними методами статистичного усереднення. Система рекомендована для використання як інструмент персонального фінансового менеджменту.

Ключові слова: мікросервісна архітектура, Docker, RabbitMQ, NestJS, React.js, PostgreSQL, лінійна регресія, фінансове прогнозування, асинхронна взаємодія, інтелектуальний аналіз.

ABSTRACT

The thesis consists of 66 pages, includes 16 illustrations, 2 tables, 2 appendices, and 20 sources in the list of references.

The purpose of the work is the design and development of a scalable microservice system for personal finance management "FinanceLibre", which provides automated accounting of operations, analytical data processing, and intelligent forecasting of future expenses.

Methods and tools: microservice architecture, Docker containerization, RabbitMQ message broker, NestJS framework, React.js library, PostgreSQL DBMS, weighted linear regression method for time series analysis, and JWT authentication mechanisms.

The main content of the work is to create a distributed system where each functional unit (users, transactions, goals, forecasts) functions as an independent service. The work describes in detail the implementation of synchronous and asynchronous inter-service interaction, which allows for minimizing delays and ensuring high fault tolerance. Particular attention is paid to the mathematical support of the analytical module, which builds a forecast of the user's financial condition based on historical data.

The result of the implementation is a viable software system that allows users to effectively plan their budget. It has been experimentally confirmed that the use of a weighted regression model increases the accuracy of identifying financial trends compared to standard statistical averaging methods. The system is recommended for use as a tool for personal financial management.

Keywords: microservice architecture, Docker, RabbitMQ, NestJS, React.js, PostgreSQL, linear regression, financial forecasting, asynchronous interaction, data analy.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.....	8
ВСТУП.....	10
1. АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧІ СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ.....	12
1.1 Постановка задачі та мета дослідження.....	12
1.2 Аналіз предметної області управління фінансами.....	13
1.3 Аналіз існуючих рішень.....	15
1.4 Обґрунтування вибору архітектури.....	16
1.5 Обґрунтування вибору технологічного стеку	18
2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ.....	20
2.1 Модель мікросервісної взаємодії.....	20
2.2 Організація асинхронного обміну повідомленнями (RabbitMQ)	23
2.2.1 Паттерни проектування черг повідомлень.....	24
2.2.2 Забезпечення цілісності даних при асинхронній передачі.....	25
2.3 Контейнеризація системи	26
2.3.1 Створення ізольованих середовищ для сервісів.....	26
2.3.2 Конфігурація мережевої взаємодії в Docker Compose.....	27
2.4 Порівняльний аналіз методів прогнозування часових рядів	29
2.5 Математичне забезпечення та алгоритм лінійної регресії.....	31
3. РЕАЛІЗАЦІЯ ПРОГРАМНОЇ СИСТЕМИ.....	34
3.1 Архітектура системи та проектування логіки.....	34
3.1.1 UML-діаграми (діаграма класів, компонентів)	34
3.1.2 Діаграма послідовності.....	37
3.2 Проектування бази даних.....	40
3.3 Реалізація User Service та механізмів автентифікації	42
3.4 Реалізація Transaction Service для обліку фінансових операцій	44
3.5 Реалізація Goal Service для моніторингу та управління фінансовими цілями	45
3.6 Реалізація Forecast Service для предиктивного аналізу та прогнозування витрат	46

3.7 Реалізація графічного інтерфейсу користувача та засобів візуалізації фінансової статистики	48
4. ТЕСТУВАННЯ ТА АНАЛІЗ РОБОТИ СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ.....	50
4.1 Інсталяція програмного забезпечення та вимоги до середовища.....	50
4.2 Демонстрація функціоналу та сценарії роботи системи.....	51
ВИСНОВКИ.....	55
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	57
ДОДАТОК А	59
ДОДАТОК Б	65

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ACID (Atomicity, Consistency, Isolation, Durability) – набір властивостей (атомарність, консистентність, ізолюваність, довговічність), що гарантують надійну обробку транзакцій у системі керування базами даних.

AMQP (Advanced Message Queuing Protocol) – відкритий протокол прикладного рівня для передачі повідомлень між компонентами системи з гарантією доставки та гнучкою маршрутизацією.

CRUD (Create, Read, Update, Delete) – чотири базові функції, що використовуються при роботі з базами даних для управління об'єктами.

DTO (Data Transfer Object) – об'єкт, призначений для передачі даних між процесами або компонентами системи, що не містить бізнес-логіки та дозволяє оптимізувати мережевий трафік.

JWT (JSON Web Token) – компактний, безпечний для URL стандарт представлення претензій, що передаються між двома сторонами, який використовується для автентифікації та обміну інформацією.

МНК (Метод найменших квадратів) – математичний метод, що використовується в регресійному аналізі для знаходження параметрів моделі шляхом мінімізації суми квадратів відхилень реальних даних від прогнозних.

ORM (Object-Relational Mapping) – технологія програмування, яка зв'язує бази даних з концепціями об'єктно-орієнтованих мов програмування, дозволяючи працювати з таблицями як з об'єктами.

RPC (Remote Procedure Call) – протокол, що дозволяє програмі ініціювати виконання процедури або підпрограми в іншому адресному просторі (на іншому сервісі).

SPA (Single Page Application) – веб-додаток, що використовує один HTML-документ як оболонку для всіх сторінок і динамічно оновлює вміст без повного перезавантаження сторінки.

UML (Unified Modeling Language) – уніфікована мова моделювання, що використовується для візуалізації, специфікації та документування структури і динаміки програмних систем.

UUID (Universally Unique Identifier) – стандарт ідентифікації, що використовується для створення унікальних значень у розподілених системах без необхідності централізованого управління.

ВСТУП

У сучасних умовах цифровізації суспільства та активного розвитку фінансових технологій ефективне управління особистими коштами набуває важливого значення як складова фінансової стабільності користувачів. Існуючі програмні засоби обліку витрат здебільшого орієнтовані на збереження інформації про фінансові операції, однак мають обмежені можливості аналітичної обробки даних і прогнозування майбутніх витрат. У зв'язку з цим актуальним є створення програмних систем, що поєднують функції обліку, аналізу та прогнозування персональних фінансових показників на основі накопиченої статистики користувача.

Метою дипломної роботи є проектування та розробка масштабованої мікросервісної системи управління особистими фінансами «FinanceLibre», яка забезпечує автоматизований облік операцій користувача, аналітичну обробку даних та прогнозування майбутніх витрат.

Об'єктом дослідження є процеси обліку та аналізу персональних фінансових показників у програмних інформаційних системах.

Предметом дослідження є теоретичні, методичні та практичні аспекти розробки й функціонування мікросервісної програмної системи управління персональними фінансами з можливостями прогнозування.

Для досягнення поставленої мети у роботі вирішуються завдання аналізу існуючих програмних рішень управління персональними фінансами, побудови архітектури системи, організації взаємодії між мікросервісами та реалізації модуля прогнозування витрат користувача.

У роботі використано методи об'єктно-орієнтованого проектування програмних систем, методи проектування реляційних баз даних та методи статистичного аналізу часових рядів.

Наукова новизна роботи полягає у розробці архітектури мікросервісної системи управління персональними фінансами з інтегрованим модулем

прогнозування витрат користувача на основі аналізу часових рядів, що забезпечує підтримку прийняття рішень під час планування особистого бюджету.

Практичне значення отриманих результатів полягає у створенні програмної системи, яка забезпечує ведення обліку доходів і витрат користувача, виконання аналітичної обробки даних та прогнозування майбутніх фінансових показників із використанням методу лінійної регресії.

Програмна реалізація системи виконана з використанням сучасного стеку технологій, зокрема фреймворків NestJS та React.js, системи керування базами даних PostgreSQL, брокера повідомлень RabbitMQ та засобів контейнеризації Docker, що забезпечує масштабованість, надійність та ефективність функціонування системи.

Апробація результатів. Результати дослідження були апробовані на XXIII Міжнародній науково-практичній конференції молодих вчених та студентів «Сучасні проблеми наукового забезпечення енергетики», секція 10 «Інформаційні технології та комп'ютерне моделювання» (доповідь «Financial Management System with Forecasting Capabilities», квітень 2026 р.).

1. АНАЛІЗ ТА ПОСТАНОВКА ЗАДАЧІ СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ

Перший розділ присвячено комплексному аналізу предметної області та формалізації вимог до проєктування системи управління персональними фінансами. У його межах визначено мету, об'єкт, предмет та ключові технічні завдання дослідження, а також проведено аналіз функціональних обмежень наявних ринкових аналогів. Особливу увагу приділено аналітичному порівнянню архітектурних підходів та науково-технічному обґрунтуванню вибору мікросервісної моделі й технологічного стеку розробки, що покликані забезпечити відмовостійкість, масштабованість та надійність програмного комплексу.

1.1 Постановка задачі та мета дослідження

Метою даної роботи є розробка інтелектуальної програмної системи управління персональними фінансами, спрямованої на автоматизацію процесів обліку доходів і витрат, забезпечення надійного збереження фінансової інформації, а також виконання предиктивного аналізу майбутніх витрат на основі накопичених статистичних даних.

Для досягнення визначеної мети розроблювана система повинна вирішувати наступні функціональні та технічні завдання:

1. **Архітектурна реалізація:** побудова системи на засадах мікросервісної архітектури, де кожен функціональний вузол (авторизація, транзакції, цілі, прогнозування) функціонує як незалежний сервіс. Це повинно забезпечити високу доступність системи та можливість її горизонтального масштабування.

2. **Управління даними:** реалізація механізмів збору та систематизації фінансових операцій користувача з можливістю їх класифікації за категоріями. Система має гарантувати цілісність даних та підтримку ACID-транзакцій.

3. **Міжсервісна взаємодія:** організація ефективного обміну даними між компонентами системи через асинхронні черги повідомлень, що дозволить мінімізувати час відгуку інтерфейсу та забезпечити стійкість до пікових навантажень.

4. **Аналітична обробка та прогнозування:** розробка математичного модуля для аналізу часових рядів фінансових показників. На основі ретроспективних даних про витрати користувача система повинна автоматично будувати прогноз на майбутні періоди, використовуючи методи статистичного аналізу.

5. **Візуалізація та інтерфейс:** створення інтуїтивно зрозумілого графічного інтерфейсу, що забезпечує візуалізацію фінансової статистики у вигляді динамічних діаграм і звітів для підвищення ефективності прийняття фінансових рішень користувачем.

Розроблена система має відповідати вимогам надійності, масштабованості та безпеки, забезпечуючи захист персональних даних користувачів за допомогою сучасних механізмів автентифікації. Очікуваним результатом є створення працездатного програмного комплексу, здатного функціонувати в ізольованих контейнеризованих середовищах.

1.2 Аналіз предметної області управління фінансами

Предметна область даного дослідження охоплює процеси управління персональними фінансами, що включають збір, збереження, систематизацію, аналіз та планування грошових потоків окремого користувача [1]. У сучасних економічних умовах ефективне фінансове планування є важливим інструментом

контролю витрат, оптимізації структури споживання та досягнення довгострокових фінансових цілей. Управління персональними фінансами передбачає регулярний моніторинг доходів і витрат, їх класифікацію за категоріями та аналіз змін фінансового стану користувача в часі.

Однією з основних проблем предметної області є складність ведення систематичного обліку великої кількості щоденних фінансових операцій та відсутність доступних інструментів прогнозування майбутніх витрат. Через це користувачі часто стикаються з ситуаціями виникнення непередбачених витрат, що ускладнює планування особистого бюджету. Існуючі підходи до вирішення цих завдань включають використання паперових записів, електронних таблиць та мобільних банківських застосунків. Проте банківські системи зазвичай обмежені операціями лише в межах однієї установи, що створює потребу у незалежних програмних засобах управління фінансами з можливістю централізованого аналізу даних користувача.

Важливим напрямом розвитку сучасних систем управління персональними фінансами є використання методів інтелектуального аналізу даних. Такі системи повинні не лише відображати поточний фінансовий стан користувача, а й забезпечувати можливість прогнозування майбутніх витрат на основі аналізу накопичених статистичних даних. Застосування методів аналізу часових рядів дозволяє виявляти періодичні платежі, сезонні зміни витрат та інші закономірності фінансової поведінки користувача, що підвищує ефективність планування бюджету [2].

З технічної точки зору предметна область висуває підвищені вимоги до забезпечення конфіденційності та цілісності даних користувача, тому система повинна будуватися з використанням архітектурних підходів, що забезпечують модульність, масштабованість та захищену взаємодію компонентів. Використання мікросервісної архітектури дозволяє реалізувати гнучку структуру системи, у якій модулі обліку фінансових операцій, аналітичної обробки даних та прогнозування

можуть функціонувати незалежно один від одного, що спрощує подальший розвиток програмного продукту.

1.3 Аналіз існуючих рішень

На сьогоднішній день існує чимало застосунків для ведення особистих фінансів, які можна розділити на три основні категорії: банківські додатки, спеціалізовані менеджери витрат та веб-сервіси бюджетування.

Банківські додатки, зокрема вбудована аналітика Монобанку або Приватбанку, є зручними завдяки автоматичному підтягуванню операцій. Однак вони відображають лише ті транзакції, що проходять через рахунки конкретної установи. Готівкові розрахунки та операції в інших банках залишаються поза обліком, що унеможливорює формування повної картини фінансового стану користувача. Спеціалізовані мобільні застосунки, такі як CoinKeeper, Money Lover, Wallet знімають проблему прив'язки до банку, дозволяючи вносити операції вручну. Проте аналітичні можливості таких рішень здебільшого обмежені базовими звітами та сповіщеннями про перевищення бюджету. Функціонал прогнозування витрат на основі накопичених даних у більшості з них відсутній [3]. Декі веб-сервіси пропонують глибшу аналітику, проте орієнтовані переважно на іноземні ринки, а їхня інтеграція з українськими банками через відкриті API залишається обмеженою.

Таким чином, незважаючи на велику кількість наявних рішень, більшість із них мають спільні недоліки, а саме: залежність від банківської інфраструктури, неповний облік готівкових операцій та відсутність інтелектуального прогнозування потенційних витрат та доходів. Саме ці прогалини визначили актуальність даної роботи. Розроблювана система реалізує підхід, при якому незалежний облік усіх типів операцій поєднується з модулем предиктивного

аналізу, що в сукупності недоступне в жодному з розглянутих рішень у такому вигляді.

