ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка платформи FamilyBudgeter для спільного планування та контролю сімейних фінансів»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Милана ШРЕМФ
(підпис)

Виконав: здобувачка вищої освіти групи ПД-42

Милана ШРЕМФ

Керівник: Тимур ДОВЖЕНКО
к.т.н.

Рецензент: _____

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Шремф Милані Олександрівні

1. Тема кваліфікаційної роботи: «Розробка платформи FamilyBudgeter для спільного планування та контролю сімейних фінансів»
керівник кваліфікаційної роботи доц. каф. к.т.н.Тимур ДОВЖЕНКО,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.

2. Строк подання кваліфікаційної роботи «02» червня 2025 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи планування сімейного бюджету, огляд сучасних програм-аналогів, опис архітектурних підходів у веб-розробці, технічна документація з реалізації веб-застосунків на базі ASP.NET Core та React, вимоги до функціональності фінансових систем, рекомендації щодо зручності інтерфейсу користувача.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Огляд та аналіз існуючих рішень для обліку та планування сімейного бюджету, визначення недоліків сучасних аналогів.
2. Проєктування веб-платформи FamilyBudgeter з урахуванням ролей користувачів, бюджетних лімітів, цілей, звітності та сповіщень.

3. Програмна реалізація клієнтської та серверної частини застосунку з використанням ASP.NET Core та React.
4. Тестування платформи та аналіз продуктивності.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів.
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації.
4. Діаграма варіантів використання.
5. Алгоритм роботи застосунку.
6. Діаграма класів.
7. Екранні форми.
8. Апробація результатів дослідження

6. Дата видачі завдання«25» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2025	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2025	
3	Огляд існуючих рішень для фінансового планування та архітектурних підходів у веб-розробці	14.03–20.03.2025	
4	Проєктування веб-застосунку FamilyBudgeter: структура бази даних, архітектура, основні компоненти	21.03–01.04.2025	
5	Програмна реалізація клієнтської та серверної частини застосунку	02.04–22.04.2025	
6	Тестування платформи, перевірка продуктивності та функціональності	23.04–28.04.2025	
7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2025	
8	Розробка демонстраційних матеріалів	12.05-18.05.2025	
9	Попередній захист роботи	19.05-30.05.2025	

Здобувачка вищої освіти _____ Милана ШРЕМФ
(підпис)

Керівник
кваліфікаційної роботи _____ Тимур ДОВЖЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 60 стор., 1 табл., 28 рис., 24 джерел.

Мета роботи — покращення цифрового управління спільного планування та контролю сімейного бюджету з можливістю ведення транзакцій, встановлення бюджетів, створення фінансових цілей та формування аналітичних звітів.

Об'єкт дослідження — процес цифрового управління сімейними фінансами в умовах багато користувацького доступу.

Предмет дослідження — веб-застосунок для спільного планування бюджету, обліку доходів і витрат, візуалізації фінансових потоків і досягнення фінансових цілей.

Короткий зміст роботи: у роботі проаналізовано сучасні інструменти та підходи до планування сімейного бюджету, проведено огляд програм-аналогів, таких як Mint, YNAB і Spendee. Запропоновано архітектурну модель системи FamilyBudgeter, яка базується на принципах Clean Architecture та багаторівневої (N-Layer) структури. Реалізовано серверну частину з використанням ASP.NET Core та Entity Framework Core для роботи з базою даних Microsoft SQL Server, клієнтську частину — за допомогою React та TypeScript. Застосунок підтримує створення сімей, додавання учасників з розмежуванням прав доступу, створення бюджетів, транзакцій, фінансових цілей, а також автоматичне генерування регулярних платежів. Додатково реалізовано систему аналітики й візуалізації фінансових показників. Проведено функціональне тестування системи, оцінено її продуктивність і зручність користування. Результати свідчать про ефективність запропонованого рішення в контексті побутового управління фінансами.

Сфера використання розробленої платформи — домашнє планування бюджету, облік витрат родини, досягнення фінансових цілей та підвищення прозорості у веденні спільних витрат.

КЛЮЧОВІ СЛОВА: БЮДЖЕТУВАННЯ, ВЕБ-ПЛАТФОРМА, ASP.NET CORE, REACT, СІМЕЙНІ ФІНАНСИ, ТРАНЗАКЦІЇ, АНАЛІТИКА, CLEAN ARCHITECTURE.

ЗМІСТ

ВСТУП

У сучасних умовах, коли темп життя стрімко зростає, дедалі більше родин стикаються з викликами ефективного фінансового управління: планування бюджету, обліку витрат і доходів, а також досягнення спільних економічних цілей. Зростання інфляції, нестабільна економічна ситуація та надлишок фінансової інформації, що потребує обробки, ускладнюють процес контролю коштів, особливо якщо його потрібно здійснювати колективно. Існуючі методи, як-от ручний облік чи прості електронні таблиці, вже не відповідають сучасним потребам.

Представлена велика кількість фінансових застосунків, переважна більшість з них орієнтована на індивідуальне користування та не враховує потреб родини як колективного фінансового суб'єкта. Відсутність функціоналу для розподілу ролей між членами сім'ї, недостатній рівень локалізації, зокрема української мови, підтримки гривні та звичних форматів звітності, створюють бар'єри для їхнього повноцінного використання в українських реаліях. Це зумовлює актуальність розробки інструменту, який забезпечить зручне, прозоре та адаптоване до локального контексту ведення сімейного бюджету з можливістю колективної участі.

Метою даного дослідження покращення цифрового управління спільного планування та контролю за сімейними фінансами, який буде водночас зручним, функціональним, безпечним та адаптивним до умов українського ринку.

Об'єктом дослідження виступає процес цифрового управління родинним бюджетом.

Предметом є програмна реалізація інструменту для планування, обліку та аналізу сімейних фінансів із можливістю колективного доступу.

Наукова новизна полягає в розробці системи, яка поєднує фінансовий облік, постановку цілей, автоматизацію регулярних витрат та візуалізацію даних, з урахуванням потреб декількох користувачів у межах однієї родини. Такий підхід

дозволяє не лише підвищити ефективність фінансового планування, а й забезпечити кращу взаємодію між членами родини у прийнятті бюджетних рішень.

Практична значимість роботи полягає в тому, що запропонований веб-застосунок може бути безпосередньо впроваджений у повсякденне життя родин, допомагаючи структурувати та контролювати фінансові потоки, підвищувати фінансову грамотність і досягати спільних економічних цілей. Крім того, створена система може стати основою для подальших розробок і бути адаптована до потреб інших цільових аудиторій.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Проблеми планування сімейного бюджету

У сучасному світі управління сімейними фінансами стає дедалі важливішим елементом стабільного та гармонійного життя. Попри наявність різноманітних засобів для обліку доходів і витрат, багато сімей все ще зіштовхуються з труднощами у плануванні бюджету. Однією з основних проблем є відсутність прозорості у фінансових стосунках між членами родини. Нерідко трапляється, що витрати фіксуються лише окремими особами, а інші учасники залишаються неінформованими про реальний стан справ, що призводить до конфліктів, непередбачених боргів або перевищення лімітів витрат.

Ще однією поширеною проблемою є низький рівень фінансової дисципліни. Без системного підходу до контролю бюджету, родини можуть нехтувати регулярним обліком витрат, що ускладнює аналіз фінансових звичок, прогнозування витрат та планування накопичень. Внаслідок цього сім’я часто живе «від зарплати до зарплати», не маючи змоги формувати фінансову подушку безпеки чи досягати довгострокових цілей.

Крім того, сучасні фінансові потреби є досить різноманітними: це не лише базові щомісячні витрати, а й планування великих покупок, відпусток, оплату навчання дітей чи непередбачених медичних послуг. Відсутність централізованого інструменту, що об’єднує всі ці аспекти, ускладнює загальне управління фінансами. Важливу роль також відіграє людський фактор — забуті платежі, відсутність нагадувань чи звітності можуть створити фінансові "дірки", які важко відстежити без відповідного інструменту.

Окремо варто відзначити технічні труднощі. Багато існуючих фінансових застосунків орієнтовані на індивідуальне використання та не передбачають спільного доступу чи розподілу ролей між користувачами. Це значно обмежує можливості сімей для колективного планування, адже зникає можливість

встановлення спільних фінансових цілей, ведення бюджету за категоріями відповідно до потреб кожного члена, або обмеження доступу до певної інформації.

Таким чином, ефективне планування сімейного бюджету потребує інструменту, що забезпечить прозору взаємодію між членами родини, надасть зручні засоби обліку та аналітики, дозволить встановлювати цілі та контролювати витрати, а також враховуватиме рівень фінансової грамотності користувачів. Відсутність такого інструменту стає причиною втрати контролю над фінансами, зменшення довіри між членами сім'ї та неможливості досягати важливих матеріальних цілей.

1.2 Аналіз існуючих рішень

У сучасному цифровому середовищі користувачі мають доступ до десятків застосунків, призначених для обліку особистих фінансів. Вони пропонують базову функціональність для введення доходів і витрат, формування бюджетів, перегляду статистики та іноді — встановлення фінансових цілей. Незважаючи на широкий вибір, більшість таких продуктів орієнтовані на індивідуального користувача й майже не враховують особливості взаємодії між кількома людьми в межах однієї фінансової спільноти — зокрема, родини. Це створює певний вакуум у сфері колективного планування та контролю бюджету, де кожен учасник має різні повноваження, відповідальність і потреби.

Основна проблема полягає в тому, що навіть ті сервіси, які допускають спільне користування — як-от створення спільного рахунку чи гаманця — не пропонують гнучкого розмежування прав доступу або повноцінного управління ролями. Вони не дозволяють, наприклад, батькам бачити повний облік, а дітям — лише власні витрати; або одному з партнерів — налаштовувати цілі, а іншому — лише додавати транзакції. Більше того, в багатьох випадках користувачі змушені ділити єдиний обліковий запис, що суперечить базовим принципам безпеки, прозорості та персоналізації.

Додатковим недоліком є недостатня автоматизація та відсутність адаптованої аналітики. Більшість популярних рішень не мають потужної системи регулярних платежів, не передбачають попереджень про перевищення лімітів, і не дозволяють гнучко налаштовувати фінансові цілі або прогнозувати майбутні витрати на основі історичних даних. Часто інтерфейс є перевантаженим або навпаки — занадто спрощеним, що знижує ефективність у довготривалому використанні.

Нарешті, варто враховувати й локалізаційні бар'єри. Багато рішень, створених на західному ринку, не підтримують українську мову, локальні банки або валюту, що суттєво обмежує їх використання в українських реаліях. У такому контексті стає очевидною потреба у спеціалізованому інструменті, який би поєднував зручність, безпеку, колективну взаємодію та адаптованість під місцевого користувача.

Для глибшого розуміння сильних та слабких сторін конкурентних рішень розглянемо приклади трьох популярних застосунків: Mint, YNAB та Spendee — кожен із яких по-своєму вирішує завдання фінансового планування, проте жоден не надає комплексного підходу до сімейного бюджетування такого рівня, як платформа FamilyBudgeter.

1.2.1 Mint

Mint — це один із найвідоміших сервісів для обліку фінансів, який довгий час утримував популярність серед індивідуальних користувачів. Його головна особливість полягає в автоматичному зборі транзакцій із банківських рахунків, що дозволяє швидко отримувати загальну фінансову картину без ручного введення. Застосунок також дозволяє створювати бюджетні ліміти, переглядати категоризовані витрати та отримувати поради щодо заощадження.

Проте, попри довгу історію та відому назву, Mint має низку принципових обмежень. Передусім, система повністю орієнтована на особисте використання. Вона не має навіть базових інструментів для спільного обліку фінансів у родині. Усі транзакції, бюджети й аналітика зосереджені в межах одного облікового запису без підтримки спільного доступу або розподілу ролей між кількома користувачами.

У ситуації, коли фінансами опікуються кілька людей — наприклад, подружжя або батьки з дітьми — це створює певний дискомфорт і змушує вдаватися до небезпечної практики спільного входу в акаунт.

Крім того, інтерфейс Mint хоча й здається зручним, проте перевантажений зайвою інформацією. Система не дозволяє налаштувати логіку обліку відповідно до специфіки сімейного бюджету — наприклад, вона не враховує потреби окремих учасників, не дає змоги вести бюджети за підсім'ями або групами категорій, а автоматична категоризація витрат часто працює некоректно.

На рисунку 1.1 зображено типовий інтерфейс Mint, в якому видно категорії витрат, автоматично зібрані транзакції та коротку статистику по бюджету.

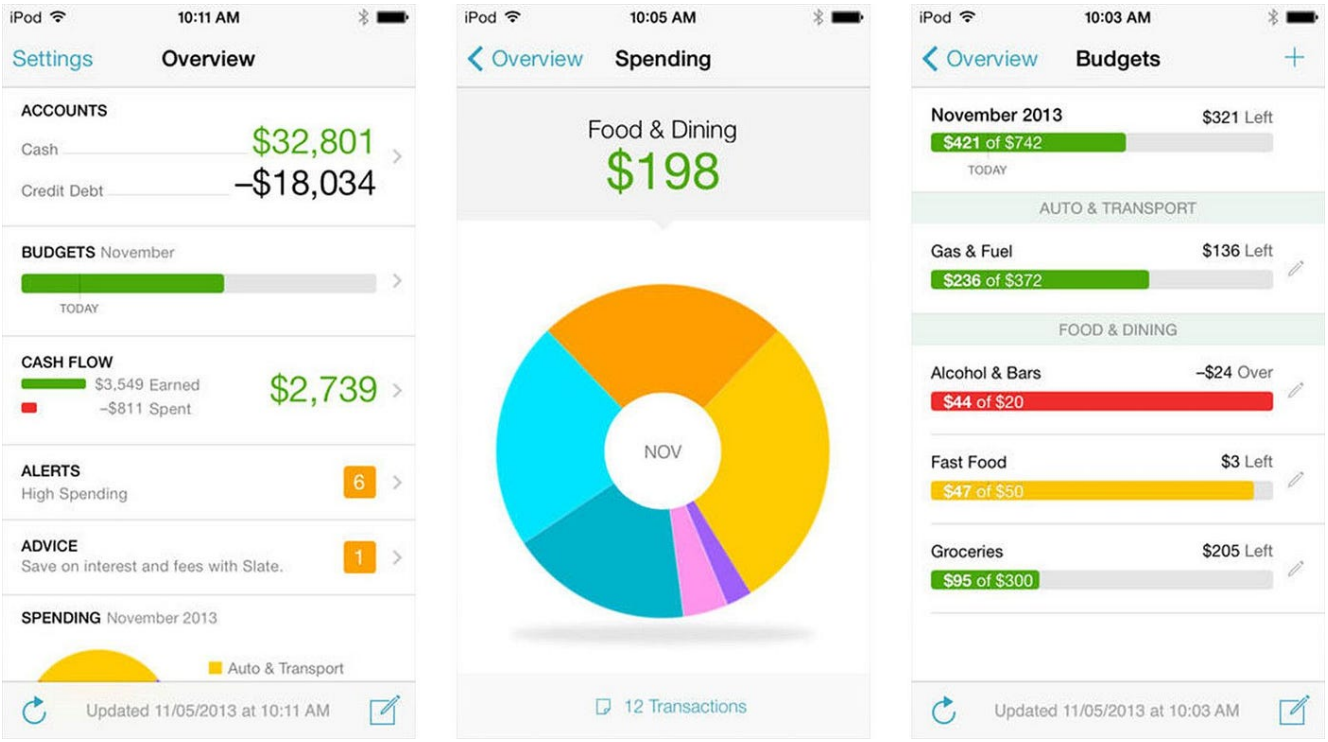


Рис 1.1 Інтерфейс Mint

1.2.2 YNAB

Застосунок YNAB вирізняється серед конкурентів тим, що більше орієнтується не на зручність чи автоматизацію, а на виховання фінансової дисципліни. Його розробники пропонують користувачам дотримуватися власної методики управління грошима, де кожна одиниця доходу має бути розподілена по

категоріях ще до моменту витрати. Це змушує користувача свідомо підходити до планування бюджету та формує нові звички.

Однак такий підхід, хоча й може бути корисним в індивідуальному фінансовому самовихованні, є занадто громіздким для практичного використання в контексті родини або спільної роботи над фінансами. YNAB не підтримує розмежування прав доступу між користувачами, не дозволяє будувати персоніфіковані звіти або обмежувати можливості окремих членів у межах одного бюджету. Спільне використання передбачає лише поділ одного акаунта між кількома людьми, що не забезпечує ані конфіденційності, ані контролю.

Додатковим бар'єром є складність інтерфейсу та специфіка філософії сервісу. Для новачків YNAB може здатися заплутаним, а сам процес ведення бюджету — занадто формалізованим. Такий підхід не є універсальним, і він рідко підходить родинам, які прагнуть не навчання, а простого й ефективного інструменту для спільного контролю витрат.

На рисунку 1.2 представлено фрагмент інтерфейсу YNAB з бюджетною таблицею та прогнозами на основі залишків.

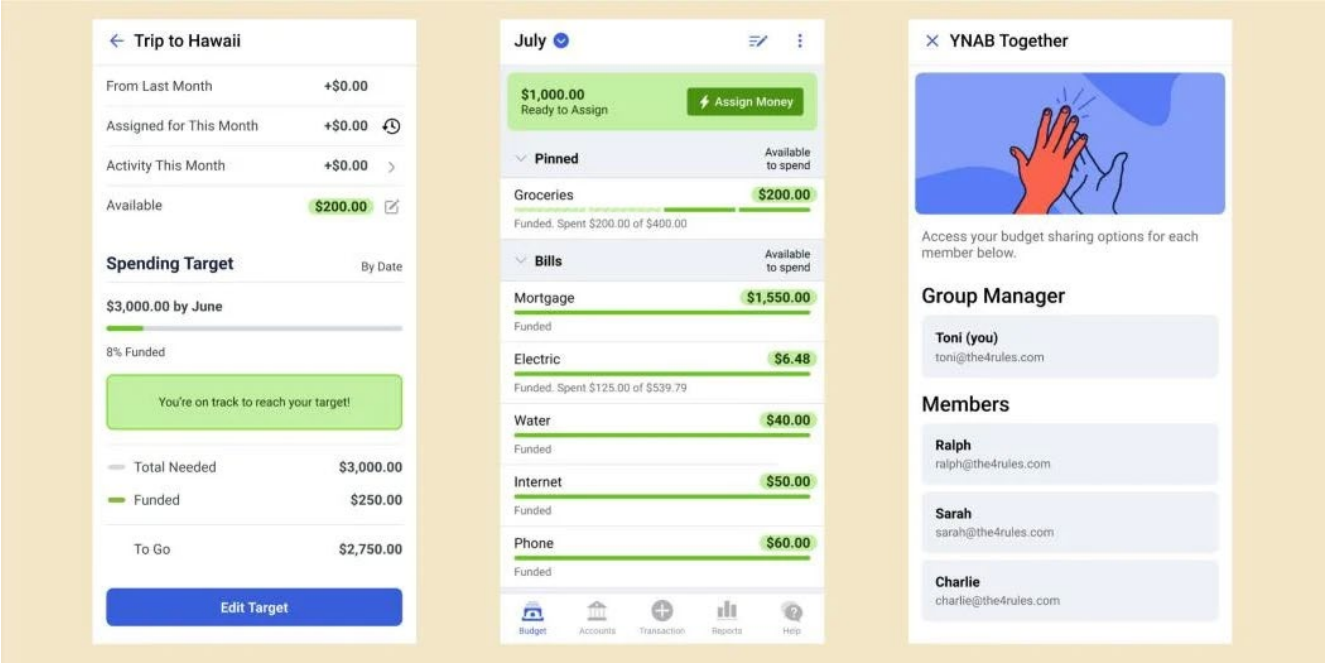


Рис. 1.2 Інтерфейс YNAB

1.2.3 Spendee

Spendee — це застосунок, який принаймні частково намагається врахувати потреби кількох користувачів. Його ключовою особливістю є можливість створення спільних гаманців, до яких можуть підключатися інші учасники, щоб вносити транзакції або переглядати статистику. Така функціональність дійсно ближча до концепції сімейного бюджету, ніж в інших згаданих аналогах.

Проте реалізація цієї можливості виявляється обмеженою та не надто гнучкою. Усі учасники спільного гаманця мають однакові права, що унеможливлює побудову логічної ієрархії доступу. Система не дозволяє контролювати, хто саме має право переглядати всі фінанси, а хто — лише свої. Відсутність ролей, обмежень і сценаріїв використання забороняє повноцінно використовувати Spendee у складних фінансових взаємодіях, як-от ведення бюджету з дітьми, окремими витратами подружжя, тощо.

Окрім цього, функціонал Spendee у безкоштовній версії сильно обмежений: гаманці мають обмеження за кількістю, звіти примітивні, а аналітика зводиться до кількох кругових діаграм. Також в сервісі відсутня повноцінна система фінансових цілей і нагадувань, що знижує його користь як довготривалого інструменту.

На рисунку 1.3 зображено інтерфейс Spendee із візуалізацією витрат за категоріями у межах спільного бюджету.

