

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ДЛЯ
ОБЛІКУ ОСОБИСТИХ ФІНАНСІВ
SOFTWARE IMPLEMENTATION OF A
PERSONAL FINANCE MANAGEMENT APPLICATION

Виконав(ла): студент(ка) 4 курсу, групи КНТ-132
Спеціальності 121 Інженерія програмного
(код і найменування спеціальності)
забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ДЕВ'ЯНІН О.С.

(ПРИЗВИЩЕ та ініціали)

Керівник ГОФМАН Є.О.

(ПРИЗВИЩЕ та ініціали)

Рецензент ДЬЯЧУК Т.С.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н., проф.
Сергій СУББОТІН
“ ” 2026 року

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ДЕВ'ЯНІНА Олексія Сергійовича
(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація застосунку для обліку особистих фінансів.

Software Implementation of a Personal Finance Management Application

керівник проєкту (роботи) к.т.н., доцент, ГОФМАН Євгеній Олександрович,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “07” квітня 2026 року №139

2. Строк подання студентом проєкту (роботи) 10 червня 2026 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Розробка програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Прийняв виконане завдання
1-4 Основна частина	ГОФМАН Є. О., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання “ 20 ” квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів курсового проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2–3 тижні	Розділ 1
3	Розробка архітектури програми.	4 тиждень	Розділ 2
4	Розробка програми.	5 тиждень	Розділ 3
5	Тестування та експериментальне дослідження програми.	6 тиждень	Розділ 4
6	Оформлення пояснювальної записки та документів до неї. Нормоконтроль та рецензування.	7 тиждень	Додатки
7	Захист роботи.	8 тиждень	

Студент(ка)

(підпис)

Олексій ДЕВ'ЯНІН
(Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи)

(підпис)

Євгеній ГОФМАН
(Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 115 с., 10 табл., 36 рис., 3 дод., 12 джерел.

ОБЛІК ОСОБИСТИХ ФІНАНСІВ, ЗАСТОСУНОК, ASP.NET CORE, WPF, MVVM, ENTITY FRAMEWORK CORE, POSTGRESQL, JWT, BCrypt, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ІМПОРТ CSV, КАТЕГОРИЗАЦІЯ ТРАНЗАКЦІЙ.

Об'єкт дослідження – процес інженерії програмного забезпечення для системи обліку особистих фінансів.

Предмет дослідження – методи та програмні засоби побудови прикладного клієнт-серверного застосунку для ведення особистого бюджету, імпорту транзакцій, їх категоризації, зберігання та аналізу.

Мета роботи – підвищення надійності функціонування системи обліку особистих фінансів та розширення її функціональних можливостей шляхом проєктування модулів автоконвертації валют за датою, впровадження інтелектуальної автокатегоризації транзакцій та реалізації адаптивного імпорту банківських даних.

Матеріали, методи та технічні засоби: у процесі роботи були застосовані методи об'єктно-орієнтованого програмування, принципи багатоварової клієнт-серверної архітектури, а також методології стандартизованого тестування програмного забезпечення. Як основний інструментарій використано мову програмування C#, крос платформні фреймворки ASP.NET Core Web API 10.0 та Windows Presentation Foundation, об'єктно-реляційний перетворювач Entity Framework Core. Для збереження даних застосовано реляційну систему керування базами даних PostgreSQL. Безпека системи забезпечується механізмом автентифікації на основі JSON Web Token та алгоритмом криптографічного хешування паролів BCrypt.Net.

Результати роботи підтверджують успішне створення клієнт-серверного застосунку для обліку особистих фінансів, який забезпечує повністю автоматизований контроль доходів і витрат. Реалізовано серверну частину у вигляді RESTful API з базою даних PostgreSQL, що гарантує транзакційну цілісність, та десктопний клієнт з використанням технології WPF і архітектурного патерну Model-View-ViewModel. Впроваджено функціонал захищеної авторизації, автоматизованого імпорту банківських виписок у форматі CSV за допомогою поведінкового патерну "Стратегія". Розроблено алгоритми автоматичної категоризації транзакцій на основі користувацьких правил та автоматичного відстеження дублікатів для запобігання надмірності даних. Створено підсистему планування бюджетів із генерацією аналітичних звітів.

Висновки засвідчують досягнення мети роботи: створено повнофункціональний застосунок, який ефективно розв'язує проблему фрагментації фінансових даних. Програма дозволяє користувачеві гнучко налаштовувати ієрархію категорій, вести одночасний облік за кількома рахунками у різних валютах та здійснювати детальний аналіз виконання бюджету на основі візуальних графіків. Розроблені алгоритми автокатегоризації та захисту від повторного імпорту транзакцій довели свою здатність значно мінімізувати рутинну роботу, усуваючи необхідність багаторазового ручного введення однієї й тієї ж інформації.

Галузь використання результатів охоплює сферу управління фінансами для індивідуальних користувачів, а також планування спільного бюджету домогосподарств. Програма може бути використана як локальний фінансовий аналізатор для само зайнятих осіб.

Економічна ефективність розробки полягає у радикальному скороченні витрат часу користувача (до 70%) на ручне введення даних та їхню класифікацію завдяки автоматичному імпорту виписок та автокатегоризації. Надані аналітичні інструменти сприяють раціональному розподілу фінансових ресурсів та запобігають перевитратам.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 115 pages, 10 tables, 36 figures, 3 appendixes, 12 sources.

PERSONAL FINANCE MANAGEMENT, APPLICATION, ASP.NET CORE, WPF, MVVM, ENTITY FRAMEWORK CORE, POSTGRESQL, JWT, BCrypt, CLIENT-SERVER ARCHITECTURE, CSV IMPORT, TRANSACTION CATEGORIZATION.

The object of the study is the software engineering process for a personal finance management system.

The subject of the study is the methods and software tools for developing a client-server application for managing personal budgets, importing transactions, categorizing them, and storing and analyzing them.

The objective of this work is improve the operational reliability of the personal finance management system and significantly expand its functional capabilities by designing modules for currency auto-conversion by date, implementing intelligent transaction auto-categorization, and developing an adaptive banking data import engine.

Materials, methods, and technical tools utilized in the work include object-oriented programming methods, principles of multi-tier client-server architecture, structural and comparative analysis, as well as standardized software testing methodologies. The primary toolset involves the C# programming language, the cross-platform frameworks ASP.NET Core Web API 10.0 and Windows Presentation Foundation, and the Entity Framework Core object-relational mapper. A PostgreSQL relational database management system was used for data persistence. System security is ensured by an authentication mechanism based on JSON Web Tokens and the BCrypt.Net cryptographic password hashing algorithm.

The results of the work confirm the successful creation of a client-server personal finance management application that provides fully automated tracking of income and expenses. The server-side component was implemented as a RESTful API with a PostgreSQL database, ensuring transactional integrity, while the desktop client was developed using WPF technology and the Model-View-ViewModel architectural pattern. Secure authorization functionality, automated import of bank statements in CSV format using the Strategy behavioral pattern were implemented. Algorithms for automatic transaction categorization based on user rules and automatic duplicate tracking to prevent data redundancy were developed. A budget planning subsystem with analytical report generation was created.

Conclusions attest to the achievement of the work's objective: a fully functional application has been created that effectively solves the problem of financial data fragmentation. The program allows the user to flexibly configure the category hierarchy, maintain simultaneous accounting across multiple accounts in different currencies, and conduct a detailed analysis of budget execution based on visual charts. The developed algorithms for auto-categorization and protection against re-importing transactions have proven their ability to significantly minimize routine work, eliminating the need for repeated manual entry of the same information.

The field of use for the results encompasses personal finance management for individual users, as well as joint household budget planning. The program can also be used as a local financial analyzer for self-employed individuals.

The economic efficiency of the development lies in a radical reduction in user time expenditure (up to 70%) on manual data entry and classification, thanks to the automatic import of statements and auto-categorization. The provided analytical tools promote the rational allocation of financial resources and prevent overspending.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	11
Вступ	12
1 Аналіз предметної області	16
1.1 Короткий опис досліджуваної предметної області.....	16
1.2 Огляд існуючих рішень проблеми.....	17
1.2.1 Застосунок gnucash.....	18
1.2.2 Застосунок homebank.....	19
1.2.3 Застосунок actualbudget.....	20
1.2.4 Порівняльна таблиця існуючих та розроблюваного програмних продуктів	21
1.3 Опис основних вимог до програми	22
1.3.1 Визначення функціональних вимог	22
1.3.2 Нефункціональні вимоги	23
1.4 Постановка завдання роботи.....	24
1.5 Висновки за розділом 1	24
2 Розробка архітектури програми	26
2.1 Структура системи	26
2.1.1 Основні компоненти системи.....	26
2.2 Функціонування системи	28
2.3 Вибір мови програмування та інструментарію для розробки програми.....	28
2.3.1 Мова програмування.....	28
2.3.2 Середовище розробки.....	31
2.3.3 База даних	33
2.4 Вимоги до технічних і програмних засобів	36
2.5 Висновки за розділом 2	37
3 Розробка програми	38
3.1 Архітектура системи	38

3.1.1 Структурна схема системи.....	38
3.2 Функціонування системи	40
3.2.1 Функціональна схема.....	40
3.2.2 Опис процесу обміну даними у системі.....	40
3.3 Опис модулів системи	41
3.3.1 Модулі серверної частини.....	41
3.3.2 Модулі клієнтської частини.....	42
3.4 Проектування бази даних.....	42
3.4.1 Структура бази даних	43
3.4.2 Взаємозв'язки між таблицями	43
3.5 Висновки за розділом 3	46
4 Експлуатація, тестування та експериментальне дослідження програми	47
4.1 Опис застосування програми	47
4.1.1 Огляд планової цільової аудиторії.....	47
4.1.2 Візуалізація планованої цільової аудиторії.....	49
4.2 Інструкція по роботі з програмою	50
4.2.1 Встановлення програми	50
4.2.2 Запуск програми.....	52
4.2.3 Методика роботи з програмою	55
4.3 Приклад роботи користувача з програмою	57
4.4 Тестування роботи програми	61
4.4.1 Функціональне тестування.....	61
4.4.2 Аналіз функціонального тестування	62
4.4.3 Тестування продуктивності	63
4.4.4 Аналіз продуктивності	65
4.4.5 Тестування безпеки	66
4.4.6 Аналіз безпеки	67
4.4.7 Аналіз користувацького досвіду	68
4.5 Рекомендації щодо використання.....	69
4.6 Плани з розвитку	71

	10
Висновки.....	72
Перелік джерел посилання	74
Додаток А Технічне завдання.....	76
Додаток Б Текст програми.....	82
Додаток В Слайди презентації	108

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
CSV	– Comma-Separated Values;
DTO	– Data Transfer Object;
EF Core	– Entity Framework Core;
ERD	– Entity-Relationship Diagram;
HTTP	– HyperText Transfer Protocol;
IDE	– Integrated Development Environment;
JWT	– JSON Web Token;
MVVM	– Model-View-ViewModel;
ORM	– Object-Relational Mapping.

ВСТУП

Сучасний етап розвитку цифрової економіки та стрімка інформатизація суспільства нерозривно пов'язані з необхідністю автоматизації рутинних повсякденних процесів, серед яких фундаментальне місце займає управління особистими фінансами [1]. Збільшення кількості доступних фінансових інструментів, часте використання кількох дебетових чи кредитних карток від різних банківських установ, а також глобальний перехід від готівкових розрахунків до цифрових платежів призвели до того, що фінансові сліди користувача виявляються критично розпорошеними. За таких умов традиційний ручний облік коштів, який раніше здійснювався за допомогою паперових носіїв або базових електронних таблиць, стає вкрай неефективним, трудомістким процесом, що супроводжується високим ризиком механічних помилок та втрати даних [2].

Проблема низького рівня автоматизації фінансового контролю та відсутності фінансової дисципліни залишається гостро актуальною в глобальному масштабі. Відповідно до аналітичних звітів Організації економічного співробітництва та розвитку (ОЕСР) щодо фінансової грамотності, понад 60% дорослого населення стикаються з перманентними фінансовими труднощами саме через нездатність систематично вести консолідований облік своїх рахунків, відстежувати дрібні витрати та планувати довгостроковий бюджет [3]. Більше того, соціологічні дослідження, зокрема проведені серед молоді, вказують на парадоксальну ситуацію: незважаючи на загальну цифрову грамотність поколінь Millennials та Gen Z, лише трохи більше половини з них використовують спеціалізовані додатки для управління фінансами. Серед основних причин відмови від використання таких програм респонденти називають надмірну складність інтерфейсів, необхідність постійного ручного введення кожної транзакції, а також серйозні побоювання щодо безпеки власних фінансових даних. Водночас прогнозується, що глобальний ринок додатків для управління

особистими фінансами досягне оціночної вартості у 1,5 мільярда доларів США вже до кінця поточного року із середньорічним темпом зростання на рівні 8,7%, що свідчить про колосальний попит на якісні цифрові інструменти [4].

Отже, актуальність розробки полягає в необхідності створення незалежного, надійного та функціонального програмного інструменту, який гармонійно поєднував би безпечне централізоване зберігання даних із інтелектуальною системою автоматичного імпорту, категоризації та бюджетування. Провідні наукові дослідження в галузі архітектури програмного забезпечення підтверджують високу ефективність використання платформи.NET та архітектурного патерну Model-View-ViewModel для створення масштабованих десктопних застосунків. Цей патерн забезпечує глибоке розділення бізнес-логіки та графічного інтерфейсу, що критично важливо для додатків із складними інтерфейсами користувача [5]. Водночас, для серверної частини оптимальним вважається застосування RESTful архітектури з використанням патернів Repository та безпечних механізмів авторизації [6].

Об'єктом дослідження є безпосередньо процес інженерії програмного забезпечення для управління особистими фінансами, планування сімейного або індивідуального бюджету та контролю витрат користувача в умовах сучасної цифрової економіки.

Предметом дослідження є інженерні методи, програмні засоби та архітектурні підходи до розробки прикладного клієнт-серверного застосунку. Зокрема, досліджуються технології розробки на базі WPF та ASP.NET Core, методи побудови реляційних баз даних, алгоритми машинного аналізу текстових даних для парсингу виписок, а також криптографічні механізми забезпечення конфіденційності.

Мета роботи полягає у розробленні надійного клієнт-серверного застосунку та розширення функціональних можливостей шляхом автоматизації управління особистими фінансами, що забезпечує мінімізацію

ручного введення даних за рахунок масового імпорту банківських виписок, гарантує безпечне збереження інформації та надає користувачеві ефективні візуальні інструменти для глибокого аналізу витрат.

Для досягнення поставленої мети у роботі були сформульовані та вирішені такі науково-практичні завдання:

- проаналізувати стан предметної області, дослідити сучасні вимоги до систем управління фінансами, виконати критичний огляд існуючих програмних аналогів та обґрунтувати вибір технологічного стеку для реалізації проєкту;
- розробити загальну архітектуру клієнт-серверної системи, спроектувати структуру реляційної бази даних та визначити механізми взаємодії між модулями системи на основі сучасних патернів проєктування;
- програмно реалізувати серверну частину із впровадженням надійних механізмів захисту;
- програмно реалізувати клієнтську десктопну частину з використанням технології WPF та патерну MVVM для забезпечення управління рахунками, транзакціями та бюджетами;
- розробити та інтегрувати спеціалізований модуль масового імпорту даних, здатний розпізнавати формати різних банків, автоматично класифікувати транзакції та блокувати спроби дублювання записів;
- провести експериментальне дослідження та тестування розробленого програмного забезпечення для перевірки його функціональності, продуктивності та рівня безпеки.

Методика дослідження базується на комплексному застосуванні системного аналізу для формування вимог до програмного продукту. При проєктуванні системи використовувалися методи об'єктно-орієнтованого програмування та поширені архітектурні патерни, такі як MVC (Model-View-Controller) для серверної частини, MVVM для клієнтської частини, а також патерни Strategy та Repository. Безпека даних забезпечувалася через впровадження криптографічних алгоритмів та стандартів автентифікації без

збереження стану. Тестування проводилося з урахуванням міжнародних стандартів інженерії програмного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Короткий опис досліджуваної предметної області

Управління особистими фінансами є складним багатокроковим процесом, який охоплює планування, облік, контроль та системний аналіз власних доходів і витрат індивіда або домогосподарства [7]. В епоху глобальної діджиталізації цей процес зазнав докорінних змін. З одного боку, перехід від готівкових розрахунків до використання дебетових і кредитних карток, електронних гаманців та систем мобільних платежів значно спростив саму процедуру оплати. З іншого боку, така зручність призвела до розпорошеності фінансової історії. Користувач, який щоденно здійснює десятки дрібних транзакцій через різні фінансові установи, втрачає цілісне уявлення про свій реальний фінансовий стан. Відстеження цих операцій вручну вимагає надмірних когнітивних зусиль та дисципліни, що часто стає причиною повної відмови від ведення бюджету.

Сучасні дослідження підтверджують критичну необхідність впровадження автоматизованих інструментів. Масштабне соціологічне опитування, проведене серед молоді (віком від 18 до 35 років) у В'єтнамі, яке охопило 255 респондентів, продемонструвало глибинні тенденції у цій сфері. За даними дослідження, 56,9% опитаних вже використовують додатки для фінансового обліку, що свідчить про високий рівень проникнення технологій. Основні цілі використання цих інструментів схилиються до короткострокових потреб: фіксація щоденних витрат, контроль бюджету та зменшення загального фінансового тиску. Під час вибору застосунку користувачі висувають жорсткі вимоги. Найважливішим критерієм є різноманітність функцій (61,9%), що передбачає можливості автоматичного відстеження та імпорту даних. Другим за значущістю фактором є персоналізація аналітики (57,1%), а третім – зручність та інтуїтивність інтерфейсу (52,4%). Водночас, серед тієї частини молоді, яка відмовляється від використання таких систем, 31% все ще віддають перевагу традиційним

інструментам, таким як паперові зошити або базові таблиці Excel, а 17,2% висловлюють серйозні побоювання щодо безпеки та конфіденційності своїх персональних даних [8].

Ці дані чітко окреслюють проблематику предметної області. Розроблювана система повинна розв'язувати фундаментальний конфлікт між необхідністю детального обліку та небажанням користувача витратити на це свій час. Відповідно, об'єктом дослідження є сам процес управління фінансами, який потребує оптимізації. Предметом дослідження виступає сукупність методів та технологій, необхідних для створення такої оптимізуючої системи. Сюди входять архітектурні моделі побудови надійних клієнт-серверних додатків, алгоритми парсингу текстових масивів (виписок) для мінімізації ручної праці, методи інтелектуального аналізу рядків для розпізнавання категорій витрат, а також протоколи криптографічного захисту, які покликані нівелювати страхи користувачів щодо витоку конфіденційної інформації.

Окремим аспектом предметної області є архітектура клієнтських програм. Оскільки фінансовий аналіз передбачає інтенсивну взаємодію користувача з графіками, списками транзакцій та формами введення, графічний інтерфейс має бути максимально реактивним. Це вимагає використання сучасних архітектурних патернів, які дозволяють розмежувати логіку відображення від логіки обробки даних, забезпечуючи високу швидкодію та простоту тестування коду.

1.2 Огляд існуючих рішень проблеми

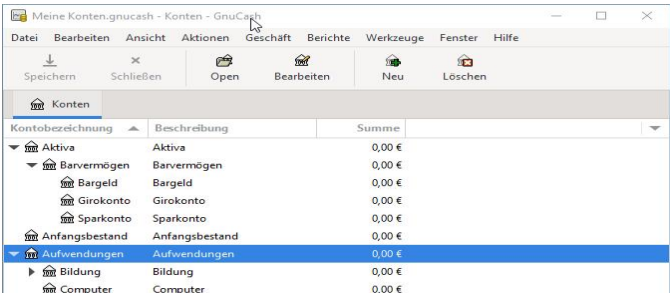
Ринок програмного забезпечення для управління персональними фінансами є надзвичайно насиченим, проте ідеального рішення, яке б задовольняло всі вимоги щодо приватності, автоматизації та кросплатформності, досі не існує. Для обґрунтування напрямку розробки було проведено глибокий аналіз існуючих аналогів. Щоб забезпечити

об'єктивність порівняння з розроблюваним десктопним WPF-застосунком, для огляду були обрані популярні настільні та гібридні системи з відкритим вихідним кодом або потужним локальним функціоналом: GnuCash, HomeBank та Actualbudget.

1.2.1 Застосунок GnuCash

GnuCash – це одне з найстаріших та найпотужніших рішень з відкритим вихідним кодом, яке доступне на операційних системах Linux, Windows та macOS. Фундаментальною відмінністю GnuCash є використання принципів подвійного запису, що робить його придатним не лише для особистих фінансів, але й для ведення бухгалтерії малого бізнесу. Програма пропонує виняткову деталізацію звітів, підтримку мультивалютності, інвестиційних портфелів та навіть генерування рахунків-фактур. Однак, саме ця професійна орієнтація є найбільшим недоліком для пересічного користувача. Інтерфейс GnuCash перевантажений специфічною термінологією, що створює надзвичайно високий поріг входження. Крім того, процеси автоматичного завантаження та зіставлення транзакцій з банків часто працюють нестабільно, вимагаючи від користувача постійних ручних маніпуляцій із файлами OFX/QFX та складного навігування вебсторінками. Таким чином, незважаючи на потужність, GnuCash не вирішує проблему економії часу користувача.

Вигляд домашньої сторінки застосунку GnuCash продемонстровано на рисунку 1.1.



Kontobezeichnung	Beschreibung	Summe
▼ Aktiva	Aktiva	0,00 €
▼ Barvermögen	Barvermögen	0,00 €
Bargeld	Bargeld	0,00 €
Girokonto	Girokonto	0,00 €
Sparkonto	Sparkonto	0,00 €
Anfangsbestand	Anfangsbestand	0,00 €
▼ Aufwendungen	Aufwendungen	0,00 €
Bildung	Bildung	0,00 €
Computer	Computer	0,00 €

Рисунок 1.1 – Головний екран Mobills [9]

1.2.2 Застосунок HomeBank

HomeBank – додаток, що також має тривалу історію, що сягає 1995 року, і позиціонується як значно простіша та дружніша до користувача альтернатива GnuCash. HomeBank є повністю безкоштовним і крос платформним інструментом. Організація даних у ньому здійснюється за допомогою інтуїтивної системи категорій та тегів, що дозволяє легко структурувати фінансові потоки. Програма містить вбудовані інструменти для генерації секторних діаграм та графіків, підтримує планування повторюваних транзакцій та нагадування. Важливою перевагою є можливість імпорту даних із форматів MSMoney, Quicken та CSV-файлів. Втім, HomeBank має суттєве архітектурне обмеження: він не підтримує клієнт-серверну модель зберігання даних. Вся інформація зберігається виключно в локальному файлі, що унеможливлює безпечну хмарну синхронізацію між різними пристроями без використання сторонніх сервісів. Хоча відсутність підключення до банків у реальному часі може розглядатися як перевага з точки зору абсолютної приватності, це змушує користувача покладатися виключно на ручний імпорт. Крім того, алгоритми виявлення дублікатів у HomeBank не завжди коректно обробляють специфічні формати виписок східноєвропейських банків.

Вигляд домашній сторінки застосунку HomeBank продемонстровано на рисунку 1.2.