1.4 Обґрунтування вибору архітектури

Під час вибору архітектурного підходу для системи управління персональними фінансами було проведено порівняльний аналіз монолітної та мікросервісної моделей. У монолітній архітектурі всі модулі використовують спільні ресурси оперативної пам'яті та процесора [4], що створює ризик нестачі ресурсів для даного проєкту. Виконання лінійної регресії на великому історичному періоді може заблокувати обробку простих HTTP-запитів, таких як авторизація користувача або додавання запису про витрату. Крім того, спільна база даних у монолітній архітектурі ускладнює розмежування прав доступу на фізичному рівні: сервіс статистики не повинен мати доступу до хешованих паролів користувачів, однак у межах однієї бази таке розділення є лише логічним і вразливим до помилок. Вибір мікросервісної моделі дозволяє розділити систему на автономні одиниці з окремим життєвим циклом кожної [5]. Це рішення обґрунтовується такими технічними аргументами:

1. **Профілювання ресурсів.** Сервіс `forecast-service` виконує складні математичні операції, що навантажують процесор, тоді як `transaction-service` переважно здійснює операції введення-виведення при роботі з базою даних. У контейнеризованому середовищі `Docker` можливо встановити жорсткі ліміти ресурсів окремо для аналітичного модуля, що гарантує відсутність впливу його роботи на стабільність шлюзу.

2. **Ізоляція доменів та баз даних.** Застосування однієї бази даних на сервіс дозволяє використовувати різні схеми даних. Наприклад, `user_db` містить лише таблиці профілів та ролей, що спрощує проходження аудитів безпеки, `transaction_db` оптимізована під швидкий запис та агрегацію транзакцій, а `goal_db`

— під часті оновлення статусів. Такий підхід унеможливорює каскадне пошкодження даних.

3. Незалежна експлуатація. Можливість оновлювати алгоритм прогнозування в forecast-service без перезапуску всієї системи є критичною для безперервного надання сервісу.

Порівняльна характеристика монолітної та мікросервісної архітектур представлена у таблиці.(таблиця 1.1).

Таблиця 1.1 — Технічне порівняння архітектурних рішень

Характеристика	Монолітний підхід	Мікросервісний підхід
Управління ресурсами	Спільний пул для всіх задач	Ізольовані контейнери з CPU/RAM лімітами
Цілісність даних	Єдина база, ризик надмірних зв'язків	Фізична ізоляція (Database per Service)
Масштабування	Тільки всієї системи одночасно	Тільки навантажених вузлів
Взаємозалежність	Жорстка	Слабка через RabbitMQ
Відмовостійкість	Повна зупинка при збої модуля	Збій аналітики не зупиняє вхід та облік

Хоча мікросервісна архітектура вносить додаткову складність у вигляді мережових затримок та необхідності оркестрації через Docker Compose, ці витрати є виправданими.

1.5 Обґрунтування вибору технологічного стеку

Для забезпечення високої продуктивності, масштабованості та надійності розроблюваної системи було проведено ретельний аналіз доступних технологічних рішень, за результатами якого сформовано цілісний стек інструментів. Основним критерієм вибору стала здатність технологій ефективно підтримувати мікросервісну архітектуру та забезпечувати високу швидкість розробки без втрати якості програмного коду.

Вибір платформи Node.js для серверної частини обумовлений її подієво-орієнтованою моделлю та неблокуючим введенням-виведенням [6], що є важливим для систем із великою кількістю асинхронних запитів. На базі Node.js використано фреймворк NestJS [7], який надає чітку архітектурну структуру та вбудовану підтримку мікросервісних паттернів [8]. Використання мови TypeScript дозволило впровадити строгу типізацію [9], що значно знижує ймовірність виникнення помилок на етапі компіляції та спрощує підтримку складних фінансових структур даних.

Клієнтська частина системи базується на бібліотеці React.js [10], яка дозволяє створювати динамічні та швидкі Single Page Applications. Вибір React.js зумовлений його компонентним підходом, що сприяє перевикористанню коду, та наявністю потужних інструментів для візуалізації даних. Оскільки аналітична складова є надзвичайно важливою для системи управління фінансами, то інтеграція бібліотек для побудови графіків дозволяє наочно демонструвати результати прогнозування.

Для зберігання даних обрано реляційну систему управління базами даних PostgreSQL [11]. В контексті фінансового обліку критично важливим є дотримання принципів ACID, які PostgreSQL гарантує на найвищому рівні. Використання паттерну «Database per Service» реалізовано шляхом створення ізольованих логічних баз даних для кожного мікросервісу, що виключає жорстку залежність між сховищами та дозволяє кожному модулю мати власну оптимізовану схему

даних. Це також спрощує процедуру резервного копіювання та міграції окремих частин системи.

Асинхронна взаємодія між компонентами та надійна передача повідомлень забезпечується брокером RabbitMQ [12]. Це рішення було обумовлене тим, що брокер підтримує протокол AMQP, який гарантує доставку повідомлень навіть у разі пікових навантажень або тимчасової недоступності окремих вузлів. RabbitMQ дозволяє реалізувати асинхронні ланцюжки оновлення даних, наприклад, синхронізацію балансу цілей після створення транзакції, не блокуючи основний потік роботи додатка.

Також, система передбачає використання засобів контейнеризації Docker та інструменту оркестрації Docker Compose [13] для ізоляції кожного мікросервісу в окремому контейнері та запобіганню конфліктам залежностей.

2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ

У другому розділі висвітлено етапи проєктування архітектурних, інфраструктурних та обчислювальних рішень системи. Зокрема, побудовано модель взаємодії мікросервісів і визначено механізми надійного асинхронного обміну даними через RabbitMQ. Описано процеси розгортання ізольованих середовищ за допомогою Docker та їхню оркестрацію. Окрему увагу приділено аналітичному ядру: здійснено порівняння методів обробки часових рядів і формалізовано математичний алгоритм лінійної регресії для побудови фінансових прогнозів.

2.1 Модель мікросервісної взаємодії

На відміну від монолітної архітектури, мікросервісний підхід передбачає декомпозицію системи на невеликі незалежні компоненти, кожен з яких відповідає за окрему бізнес-функцію, розгортається самостійно та взаємодіє з іншими через стандартні протоколи. Основними перевагами такого підходу є ізоляція збоїв, можливість незалежного масштабування окремих модулів та гнучкість у виборі технологій для кожного компонента.

Система управління персональними фінансами включає різні за функціями підсистеми. Облік транзакцій потребує швидкої обробки даних і активної взаємодії з базою даних, тоді як формування прогнозних моделей є більш ресурсомістким процесом і може виконуватися асинхронно. Мікросервісна декомпозиція дозволяє масштабувати сервіс прогнозування незалежно від інших компонентів, виділяючи йому обчислювальні ресурси лише за потреби. Ізоляція збоїв гарантує, що відмова аналітичного модуля не позначиться на базових операціях збереження фінансових записів.

Загальна структура системи базується на багаторівневій розподіленій моделі. Клієнтська частина реалізована у вигляді Single Page Application на основі бібліотеки React. Серверна частина складається з набору незалежних мікросервісів на платформі Node.js з фреймворком NestJS та мовою TypeScript. Дані зберігаються у реляційній базі даних PostgreSQL, міжсервісна взаємодія організована через брокер повідомлень RabbitMQ, а вся інфраструктура ізольована у Docker-контейнерах.

Відповідно до принципів предметно-орієнтованого проєктування серверну частину декомпоновано на п'ять основних мікросервісів. API Gateway є єдиною точкою входу для клієнтського застосунку - він приймає всі зовнішні запити, виконує первинну автентифікацію та маршрутизує їх до відповідних сервісів. Сервіс управління користувачами (user-service) відповідає за реєстрацію, зберігання облікових даних та видачу JWT-токенів. Сервіс транзакцій (transaction-service) реалізує основні бізнес-процеси: обробку доходів і витрат та їх збереження у базі даних. Сервіс фінансових цілей (goal-service) забезпечує встановлення цілей накопичення та відстеження їхнього виконання на підставі актуальних даних про транзакції. Сервіс прогнозування (forecast-service) є обчислювальним модулем, який на основі історичних даних формує прогноз майбутніх фінансових показників методом лінійної регресії.

Оскільки кожен мікросервіс у розробленій системі є самостійним програмним застосунком, для організації його внутрішнього коду було застосовано принцип багат шарової архітектури. Цей підхід передбачає чітке розділення коду на три логічні рівні: шар маршрутизації (контролери), шар бізнес-логіки (сервіси) та шар доступу до даних (репозиторії). Таке розділення дозволяє ізолювати різні процеси один від одного і значно спрощує розширення системи. Загальна схема внутрішньої структури мікросервісу та потоків обробки даних представлена на рисунку 2.1.

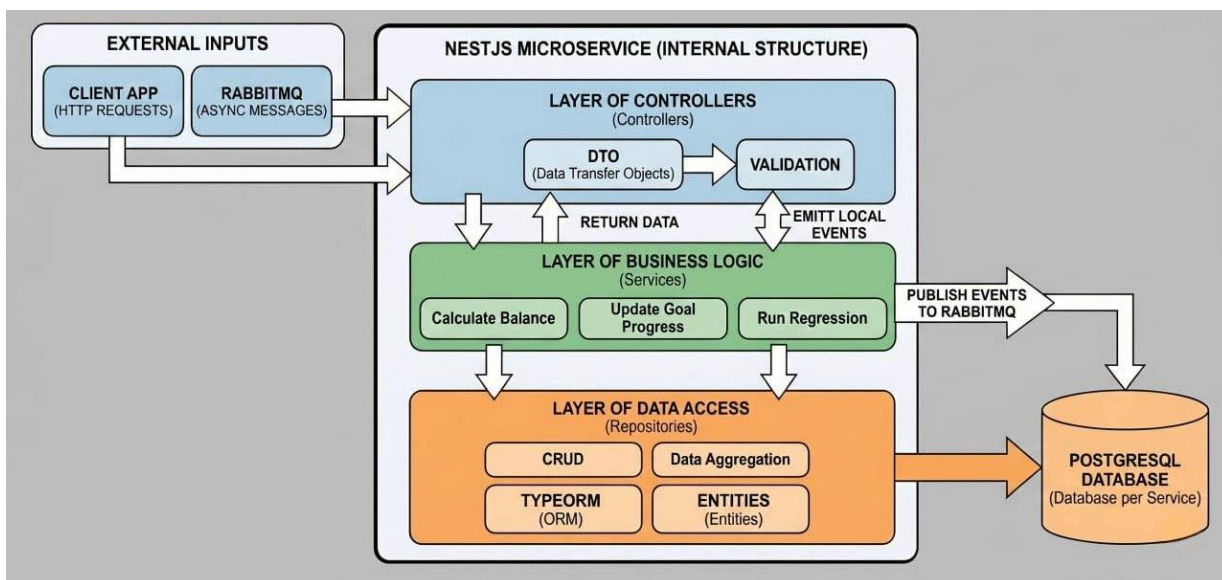


Рисунок 2.1 – Структурна схема мікросервісу та потоків обробки даних

Шар контролерів виступає своєрідною точкою входу для кожного мікросервісу. У даній розподіленій архітектурі контролери здебільшого діють як слухачі подій. Головна задача контролера - прийняти вхідні дані, виконати їх первинну перевірку та передати до наступного шару. Контролери не містять жодних розрахунків чи правил.

Шар сервісів утворює алгоритмічне ядро мікросервісу, оскільки саме тут реалізується вся бізнес-логіка системи. Отримавши запит від контролера, сервіс виконує необхідні операції: наприклад, розраховує баланс користувача, автоматично оновлює прогрес накопичень або запускає алгоритм лінійної регресії. Якщо для виконання цих задач потрібне збереження або зчитування інформації, сервіс делегує це завдання наступному рівню.

Шар репозиторіїв відповідає виключно за взаємодію з фізичними базами даних PostgreSQL. Для зручної роботи з даними використовується ORM-бібліотека TypeORM. Репозиторії абстрагують складні SQL-запити, тобто замість написання прямих команд до таблиць, програма працює з даними як із сутностями. Це забезпечує надійний захист бази від помилок і дозволяє зручно керувати збереженням фінансових транзакцій і цілей.

2.2 Організація асинхронного обміну повідомленнями (RabbitMQ)

Ефективна взаємодія між незалежними мікросервісами є ключовою умовою стабільної роботи розподіленої системи управління фінансами. Традиційні синхронні методи зв'язку на основі протоколу HTTP створюють жорстку часову залежність між компонентами: затримка або відмова одного сервісу безпосередньо блокує ресурси у всьому ланцюжку викликів. Для усунення цієї проблеми та забезпечення високої доступності у розробленій системі застосовано брокер повідомлень RabbitMQ [14], який реалізує передачу даних в асинхронному режимі. Це дозволяє сервісу, що ініціює подію, одразу формувати відповідь користувачеві, тоді як інші модулі здійснюють обробку отриманої інформації у фоновому режимі незалежно один від одного.

Технологічною основою обраного рішення є протокол AMQP (Advanced Message Queuing Protocol) [15]. Логіка маршрутизації у даному протоколі будується на взаємодії трьох ключових елементів: обмінника, черг та зв'язків. Під час проектування архітектури було проаналізовано різні логіки маршрутизації. Для зв'язку між API Gateway та функціональними мікросервісами застосовано тип Direct Exchange, що базується на точній відповідності ключа маршрутизації назві цільової черги. Водночас для організації внутрішньої синхронізації станів компонентів було впроваджено механізм Topic Exchange. Використання шаблонів та символів підстановки у ключах маршрутизації забезпечує високу масштабованість системи, дозволяючи додавати нових споживачів фінансових подій без необхідності рефакторингу вихідного коду сервісів-відправників.

Загальна схема взаємодії побудована на використанні іменованих черг. Кожен мікросервіс ініціалізує власну чергу, яка виступає його персональною точкою входу:

- user-service слухає чергу `api_gateway_queue`,
- transaction-service - `transaction_queue`,

- goal-service - goal_queue,
- forecast-service - forecast_queue.