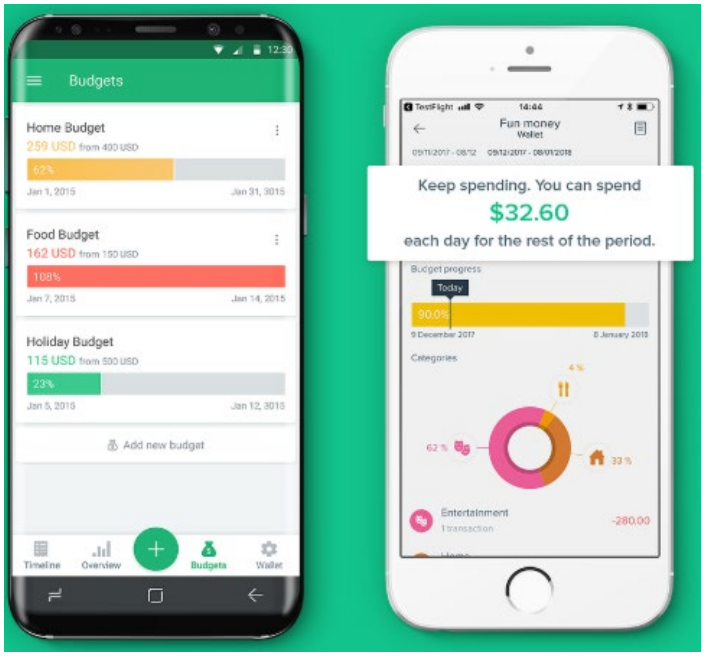


Рис. 1.3 Інтерфейс Spendee

1.3 Порівняння аналогів

Для об'єктивної оцінки якості програмного рішення, розробленого в межах цієї дипломної роботи, було проведено порівняння з найбільш відомими програмами-аналогами — Mint, YNAB та Spendee. Вибір саме цих застосунків зумовлений їхньою широкою популярністю серед користувачів, спрямованістю на особисте або частково спільне планування бюджету, а також доступністю для аналізу.

Щоб порівняння було змістовним і репрезентативним, були визначені основні критерії, які мають вирішальне значення для користувача в контексті сімейного планування та контролю фінансів.

Спільне використання — можливість декільком користувачам вести спільний бюджет із розмежуванням доступу.

Підтримка ролей — наявність функціональності для призначення ролей учасникам (наприклад, адміністратор, гість, дитина тощо).

Фінансові цілі — можливість створювати цілі накопичень або витрат і відстежувати їх виконання.

Регулярні платежі — підтримка повторюваних транзакцій (наприклад, оренда, підписки).

Аналітика та звіти — наявність візуалізації бюджету, графіків, прогнозів та порівнянь.

Автоматизація — здатність системи автоматично створювати записи, сповіщати про перевищення лімітів, пропонувати аналітику.

Гнучкість налаштувань — можливість налаштовувати категорії, бюджети, структуру гаманців під конкретні потреби родини.

Дружній інтерфейс — зручність користування навіть для користувачів без технічних знань.

Мовна та валютна адаптація — підтримка української мови та гривні.

Усі ці параметри мають пряме відношення до зручності, функціональності та адаптивності застосунку у реальних побутових умовах.

В таблиці 1.1 наведене порівняння існуючих програм-аналогів із платформою FamilyBudgeter.

Таблиця 1.1

Порівняння існуючих програм-аналогів із платформою FamilyBudgeter

Критерій	Mint	YNAB	Spendee	FamilyBudgeter
Спільне використання	-	-	+	+
Підтримка ролей	-	-	-	+
Фінансові цілі	+	+	+	+
Регулярні платежі	-	-	+	+
Аналітика	+	+	-	+
Автоматизація	-	-	-	+
Гнучкість налаштувань	-	+	-	+
Мовна та валютна адаптація	-	-	-	+

Як видно з таблиці, жоден з аналізованих сервісів не охоплює всі ключові аспекти, необхідні для повноцінного сімейного фінансового планування. Наприклад, Mint та YNAB зосереджені на індивідуальному обліку, що робить їх менш придатними для родинного використання. Spendee робить крок у бік спільного бюджету, але реалізація ролей, безпеки й аналітики залишається на базовому рівні.

На цьому тлі FamilyBudgeter вигідно вирізняється цілісністю підходу: кожен із визначених критеріїв було враховано ще на етапі проектування, що дозволило створити систему, максимально адаптовану до реального користувача, з прозорою структурою, гнучкими налаштуваннями, високим рівнем автоматизації та сучасним інтерфейсом.

1.4 Вимоги до програмного забезпечення

На основі аналізу предметної області, огляду аналогів та визначених функціональних цілей системи було сформовано перелік вимог до програмного забезпечення, необхідних для реалізації платформи FamilyBudgeter. Ці вимоги охоплюють як функціональні, так і нефункціональні аспекти, які є критично важливими для забезпечення якості, зручності та ефективності роботи системи в реальних умовах використання.

Функціональні вимоги формують основу поведінки системи. Програма повинна забезпечувати можливість:

- реєстрації користувача та створення сім'ї з подальшим запрошенням учасників через унікальний код;
- призначення ролей членам сім'ї (адміністратор, повний учасник, обмежений учасник) з відповідними правами доступу;
- додавання доходів і витрат, редагування та фільтрації транзакцій за категоріями, датами та типами;
- встановлення бюджетних лімітів для категорій витрат, з можливістю контролю їх перевищення;
- формування спільних фінансових цілей та відстеження прогресу їх досягнення;
- налаштування регулярних (повторюваних) платежів за заданим інтервалом часу;
- автоматичної генерації транзакцій на основі налаштувань регулярних витрат;
- побудови графіків і звітів про доходи, витрати, дотримання бюджету, досягнення цілей тощо;
- отримання сповіщень про важливі події: наближення до ліміту, пропущені платежі, досягнуті цілі.

Система FamilyBudgeter повинна не лише реалізовувати необхідну функціональність, але й відповідати сучасним вимогам до якості, масштабованості, надійності й зручності. Нижче наведено ключові нефункціональні вимоги, які були враховані під час розробки:

Зрозумілий та адаптивний інтерфейс. Інтерфейс користувача повинен бути інтуїтивно зрозумілим навіть для користувачів без технічної підготовки. Для цього застосовано компонентний підхід на React із підтримкою адаптивної верстки. Інтерфейс коректно масштабується на смартфонах, планшетах та десктопах, забезпечуючи повноцінну взаємодію незалежно від типу пристрою. Дизайн орієнтований на мінімалізм, візуальну легкість та логічне групування елементів.

Безпека та контроль доступу. Аутентифікація користувачів реалізована на основі JWT токенів, що дозволяє захищати API-запити й уникати несанкціонованого доступу. Доступ до функцій системи розмежовується відповідно до ролей: адміністратор має повний контроль над фінансами й користувачами, інші ролі мають обмеження на додавання, редагування або перегляд інформації. Крім того, усі критичні дії — як-от видалення транзакцій або зміна бюджету — супроводжуються перевіркою прав користувача.

Масштабованість. Архітектура побудована за принципами Clean Architecture і багаторівневої моделі (N-Layer), що забезпечує легкість розширення системи. Наприклад, додавання нового типу фінансових операцій, модуля заощаджень чи інтеграції з банківськими API може бути реалізоване без зміни базової логіки завдяки чіткому розділенню обов’язків між рівнями API, бізнес-логіки, доступу до даних та доменних моделей.

Надійність і стійкість до помилок. Усі запити до API супроводжуються обробкою винятків, валідацією вхідних даних (через FluentValidation) та логуванням помилок. У разі відмови окремих сервісів або некоректних дій користувача система не завершує роботу, а повертає інформативні повідомлення про помилку. Клієнтська частина також має механізми обробки помилок мережі та недоступності сервера з відповідними повідомленнями для користувача.

Продуктивність. Час обробки основних запитів (отримання транзакцій, формування звітів, оновлення бюджету) не повинен перевищувати 300–500 мс при звичайному навантаженні. Для оптимізації продуктивності використовуються ефективні запити до бази даних (LINQ з проекціями DTO), кешування конфігураційних даних і мінімізація обміну надлишковими даними між клієнтом і сервером. На фронтенді використовується Vite як швидкий інструмент збирання проєкту.

Підтримка локалізації та валют. Система спочатку орієнтована на українського користувача: весь інтерфейс локалізований українською мовою, усі фінансові операції виконуються в гривні (UAH), а формати дат і чисел адаптовані до українських стандартів. У майбутньому можлива підтримка мультивалютності та багатомовності без переробки основного коду завдяки відповідним структурам у моделі та фронтенді.

Якість коду та підтримуваність. Уся система побудована з дотриманням принципів SOLID, що спрощує тестування, підтримку та розвиток проєкту. Базова бізнес-логіка ізольована від інфраструктурних деталей, що дозволяє замінювати окремі компоненти (наприклад, джерело даних або метод аутентифікації) без порушення загальної цілісності системи. Кожен сервіс виконує чітко визначену функцію, що відповідає принципу єдиної відповідальності.

Таким чином, нефункціональні вимоги системи не є другорядними — вони визначають стабільність, ефективність та зручність взаємодії користувача із системою FamilyBudgeter, забезпечуючи її придатність до повсякденного використання родиною з будь-яким рівнем технічної підготовки.

2 ПРОЄКТУВАННЯ СИСТЕМИ FAMILYBUDGETER

2.1 Загальний опис архітектури системи

Система FamilyBudgeter спроєктована з урахуванням сучасних вимог до масштабованості, модульності, розділення відповідальностей та підтримуваності коду. Архітектура платформи базується на принципах чистої архітектури (Clean Architecture), а також використовує багаторівневу модель (N-Layer Architecture), яка дозволяє чітко відокремити різні аспекти системи — від представлення до бізнес-логіки та роботи з даними.

Архітектурна модель умовно розділена на два основні рівні: серверна частина (backend) та клієнтська частина (frontend). Кожен з них виконує свою роль у загальній роботі платформи.

Серверна частина реалізована на основі ASP.NET Core 8. Основні шари архітектури:

- API Layer — відповідає за взаємодію з клієнтською частиною. Тут реалізовані REST-контролери, які приймають запити, виконують валідацію даних і делегують обробку бізнес-логіці;
- Business Logic Layer (BLL) — шар бізнес-логіки, який містить сервіси, що реалізують правила роботи програми. Саме тут відбувається обробка транзакцій, бюджетів, цілей, перевірка прав доступу та інші важливі операції;
- Data Access Layer (DAL) — містить реалізації репозиторіїв та шаблон Unit of Work для ефективної та контрольованої роботи з базою даних;
- Domain Layer — включає доменні моделі та ентити, які є основою для всієї логіки програми. Цей шар не залежить від жодної зовнішньої інфраструктури та зосереджує увагу виключно на бізнес-сутностях, таких як транзакція, бюджет, ціль тощо.

Такий підхід забезпечує гнучкість системи, адже дозволяє змінювати інтерфейс, базу даних або метод автентифікації без зміни основної логіки.

Клієнтська частина побудована з використанням React 18 і TypeScript. Архітектура реалізована за принципами компонентної моделі, що дозволяє повторно використовувати окремі частини інтерфейсу.

Основні особливості:

— управління станом реалізовано через React Context API у поєднанні з кастомними хуками, що забезпечує централізовану обробку сесії, ролей, повідомлень та глобального стану;

— взаємодія з сервером виконується через Axios з інтерцепторами, які обробляють токени, повідомлення про помилки та відповіді від API;

— маршрутизація реалізована за допомогою React Router v6, що дозволяє легко розділяти доступ за ролями, створювати захищені маршрути та забезпечувати гнучку навігацію;

— адаптивність інтерфейсу реалізовано за допомогою CSS Modules та CSS Variables, що дозволяє зручно налаштовувати стилі й забезпечити кросплатформеність.

Клієнтська частина логічно розділена на сторінки, компоненти, сервіси API, утиліти та типи, що відповідає принципам зручної масштабованої розробки.

Взаємодія між фронтендом і бекендом відбувається за допомогою REST API через HTTPS. Запити надсилаються у форматі JSON. Кожен запит проходить перевірку аутентифікації та авторизації. Дані передаються структуровано — через DTO-об'єкти, що ізолює внутрішню реалізацію від зовнішніх клієнтів.

Для зберігання даних використовується Microsoft SQL Server, що забезпечує надійність, транзакційність і можливість масштабування. Робота з базою виконується за допомогою Entity Framework Core, що дозволяє працювати з даними через об’єктно-реляційне відображення (ORM), не порушуючи принципи чистої архітектури.

2.2 Вибір технологій розробки

Вибір технологій для реалізації системи FamilyBudgeter здійснювався на основі детального аналізу як функціональних, так і нефункціональних вимог до системи. Одним із ключових пріоритетів під час ухвалення рішень було забезпечення масштабованості архітектури, тобто здатності системи легко розширюватися у майбутньому, наприклад, додаванням нових модулів (управління боргами, інтеграції з банками тощо) без потреби кардинальних змін у вже реалізованій структурі. Також важливою була продуктивність — усі обрані інструменти повинні забезпечувати мінімальні затримки у відповіді на користувацькі запити, що критично важливо для зручного та швидкого щоденного використання.

Ще одним вагомим фактором була стабільність технологій. Застарілі або маловідомі фреймворки не розглядалися, адже вони часто мають обмежену підтримку та невелику спільноту. Натомість було обрано перевірені рішення, які активно розвиваються, мають велику базу знань і підтримуються компаніями-лідерами на ринку. Це дозволяє не лише прискорити процес розробки, але й мінімізувати ризики, пов’язані з безпекою, помилками або відсутністю критичної документації.

Особливу увагу також приділено адаптації до локального (українського) ринку. Враховано необхідність підтримки української мови, гривні як основної валюти, звичних форматів дат та чисел, а також потенційних регуляторних вимог. Вибрані технології дозволяють реалізувати повну локалізацію інтерфейсу, зберігаючи при цьому можливість легко додати додаткові мови в майбутньому.

Оскільки FamilyBudgeter є веб-платформою, її архітектура поділяється на два незалежних, але взаємопов’язаних середовища:

— серверна частина (backend) відповідає за обробку бізнес-логіки, взаємодію з базою даних, реалізацію API для обміну даними з клієнтами, а також аутентифікацію й авторизацію користувачів. Це ядро всієї системи, що забезпечує цілісність, безпеку й логіку всіх дій;

— клієнтська частина (frontend) відповідає за взаємодію з користувачем через браузер, візуалізацію даних, обробку подій і надсилання запитів до API. Саме цей рівень формує користувацький досвід, тому до нього висувалися високі вимоги щодо зручності, адаптивності та швидкодії.

Кожен із цих рівнів реалізовано за допомогою сучасного інструментарію, що максимально відповідає поставленим цілям системи. Подальші підрозділи детально описують вибрані технології для кожної частини та обґрунтовують їх доцільність у контексті реалізації FamilyBudgeter.

2.2.1 Серверна частина: ASP.NET Core 8

Для реалізації серверної логіки системи було обрано ASP.NET Core 8 — сучасний, кросплатформений фреймворк від компанії Microsoft, призначений для побудови високопродуктивних веб-додатків і API. Його вибір зумовлений кількома важливими перевагами. Насамперед, ASP.NET Core забезпечує високу продуктивність при обробці запитів, що є критичним для системи з потенційно великою кількістю користувачів і транзакцій. Крім того, фреймворк підтримує впровадження розділення відповідальностей завдяки зручній реалізації шаблонів залежностей (Dependency Injection), middleware, атрибутів контролю доступу, фільтрів та ін.

ASP.NET Core активно підтримує архітектуру з багатьма рівнями (N-Layer) та легко інтегрується з базами даних, зовнішніми сервісами й бібліотеками. Важливою перевагою також є наявність активної спільноти та великої кількості прикладів і документації, що спрощує розробку, тестування та масштабування застосунку.

Для FamilyBudgeter дана технологія стала основою серверної частини, оскільки дозволяє створювати чисту, підтримувану та безпечну бізнес-логіку, організовану у вигляді REST API, яким взаємодіє клієнтська частина.

2.2.2 Доступ до даних: Entity Framework Core

Як засіб роботи з базою даних обрано Entity Framework Core (EF Core) — об'єктно-реляційний мапер (ORM), що дозволяє працювати з таблицями бази даних через об'єкти мови C# без необхідності безпосередньо писати SQL-запити. EF Core підтримує автоматичну генерацію міграцій, що значно спрощує еволюцію структури бази даних у процесі розробки. Також він дозволяє реалізовувати складні зв’язки між сутностями, фільтрацію, сортування, lazy та eager loading, що є особливо корисним для роботи з великою кількістю фінансових операцій.

EF Core відмінно інтегрується з ASP.NET Core, має гнучкий механізм конфігурації моделей (через Fluent API) і повністю підтримує шаблон Repository + Unit of Work, який застосовується у FamilyBudgeter. Завдяки цьому реалізується ізоляція бізнес-логіки від конкретної реалізації зберігання даних, що відповідає принципам Clean Architecture.

2.2.3 База даних: Microsoft SQL Server

У ролі основної системи управління базами даних використовується Microsoft SQL Server. Це потужна, стабільна та надійна СУБД, яка підтримує транзакційність, складні запити, тригери, індексацію та механізми збережених процедур. SQL Server добре масштабується та ефективно працює навіть із великими обсягами фінансових даних, що важливо для довготривалого зберігання історії транзакцій, бюджетів, цілей і сповіщень.

Вибір саме цього рішення пояснюється також повною сумісністю з інструментами Microsoft та Entity Framework Core, що забезпечує стабільну інтеграцію, спрощене адміністрування і можливість гнучкої оптимізації запитів при потребі.

2.2.4 Клієнтська частина: React 18 + TypeScript

Для побудови клієнтської частини було обрано React — одну з найпопулярніших бібліотек для створення інтерфейсів користувача. Остання версія React 18 забезпечує ще вищу продуктивність завдяки оптимізованому рендерингу

та новим можливостям роботи з асинхронними даними. У поєднанні з TypeScript — мовою програмування, яка розширює JavaScript статичною типізацією, — забезпечується не лише зручність, але й висока надійність клієнтського коду, зменшення кількості помилок та підвищення підтримуваності.

React дозволяє створювати адаптивний, багаторазово використовуваний інтерфейс на основі компонетної архітектури. Це відповідає вимогам до фронтенду системи FamilyBudgeter, де важливо реалізувати повторно використовувані компоненти для форм, списків, діаграм, повідомлень тощо. Додатково, React Context API та кастомні хуки дозволяють централізовано управляти станом додатку (сесія, сповіщення, доступ) без необхідності використовувати громіздкі бібліотеки для керування станом.

2.2.5 Взаємодія з сервером: Axios

Для надсилання HTTP-запитів із клієнтської частини використовується Axios — потужний, простий і гнучкий HTTP-клієнт. Його переваги включають підтримку інтерцепторів для централізованої обробки токенів (JWT), обробку глобальних помилок, автоматичну серіалізацію/десеріалізацію JSON-даних, таймаути, та повтори запитів. Axios дозволяє розділити логіку роботи з API на модулі, що підвищує чистоту та підтримуваність коду.

2.2.6 Системи стилізації: Bootstrap 5

Для стилізації інтерфейсу користувача в системі FamilyBudgeter було обрано Bootstrap 5 — один із найпопулярніших CSS-фреймворків для побудови адаптивних і кросбраузерних інтерфейсів. Основною причиною такого вибору є швидкість розробки та уніфікований підхід до побудови інтерфейсу, що особливо важливо у проєкті з великою кількістю однотипних форм, таблиць і модальних вікон.

Bootstrap забезпечує велику кількість готових компонентів: кнопки, навігаційні панелі, сповіщення, вкладки, таблиці, інпут-групи тощо. Це дозволяє суттєво скоротити час розробки, зосередившись на бізнес-логіці та

функціональності замість створення стилів «з нуля». Завдяки системі сітки (Grid System) та класам для адаптивності, інтерфейс коректно відображається на різних пристроях — від мобільних телефонів до великих моніторів.

Крім того, Bootstrap підтримує кастомізацію за допомогою змінних, що дозволяє швидко змінювати кольорову схему, розміри шрифтів або інші параметри для адаптації до візуального стилю бренду. У FamilyBudgeter це використано для створення чистого, мінімалістичного дизайну з приємною кольоровою палітрою.

Завдяки широкій підтримці, великій документації та активній спільноті Bootstrap є надійним рішенням для розробки масштабованих інтерфейсів, особливо у проєктах із обмеженим часом на створення унікального дизайну з нуля.

2.3 Архітектурна модель

Під час розробки системи FamilyBudgeter було прийнято рішення дотримуватися принципів чистої архітектури (Clean Architecture) у поєднанні з багаторівневою структурою (N-Layer Architecture). Такий підхід дозволяє досягти високого рівня ізоляції між компонентами, гнучкості, простоти тестування та легкості масштабування системи в майбутньому. Рівні архітектури чітко розділяють відповідальність, що дає змогу змінювати або розширювати одну частину системи, не порушуючи цілісність усієї структури.

Уся архітектура системи умовно поділена на два великі блоки: бекенд (серверна частина) та фронтенд (клієнтська частина). Кожен із них реалізує свій рівень відповідальності згідно з принципами розділення логіки, даних і представлення.