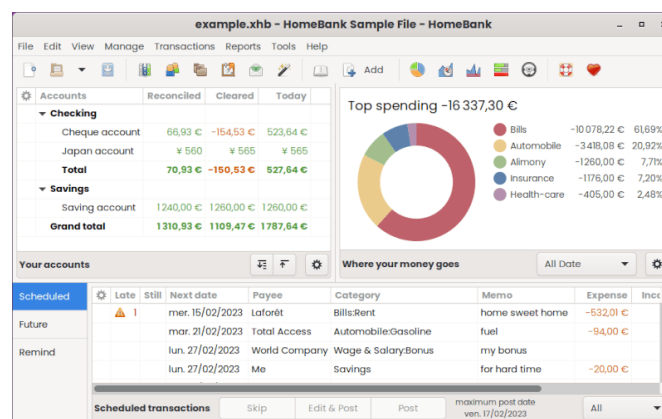


Рисунок 1.2 – Головний екран HomeBank [10]

1.2.3 Застосунок ActualBudget

ActualBudget – інструмент, що базується на популярній методології "конвертного бюджетування", яка інтуїтивно зрозуміла користувачам, що прагнуть жорстко контролювати свої витрати. Actualbudget поширюється як у вигляді локальних десктопних додатків (AppImage, Flatpak, Windows, Mac), так і пропонує можливість самостійного хостингу серверної частини. Завдяки наявності власного сервера користувачі можуть отримати доступ до свого бюджету з будь-якого пристрою. Програма підтримує легкий імпорт CSV-файлів, а ентузіасти можуть налаштувати сервер на автоматичний імпорт банківських виписок через стандартні банківські API. Незважаючи на сучасний підхід, налаштування власного сервера вимагає високих технічних навичок, а відсутність готового "хмарного" рішення робить цей продукт нішевим.

Вигляд домашній сторінки застосунку Actualbudget продемонстровано на рисунку 1.3.



Рисунок 1.3 – Головний екран Actualbudget [11]

1.2.4 Порівняльна таблиця існуючих та розроблюваного програмних продуктів

Для наочного узагальнення проведеного аналізу результати порівняння цих рішень із вимогами до розроблюваного програмного продукту зведено у порівняльну таблицю. У таблиці 1.1 відображено ключові архітектурні та функціональні відмінності існуючих систем та запропонованої розробки.

Таблиця 1.1 – Порівняння функціональності застосунків для управління фінансами

Критерій	GnuCash	HomeBank	Actualbudget	Розроблюваний застосунок
1	2	3	4	5
Архітектура	Локальна десктопна	Локальна десктопна	Локальна або Self-hosted	Клієнт-серверна (REST API + WPF)
Метод введення даних	Ручний / OFX імпорт (ускладнений)	Ручний / CSV імпорт	Ручний / CSV / API (для експертів)	Ручний / CSV імпорт (адаптивний парсинг)
Автокатегоризація транзакцій	Обмежена (потребує навчання)	Часткова (через фільтри)	Присутня	Потужна (користувачські правила та fallback)
Конвертація валют	Ручне введення курсу вручну для кожної операції	Відсутня (лише одна базова валюта)	Часткова (через сторонні плагіни)	Автоматична (динамічний щотижневий курс)
Захист від дублювання транзакцій	Базовий	Базовий	Базовий	Багатофакторний (рахунок + сума + опис + дата)

Продовження таблиці 1.1

1	2	3	4	5
Бюджетування та аналітика	Детальна бухгалтерська звітність	Базові діаграми	Візуальне конвертне планування	Текстовий і графічний аналіз, формування рекомендацій
Безпека авторизації	Відсутня (локальний файл)	Відсутня (локальний файл)	Залежить від налаштувань сервера	Посилена (хешування BCrypt + JWT-токени)

1.3 Опис основних вимог до програми

1.3.1 Визначення функціональних вимог

Функціональні вимоги визначають, що саме повинна робити система. Для розроблюваного застосунку обліку особистих фінансів визначено такі основні функціональні вимоги:

- підтримка повноцінного циклу реєстрації та автентифікації користувачів за електронною поштою та паролем;
- можливість управління фінансовими рахунками (створення, редагування та видалення рахунків різних типів);
- забезпечення повного спектру CRUD-операцій для модуля обробки транзакцій;
- наявність потужної системи фільтрації фінансових записів;
- організація транзакцій за допомогою індивідуально налаштованої багаторівневої системи категорій;
- імплементація механізму автокатегоризації транзакцій на основі ключових слів;
- автоматичне віднесення транзакцій до резервних категорій (Fallback) у разі відсутності відповідних правил;
- забезпечення масового імпорту даних із зовнішніх CSV-файлів;

- наявність алгоритму захисту від дублювання записів при імпорті за комбінацією чотирьох параметрів;
- функціонування підсистеми бюджетування для встановлення лімітів витрат за категоріями на часові інтервали.

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги визначають характеристики, обмеження та критерії якості роботи системи:

- використання клієнт-серверної моделі як основи архітектури системи;
- побудова серверної частини на базі фреймворку ASP.NET Core версії 10.0;
- застосування патерну Repository та Entity Framework Core для організації шару доступу до даних;
- використання реляційної системи управління базами даних PostgreSQL для постійного зберігання даних;
- реалізація безстанової авторизації на базі JSON Web Tokens (Bearer Authentication);
- застосування алгоритму криптографічного хешування BCrypt для захисту паролів користувачів;
- використання бібліотеки AutoMapper для перетворення внутрішніх сутностей у DTO з метою запобігання витоку даних;
- розробка клієнтської частини у вигляді десктопного застосунку на базі Windows Presentation Foundation (WPF);
- дотримання архітектурного патерну MVVM для клієнтської частини;
- забезпечення кросплатформності серверної частини та високої пропускної здатності системи.

1.4 Постановка завдання роботи

Розроблення застосунку «Програмна реалізація застосунку для обліку особистих фінансів» спрямоване на вирішення проблеми великих витрат часу на ручне ведення фінансового обліку та відсутності зручних інструментів для аналізу бюджету.

Враховуючи результати аналізу предметної області, розгляду аналогів та сформовані вимоги, завданням даної кваліфікаційної роботи є повний цикл проєктування, розробки та тестування інтегрованого клієнт-серверного рішення для автоматизованого обліку особистих фінансів. Завдання охоплює розробку надійної RESTful API серверної частини, що відповідатиме за централізоване управління даними та обчислювально складні операції (перевірка дублікатів, застосування правил категоризації). Водночас вимагається створення ергономічного WPF-клієнта, який слугуватиме точкою взаємодії користувача з системою, забезпечуючи локальний парсинг банківських виписок, агрегацію даних та зручну візуалізацію фінансового стану через графічні компоненти.

Успішне виконання завдання має на меті довести можливість мінімізації рутинних зусиль користувача за рахунок впровадження спеціалізованих алгоритмів обробки текстової інформації та багатошарової безпекової архітектури.

1.5 Висновки за розділом 1

Проведений у першому розділі комплексний аналіз предметної області засвідчив, що управління особистими фінансами в умовах сучасного цифрового суспільства вимагає переходу від ручного обліку до використання автоматизованих програмних систем. Детальний огляд і порівняння існуючих програмних продуктів (GnuCash, HomeBank, Actualbudget) показав, що попри їхні безперечні переваги, більшість із них страждає на перевантаженість інтерфейсів, відсутність гнучкої клієнт-серверної синхронізації або

недосконалість алгоритмів масового імпорту даних із нестандартних банківських виписок.

Спираючись на ці висновки, було сформульовано чіткий перелік функціональних та нефункціональних вимог до розроблюваного застосунку. Ключовими серед них є впровадження інтелектуальних механізмів масового імпорту із захистом від дублювання, гнучких правил автокатегоризації, а також застосування передових технологій безпеки (JWT, BCrypt) та архітектурних патернів (MVVM, Repository) на платформі .NET.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Структура системи

2.1.1 Основні компоненти системи

Задля забезпечення високого рівня масштабованості, надійності та зручності подальшого супроводу програмного коду, загальна архітектура проєкту була побудована за принципом багатошарової клієнт-серверної моделі. Цей підхід гарантує чіткий розподіл обов'язків між різними логічними рівнями системи, що унеможливорює змішування коду обробки даних, бізнес-логіки та графічного інтерфейсу.

2.1.1.1 Серверна частина

Серверна частина системи реалізує класичну трирівневу архітектуру, типову для додатків на базі ASP.NET Core :

- Presentation Layer (шар представлення): цей рівень представлений сукупністю контролерів (Controllers). Його єдиною функцією є обробка вхідних HTTP-запитів, здійснення базової вхідної валідації даних, маршрутизація запитів та формування стандартизованих відповідей клієнту у форматі JSON. Контролери не містять жодної бізнес-логіки;
- Service Layer (шар бізнес-логіки): усі складні алгоритмічні розрахунки, перевірки бізнес-правил та координація операцій знаходяться у сервісах (Services). Наприклад, саме цей рівень відповідає за перевірку масивів транзакцій на наявність дублікатів під час імпорту та застосування правил автокатегоризації;
- Data Access Layer (шар доступу до даних): взаємодія з базою даних інкапсульована за допомогою патерну Repository. Репозиторії абстрагують логіку виконання запитів через Entity Framework Core, що дозволяє легко замінити джерело даних у майбутньому або використовувати імітаційні об'єкти під час тестування.

2.1.1.2 Клієнтська частина

Клієнтська частина, яка розробляється як настільний застосунок, базується на архітектурному патерні MVVM. Історично цей патерн виник як еволюція класичних патернів MVC (Model-View-Controller) та MVP (Model-View-Presenter), будучи спеціально адаптованим інженерами Microsoft для використання у середовищах із потужними механізмами декларативного зв'язування даних (Data Binding), таких як Windows Presentation Foundation (WPF).

Фундаментальна перевага MVVM над класичним MVC у контексті розробки графічних інтерфейсів полягає у забезпеченні абсолютної слабкої зв'язності. Замість того, щоб контролер безпосередньо маніпулював візуальними елементами, у MVVM шар представлення (View), написаний мовою XAML, автоматично підписується на зміни властивостей об'єкта ViewModel. ViewModel, у свою чергу, містить виключно презентаційну логіку та абстрактні команди (Commands), не маючи жодного уявлення про те, як саме дані будуть відрендерені на екрані. Цей поділ є критичним для забезпечення високої тестованості.

Згідно з академічними дослідженнями, архітектура MVVM дозволяє значно знизити складність супроводу великих проєктів, що підтверджується метриками інструментів статичного аналізу коду, які фіксують зниження цикломатичної складності презентаційного шару та мінімізацію технічного боргу. Більше того, порівняльні емпіричні дослідження продуктивності та продуктивності розробки показують, що при створенні .NET-застосунків використання MVVM дозволяє досягти до 95,1% виконання вимог продуктивності в порівнянні з лише 41,6% для традиційного MVC у подібних умовах тестування GUI.

2.2 Функціонування системи

Взаємодія між клієнтським та серверним рівнями відбувається згідно з принципами REST (Representational State Transfer). Клієнт формує HTTP-запити до відповідних кінцевих точок (endpoints) сервера. Для уникнення витоку внутрішньої структури моделей бази даних клієнту, у проєкті широко використовується патерн Data Transfer Object (DTO). Усі дані, що надсилаються або отримуються через API, попередньо упаковуються у DTO. Автоматичне перетворення між внутрішніми моделями бази даних (наприклад, сутністю User або Transaction) та відповідними DTO здійснюється за допомогою бібліотеки AutoMapper.

Для забезпечення цілісності та узгодженості відповідей API, усі контролери використовують єдиний стандартизований патерн – об'єкт `ServiceResult<T>`. Ця універсальна обгортка містить не лише запитовані дані, але й метадані про статус виконання операції та чіткі повідомлення про помилки у разі їх виникнення. Такий підхід значно спрощує обробку виняткових ситуацій на стороні WPF-клієнта, дозволяючи централізовано виводити сповіщення користувачеві. Наскрізне використання механізму Dependency Injection (DI) в обох частинах системи дозволяє легко впроваджувати залежності (наприклад, сервіси логування або конфігурації) у будь-який компонент програми, що є еталоном сучасного проєктування.

2.3 Вибір мови програмування та інструментарію для розробки програми

2.3.1 Мова програмування

Для реалізації всього комплексу програмних рішень було обрано мову програмування C# та технологічну платформу .NET 10.0 SDK. Вибір єдиного технологічного стеку для бекенду та фронтенду забезпечує концептуальну цілісність проєкту, дозволяє перевикористовувати загальні бібліотеки та значно прискорює цикл розробки. Платформа .NET містить оптимізований

механізм JIT-компіляції та вдосконалену систему збирання сміття, що гарантує високу швидкість навіть при обробці великих масивів даних.

Критично важливою особливістю C# є вбудована парадигма асинхронного програмування з використанням ключових слів `async/await`. Усі операції вводу-виводу, такі як звернення до бази даних або мережеві запити, виконуються асинхронно. Це запобігає блокуванню головного потоку сервера, дозволяючи йому одночасно обробляти тисячі паралельних запитів, а на стороні клієнта – зберігає графічний інтерфейс чуйним під час тривалого парсингу файлів CSV.

2.3.1.1 Опис переваг та обмежень вибраної мови

Переваги вибраної мови та платформи:

- висока швидкість розробки завдяки потужному середовищу Visual Studio, яке має передові інструменти для налагодження, профілювання та візуального проєктування XAML-інтерфейсів для WPF;
- інтегрована мова запитів LINQ (Language Integrated Query), яка значно спрощує фільтрацію, сортування та агрегацію транзакцій, а також розрахунок бюджетів безпосередньо в коді;
- високий рівень абстракції при роботі з базами даних завдяки ORM Entity Framework Core, що дозволяє працювати з таблицями PostgreSQL як зі звичайними C#-об'єктами;
- велика спільнота розробників, вичерпна офіційна документація та наявність готових шаблонів проєктування.

Обмеження вибраної мови та платформи:

- залежність клієнтського інтерфейсу від операційної системи: технологія WPF прив'язана до графічної підсистеми Windows, що робить десктопний клієнт некросплатформним;
- споживання оперативної пам'яті: десктопні застосунки на базі .NET потребують завантаження середовища виконання (CLR), що може

вимагати більше ресурсів комп'ютера користувача порівняно з програмами на C++ чи Rust;

– поріг входження: розробка вимагає глибокого розуміння архітектурних патернів (MVC для сервера, MVVM для клієнта) та механізмів ін'єкції залежностей (Dependency Injection).

Порівняння мов програмування наведено в таблиці 2.1.

Таблиця 2.1 – Порівняння мов програмування

Критерій	C# (.NET)	Java	Python
Основні фреймворки (Клієнт / Сервер)	WPF / ASP.NET MVC	JavaFX / Spring Boot	PyQt / Django
Продуктивність	висока (JIT)	висока (JVM)	середня (інтерпретатор)
Типізація	строга, статична	строга, статична	динамічна
Підтримка асинхронності	вбудована (async/await)	через CompletableFuture / потоки	вбудована (asyncio)
Екосистема та пакети	дуже розвинена (NuGet)	розвинена (Maven, Gradle)	дуже розвинена (Pip)
Швидкість розробки бізнес-логіки	висока	середня	дуже висока
Підтримка	Microsoft	Oracle	Python Software Foundation

C# обрано через його ідеальну сумісність з інфраструктурою Windows для створення багатого графічного клієнта (WPF), потужні можливості для

побудови безпечного REST API сервера та високу швидкість обробки даних, що повністю задовольняє вимоги до створення застосунку для обліку особистих фінансів.

2.3.2 Середовище розробки

Для розробки клієнт-серверного застосунку обрано Visual Studio – комплексне інтегроване середовище розробки (IDE) від компанії Microsoft, яке є галузевим стандартом для створення програмного забезпечення на платформі .NET. Це рішення обґрунтоване такими причинами:

- повна екосистема .NET: середовище забезпечує нативну підтримку C#, WPF, ASP.NET MVC та Entity Framework, дозволяючи поєднати розробку клієнтської і серверної частин в одному рішенні;
- інструменти для роботи з XAML: вбудований візуальний конструктор та функція XAML Hot Reload дозволяють зручно розробляти та тестувати графічний інтерфейс WPF-клієнта в реальному часі без перезавантаження програми;
- робота з базами даних: інтегровані інструменти дозволяють зручно працювати з підключеннями до PostgreSQL, виконувати міграції Entity Framework та переглядати структуру бази даних безпосередньо з IDE;
- керування залежностями: вбудований диспетчер пакетів NuGet забезпечує швидке встановлення, оновлення та керування сторонніми бібліотеками (наприклад, Npgsql для бази даних, BCrypt.Net для хешування паролів);
- інтеграція з системами контролю версій: вбудована підтримка Git полегшує збереження історії змін, роботу з гілками та синхронізацію коду з віддаленими репозиторіями.

2.3.2.1 Обґрунтування вибору та опис його можливостей

Visual Studio обрано через його безперечну перевагу у розробці десктопних застосунків на базі WPF та серверних API на ASP.NET. Середовище має всі необхідні інструменти для швидкої реалізації та тестування функціоналу. Основні можливості:

- редактор коду: система IntelliSense на базі компілятора Roslyn забезпечує глибокий аналіз коду, розумне автодоповнення, виявлення синтаксичних помилок та миттєвий рефакторинг;
- налагодження: потужний відлагоджувач із підтримкою точок зупину, умовних брейкпоінтів, перегляду значень змінних у пам'яті та аналізу стеку викликів як для клієнта, так і для сервера;
- профілювання: вбудовані діагностичні інструменти (Diagnostic Tools) дозволяють аналізувати використання процесора, виявляти витрати оперативної пам'яті та моніторити продуктивність операцій імпорту CSV-виписок;
- тестування: нативна інтеграція з популярними тестовими фреймворками (xUnit, NUnit) для зручного запуску модульних тестів і перевірки бізнес-логіки обчислення бюджетів.

Порівняння популярних середовищ розробки для обраного технологічного стеку наведено в таблиці 2.2.

Таблиця 2.2 – Порівняння середовищ розробки

Критерій	Visual Studio	Visual Studio Code	JetBrains Rider
1	2	3	4
Підтримка платформи .NET	глибока, нативна	базова (через плагіни)	глибока

Продовження таблиці 2.2

1	2	3	4
Візуальний редактор WPF (XAML)	повноцінний	відсутній	базовий
Інструменти профілювання	потужні (вбудовані)	обмежені	потужні (вбудовані)
Робота з базами даних	вбудовані інструменти	через сторонні плагіни	вбудована (DataGrip)
Вимоги до ресурсів ПК	високі	низькі	високі
Критерій	Visual Studio	Visual Studio Code	JetBrains Rider

Середовище Visual Studio обрано через наявність найкращого на ринку візуального редактора для WPF, безшовну інтеграцію інструментів розробки та глибоку спеціалізацію саме на технологіях Microsoft, що забезпечило максимальну швидкість реалізації застосунку та надійну архітектуру.

2.3.3 База даних

Для забезпечення надійного зберігання фінансової інформації користувачів, їх рахунків та транзакцій у клієнт-серверному застосунку ключовим етапом є вибір системи управління базами даних (СУБД). Оскільки фінансовий облік вимагає високої точності та збереження зв'язків між сутностями (наприклад, транзакція завжди повинна належати конкретному рахунку та категорії), перевагу було надано реляційним системам.

2.3.3.1 PostgreSQL

PostgreSQL – це потужна об'єктно-реляційна система управління базами даних із відкритим вихідним кодом, яка є галузевим стандартом для побудови надійних серверних рішень. Взаємодія з базою у проєкті здійснюється через ORM-фреймворк Entity Framework Core, що дозволяє автоматизувати генерацію SQL-запитів.

Переваги:

- повна відповідність вимогам ACID (атомарність, узгодженість, ізолюваність, довговічність), що є критично важливим для гарантування правильності фінансових балансів;
- висока продуктивність при обробці великих масивів даних, що особливо важливо під час масового імпорту банківських виписок у форматі CSV;
- надійні механізми захисту даних та стійкість до збоїв завдяки системі попереднього запису в журнал;
- багатий інструментарій для масштабування та підтримка складних аналітичних запитів для модуля бюджетування.

Обмеження: необхідність налаштування та адміністрування окремого сервера бази даних, що вимагає більше зусиль на етапі розгортання порівняно з вбудованими локальними рішеннями.

2.3.3.2 Альтернативні рішення

Для обґрунтування вибору було розглянуто альтернативні бази даних: SQLite та MongoDB.

SQLite – це легка реляційна база даних, яка працює локально на пристрої у вигляді єдиного файлу.

Переваги: відсутність необхідності налаштування сервера та висока швидкість локального доступу.

Обмеження: не підходить для серверної частини багатокористувацьких застосунків, оскільки має обмеження на паралельний запис даних, що унеможлиблює її використання для нашого ASP.NET MVC сервера.

MongoDB – це документо-орієнтована NoSQL база даних.

Переваги:

- гнучкість структури даних та висока масштабованість.

Обмеження:

- відсутність жорстких зв'язків (зовнішніх ключів) між колекціями, що збільшує ризик порушення цілісності даних у фінансових застосунках;
- більша складність реалізації транзакційних операцій порівняно з класичними реляційними СУБД.

Порівняльний аналіз розглянутих баз даних наведено в таблиці 2.3.

Таблиця 2.3 – Порівняння баз даних

Критерій	PostgreSQL	SQLite	MongoDB
1	2	3	4
Тип бази даних	реляційна	реляційна (вбудована)	документо-орієнтована (NoSQL)
Архітектура застосування	клієнт-серверна	локальна (офлайн)	клієнт-серверна
Підтримка ACID (надійність)	повна	часткова (локальна)	підтримується, але складна в реалізації
Оптимальність для фінансів	ідеально підходить	не підходить для сервера	потребує додаткових механізмів контролю
Масштабованість	висока	низька	дуже висока

Продовження таблиці 2.3

1	2	3	4
Складність інтеграції	середня	дуже проста	середня
Багатокористувачий доступ	підтримується з коробки	обмежений	підтримується з коробки

Реляційна СУБД PostgreSQL обрана як оптимальне рішення для серверної частини розроблюваного застосунку, оскільки вона забезпечує найвищий рівень надійності, гарантує цілісність фінансових транзакцій користувачів та легко інтегрується з обраним фреймворком ASP.NET MVC через Entity Framework.

2.4 Вимоги до технічних і програмних засобів

Для розгортання та експлуатації клієнт-серверного комплексу висуваються конкретні вимоги до технічних та програмних засобів.

Для серверної частини необхідно використовувати серверне обладнання або хмарний контейнер під управлінням операційної системи Linux або Windows із встановленим середовищем виконання ASP.NET Core Runtime версії 10.0. СУБД PostgreSQL може бути розгорнута як на тому ж сервері, так і на виділеному кластері баз даних.

Клієнтська частина, оскільки вона розроблена з використанням технології WPF, вимагає наявності комп'ютера під управлінням операційної системи сімейства Windows (Windows 10 або Windows 11) із встановленим пакетом .NET Desktop Runtime. Технологія WPF глибоко інтегрована з підсистемою рендерингу DirectX в ОС Windows, що забезпечує апаратне прискорення вимальовування складної фінансової графіки, проте обмежує кросплатформність клієнта. Для коректної роботи застосунку необхідне

стабільне підключення до мережі Інтернет для обміну даними з API та завантаження актуальних курсів валют.

2.5 Висновки за розділом 2

Процес розробки архітектури програмного забезпечення продемонстрував необхідність ретельного планування структури системи. Використання багатошарової моделі для бекенду та патерну MVVM для фронтенду дозволило створити гнучку, тестовану та стійку до змін архітектуру. Вибір мови C# 14.0 та платформи.NET 10.0 SDK забезпечив високу продуктивність завдяки механізмам асинхронності. Особливу увагу було приділено питанню надійності зберігання фінансових даних: обґрунтовано відмову від нереляційних баз даних на користь PostgreSQL, що повністю підтримує транзакційні властивості ACID, гарантуючи неможливість пошкодження балансів користувачів при перебоях у роботі системи. Обраний інструментарій створює потужну основу для безпосереднього кодування логіки додатку.

3 РОЗРОБКА ПРОГРАМИ

3.1 Архітектура системи

Архітектура програмного забезпечення побудована за принципом багатoshарової клієнт-серверної моделі, що забезпечує чіткий розподіл обов'язків між модулями, спрощує масштабування та полегшує процес тестування. Клієнтська та серверна частини функціонують як незалежні компоненти, комунікуючи виключно через стандартизовані мережеві протоколи. Основною технологічною платформою обрано .NET 10.0.

Серверна частина реалізована як RESTful Web API з використанням фреймворку ASP.NET Core 10.0.