API Gateway функціонує як центр системи, що забезпечує первинну обробку та маршрутизацію запитів через уніфіковану точку входу. Така архітектурна децентралізація дозволяє розмежувати зони відповідальності мікросервісів та мінімізувати рівень жорсткої зв'язності між ними. Застосування асинхронної моделі взаємодії є важливим для підтримання високої продуктивності при виконанні складних аналітичних операцій. Наприклад, розрахунок прогнозних показників за методом лінійної регресії є обчислювально інтенсивним процесом, час виконання якого значно перевищує тривалість стандартних операцій запису чи оновлення даних. Використання черг повідомлень дозволяє сервісу транзакцій миттєво підтверджувати фіксацію фінансових записів, тоді як аналітичний модуль проводить необхідні розрахунки паралельно у фоновому режимі. Це унеможливлює виникнення затримок в інтерфейсі та гарантує стабільну швидкість роботи системи незалежно від поточного обчислювального навантаження на об'єктні моделі.

2.2.1 Паттерни проектування черг повідомлень

Залежно від характеру взаємодії між компонентами у системі застосовуються два різних патерни обміну повідомленнями. Для комунікації між API Gateway та мікросервісами використовується патерн Request/Reply, реалізований засобами фреймворку NestJS Microservices. Gateway надсилає повідомлення до черги відповідного сервісу та очікує на відповідь. З точки зору клієнта така взаємодія виглядає синхронною, однак на транспортному рівні вона є асинхронною - Gateway не блокує потік виконання, а лише підписується на відповідь у тимчасовій черзі зворотного зв'язку. Це дозволяє обробляти кілька запитів паралельно без створення окремих HTTP-з'єднань між сервісами.

Для міжсервісної взаємодії, ініційованої внутрішніми подіями системи, застосовується патерн Event-Driven. Після збереження нової транзакції `transaction-service` публікує подію `transaction_created_event_v1` без очікування будь-якої відповіді. `Goal-service` підписується на цю подію та автоматично перераховує прогрес фінансових цілей користувача. Та сама подія публікується і при видаленні транзакції, забезпечуючи актуальність даних у `goal-service`. Ключовою перевагою цього патерну є повна незалежність сервісів: `transaction-service` не знає про існування `goal-service`, що відповідає принципу слабого зв'язку в мікросервісній архітектурі та спрощує подальше розширення системи новими підписниками без зміни логіки видавця.

2.2.2 Забезпечення цілісності даних при асинхронній передачі

У розподілених системах забезпечення цілісності даних є складнішим завданням, ніж у монолітних. Транзакційні властивості ACID, які PostgreSQL гарантує в межах одного сервісу, не поширюються на міжсервісну взаємодію через брокер повідомлень. Оскільки кожен мікросервіс має власну базу даних, узгодженість між ними має забезпечуватися на рівні транспортного механізму.

У реалізованій системі надійність доставки забезпечується механізмом підтвердження обробки повідомлень, який NestJS використовує при взаємодії через RabbitMQ. Сервіс-споживач надсилає підтвердження лише після того, як повідомлення було успішно оброблено і дані збережено. У разі виникнення помилки або несподіваного завершення роботи сервісу повідомлення автоматично повертається до черги і буде повторно доставлено, щойно сервіс відновить роботу. Це гарантує, що жоден запит не буде втрачений навіть у випадку тимчасової недоступності одного з компонентів.

Оскільки збереження даних у різних сервісах відбувається не одночасно, система реалізує модель узгодженості в кінцевому підсумку. Це означає, що аналітичні дані у `goal-service` або `forecast-service` можуть оновлюватися з невеликою

затримкою відносно основних транзакційних даних. Однак завдяки механізму підтвердження доставки система гарантує, що всі події будуть опрацьовані і дані врешті-решт досягнуть узгодженого стану. Для фінансової системи такого рівня, де операції не є критично залежними від миттєвої синхронізації між модулями, ця модель є прийнятним компромісом між складністю реалізації та надійністю результату.

2.3 Контейнеризація системи

Для забезпечення портативності та ідентичності середовищ розробки і розгортання система була повністю контейнеризована засобами Docker. Це дозволило об'єднати всі п'ять мікросервісів, базу даних та брокер повідомлень в єдиний керований стек, ізольований від основної операційної системи. Завдяки цьому середовище, в якому система тестувалася під час розробки, є повністю ідентичним до того, в якому вона розгортається на будь-якій іншій машині.

2.3.1 Створення ізольованих середовищ для сервісів

Кожен мікросервіс системи - `api-gateway`, `user-service`, `transaction-service`, `goal-service` та `forecast-service` - має власний `Dockerfile`, розміщений у відповідній директорії проєкту. Для збірки образів обрано базовий образ `node:18-alpine`, що забезпечує компактне середовище виконання з мінімальною кількістю системних бібліотек. Це зменшує розмір фінального образу, скорочує час холодного старту сервісів та мінімізує поверхню атаки. Структура кожного `Dockerfile` реалізує чітку послідовність кроків: встановлення залежностей, компіляція TypeScript у JavaScript та запуск скомпільованого застосунку. Версії середовища виконання Node.js та всіх бібліотек жорстко зафіксовані в образі, що гарантує відтворюваність системи

незалежно від платформи розгортання. Кожен сервіс функціонує у повністю автономному оточенні - зміни у файловій системі або залежностях одного контейнера не впливають на інші. Параметри підключення до бази даних, адреси брокера повідомлень та секретні ключі JWT передаються до контейнерів через змінні середовища, визначені у файлі `.env`. Це розмежовує конфігураційні дані і програмний код, що відповідає принципам безпечного розгортання застосунків.

2.3.2 Конфігурація мережевої взаємодії в Docker Compose

Для керування багатоконтейнерною інфраструктурою використовується інструмент Docker Compose. Файл конфігурації `docker-compose.yml` є єдиною точкою налаштування всього стеку, оскільки він детально описує топологію сервісів, мережеву взаємодію між ними, томи для постійного збереження даних та чіткий порядок ініціалізації інфраструктурних компонентів.

Усі розроблені сервіси об'єднані у спільну віртуальну мережу `app-network`. У її межах контейнери звертаються один до одного за логічними іменами сервісів, визначеними у файлі конфігурації. Наприклад, мікросервіси підключаються до бази даних через хост `postgres`, а до брокера повідомлень через хост `rabbitmq`. Такий механізм виявлення сервісів через внутрішню систему доменних імен Docker повністю усуває потребу у жорсткому прив'язуванні до статичних IP-адрес і значно спрощує динамічне керування інфраструктурою комплексу.

Для зручного моніторингу статусу застосунку та управління запущеними контейнерами під час розробки використовується графічний інтерфейс Docker Desktop. Робоче середовище із запущеним комплексом мікросервісів представлено на рисунку 2.2.

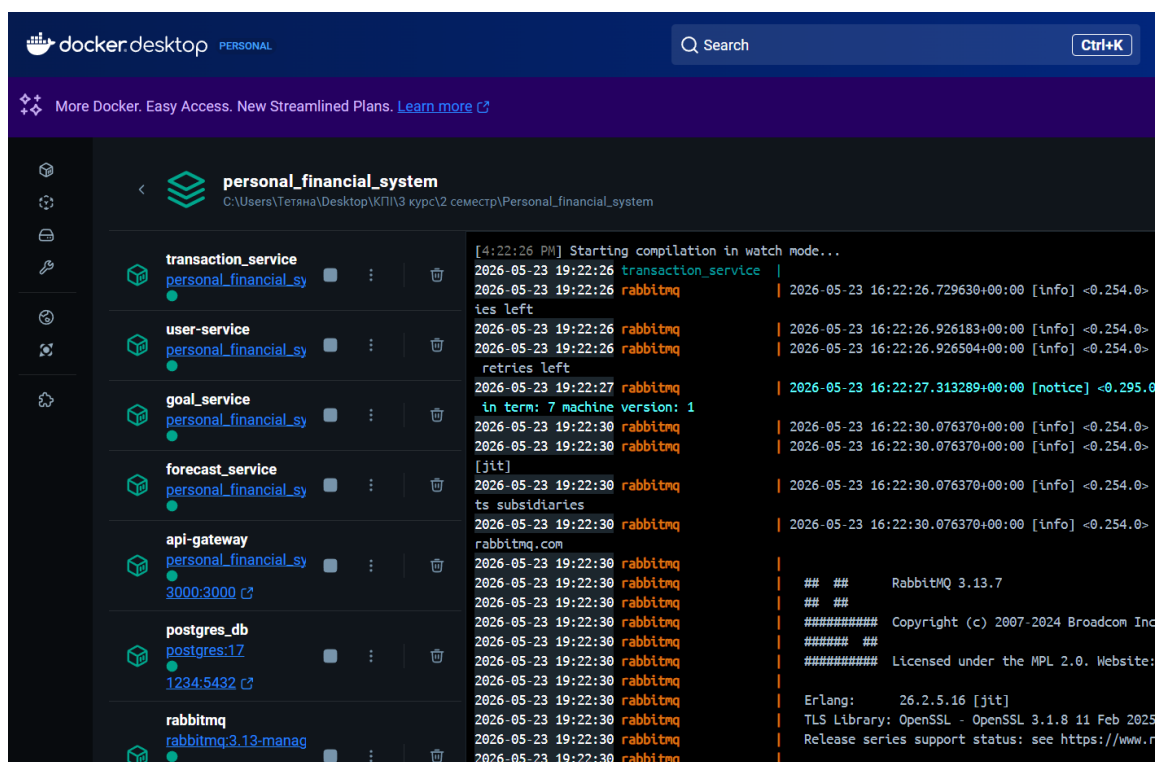


Рисунок 2.2 – Робоче середовище Docker Desktop із запущеними контейнерами системи

В архітектурі чітко розмежовані внутрішні та зовнішні порти. Назовні публікується лише порт API Gateway, тоді як PostgreSQL, RabbitMQ і самі мікросервіси залишаються доступними виключно всередині мережі Docker. Це суттєво зменшує поверхню атаки та унеможливорює прямий зовнішній доступ до внутрішніх компонентів системи.

Для збереження даних PostgreSQL використовується іменованний том `postgres_data`, змонтований у внутрішню директорію контейнера. Це гарантує, що дані зберігаються після перезапуску або повного перестворення контейнерів. Директива `depends_on` визначає послідовність ініціалізації: PostgreSQL та RabbitMQ стартують раніше за мікросервіси, що запобігає помилкам підключення при першому запуску системи.

2.4 Порівняльний аналіз методів прогнозування часових рядів

Ефективність інтелектуального модуля залежить від обраного методу математичного моделювання. В рамках дослідження було проведено порівняльний аналіз методів прогнозування, які потенційно могли бути застосовані в системі управління фінансами:

- Метод ковзного середнього. Найпростіший підхід, який базується на усередненні показників за певний період. Його перевагою є легкість реалізації, проте він має значну інерційність і не здатний виявляти тренди зростання або падіння витрат, що робить його малоефективним для стратегічного планування бюджету.

- Модель ARIMA (AutoRegressive Integrated Moving Average). Потужний статистичний інструмент для складних часових рядів. ARIMA враховує сезонність та нестационарність даних. Однак цей метод потребує значної кількості історичних записів для коректного налаштування параметрів. В умовах персонального фінансового обліку, де користувач може мати лише 5-10 місячних записів, ARIMA демонструє нестабільні результати.

- Нейронні мережі (LSTM/RNN). Рекурентні нейронні мережі здатні виявляти надскладні нелінійні залежності. Проте їх використання в даній системі є надлишковим, оскільки навчання моделі потребує значних обчислювальних ресурсів сервера і великих обсягів даних, що суперечить концепції мікросервісу.

- Метод лінійної регресії. Було обрано саме цей метод через його здатність чітко ідентифікувати вектор розвитку навіть на малих вибірках даних. Лінійна регресія забезпечує оптимальний баланс між швидкістю обчислень, що критично для stateless-сервісу, та точністю передбачень. Впровадження вагів для часових періодів дозволило нівелювати головний недолік регресії - чутливість до застарілих даних, надаючи пріоритет останній фінансовій поведінці користувача.

Узагальнену порівняльну характеристику розглянутих методів прогнозування за ключовими критеріями наведено у таблиці 2.1.

Таблиця 2.1 - Порівняльний аналіз методів прогнозування часових рядів

Критерій	Ковзне середнє	ARIMA	Нейронні мережі	Лінійна регресія
Мінімальна к-сть записів	3–5	24+	50+	2+
Швидкість обчислень	Висока	Низька	Низька	Висока
Точність на малих вибірках	Низька	Низька	Низька	Висока
Виявлення тренду	Ні	Так	Так	Так
Адаптивність до змін поведінки	Низька	Середня	Висока	Висока (з вагами)
Складність реалізації	Низька	Висока	Висока	Низька
Потреба у навчанні моделі	Ні	Так	Так	Ні

Порівняльний аналіз, узагальнений у таблиці 2.1, підтверджує що жоден з альтернативних методів не відповідає одночасно всім вимогам системи. ARIMA та нейронні мережі потребують значно більших обсягів даних і обчислювальних ресурсів, тоді як ковзне середнє не здатне виявляти тренди. Метод лінійної регресії з ваговими коефіцієнтами є єдиним підходом, який поєднує мінімальні вимоги до вхідних даних, високу швидкість обчислень та здатність адаптуватися до змін фінансової поведінки користувача.

2.5 Математичне забезпечення та алгоритм лінійної регресії

Предиктивний аналіз у розробленій системі базується на комплексній обробці накопичених історичних даних про витрати та доходи користувача. Головною метою цього процесу є виявлення прихованих закономірностей фінансової поведінки та точне визначення майбутніх трендів. Для математичної реалізації даного завдання було обрано метод лінійної регресії - класичний та надійний інструмент статистичного аналізу часових рядів. Він дозволяє формалізувати залежність між часовими періодами та грошовими сумами для подальшої екстраполяції даних на майбутнє. Використання саме цього алгоритму обґрунтовується тим, що він забезпечує оптимальний баланс між обчислювальною складністю та точністю результатів.