Серверна частина реалізована за допомогою ASP.NET Core 8 і поділена на чотири основні рівні:

— API Layer. Відповідає за зовнішню взаємодію з клієнтською частиною. Тут реалізовані REST-контролери, які приймають HTTP-запити, виконують базову перевірку даних (валидацію DTO) та передають їх на обробку у бізнес-логіку. Контролери не містять логіки, що не відповідає їхньому призначенню;

— Business Logic Layer (BLL). Містить сервіси, які реалізують правила роботи додатку — обробку транзакцій, бюджетів, розрахунки лімітів, фільтрацію даних тощо. Цей шар ізольований від інфраструктури та повністю концентрується на логіці, що відповідає вимогам до функціональності системи;

— Data Access Layer (DAL). Реалізує взаємодію з базою даних за допомогою шаблонів Repository та Unit of Work. Цей шар абстрагує доступ до таблиць БД, дозволяє ефективно працювати з великими обсягами даних, та забезпечує гнучкість при зміні джерела даних у майбутньому;

— Domain Layer. Містить доменні моделі (сутності), які описують основні об’єкти предметної області: транзакцію, бюджет, категорію, фінансову ціль, користувача тощо. Цей рівень повністю незалежний від решти системи, не залежить від бази даних або UI, і є основою для всієї бізнес-логіки.

Клієнтська частина реалізована на основі React 18 із використанням TypeScript. Вона побудована на компонентній архітектурі з логічним розділенням структури на сторінки, модулі API, утиліти та окремі функціональні блоки. Дані надсилаються та отримуються з API у вигляді DTO-об’єктів.

Використовується React Context API для централізованого зберігання глобального стану (сесія, ролі, повідомлення), а також кастомні хуки для повторного використання логіки. Axios із інтерцепторами дозволяє обробляти автентифікацію, помилки та відповіді від серверної частини централізовано. Bootstrap використовується для швидкої побудови адаптивного інтерфейсу.

На рисунку 2.1 наведено узагальнену схему архітектури системи FamilyBudgeter, що демонструє взаємозв’язки між основними рівнями бекенду та клієнтською частиною:

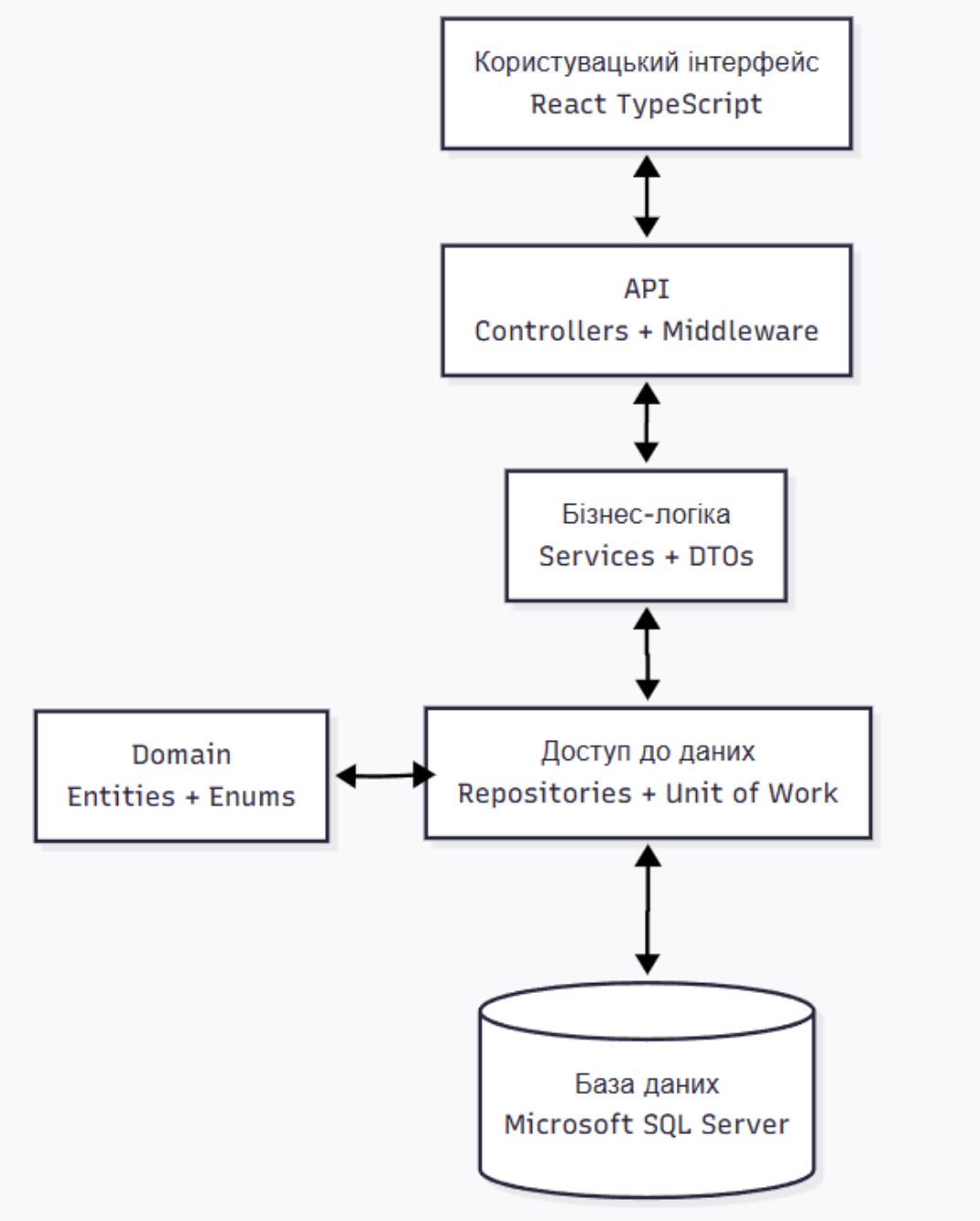


Рис. 2.1 Узагальнену схему архітектури системи

Архітектурна модель дозволяє підтримувати чітку структуру, уникати дублювання логіки, забезпечувати повторне використання коду та значно спрощує тестування, налагодження й подальше розширення системи. У перспективі це створює умови для довготривалої підтримки платформи та її поступового розвитку відповідно до потреб користувачів.

2.4 Структура бази даних

База даних є критично важливим компонентом інформаційної системи FamilyBudgeter, оскільки вона забезпечує не лише фізичне зберігання даних, але й підтримує логіку взаємозв’язків між сутностями, дозволяє контролювати цілісність інформації, обробляти запити до великого обсягу фінансових операцій і забезпечувати доступ до інформації у реальному часі. Враховуючи характер платформи, яка орієнтована на довготривале ведення бюджету, накопичення фінансової історії, спільну взаємодію кількох користувачів, до проєктування бази даних було поставлено підвищені вимоги.

Для реалізації зберігання даних використовується реляційна система управління базами даних Microsoft SQL Server, яка є однією з найнадійніших і найпоширеніших у корпоративному середовищі. Її вибір був обґрунтований такими факторами, як підтримка складної реляційної моделі, наявність розширених можливостей для транзакційної обробки, масштабованість, підтримка безпеки, а також чудова інтеграція з екосистемою .NET та ORM-бібліотекою Entity Framework Core.

SQL Server дозволяє ефективно реалізовувати зв’язки «один до багатьох», які є типовими для зв’язку між сім’єю та її учасниками, транзакціями й категоріями, а також «багато до багатьох», що застосовуються у випадках, коли користувачі можуть належати до кількох сімей. Важливою перевагою є підтримка індексів, тригерів і обмежень цілісності, що дозволяє підвищити швидкодію системи, забезпечити достовірність та узгодженість даних навіть при великому навантаженні.

Уся структура бази даних побудована так, щоб у перспективі можливо було реалізувати аналітичні запити, звітність, агрегацію даних, а також підтримку історії змін (наприклад, для аудиту бюджету або відстеження динаміки витрат за певний період). Модель також оптимізована під типові сценарії використання: швидкий

доступ до транзакцій за обраний період, фільтрація по категоріях, побудова статистики для звітів тощо.

Таким чином, структура бази даних FamilyBudgeter — це не просто набір таблиць для зберігання чисел, а логічно продумана, масштабована основа всієї системи, яка дозволяє ефективно обслуговувати як окремого користувача, так і цілу родину в довготривалій перспективі.Основні сутності та їх призначення:

- user – зберігає інформацію про користувачів системи: ім’я, електронну пошту, хеш пароля, роль у сім’ї. Кожен користувач може бути учасником однієї або кількох сімей.
- family – основна організаційна одиниця. Містить загальні налаштування сім’ї, назву, код для запрошення, список учасників;
- userFamily – проміжна таблиця для реалізації зв’язку «багато до багатьох» між користувачами та сім’ями, зберігає також роль користувача в межах конкретної сім’ї;
- transaction – зберігає всі фінансові операції: тип (дохід/витрата), суму, дату, опис, прикріплений чек (фото) та зв’язок із категорією;
- financeCategory – дозволяє структурувати доходи й витрати. Категорії можуть бути загальними або унікальними для кожної сім’ї. Для кожної категорії можна встановити бюджетний ліміт;
- budget – описує бюджети різних типів (місячний, річний, спеціальний), з можливістю прив’язки до сім’ї та категорій;
- goal – представляє фінансову ціль, яку ставить собі родина або її окремий учасник. Містить назву, бажану суму, прогрес накопичень і статус;
- recurringPayment – зберігає дані про регулярні платежі, включаючи періодичність (щомісяця, щотижня тощо), дату наступного списання, суму, категорію та опис;
- notification – таблиця для зберігання сповіщень, які система надсилає користувачам: перевищення ліміту, нагадування про регулярний платіж, досягнення цілі тощо.

Для візуального представлення взаємозв’язків між таблицями база даних моделюється у вигляді ER-діаграми. На ній показано ключові сутності, типи зв’язків (один-до-багатьох, багато-до-багатьох) і зовнішні ключі.

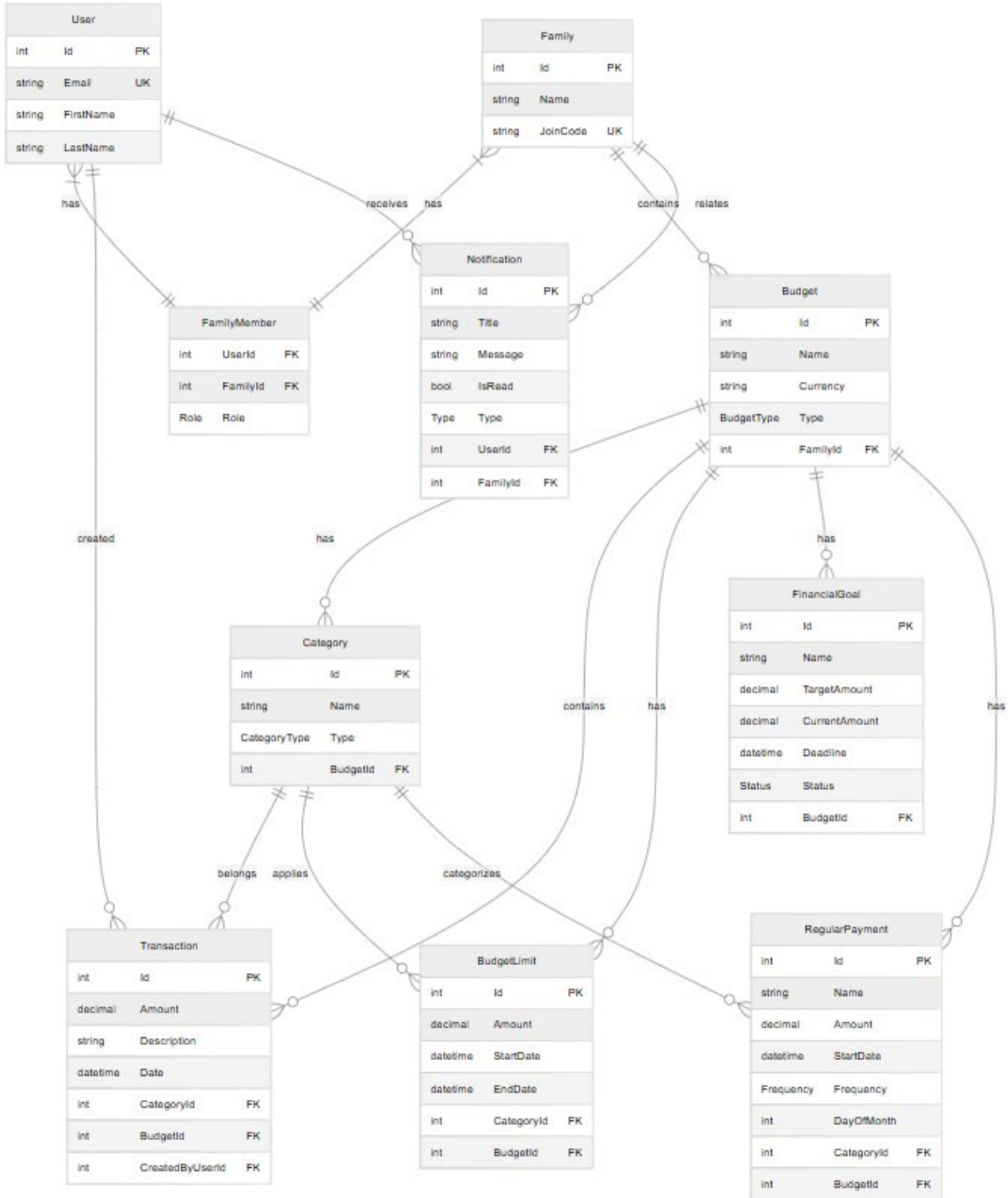


Рис. 2.2 Діаграма структури бази даних системи FamilyBudgeter

2.5 Візуальне моделювання системи

Для кращого розуміння структури та функціонування платформи FamilyBudgeter, на етапі проєктування було створено низку візуальних моделей, які демонструють поведінку системи, її взаємодію з користувачами та логіку взаємодії між окремими технічними компонентами. Такі моделі сприяють узагальненому баченню архітектури, дозволяють виявити приховані залежності та підвищують ефективність подальшої реалізації, тестування й супроводу системи.

2.5.1 Use Case-діаграма

Use Case-діаграма відображає основні сценарії взаємодії користувачів із системою. Вона дозволяє візуалізувати функціональні можливості застосунку з позиції кінцевого користувача, визначити ролі, а також описати доступні дії кожного типу користувачів. У системі FamilyBudgeter визначено кілька ключових ролей:

- адміністратор сім’ї — має повний доступ до всіх функцій, включаючи налаштування сім’ї, додавання/видалення учасників, перегляд усіх транзакцій, створення бюджетів і цілей.
- учасник із повними правами — може створювати й переглядати власні транзакції, брати участь у досягненні спільних цілей, отримувати сповіщення, але не має доступу до керування сім’єю.
- обмежений учасник — має мінімальний доступ: наприклад, лише перегляд звітів або додавання окремих витрат.

На діаграмі показано, які функції доступні кожному типу користувача, включаючи сценарії: реєстрація, приєднання до сім’ї, створення транзакцій, встановлення лімітів, перегляд аналітики, отримання сповіщень тощо.

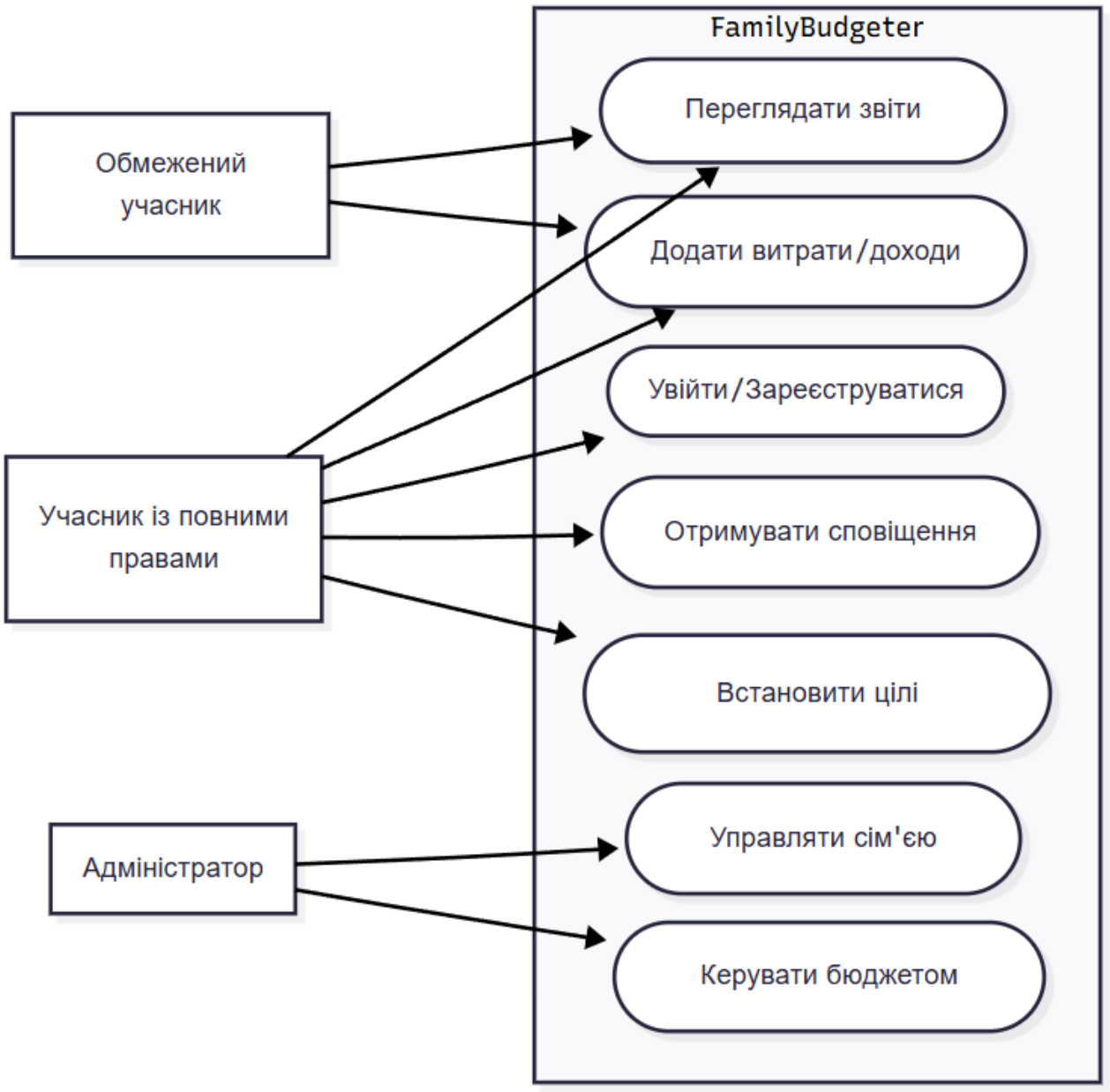


Рис. 2.3 Основні сценарії взаємодії користувачів

2.5.2 Діаграма взаємодії компонентів

Для демонстрації взаємозв’язків між технічними частинами системи було створено діаграму взаємодії компонентів. Вона описує, як саме між собою взаємодіють фронтенд, API-контролери, сервіси бізнес-логіки, репозиторії та база даних під час обробки типового запиту.

У прикладі обробки запиту на створення транзакції послідовність виглядає так:

— користувач взаємодіє з інтерфейсом React (введення суми, опису, категорії);

— axios надсилає POST-запит до відповідного REST API-методу;

— API-контролер приймає запит, виконує валідацію даних і викликає відповідний сервіс у BLL;

— сервіс у BLL обробляє логіку (перевірка прав доступу, категорії, бюджету) та передає запит до DAL;

— репозиторій у DAL взаємодіє з базою даних через EF Core;

— результат повертається через усі рівні назад до клієнта.

На рисунку 2.4 зображена діаграма взаємодії компонетнів.

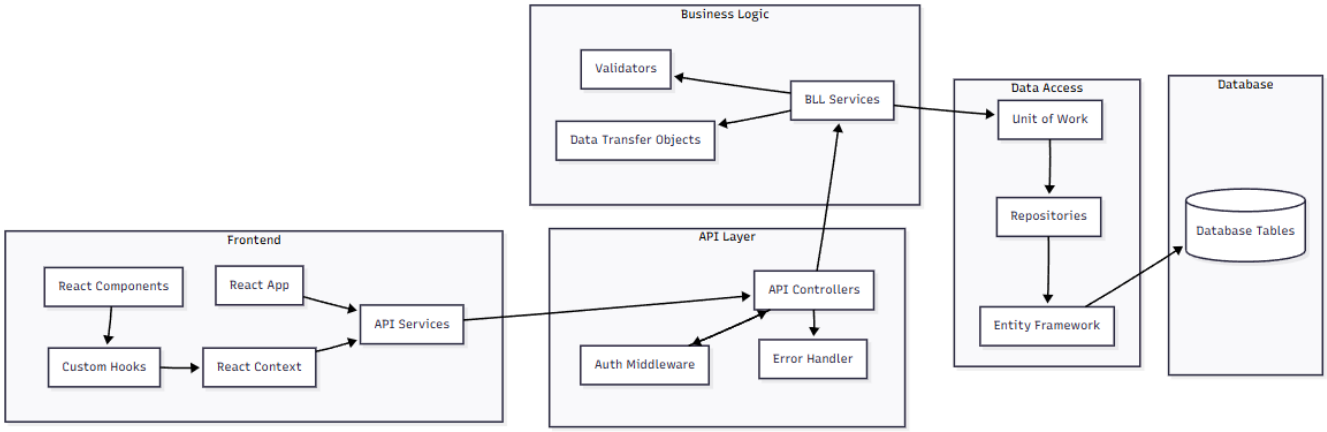


Рис. 2.4 Діаграма взаємодії компонетнів

Ця модель демонструє застосування принципів розділення відповідальностей, ізоляцію бізнес-логіки від інтерфейсу та даних, а також підтверджує відповідність реалізації вимогам Clean Architecture.