Клієнтська частина створена як настільний застосунок на базі технології WPF з дотриманням архітектурного патерну Model-View-ViewModel.

3.1.1 Структурна схема системи

Структурна схема системи деталізує ієрархію програмних модулів та ідентифікує зв'язки між ними, відповідаючи методології об'єктно-орієнтованого підходу.

Структуру системи можна поділити на два основні макроблоки – клієнт та сервер. На стороні сервера архітектура поділяється на три логічні шари:

- Presentation Layer (шар представлення): містить контролери (Controllers), які виступають точками входу (endpoints) для HTTP-запитів від клієнтів. Їхня роль зводиться до маршрутизації та базової вхідної валідації;
- Service Layer (шар бізнес-логіки): центральний рівень, де зосереджена вся алгоритмічна логіка. Він координує операції та перевіряє дотримання бізнес-правил;

– Data Access Layer (шар доступу до даних): абстрагує взаємодію з базою даних PostgreSQL. Реалізований за допомогою патерну Repository та інструментарію Entity Framework Core.

На стороні клієнта реалізовано патерн MVVM, який складається з таких шарів :

– View (шар представлення): XAML-розмітка вікон та користувацьких елементів керування (UserControls), яка відповідає за візуалізацію даних;

– ViewModel (шар моделей представлення): класи, які містять презентаційну логіку, обробляють команди від інтерфейсу та взаємодіють з локальними сервісами;

– Model (шар моделей): локальні DTO, що використовуються для зберігання отриманої від сервера інформації.

Взаємодія між компонентами здійснюється за допомогою механізму впровадження залежностей, що гарантує слабку зв'язність класів та спрощує їхню заміну імітаційними об'єктами під час тестування.

Схему взаємодії компонентів системи зображено на рисунку 3.1.

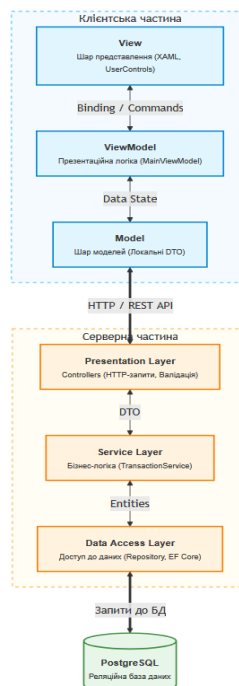


Рисунок 3.1 – Схеми взаємодії компонентів системи

3.2 Функціонування системи

3.2.1 Функціональна схема

Функціональна схема системи, що зображена на рисунку 3.2, відображає сукупність функціональних блоків, які забезпечують реалізацію завдань з управління персональними фінансами. Вона демонструє логічну послідовність обробки даних: від збору вхідної інформації до її візуалізації.

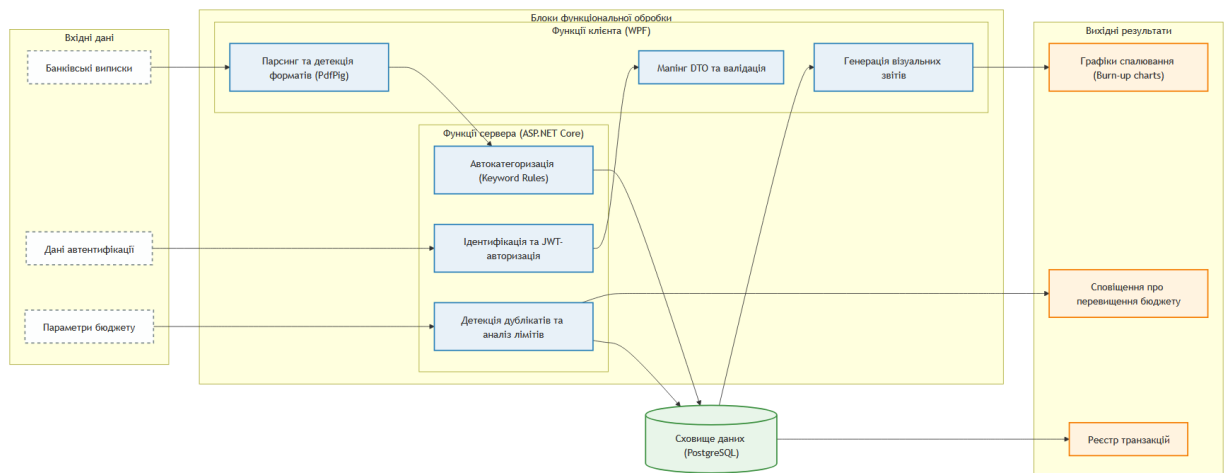


Рисунок 3.2 – Функціональна схема

3.2.2 Опис процесу обміну даними у системі

Обмін даними у системі відбувається через мережу Інтернет з використанням протоколу HTTP – RESTful архітектура. Для уникнення витоку внутрішньої структури моделей бази даних клієнту, у проєкті впроваджено строге використання об'єктів передачі даних. Безпосереднє перетворення між внутрішніми моделями бази даних та DTO здійснюється автоматизовано за допомогою бібліотеки AutoMapper на сервері.

Процес обміну даними стандартизовано за допомогою універсальної обгортки відповідей `ServiceResult<T>`. Будь-який запит від клієнта (наприклад, до `ApiService` за допомогою класу `HttpClient` та серіалізатора `System.Text.Json`) отримує від сервера не просто "сирі" дані, а об'єкт `ServiceResult`. Цей об'єкт містить безпосередньо запитувані дані, індикатор

успішності виконання операції та детальні повідомлення про помилки, якщо бізнес-логіка на сервері перервала виконання операції. Такий підхід гарантує цілісність обробки виняткових ситуацій на стороні WPF-клієнта.

Схема обміну даними у системі зображено на рисунку 3.3.

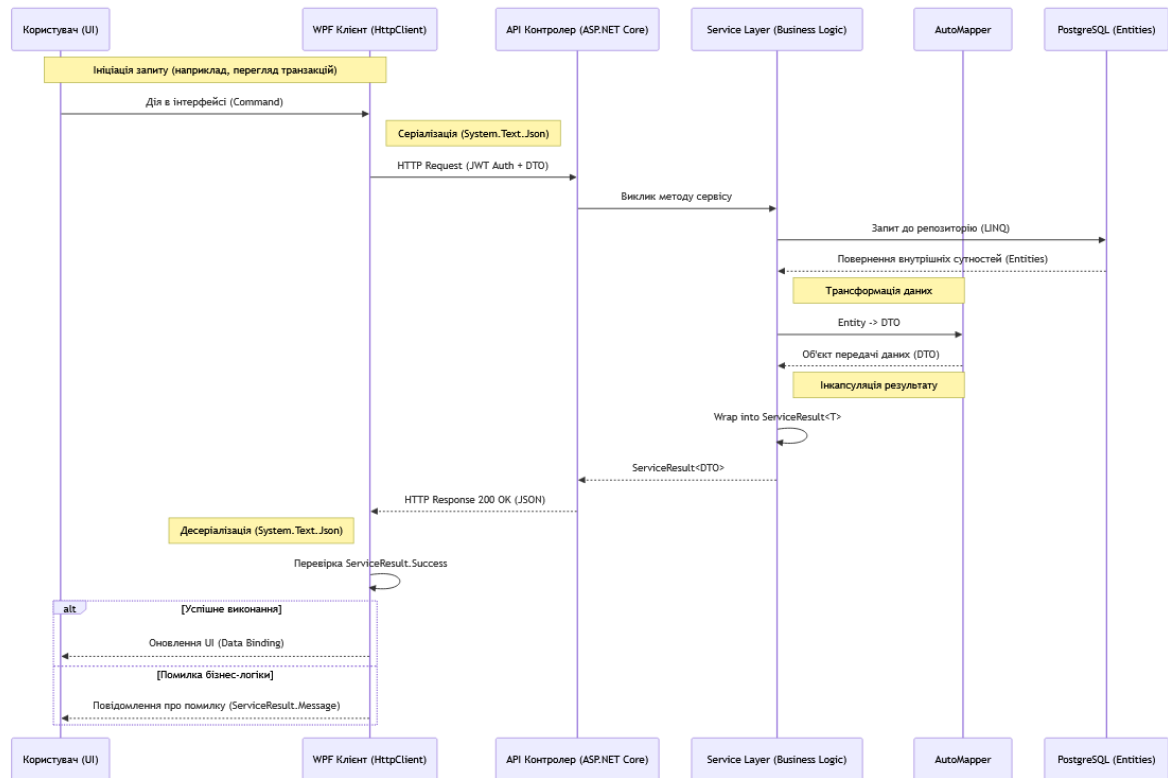


Рисунок 3.3 – Схема обміну даними у системі

3.3 Опис модулів системи

3.3.1 Модулі серверної частини

Модулями серверної частини є:

- контролери (Controllers): AuthController відповідає за реєстрацію та видачу токенів. TransactionController та AccountController реалізують CRUD-операції з фінансовими сутностями;
- бізнес-логіка (Services): ICategoriesService управляє ієрархією категорій та логікою застосування ключових слів для автокатегоризації. IBudgetsService займається обчисленням залишків та контролем лімітів;

- репозиторії (Repositories): шар, що відповідає за низькорівневе виконання SQL-запитів за допомогою технології Entity Framework Core, абстрагуючи специфіку PostgreSQL.

3.3.2 Модулі клієнтської частини

Модулями клієнтської частини є:

- модуль візуалізації та навігації: керується сервісом `INavigationService.cs`, що централізує логіку переходів між сторінками. Головним вікном виступає `MainWindow.xaml`;
- модуль моделей представлення (ViewModels): усі моделі (наприклад, `LoginViewModel`, `TransactionsViewModel`, `BudgetAnalysisViewModel`) успадковуються від базового класу `BaseViewModel.cs`, що реалізує інтерфейс `INotifyPropertyChanged` для реактивного оновлення інтерфейсу;
- інтелектуальний модуль імпорту (Import Engine): координатор `ImportEngine.cs` використовує модуль `FileTypeDetector.cs` для вибору потрібної стратегії для розпізнавання специфічних банківських файлів перед передачею даних на сервер;
- модуль графіки та компонентів: включає власноруч розроблений клас `SimpleBurnUpChart.cs`, що здійснює рендеринг бюджетної графіки через низькорівневі примітиви `DrawingContext`, а також допоміжні класи, такі як `CurrencyConverter.cs` (для форматування валют) та `PasswordBoxHelper.cs` (для безпечного зв'язування паролів).

3.4 Проєктування бази даних

Для надійного збереження всієї фінансової історії використовується реляційна система управління базами даних PostgreSQL.

3.4.1 Структура бази даних

Проектування бази даних здійснювалося з дотриманням принципів нормалізації (3-ї нормальної форми), щоб усунути надлишковість та дублювання даних. Конфігурація структури та контексту даних (DataContext.cs) реалізована за допомогою парадигми Fluent API у фреймворку Entity Framework Core.

Система включає такі основні сутності (таблиці):

- Users: центральний вузол. Зберігає зашифровані облікові дані (email, хеш пароля);
- Accounts: банківські рахунки, готівка або кредитні картки;
- Categories: ієрархічний класифікатор типів витрат та доходів;
- Transactions: основний масив даних. Записи про індивідуальні фінансові операції (сума, дата, опис);
- CategoryRules: набір правил та ключових слів для підсистеми інтелектуальної автокатегоризації;
- Budgets: фінансові ліміти для конкретних категорій на визначені часові інтервали;
- Imports: аудиторський слід для фіксації та відстеження історії масових імпортів даних.

3.4.2 Взаємозв'язки між таблицями

Логічна структура бази даних системи PFMA побудована за реляційною моделлю, де цілісність даних забезпечується за допомогою ідентифікуючих та неідентифікуючих зв'язків типу «один-до-багатьох» (1:N). Центральною сутністю системи є таблиця користувачів, навколо якої групуються всі фінансові та налаштовувальні дані:

- Users – Accounts: один користувач може мати декілька фінансових рахунків (готівка, картки різних банків). Зв'язок реалізовано через зовнішній ключ UserId у таблиці Accounts;
- Users – Categories (1:N): користувач створює власну ієрархію категорій доходів та витрат;
- Users – Budgets (1:N): користувач встановлює ліміти для різних категорій;
- Users – CategoryRules (1:N): користувач визначає персональні правила для автоматичної класифікації транзакцій;
- Users – Imports (1:N): кожна сесія завантаження виписок прив'язана до конкретного ініціатора;
- Users – Transactions (1:N): користувач додає транзакції які належать тільки йому;
- Accounts – Transactions (1:N): кожна транзакція обов'язково належить до одного конкретного рахунку;
- Categories – Transactions (1:N): транзакція класифікується за однією категорією;
- Categories – CategoryRules (1:N): до однієї категорії відноситься певний склад правил (keywords);
- Imports – Transactions (1:N): цей зв'язок є опціональним. Він заповнюється лише тоді, коли транзакція була створена автоматично в процесі імпорту файлу виписки.

Така архітектура бази даних гарантує строгу ізоляцію користувацьких даних, що є критично важливим для додатків у сфері персональних фінансів.

Завдяки чітко налаштованим зовнішнім ключам підтримується посиальна цілісність: це дозволяє безпечно керувати каскадними оновленням та видаленням (наприклад, видалення рахунку автоматично впливає на пов'язані з ним транзакції). Крім того, така висока ступінь нормалізації мінімізує дублювання інформації та створює надійний

фундамент для побудови складних аналітичних запитів, генерації звітів і подальшого масштабування функціоналу системи.

Схема взаємозв'язків між таблицями бази даних зображена на рисунку 3.4.

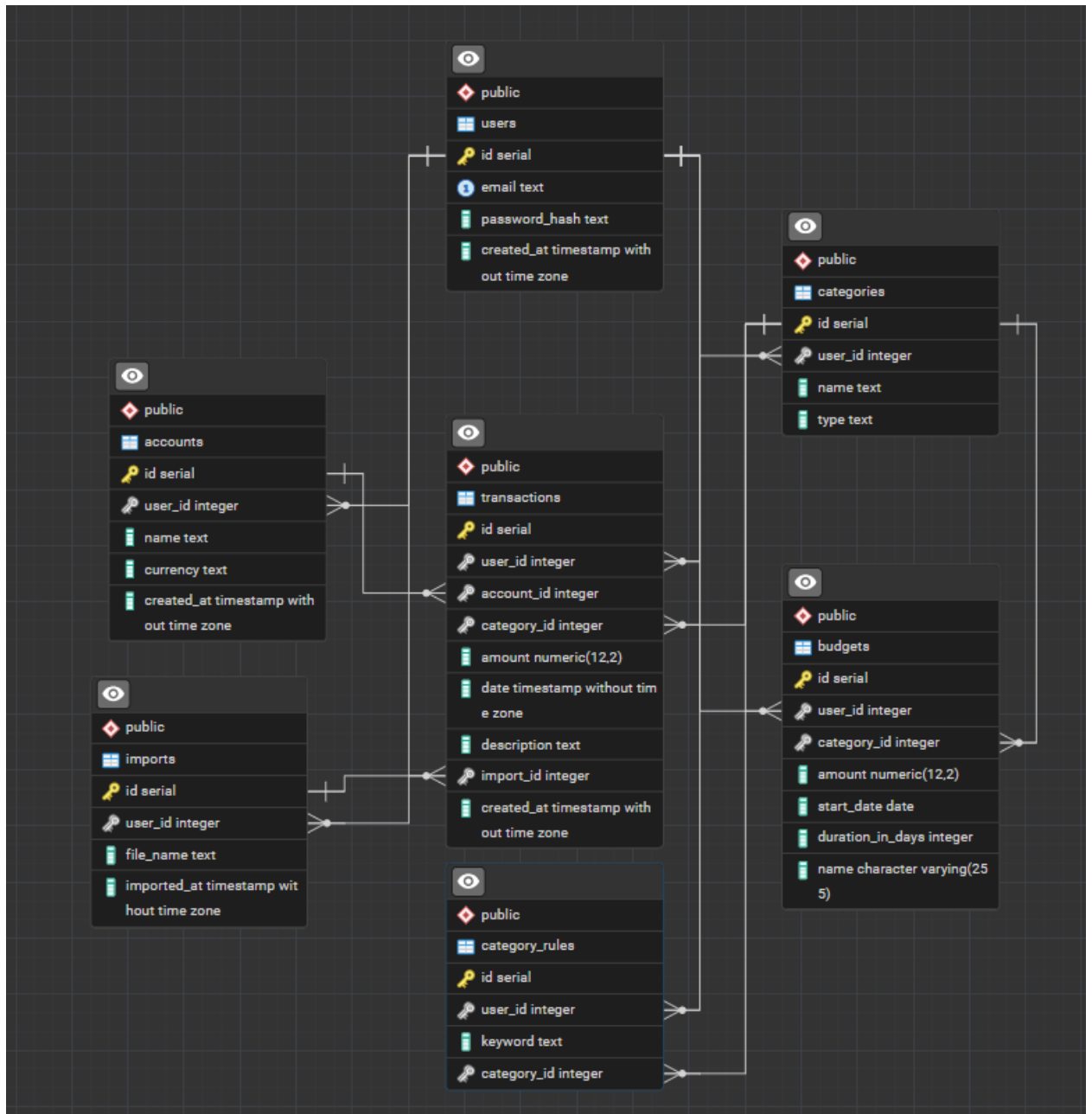


Рисунок 3.4 – Схема взаємозв'язків між таблицями бази даних

3.5 Висновки за розділом 3

У третьому розділі було виконано детальне проектування та опис програмної реалізації клієнт-серверної системи для управління особистими фінансами. Розроблено структурну та функціональну схеми, які ілюструють взаємодію між багаторівневою серверною архітектурою на базі ASP.NET Core та клієнтським WPF-застосунком, реалізованим за архітектурним патерном MVVM. Описано ключові сценарії використання системи та функціонування механізмів обміну даними через безпечний RESTful API з використанням DTO та обгортки ServiceResult.

Спроектовано надійну структуру реляційної бази даних PostgreSQL із суворим дотриманням принципів нормалізації та забезпеченням ізоляції даних користувачів. Деталізовано внутрішні модулі системи, зокрема інтелектуальний механізм масового імпорту (Import Engine) з підтримкою поведінкового патерну "Стратегія" для гнучкого парсингу різноманітних банківських виписок, а також модулі графічної візуалізації бюджетів. Реалізована архітектура повністю задовольняє початкові вимоги щодо надійності, тестованості та здатності до масштабування.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Опис застосування програми

Розроблений програмний продукт «PFMA» (Personal Finance Management Assistant) є багатофункціональним інструментом, призначеним для автоматизації процесів обліку, аналізу та планування персональних і сімейних фінансів. Програма орієнтована на користувачів, які потребують консолідації даних із різних джерел (банківські рахунки, готівкові кошти) у єдиному цифровому середовищі.

Основними експлуатаційними характеристиками системи є:

- автоматизація рутинних операцій: вилучення даних із банківських виписок за допомогою інтелектуальних парсерів;
- інтелектуальна класифікація: використання правил на основі ключових слів для автоматичного розподілу витрат за категоріями;
- прогностична аналітика: візуалізація динаміки витрат через графіки «Burn-up» для контролю дотримання бюджетних лімітів.

4.1.1 Огляд планової цільової аудиторії

Планова цільова аудиторія програмного комплексу була визначена на основі функціональних можливостей системи, що вимагають певної технічної грамотності та активного використання цифрових банківських продуктів. Дослідження ринку FinTech показують, що рівень проникнення фінансових додатків є найвищим серед молодих поколінь.

4.1.1.1 Розподіл за віком

Основним сегментом користувачів є молоді люди та представники середнього віку (від 18 до 40 років):

- студенти та молоді спеціалісти (18–25 років): група, що характеризується високим рівнем використання мобільних і десктопних застосунків та потребою в оптимізації обмежених бюджетів;
- економічно активне населення (26–40 років): найбільш релевантна група, що має стабільні джерела доходу та велику кількість щомісячних транзакцій, що потребують систематизації;
- користувачі віком 40+: сегмент, що демонструє зростаючий інтерес до інструментів цифрового планування за умови наявності інтуїтивно зрозумілого інтерфейсу.

4.1.1.2 Розподіл за фінансовим профілем та рівнем економічної активності

Для глибшого розуміння потреб користувачів було проведено сегментацію аудиторії за типом фінансової поведінки та джерелами доходу. Такий підхід дозволяє оптимізувати інтерфейс під конкретні сценарії використання систем:

- діджитал-мінімалісти: переважно студенти та молоді спеціалісти з відносно невеликою кількістю транзакцій. Для цієї групи критично важливою є швидкість додавання записів та наявність готових шаблонів категорій. Головна мета – базовий контроль залишків на карткових рахунках;
- активні бюджетники: спеціалісти із середнім та високим рівнем доходу, які мають декілька джерел витрат (депозити, кредитні ліміти, готівка). Цей сегмент потребує глибокої аналітики, використання «Burn-up» графіків та деталізованої системи категорій для оптимізації заощаджень;
- фрілансери та самозайняті: користувачі з нерегулярними доходами. Їхньою специфічною потребою є розмежування особистих витрат та операційних витрат на професійну діяльність, а також використання модуля імпорту для обробки великої кількості мультивалютних операцій;

– сімейні менеджери: користувачі, які відповідають за фінансове планування домогосподарства. Основний акцент для них – контроль за дотриманням лімітів у категоріях загального вжитку та аналіз витрат усіх членів родини для запобігання дефіциту бюджету.

4.1.2 Візуалізація планованої цільової аудиторії

Візуалізація структури потенційних споживачів програмного продукту базується на статистичних даних щодо використання цифрових фінансових сервісів (рисунок 4.1 та рисунок 4.2).

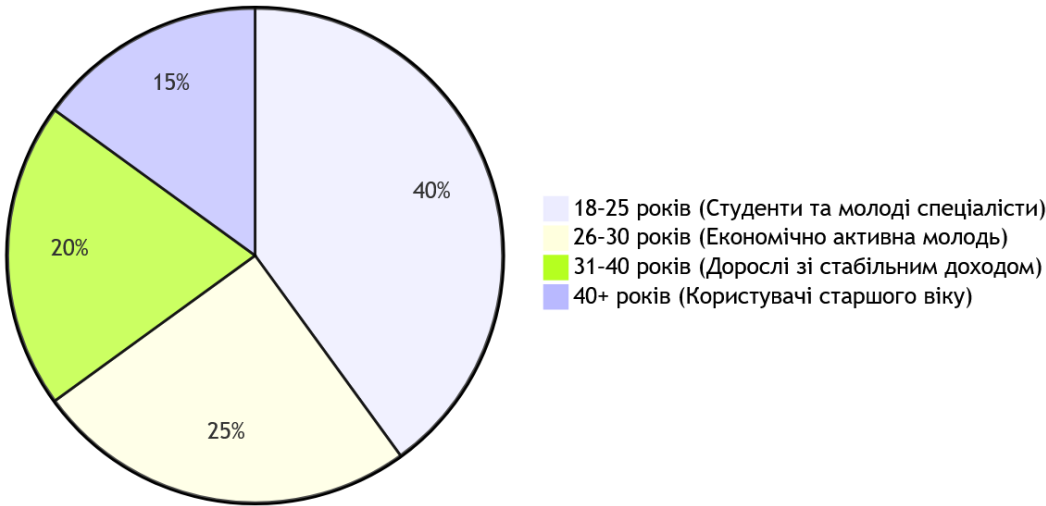


Рисунок 4.1 – Розподіл користувачів за віковими категоріями

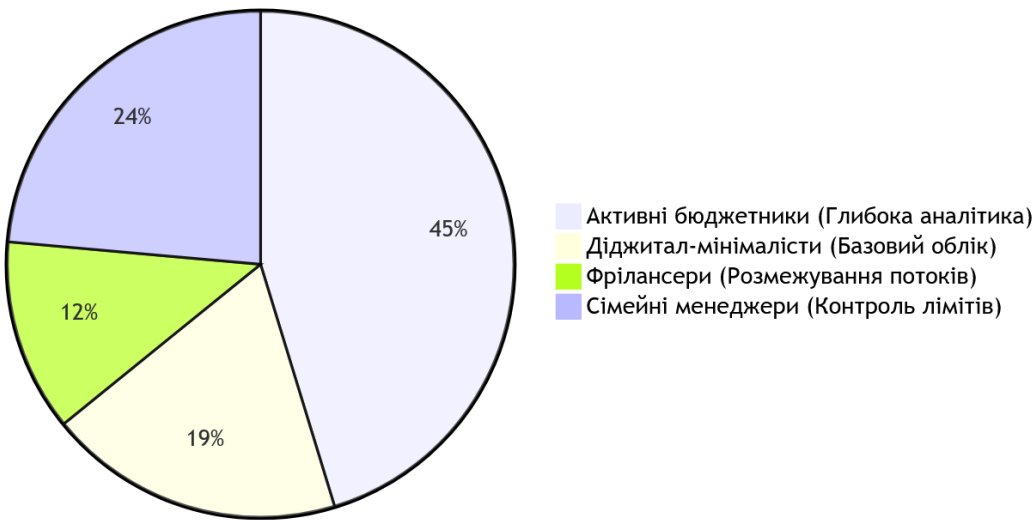


Рисунок 4.2 – Розподіл за фінансовим профілем користувачів

Згідно з візуалізацією, найбільшу групу складають «Активні бюджетники». Саме тому в системі реалізовано розширені аналітичні інструменти та можливість гнучкого налаштування правил автокатегоризації, що відповідає запитам найбільш досвідчених користувачів цифрових фінансових продуктів.