Математична модель простої лінійної регресії описує залежність між залежною змінною, а саме сумою витрат та незалежною змінною - часовим періодом у вигляді лінійного рівняння [16]:

$$y = \beta_0 + \beta_1 x + \varepsilon \quad (1)$$

де: y - прогнозована величина витрат;

x - часовий показник (порядковий номер місяця або дня);

β_0 - вільний член (точка перетину з віссю ординат), що відображає базовий рівень витрат;

β_1 - коефіцієнт регресії, що визначає нахил лінії тренду та інтенсивність зміни витрат у часі;

ε - випадкова похибка моделі.

Для знаходження оптимальних значень параметрів β_0 та β_1 у системі використано метод найменших квадратів (OLS). Суть методу полягає у мінімізації суми квадратів відхилень реальних значень y_i від теоретичних значень $\hat{y}_i$, отриманих за допомогою моделі:

$$S = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \rightarrow \min \quad (2)$$

В результаті математичного виведення формули для розрахунку коефіцієнтів набувають наступного вигляду:

$$\beta_1 = \frac{n \sum (x_i y_i) - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}, \quad \beta_0 = \bar{y} - \beta_1 \bar{x} \quad (3)$$

де $\bar{y}$ та $\bar{x}$ – середні арифметичні значення відповідних вибірок

Алгоритмічна реалізація в мікросервісі forecast-service складається з кількох послідовних етапів. На першому етапі виконується агрегація даних: транзакції користувача групуються за місяцями, і для кожного місяця розраховується сумарна величина витрат. На другому етапі проводиться перевірка вибірки на достатність - модель вимагає щонайменше двох точок даних для побудови тренду; якщо даних недостатньо, система повертає відповідний статус і прогноз не формується. На третьому етапі розраховуються коефіцієнти регресії та будується лінія тренду, яка екстраполюється на наступні часові періоди.

Для підвищення актуальності прогнозу реалізовано модифікований підхід, при якому вага спостережень залежить від їхньої давності - новіші дані мають більший вплив на результат. Це дозволяє моделі швидше адаптуватися до змін у фінансових звичках користувача.

Оцінка якості побудованої моделі здійснюється за допомогою коефіцієнта детермінації R^2 :

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2} \quad (4)$$

Коефіцієнт R^2 набуває значень від 0 до 1, де значення, близьке до одиниці, свідчить про високу точність апроксимації та надійність отриманого прогнозу. Результати прогнозування річного циклу витрат та розраховані статистичні показники представлено на рисунку 2.5.



Рисунок 2.5 – Графік апроксимації фінансових витрат методом найменших квадратів

Таким чином, обраний математичний апарат дозволяє не лише візуалізувати динаміку бюджету, а й формувати обґрунтовані прогнози.

3. РЕАЛІЗАЦІЯ ПРОГРАМНОЇ СИСТЕМИ

У третьому розділі наведено опис програмної реалізації системи управління персональними фінансами на основі раніше визначених архітектурних рішень. Основну увагу приділено практичному втіленню мікросервісів на базі фреймворку NestJS, організації доступу до даних через TypeORM та розробці клієнтського інтерфейсу засобами React.js. Для кожного компонента системи представлено структуру програмних модулів, опис класів-сутностей, а також логіку роботи ключових функцій та інтерфейсів. Описано практичні аспекти реалізації асинхронної взаємодії між модулями та механізми обробки фінансових даних у реальному часі.

3.1 Архітектура системи та проектування логіки

Даний підрозділ містить UML-діаграми, що описують статичну структуру системи та динаміку взаємодії між її компонентами, а також діаграми послідовності для ключових сценаріїв використання.

3.1.1 UML-діаграми (діаграма класів, компонентів)

Для візуалізації архітектурної структури та логічної моделі даних розробленої системи використано діаграми компонентів та класів. Фізичну організацію розподіленої архітектури, топологію розгортання мікросервісів та механізми їхньої комунікації відображає діаграма компонентів, яку наведено на рисунку 3.1

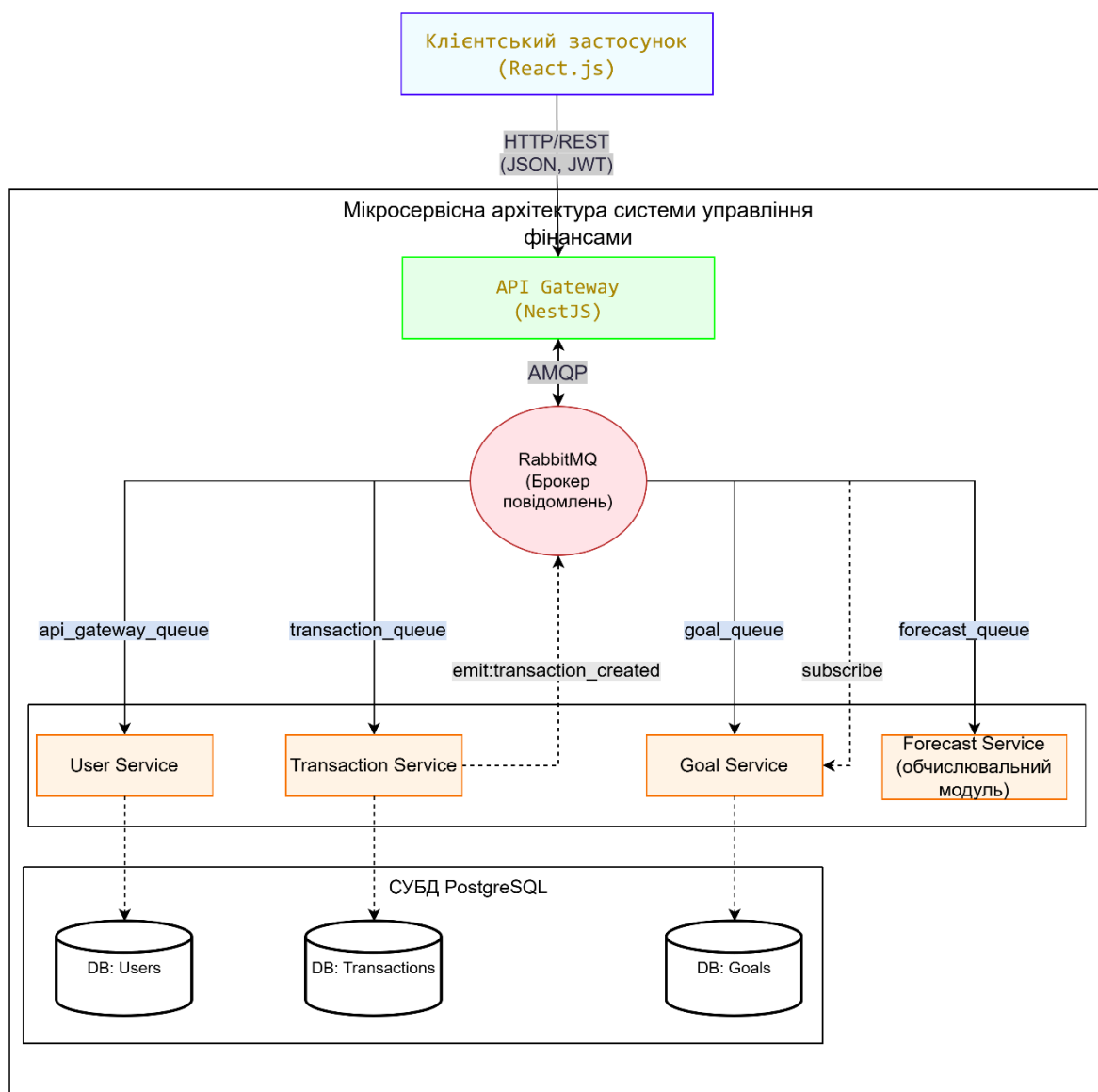


Рисунок 3.1 – Діаграма компонентів системи управління фінансами

Діаграма компонентів ілюструє архітектурну структуру системи та ключові зв'язки між її складовими:

1. Централізована маршрутизація: API Gateway виступає точкою входу, що з'єднує клієнтську частину з усіма внутрішніми мікросервісами через брокер повідомлень RabbitMQ.
2. Асинхронні черги: Для передачі команд використовуються виділені іменовані черги: `api_gateway_queue`, `transaction_queue`, `goal_queue` та `forecast_queue`.

3. Внутрішня взаємодія: Окрім обробки прямих запитів, сервіси комунікують між собою асинхронно. Наприклад, goal-service підписаний на подію transaction_created, яку генерує transaction-service, що дозволяє автоматично оновлювати стан фінансових цілей при додаванні нових операцій.

4. Ізоляція даних: Для дотримання принципу автономності кожен мікросервіс (за винятком обчислювального модуля forecast-service) взаємодіє з власною незалежною базою даних на базі СУБД PostgreSQL.

Представлена на рисунку 3.2 діаграма класів описує логічну структуру та внутрішні взаємозв'язки об'єктної моделі системи.

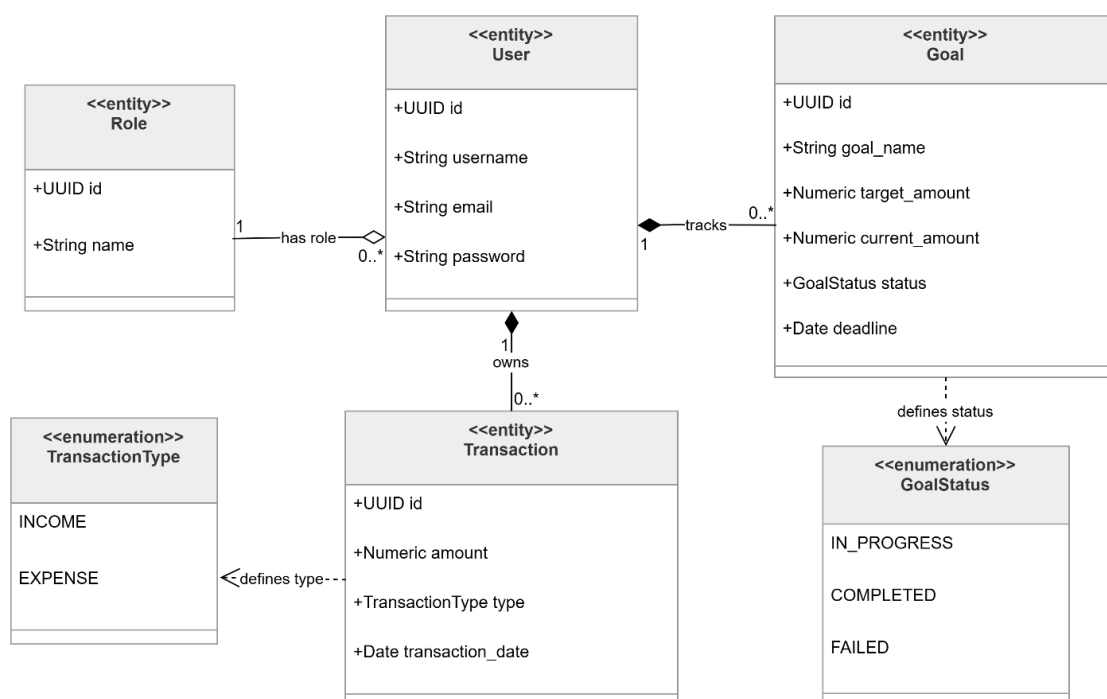


Рисунок 3.2 – Діаграма класів об'єктної моделі системи

Наведена діаграма класів відображає логічну будову системи та описує, як її основні частини взаємодіють між собою на рівні даних. У центрі моделі знаходиться клас користувача, який об'єднує навколо себе всі фінансові записи та цілі. Для зв'язку користувача з транзакціями та цілями використано відношення композиції, що позначається зафарбованим ромбом. Це вказує на повну залежність цих об'єктів від власника профілю, адже фінансові операції та цілі не можуть

існувати самостійно, тому при видаленні акаунта користувача вони автоматично видаляються разом із ним. Натомість зв'язок із класом ролей реалізовано через агрегацію, зображену порожнім ромбом. Це пояснюється тим, що ролі в системі є незалежними довідниками, які продовжують існувати та можуть призначатися іншим людям навіть після видалення конкретного профілю.

Окрему увагу на діаграмі приділено набору властивостей для кожного класу та їхнім типам. Для створення унікальних номерів усіх записів обрано формат UUID, який гарантує, що ідентифікатори не будуть повторюватися при роботі системи у розподіленому середовищі мікросервісів. Для збереження грошових сум використано тип Numeric, оскільки він забезпечує високу точність фінансових розрахунків і дозволяє уникнути помилок при математичних операціях. Також на схемі зображено спеціальні переліки для типів транзакцій та статусів цілей, що дозволяють суворо обмежити можливі стани об'єктів лише тими варіантами, які передбачені логікою програми. Пунктирні стрілки вказують на залежність основних класів від цих переліків.

3.1.2 Діаграма послідовності

Для опису динамічної взаємодії між компонентами системи та визначення життєвого циклу об'єктів у часі було розроблено діаграми послідовності для основних прецедентів.

Будь-яка взаємодія з системою розпочинається з ідентифікації суб'єкта та отримання прав доступу. Логіка цього процесу, що базується на делегуванні повноважень від шлюзу до сервісу користувачів, представлена на рисунку 3.3.