2.5.3 Діаграма потоку даних

Діаграма потоку даних дає змогу візуалізувати, як саме дані переміщуються в межах системи FamilyBudgeter. Вона відображає джерела/приймачі даних (акторів), основні процеси системи, а також сховища даних, з якими ці процеси взаємодіють. На відміну від Use Case-діаграми, яка зосереджена на функціональній

взаємодії користувачів із системою, DFD дозволяє простежити шлях даних — від введення до обробки та збереження.

На рисунку 2.5 зображена діаграма потоку даних.

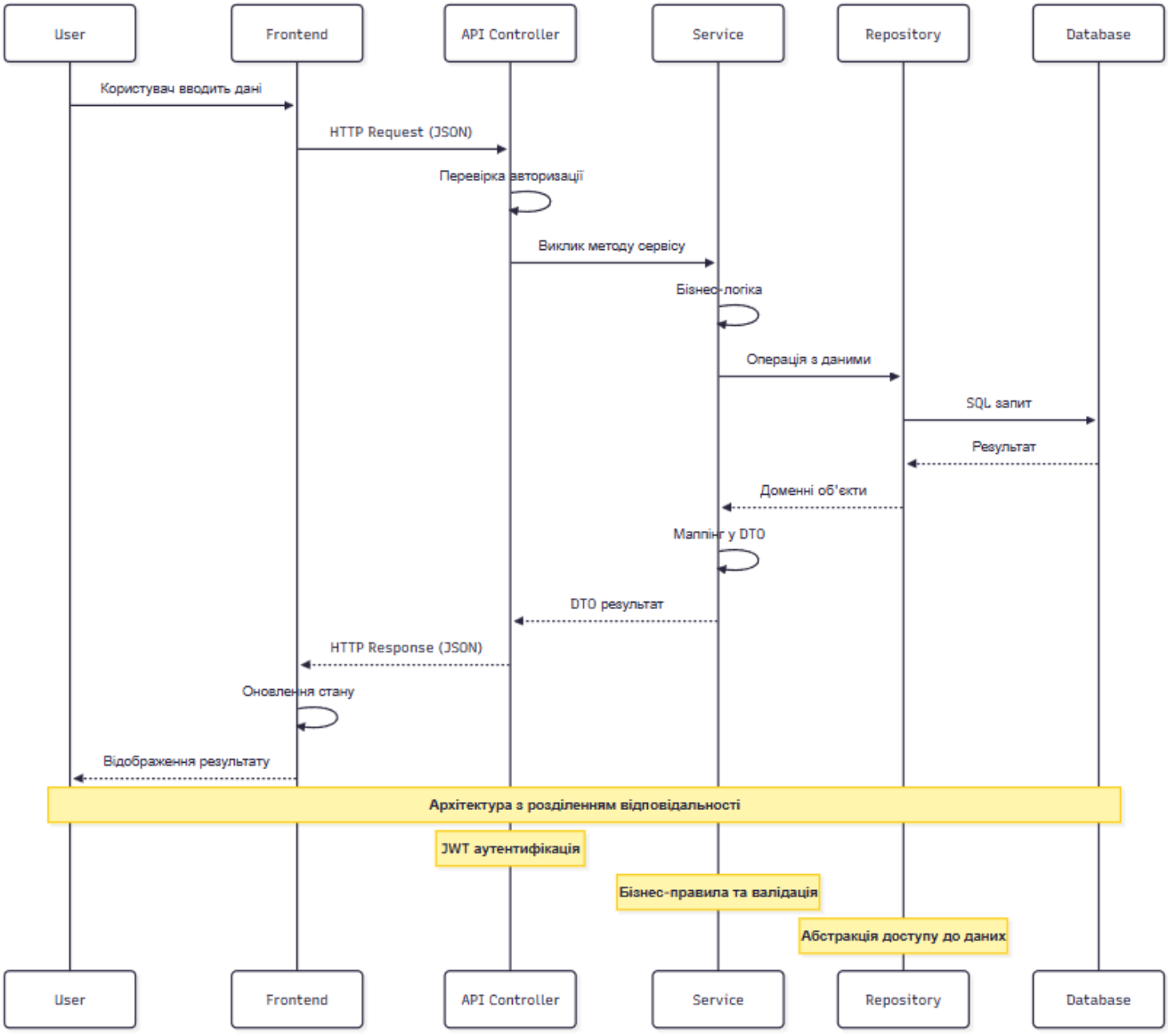


Рис. 2.5 Діаграма потоку даних

DFD дозволяє не лише краще зрозуміти загальну логіку системи, а й виявити потенційні вузькі місця в обробці даних, що особливо важливо на етапі проєктування та тестування.

2.5.4 Діаграма основних сценаріїв

Окрім базової Use Case-діаграми, що була представлена раніше, доцільно окремо виділити розширену діаграму основних сценаріїв використання системи. Вона дозволяє поглиблено розглянути ключові дії в межах найбільш значущих процесів: управління бюджетами, взаємодія з транзакціями, аналітика та цілі.

На рисунку 2.6 зображена діаграма основних сценаріїв

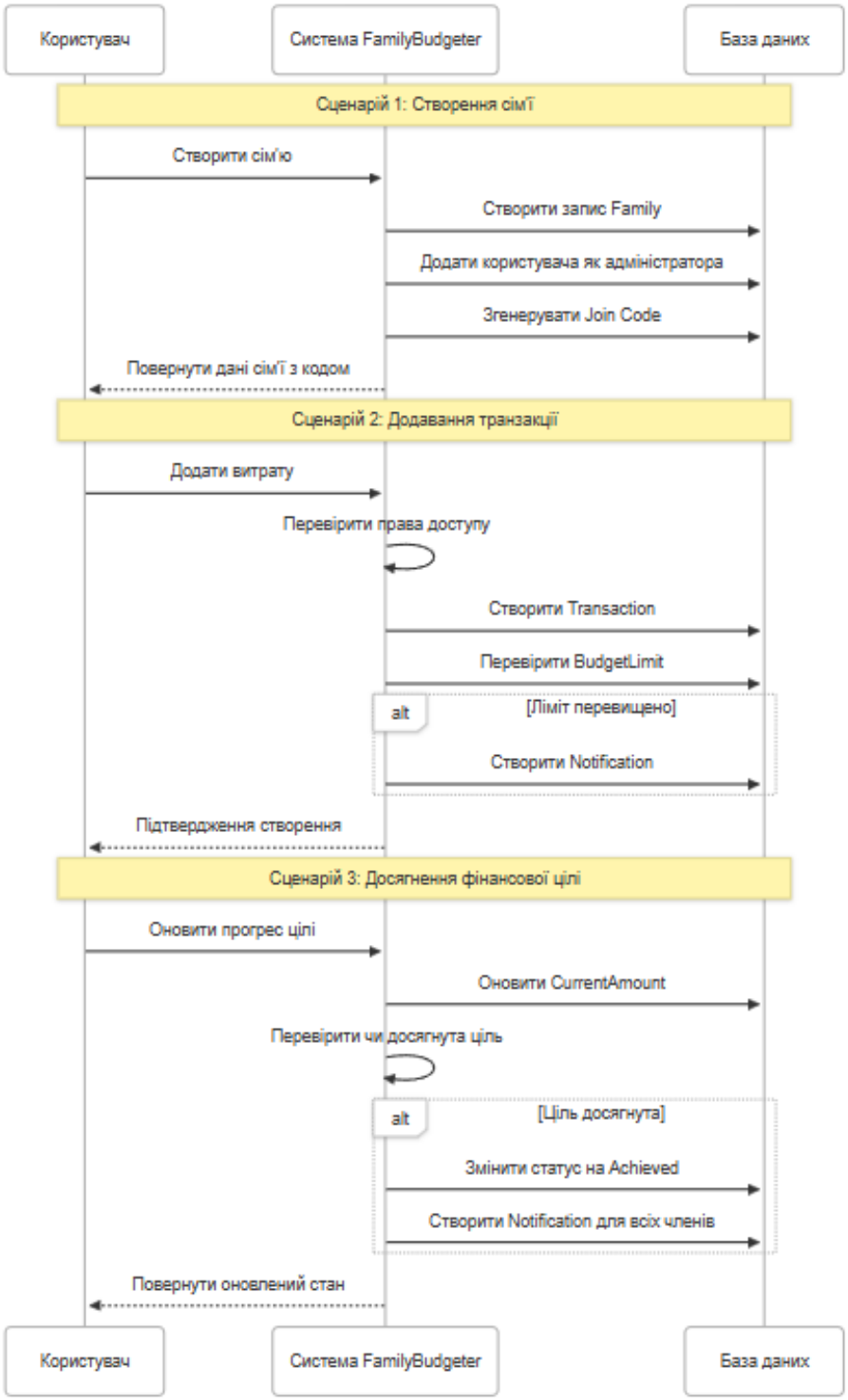


Рис. 2.6 Діаграма основних сценаріїв

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Реалізація бекенд-частини

Бекенд-частина платформи FamilyBudgeter реалізована з використанням ASP.NET Core 8, що забезпечує високу продуктивність, підтримку сучасної архітектури та зручну інтеграцію з іншими технологіями .NET-екосистеми. Уся логіка системи структурована за принципами чистої архітектури (Clean Architecture) із чітким розділенням відповідальностей між окремими шарами. Такий підхід забезпечує підтримуваність, тестованість та можливість легкого розширення проєкту.

Архітектура бекенду поділена на чотири ключові шари, кожен з яких виконує власну роль у загальній структурі системи.

3.1.1 API Layer

Цей шар відповідає за прийом HTTP-запитів від клієнтської частини (React), обробку запитів, валідацію вхідних даних та повернення відповідей. У контролерах цього рівня реалізовані ендпоінти для роботи з транзакціями, бюджетами, категоріями, цілями, регулярними платежами та користувачами.

Контролери не містять бізнес-логіки — вони лише приймають вхідні DTO-об’єкти, викликають відповідні сервіси BLL і повертають результат клієнту. Це забезпечує легкість тестування та дозволяє уникнути дублювання коду.

На рисунку 3.1 зображена діаграма основних класів API Layer.

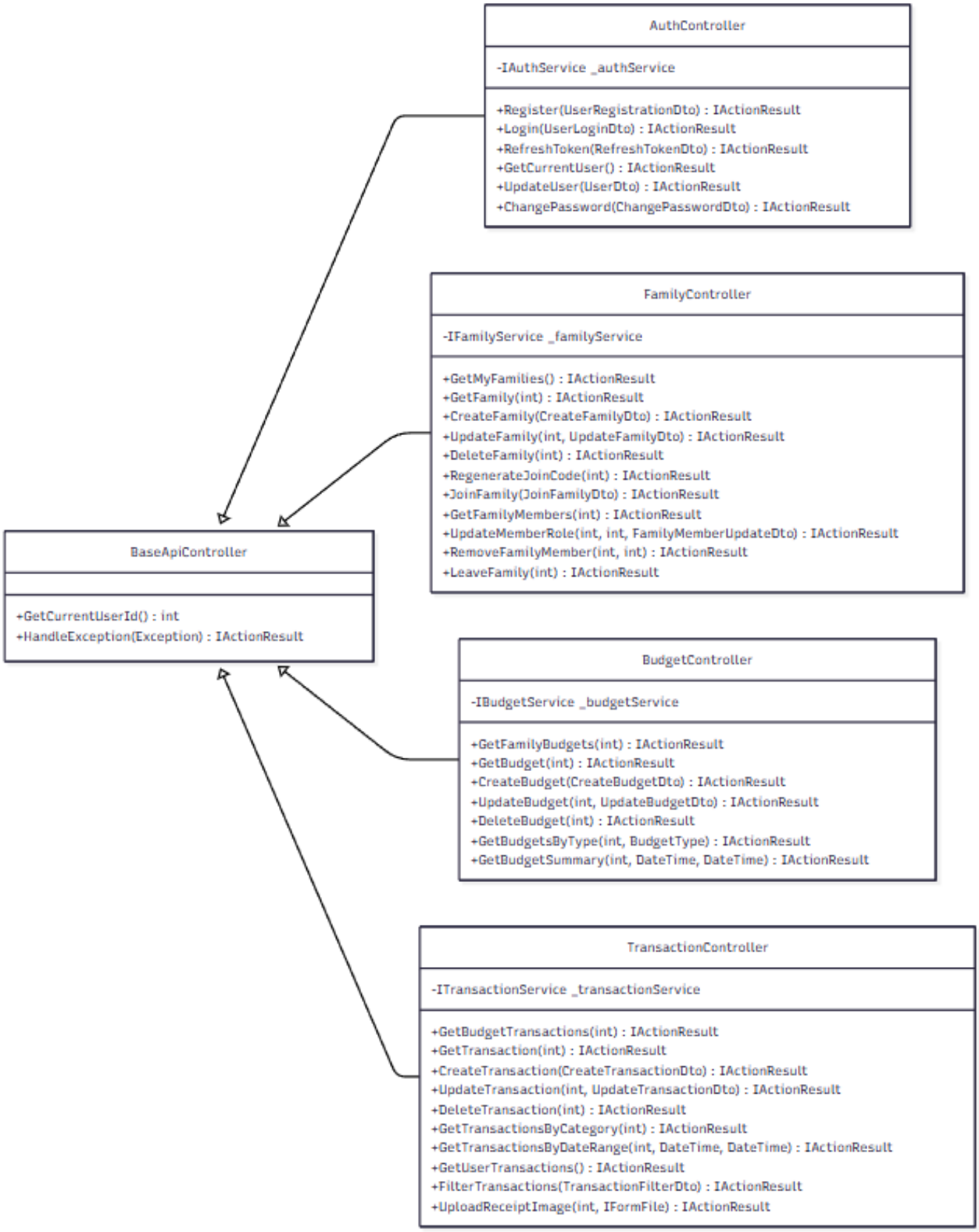


Рис. 3.1 Діаграма основних класів API Layer

3.1.2 Business Logic Layer

У цьому шарі реалізована основна бізнес-логіка програми. Саме тут відбувається обробка вхідних запитів, перевірка доступу, логіка розрахунку лімітів, перевірка категорій, розрахунок прогресу фінансових цілей тощо. Кожна група операцій винесена в окремий сервіс, наприклад: AuthService

Сервіси мають залежності лише від інтерфейсів репозиторіїв (а не від їх конкретної реалізації), що дозволяє легко проводити юніт-тестування, замінювати джерела даних і дотримуватися принципу інверсії залежностей.

На рисунку 3.2 зображена діаграма основних класів Business Logic Layer.

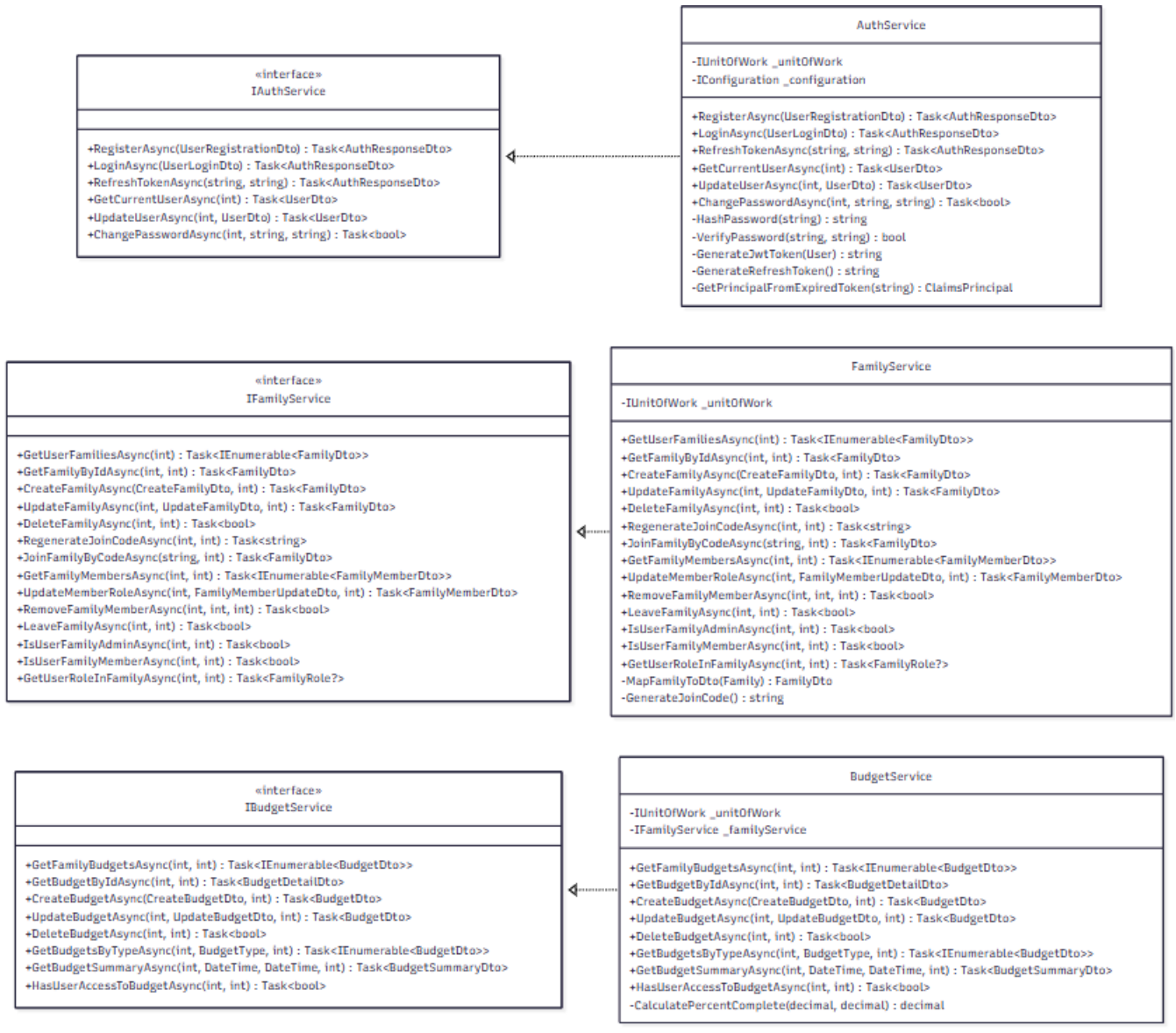


Рис. 3.2 Діаграма основних класів Business Logic Layer

3.1.3 Data Access Layer

Цей шар відповідає за взаємодію з базою даних. Для роботи з даними реалізовано патерни Repository та Unit of Work. Кожен репозиторій інкапсулює доступ до певної сутності (наприклад, транзакцій або категорій) і виконує операції читання/запису з використанням Entity Framework Core.

Окрім базових CRUD-операцій, деякі репозиторії містять методи з фільтрацією, сортуванням і агрегацією, оптимізовані під потреби системи.

На рисунку 3.3 зображена діаграма основних класів Data Access Layer.

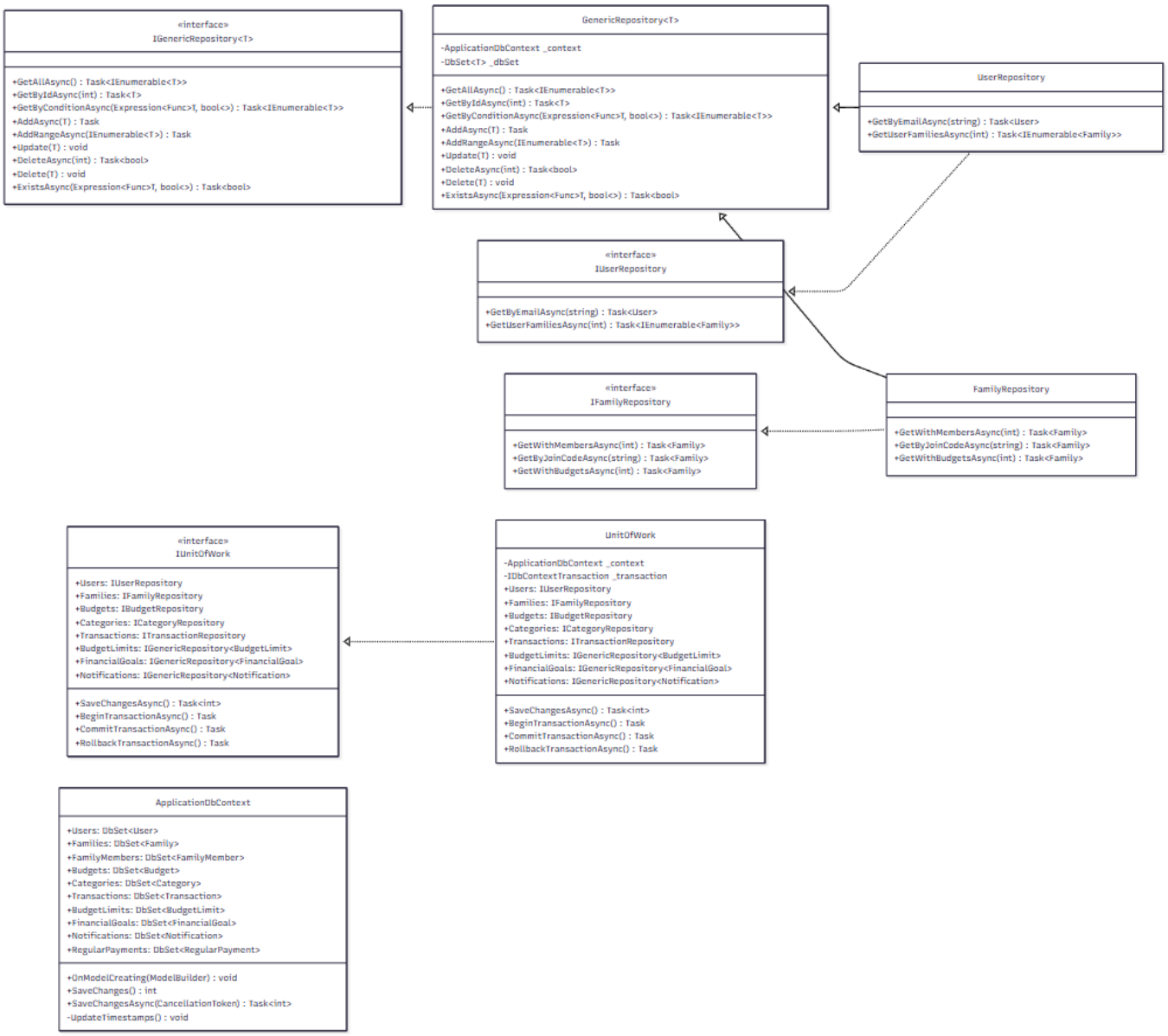


Рис. 3.3 Діаграма основних класів Data Access Layer

3.1.4 Domain Layer

Цей шар містить доменні моделі — класи, які описують сутності предметної області: Transaction, Category, User, Budget тощо. Кожна сутність має властивості, які відповідають структурі відповідної таблиці в базі даних, а також може включати просту доменну логіку (наприклад, метод для оновлення залишку по цілі).