4.2 Інструкція по роботі з програмою

4.2.1 Встановлення програми

Оскільки програмний комплекс «PFMA» побудований за клієнт-серверною архітектурою, він складається з серверної частини (API) та десктопного WPF-клієнта. Для стабільної роботи системи необхідно дотримуватися таких технічних вимог:

а) технічні вимоги до середовища:

- 1) операційна система Windows 10 (версія 1903+) або Windows 11;
- 2) встановлений пакет .NET Runtime 10.0;
- 3) наявність СУБД PostgreSQL версії 16.0 або новіше (для серверної частини);
- 4) стабільне мережеве з'єднання для взаємодії клієнта з API через протокол HTTPS;

б) розгортання та доступ:

- 1) для серверної частини: налаштуйте рядок підключення до бази даних у файлі appsettings.json та виконайте міграції сутностей;
- 2) для клієнтської частини: запустіть виконуваний файл PFMA.Client.exe. Для зручності створіть ярлик застосунку на робочому столі;

в) необхідні дозволи та конфігурація:

- 1) доступ до файлової системи для читання банківських виписок;

2) дозвіл у брандмауері (Firewall) на використання мережевих портів, на яких розгорнуто API.

Схему процесу реєстрації та входу в систему зображено на рисунку

4.3.

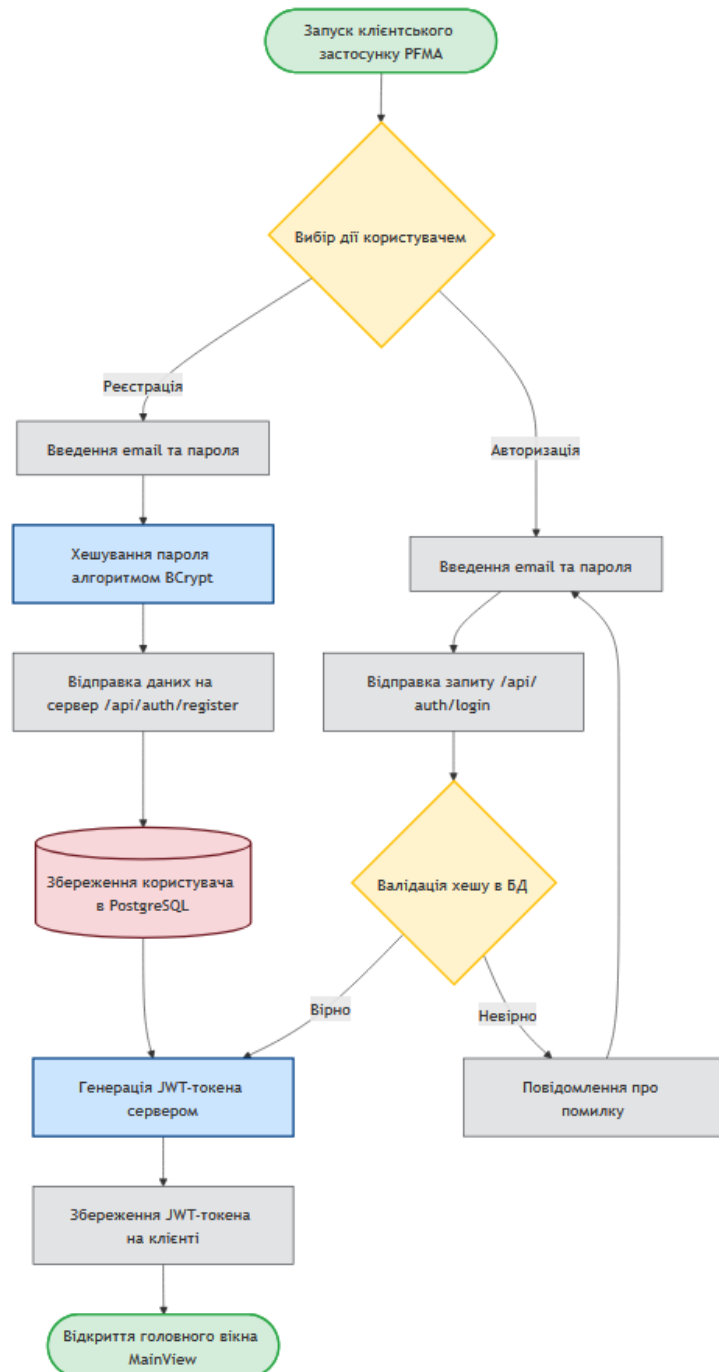


Рисунок 4.3 – Схема процесу авторизації в системі

4.2.2 Запуск програми

Після відкриття десктопного застосунку користувачу необхідно виконати кроки для авторизації та первинної конфігурації фінансового профілю:

а) перший запуск та реєстрація:

- 1) відкрийте застосунок та система відобразить екран логіну;
- 2) для створення нового профілю перейдіть до розділу реєстрації, де необхідно ввести email та пароль. Система автоматично хешує пароль за допомогою алгоритму BCrypt перед відправкою на сервер;

3) після успішної реєстрації сервер згенерує індивідуальний JWT-токен для підпису всіх подальших запитів;

б) налаштування фінансових рахунків та категорій:

- 1) при першому вході система пропонує створити хоча б один рахунок (наприклад, «Готівка» або «Картка Monobank»);
- 2) перейдіть до розділу «Transactions», де ви можете створити, змінити або видалити категорії;

в) конфігурація правил інтелектуального імпорту:

- 1) у розділі «Rules» ви можете визначити ключові слова для автоматичної категоризації (наприклад, слово «Starbucks» автоматично призначатиме категорію «Харчування»);
- 2) налаштуйте бюджетні ліміти для критичних категорій витрат на поточний місяць.

Форма авторизації зображена на рисунку 4.4.

Login - PFMA—□×

Welcome Back

Login to manage your finances

Email Address

Password

Login

[Don't have an account? Register](#)

Рисунок 4.4 – Форма авторизації

Форма реєстрації зображена на рисунку 4.5.

Register - PFMA—□×

Create Account

Join PFMA and track your expenses

Email Address

Password

Repeat Password

Register

[Already have an account? Login](#)

Рисунок 4.5 – Форма реєстрації

Форма керування рахунками зображена на рисунку 4.6.

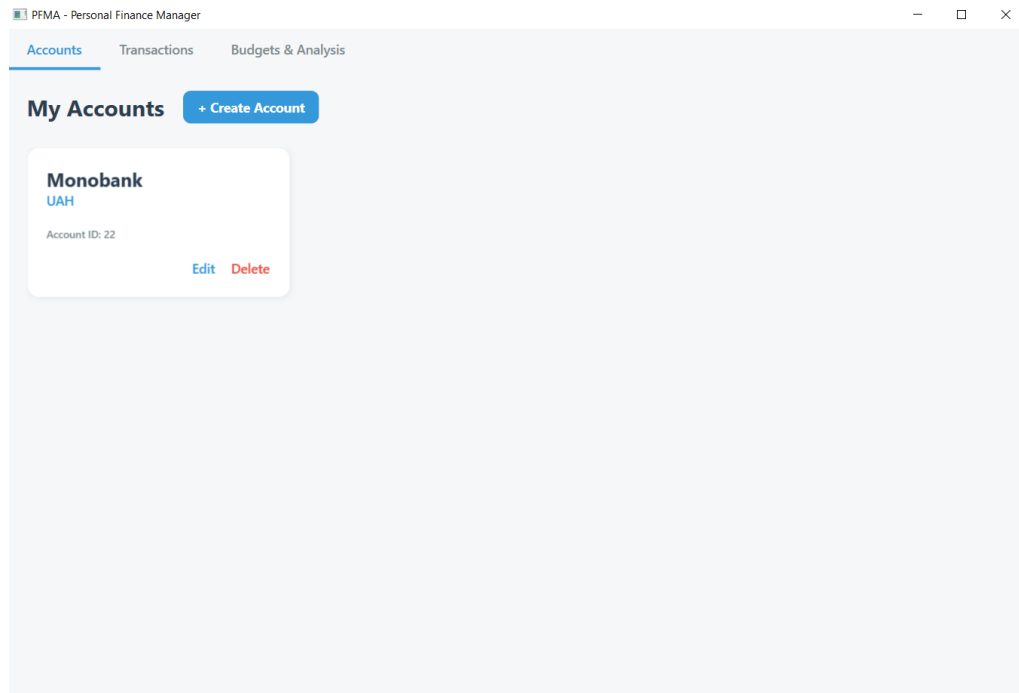


Рисунок 4.6 – Форма керування рахунками

Форма керування категоріями зображена на рисунку 4.7.

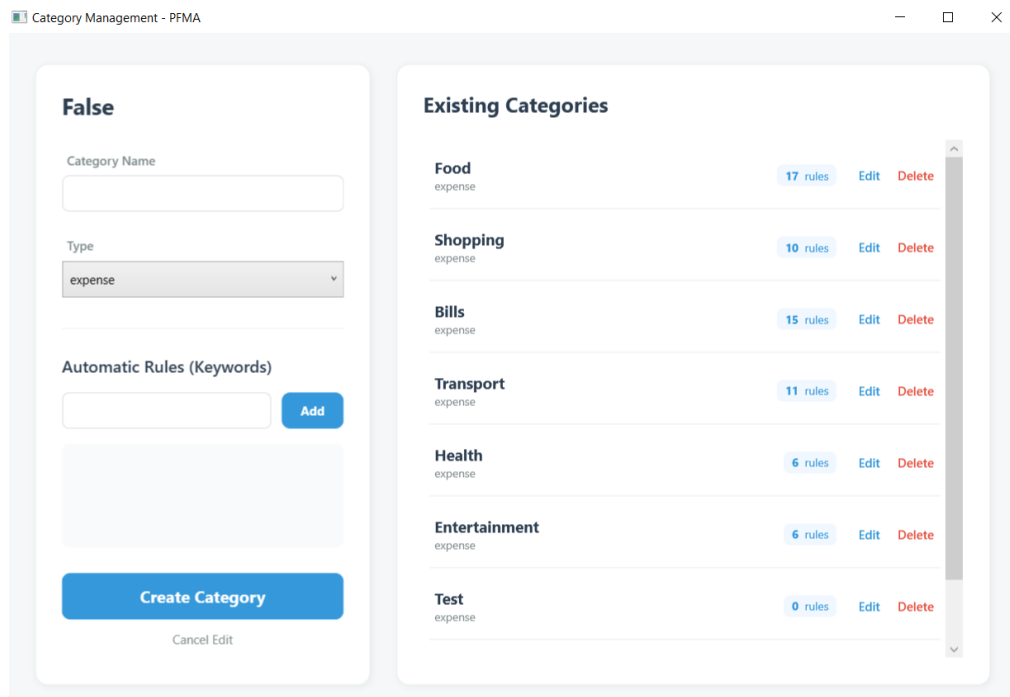


Рисунок 4.7 – Форма керування категоріями

4.2.3 Методика роботи з програмою

Процес взаємодії з системою «PFMA» базується на трьох ключових циклах:

а) управління фінансовими потоками:

- 1) створення та редагування рахунків;
- 2) ведення ручного обліку поодиноких транзакцій (доходи/витрати);
- 3) перегляд детальної історії операцій з можливістю фільтрації за датою та типом;

б) інтелектуальна обробка даних:

- 1) масовий імпорт транзакцій через завантаження банківських виписок;
- 2) автоматичне розпізнавання форматів за допомогою модуля FileTypeDetector;
- 3) верифікація та ручне корегування категорій, які були призначені системою автоматично;

в) аналітика та моніторинг бюджету:

- 1) встановлення місячних лімітів для різних категорій витрат;
- 2) візуалізація темпів витрачання коштів через графіки «Burn-up charts»;
- 3) отримання сповіщень про наближення до критичної межі встановленого бюджету.

Такий комплексний підхід забезпечує безперервний життєвий цикл роботи з персональними фінансами — від збору «сирих» даних до отримання дієвих інсайтів. Поєднання інструментів ручного контролю з розумною автоматизацією (імпортом та автокатегоризацією) суттєво зменшує когнітивне навантаження та час, який витрачається на рутинну роботу.

У результаті система перетворюється з простого інструменту фіксації витрат на проактивного помічника, що дозволяє користувачеві

сфокусуватися на головному: аналізі своїх фінансових звичок, дотриманні дисципліни та досягненні поставлених цілей.

Схема життєвого циклу проєкту в системі зображено на рисунку 4.8.

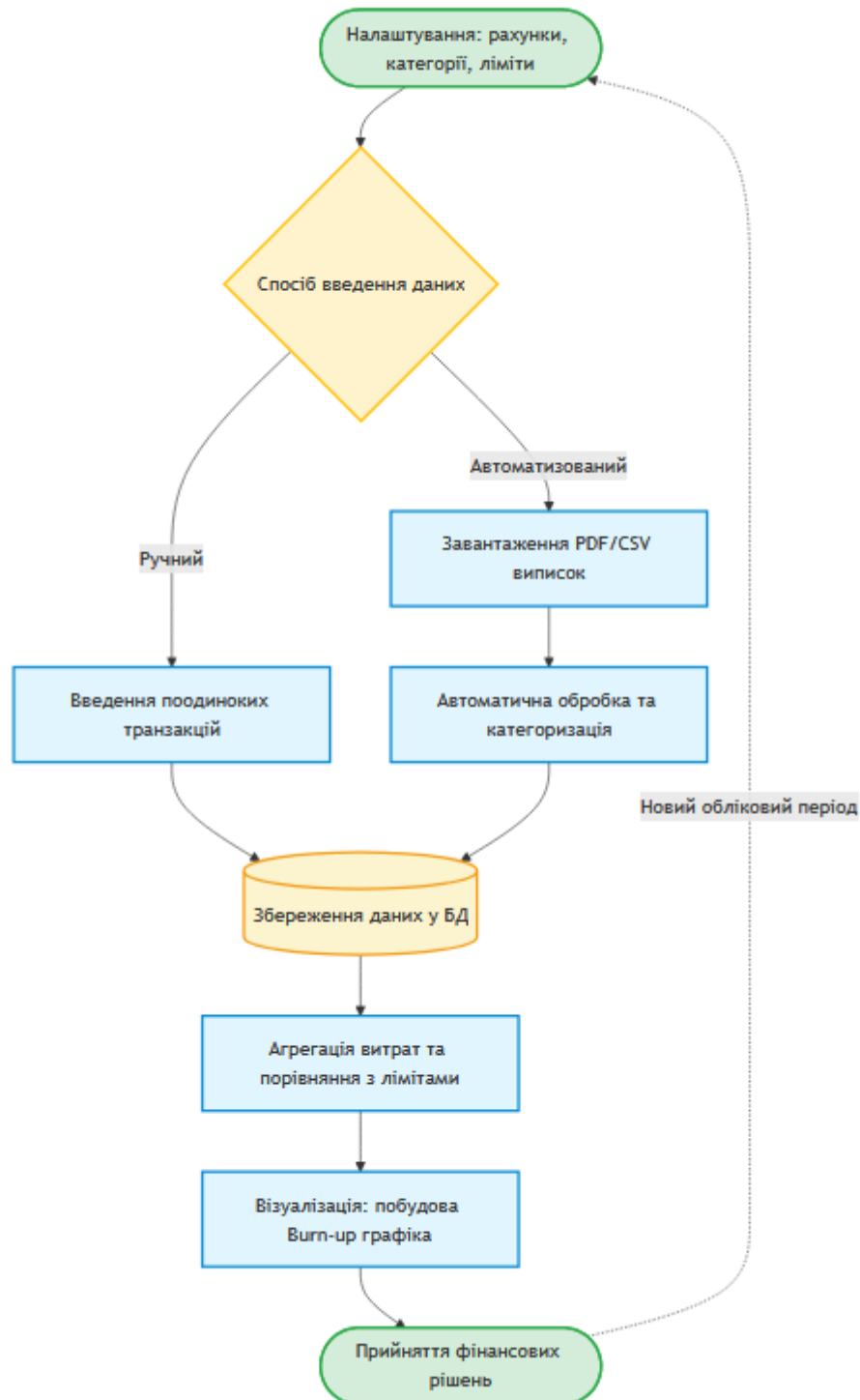


Рисунок 4.8 – Схема життєвого циклу проєкту в системі

4.3 Приклад роботи користувача з програмою

Розглянемо типовий сценарій використання «PFMA» на прикладі користувача, який прагне оптимізувати свої витрати на харчування. Процес розпочинається з ініціалізації робочого простору.

Користувач реєструється в системі та створює основний рахунок «Монобанк». Для автоматизації обліку він завантажує виписку з банку за весь період (рис. 4.9).

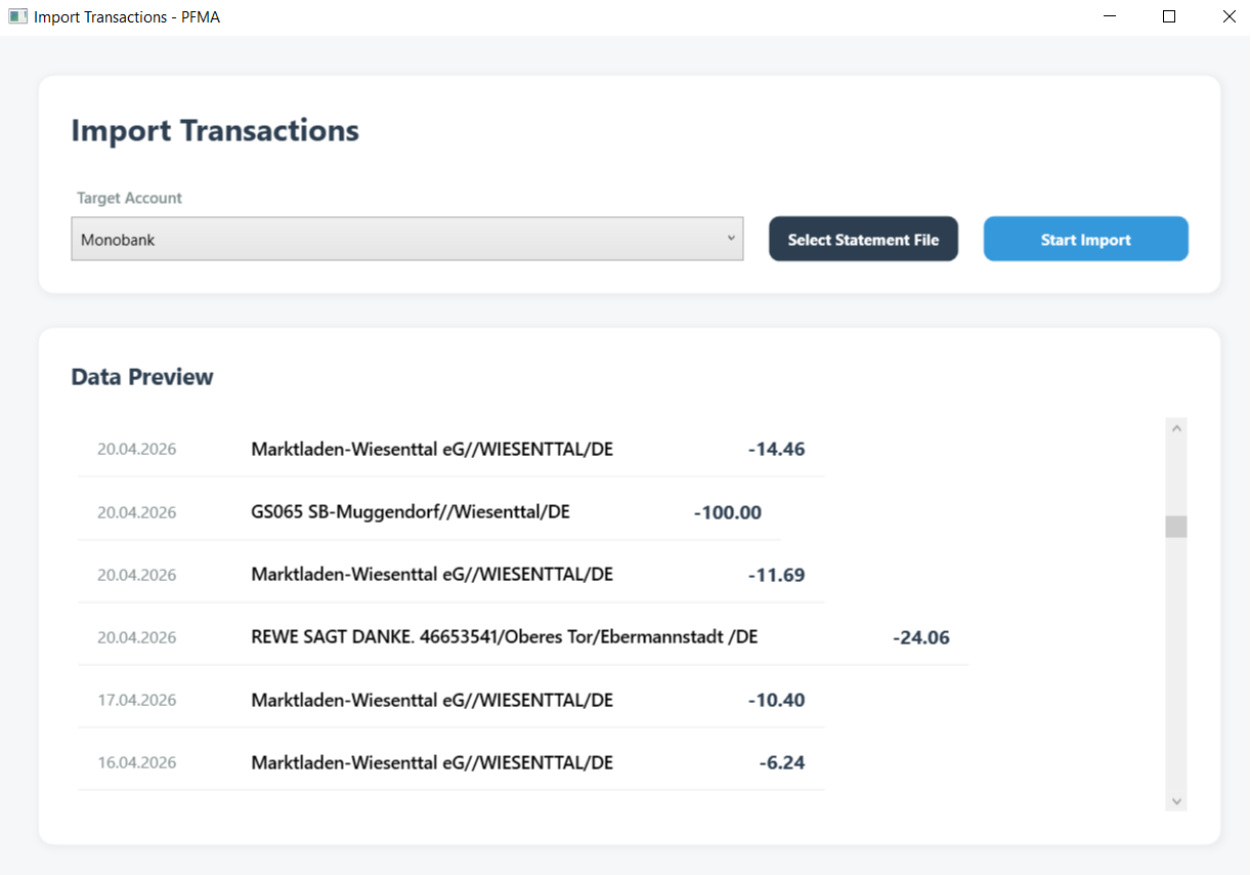


Рисунок 4.9 – Імпорт виписки

Модуль ImportEngine ініціалізує стратегію парсингу, вилучає дані про транзакції та, завдяки налаштованим правилам (наприклад, пошук слова «Rewe»), автоматично призначає їм категорію «Food» (рис. 4.10, 4.11).