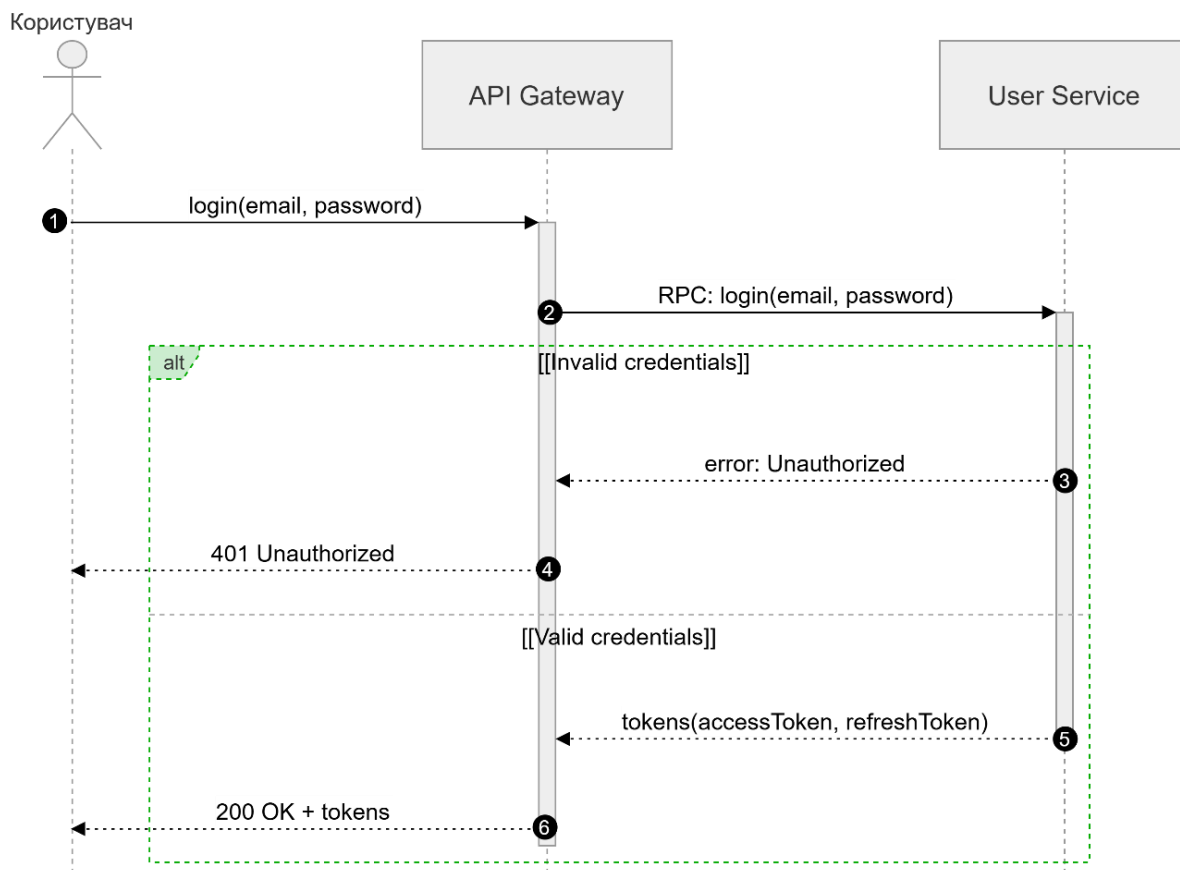


Рисунок 3.3 – Діаграма послідовності входу в систему

На наведеній схемі відображено роль API Gateway як захисного бар'єру, який приймає вхідні облікові дані та ініціює синхронний RPC-запит до User Service. Використання блоку альтернативи (alt) дозволяє візуалізувати розгалуження логіки: у разі успішної верифікації генерується пара JWT-токенів, що забезпечує стан авторизованого доступу для клієнтської частини.

Центральним процесом системи є облік транзакцій, який потребує негайного оновлення пов'язаних показників у інших модулях. Діаграма на рисунку 3.4 демонструє поєднання синхронного та асинхронного типів взаємодії.

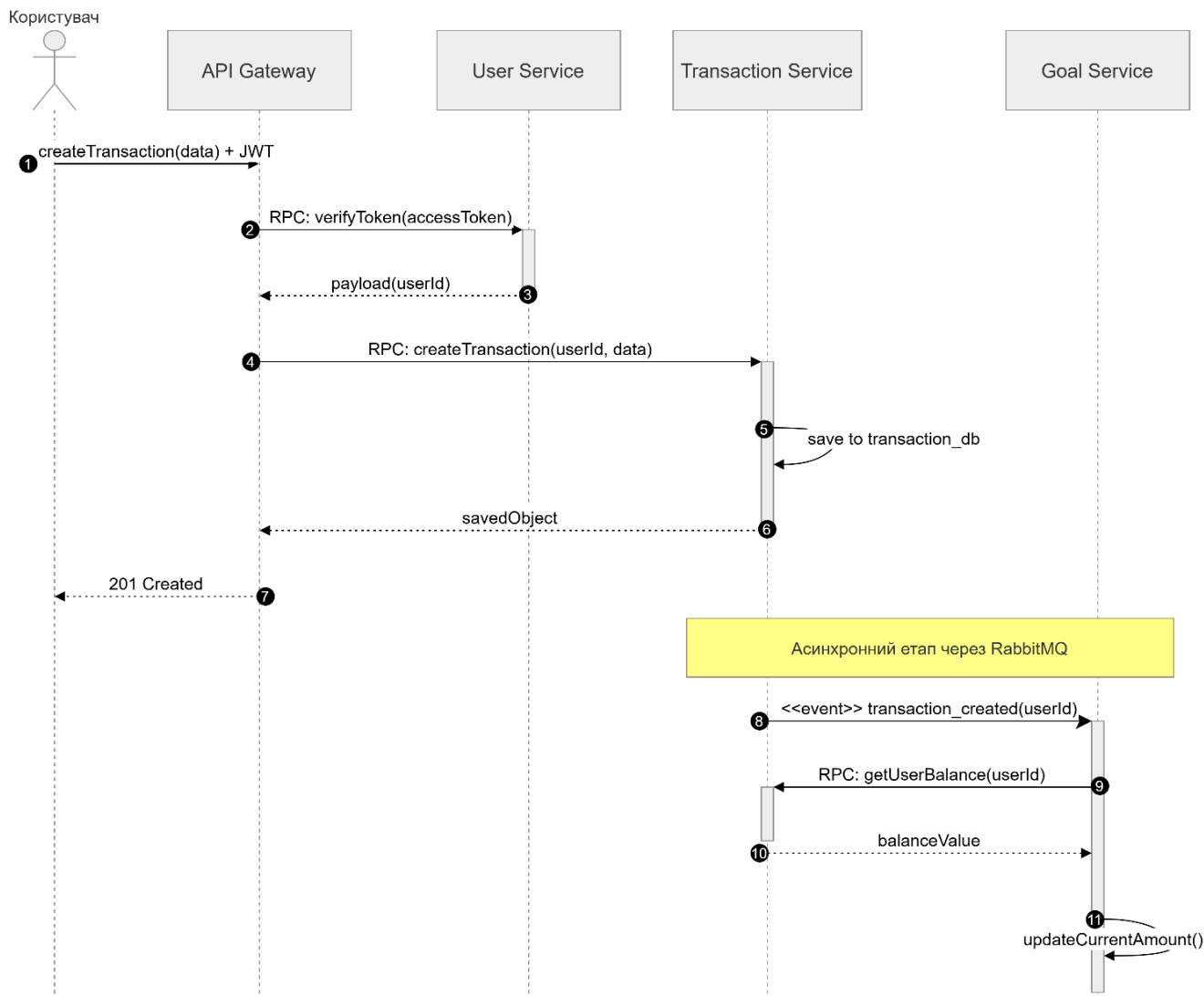


Рисунок 3.4 – Діаграма послідовності обробки транзакції та оновлення цілей

Після фіксації транзакції Transaction Service миттєво повертає підтвердження клієнту, одночасно ініціюючи асинхронне повідомлення у чергу RabbitMQ. Подальша активація сервісу цілей відбувається незалежно від основного потоку, де сервіс виконує рекурсивний запит до постачальника даних для отримання актуального агрегованого балансу. Це дозволяє реалізувати механізм оновлення стану системи без затримки відповіді користувачу.

Послідовність дій при створенні фінансової цілі та механізм перевірки прав доступу на рівні шлюзу представлено на рисунку 3.5.

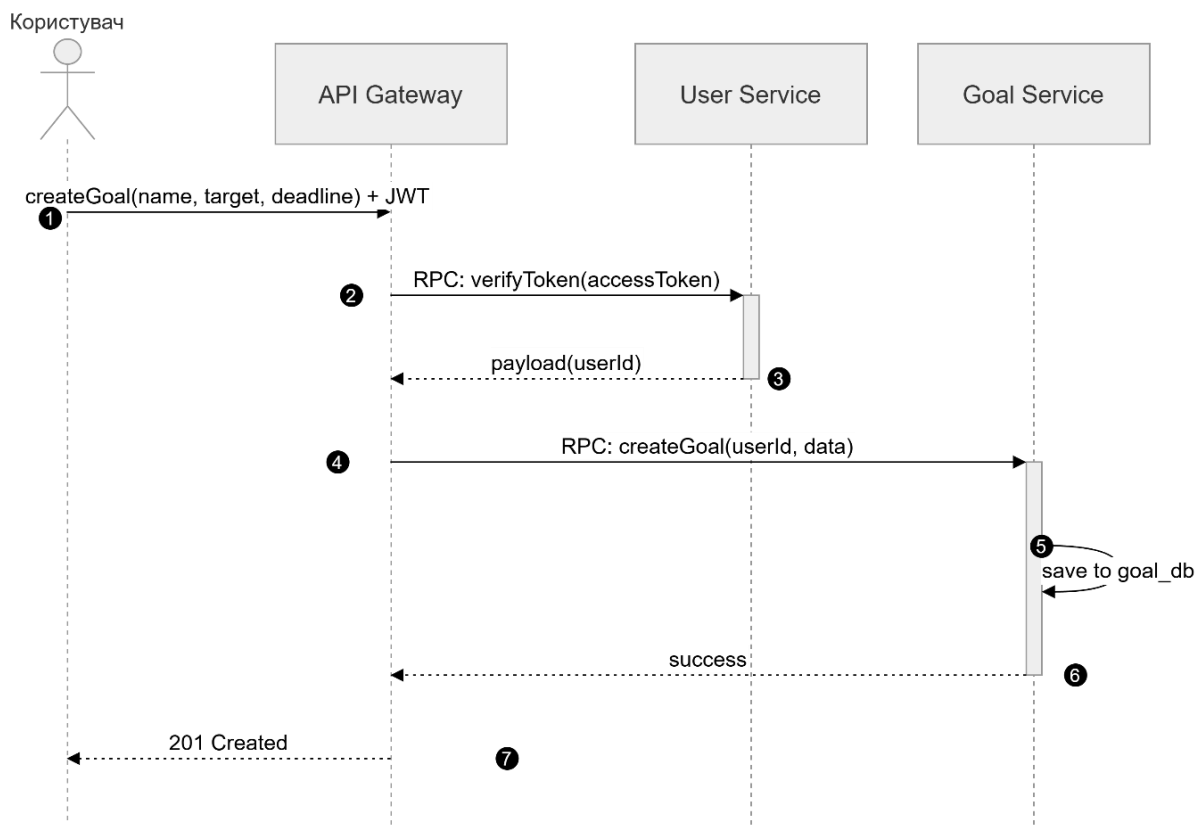


Рисунок 3.5 – Діаграма послідовності створення фінансової цілі

На схемі відображено, як API Gateway спочатку перевіряє токен через сервіс користувачів і лише після цього звертається до сервісу цілей. Такий порядок роботи гарантує, що мікросервіс отримує вже перевірений ідентифікатор власника, що забезпечує надійний захист даних.

3.2 Проектування бази даних

Збереження даних реалізовано за патерном «Database per Service», відповідно до якого мікросервіси для управління користувачами, транзакціями та фінансовими цілями використовують власні ізольовані бази даних PostgreSQL. Це забезпечує автономність компонентів і захищає інформацію від прямого доступу в обхід програмного шлюзу. Ініціалізація цих сховищ виконується автоматично при

першому запуску системи за допомогою спеціального скрипту `init-db.sh`. Він послідовно створює необхідні бази даних, підключає розширення `uuid-ossr` для генерації унікальних ідентифікаторів у кожній з них, після чого формує таблиці, визначає перелічувані типи та наповнює сховища початковими тестовими даними.

У базі даних `user_db` визначено зовнішній ключ між таблицями `users` та `roles` через поле `role_id`, що є єдиним реляційним зв'язком на фізичному рівні у всій системі. У базах `transaction_db` та `goal_db` поле `user_id` не має зовнішнього ключа на таблицю `users`, оскільки сховища повністю ізольовані. Зв'язок між сутностями різних сервісів підтримується виключно на рівні бізнес-логіки застосунку, де кожен сервіс самостійно передає необхідний ідентифікатор користувача при виконанні запитів. Для реалізації перелічуваних типів у PostgreSQL впроваджено власні структури, зокрема типи `transactions_type_enum` з варіантами `income` та `expense` у модулі транзакцій, а також `goals_status_enum` з варіантами `in_progress`, `completed` та `failed` у модулі цілей.

Взаємодія між мікросервісами та базами даних реалізована через об'єктно-реляційну бібліотеку `TypeORM` [17]. Кожна сутність описана у вигляді класу мови `TypeScript` із декораторами, що визначають відповідність між полями об'єкта і стовпцями таблиці. Наприклад, декоратор `Entity` позначає клас як сутність бази даних, декоратор `PrimaryGeneratedColumn` із параметром `uuid` вказує на автоматичну генерацію первинного ключа, а декоратор `Column` описує тип і обмеження конкретного поля. Такий підхід дозволяє працювати з інформацією через об'єкти без написання прямих SQL-запитів, зберігаючи при цьому повний контроль над структурою сховища. Для детальної ілюстрації взаємодії з таблицями на рисунку 3.6 наведено фрагмент програмної реалізації класу сутності для обліку фінансових операцій.

```

@Entity('transactions')
export class Transaction {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'uuid' })
  user_id: string;

  @Column({ type: 'decimal', precision: 10, scale: 2 })
  amount: number;

  @Column({ type: 'enum', enum: ['income', 'expense'] })
  type: 'income' | 'expense';

  @Column({ type: 'varchar', length: 255 })
  category: string;

  @Column({ type: 'text', nullable: true })
  description: string;

  @Column({ type: 'timestamp' })
  transaction_date: Date;

  @Column({ type: 'timestamp', default: () => 'CURRENT_TIMESTAMP' })
  created_at: Date;
}

```

Рисунок 3.6 – Сутність транзакції на базі бібліотеки TypeORM

Зображений на рисунку 3.6 декоратор первинного ключа автоматично генерує унікальний ідентифікатор стандарту UUID при створенні кожного запису, що повністю унеможлиблює появу колізій у розподіленому середовищі мікросервісів. Тип даних для збереження суми вказується як numeric з чітким визначенням двох знаків після коми. Це гарантує високу точність фінансових розрахунків і надійно захищає систему від похибок округлення, які є характерними для стандартного типу float і вважаються критичними для будь-яких грошових операцій.