Доменні моделі не містять залежностей від інфраструктури, що дозволяє легко перенести їх у будь-яке інше середовище. Це відповідає принципу незалежності ядра (Core Independence) в Clean Architecture.

На рисунку 3.4 зображена діаграма основних класів Domain Layer.

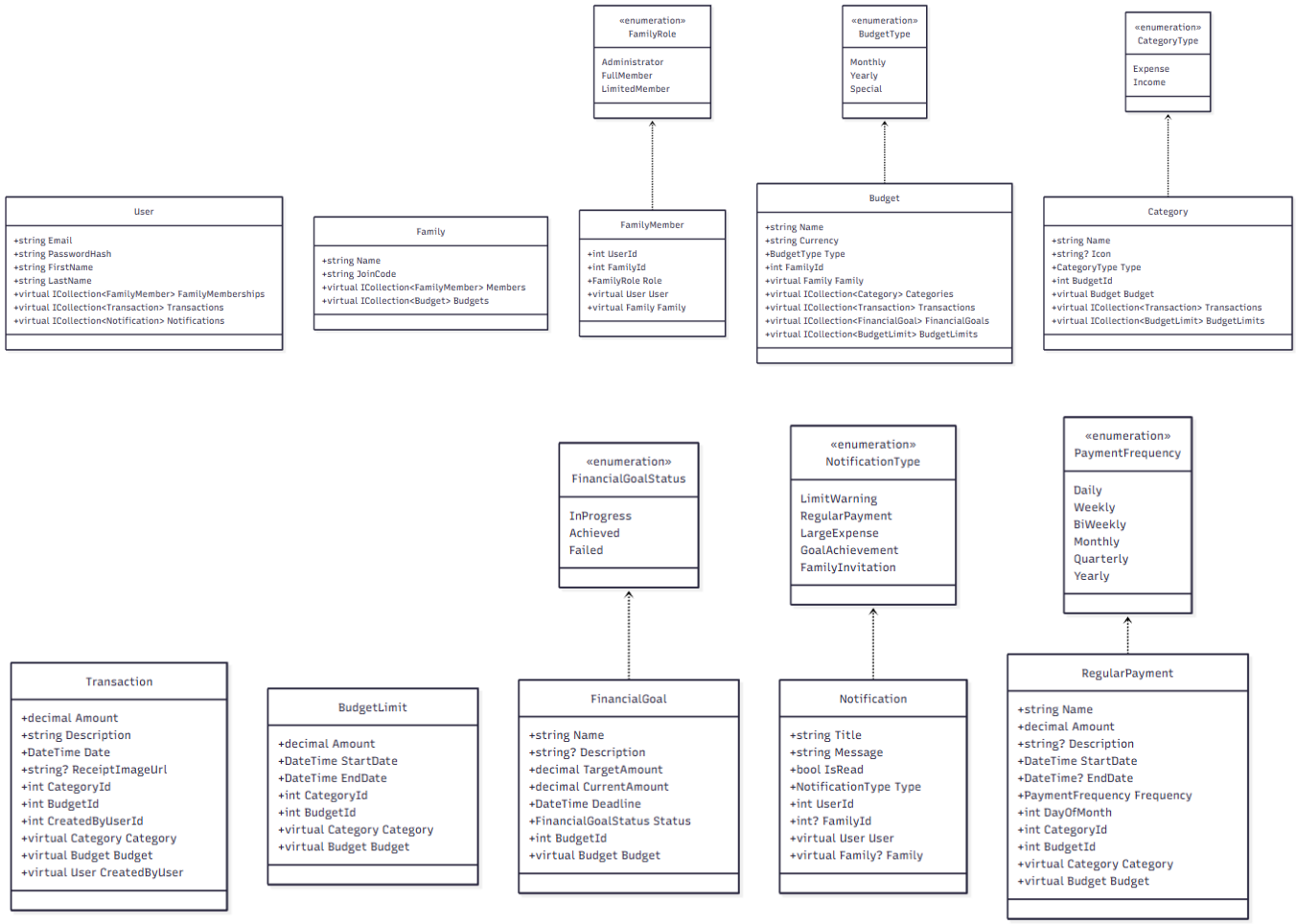


Рис. 3.4 Діаграма основних класів Domain Layer

3.2 Реалізація фронтенд-частини

Клієнтська частина платформи FamilyBudgeter реалізована з використанням сучасного стеку технологій, що забезпечує створення інтуїтивного, швидкого та надійного веб-додатку. Основою фронтенд-застосунку є бібліотека React 18, яка поєднується з мовою програмування TypeScript для забезпечення типобезпеки коду та покращення процесу розробки. Такий підхід дозволяє створювати компоненти з чітко визначеними типами даних, що мінімізує кількість помилок під час виконання та значно полегшує рефакторинг і підтримку коду.

Архітектура фронтенд-застосунку побудована за принципом компонентно-орієнтованого програмування, де кожен елемент інтерфейсу реалізується як незалежний компонент. Це забезпечує високий рівень модульності та дозволяє ефективно використовувати компоненти повторно в різних частинах додатку. Компоненти логічно згруповані за функціональними областями: авторизації та аутентифікації, управління сім'ями, бюджетування, транзакцій, звітності та налаштувань. Кожна група компонентів інкапсулює свою логіку та стан, що дозволяє розробляти та тестувати їх незалежно один від одного.

Для стилізації та забезпечення адаптивного дизайну використовується фреймворк Bootstrap 5, який надає готові компоненти інтерфейсу та сітку для створення респонсивних макетів. Цей вибір дозволяє швидко створювати сучасний інтерфейс, який однаково добре виглядає на різних пристроях - від мобільних телефонів до настільних комп'ютерів. Bootstrap класи використовуються разом з кастомними стилями для створення унікального дизайну платформи, який підкреслює її функціональне призначення.

Взаємодія з серверною частиною організована за допомогою бібліотеки Axios, яка забезпечує зручний API для виконання HTTP-запитів до backend. Для централізованої обробки запитів та відповідей використовуються інтерцептори Axios, які автоматично додають до кожного запиту необхідні заголовки (наприклад, JWT-токен для аутентифікації), обробляють помилки на рівні HTTP та

забезпечують єдиний підхід до логування та моніторингу мережевого трафіку. Це значно спрощує обробку помилок та забезпечує консистентність в роботі з API.

Маршрутизація в додатку реалізована за допомогою React Router v6, який забезпечує декларативний підхід до організації навігації. Система маршрутів включає підтримку приватних маршрутів, які перенаправляють неавторизованих користувачів на сторінку входу, а також роле-орієнтовані маршрути, які надають різні рівні доступу в залежності від ролі користувача в сім'ї. Така організація забезпечує безпеку додатку та відповідний користувацький досвід для різних категорій користувачів.

Структура проєкту організована з урахуванням масштабованості та зручності супроводження. Компоненти розподілені по папках відповідно до функціональних областей, кожен модуль містить власні типи TypeScript, кастомні хуки для управління станом та допоміжні утиліти. Стан додатку керується комбінацією React Context для глобального стану (наприклад, інформації про аутентифікованого користувача) та локального стану компонентів для специфічної функціональності. Такий підхід дозволяє ефективно обробляти дані на різних рівнях додатку без її надмірного ускладнення.

Інтерфейс користувача побудований з акцентом на зручність використання та доступність. Кожна сторінка додатку дизайниться з урахуванням принципів UX, забезпечуючи логічний розподіл елементів, зрозумілу навігацію та швидке виконання основних операцій. Форми валідуються як на клієнтському, так і на серверном рівні, що забезпечує миттєвий зворотний зв'язок для користувача та запобігає відправленню некоректних даних. Завдяки використанню TypeScript типізація поширюється не тільки на компоненти, але й на API-взаємодію, що дозволяє забезпечити консистентність даних між клієнтом та сервером.

Перше знайомство користувача з платформою відбувається через сторінки аутентифікації, які демонструють мінімалістичний та зрозумілий дизайн. Рисунок 3.5 Сторінка входу в систему показує головну сторінку входу, де користувач може ввести свої облікові дані. Форма входу містить поля для електронної пошти та пароля з вбудованою валідацією на стороні клієнта.

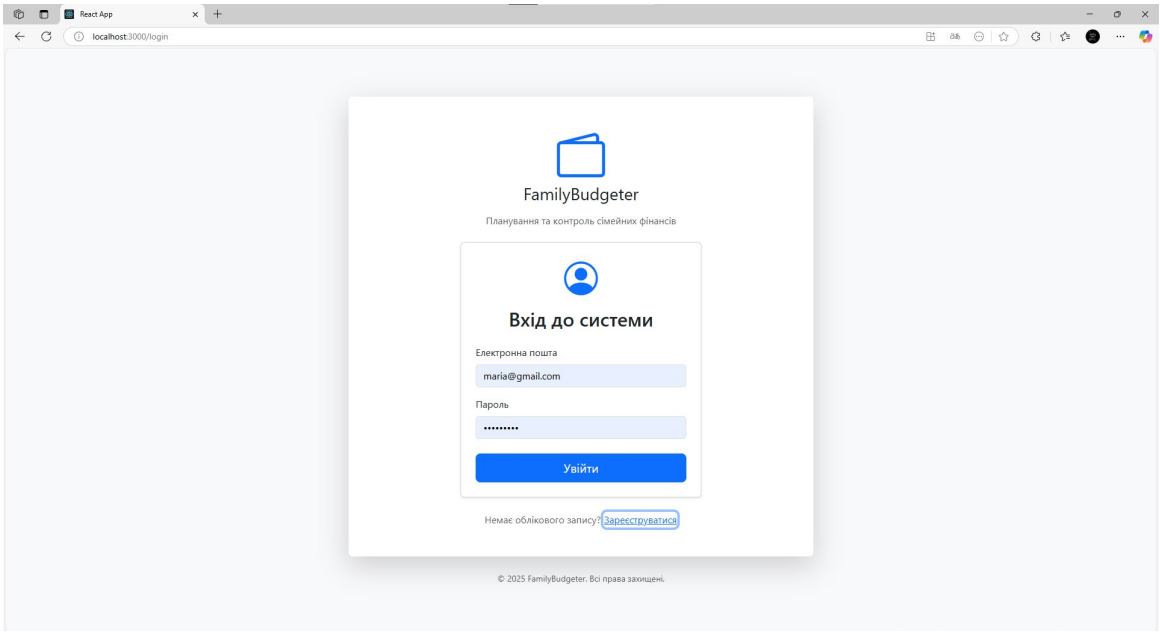


Рис. 3.5 Сторінка входу в систему

Сторінка реєстрації, яка зображена на рисунку 3.6 пропонує інтуїтивний процес створення облікового запису. Форма містить поля для введення імені, прізвища, електронної пошти та пароля з підтвердженням. Валідація відбувається в реальному часі, надаючи користувачу миттєвий зворотний зв'язок про коректність введених даних. Дизайн форм виконаний в едином стилі з використанням Bootstrap компонентів, що забезпечує консистентність інтерфейсу.

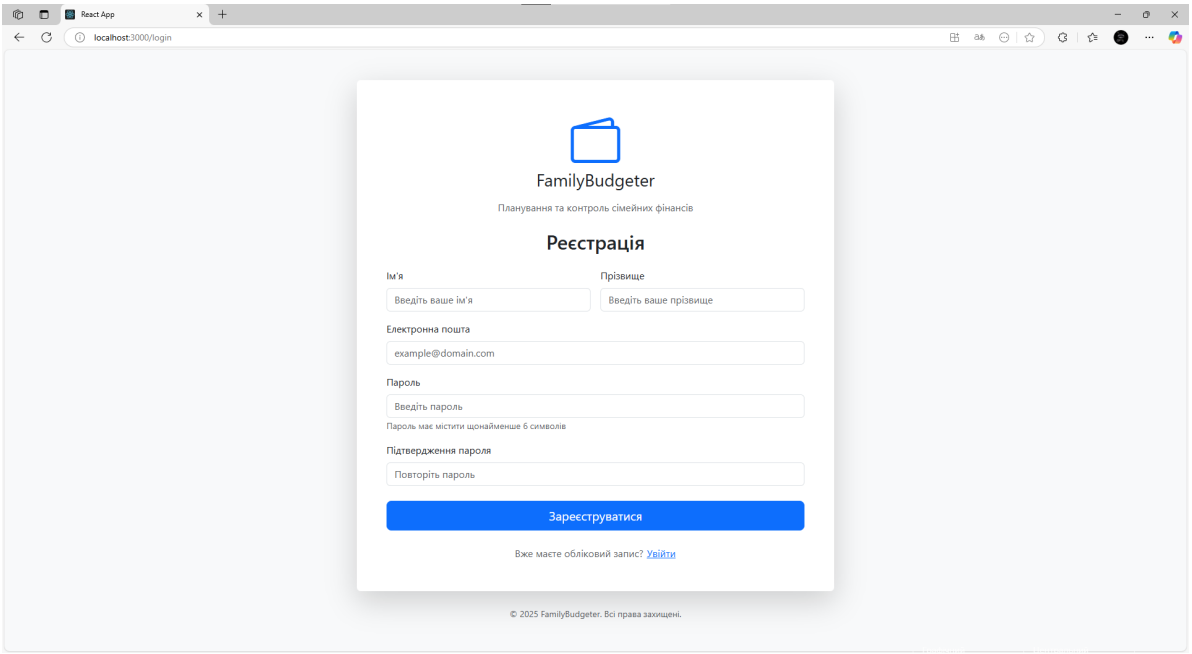


Рис. 3.6 Форма реєстрації нового користувача

Після успішної авторизації користувач потрапляє на головну панель управління, яка надає швидкий огляд ключової інформації про фінансовий стан сім'ї. Панель містить віджети з актуальною інформацією: поточний баланс, найбільші витрати поточного місяця, прогрес досягнення фінансових цілей та останні транзакції. Кожен віджет є інтерактивним та веде до відповідної детальної сторінки при натисканні.

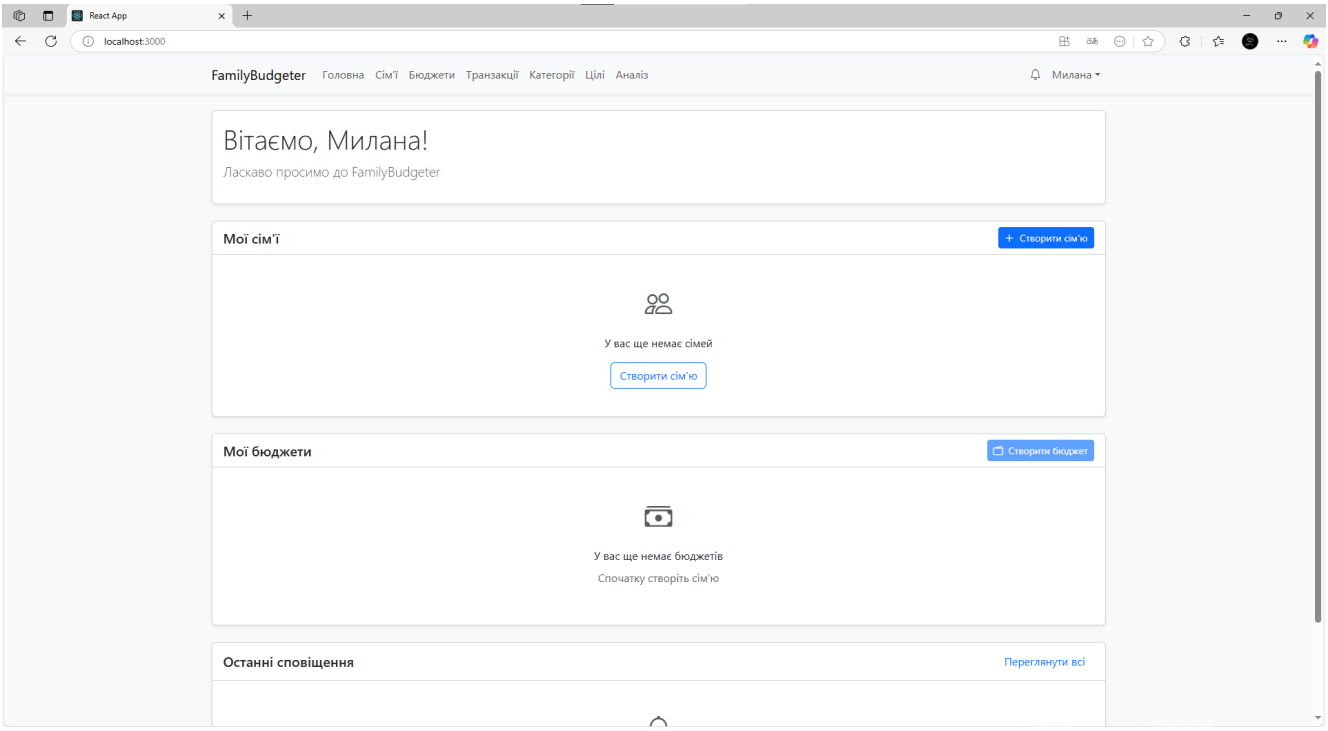


Рис. 3.7 Головна панель Dashboard

Навігаційне меню, яке продемонстроване на рисунку 3.8 розташоване в верхній частині екрану та містить основні розділи додатку: "Головна", "Сім'я", "Бюджети", "Транзакції", "Категорії", "Аналіз" та "Налаштування". Адаптивний дизайн забезпечує коректне відображення меню на мобільних пристроях, де воно згортається в "бургер-меню". Активний розділ виділяється кольором, що полегшує орієнтування користувача в системі.

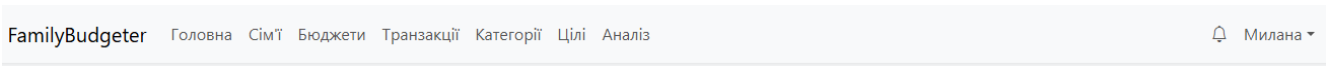


Рис. 3.8 Головне меню навігації

Розділ управління сім'єю (рисунок 3.9) дозволяє користувачам створювати нові сім'ї, приєднуватися до існуючих за допомогою коду запрошення та управляти членами родини. Сторінка містить карточки з інформацією про кожного члена сім'ї, включаючи ім'я, роль та дату приєднання. Адміністратори можуть змінювати ролі користувачів або виключати їх з сім'ї через контекстне меню.

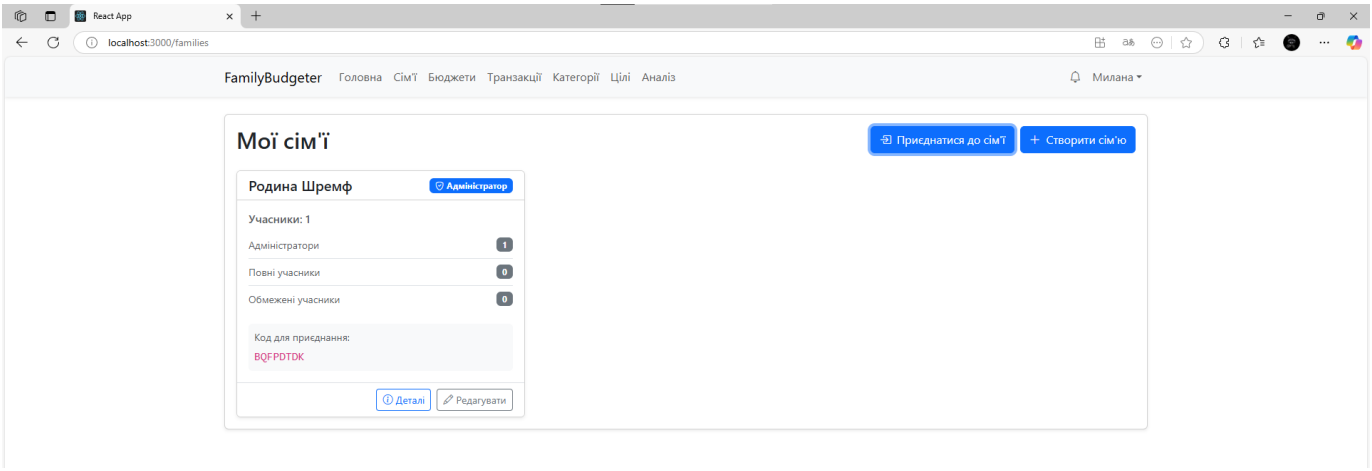


Рис. 3.9 Сторінка управління сім'єю

Модальне вікно приєднання до сім’ї (рисунок 3.10) з'являється при натисканні на кнопку "Приєднатись до сім’ї" і містить форму в яку можна ввести унікальний код приєднання. Інтерфейс також дозволяє регенерувати код у разі необхідності збільшити безпеку.