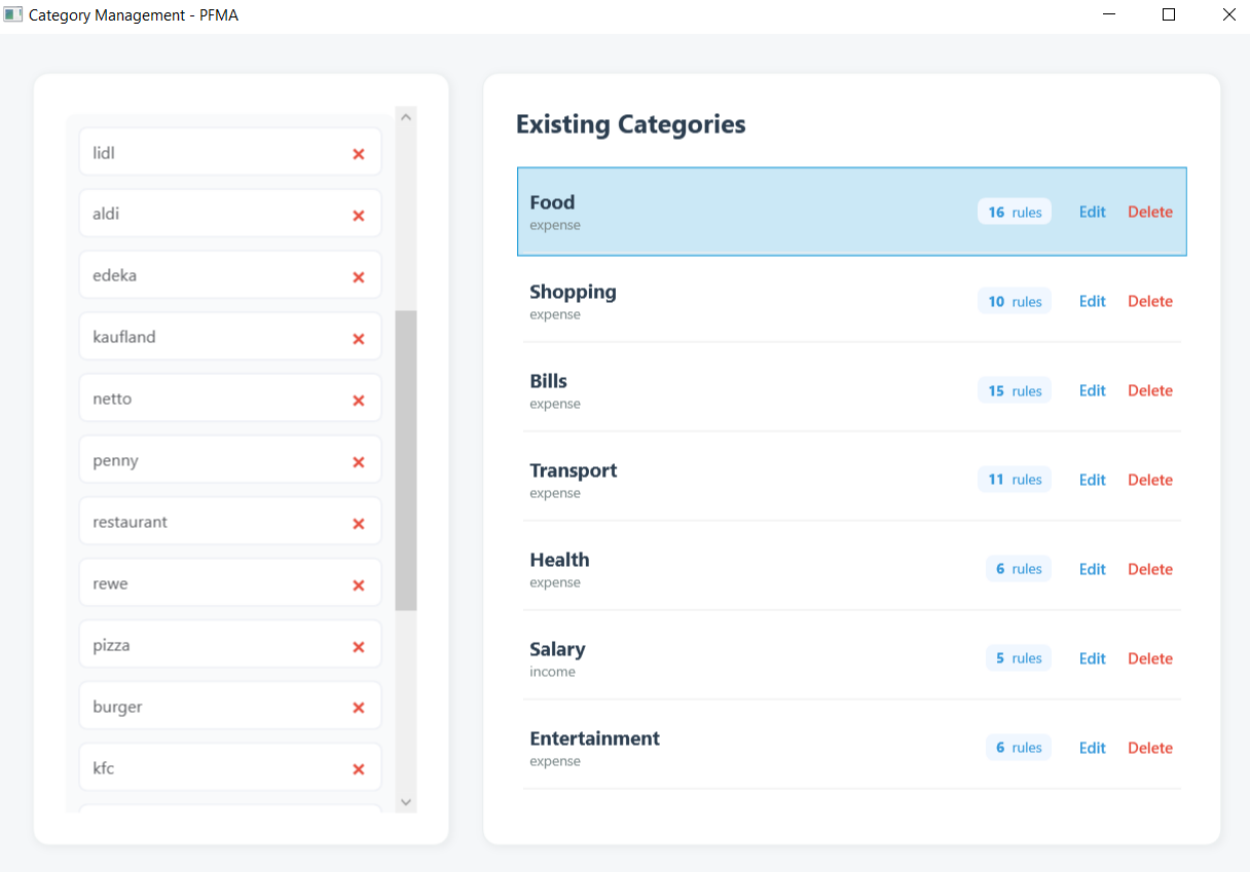


Рисунок 4.10 – Правила категорії "Food"

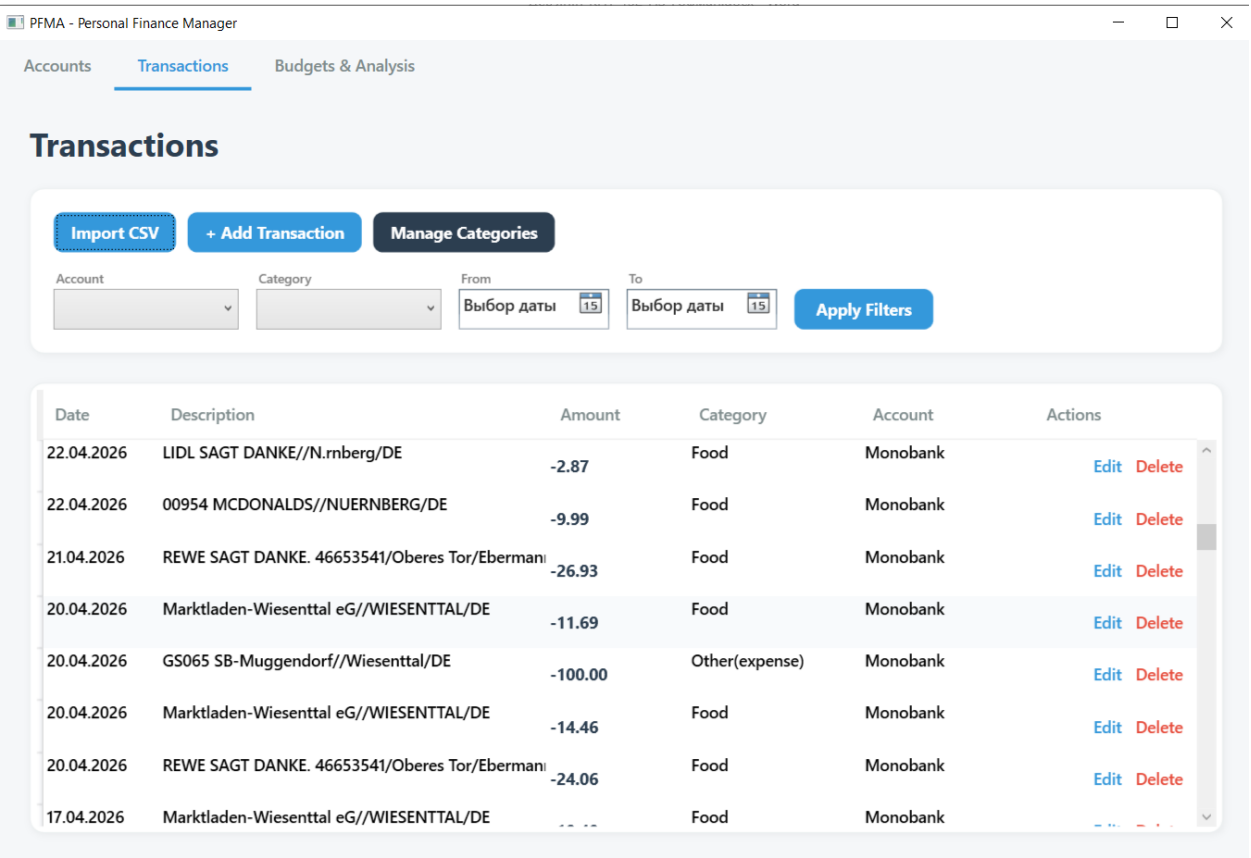


Рисунок 4.11 – Автоматична категоризація

Далі користувач встановлює ліміт часом в 100 днів на категорію «Food» у розмірі 1000 євро (рис. 4.12).

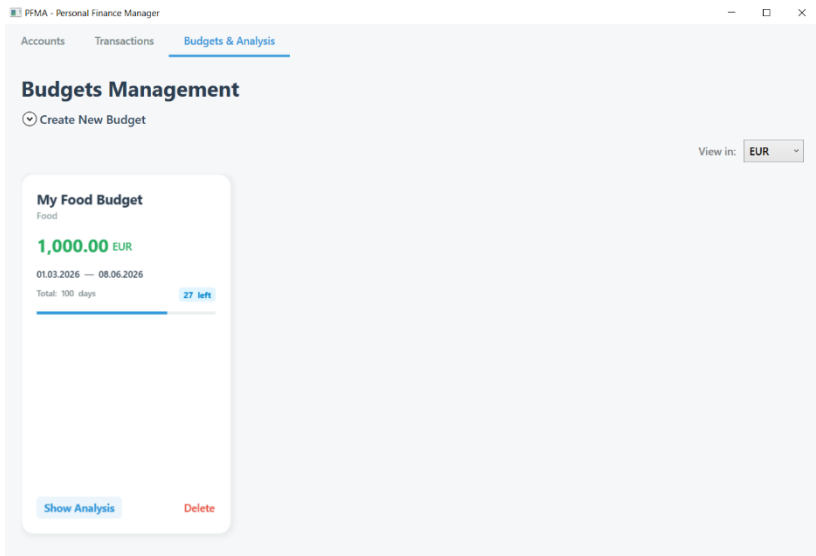


Рисунок 4.12 – Створений бюджет

Додаючи нові витрати, користувач відкриває розділ аналітики, де система будує «Burn-up chart» (рис. 4.13). На графіку він бачить синю лінію фактичних витрат, що рухається відносно зеленої лінії ідеального щоденного бюджету.

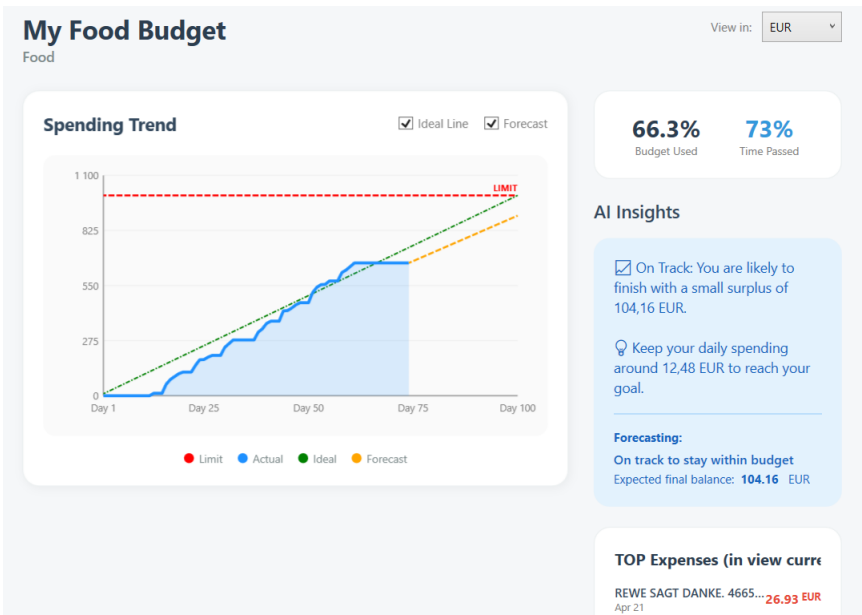


Рисунок 4.13 – Розділ аналітики

Якщо користувач здійснює велику покупку, що вибивається з плану, лінія фактичних витрат різко зростає, сигналізуючи про можливе перевищення ліміту до кінця місяця (рис 4.14).

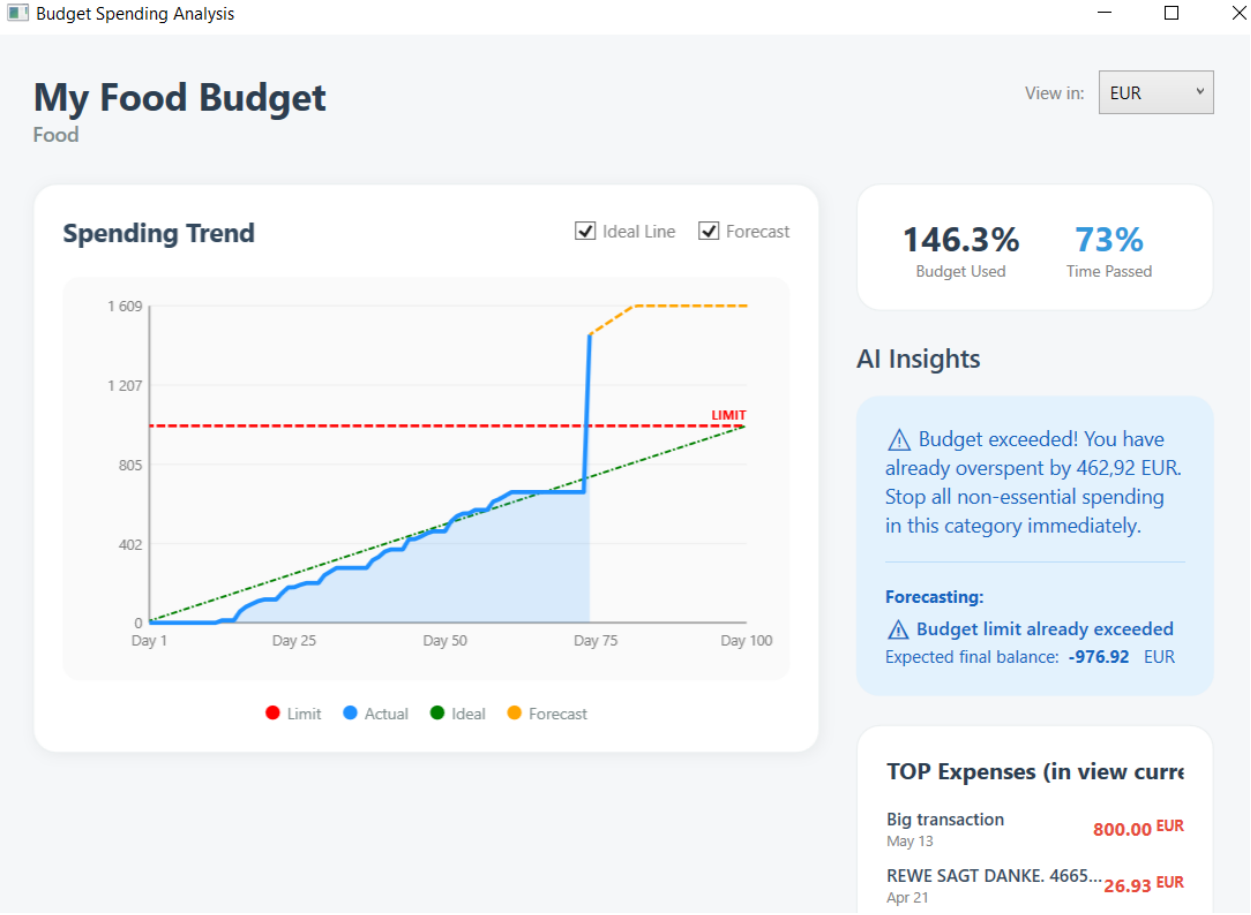


Рисунок 4.14 – Критичне перевищення запланованого бюджету

Завдяки цій візуалізації користувач може оперативно прийняти рішення про економію в наступні дні. Всі зміни балансів та нові транзакції миттєво синхронізуються з сервером через API, забезпечуючи цілісність даних.

4.4 Тестування роботи програми

4.4.1 Функціональне тестування

Для забезпечення надійності та коректності роботи всіх модулів системи «PFMA» було проведено комплексне функціональне тестування, яке включало такі методи:

а) модульне тестування (Unit Testing):

- 1) перевірка сервісів бізнес-логіки (TransactionService, AuthService);
- 2) тестування алгоритмів парсингу банківських виписок у модулі ImportEngine;
- 3) валідація логіки розрахунку залишків та лімітів у BudgetService;
- 4) перевірка коректності мапінгу сутностей через AutoMapper;

б) інтеграційне тестування:

- 1) перевірка взаємодії між API та базою даних PostgreSQL через Entity Framework Core;
- 2) тестування механізмів автентифікації та валідації JWT-токена;
- 3) перевірка цілісності передачі даних між WPF-клієнтом та сервером через HttpClient;

в) тестування користувацького інтерфейсу (UI Testing):

- 1) перевірка реактивності інтерфейсу при зміні станів ViewModels (INotifyPropertyChanged);
- 2) тестування коректності рендерингу графіків SimpleBurnUpChart через DrawingContext;
- 3) перевірка навігації між розділами Dashboard, Transactions та Budgets.

Результати функціонального тестування основних модулів системи наведено в таблиці 4.1.

Таблиця 4.1 – Результати функціонального тестування PFMA

Функціональний модуль	Тест-кейси	Результат	Виправлені помилки
Автентифікація (JWT/BCrypt)	8	Успішно	1
Управління рахунками	12	Успішно	1
Модуль імпорту (PDF/CSV)	20	Успішно	4
Аналіз бюджетів та лімітів	10	Успішно	2
Візуалізація (Burn-up charts)	6	Успішно	1

4.4.2 Аналіз функціонального тестування

Детальна оцінка результатів проведеного тестування підтвердила стабільність основних компонентів системи «PFMA». Загальний показник надійності відповідає встановленим технічним вимогам.

Результати модульного тестування:

- загальна кількість виконаних тестів: 56;
- успішно пройдено: 51 (~91%);
- виявлено технічних невідповідностей: 9;
- успішно виправлено дефектів: 7.

Основні виправлені дефекти:

- дублювання транзакцій: усунуто помилку повторного завантаження однакових записів при декількох послідовних натисканнях кнопки «Import»;

- помилки парсингу PDF: виправлено некоректне зчитування суми транзакцій у виписках з нетиповим форматуванням чисел (розділювачі тисяч);
- валідація лімітів: додано перевірку на від’ємні значення при встановленні місячного бюджету;
- оновлення графіків: виправлено збій рендерингу SimpleBurnUpChart, що виникав при повній відсутності транзакцій у вибраному періоді.

Деталізація критичних помилок, виявлених та усунених під час розробки, наведена в таблиці 4.2.

Таблиця 4.2 – Аналіз виявлених та усунених помилок у PFMA

Тип помилки	Пріоритет	Час виявлення	Час виправлення	Статус
Дублювання даних при імпорті	Високий	10.04.2026	11.04.2026	Виправлено
Помилка валідації JWT	Критичний	15.04.2026	15.04.2026	Виправлено
Розрахунок залишку ліміту	Середній	20.04.2026	21.04.2026	Виправлено
Рендеринг легенди графіка	Низький	25.04.2026	25.04.2026	Виправлено

4.4.3 Тестування продуктивності

Метою тестування продуктивності було визначення стабільності роботи системи під навантаженням та оцінка ефективності використання системних ресурсів клієнтською частиною (WPF).

Методи оцінки продуктивності системи:

а) навантажувальне тестування (Load Testing):

1) перевірка здатності API обробляти запити від 50+ користувачів одночасно;

2) аналіз швидкості виконання LINQ-запитів до PostgreSQL при обсязі бази даних понад 10 000 транзакцій;

б) аналіз споживання системних ресурсів:

1) використання пам'яті: моніторинг Managed Heap у Visual Studio 2026 під час завантаження великих списків транзакцій та перевірка на відсутність витоків пам'яті (Memory Leaks);

2) навантаження на процесор: оцінка інтенсивності обчислень під час роботи ImportEngine та рендерингу складної графіки через DrawingContext;

3) мережева затримка (Latency): вимірювання часу відгуку сервера на запити отримання аналітики (ServiceResult);

Графік динаміки використання пам'яті клієнтською частиною застосунку під час активного робочого сеансу (з моменту завантаження Dashboard до завершення імпорту великої виписки) зображено на рисунку 4.15.

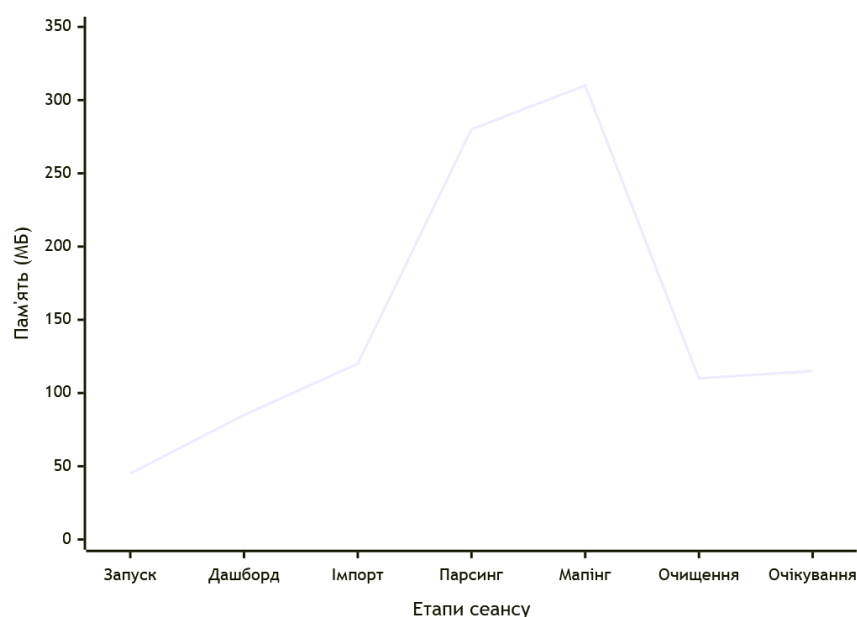


Рисунок 4.15 – Використання пам'яті під час роботи

4.4.4 Аналіз продуктивності

Комплексне оцінювання продуктивності системи «PFMA» проводилося в умовах реальної експлуатації на персональних комп'ютерах під управлінням ОС Windows 10 та Windows 11. Результати демонструють високу швидкість обробки даних та стабільну роботу графічної підсистеми WPF під час побудови аналітичних звітів.

Час ініціалізації та виконання ресурсоємних операцій клієнтським застосунком:

- «холодний» запуск програми (відкриття вікна авторизації): 1.2–1.5 секунди;
- ініціалізація головного вікна (MainView) із завантаженням транзакцій: 0.8–1.2 секунди.

Показники споживання системних ресурсів клієнтською частиною при різних сценаріях використання наведено в таблиці 4.3.

Таблиця 4.3 – Споживання ресурсів клієнтською частиною PFMA

Режим роботи	Використання пам'яті (RAM)	Навантаження на CPU
Очікування (згорнуте вікно)	~45 MB	< 0.1%
Активний перегляд та ручне введення	~85 MB	2–4%
Інтенсивний парсинг виписок	280–320 MB	15–25%

Швидкість виконання ключових операцій (взаємодія з API):

- додавання ручної транзакції: < 0.3 секунди;
- автоматична категоризація 100+ транзакцій після імпорту: < 0.8 секунди;
- рендеринг графіка Burn-up chart через DrawingContext: < 0.1 секунди.

Аналіз цих показників свідчить про те, що використання багатопотоковості (асинхронні виклики `async/await`) у поєднанні з оптимізованими LINQ-запитами Entity Framework Core забезпечує миттєвий відгук інтерфейсу. Операції з розпізнавання файлів не блокують головний потік інтерфейсу (UI Thread).

4.4.5 Тестування безпеки

Для гарантування конфіденційності фінансових даних користувачів та захисту від несанкціонованого доступу було проведено багаторівневе тестування безпеки серверної частини «PFMA API».

Основні напрямки тестування безпеки:

а) тестування автентифікації та сесій:

1) перевірка JWT-токенів: валідація алгоритмів підпису (HMACSHA256), контролю терміну дії та коректності вилучення ідентифікатора користувача;

2) надійність паролів: тестування алгоритму одностороннього хешування BCrypt та перевірка неможливості відновлення пароля з бази даних;

б) тестування захисту даних та API:

1) валідація вхідних даних: перевірка DTO-моделей на сервері на стійкість до некоректних або шкідливих запитів (переповнення значень, від'ємні суми);

2) захист від ін'єкцій: перевірка стійкості до SQL-ін'єкцій завдяки використанню параметризованих запитів в Entity Framework Core;

в) тестування ізоляції даних (Data Privacy):

1) перевірка горизонтального розмежування: тестування сценаріїв, за яких користувач намагається отримати або змінити транзакції/рахунки іншого користувача шляхом підміни ID у HTTP-запиті.

Результати проведеного тестування безпеки наведено в таблиці 4.4.

Таблиця 4.4 – Результати тестування безпеки

Категорія тестування	Кількість тестів	Виявлені вразливості	Усунені вразливості
Автентифікація та підпис JWT	8	1	1
Валідація моделей DTO	12	2	2
Ізоляція фінансових даних користувачів	6	2	2

4.4.6 Аналіз безпеки

За результатами проведеного тестування було виконано комплексний аналіз безпеки застосунку «PFMA», який підтвердив високу ефективність механізмів захисту, закладених в архітектуру ASP.NET Core.

Особлива увага приділялася безпеці кінцевих точок. Аналіз показав, що впровадження атрибута Authorize надійно захищає API від анонімних запитів. Результати симульованого тестування наведено в таблиці 4.5.

Таблиця 4.5 – Результати сканування безпеки PFMA

Метод перевірки	Кількість знайдених вразливостей	Критичні дефекти	Статус усунення
Статичний аналіз коду (SAST)	4	0	Виправлено
Валідація залежностей (NuGet)	2	0	Виправлено
Ручний аналіз (IDOR тестування)	3	1	Виправлено

Основні результати аналізу та виправлення:

- критична вразливість (IDOR): було виявлено можливість модифікації чужої транзакції через передачу чужого TransactionId у PUT-запиті. Проблему усунуто шляхом додавання обов'язкової перевірки userId (з JWT-токена) на рівні репозиторію (TransactionRepository);
- статичний аналіз: виявлено використання DTO без достатньої валідації. Додано атрибути Required та Range для перевірки сум та дат на рівні контролерів.

4.4.7 Аналіз користувацького досвіду

Для оцінки ергономічності та ефективності інтерфейсу системи «PFMA» було проведено апробацію застосунку фокус-групою з 5 осіб після тижневого циклу ведення фінансів (рис. 4.16).

Показники задоволеності інтерфейсом (UI/UX):

- високий рівень (дуже задоволені): 60% (відзначили швидкість модуля імпорту та наочність Burn-up графіків);
- задовільний рівень (задоволені): 20% (позитивно оцінили процес автокатегоризації);
- нейтральний рівень: 20% (висловили пропозиції щодо розширення кольорової палітри графіків).