3.3 Реалізація User Service та механізмів автентифікації

User-service відповідає за управління обліковими записами користувачів та забезпечення безпечного доступу до системи. Сервіс складається з двох логічно розділених модулів - управління користувачами та автентифікації, що дозволяє

розподіляти відповідальність між операціями з обліковими даними і операціями з токенами.

Усі операції сервісу доступні виключно через RabbitMQ, отже HTTP-інтерфейсу не передбачено. Такий підхід робить неможливим прямий зовнішній доступ до сервісу в обхід API Gateway, що підвищує рівень безпеки системи. Операції управління користувачами охоплюють реєстрацію, вхід, отримання профілю, оновлення даних, зміну та скидання пароля і видалення облікового запису. Модуль автентифікації окремо обробляє генерацію токенів, їх верифікацію та оновлення через refresh-токен.

При реєстрації нового користувача сервіс спочатку перевіряє існування вказаної ролі в базі даних і лише після цього створює обліковий запис. Одразу після збереження користувача генерується пара токенів і повертається клієнту, таким чином реєстрація і автоматичний вхід відбуваються в межах однієї операції без потреби виконувати окремий запит на авторизацію.

Автентифікація побудована на основі механізму JSON Web Token [18]. JWT - це відкритий стандарт передачі інформації між сторонами у вигляді підписаного токена, що складається з трьох частин: заголовка з алгоритмом підпису, корисного навантаження з даними користувача та підпису, сформованого на основі секретного ключа. Підробити токен без знання цього ключа неможливо, що робить його надійним механізмом підтвердження особи користувача.

У системі реалізовано схему з двома токенами. Access-токен має короткий термін дії тривалістю одну добу і передається клієнтом у заголовку кожного захищеного запиту. Його корисне навантаження містить ідентифікатор користувача та його роль, що дозволяє API Gateway перевіряти права доступу без додаткових звернень до бази даних. Refresh-токен діє сім днів і використовується виключно для отримання нової пари токенів після закінчення терміну дії access-токена. Така схема мінімізує ризики і навіть якщо access-токен буде перехоплений злоумисником, він стане недійсним через короткий проміжок часу, тоді як refresh-токен зберігається і використовується значно рідше.

Верифікація токена відбувається при кожному захищеному запиті на рівні API Gateway. AuthGuard витягує токен з заголовка запиту і надсилає його до user-service через RabbitMQ. Сервіс перевіряє підпис і термін дії токена та повертає декодований payload або помилку. Централізація логіки верифікації в одному сервісі виключає дублювання коду перевірки в інших мікросервісах і гарантує єдину точку управління політикою доступу.

3.4 Реалізація Transaction Service для обліку фінансових операцій

Transaction-service є центральним сервісом системи оскільки саме через нього проходять усі фінансові операції користувача, і від коректності його роботи залежить правильність даних у goal-service та forecast-service.

Особливістю архітектури сервісу є те, що він одночасно виконує дві ролі: приймає команди від API Gateway і сам ініціює взаємодію між сервісами, публікуючи події для goal-service після кожної зміни даних. Це нетипова роль для сервісу в мікросервісній архітектурі, де компоненти зазвичай лише відповідають на запити, проте саме такий підхід необхідний для того, щоб оновлення фінансових цілей відбувалося без участі клієнта.

Створення транзакції побудоване з урахуванням порядку операцій: спочатку запис зберігається в базі даних, і лише після підтвердження збереження публікується подія для goal-service. Дотримання такого порядку дозволяє уникнути розбіжностей: якби подія публікувалася до фактичного збереження, сервіс цілей міг би оновити баланс за даними, які фізично ще відсутні в системі. Видалення транзакції побудоване за тією самою логікою: перед видаленням сервіс зчитує запис щоб отримати ідентифікатор користувача, потім видаляє і публікує ту саму подію. Використання однієї події як для створення, так і для видалення є свідомим рішенням тому що goal-service не потребує знати про тип операції, оскільки в обох

випадках його ціллю є отримати актуальний баланс і перерахувати прогрес цілей. Отримання списку транзакцій підтримує комбіновану фільтрацію за типом операції, категорією та місяцем, де кожен параметр є необов'язковим. Фільтрація реалізована на рівні бази даних. Замість того щоб завантажувати всі транзакції і фільтрувати їх у пам'яті застосунку, сервіс формує запит з необхідними умовами і отримує вже відібрані дані. При великій кількості записів це суттєво впливає на продуктивність. Підрахунок балансу реалізований як агрегуючий запит, де PostgreSQL самостійно підсумовує доходи і витрати та повертає одне числове значення. Зведення транзакцій поєднує місячну та загальну статистику в одній відповіді, що дозволяє клієнту за один запит отримати всі необхідні дані для відображення дашборду без додаткових звернень до сервера.

3.5 Реалізація Goal Service для моніторингу та управління фінансовими цілями

Goal-service відповідає за управління фінансовими цілями користувача. На відміну від інших сервісів системи, він не лише обслуговує команди від API Gateway, а й реагує на внутрішні події системи. Сервіс підтримує повний цикл управління цілями: створення нової цілі з вказанням назви, цільової суми та дедлайну, отримання списку всіх цілей користувача, оновлення параметрів існуючої цілі та її видалення. На рівні API Gateway кожна з цих операцій захищена таймаутом у п'ять секунд, якщо сервіс не відповідає протягом цього часу, клієнт отримує повідомлення про помилку замість зависання запиту.

Центральною з архітектурної точки зору є логіка автоматичного перерахунку прогресу цілей. Сервіс підписується на подію `transaction_created_event_v1` через декоратор `@EventPattern`, що принципово відрізняється від звичайного `@MessagePattern` - відправник не очікує відповіді, і перерахунок відбувається повністю у фоновому режимі, не блокуючи жодних операцій. Щоразу коли

надходить така подія, сервіс витягує з неї ідентифікатор користувача і запускає процес оновлення. Процес оновлення складається з кількох послідовних кроків. Спочатку goal-service звертається до transaction-service через RabbitMQ і запитує актуальний баланс користувача. Після отримання відповіді він завантажує всі цілі цього користувача зі своєї бази даних і оновлює поле `current_amount` у кожній з них. При оновленні застосовується логіка обмеження, тобто якщо баланс перевищує цільову суму, поточна сума фіксується на рівні цілі, а не продовжує зростати.

Важливо розуміти модель даних, яку реалізує сервіс: `current_amount` відображає не кошти, відкладені спеціально на конкретну ціль, а загальний фінансовий баланс користувача на момент останнього перерахунку. Тобто всі активні цілі користувача мають однакове значення `current_amount`, що дорівнює його поточному балансу. Система показує наскільки користувач наблизився до кожної цілі виходячи із загального стану його фінансів, не розподіляючи кошти між цілями вручну.

На рівні клієнтської частини сторінка цілей додатково обчислює зведену статистику на основі отриманих даних: кількість активних та завершених цілей, загальний відсоток прогресу по всіх активних цілях та загальну суму яку ще потрібно накопичити. Ці обчислення виконуються на стороні клієнта без додаткових запитів до сервера, що зменшує навантаження на систему.

3.6 Реалізація Forecast Service для предиктивного аналізу та прогнозування витрат

Forecast-service є обчислювальним модулем системи, що відповідає за аналіз фінансової історії користувача та формування прогнозу на наступний місяць. На відміну від інших сервісів, він не має власної бази даних, а натомість отримує необхідні дані від transaction-service через RabbitMQ і виконує всі обчислення в пам'яті. Сервіс підтримує два режими роботи. Перший - відповідь на прямий запит

від API Gateway, коли користувач відкриває сторінку аналітики. Другий - реакція на подію `transaction_created_event_v1`, що дозволяє оновлювати прогноз у фоновому режимі при кожній новій транзакції. Обидва режими викликають один і той самий метод `getForecast`, щоб уникнути дублювання логіки.

Процес формування прогнозу починається із запиту всіх транзакцій користувача від `transaction-service`. Після отримання даних сервіс відфільтровує транзакції поточного місяця - до уваги беруться лише завершені місяці, оскільки включення неповного поточного місяця у розрахунок призводить до спотворених результатів. Відфільтровані транзакції агрегуються по місяцях: для кожного місяця окремо підраховується сума доходів і сума витрат. Отримані місячні значення сортуються хронологічно і передаються до алгоритму прогнозування.

Алгоритм прогнозування реалізований як зважена лінійна регресія. На відміну від класичної регресії де всі точки мають однакову вагу, у даній реалізації кожному спостереженню присвоюється вага що дорівнює його порядковому номеру, таким чином останні місяці мають більший вплив на результат ніж давніші. Для підвищення стійкості моделі до аномальних значень перед розрахунком застосовується фільтрація викидів на основі медіани: якщо місячний показник перевищує медіанне значення вибірки більш ніж утричі, він замінюється медіаною. Це дозволяє нівелювати вплив одноразових великих витрат, таких як придбання автомобіля чи техніки, на загальний прогноз бюджету. Коефіцієнти нахилу і зсуву розраховуються за формулами методу найменших квадратів з урахуванням ваг, після чого лінія тренду екстраполюється на наступний часовий період. Якщо отримане прогнозне значення від'ємне, воно замінюється нулем, адже від'ємний прогноз витрат або доходів не має економічного сенсу. У випадку коли всі місячні значення однакові і знаменник формули регресії наближається до нуля, сервіс повертає останнє відоме значення як прогноз, уникаючи ділення на нуль.

Окрім загального прогнозу доходів і витрат, сервіс формує прогноз витрат у розрізі категорій. Для кожної категорії підраховується загальна сума витрат за всю історію і ділиться на кількість місяців, що дає середньомісячне значення по

категорії. У відповідь повертається п'ять категорій з найбільшими прогнозованими витратами, що дозволяє користувачу зрозуміти на що він витрачає найбільше. Відповідь сервісу містить прогнозовані суми доходів і витрат на наступний місяць, прогноз по категоріях, прапорець `hasEnoughData` що вказує чи достатньо даних для побудови надійного прогнозу, кількість місяців використаних для розрахунку та час останнього оновлення. Якщо транзакцій немає або всі вони відносяться до поточного місяця, сервіс повертає порожню відповідь з нульовими значеннями замість помилки.

3.7 Реалізація графічного інтерфейсу користувача та засобів візуалізації фінансової статистики

Клієнтська частина системи реалізована як Single Page Application на основі бібліотеки React. Застосунок має п'ять основних сторінок і набір компонентів, що забезпечують єдиний візуальний стиль інтерфейсу. Навігація між сторінками реалізована через керування станом у кореневому компоненті [19] без використання зовнішніх бібліотек маршрутизації. Захищені сторінки відображаються лише для авторизованого користувача, а спроба отримати до них доступ без активної сесії автоматично перенаправляє на сторінку входу. Уся взаємодія з API Gateway проходить через централізований модуль авторизованих запитів. Він автоматично додає токен доступу до кожного запиту і коли токен закінчується, то клієнт непомітно для користувача оновлює сесію через `refresh-token` і повторює запит. Токени зберігаються між сесіями, тому після перезавантаження сторінки користувач залишається авторизованим. Ідентифікатор користувача зчитується безпосередньо з `payload` JWT-токена на стороні клієнта без окремого звернення до сервера.

Дашборд є головною сторінкою застосунку і найбагатшою за функціоналом. При відкритті вона завантажує загальний баланс, транзакції за обраний місяць,

місячну статистику та прогноз на наступний місяць. Транзакції можна фільтрувати за типом та категорією прямо на сторінці без додаткових запитів до сервера, а перелік доступних категорій формується динамічно з поточних даних. Додавання нової транзакції відбувається через модальне вікно. Для наочності фінансової картини на сторінці розміщено два графіки: лінійний графік динаміки доходів і витрат за поточний місяць та кругова діаграма розподілу витрат за категоріями.

Сторінка аналітики відображає детальні результати роботи алгоритму прогнозування. Три картки у верхній частині показують очікуваний дохід, витрати та залишок на наступний місяць, а горизонтальний стовпчастий графік розкриває розподіл витрат за категоріями. Система аналізує співвідношення доходів і витрат, виявляє найбільшу категорію витрат і формує конкретні поради для користувача. При від'ємному прогнозі користувач отримає попередження про дефіцит, а при залишку більше 30% доходу - пропозицію спрямувати кошти на досягнення фінансових цілей.

Сторінка цілей відображає список фінансових цілей з прогрес-баром для кожної. Зверху виводиться зведена статистика по всіх цілях, а саме кількість активних, загальний відсоток прогресу і сума яку ще потрібно накопичити. Інтерфейс оновлюється миттєво при будь-яких змінах без очікування відповіді від сервера. Сторінка налаштувань надає можливість змінити пароль та видалити обліковий запис, який захищений модальним вікном підтвердження.

Повторювані елементи інтерфейсу, такі як картки статистики, таблиця транзакцій, форми, фільтри, селектор місяця та модальні вікна винесені в окремі компоненти для спрощення їх повторного використання на різних сторінках. Інтерфейс виконаний у єдиній темній темі, стилізація побудована на системі CSS-змінних, а графіки реалізовані за допомогою бібліотеки Recharts [20] і підтримують інтерактивні підказки при наведенні.

4. ТЕСТУВАННЯ ТА АНАЛІЗ РОБОТИ СИСТЕМИ УПРАВЛІННЯ ФІНАНСАМИ

У четвертому розділі висвітлено етапи розгортання та перевірки працездатності розробленої програмної системи. Визначено апаратні й програмні вимоги до середовища експлуатації та описано процес автоматизованої інсталяції комплексу за допомогою засобів контейнеризації. Також проведено комплексне функціональне тестування, що супроводжується покроковою демонстрацією ключових користувацьких сценаріїв: від авторизації та фіксації фінансових транзакцій до автоматичного перерахунку цілей і формування предиктивної аналітики.