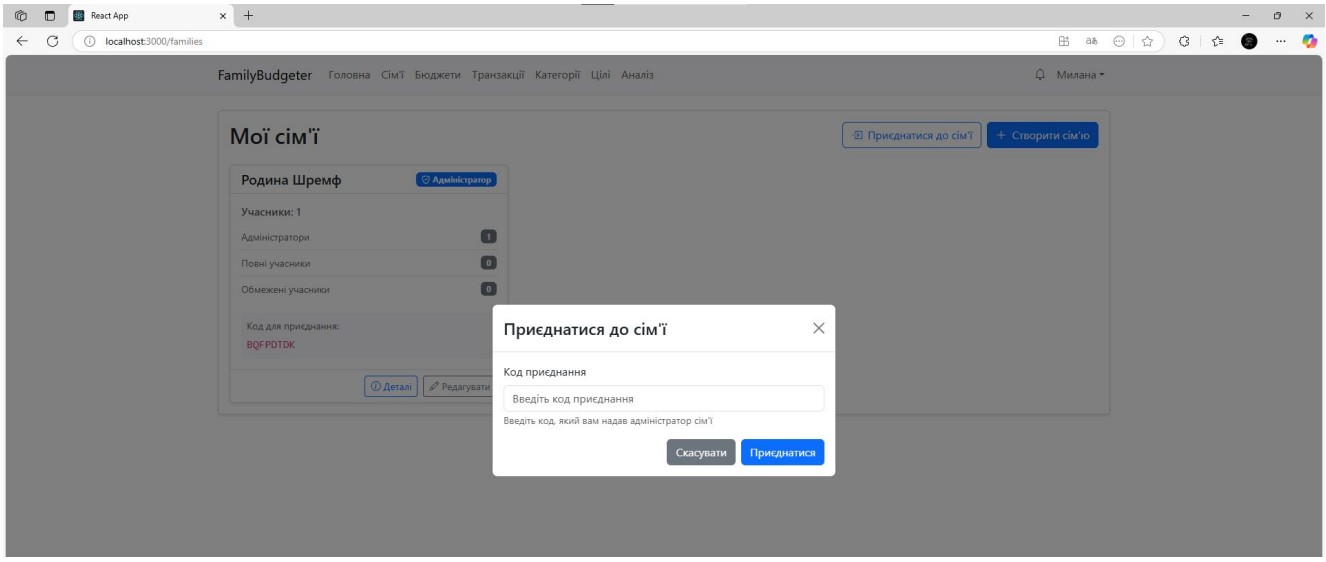


Рис. 3.10 Форма приєднання до сім’ї

Сторінка бюджетів (рисунок 3.11) відображає всі активні бюджети сім'ї у вигляді карточок з ключовою інформацією: назва, тип (місячний, річний, спеціальний), валюта та поточний баланс. Кнопки "Деталі", "Додати транзікцію забезпечують швидкий доступ до основних операцій.

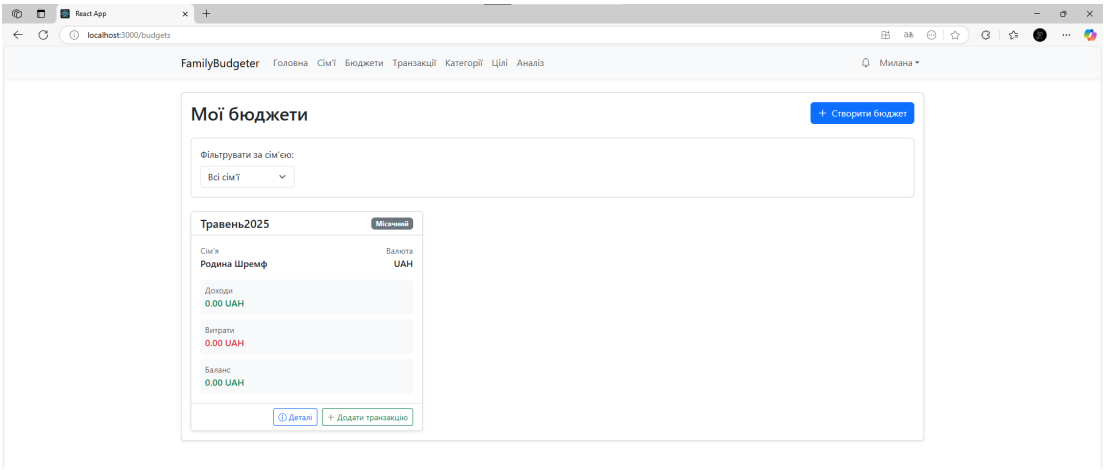


Рис. 3.11 Список сімейних бюджетів

Детальна сторінка бюджету (рисунок 3.12) надає глибокий аналіз фінансових потоків. Тут розміщені Витрат за категоріями, таблиця лімітів з індикаторами перевищення, список фінансових цілей з прогрес-барами та останні транзакції. Інтерактивні елементи дозволяють швидко переходити між різними періодами та фільтрувати інформацію.

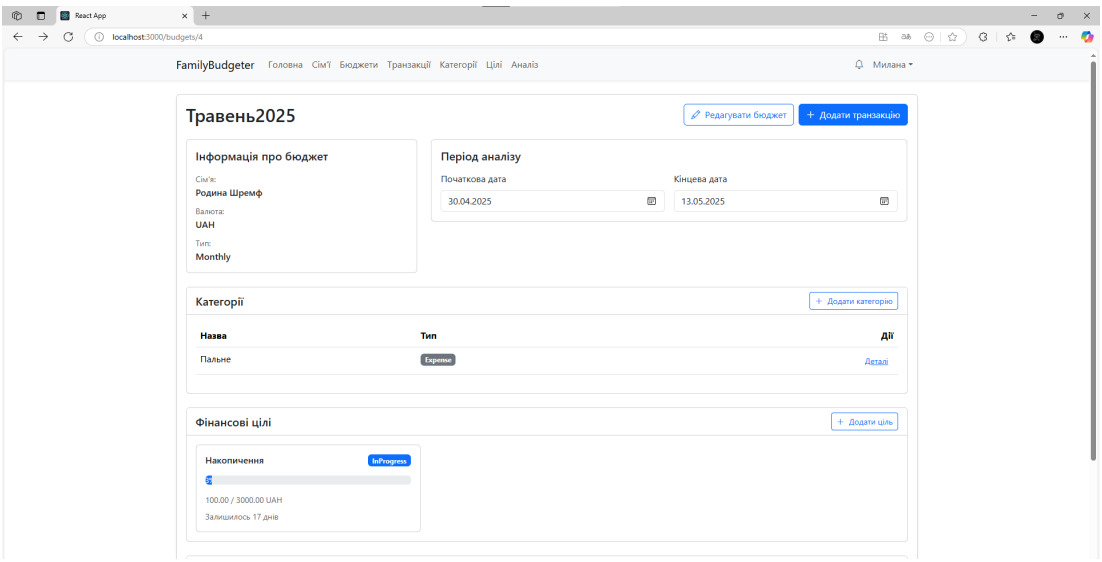


Рис. 3.12 Детальний перегляд бюджету

Розділ категорій є одним з найважливіших компонентів системи, оскільки саме через них здійснюється класифікація всіх фінансових операцій. Головна сторінка категорій (рисунок 3.13) відображає всі категорії поточного бюджету, розділені на дві групи: доходи та витрати. Кожна категорія представлена у вигляді картки з іконкою, назвою, типом та статистикою використання за поточний місяць. Для кращої візуалізації категорії витрат відображаються червоними акцентами, а доходи - зеленими.

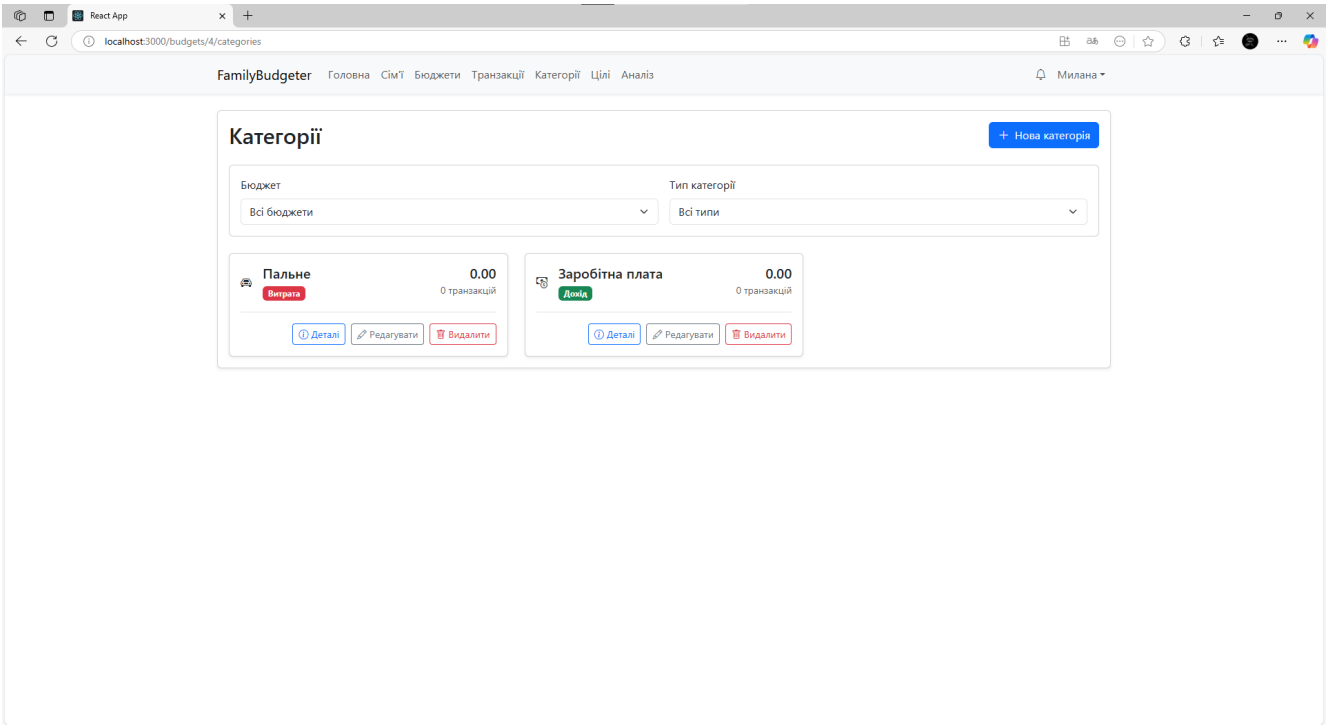


Рис 3.13 Список категорій для поточного бюджету

Інтерфейс підтримує швидкі дії для кожної категорії: редагування через модальне вікно, видалення з підтвердженням та перегляд детальної статистики. Над списком розташована панель з перемикачами фільтрів, що дозволяє швидко переключатися між типами категорій. API-інтерфейс взаємодії

Сторінка створення нової категорії (рисунок 3.14) відкривається в модальному вікні або на окремій сторінці залежно від розміру екрана. Форма містить обов'язкове поле для назви категорії, радіо-кнопки для вибору типу (дохід

або витрата) та селектор іконки. Бібліотека іконок включає понад 100 символів, згрупованих за темами: їжа, транспорт, розваги, комунальні послуги, освіта та інші.

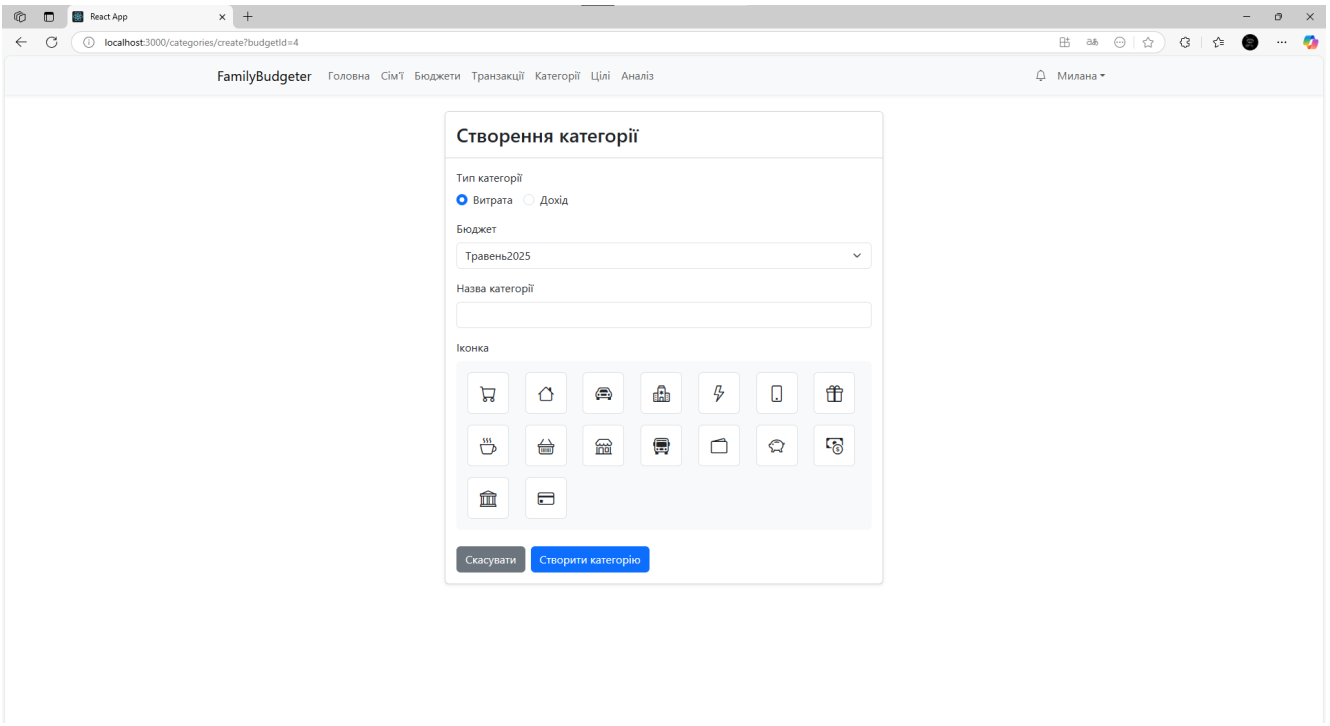


Рис 3.14 Форма створення нової категорії

Валідація форми відбувається в реальному часі: система перевіряє унікальність назви категорії в межах бюджету та коректність заповнення всіх полів. При спробі створити категорію з вже існуючою назвою з'являється попередження з пропозицією змінити назву або відкрити існуючу категорію для редагування.

Інтерфейс додавання транзакцій (рисунок 3.15) розроблений з акцентом на швидкості введення даних. Форма містить поля для суми, опису, дати, вибору категорії та можливості прикріплення фото чека. Випадний список категорій автоматично фільтрується за типом (доходи або витрати).

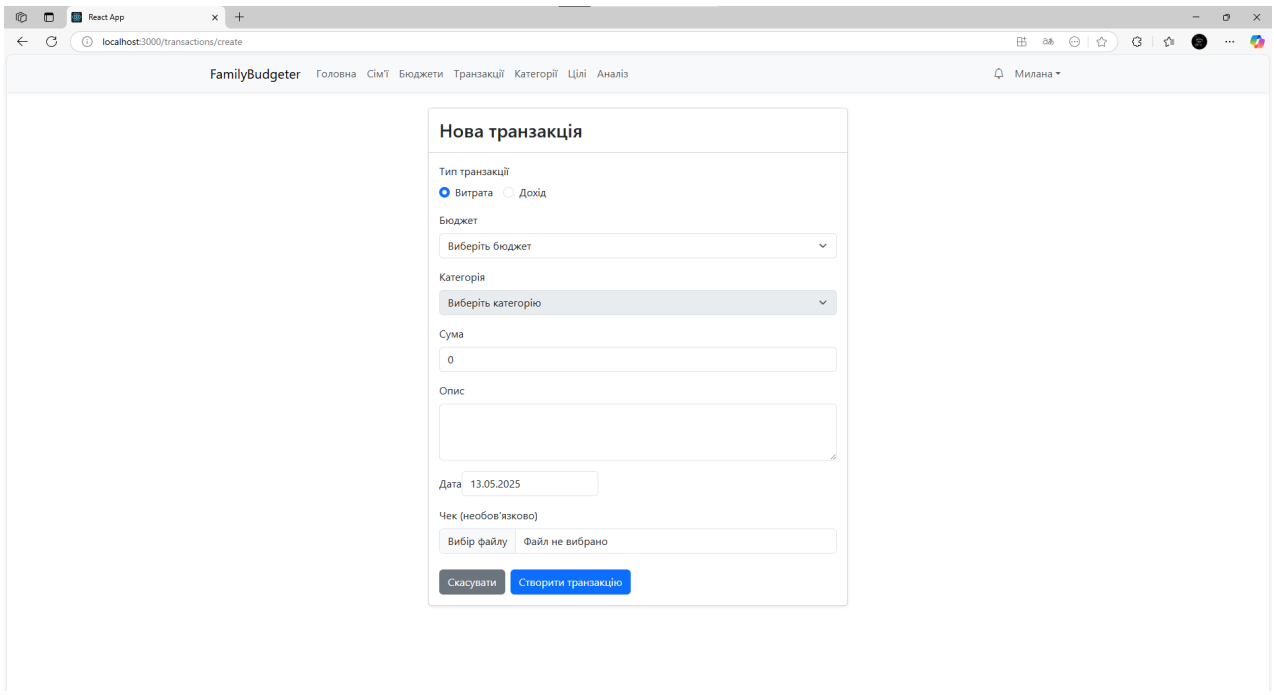


Рис. 3.15 Форма створення нової транзакції

Список транзакцій (рисунок 3.16) організований у вигляді таблиці з можливістю сортування за будь-яким стовпцем. Панель фільтрів вгорі дозволяє швидко знайти потрібні операції за датою, категорією, сумою або описом. Кожен рядок містить іконку категорії, суму з кольоровим індикатором (зелений для доходів, червоний для витрат), дату та ім'я користувача, який створив операцію.

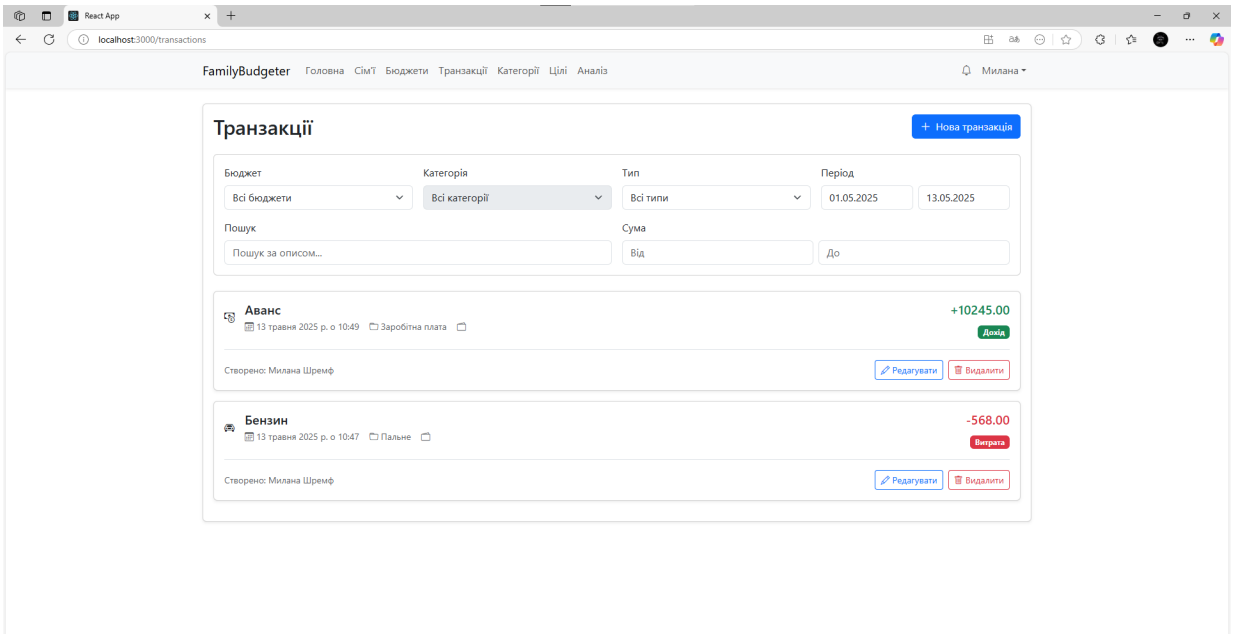


Рис. 3.16 Таблиця транзакцій з фільтрами

Маршрутизація в додатку реалізована за допомогою React Router v6, який забезпечує декларативний підхід до організації навігації. Система маршрутів включає підтримку приватних маршрутів, які перенаправляють неавторизованих користувачів на сторінку входу, а також роле-орієнтовані маршрути, які надають різні рівні доступу в залежності від ролі користувача в сім'ї. Така організація забезпечує безпеку додатку та відповідний користувацький досвід для різних категорій користувачів.