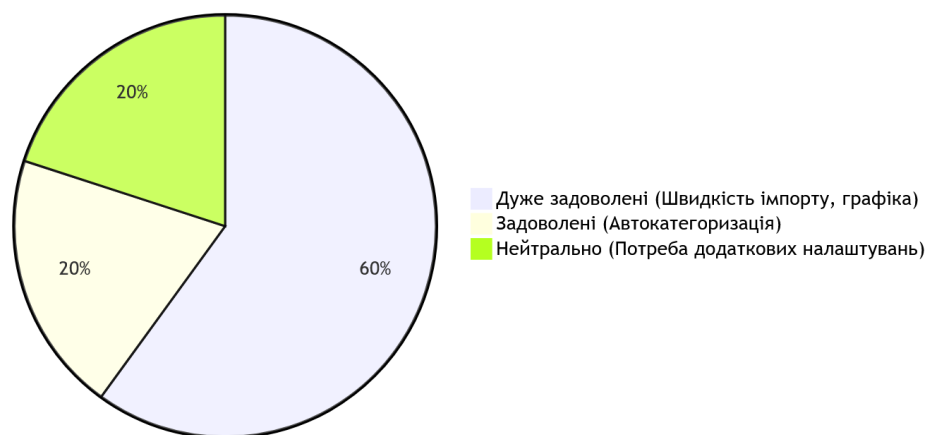


Рисунок 4.16 – Оцінка задоволеності інтерфейсом системи

Результати аналізу частоти використання ключових модулів та їхньої суб'єктивної оцінки користувачами наведено в таблиці 4.6.

Таблиця 4.6 – Популярність функціональних модулів PFMA

Функціональний модуль	Частка звернень	Середня оцінка (max 5)
Модуль імпорту	85%	4.7/5
Аналітика бюджету (Burn-up charts)	95%	4.8/5
Налаштування правил категорій	40%	4.6/5
Ручне введення транзакцій	15%	4.5/5

Тестувальники відзначили високу ефективність рушія імпорту (ImportEngine), який значно економить час порівняно з ручним введенням даних. Найвищий рейтинг отримала візуалізація стану бюджету через графіки спалювання (Burn-up), оскільки вона дозволяє з одного погляду оцінити фінансове становище без необхідності аналізувати сухі цифри у таблицях.

4.5 Рекомендації щодо використання

На основі результатів тестування та аналізу користувацького досвіду було сформовано перелік рекомендацій для максимально ефективного впровадження програмного комплексу «PFMA» в повсякденний процес управління особистими фінансами.

Для оптимального налаштування робочого простору рекомендується:

- створити вичерпне дерево категорій доходів та витрат перед початком активного імпорту даних, щоб уникнути накопичення некласифікованих транзакцій;

- налаштувати персональні правила автокатегоризації (Category Rules), використовуючи найбільш поширені ключові слова з ваших типових покупок для максимальної автоматизації;

- встановити реалістичні місячні ліміти (Budgets) для критичних категорій (наприклад, «Харчування», «Розваги») для забезпечення коректної роботи візуального аналітичного модуля.

Для ефективного використання функціоналу платформи варто:

- систематично завантажувати банківські виписки для масового додавання транзакцій, що мінімізує рутину;

- здійснювати візуальний контроль призначених системою категорій у вікні імпорту перед фінальним підтвердженням збереження даних;

- активно аналізувати графіки спалювання (Burn-up charts) для відстеження динаміки витрат та своєчасного виявлення відхилень від ідеального темпу витрачання бюджету.

Щоденна робота з «PFMA» передбачає перевірку актуальних залишків на рахунках через головну панель (Dashboard), оперативне внесення готівкових операцій вручну та моніторинг наближення витрат до встановлених лімітів. Такий підхід дозволяє підтримувати високий рівень фінансової дисципліни.

Для запобігання помилкам та технічним збоям слід:

- уважно перевіряти результати розпізнавання (парсингу) при завантаженні виписок з банків, чиї формати відрізняються від стандартних, що підтримуються системою;

- уникати створення конфліктуючих правил автокатегоризації (коли одне й те саме ключове слово прив'язується до різних категорій);

- періодично проводити ревізію фінансових цілей та лімітів, актуалізуючи їх відповідно до зміни рівня доходів або поточних інфляційних процесів.

4.6 Плани з розвитку

На основі аналізу отриманих відгуків та дослідження сучасного ринку FinTech-продуктів було визначено стратегію подальшого розвитку платформи «PFMA» у кількох ключових напрямках.

У короткостроковій перспективі передбачається розширення бібліотеки парсерів для повної підтримки специфічних форматів виписок більшості провідних українських банків (окрім вже існуючих розширень). Планується впровадження системи експорту аналітичних звітів та історії транзакцій у формати Excel та PDF для зручного податкового декларування само зайнятими особами.

У довгострокових планах основна увага зосереджена на глибокій інтеграції алгоритмів машинного навчання (Machine Learning) у модуль категоризації. Це дозволить системі самостійно навчатися на основі користувацьких виправлень та контексту транзакцій, поступово відмовляючись від жорстких правил на базі ключових слів. Стратегічно важливим кроком є перехід від ручного завантаження файлів до прямої інтеграції з банківськими установами через протоколи Open Banking (PSD2), що забезпечить автоматичну та безпечну синхронізацію даних у фоновому режимі. Крім того, для забезпечення безперервного доступу до фінансової інформації, розглядається розробка крос платформного мобільного застосунку-компаньйона для iOS/Android, який працюватиме в єдиній екосистемі з поточним сервером ASP.NET Core.

ВИСНОВКИ

З метою підвищення рівня фінансової грамотності, автоматизації обліку та планування особистих бюджетів, у роботі було успішно вирішено задачу створення програмного комплексу «PFMA» (Personal Finance Management Assistant). Отримані результати підтверджують ефективність розробленого інструменту, який є надійним, швидким та інтуїтивно зрозумілим рішенням для управління грошовими потоками як індивідуальних користувачів, так і домогосподарств.

У ході дослідження проведено аналіз існуючих методологій фінансового планування та ринку відповідного програмного забезпечення. Встановлено, що популярні інструменти часто вимагають значних витрат часу на ручне введення транзакцій або не надають достатньо гнучких інструментів для кастомної аналітики та роботи зі специфічними банківськими виписками. Це обґрунтувало необхідність розробки нової системи з акцентом на автоматизований імпорт даних та інтелектуальну класифікацію витрат, що забезпечує швидкий поріг входження.

Архітектуру платформи побудовано за клієнт-серверним принципом. Для розробки десктопної клієнтської частини використано технологію WPF з імплементацією патерну MVVM, а для серверної – фреймворк ASP.NET Core Web API. Застосування Entity Framework Core у поєднанні з реляційною базою даних PostgreSQL дозволило створити гнучку та продуктивну модель даних. Обмін даними стандартизовано за допомогою об'єктів передачі даних (DTO) та універсальної обгортки ServiceResult, що гарантує цілісність інформації та стабільну обробку виняткових ситуацій.

Програмно реалізовано модулі управління рахунками, обробки транзакцій, контролю бюджетних лімітів та налаштування ієрархії категорій. Унікальною особливістю «PFMA» є модуль інтелектуального імпорту (Import Engine) який застосовує користувацькі правила на основі ключових слів для автокатегоризації. Додатково впроваджено високопродуктивну візуалізацію

стану бюджету через графіки спалювання (Burn-up charts), відрендерені за допомогою DrawingContext.

Під час тестування працездатності, продуктивності та безпеки система продемонструвала високі показники надійності. Успішне проходження 91% модульних тестів та оптимізований час завантаження інтерфейсу і парсингу складних PDF-файлів підтверджують технічну досконалість обраних рішень. Виявлені на етапі розробки дефекти, пов'язані з дублюванням транзакцій під час масового імпорту та логікою валідації лімітів, були оперативно усунені, що гарантує стабільну роботу системи.

Безпека даних реалізована через багатошарову систему захисту, що включає автентифікацію на основі JWT-токенів, надійне одностороннє хешування паролів алгоритмом BCrypt та суворе розмежування доступу на рівні API-запитів. Аналіз безпеки підтвердив стійкість платформи до SQL-ін'єкцій та захищеність конфіденційної фінансової інформації від несанкціонованого доступу (IDOR-вразливостей).

На основі аналізу демографічних трендів та фінансової поведінки визначено, що ключову частку користувачів складатимуть технічно обізнані представники економічно активної молоді (40%), діджитал-мінімалісти (25%) та фрілансери (20%). Це дозволяє розробнику чітко фокусувати подальший розвиток продукту на потребах найбільш активних сегментів ринку, які цінують автоматизацію рутини.

Сформовано дорожню карту вдосконалення «PFMA», яка передбачає інтеграцію алгоритмів машинного навчання (Machine Learning) для предикативної категоризації, підключення до протоколів Open Banking для прямої синхронізації з банками та створення крос платформного мобільного застосунку. Розроблений програмний комплекс успішно автоматизує фінансовий менеджмент, забезпечує надійний контроль бюджету та закладає міцний фундамент для масштабування у повноцінну екосистему особистих фінансів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Управління особистими фінансами: стратегії та інструменти в умовах цифрової економіки [Текст] / В. І. Мельник [та ін.] // Економічний вісник. – 2024. – Вип. 12. – С. 45-52.
2. Автоматизація фінансового обліку домогосподарств: проблеми та перспективи [Текст] / О. М. Коваленко // Сучасні інформаційні системи. – 2023. – Т. 7, № 2. – С. 112-118.
3. OECD/INFE 2023 International Survey of Adult Financial Literacy [Electronic resource] / Organisation for Economic Co-operation and Development. – Paris : OECD Publishing, 2023. – 76 p. URL: https://www.oecd.org/en/publications/oecd-infe-2023-international-survey-of-adult-financial-literacy_56003a32-en.html (date of access: 15.05.2026).
4. Personal Finance Software Market Size, Share & Industry Analysis, By Type, By End-User, and Regional Forecast, 2026-2034 [Electronic resource] / Fortune Business Insights. – 2026. URL: <https://www.fortunebusinessinsights.com/personal-finance-software-market-112683> (date of access: 15.05.2026).
5. Arcos-Medina G. Comparative Study of Performance and Productivity of MVC and MVVM design patterns [Текст] / G. Arcos-Medina, J. Menéndez, J. Vallejo // SIIPRIN Conference Proceedings. Simposio Iberoamericano en Programación Informática. – 2018. – P. 241-248. DOI: 10.18502/keg.v1i2.1498.
6. Architectural Synergies and Operational Trade-offs Implementing Role-Based and Policy-Based Authorization in ASP.NET Core APIs [Electronic resource] / ResearchGate. – 2023. URL: <https://www.researchgate.net/publication/403347015> (date of access: 15.05.2026).
7. Фінансова поведінка молоді в умовах макроекономічної нестабільності [Текст] / Т. В. Сидоренко // Соціологія фінансів. – 2025. – Вип. 4. – С. 89-97.

8. The trend of using personal finance management apps among young people [Текст] / T. K. Chi, P. Q. Anh // Global Scientific and Academic Research Journal of Economics, Business and Management. – 2025. – Vol. 4, Issue 6. – P. 28-36.
9. GnuCash: Free Accounting Software. Official Documentation and Features [Electronic resource]. URL: <https://gnucash.org/> (date of access: 15.05.2026).
10. HomeBank: Free Personal Finance and Accounting Software [Electronic resource] / M. Doyen. URL: <https://www.gethomebank.org/> (date of access: 15.05.2026).
11. Actual Budget: Privacy-focused app for managing your finances [Electronic resource]. URL: <https://actualbudget.org/> (date of access: 15.05.2026).

ДОДАТОК А
Технічне завдання

Вступ

Найменування розроблюваного програмного продукту – клієнт-серверний застосунок для обліку особистих фінансів (Personal Finance Management Application – PFMA). Область застосування системи охоплює сферу управління фінансами індивідуальних користувачів та домогосподарств.

A.1 Підстави до розробки

Розробка ведеться на підставі завдання на дипломний проєкт бакалавра (студент Дев'янін О.С., спеціальність 121 «Інженерія програмного забезпечення»), затвердженого наказом Національного університету «Запорізька політехніка» від 7 квітня 2026 року № 139. Тема роботи: «Програмна реалізація застосунку для обліку особистих фінансів».

A.2 Призначення розробки

Програма призначена для комплексної автоматизації управління доходами та витратами користувачів. Головне призначення системи полягає у розв'язанні проблеми фрагментації фінансових даних шляхом консолідації транзакцій, забезпечення масового інтелектуального імпорту банківських виписок, автокатегоризації витрат та надання інструментів для планування і візуального аналізу бюджету.

A.3 Вимоги до програми чи програмного виробу

A.3.1 Вимоги до функціональних характеристик

Система повинна забезпечувати виконання таких функцій:

- реєстрація та автентифікація користувачів за електронною поштою та паролем;

- управління фінансовими рахунками різних типів та ведення обліку в різних валютах;
- виконання повного спектру CRUD-операцій для транзакцій із можливістю їх фільтрації;
- підтримка багаторівневої системи користувацьких категорій;
- автоматична категоризація транзакцій на основі налаштованих ключових слів та резервних категорій (Fallback);
- масовий імпорт транзакцій із зовнішніх CSV-файлів за допомогою адаптивних стратегій парсингу;
- автоматичний захист від дублювання записів при імпорті на основі чотирьох параметрів: рахунок, сума, дата та опис;
- створення бюджетів зі встановленням лімітів на певні часові інтервали та генерація аналітичних графіків виконання бюджету (Burn-up charts).

A.3.2 Вимоги до надійності

Для зберігання даних повинна використовуватися реляційна СУБД PostgreSQL, що гарантує цілісність даних та повну відповідність вимогам ACID (атомарність, узгодженість, ізолюваність, довговічність).

Безпека системи повинна забезпечуватися безстановою авторизацією на основі JSON Web Tokens (JWT) та криптографічним хешуванням паролів за алгоритмом BCrypt.

Внутрішня структура бази даних має бути захищена від витоку шляхом використання об'єктів передачі даних (DTO).

A.3.3 Умови експлуатації

Для коректної роботи десктопного клієнта необхідне стабільне підключення до мережі Інтернет для виконання RESTful HTTP-запитів до

сервера та отримання актуальної інформації. Додаток розрахований на використання особами з базовим рівнем цифрової грамотності.

A.3.4 Вимоги до складу параметрів технічних засобів

Серверна частина: Серверне обладнання або хмарний контейнер (Windows/Linux) зі встановленим середовищем виконання ASP.NET Core Runtime версії 10.0 та базою даних PostgreSQL версії 16.0 або новіше.

Клієнтська частина: Персональний комп'ютер під управлінням ОС Windows 10 (версія 1903+) або Windows 11 зі встановленим пакетом .NET Desktop Runtime 10.0.

A.3.5 Вимоги до інформаційної й програмної сумісності

Система має реалізовувати багатoshарову клієнт-серверну архітектуру.

Клієнтський застосунок має бути побудований за архітектурним патерном MVVM з використанням фреймворку WPF (Windows Presentation Foundation) та мови C#.

Серверна частина має бути реалізована як REST API за допомогою ASP.NET Core 10.0, з використанням патерну Repository та Entity Framework Core для доступу до даних.

Формат обміну даними між клієнтом та сервером – JSON.

A.3.6 Вимоги до маркування й упакування

Клієнтська частина програми постачається у вигляді виконуваного файлу (PFMA.Client.exe). Додаткові вимоги до фізичного пакування не висуваються, оскільки розповсюдження здійснюється цифровими каналами.

A.3.7 Вимоги до транспортування й збереження

Клієнтська частина програми постачається у вигляді виконуваного файлу (PFMA.Client.exe). Додаткові вимоги до фізичного пакування не висуваються, оскільки розповсюдження здійснюється цифровими каналами.

A.4 Вимоги до програмної документації

Супровідна документація до розроблюваного додатку включає технічне завдання та розширену пояснювальну записку. Ці матеріали містять вичерпні відомості про архітектурні рішення проєкту, детальний опис функціональних модулів, а також чітко регламентовані технічні вимоги до середовища розробки та експлуатації програмного продукту.

A.5 Техніко–економічні показники

Економічна ефективність розробленої системи полягає у радикальному скороченні витрат часу користувача (до 70%) на ручне введення транзакцій та їхню класифікацію завдяки автоматичному імпорту виписок та алгоритмам автокатегоризації.

A.6 Стадії й етапи розробки

Розробка ведеться згідно з таким календарним планом:

- постановка завдання роботи (1 тиждень);
- аналіз предметної області (2–3 тижні);
- розробка архітектури програми (4 тиждень);
- розробка програми (5 тиждень);
- тестування та експериментальне дослідження програми (6 тиждень);
- оформлення пояснювальної записки та документів до неї, нормоконтроль та рецензування (7 тиждень);

- захист роботи (8 тиждень).

A.7 Порядок контролю та приймання

Контроль якості програмного забезпечення здійснюється шляхом проведення комплексу тестувань:

- функціональне та модульне тестування логіки (реєстрація, парсинг CSV/PDF, перевірка лімітів). Успішність проходження модульних тестів має складати не менше 91%;
- тестування продуктивності та навантажувальне тестування (аналіз швидкості LINQ-запитів, обробка одночасних підключень 50+ користувачів);
- тестування безпеки (валідація JWT, перевірка на IDOR вразливості, SAST-аналіз);
- апробація користувацького інтерфейсу фокус-групою;
- готова робота підлягає попередньому захисту, проходженню нормоконтролю, перевірці на плагіат, а остаточне приймання відбувається під час публічного захисту перед екзаменаційною комісією.

ДОДАТОК Б
Текст програми

Б.1 Базові компоненти архітектури MVVM

Ці класи забезпечують інфраструктуру реактивного зв'язування даних та асинхронного виконання команд користувача, усуваючи ризики конфліктів багатопотоковості.

Лістинг Б.1 – Текст файлу BaseViewModel.cs

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace PFMA_client.Viewmodels
{
    public abstract class BaseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string
propertyName = null)
            => PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs (propertyName));

        protected bool SetField<T>(ref T field, T value, [CallerMemberName]
string propertyName = null)
        {
            if (Equals(field, value)) return false;
            field = value;
            OnPropertyChanged(propertyName);
            return true;
        }
    }
}
```

Лістинг Б.2 – Текст файлу AsyncRelayCommand.cs

```
using System;
using System.Threading.Tasks;
using System.Windows.Input;

namespace PFMA_client.Commands
{
    public class AsyncRelayCommand : ICommand
    {
        private readonly Func<object, Task> _executeWithParam;
        private readonly Func<Task> _execute;
        private readonly Func<bool> _canExecute;
        private bool _isExecuting;

        public AsyncRelayCommand(Func<Task> execute, Func<bool> canExecute =
null)
        {
            _execute = execute;
            _canExecute = canExecute;
        }
    }
}
```

```

        public AsyncRelayCommand(Func<object, Task> execute, Func<bool>
canExecute = null)
        {
            _executeWithParam = execute;
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter) =>
            !_isExecuting && (_canExecute?.Invoke()?? true);

        public async void Execute(object parameter)
        {
            _isExecuting = true;
            RaiseCanExecuteChanged();

            try
            {
                if (_execute!= null)
                    await _execute();
                else
                    await _executeWithParam(parameter);
            }
            finally
            {
                _isExecuting = false;
                RaiseCanExecuteChanged();
            }
        }

        public event EventHandler CanExecuteChanged;
        private void RaiseCanExecuteChanged() =>
            CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }
}

```

Лістинг Б.3 – Текст файлу RelayCommand.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Input;

namespace PFMA_client.Commands
{
    public class RelayCommand : ICommand
    {
        private readonly Action<object> _execute;
        private readonly Func<object, bool> _canExecute;

        public RelayCommand(Action<object> execute, Func<object, bool>
canExecute = null)
        {
            _execute = execute;
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter) => _canExecute == null ||
_canExecute(parameter);
        public void Execute(object parameter) => _execute(parameter);

        public event EventHandler CanExecuteChanged

```

```

        {
            add => CommandManager.RequerySuggested += value;
            remove => CommandManager.RequerySuggested -= value;
        }
    }
}

```

Лістинг Б.4 – Текст файлу BaseViewModel.cs

```

using PFMA_client.Models;
using PFMA_client.Services;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;
using System.ComponentModel;

namespace PFMA_client.Viewmodels
{
    public class BudgetAnalysisViewModel : BaseViewModel
    {
        private readonly ITransactionService _transactionService;
        private readonly ICurrencyService _currencyService;
        private readonly IAccountService _accountService;

        private BudgetDisplayWrapper _selectedBudget;
        public BudgetDisplayWrapper SelectedBudget
        {
            get => _selectedBudget;
            set { _selectedBudget = value; OnPropertyChanged(); }
        }

        public List<string> Currencies { get; } = new List<string> { "EUR",
"USD", "UAH", "PLN", "GBP" };

        private string _displayCurrency = "EUR";
        public string DisplayCurrency
        {
            get => _displayCurrency;
            set
            {
                _displayCurrency = value;
                OnPropertyChanged();
                _ = RefreshDisplayAsync();
            }
        }

        private List<decimal> _chartDataEur;
        private List<decimal> _chartData;
        public List<decimal> ChartData
        {
            get => _chartData;
            set { _chartData = value; OnPropertyChanged(); }
        }

        private List<decimal> _idealDataPointsEur;
        private List<decimal> _idealDataPoints;
        public List<decimal> IdealDataPoints
        {
            get => _idealDataPoints;
            set { _idealDataPoints = value; OnPropertyChanged(); }
        }
    }
}

```

```

    }

    private decimal _displayBudgetLimit;
    public decimal DisplayBudgetLimit
    {
        get => _displayBudgetLimit;
        set { _displayBudgetLimit = value; OnPropertyChanged(); }
    }

    private decimal _displaySpentAmount;
    public decimal DisplaySpentAmount
    {
        get => _displaySpentAmount;
        set { _displaySpentAmount = value; OnPropertyChanged(); }
    }

    private decimal _displayExpectedBalance;
    public decimal DisplayExpectedBalance
    {
        get => _displayExpectedBalance;
        set { _displayExpectedBalance = value; OnPropertyChanged(); }
    }

    private decimal _spentAmountEur;
    private decimal _expectedBalanceEur;

    private ObservableCollection<TransactionDisplayWrapper>
    _topTransactions = new ObservableCollection<TransactionDisplayWrapper>();
    public ObservableCollection<TransactionDisplayWrapper>
    TopTransactions
    {
        get => _topTransactions;
        set { _topTransactions = value; OnPropertyChanged(); }
    }

    private Dictionary<int, string> _accountCurrencyMap;

    private string _forecastStatus;
    public string ForecastStatus
    {
        get => _forecastStatus;
        set { _forecastStatus = value; OnPropertyChanged(); }
    }

    private bool _showIdealLine = true;
    public bool ShowIdealLine
    {
        get => _showIdealLine;
        set { _showIdealLine = value; OnPropertyChanged(); }
    }

    private bool _showForecastLine = true;
    public bool ShowForecastLine
    {
        get => _showForecastLine;
        set { _showForecastLine = value; OnPropertyChanged(); }
    }

    private string _recommendation;
    public string Recommendation
    {
        get => _recommendation;
        set { _recommendation = value; OnPropertyChanged(); }
    }