4.1 Інсталяція програмного забезпечення та вимоги до середовища

Завдяки повній контейнеризації системи процес її розгортання є максимально спрощеним. Єдиною програмною вимогою є наявність встановленого Docker на цільовій машині - окрема інсталяція Node.js, PostgreSQL чи RabbitMQ не потрібна, оскільки всі залежності вже включені у відповідні образи контейнерів. Система сумісна з операційними системами Windows 10/11, macOS та дистрибутивами Linux. З точки зору апаратних ресурсів система одночасно підтримує сім контейнерів, тому для стабільної роботи рекомендується не менше 8 ГБ оперативної пам'яті та не менше 3 ГБ вільного місця на диску для зберігання образів. Перед першим запуском необхідно налаштувати файл .env у кореневій директорії проєкту. Після налаштування конфігурації весь стек запускається однією командою `docker-compose up --build`. Система автоматично збирає образи сервісів, запускає контейнери у правильній послідовності та ініціалізує бази даних.

4.2 Демонстрація функціоналу та сценарії роботи системи

Для підтвердження коректності реалізації всіх заявлених вимог було проведено функціональне тестування системи шляхом відтворення типових сценаріїв взаємодії користувача з застосунком.

Тестування розпочалося з перевірки модуля реєстрації. На сторінці реєстрації користувач вводить ім'я, електронну пошту та пароль. Після успішної реєстрації система автоматично виконує вхід і перенаправляє користувача на головну сторінку без потреби повторно вводити облікові дані. Для вже зареєстрованих користувачів передбачена окрема сторінка входу. Процес реєстрації та входу зображений на рисунку 4.1.

Рисунок 4.1 – Сторінка реєстрації та входу

Після авторизації користувач потрапляє на головну сторінку застосунку – дашборд (рисунок 4.2). Було перевірено що при зміні обраного місяця через селектор усі дані на сторінці оновлюються одночасно і відображають коректні значення для обраного періоду.

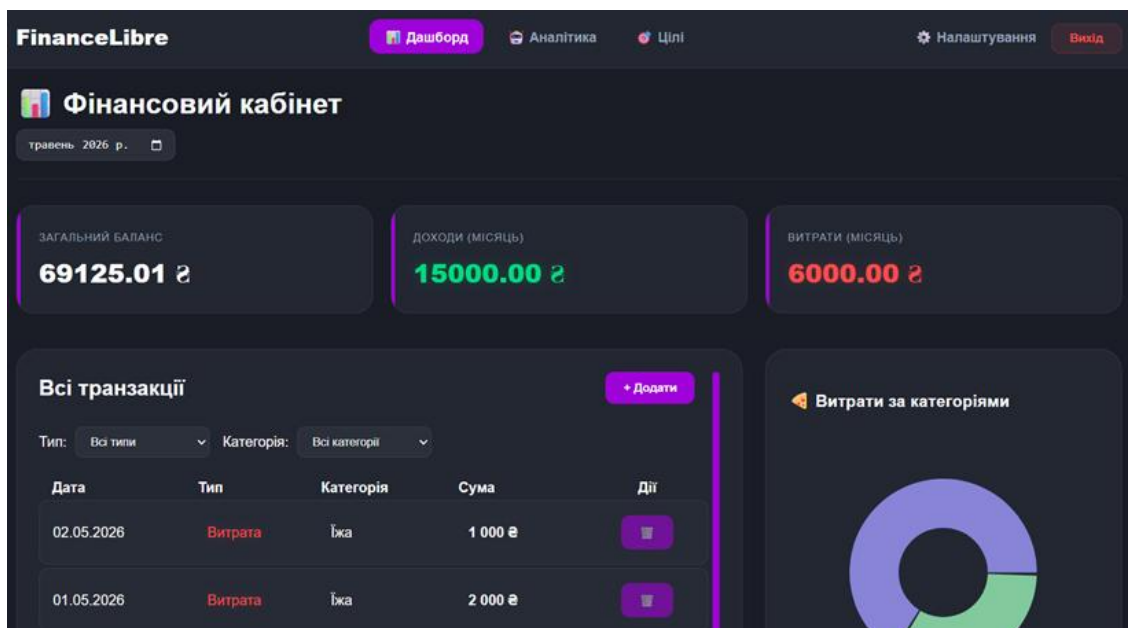


Рисунок 4.2 – Дашборд

Наступним кроком перевірялось додавання нової транзакції. Після натискання відповідної кнопки відкривається модальне вікно(рисунок 4.3), де користувач вводить суму, обирає тип операції, категорію та дату.

The screenshot shows a modal window titled 'Нова транзакція'. It contains several input fields: 'Сума:' with the value '2910', 'Тип:' with a dropdown menu showing 'Дохід', 'Категорія:' with a dropdown menu showing 'Стипендія', and 'Дата:' with a date picker showing '09.05.2026'. At the bottom, there are two buttons: 'Зберегти' and 'Скасувати'.

Рисунок 4.3 – Модальне вікно транзакції

Було підтверджено що після збереження транзакція миттєво з'являється у таблиці, загальний баланс та місячна статистика перераховуються коректно, а графіки оновлюються відповідно до нових даних(рисунок 4.4).

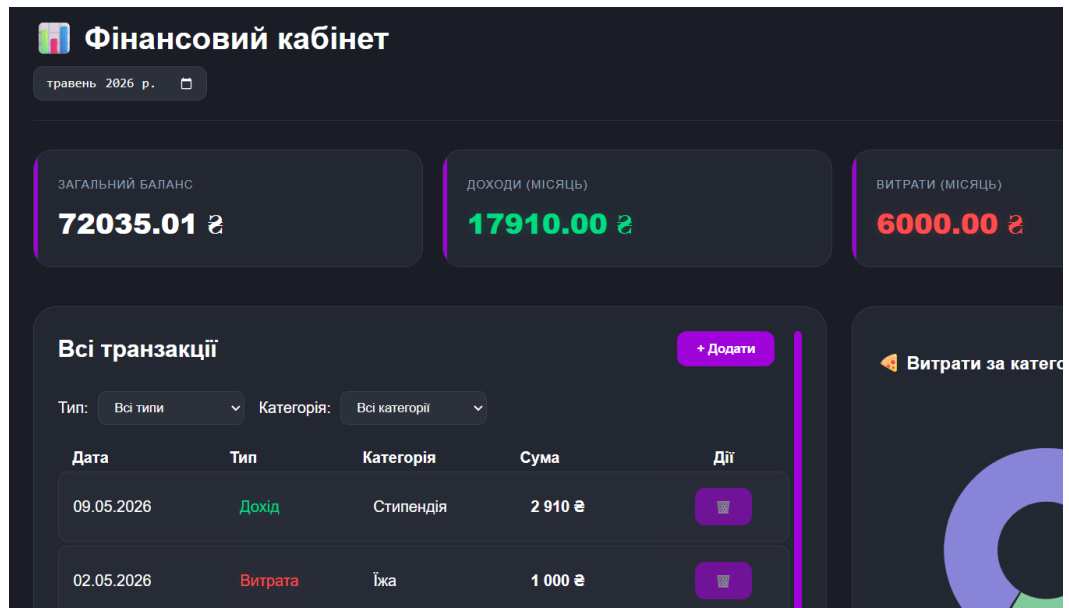


Рисунок 4.4 – Оновлена сторінка після додавання транзакції

Паралельно з тестуванням транзакцій перевірялась коректність міжсервісної взаємодії. При кожному додаванні або видаленні транзакції goal-service автоматично отримує подію через RabbitMQ і перераховує прогрес активних цілей. Сторінка цілей з оновленим балансом зображена на рисунку 4.5.

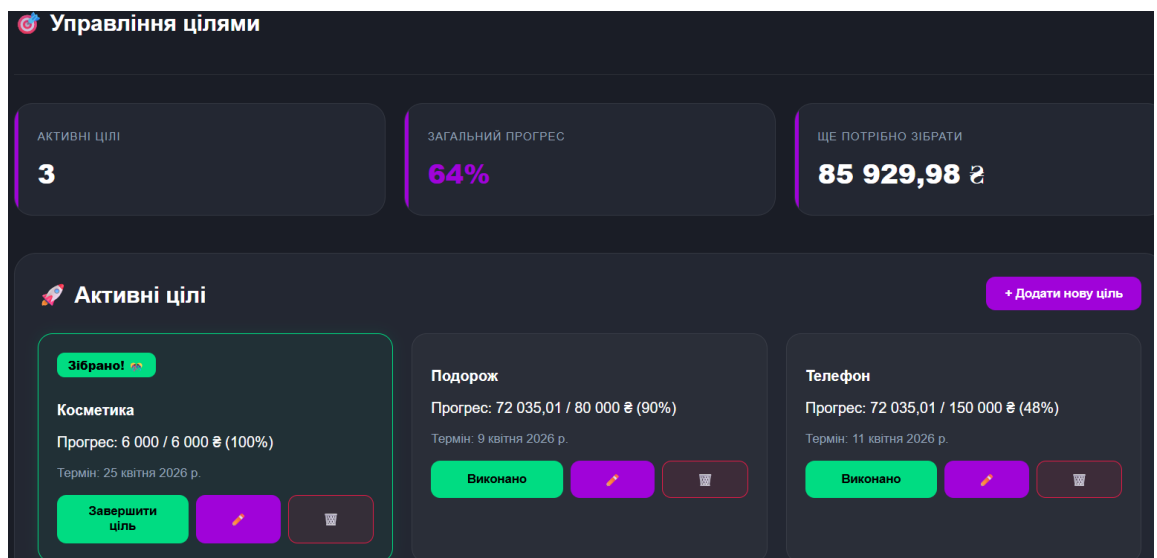


Рисунок 4.5 – Сторінка цілей

Наступним стало тестування предиктивного модуля на сторінці аналітики (рисунок 4.6). Перевірка підтвердила, що алгоритм зваженої лінійної регресії коректно розраховує майбутні фінансові тренди.

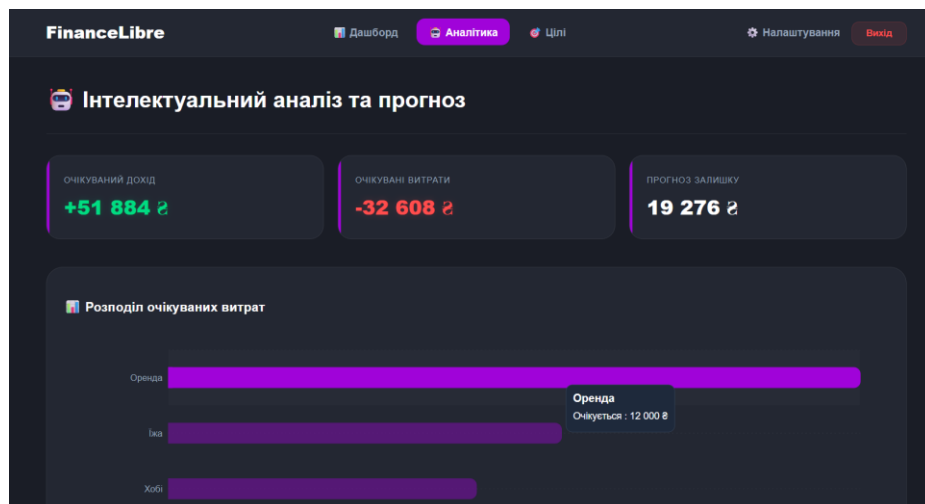


Рисунок 4.6 – Сторінка аналітики

Завершальним етапом стала перевірка сторінки налаштувань (рисунок 4.7). Зміна пароля та видалення аккаунту відпрацювало коректно.

The screenshot shows the 'Налаштування профілю' (Profile Settings) page. It is divided into two main sections: 'БЕЗПЕКА' (Security) and 'ВИДАЛЕННЯ АКАУНТУ' (Account Deletion). Under 'БЕЗПЕКА', there's a 'Новий пароль' (New Password) section with a text input field 'Введіть новий пароль' and a 'Підтвердження пароля' (Password Confirmation) section with a text input field 'Повторіть пароль'. A large orange button labeled 'Оновити пароль' (Update Password) is positioned below these fields. Under 'ВИДАЛЕННЯ АКАУНТУ', there's a warning message: 'Ця дія призведе до безповоротного видалення ваших даних.' (This action will lead to the irreversible deletion of your data.) and a red button labeled 'Видалити акаунт' (Delete Account).

Рисунок 4.7 – Сторінка налаштувань

Отже, система відповідає визначеним функціональним вимогам.

ВИСНОВКИ

На основі проведеного аналізу підходів, методів та алгоритмів розв’язання поставленої задачі було обґрунтовано доцільність використання мікросервісної архітектури, що дозволило забезпечити високу відмовостійкість та незалежне масштабування обчислювальних вузлів системи. Математичне забезпечення предиктивного модуля базується на методі зваженої лінійної регресії, який продемонстрував здатність ефективно ідентифікувати фінансові тренди на малих та великих вибірках даних. Використання механізмів асинхронної трансляції подій через брокер повідомлень RabbitMQ дозволило реалізувати динамічну синхронізацію станів між ізольованими базами даних. Це забезпечило дотримання концепції узгодженості в кінцевому підсумку, що є критично важливим для стабільної роботи розподіленої фінансової платформи.

За результатами аналізу програмних засобів для реалізації системи було обрано та впроваджено сучасний технологічний стек, який включає платформу Node.js із фреймворком NestJS для побудови серверної логіки та бібліотеку React.js для розробки динамічного інтерфейсу користувача. Для забезпечення надійного зберігання даних та виконання принципів ACID обрано реляційну СУБД PostgreSQL, а процеси розгортання та оркестрації інфраструктури автоматизовано за допомогою платформи Docker та інструменту Docker Compose. Застосування мови TypeScript на всіх рівнях розробки дозволило впровадити сувору типізацію даних, що суттєво знизило ймовірність виникнення помилок у міжсервісній комунікації та підвищило загальну якість програмного коду.

У ході роботи було розроблено комплексну програмну систему управління персональними фінансами «FinanceLibre», яка складається з п’яти автономних сервісів та реалізує повний цикл фінансового менеджменту. Система дозволяє користувачу проводити облік доходів і витрат, встановлювати довгострокові фінансові цілі та отримувати інтелектуальні прогнози на майбутні періоди. Практична цінність розробки полягає в реалізації інтелектуального модуля, який

трансформує результати математичного аналізу в персоналізовані фінансові рекомендації, допомагаючи користувачу оптимізувати витрати та ефективно розподіляти власний бюджет.