Розділ фінансових цілей (рисунок 3.17) представляє активні цілі сім'ї у вигляді візуально привабливих карточок. Кожна картка містить назву цілі, опис, цільову суму, поточний прогрес у вигляді прогрес-бара та дедлайн. Кольорове кодування допомагає швидко оцінити статус: зелений для досягнутих цілей, жовтий для тих, що наближаються до дедлайну, та червоний для прострочених.

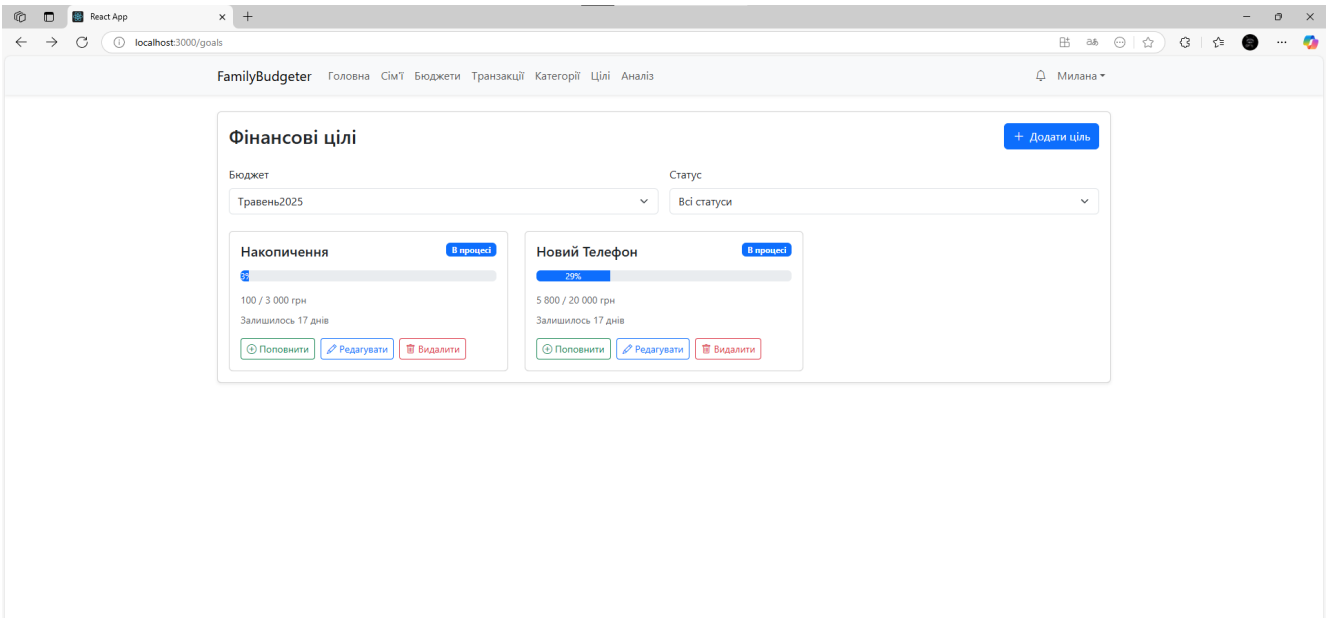


Рис. 3.17 Карточки фінансових цілей

Сторінка звітів (рисунок 3.18) надає комплексний огляд фінансової ситуації через різноманітні графіки та діаграми. Лінійний графік показує динаміку доходів та витрат за вибраний період, гістограма відображає витрати за категоріями, а кругова діаграма демонструє структуру витрат у відсотках. Інтерактивність

графіків дозволяє користувачам деталізувати дані шляхом натискання на окремі сегменти.

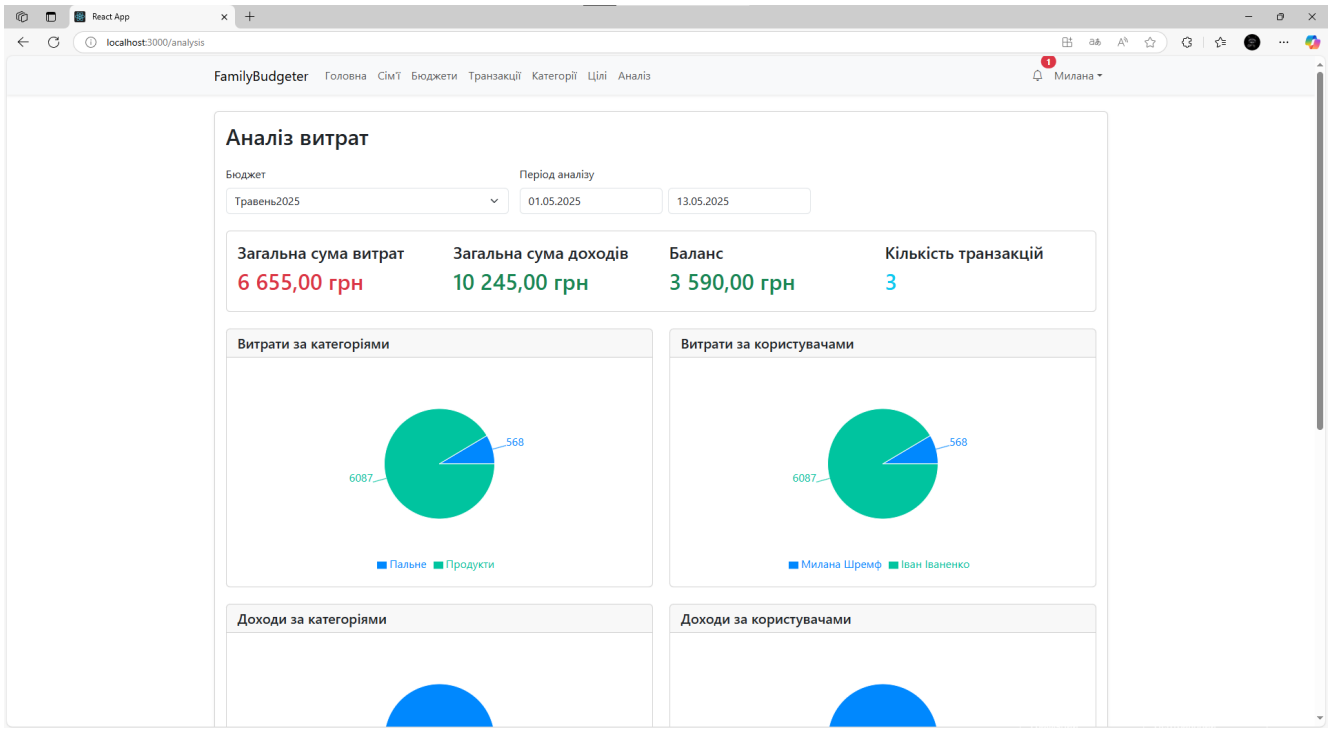


Рис 3.18 Панель аналітики

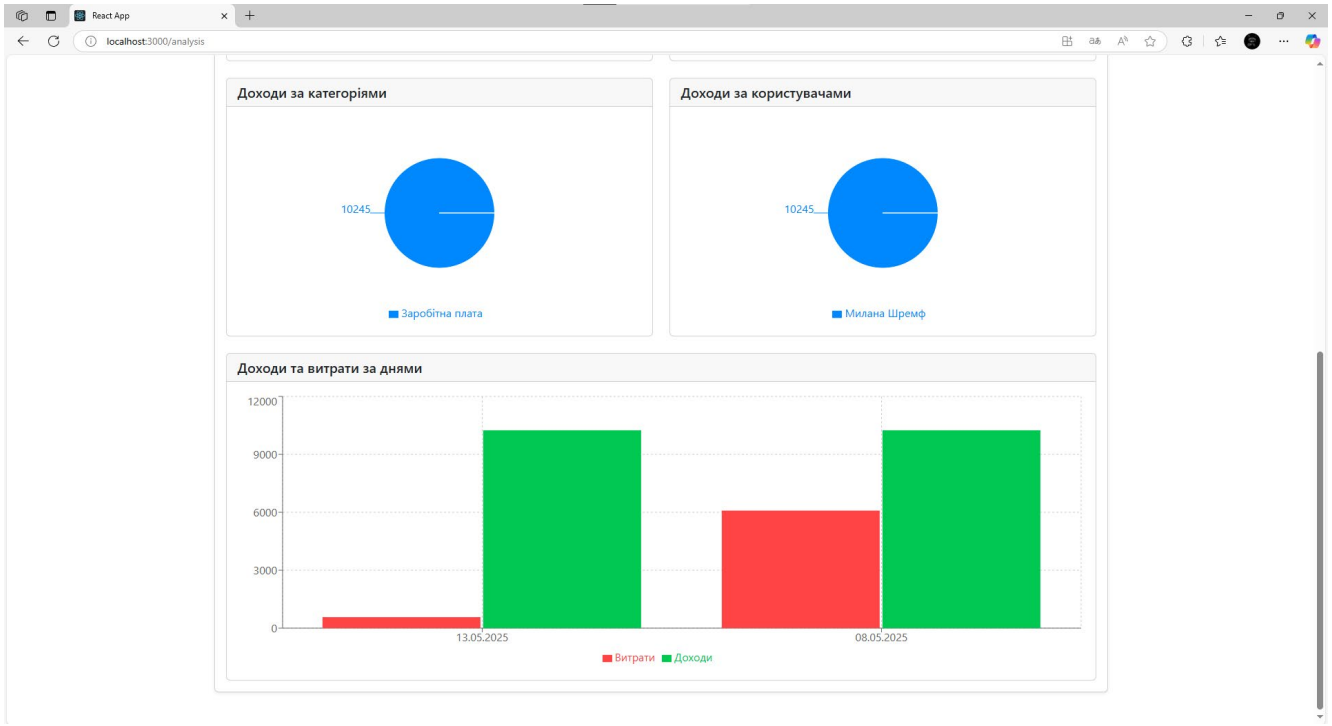


Рис 3.19 Панель аналітики доходів та витрат

4 ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ

4.1 Методика тестування

Якість роботи програмного забезпечення напряму залежить від ефективності та повноти проведеного тестування. У випадку з платформою FamilyBudgeter, основна увага була зосереджена на перевірці ключових функціональних сценаріїв, коректності обробки даних, відповідності поведінки системи очікуваним результатам, а також на стабільності взаємодії між фронтендом і бекендом. Тестування проводилось поетапно на кожному з рівнів розробки та охоплювало як окремі модулі, так і комплексну поведінку системи в цілому.

Перший етап тестування здійснювався під час розробки окремих компонентів — це так зване ручне модульне тестування. Розробник одразу перевіряв поведінку сервісів, контролерів, форм та запитів API на відповідність вимогам. Особливу увагу було приділено правильності маршрутизації, валідації даних та обробці помилок. На цьому етапі також перевірялась робота інтерцепторів, збереження JWT токенів, логіка захищених маршрутів на фронтенді та правильність розподілу доступу відповідно до ролей користувачів.

Наступним кроком було тестування основних сценаріїв використання платформи у браузері, в умовах, наближених до реального використання. Було перевірено можливість реєстрації нового користувача, створення сім’ї, додавання учасників, створення категорій, проведення транзакцій, встановлення бюджетних обмежень та формування фінансових цілей. Кожен сценарій супроводжувався спробами ввести некоректні дані (наприклад, пусті поля, помилкові типи даних, доступ до чужої інформації), щоб переконатися в надійності валідації як на фронтенді, так і на бекенді.

Окремо тестувалась поведінка системи у разі перевищення бюджетного ліміту, а також автоматичне створення повторюваних платежів. Перевірялась точність формування звітів, коректність відображення графіків, відстеження

прогресу цілей та обробка сповіщень про фінансові події. Також проводився аналіз обробки помилок: наприклад, якщо сервер недоступний, або якщо токен доступу неактуальний — користувач повинен отримувати зрозуміле повідомлення, а не технічну помилку.

На завершальному етапі проводилось кросбраузерне тестування, а також перевірка відображення інтерфейсу на мобільних пристроях. Система адаптувалась під різні розміри екранів, що є особливо важливим для сучасних користувачів, які можуть взаємодіяти з платформою зі смартфонів, планшетів або ноутбуків.

Уся методика тестування була спрямована на виявлення помилок, покращення зручності користування та забезпечення стабільної роботи системи у повному циклі її використання — від входу користувача до формування фінансового звіту. Проведене тестування дозволило переконатися у надійності реалізованої архітектури та підтвердити функціональну готовність продукту до використання.

4.2 Результати тестування функціональності

Після завершення основного етапу розробки було проведено комплексне функціональне тестування платформи FamilyBudgeter, яке охоплювало всі ключові сценарії використання системи. Метою було переконатися, що кожна функція працює відповідно до вимог, описаних у технічному завданні, та що взаємодія між компонентами системи відбувається безпомилково.

Тестування розпочалося зі сценаріїв, пов’язаних із автентифікацією та авторизацією. Було підтверджено, що користувачі можуть успішно реєструватися в системі, входити до своїх облікових записів, отримувати та зберігати JWT токени, а також отримувати доступ лише до дозволених функцій згідно з призначеною роллю. При спробі доступу до захищених маршрутів без авторизації система коректно перенаправляє користувача на сторінку входу.

У процесі перевірки функціональності взаємодії з сім’єю було успішно протестовано створення нової сім’ї, генерацію унікального коду для запрошення,

приєднання інших користувачів за цим кодом, а також призначення ролей (адміністратор, учасник, обмежений учасник). Було перевірено, що кожна роль має відповідний рівень доступу: наприклад, адміністратор має повний контроль, а обмежений учасник може лише переглядати деякі дані.

Окрему увагу було приділено тестуванню транзакцій: додавання, редагування, фільтрація, сортування та видалення доходів і витрат працювали стабільно. Додатково перевірялась можливість прикріплення фото чеків, їх передача до сервера та правильне відображення на фронтенді. Категорії, які використовуються для класифікації транзакцій, створювались і редагувались без помилок, а також правильно підраховувались витрати за кожною з них.

Тестування бюджету показало, що встановлення лімітів за категоріями працює очікувано: система фіксує витрати, порівнює їх із заданими межами та, у разі перевищення, активує відповідне сповіщення. Також успішно перевірено формування щомісячних, річних та спеціальних бюджетів, їх зміну й видалення.

Функціональність фінансових цілей, включаючи створення, відстеження прогресу, автоматичне оновлення статусу та досягнення мети, працювала стабільно. Сторінки з аналітикою коректно формували звіти про доходи та витрати, візуалізували інформацію у вигляді діаграм, а також дозволяли порівняння планових і фактичних бюджетів за різні періоди.

Було перевірено стабільну роботу повторюваних платежів. Система створювала нові транзакції на основі налаштувань (щоденно, щотижня, щомісяця) без потреби втручання користувача, що повністю відповідало описаній бізнес-логіці.

Усі функції, що стосуються обробки помилок, також працювали належним чином. Неправильні запити, відсутність доступу, невалідні дані чи неавторизовані дії супроводжувалися чіткими повідомленнями на клієнтській частині, що позитивно впливає на користувацький досвід.

Загалом результати тестування показали, що функціональність системи FamilyBudgeter реалізована успішно, відповідає поставленим завданням і готова до подальшого розгортання та використання у реальному середовищі.

4.3 Аналіз продуктивності системи

Продуктивність є одним із ключових показників якості програмного забезпечення, особливо в системах, які взаємодіють із великою кількістю даних та передбачають активне використання багатьма користувачами. У випадку FamilyBudgeter важливо було забезпечити швидке реагування на запити, оптимізовану роботу з базою даних та стабільну взаємодію між клієнтською і серверною частинами системи.

Аналіз продуктивності проводився в умовах, максимально наближених до реального використання. Для цього система була розгорнута на локальному сервері з типовими для невеликого SaaS-рішення параметрами: процесор середнього класу, 8 ГБ оперативної пам’яті, SSD-накопичувач та SQL Server Express Edition. Тестування охоплювало різні типи запитів: читання, запис, фільтрацію, генерацію звітів, а також взаємодію з основними сервісами — транзакціями, бюджетами, категоріями, користувачами.

Одним із важливих аспектів було вимірювання часу відповіді API. У більшості випадків запити на отримання списку транзакцій, створення нової операції чи оновлення бюджету оброблялися в межах 100–300 мілісекунд, що вважається хорошим показником для систем із серверною логікою. Найдовші запити (наприклад, генерація аналітичного звіту з фільтрацією по категоріях і датах) не перевищували 600 мілісекунд, що також є прийнятним рівнем для звичайного користувача.

Система показала стабільну роботу при одночасному використанні кількох сесій. Навіть за умови активного додавання транзакцій з різних акаунтів, швидкість взаємодії залишалась сталою. Це свідчить про ефективну реалізацію доступу до бази даних через ORM-бібліотеку Entity Framework Core із використанням оптимізованих запитів та правильного використання асинхронних методів.

Ще одним аспектом було тестування швидкодії клієнтської частини. Завдяки використанню React 18 та Vite як інструменту збирання, фронтенд додатку завантажується швидко, навіть при повільному інтернет-з’єднанні. Використання

кешування, оптимізованої маршрутизації, а також компонентного підходу до побудови інтерфейсу дозволило зменшити кількість зайвих перерендерів і прискорити відображення сторінок. Навіть на мобільних пристроях з обмеженими ресурсами інтерфейс залишався плавним та швидким у взаємодії.

Важливо зазначити, що застосунок коректно обробляє ситуації зі зниженою доступністю сервера, повільною мережею або помилками запитів — завдяки використанню інтерцепторів у Axios користувач отримує зрозуміле повідомлення, а не просто технічну помилку. Це не лише покращує користувацький досвід, а й знижує навантаження на службу підтримки.

Таким чином, проведений аналіз продуктивності показав, що система FamilyBudgeter відповідає основним вимогам до швидкодії, стабільності й масштабованості. Вона демонструє хорошу реакцію на дії користувача, оптимальну роботу з даними та забезпечує комфортну роботу навіть на пристроях зі середніми технічними характеристиками.

4.4 Оцінка зручності користування

У процесі тестування особливу увагу було приділено оцінці зручності користування системою FamilyBudgeter. Інтерфейс платформи розроблений з урахуванням принципів простоти, доступності та логічності. Завдяки використанню Bootstrap інтерфейс є адаптивним і коректно відображається на різних типах пристроїв.

Користувачі легко орієнтуються в системі завдяки зрозумілій структурі меню, логічному розташуванню елементів керування та чітким підказкам. Усі ключові дії (створення транзакцій, формування бюджету, перегляд звітів) реалізовані у вигляді простих, повторюваних форм, що мінімізує потребу в навчанні.

Результати взаємодії з тестовими користувачами показали, що навіть особи без технічної підготовки можуть почати користуватись системою без зовнішньої допомоги, що свідчить про її високу зручність та інтуїтивність.

ВИСНОВКИ

У ході виконання дипломної роботи було повністю реалізовано повнофункціональну веб-платформу FamilyBudgeter, яка забезпечує ефективне спільне планування та контроль сімейних фінансів. Розроблена система об’єднує ключові можливості фінансового менеджменту: ведення доходів і витрат, створення бюджетів і лімітів, досягнення фінансових цілей, обробку регулярних платежів, генерацію звітів та аналітики — усе в межах зручного і безпечного багатокористувацького середовища.

Розробка платформи здійснювалась із дотриманням сучасних підходів до архітектури програмного забезпечення. Застосування принципів Clean Architecture, багаторівневої структури (N-Layer), шаблонів Repository та Unit of Work забезпечило високий рівень модульності, масштабованості та підтримуваності коду. Клієнтська частина, реалізована на React + TypeScript, є адаптивною, швидкою та інтуїтивно зрозумілою для користувача, що суттєво підвищує зручність роботи з платформою.

Платформа успішно пройшла тестування функціональності, продуктивності та зручності користування. Усі заплановані сценарії реалізовано повністю: система коректно обробляє дії користувача, швидко реагує на запити, виявляє перевищення лімітів, формує звіти та забезпечує високий рівень безпеки. Застосунок демонструє стабільну роботу навіть за умов активної взаємодії кількох користувачів одночасно.

Отримані результати підтверджують, що розроблений застосунок має високу прикладну цінність, повністю відповідає поставленим завданням та може використовуватися як у приватному середовищі, так і як основа для подальшого розвитку масштабованих фінансових сервісів. Робота відзначається комплексністю, технічною грамотністю реалізації та орієнтацією на реальні потреби сучасного користувача.

Результати кваліфікаційної роботи апробовано на конференціях:

1. Шремф М.О., Довженко Т.П., “Розробка платформи для спільного планування та контролю сімейних фінансів”. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.25, ДУІКТ, Київ, Україна, с. 113-114.
2. Шремф М.О., Довженко Т.П., “ Методи та засоби штучного інтелекту у розробці веб-застосунку для спільного планування та контролю сімейних фінансів ”. Матеріали V Всеукраїнської науково-практичної конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15.05.25, ДУІКТ, Київ, Україна, с. (подано до друку).