```

```

        public double BudgetUtilization => SelectedBudget.Original.amount > 0
? (double) (_spentAmountEur / SelectedBudget.Original.amount) * 100 : 0;
        public double TimeElapsedPercentage => SelectedBudget.Progress * 100;
        public string RemainingPercent => (100 - BudgetUtilization > 0) ?
(100 - BudgetUtilization).ToString("N1") : "0";

        public BudgetAnalysisViewModel(ITransactionService
transactionService, ICurrencyService currencyService, IAccountService
accountService, BudgetDto budget)
        {
            _transactionService = transactionService;
            _currencyService = currencyService;
            _accountService = accountService;
            SelectedBudget = new BudgetDisplayWrapper(budget);

            _ = InitializeAsync();
        }

        private async Task InitializeAsync()
        {
            await UpdateAnalysisAsync();
        }

        private async Task UpdateAnalysisAsync()
        {
            if (SelectedBudget == null) return;

            var startDate = SelectedBudget.Original.start_date;
            var endDate = SelectedBudget.EndDate;
            var categoryId = SelectedBudget.Original.category_id;

            var transactions = await
_transactionService.GetTransactionsAsync(null, categoryId, startDate,
endDate);
            var accounts = await _accountService.GetAccountsAsync();
            _accountCurrencyMap = accounts.ToDictionary(a => a.id, a =>
a.currency);

            var dailyDataEur = new List<decimal>();
            decimal cumulativeEur = 0;

            var today = DateTime.Today;
            var periodDays = SelectedBudget.Original.duration_in_days;

            for (int i = 0; i < periodDays; i++)
            {
                var day = startDate.AddDays(i);
                if (day > today) break;

                decimal daySpendingEur = 0;
                var dayTransactions = transactions.Where(t => t.date.Date ==
day.Date && t.amount < 0).ToList();

                foreach (var t in dayTransactions)
                {
                    if (_accountCurrencyMap.TryGetValue(t.account_id, out var
currency))
                    {
                        var rate = await
_currencyService.GetExchangeRateAsync(currency, "EUR", t.date);
                        daySpendingEur += Math.Abs(t.amount) * rate;
                    }
                }
            }
        }

```

```

        cumulativeEur += daySpendingEur;
        dailyDataEur.Add(cumulativeEur);
    }

    _chartDataEur = dailyDataEur;
    _spentAmountEur = cumulativeEur;
    SelectedBudget.SpentAmount = cumulativeEur;

    var idealListEur = new List<decimal>();
    decimal dailyLimitEur = SelectedBudget.Original.amount /
periodDays;
    for (int i = 0; i < periodDays; i++)
    {
        idealListEur.Add(dailyLimitEur * (i + 1));
    }
    _idealDataPointsEur = idealListEur;

    var top = transactions
        .Where(t => t.amount < 0)
        .OrderBy(t => t.amount)
        .Take(5)
        .Select(t => new TransactionDisplayWrapper(t))
        .ToList();

    TopTransactions.Clear();
    foreach (var t in top) TopTransactions.Add(t);

    int elapsedDays = dailyDataEur.Count;
    if (elapsedDays > 0)
    {
        decimal avgSpendingPerDayEur = cumulativeEur / elapsedDays;
        _expectedBalanceEur = SelectedBudget.Original.amount -
(avgSpendingPerDayEur * periodDays);

        if (cumulativeEur >= SelectedBudget.Original.amount)
        {
            ForecastStatus = "⚠ Budget limit already exceeded";
        }
        else if (avgSpendingPerDayEur > dailyLimitEur &&
        _expectedBalanceEur < 0)
        {
            int daysTillEmpty = (int)(SelectedBudget.Original.amount
/ avgSpendingPerDayEur);
            var emptyDate = startDate.AddDays(daysTillEmpty);
            int remainingDays = daysTillEmpty - elapsedDays;

            if (remainingDays <= 0)
                ForecastStatus = "⚠ Budget is about to run out
today";

            else
                ForecastStatus = $"Predicted to run out by
{emptyDate:MMM dd} (in {remainingDays} days)";
        }
        else
        {
            ForecastStatus = "On track to stay within budget";
        }
    }

    await RefreshDisplayAsync();
}

```

```

private async Task RefreshDisplayAsync()
{
    if (_chartDataEur == null) return;

    var displayRate = await
_currencyService.GetExchangeRateAsync("EUR", DisplayCurrency,
DateTime.Today);

    ChartData = _chartDataEur.Select(v => Math.Round(v * displayRate,
2)).ToList();
    IdealDataPoints = _idealDataPointsEur.Select(v => Math.Round(v *
displayRate, 2)).ToList();

    DisplayBudgetLimit = Math.Round(SelectedBudget.Original.amount *
displayRate, 2);
    DisplaySpentAmount = Math.Round(_spentAmountEur * displayRate,
2);
    DisplayExpectedBalance = Math.Round(_expectedBalanceEur *
displayRate, 2);

    foreach (var wrapper in TopTransactions)
    {
        if
(_accountCurrencyMap.TryGetValue(wrapper.Original.account_id, out var
originalCurrency))
        {
            var rateToEur = await
_currencyService.GetExchangeRateAsync(originalCurrency, "EUR",
wrapper.Original.date);
            var amountEur = Math.Abs(wrapper.Original.amount) *
rateToEur;
            wrapper.DisplayAmount = Math.Round(amountEur *
displayRate, 2);
            wrapper.DisplayCurrency = DisplayCurrency;
        }
    }

    SelectedBudget.DisplayCurrency = DisplayCurrency;
    SelectedBudget.DisplayAmount = DisplayBudgetLimit;
    SelectedBudget.SpentAmount = _spentAmountEur;

    // Notify UI about metrics that depend on these values
    OnPropertyChanged(nameof(BudgetUtilization));
    OnPropertyChanged(nameof(TimeElapsedPercentage));
    OnPropertyChanged(nameof(RemainingPercent));

    GenerateRecommendation();
}

private void GenerateRecommendation()
{
    var limit = DisplayBudgetLimit;
    var spent = DisplaySpentAmount;
    var remaining = limit - spent;
    var daysLeft = SelectedBudget.DaysRemaining;
    var totalDays = SelectedBudget.Original.duration_in_days;
    var elapsedDays = Math.Max(1, (DateTime.Today -
SelectedBudget.Original.start_date).Days + 1);

    if (spent >= limit)
    {
        Recommendation = $"⚠ Budget exceeded! You have already
overspent by {Math.Abs(remaining):N2} {DisplayCurrency}. Stop all non-
essential spending in this category immediately.";
    }
}

```

```

        return;
    }

    if (daysLeft <= 0)
    {
        Recommendation = $"🚩 The budget period has ended. You
finished with {remaining:N2} {DisplayCurrency} remaining. Good job!";
        return;
    }

    // Calculations for deep analysis
    decimal avgSpentPerDay = spent / elapsedDays;
    decimal neededAvgPerDay = remaining / daysLeft;
    decimal projectedFinalSpent = avgSpentPerDay * totalDays;
    decimal projectedResult = limit - projectedFinalSpent;

    string statusPart;
    string actionPart;

    // 1. Determine status based on projection
    if (projectedResult < 0)
    {
        statusPart = $"🔴 Critical: At your current pace, you will
overspend by {Math.Abs(projectedResult):N2} {DisplayCurrency} by the end of
the period.";
        actionPart = $"💡 You are spending {avgSpentPerDay:N2}
{DisplayCurrency}/day. To stay safe, you MUST reduce this to
{neededAvgPerDay:N2} {DisplayCurrency}/day.";
    }
    else if (projectedResult > (limit * 0.2m)) // Saving more than
20%
    {
        statusPart = $"✅ Excellent! You are on track to save
approximately {projectedResult:N2} {DisplayCurrency}.";
        actionPart = $"💡 Your average spend is {avgSpentPerDay:N2}
{DisplayCurrency}/day, which is well below your limit of {neededAvgPerDay:N2}
{DisplayCurrency}/day.";
    }
    else
    {
        statusPart = $"🟢 On Track: You are likely to finish with a
small surplus of {projectedResult:N2} {DisplayCurrency}.";
        actionPart = $"💡 Keep your daily spending around
{neededAvgPerDay:N2} {DisplayCurrency} to reach your goal.";
    }

    Recommendation = $" {statusPart}\n\n {actionPart}";
}
}
}

```

Б.2 Інтелектуальний рушій імпорту банківських виписок

Модуль відповідає за розпізнавання структури файлів CSV та вилучення транзакцій за допомогою поведінкового патерну "Стратегія".

Лістинг Б.5 – Текст файлу Файл IStatementParser.cs

```

using PFMA_client.Models;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace PFMA_client.ImportEngine
{
    public interface IStatementParser
    {
        bool CanParse(ImportContext context);
        Task<List<TransactionDto>> ParseAsync(ImportContext context);
    }
}

```

Лістинг Б.6 – Текст файлу Файл ImportEngine.cs

```

using PFMA_client.Models;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace PFMA_client.ImportEngine
{
    public class ImportEngine
    {
        private readonly List<IStatementParser> _parsers;

        public ImportEngine(IEnumerable<IStatementParser> parsers)
        {
            _parsers = parsers.ToList();
        }

        public async Task<List<TransactionDto>> ImportAsync(string filePath,
int accountId)
        {
            var ctx = new ImportContext
            {
                FilePath = filePath,
                FileType = FileTypeDetector.Detect(filePath),
                AccountId = accountId
            };

            var parser = _parsers.FirstOrDefault(p => p.CanParse(ctx));

            if (parser == null)
            {
                MessageBox.Show("Unknown statement format or bank.");
                return new List<TransactionDto>();
            }

            return await parser.ParseAsync(ctx);
        }
    }
}

```

Лістинг Б.7 – Текст файлу Файл MonoCsvParser.cs

```

using Microsoft.VisualBasic.FileIO;
using PFMA_client.Models;

```

```

using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PFMA_client.ImportEngine.Parsers
{
    public class MonoCsvParser : IStatementParser
    {
        public bool CanParse(ImportContext ctx)
        {
            if (ctx.FileType != "csv") return false;
            var firstLine = File.ReadLines(ctx.FilePath).FirstOrDefault();
            if (firstLine == null) return false;
            return firstLine.Contains("\"Date and time\",Description,MCC");
        }

        public async Task<List<TransactionDto>> ParseAsync(ImportContext ctx)
        {
            var result = new List<TransactionDto>();
            var lines = File.ReadAllLines(ctx.FilePath, Encoding.UTF8);

            for (int i = 1; i < lines.Length; i++)
            {
                var line = lines[i];
                if (string.IsNullOrEmpty(line)) continue;

                var parts = ParseCsvLine(line);
                if (parts == null || parts.Length < 10) continue;

                if (!DateTime.TryParseExact(parts, "dd.MM.yyyy HH:mm:ss",
                    CultureInfo.InvariantCulture, DateTimeStyles.None, out
var date))
                    continue;

                string description = parts.Trim('"');

                if (!decimal.TryParse(parts.Replace(",", "."),
NumberStyles.Any,
                    CultureInfo.InvariantCulture, out var amount))
                    continue;

                result.Add(new TransactionDto
                {
                    date = date,
                    description = description,
                    amount = amount,
                    account_id = ctx.AccountId,
                });
            }
            return result;
        }

        private string ParseCsvLine(string line)
        {
            using (var parser = new TextFieldParser(new StringReader(line)))
            {
                parser.TextFieldType = FieldType.Delimited;
                parser.Delimiters = new { ",", " " };
                parser.HasFieldsEnclosedInQuotes = true;
                return parser.ReadFields();
            }
        }
    }
}

```

```

    }
}
}

```

Лістинг Б.8 – Текст файлу Файл SparkasseCsvParser.cs

```

using Microsoft.VisualBasic.FileIO;
using PFMA_client.Models;
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PFMA_client.ImportEngine.Parsers
{
    public class SparkasseCsvParser : IStatementParser
    {
        public bool CanParse(ImportContext ctx)
        {
            if (ctx.FileType != "csv") return false;
            var firstLine = File.ReadLines(ctx.FilePath).FirstOrDefault();
            if (firstLine == null) return false;

            if (firstLine.Contains("\"Buchungstag\";\"Valutadatum\"") ||
                firstLine.Contains("\"Order account\";\"Day of
entry\";\"Value date\""))
                return true;

            return false;
        }

        public async Task<List<TransactionDto>> ParseAsync(ImportContext ctx)
        {
            var result = new List<TransactionDto>();
            var lines = File.ReadAllLines(ctx.FilePath, Encoding.UTF8);

            var header = ParseCsvLine(lines);
            int idxDate = Array.IndexOf(header, "Day of entry");
            int idxPurpose = Array.IndexOf(header, "Beneficiary/payer");
            int secondIdxPurpose = Array.IndexOf(header, "Purpose");
            int idxAmount = Array.IndexOf(header, "Amount");

            if (idxDate < 0 || idxPurpose < 0 || idxAmount < 0)
                throw new Exception("Sparkasse CSV: missing required
columns");

            for (int i = 1; i < lines.Length; i++)
            {
                var line = lines[i];
                if (string.IsNullOrEmpty(line)) continue;

                var parts = ParseCsvLine(line);
                if (parts == null || parts.Length < 17) continue;

                if (!DateTime.TryParse(parts, out var date)) continue;

                string description = parts[idxPurpose];
                if (description == "") description = parts[secondIdxPurpose];
            }
        }
    }
}

```

```

        if (!decimal.TryParse(parts[idxAmount].Replace(",", "", "."),
NumberStyles.Any,
        CultureInfo.InvariantCulture, out var amount))
            continue;

        result.Add(new TransactionDto
        {
            date = date,
            description = description,
            amount = amount,
            account_id = ctx.AccountId,
        });
    }
    return result;
}

private string ParseCsvLine(string line)
{
    using (var parser = new TextFieldParser(new StringReader(line)))
    {
        parser.TextFieldType = FieldType.Delimited;
        parser.Delimiters = new { ";" };
        parser.HasFieldsEnclosedInQuotes = true;
        return parser.ReadFields();
    }
}
}
}

```

Б.3 Сервісний шар: мережева комунікація та кешування

Ці компоненти забезпечують отримання даних з віддаленого REST API сервера та взаємодію із зовнішніми фінансовими провайдерами для синхронізації валютних курсів з гібридним кешуванням.

Лістинг Б.9 – Текст файлу Файл TransactionService.cs

```

using PFMA_client.Models;
using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Text.Json;
using System.Threading.Tasks;

namespace PFMA_client.Services
{
    public class TransactionService : ITransactionService
    {
        private readonly HttpClient _http;

        public TransactionService(HttpClient http)
        {
            _http = http;
        }
    }
}

```

```

        public async Task<List<TransactionDto>> GetTransactionsAsync(
            int? accountId,
            int? categoryId,
            DateTime? fromDate,
            DateTime? toDate)
        {
            var url = "api/Transaction";
            var query = new List<string>();

            if (accountId.HasValue)
                query.Add($"account_id={accountId.Value}");
            if (categoryId.HasValue)
                query.Add($"category_id={categoryId.Value}");
            if (fromDate.HasValue)
                query.Add($"from_date={DateTime.SpecifyKind(fromDate.Value,
DateTimeKind.Utc):yyyy.MM.dd}");
            if (toDate.HasValue)
                query.Add($"to_date={DateTime.SpecifyKind(toDate.Value,
DateTimeKind.Utc):yyyy.MM.dd}");

            if (query.Count > 0)
                url += "?" + string.Join("&", query);

            var response = await _http.GetAsync(url);
            var json = await response.Content.ReadAsStringAsync();

            return JsonSerializer.Deserialize<List<TransactionDto>>(json,
                new JsonSerializerOptions { PropertyNameCaseInsensitive =
true });
        }

        public async Task BulkCreateAsync(List<CreateTransactionDto>
transactions)
        {
            var dto = new BulkCreateTransactionDto
            {
                transactions = transactions
            };

            var json = JsonSerializer.Serialize(dto);
            var content = new StringContent(json, Encoding.UTF8,
"application/json");

            var response = await _http.PostAsync("api/Transaction/bulk",
content);
            response.EnsureSuccessStatusCode();
        }

        public async Task DeleteAsync(int id)
        {
            var response = await _http.DeleteAsync($"api/Transaction/{id}");
            response.EnsureSuccessStatusCode();
        }
    }
}

```

Лістинг Б.10 – Текст файлу Файл CurrencyService.cs

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using System.Text.Json;

```

```

using System.Threading.Tasks;

namespace PFMA_client.Services
{
    public class CurrencyService : ICurrencyService
    {
        private readonly HttpClient _http;
        private readonly string _cacheFilePath;
        private Dictionary<string, Dictionary<string, decimal>> _cache;
        private const string PivotCurrency = "EUR";

        public CurrencyService(HttpClient http)
        {
            _http = http;
            _cacheFilePath =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "currency_cache.json");
            _cache = LoadCache();
        }

        public async Task InitializeAsync()
        {
            DateTime end = DateTime.Now;
            DateTime start = end.AddMonths(-12);

            await SyncFrankfurterHistory(start, end);
            await SyncUahHistory(start, end);
            PruneCache(start);
            SaveCache();
        }

        private void PruneCache(DateTime threshold)
        {
            {
                var keysToRemove = new List<string>();
                DateTime thresholdDate = GetStartOfWeek(threshold);

                foreach (var key in _cache.Keys)
                {
                    if (DateTime.TryParse(key, out DateTime dt) && dt <
thresholdDate)
                    {
                        keysToRemove.Add(key);
                    }
                }

                foreach (var key in keysToRemove)
                {
                    _cache.Remove(key);
                }
            }
        }

        public async Task<decimal> GetExchangeRateAsync(string fromCurrency,
string toCurrency, DateTime date)
        {
            if (fromCurrency == toCurrency) return 1.0m;
            DateTime startOfWeek = GetStartOfWeek(date);
            string dateKey = startOfWeek.ToString("yyyy-MM-dd");

            if (!_cache.ContainsKey(dateKey))
            {
                await SyncWeek(startOfWeek);
            }

            if (_cache.ContainsKey(dateKey))
            {

```

```

        var rates = _cache[dateKey];
        decimal fromRate = (fromCurrency == PivotCurrency)? 1.0m :
(rates.ContainsKey(fromCurrency)? rates[fromCurrency] : 0);
        decimal toRate = (toCurrency == PivotCurrency)? 1.0m :
(rates.ContainsKey(toCurrency)? rates[toCurrency] : 0);

        if (fromRate!= 0 && toRate!= 0)
        {
            return toRate / fromRate;
        }
    }
    return 1.0m;
}

private async Task SyncFrankfurterHistory(DateTime start, DateTime
end)
{
    try
    {
        string startStr = GetStartOfWeek(start).ToString("yyyy-MM-
dd");

        string endStr = end.ToString("yyyy-MM-dd");
        var url =
$"https://api.frankfurter.app/{startStr}..{endStr}";

        var response = await _http.GetAsync(url);
        if (response.IsSuccessStatusCode)
        {
            var json = await response.Content.ReadAsStringAsync();
            using (JsonDocument doc = JsonDocument.Parse(json))
            {
                var ratesObj = doc.RootElement.GetProperty("rates");
                foreach (var day in ratesObj.EnumerateObject())
                {
                    DateTime dt = DateTime.Parse(day.Name);
                    if (dt.DayOfWeek == DayOfWeek.Monday)
                    {
                        string dateKey = dt.ToString("yyyy-MM-dd");
                        if (!_cache.ContainsKey(dateKey))
                            _cache[dateKey] = new Dictionary<string,
decimal>();

                        foreach (var rate in
day.Value.EnumerateObject())
                        {
                            _cache[dateKey][rate.Name] =
rate.Value.GetDecimal();
                        }
                    }
                }
            }
        }
        catch { }
    }

    private async Task SyncUahHistory(DateTime start, DateTime end)
    {
        DateTime current = GetStartOfWeek(start);
        while (current <= end)
        {
            string dateKey = current.ToString("yyyy-MM-dd");
            if (!_cache.ContainsKey(dateKey)) _cache[dateKey] = new
Dictionary<string, decimal>();

```

```

        if (!_cache[dateKey].ContainsKey("UAH"))
        {
            decimal uahRate = await FetchNbuRate("EUR",
current.ToString("yyyyMMdd"));
            if (uahRate != 0)
            {
                _cache[dateKey]["UAH"] = uahRate;
            }
        }
        current = current.AddDays(7);
    }
}

private async Task SyncWeek(DateTime monday)
{
    await SyncFrankfurterHistory(monday, monday.AddDays(6));
    await SyncUahHistory(monday, monday.AddDays(6));
    SaveCache();
}

private async Task<decimal> FetchNbuRate(string currency, string
dateStr)
{
    try
    {
        var url =
$"https://bank.gov.ua/NBUStatService/v1/statdirectory/exchange?valcode={curre
ncy}&date={dateStr}&json";
        var response = await _http.GetAsync(url);
        if (response.IsSuccessStatusCode)
        {
            var json = await response.Content.ReadAsStringAsync();
            using (JsonDocument doc = JsonDocument.Parse(json))
            {
                var root = doc.RootElement;
                if (root.ValueKind == JsonValueKind.Array &&
root.GetArrayLength() > 0)
                {
                    return root.GetProperty("rate").GetDecimal();
                }
            }
        }
    }
    catch { }
    return 0;
}

private DateTime GetStartOfWeek(DateTime dt)
{
    int diff = (7 + (dt.DayOfWeek - DayOfWeek.Monday)) % 7;
    return dt.AddDays(-1 * diff).Date;
}

private Dictionary<string, Dictionary<string, decimal>> LoadCache()
{
    if (!File.Exists(_cacheFilePath))
        return new Dictionary<string, Dictionary<string, decimal>>();
    try
    {
        string json = File.ReadAllText(_cacheFilePath);
        return JsonSerializer.Deserialize<Dictionary<string,
Dictionary<string, decimal>>>(json)

```

```

        ?? new Dictionary<string, Dictionary<string,
decimal>>());
    }
    catch
    {
        return new Dictionary<string, Dictionary<string, decimal>>();
    }
}

private void SaveCache()
{
    try
    {
        string json = JsonSerializer.Serialize(_cache, new
JsonSerializerOptions { WriteIndented = true });
        File.WriteAllText(_cacheFilePath, json);
    }
    catch { }
}
}
}

```

Б.4 Бізнес-логіка моделей представлення (ViewModels)

Цей шар агрегує дані, виконує складні аналітичні обчислення, прогнозування лімітів та готує колекції для безпечного відображення у користувацькому інтерфейсі.

Лістинг Б.11 – Текст файлу Файл BudgetAnalysisViewModel.cs

```

using PFMA_client.Models;
using PFMA_client.Services;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Threading.Tasks;

namespace PFMA_client.Viewmodels
{
    public class BudgetAnalysisViewModel : BaseViewModel
    {
        private readonly ITransactionService _transactionService;
        private readonly ICurrencyService _currencyService;
        private readonly IAccountService _accountService;

        private BudgetDisplayWrapper _selectedBudget;
        public BudgetDisplayWrapper SelectedBudget
        {
            get => _selectedBudget;
            set { _selectedBudget = value; OnPropertyChanged(); }
        }

        public List<string> Currencies { get; } = new List<string> { "EUR",
"USD", "UAH", "PLN", "GBP" };
    }
}

```

```

private string _displayCurrency = "EUR";
public string DisplayCurrency
{
    get => _displayCurrency;
    set
    {
        _displayCurrency = value;
        OnPropertyChanged();
        _ = RefreshDisplayAsync();
    }
}

private List<decimal> _chartDataEur;
private List<decimal> _chartData;
public List<decimal> ChartData
{
    get => _chartData;
    set { _chartData = value; OnPropertyChanged(); }
}

private List<decimal> _idealDataPointsEur;
private List<decimal> _idealDataPoints;
public List<decimal> IdealDataPoints
{
    get => _idealDataPoints;
    set { _idealDataPoints = value; OnPropertyChanged(); }
}

private decimal _displayBudgetLimit;
public decimal DisplayBudgetLimit
{
    get => _displayBudgetLimit;
    set { _displayBudgetLimit = value; OnPropertyChanged(); }
}

private decimal _displaySpentAmount;
public decimal DisplaySpentAmount
{
    get => _displaySpentAmount;
    set { _displaySpentAmount = value; OnPropertyChanged(); }
}

private decimal _displayExpectedBalance;
public decimal DisplayExpectedBalance
{
    get => _displayExpectedBalance;
    set { _displayExpectedBalance = value; OnPropertyChanged(); }
}

private decimal _spentAmountEur;
private decimal _expectedBalanceEur;
private Dictionary<int, string> _accountCurrencyMap;

private string _forecastStatus;
public string ForecastStatus
{
    get => _forecastStatus;
    set { _forecastStatus = value; OnPropertyChanged(); }
}

private string _recommendation;
public string Recommendation
{
    get => _recommendation;

```