На основі проведеного тестування було доведено коректність роботи всіх компонентів розробленого програмного забезпечення та верифіковано сценарії асинхронної взаємодії в контейнеризованому середовищі. Обчислювальні експерименти на реальних наборах даних підтвердили переваги зваженої моделі прогнозування, яка забезпечує на вищу точність порівняно зі стандартними статистичними методами за рахунок адаптивності до останніх змін у поведінці користувача. Подальший розвиток системи вбачається у впровадженні методів глибокого навчання для аналізу нелінійних фінансових патернів та інтеграції з банківськими API для автоматизації процесів збору даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бланк І. О. Управління фінансами: навчальний посібник. Київ: Київ. нац. торг.-екон. ун-т, 2021. 450 с.
2. Лук'яненко І. Г., Краснікова Л. І. Економетрика та прогнозування часових рядів: підручник. Київ, 2022. 315 с.
3. Hyndman R. J., Athanasopoulos G. Forecasting: principles and practice, 3rd edition. OTexts: Melbourne, Australia, 2021. URL: <https://otexts.com/fpp3> (дата звернення: 29.05.2026).
4. Fowler M., Lewis J. Microservices. ThoughtWorks, 2014. URL: <https://martinfowler.com/articles/microservices.html> (дата звернення: 29.05.2026).
5. Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2021. 614 p.
6. Node.js Documentation. Офіційний сайт Node.js. URL: <https://nodejs.org> (дата звернення: 10.05.2026).
7. NestJS Documentation. Офіційний сайт NestJS. URL: <https://nestjs.com> (дата звернення: 29.05.2026).
8. Richardson C. Microservices Patterns: With examples in Java. Manning Publications, 2018. 520 p.
9. TypeScript: Typed JavaScript at Any Scale. Офіційний сайт TypeScript. URL: <https://www.typescriptlang.org> (дата звернення: 05.05.2026).
10. React – A JavaScript library for building user interfaces. Офіційний сайт React. URL: <https://reactjs.org> (дата звернення: 30.05.2026).
11. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: <https://www.postgresql.org> (дата звернення: 01.05.2026).
12. RabbitMQ: One broker to queue them all. URL: <https://www.rabbitmq.com/> (дата звернення: 01.05.2026).
13. Docker Documentation. Офіційний сайт Docker. URL: <https://docs.docker.com> (дата звернення: 05.05.2026).

14. Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, 2017. 616 p.
15. Kurose J. F., Ross K. W. Computer Networking: A Top-Down Approach, 8th Edition. Pearson, 2020. 832 p.
16. Montgomery D. C., Peck E. A., Vining G. G. Introduction to Linear Regression Analysis. Wiley, 2021. 672 p.
17. TypeORM: Data-Mapper ORM for TypeScript, ES7, ES6, ES5. Офіційний сайт TypeORM. URL: <https://typeorm.io> (дата звернення: 11.05.2026).
18. JSON Web Token (JWT) Introduction. Офіційний сайт JWT. URL: <https://jwt.io/introduction> (дата звернення: 11.05.2026).
19. Flanagan D. JavaScript: The Definitive Guide, 7th Edition. O'Reilly Media, 2020. 706 p.
20. Recharts: A composable charting library built on React components. Офіційний сайт Recharts. URL: <https://recharts.org> (дата звернення: 12.05.2026).

ДОДАТОК А

Реалізація обчислювального модуля прогнозування

Програмні засоби:

- мова програмування – TypeScript;
- середовище розробки – Visual Studio Code

```
import { Injectable, Logger, Inject } from '@nestjs/common';
import { ClientProxy } from '@nestjs/microservices';
import { firstValueFrom } from 'rxjs';
```

```
@Injectable()
```

```
export class ForecastService {
  private readonly logger = new Logger(ForecastService.name);
```

```
  constructor(
    @Inject('TRANSACTION_SERVICE')
    private readonly transactionClient: ClientProxy,
  ) {}
```

```
  async getForecast(userId: string) {
    try {
      this.logger.log(`Запит прогнозу для ID: ${userId}`);

      const currentMonth = new Date().toISOString().slice(0, 7);
      const allTransactions: any[] = await firstValueFrom(
        this.transactionClient.send('get_transactions_for_forecast', { userId })
      );
```

```
this.logger.log(`Отримано транзакцій: ${allTransactions?.length}`);
```

```
if (!allTransactions || allTransactions.length === 0) {
  return this.getEmptyResponse();
}
```

```
// Фільтруємо — тільки минулі місяці
const pastTransactions = allTransactions.filter(t => {
  const month = new Date(t.transaction_date)
    .toISOString().slice(0, 7);
  return month < currentMonth;
});
```

```
if (pastTransactions.length === 0) {
  return this.getEmptyResponse();
}
```

```
// Агрегуємо по місяцях
const monthMap: Record<string, { income: number; expense: number }> = {};
for (const t of pastTransactions) {
  const month = new Date(t.transaction_date).toISOString().slice(0, 7);
  if (!monthMap[month]) monthMap[month] = { income: 0, expense: 0 };
  if (t.type === 'income') monthMap[month].income += Number(t.amount);
  else monthMap[month].expense += Number(t.amount);
}
```

```
const rawData = Object.entries(monthMap)
  .sort(([a], [b]) => a.localeCompare(b))
  .map(([month, v]) => ({ month, ...v }));
```

```

// Топ-5 категорій витрат
const categoryMap: Record<string, number> = {};
for (const t of pastTransactions) {
  if (t.type === 'expense') {
    categoryMap[t.category] = (categoryMap[t.category] || 0) +
Number(t.amount);
  }
}

const monthsCount = rawData.length || 1;
const categoryForecast = Object.entries(categoryMap)
  .sort(([, a], [, b]) => b - a)
  .slice(0, 5)
  .map(([name, total]) => ({
    name,
    predicted: Math.round(total / monthsCount),
  }));

const incomeHistory = rawData.map(d => d.income);
const expenseHistory = rawData.map(d => d.expense);

// Видаляємо викиди перед прогнозуванням
const cleanedIncome = this.removeOutliers(incomeHistory);
const cleanedExpense = this.removeOutliers(expenseHistory);

const nextMonthIncome = this.calculateWeightedForecast(cleanedIncome);
const nextMonthExpense = this.calculateWeightedForecast(cleanedExpense);

return {

```

```

    nextMonthIncome,
    nextMonthExpense,
    categoryForecast,
    hasEnoughData: rawData.length >= 2,
    dataPoints: rawData.length,
    updatedAt: new Date(),
  };
} catch (error: any) {
  this.logger.error(`Помилка: ${error?.message ?? String(error)}`);
  return this.getEmptyResponse();
}
}

```

// Видалення викидів на основі медіани

// Якщо значення перевищує медіану більш ніж у 3 рази - замінюємо медіаною// Це захищає прогноз від разових великих покупок

```

private removeOutliers(data: number[]): number[] {
  if (data.length < 3) return data;

  const sorted = [...data].sort((a, b) => a - b);
  const median = sorted[Math.floor(sorted.length / 2)];

  // Якщо медіана нульова — нічого не фільтруємо щоб уникнути помилок
  if (median === 0) return data;

  return data.map(v => (v > median * 3 ? median : v));
}

```

// Розрахунок прогнозу на основі зваженої лінійної регресії

```

private calculateWeightedForecast(data: number[]): number {
  const n = data.length;

  if (n === 0) return 0;
  if (n === 1) return Math.round(data[0]);

  let sumX = 0, sumY = 0, sumXY = 0, sumX2 = 0, weightTotal = 0;

  for (let i = 0; i < n; i++) {
    // Вага зростає з часом — нові дані важливіші
    const weight = i + 1;

    sumX += weight * i;
    sumY += weight * data[i];
    sumXY += weight * i * data[i];
    sumX2 += weight * i * i;
    weightTotal += weight;
  }

  const xMean = sumX / weightTotal;
  const yMean = sumY / weightTotal;

  // Знаменник для обчислення нахилу
  const denominator = sumX2 - weightTotal * xMean * xMean;

  // Якщо знаменник майже 0 - дані плоскі, повертаємо останнє значення
  if (Math.abs(denominator) < 0.00001) return Math.round(data[n - 1]);

  const slope = (sumXY - weightTotal * xMean * yMean) / denominator;

```

```

const intercept = yMean - slope * xMean;

// Прогноз на наступний період (x = n)
const prediction = slope * n + intercept;
return prediction > 0 ? Math.round(prediction) : 0;
}

// Повертає порожній об'єкт відповіді коли даних недостатньо
private getEmptyResponse() {
  return {
    nextMonthIncome: 0,
    nextMonthExpense: 0,
    categoryForecast: [],
    hasEnoughData: false,
    dataPoints: 0,
    updatedAt: new Date(),
  };
}
}

```

ДОДАТОК Б

Апробація

UDC 004.942

¹ BS's programme 4th year Holub T.D.

¹ Assoc.prof., cand.econ.sc. Segeda I.V.

<https://scholar.google.com.ua/citations?hl=en&user=cTob-AwAAAAJ>

¹ Igor Sikorsky Kyiv Polytechnic Institute

FINANCIAL MANAGEMENT SYSTEM WITH FORECASTING CAPABILITIES

Statement of the problem and its relevance. In the context of the ongoing digitalization of financial processes, there is a growing need for information systems that enable effective personal finance management. Irregular tracking of income and expenses, the lack of structured analysis of financial dynamics, and the absence of planning tools complicate the process of making well-grounded financial decisions. Most existing software solutions are implemented as monolithic systems, which limits their scalability, flexibility, and ability to integrate new functional modules. Therefore, the adoption of a microservices architecture is particularly relevant, as it ensures modularity, component independence, and deployment flexibility [1, 2].

Analysis of the latest research. In the field of modern information system development, there is a steady shift from monolithic architectures to microservices-based approaches. Microservices architecture enables the decomposition of a system into independent functional components, significantly simplifying scaling, updating, and maintaining software systems [2, 3]. This approach is especially critical for financial systems, where reliability, security, and gradual functional extensibility are key requirements.

Alongside architectural evolution, technologies for building user interfaces are also advancing. The use of component-oriented libraries such as React ensures high interactivity and responsiveness of web applications. Containerization, particularly with Docker, is considered an effective mechanism for isolated service deployment and environment standardization [3].

For storing financial data, the use of relational database management systems with transactional support and data integrity mechanisms, such as PostgreSQL, is appropriate [4].

In the domain of financial analytics, considerable attention is devoted to time series analysis. For assessing the dynamics of personal financial indicators, the application of basic statistical methods—specifically linear regression—is justified. This method enables the identification of trends and the generation of short-term forecasts [5]. The simplicity of the mathematical model ensures interpretability of results and allows the forecasting module to be implemented as a lightweight, standalone service.

The goal of this work is to develop a web-oriented personal finance management system based on a microservices architecture that provides structured accounting of financial transactions, monitoring of financial goal achievement, and budget forecasting using statistical analysis methods.

Main part. The system architecture is implemented in accordance with microservices principles, ensuring modularity, component isolation, and independent scalability of individual services. The client-side application is built using the React library, which enables the development of a dynamic and interactive user interface, supports component-based design, and simplifies further frontend updates. The server-side logic is distributed among the following components:

- user-service – implemented using Node.js; manages user profiles and provides authentication and authorization via JSON Web Token (JWT), ensuring secure data access;
- transaction-service – responsible for processing financial transactions, interacting with the database through an ORM layer, and maintaining logging and auditing of operations;
- goal-service – implements financial goal-setting logic and progress tracking, allowing users to

- evaluate achievement of planned objectives;
- analytics-service – performs mathematical calculations of trends and forecasting on the server side, including time series modeling.

A key architectural component is the API Gateway, which serves as the single entry point for all incoming requests. It acts as a protective layer, hiding the internal microservices network topology from external access. The Gateway performs centralized request routing, load balancing, and initial token validation, thereby unifying system-wide security.

Each service is deployed in a separate Docker container, ensuring isolation and enabling independent scalability. Interaction between services and the frontend is carried out via asynchronous requests to a RESTful API. Security is ensured through JWT-based authentication. Data storage is implemented using PostgreSQL, where transactional consistency and data integrity constraints are enforced for all entities.

Analytical Module. The system includes an analytical module designed to evaluate users' financial dynamics and forecast future indicators.

Linear regression is applied to formalize the general trend of income or expense changes over time. After aggregating data by month, a model of the following form is constructed:

$$y = a \cdot x + b$$

x – ordinal number of the period, y – value of the financial indicator.

The coefficient a determines the nature of the trend (growth, decline, or stability), enabling estimation of the indicator for the subsequent period.

The module supports data aggregation by various periods (day, week, month) and implements filtering and expense categorization. This allows forecasting for specific groups of financial operations, such as utilities, groceries, or entertainment.

Analytical results are presented in the form of interactive charts and diagrams, enhancing the clarity of financial information and supporting effective planning.

Conclusions. The implementation of the system based on a microservices architecture transforms standard expense tracking into a flexible analytical tool. The use of linear regression enables the evaluation of financial dynamics and the generation of short-term forecasts even with limited user data. Future work will focus on incorporating seasonal budget fluctuations and implementing secure data synchronization mechanisms between services.

References:

1. Personal Finance Management in Modern Conditions: Scientific Analysis. Ministry of Education and Science of Ukraine, 2024. URL: <https://api.dspace.wunu.edu.ua/api/core/bitstreams/62988557-59c5-49cf-8177-39be45b23a2a/content> (date of access: 14.02.2026)
2. Gajula S. Architectural Transformation of Legacy Financial Systems: A Framework for Microservices, Cloud, and API Integration. *International Journal of Information Technology and Management Information Systems*. 2025. Vol. 16, Is. 2. P. 75–84.
3. Suleymani T. *Hands-On Microservices with JavaScript: Build scalable web applications with JavaScript, Node.js, and Docker*. Packt Publishing, 2024. 450 p.
4. PostgreSQL 15 Documentation. PostgreSQL Global Development Group, 2025. URL: <https://www.postgresql.org/docs/15/index.html> (date of access: 15.02.2026).
5. Montgomery D. C., Peck E. A., Vining G. G. *Introduction to Linear Regression Analysis*. 6th ed. Wiley, 2021. 720 p. URL: <https://content.e-bookshelf.de/media/reading/L-16125104-1a3a7c5bd1.pdf> (date of access: 17.02.2026).