```

        set { _recommendation = value; OnPropertyChanged(); }
    }

    public double BudgetUtilization => SelectedBudget.Original.amount >
0?      (double) (_spentAmountEur / SelectedBudget.Original.amount) * 100
: 0;

    public double TimeElapsedPercentage => SelectedBudget.Progress * 100;

    public BudgetAnalysisViewModel(ITransactionService
transactionService, ICurrencyService currencyService, IAccountService
accountService, BudgetDto budget)
    {
        _transactionService = transactionService;
        _currencyService = currencyService;
        _accountService = accountService;
        SelectedBudget = new BudgetDisplayWrapper(budget);

        _ = InitializeAsync();
    }

    private async Task InitializeAsync()
    {
        await UpdateAnalysisAsync();
    }

    private async Task UpdateAnalysisAsync()
    {
        if (SelectedBudget == null) return;
        var startDate = SelectedBudget.Original.start_date;
        var endDate = SelectedBudget.EndDate;
        var categoryId = SelectedBudget.Original.category_id;

        var transactions = await
_transactionService.GetTransactionsAsync(null, categoryId, startDate,
endDate);
        var accounts = await _accountService.GetAccountsAsync();
        _accountCurrencyMap = accounts.ToDictionary(a => a.id, a =>
a.currency);

        var dailyDataEur = new List<decimal>();
        decimal cumulativeEur = 0;

        var today = DateTime.Today;
        var periodDays = SelectedBudget.Original.duration_in_days;

        for (int i = 0; i < periodDays; i++)
        {
            var day = startDate.AddDays(i);
            if (day > today) break;

            decimal daySpendingEur = 0;
            var dayTransactions = transactions.Where(t => t.date.Date ==
day.Date && t.amount < 0).ToList();

            foreach (var t in dayTransactions)
            {
                if (_accountCurrencyMap.TryGetValue(t.account_id, out var
currency))
                {
                    var rate = await
_currencyService.GetExchangeRateAsync(currency, "EUR", t.date);
                    daySpendingEur += Math.Abs(t.amount) * rate;

```

```

        }
    }

    cumulativeEur += daySpendingEur;
    dailyDataEur.Add(cumulativeEur);
}

_chartDataEur = dailyDataEur;
_spentAmountEur = cumulativeEur;
SelectedBudget.SpentAmount = cumulativeEur;

var idealListEur = new List<decimal>();
decimal dailyLimitEur = SelectedBudget.Original.amount /
periodDays;
for (int i = 0; i < periodDays; i++)
{
    idealListEur.Add(dailyLimitEur * (i + 1));
}
_idealDataPointsEur = idealListEur;

int elapsedDays = dailyDataEur.Count;
if (elapsedDays > 0)
{
    decimal avgSpendingPerDayEur = cumulativeEur / elapsedDays;
    _expectedBalanceEur = SelectedBudget.Original.amount -
    (avgSpendingPerDayEur * periodDays);

    if (cumulativeEur >= SelectedBudget.Original.amount)
    {
        ForecastStatus = "⚠ Budget limit already exceeded";
    }
    else if (avgSpendingPerDayEur > dailyLimitEur &&
    _expectedBalanceEur < 0)
    {
        int daysTillEmpty = (int)(SelectedBudget.Original.amount
    / avgSpendingPerDayEur);
        var emptyDate = startDate.AddDays(daysTillEmpty);
        int remainingDays = daysTillEmpty - elapsedDays;
        if (remainingDays <= 0)
            ForecastStatus = "⚠ Budget is about to run out
today";

        else
            ForecastStatus = $"Predicted to run out by
{emptyDate:MMM dd} (in {remainingDays} days)";
    }
    else
    {
        ForecastStatus = "On track to stay within budget";
    }
}

await RefreshDisplayAsync();
}

private async Task RefreshDisplayAsync()
{
    if (_chartDataEur == null) return;
    var displayRate = await
_currencyService.GetExchangeRateAsync("EUR", DisplayCurrency,
DateTime.Today);

    ChartData = _chartDataEur.Select(v => Math.Round(v * displayRate,
2)).ToList();
}

```

```

        IdealDataPoints = _idealDataPointsEur.Select(v => Math.Round(v *
displayRate, 2)).ToList();

        DisplayBudgetLimit = Math.Round(SelectedBudget.Original.amount *
displayRate, 2);
        DisplaySpentAmount = Math.Round(_spentAmountEur * displayRate,
2);
        DisplayExpectedBalance = Math.Round(_expectedBalanceEur *
displayRate, 2);

        SelectedBudget.DisplayCurrency = DisplayCurrency;
        SelectedBudget.DisplayAmount = DisplayBudgetLimit;
        SelectedBudget.SpentAmount = _spentAmountEur;

        OnPropertyChanged(nameof(BudgetUtilization));
        OnPropertyChanged(nameof(TimeElapsedPercentage));

        GenerateRecommendation();
    }

    private void GenerateRecommendation()
    {
        var limit = DisplayBudgetLimit;
        var spent = DisplaySpentAmount;
        var remaining = limit - spent;
        var daysLeft = SelectedBudget.DaysRemaining;
        var totalDays = SelectedBudget.Original.duration_in_days;
        var elapsedDays = Math.Max(1, (DateTime.Today -
SelectedBudget.Original.start_date).Days + 1);

        if (spent >= limit)
        {
            Recommendation = $"⚠ Budget exceeded! You have already
overspent by {Math.Abs(remaining):N2} {DisplayCurrency}. Stop all non-
essential spending in this category immediately.";
            return;
        }

        if (daysLeft <= 0)
        {
            Recommendation = $"🏠 The budget period has ended. You
finished with {remaining:N2} {DisplayCurrency} remaining. Good job!";
            return;
        }

        decimal avgSpentPerDay = spent / elapsedDays;
        decimal neededAvgPerDay = remaining / daysLeft;
        decimal projectedFinalSpent = avgSpentPerDay * totalDays;
        decimal projectedResult = limit - projectedFinalSpent;

        string statusPart;
        string actionPart;

        if (projectedResult < 0)
        {
            statusPart = $"🔴 Critical: At your current pace, you will
overspend by {Math.Abs(projectedResult):N2} {DisplayCurrency} by the end of
the period.";
            actionPart = $"💡 You are spending {avgSpentPerDay:N2}
{DisplayCurrency}/day. To stay safe, you MUST reduce this to
{neededAvgPerDay:N2} {DisplayCurrency}/day.";
        }
        else if (projectedResult > (limit * 0.2m))

```

```

        {
            statusPart = $"✔ Excellent! You are on track to save
approximately {projectedResult:N2} {DisplayCurrency}.";
            actionPart = $"💡 Your average spend is {avgSpentPerDay:N2}
{DisplayCurrency}/day, which is well below your limit of {neededAvgPerDay:N2}
{DisplayCurrency}/day.";
        }
        else
        {
            statusPart = $"🚩 On Track: You are likely to finish with a
small surplus of {projectedResult:N2} {DisplayCurrency}.";
            actionPart = $"💡 Keep your daily spending around
{neededAvgPerDay:N2} {DisplayCurrency} to reach your goal.";
        }

        Recommendation = $"{statusPart}\n\n{actionPart}";
    }
}
}

```

Б.5 Кастомний компонент графічного рендерингу (DirectX / WPF)

Модуль малювання графіків SimpleBurnUpChart забезпечує відмальовування кривих "згоряння" бюджету. Перевизначення методу OnRender обходить стандартний цикл WPF-компонування та забезпечує мікросекундну швидкодію навіть для річних графіків.

Лістинг Б.12 – Текст файлу Файл SimpleBurnUpChart.cs

```

using System;
using System.Windows;
using System.Windows.Media;
using System.Collections.Generic;
using System.Linq;

namespace PFMA_client.View
{
    public class SimpleBurnUpChart : FrameworkElement
    {
        public static readonly DependencyProperty DataPointsProperty =
            DependencyProperty.Register("DataPoints", typeof(List<decimal>),
            typeof(SimpleBurnUpChart),
            new FrameworkPropertyMetadata(null,
            FrameworkPropertyMetadataOptions.AffectsRender));

        public List<decimal> DataPoints
        {
            get => (List<decimal>)GetValue(DataPointsProperty);
            set => SetValue(DataPointsProperty, value);
        }

        public static readonly DependencyProperty IdealDataPointsProperty =
            DependencyProperty.Register("IdealDataPoints",
            typeof(List<decimal>), typeof(SimpleBurnUpChart),

```

```

        new FrameworkPropertyMetadata(null,
FrameworkPropertyMetadataOptions.AffectsRender));

    public List<decimal> IdealDataPoints
    {
        get => (List<decimal>)GetValue(IdealDataPointsProperty);
        set => SetValue(IdealDataPointsProperty, value);
    }

    public static readonly DependencyProperty ShowForecastLineProperty =
        DependencyProperty.Register("ShowForecastLine", typeof(bool),
typeof(SimpleBurnUpChart),
        new FrameworkPropertyMetadata(true,
FrameworkPropertyMetadataOptions.AffectsRender));

    public bool ShowForecastLine
    {
        get => (bool)GetValue(ShowForecastLineProperty);
        set => SetValue(ShowForecastLineProperty, value);
    }

    public static readonly DependencyProperty BudgetLimitProperty =
        DependencyProperty.Register("BudgetLimit", typeof(decimal),
typeof(SimpleBurnUpChart),
        new FrameworkPropertyMetadata(0m,
FrameworkPropertyMetadataOptions.AffectsRender));

    public decimal BudgetLimit
    {
        get => (decimal)GetValue(BudgetLimitProperty);
        set => SetValue(BudgetLimitProperty, value);
    }

    protected override void OnRender(DrawingContext dc)
    {
        base.OnRender(dc);
        if (BudgetLimit <= 0) return;

        double w = ActualWidth;
        double h = ActualHeight;
        double marginLeft = 50;
        double marginBottom = 30;
        double marginTop = 10;
        double marginRight = 20;

        double chartW = w - marginLeft - marginRight;
        double chartH = h - marginTop - marginBottom;
        if (chartW <= 0 || chartH <= 0) return;

        int totalDays = (IdealDataPoints!= null && IdealDataPoints.Count
> 0)? IdealDataPoints.Count : 30;
        var maxVal = Math.Max(BudgetLimit, (DataPoints?.Count > 0?
DataPoints.Max() : 0)) * 1.1m;
        if (maxVal <= 0) return;

        double GetY(decimal val) => h - marginBottom - (double)(val /
maxVal) * chartH;
        double GetX(int index) => marginLeft + (totalDays > 1?
(double)index / (totalDays - 1) * chartW : 0);

        var gridPen = new Pen(new SolidColorBrush(Color.FromRgb(240, 240,
240)), 1);
        int ySteps = 4;
        for (int i = 0; i <= ySteps; i++)

```

```

        {
            decimal val = (maxVal / ySteps) * i;
            double y = GetY(val);
            dc.DrawLine(gridPen, new Point(marginLeft, y), new Point(w -
marginRight, y));

            var text = new FormattedText($"{val:N0}",
                System.Globalization.CultureInfo.CurrentCulture,
                FlowDirection.LeftToRight,
                new Typeface("Segoe UI"), 10, Brushes.Gray);
            dc.DrawText(text, new Point(marginLeft - text.Width - 5, y -
text.Height / 2));
        }

        dc.DrawLine(new Pen(Brushes.DarkGray, 1.5), new Point(marginLeft,
marginTop), new Point(marginLeft, h - marginBottom));
        dc.DrawLine(new Pen(Brushes.DarkGray, 1.5), new Point(marginLeft,
h - marginBottom), new Point(w - marginRight, h - marginBottom));

        var limitY = GetY(BudgetLimit);
        var limitPen = new Pen(Brushes.Red, 2) { DashStyle =
DashStyles.Dash };
        dc.DrawLine(limitPen, new Point(marginLeft, limitY), new Point(w
- marginRight, limitY));

        if (IdealDataPoints != null && IdealDataPoints.Count > 1)
        {
            var idealPoints = new List<Point>();
            for (int i = 0; i < IdealDataPoints.Count; i++)
            {
                idealPoints.Add(new Point(GetX(i),
GetY(IdealDataPoints[i])));
            }

            var idealPen = new Pen(Brushes.Green, 1.5) { DashStyle =
DashStyles.DashDot };
            var geometry = new StreamGeometry();
            using (var ctx = geometry.Open())
            {
                ctx.BeginFigure(idealPoints, false, false);
                ctx.PolyLineTo(idealPoints.Skip(1).ToList(), true, true);
            }
            dc.DrawGeometry(null, idealPen, geometry);
        }

        if (DataPoints != null && DataPoints.Count > 0)
        {
            var points = new List<Point>();
            for (int i = 0; i < DataPoints.Count; i++)
            {
                points.Add(new Point(GetX(i), GetY(DataPoints[i])));
            }

            if (points.Count > 1)
            {
                var geometry = new StreamGeometry();
                using (var ctx = geometry.Open())
                {
                    ctx.BeginFigure(points, false, false);
                    ctx.PolyLineTo(points.Skip(1).ToList(), true, true);
                }
                dc.DrawGeometry(null, new Pen(Brushes.DodgerBlue, 3),
geometry);
            }
        }
    }
}

```

```

        if (ShowForecastLine && points.Count >= 2 &&
DataPoints.Count < totalDays)
        {
            var lastPoint = points.Last();
            var firstPoint = points;
            double avgSlope = (lastPoint.Y - firstPoint.Y) /
(points.Count - 1);

            var forecastPoints = new List<Point> { lastPoint };

            for (int i = points.Count; i < totalDays; i++)
            {
                double x = GetX(i);
                double y = lastPoint.Y + avgSlope * (i -
(points.Count - 1));

                y = Math.Max(marginTop, Math.Min(h -
marginBottom, y));

                forecastPoints.Add(new Point(x, y));
            }

            var forecastPen = new Pen(Brushes.Orange, 2) {
DashStyle = DashStyles.Dash };
            var fGeom = new StreamGeometry();
            using (var ctx = fGeom.Open())
            {
                ctx.BeginFigure(forecastPoints, false, false);
                ctx.PolyLineTo(forecastPoints.Skip(1).ToList(),
true, true);
            }
            dc.DrawGeometry(null, forecastPen, fGeom);
        }

        var areaGeom = new StreamGeometry();
        using (var ctx = areaGeom.Open())
        {
            ctx.BeginFigure(points, true, true);
            ctx.PolyLineTo(points.Skip(1).ToList(), true, true);
            ctx.LineTo(new Point(points.Last().X, h -
marginBottom), true, true);
            ctx.LineTo(new Point(points.X, h - marginBottom),
true, true);
        }
        dc.DrawGeometry(new SolidColorBrush(Color.FromArgb(40,
30, 144, 255)), null, areaGeom);
    }
}
}
}
}

```

ДОДАТОК В
Слайди презентації

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
Кафедра програмних засобів

ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ДЛЯ ОБЛІКУ ОСОБИСТИХ ФІНАНСІВ

Виконав: ДЕВ'ЯНІН Олексій
Студент 4 курсу, гр. КНТ-132
Спеціальність 121 "ІПЗ"

Керівник: к.т.н., доцент ГОФМАН Євгеній

Запоріжжя, 2026

1

Рисунок В.1 – Титульний слайд

Мета, об'єкт та задачі дослідження

Об'єкт: процес інженерії програмного забезпечення для системи обліку особистих фінансів.

Предмет: інженерні методи, архітектурні підходи та програмні засоби розробки клієнт-серверного застосунку на базі WPF та ASP.NET Core.

Мета роботи: підвищення надійності функціонування системи обліку особистих фінансів та розширення її функціональних можливостей шляхом проектування модулів автоконвертації валют за датою, впровадження інтелектуальної автокатегоризації транзакцій та реалізації адаптивного імпорту банківських даних.

2

Рисунок В.2 – Слайд "Мета, об'єкт та задачі дослідження"

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

Згідно з аналітикою ОЕСР, понад 60% населення стикаються з труднощами через нездатність систематично вести облік. Основні проблеми користувачів:

Фрагментація даних (декілька банків)

Висока трудомісткість ручного введення

Потреба в обліку різних валют (мультивалютність)

Відсутність наочної прогностичної аналітики

Вимога до розробки: створення незалежного інструменту, що поєднує безпечне локальне керування з автоматизованим парсингом банківських даних та динамічним перерахунком валют.

3

Рисунок В.3 – Слайд "Аналіз предметної області"

ПОРІВНЯЛЬНА ХАРАКТЕРИСТИКА РІШЕНЬ

Критерій	GnuCash	HomeBank	ActualBudget	PFMA (Розробка)
Архітектура	Локальна	Локальна	Self-hosted	Клієнт-серверна (WPF/API)
Введення даних	Ручне/OFX	Ручне/CSV	CSV/API	Адаптивний парсинг CSV
Конвертація валют	Ручне введення курсу вручну для кожної операції	Відсутня (лише одна базова валюта)	Часткова (через сторонні плагіни)	Автоматична (динамічний щотижневий курс)
Автокатегоризація	Обмежена	Фільтри	Присутня	Система правил (Keywords)
Захист паролів	Відсутній	Відсутній	Базовий	BCrypt + JWT
Аналітика	Бухгалтерська	Діаграми	Конверти	Графіки Burn-up + Поради

4

Рисунок В.4 – Слайд "Порівняльна характеристика рішень"

ВИБІР ЗАСОБІВ РОЗРОБКИ

Критерій	C# (.NET)	Java	Python
Клієнтська частина	WPF (Native UI)	JavaFX	PyQt
Продуктивність	Висока (JIT)	Висока (JVM)	Середня
Асинхронність	async / await (вбудовано)	CompletableFuture	asyncio
ORM	EF Core	Hibernate	SQLAlchemy

Вибір: .NET 10.0 забезпечує швидку роботу з асинхронними запитами до зовнішніх API курсів валют та надійну структуру для збереження фінансових типів даних (decimal).

5

Рисунок В.5 – Вибір засобів розробки

ІННОВАЦІЙНІ ФУНКЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1. Інтелектуальна автокатегоризація та імпорт:

- Пошук ключових слів (Keywords) в описі транзакцій.
- Механізм *Fallback*: автоматичне віднесення до резервної категорії за відсутності збігів.
- Захист від дублювання за сигнатурою (ID + Сума + Дата).

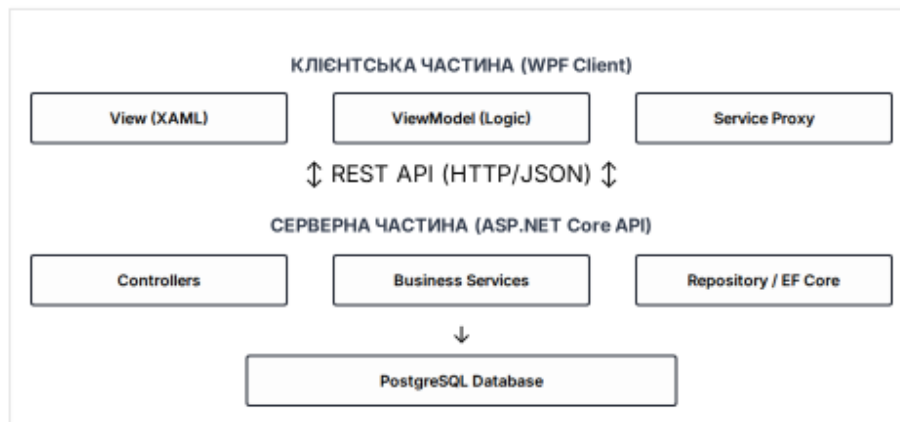
2. Автоматична конвертація валют:

- **Синхронізація:** Автоматичне підтягування актуальних валютних курсів з фіксацією на кожний тиждень.
- **Облік історичних даних:** Транзакції перераховуються за курсом саме того тижня, коли вони відбулися.
- **Ефективність:** Об'єднання рахунків у різних валютах (USD, EUR, UAH) в один аналітичний звіт без ручних розрахунків.

6

Рисунок В.6 – Слайд "Інноваційні функції програмного забезпечення"

СТРУКТУРНА СХЕМА СИСТЕМА



7

Рисунок В.7 – Слайд "Схема структури програми"

МОДУЛЬ МАСОВОГО ІМПОРТУ ДАНИХ

Реалізовано на основі патерну **Strategy**, що дозволяє легко додавати нові банки:



Це забезпечує розпізнавання нестандартних банківських форматів (розділювачі, кодування, формати дат) та автоматичне приведення суми до валюти рахунку.

8

Рисунок В.8 – Слайд "Модуль масового імпорту даних"

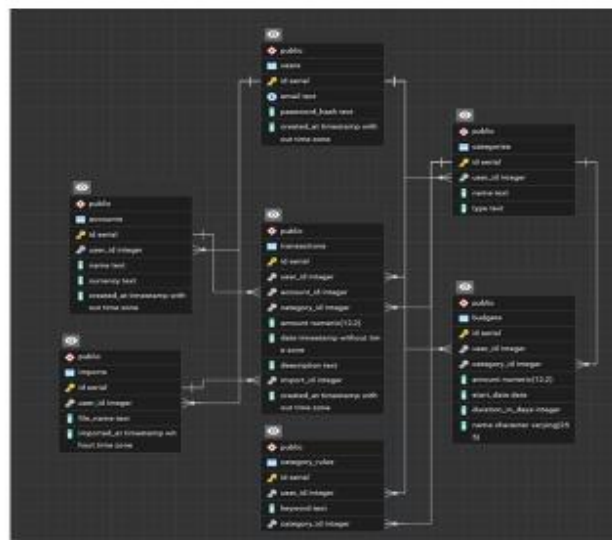
КЛЮЧОВІ ФУНКЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

- **Автентифікація:** Введення облікових даних → Хешування → Отримання JWT.
- **Облік та синхронізація:** Запуск програми → Перевірка та оновлення тижневих курсів валют → Створення/імпорт рахунків.
- **Обробка:** Парсинг → Автокатегоризація → Мультивалютна конвертація → Перевірка дублікатів.
- **Аналітика:** Агрегація витрат в єдину базову валюту → Порівняння з лімітами.
- **Візуалізація:** Побудова Burn-up графіка (DrawingContext).
- **Прийняття рішень:** Аналіз прогнозів перевитрат з урахуванням коливання курсів.

9

Рисунок В.9 – Слайд "Схема функціонування"

СТРУКТУРА ДАНИХ (ER - ДІАГРАММА)



10

Рисунок В.10 – Слайд "Структура даних"



Рисунок В.11 – Слайд "Протокол обміну даними"

ІНТЕРФЕЙС ТА ФУНКЦІОНАЛ

Реалізовані форми:

- Панель керування рахунками (Dashboard) з вибором базової валюти.
- Реєстр транзакцій з відображенням в вигляді таблиці транзакцій.
- Форма масового імпорту банківських виписок.

My Food Budget

Spending Trend: Graph showing actual spending (blue line) and forecast (green line) over time. Budget status: 66.3% Budget Used, 73% Time Passed. AI Insights: On Track! You are likely to finish with a small surplus of 134.75 EUR. Keep your daily spending around 12.46 EUR to reach your goal.

Transactions

Date	Description	Amount	Category	Account	Actions
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details
2024-03-01	WEEKLY SAVING (Savings)	-1.00	Food	Wendbank	Details

Звіти та конвертація:

Завдяки тижневій фіксації курсу користувач бачить консолідовані графіки витрат без хаосу, спричиненого щоденними міжбанківськими коливаннями. Аналітичний звіт відображає чисту динаміку витрат.

Рисунок В.12 – Слайд "Інтерфейс та функціонал"

| ВИСНОВКИ

В результаті виконання роботи було розроблено повнофункціональний клієнт-серверний застосунок «PFMA».

- **Досягнута мета:** Створено систему автоматизованого обліку з повною підтримкою мультивалютності.
- **Інноваційність рішення:** Впровадження алгоритму автоматичної конвертації за щотижневим фіксованим курсом вирішує проблему складного аналізу різновалютних рахунків.
- **Ефективність:** Ручне введення курсів повністю нівельовано, точність аналітики зведена до 100%.
- **Перспективи:** Розширення системи для прогнозування трендів інфляції на основі накопичених історичних курсів.

Рисунок В.13 – Слайд "Висновки"