ПЕРЕЛІК ПОСИЛАНЬ

1. Popular Mint features. MintIntuit: веб-сайт. URL: https://mint.intuit.com/ (дата звернення: 03.03.2025)
2. How will you spend your money life. YNAB: веб-сайт. URL: https://www.ynab.com/ (дата звернення: 03.03.2025)
3. The only app that gets your money into shape. Spendee: веб-майт. URL: https://www.spendee.com/ (дата звернення: 03.03.2025)
4. ASP.NET documentation. Learn.microsoft : веб-сайт. URL: https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0 (дата звернення: 05.03.2025)
5. REST API як спосіб спілкування компонент веб-додатків. Foxminded: веб-сайт. URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення: 05.03.2025)
6. Introduction to JSON Web Tokens. JWT Debugger: веб-сайт. URL: https://jwt.io/introduction (дата звернення: 05.03.2025)
7. Entity Framework Core. Learn.microsoft: веб-сайт. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 06.03.2025)
8. EF Core Migrations. Learn Entity Framework Core: веб-сайт. URL: https://www.learnentityframeworkcore.com/migrations (дата звернення: 06.03.2025)
9. Microsoft SQL Server. Microsoft: веб-сайт. URL: https://www.microsoft.com/en-us/sql-server (дата звернення: 07.03.2025)
10. React The library for web and native user interfaces. React: веб-сайт. URL: https://react.dev/ (дата звернення: 07.03.2025)
11. Learn how to include React Bootstrap in your project. React Bootstrap: веб-сайт. URL: https://react-bootstrap.netlify.app/docs/getting-started/introduction (дата звернення: 08.03.2025)

12. Layered (N-Layer) Architecture. Medium: веб-сайт. URL: https://medium.com/design-microservices-architecture-with-patterns/layered-n-layer-architecture-e15ffdb7fa42 (дата звернення: 03.03.2025)
13. Lock, A. ASP.NET Core in Action, Third Edition. Manning Publications. 2023. 984 с.
14. Esposito, D. Programming ASP.NET Core (Developer Reference). Microsoft Press. 2020. 614 с.
15. Smith, J. P. Entity Framework Core in Action, Second Edition. Manning Publications. 2021. 624 с.
16. Rippon, C. Learn React with TypeScript: A Beginner's Guide to Reactive Web Development with React 18 and TypeScript. Packt Publishing. 2023. 474 с.
17. Goldberg, J. Learning TypeScript: Enhance Your Web Development Skills Using Type-Safe JavaScript. O'Reilly Media. 2022. 318 с.
18. Ben-Gan, I. T-SQL Fundamentals, 4th Edition. Microsoft Press. 2023. 608 с.
19. Richards, M., Ford, N. Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media. 2020. 422 с.
20. Bell, M. B. T. Software Architect. Wiley. 2023. 432 с.
21. Shaikh, A. Web Application Architecture: The Latest Guide 2025. *Peerbits*. 2024. URL: https://www.peerbits.com/blog/web-application-architecture.html
22. West R., Assaf W., Noble E., Longoria M., D'Antoni J., Davidson L.. SQL Server 2022 Administration Inside Out. Microsoft Press. 2023. 922 с.
23. Noback, M. Advanced Web Application Architecture. Leanpub. 2020. 539 p.
24. Ford, N., Richards, M., Sadalage, P., Dehghani, Z.. Software Architecture: The Hard Parts. O’Reilly Media. 2021. 464 с.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка платформи FamilyBudgeter для спільного планування та контролю сімейних фінансів

Виконала студентка 4 курсу
Групи ПД-42
Шремф Милана Олександрівна
Керівник роботи
доцент кафедри ІПЗ Довженко Тимур Павлович,

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи — покращення цифрового контролю сімейного бюджету з можливістю ведення транзакцій, встановлення бюджетів, створення фінансових цілей та формування аналітичних звітів.

Об’єкт дослідження — процес цифрового управління сімейними фінансами в умовах багато користувацького доступу.

Предмет дослідження — веб-застосунок для спільного планування бюджету, обліку доходів і витрат, візуалізації фінансових потоків і досягнення фінансових цілей.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати сучасні підходи до планування сімейного бюджету та оглянути популярні програмні аналоги.
2. Розробити архітектуру системи FamilyBudgeter.
3. Реалізувати серверну частину.
4. Розробити клієнтську частину застосунку.
5. Забезпечити підтримку функцій: управління сім’ями, транзакціями, бюджетами, фінансовими цілями та регулярними платежами.
6. Реалізувати модулі аналітики та візуалізації фінансових показників.
7. Провести тестування системи на предмет чи відповідає вимогам.

3

АНАЛІЗ АНАЛОГІВ

Критерій	Mint	YNAB	Spendee	FamilyBudgeter
Спільне використання	-	-	+	+
Підтримка ролей	-	-	-	+
Фінансові цілі	+	+	+	+
Регулярні платежі	-	-	+	+
Аналітика	+	+	-	+
Гнучкість налаштувань	-	+	-	+
Мовна та валютна адаптація	-	-	-	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги до системи

- Реєстрація користувача, створення сім'ї та запрошення учасників через код.
- Призначення ролей (адміністратор, учасник, обмежений доступ) з відповідними правами.
- Додавання, редагування, фільтрація транзакцій за категоріями, типами, датами.
- Встановлення бюджетних лімітів, контроль перевищень.
- Створення фінансових цілей, відстеження їх прогресу.
- Налаштування регулярних платежів і автоматичне генерування транзакцій.
- Побудова звітів і графіків (доходи, витрати, бюджети, цілі).
- Сповіщення про важливі події (ліміти, пропуски, досягнення).

Нефункціональні вимоги

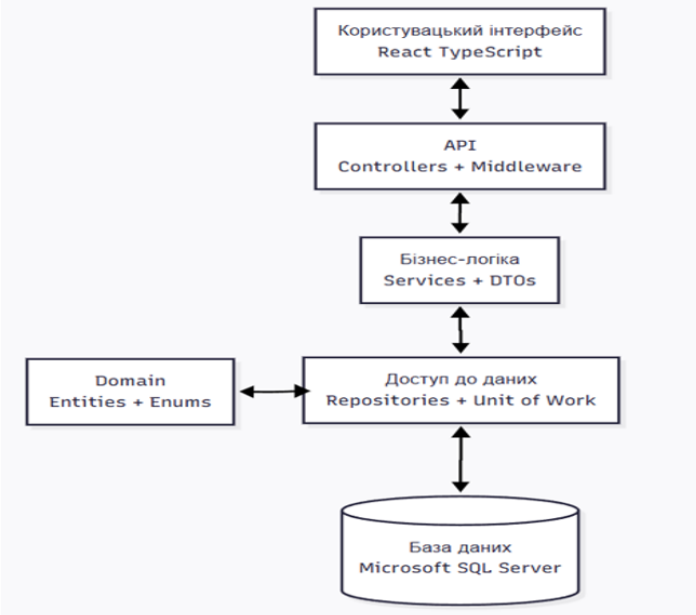
- Інтерфейс: адаптивний, зрозумілий, компонентний дизайн на React.
- Безпека: авторизація через JWT, контроль доступу за ролями, перевірка прав.
- Масштабованість: Clean Architecture, N-Layer, гнучке розширення функцій.
- Надійність: обробка помилок, валідація, логування, стабільність при збоях.
- Продуктивність: швидка обробка запитів (до 500 мс), оптимізація запитів, кешування.
- Локалізація: підтримка української мови, гривні, адаптовані формати.
- Якість коду: дотримання SOLID, ізоляція бізнес-логіки, підтримуваність коду.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

6

СХЕМА АРХІТЕКТУРИ СИСТЕМИ

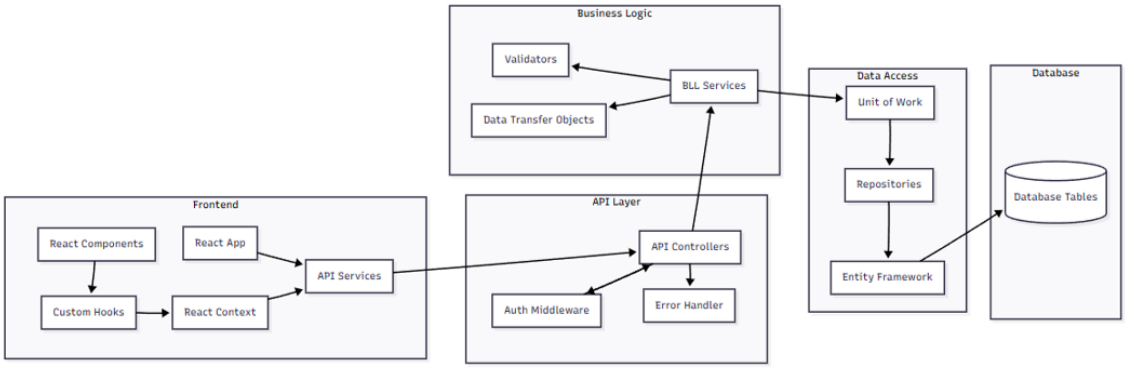


7

ДІАГРАМА СТРУКТУРИ БАЗИ ДАНИХ СИСТЕМИ

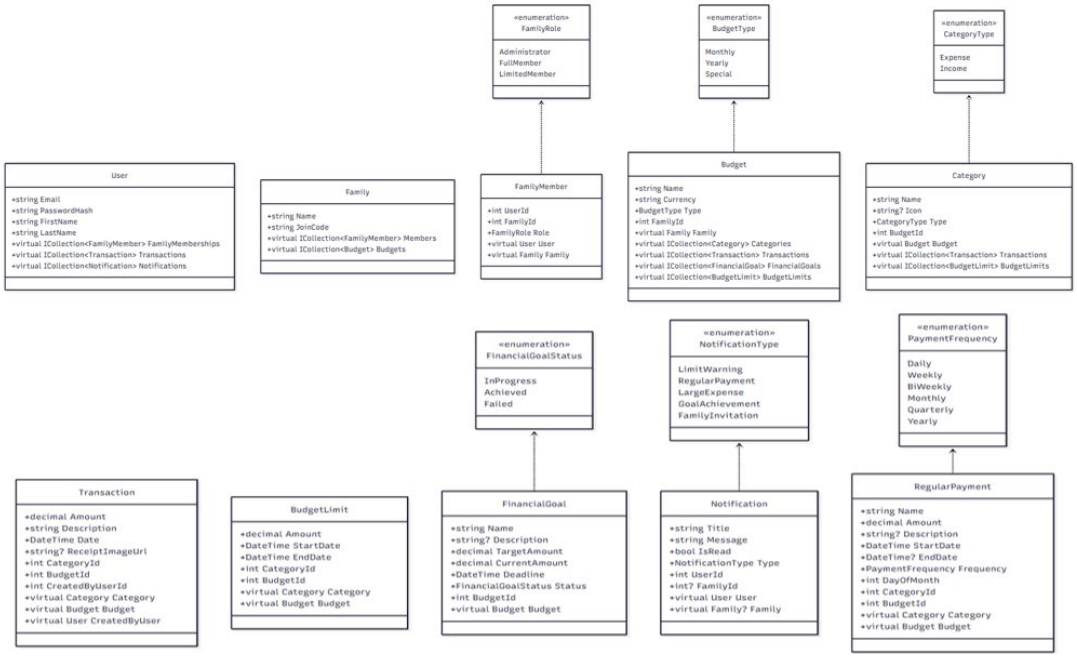
8

ДІАГРАМА ВЗАЄМОДІЇ КОМПОНЕТНІВ



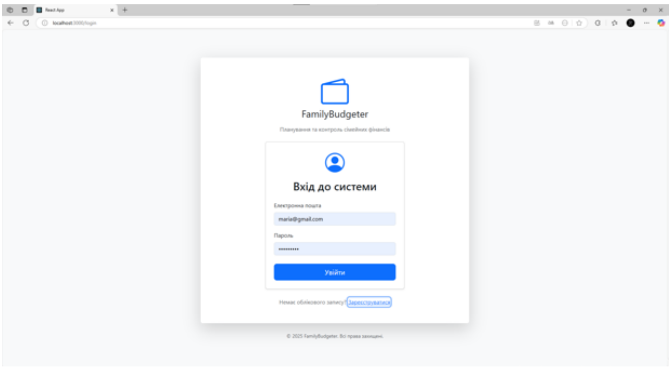
9

ДІАГРАМА ОСНОВНИХ КЛАСІВ

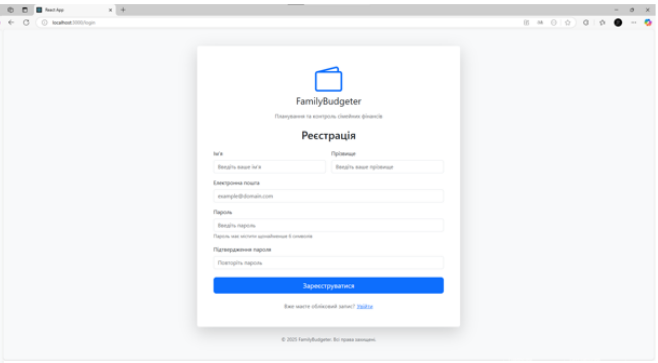


10

ЕКРАННІ ФОРМИ

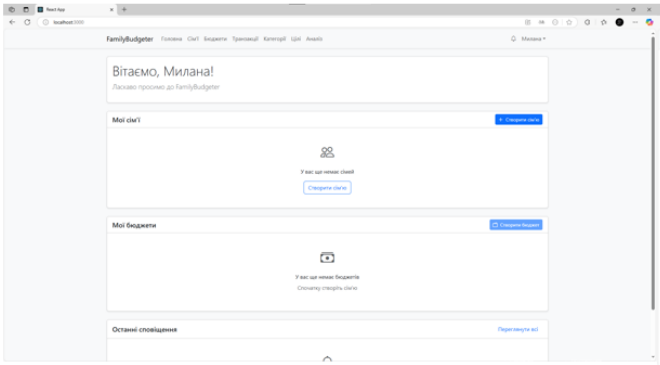


Сторінка входу в систему

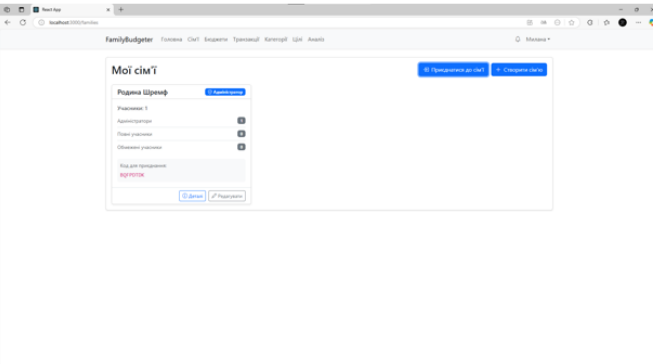


Форма реєстрації нового користувача

ЕКРАННІ ФОРМИ

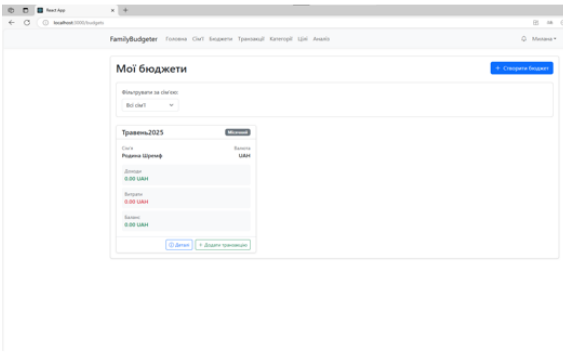


Головна панель

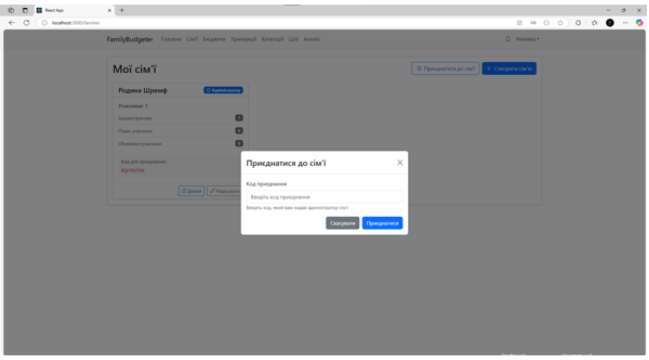


Сторінка управління сім'єю

ЕКРАННІ ФОРМИ



Список сімейних бюджетів



Форма приєднання до сім'ї

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Шремф М.О., Довженко Т.П., “Розробка платформи для спільного планування та контролю сімейних фінансів”. Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 24.04.25, ДУІКТ, Київ, Україна, с. 113-114.
2. Шремф М.О., Довженко Т.П., “ Методи та засоби штучного інтелекту у розробці веб-застосунку для спільного планування та контролю сімейних фінансів ”. Матеріали V Всеукраїнської науково-практичної конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15.05.25, ДУІКТ, Київ, Україна, с. (подано до друку).

ВИСНОВКИ

1. Було здійснено дослідження аналогів, глибокий аналіз сучасних інструментів розробки та технологій для планування сімейного бюджету, що дозволило визначити актуальні потреби користувачів у цій сфері та сформулювати обґрунтовані функціональні вимоги до майбутнього веб-застосунку.
2. Розроблено архітектуру системи FamilyBudgeter на основі підходів Clean Architecture та багаторівневої структури. Це забезпечило модульність, гнучкість та легкість у подальшому супроводі й розширенні системи .
3. Реалізовано серверну частину на платформі ASP.NET Core з використанням Entity Framework Core і СКБД Microsoft SQL Server, що гарантує надійність обробки фінансових даних, масштабованість та продуктивність застосунку .
4. Клієнтська частина створена з використанням React та TypeScript, що дозволило забезпечити високий рівень інтерактивності, адаптивний дизайн і зручність користування застосунком.
5. Система підтримує створення сімейних груп, управління правами доступу учасників, ведення транзакцій, створення бюджетів, фінансових цілей та автоматизованих регулярних платежів, а також надає гнучку аналітику з візуалізацією фінансової інформації .
6. Проведене функціональне тестування підтвердило стабільну роботу, високу продуктивність та ефективність розробленого застосунку в умовах реального використання.
7. Веб-застосунок FamilyBudgeter може бути рекомендований для впровадження у домогосподарствах з метою покращення прозорості ведення фінансів, підвищення фінансової грамотності та досягнення спільних фінансових цілей.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
using FamilyBudgeter.API.BLL.DTOs.BudgetDTOs;
using FamilyBudgeter.API.BLL.DTOs.CategoryDTOs;
using FamilyBudgeter.API.BLL.Interfaces;
using FamilyBudgeter.API.DAL.Interfaces;
using FamilyBudgeter.API.Domain.Enums;

namespace FamilyBudgeter.API.BLL.Services
{
  public class AnalysisService : IAnalysisService
  {
    private readonly IUnitOfWork _unitOfWork;
    private readonly IBudgetService _budgetService;

    public AnalysisService(IUnitOfWork unitOfWork, IBudgetService budgetService)
    {
      _unitOfWork = unitOfWork;
      _budgetService = budgetService;
    }

    public async Task<BudgetSummaryDto> GetBudgetSummaryAsync(int budgetId, DateTime startDate, DateTime endDate, int userId)
    {
      // Перевірка доступу
      if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
      {
        throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
      }

      var budget = await _unitOfWork.Budgets.GetByIdAsync(budgetId);
      if (budget == null)
      {
        throw new KeyNotFoundException("Бюджет не знайдено");
      }

      var transactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, startDate, endDate);
      var categories = await _unitOfWork.Categories.GetByBudgetIdAsync(budgetId);

      var income = transactions
        .Where(t => categories.Any(c => c.Id == t.CategoryId && c.Type == CategoryType.Income))
        .Sum(t => t.Amount);

      var expense = transactions
        .Where(t => categories.Any(c => c.Id == t.CategoryId && c.Type == CategoryType.Expense))
        .Sum(t => t.Amount);

      return new BudgetSummaryDto
      {
        Id = budget.Id,
        Name = budget.Name,
        Currency = budget.Currency,
        TotalIncome = income,
        TotalExpense = expense,
        Balance = income - expense,
        StartDate = startDate,
        EndDate = endDate
      };
    }
```

```
        public async Task<IEnumerable<CategorySummaryDto>> GetCategorySummariesAsync(int budgetId, DateTime startDate, DateTime endDate, int userId)
        {
            // Перевірка доступу
            if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
            {
                throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
            }

            var categories = await _unitOfWork.Categories.GetByBudgetIdAsync(budgetId);
            var transactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, startDate, endDate);
            var budgetLimits = await _unitOfWork.BudgetLimits.GetByBudgetIdAsync(budgetId);

            var summaries = new List<CategorySummaryDto>();
            foreach (var category in categories)
            {
                var categoryTransactions = transactions.Where(t => t.CategoryId == category.Id).ToList();
                var amount = categoryTransactions.Sum(t => t.Amount);

                // Знаходимо активний ліміт для категорії
                var activeLimit = budgetLimits
                    .FirstOrDefault(l => l.CategoryId == category.Id &&
                                         l.StartDate <= endDate &&
                                         l.EndDate >= startDate);

                var summary = new CategorySummaryDto
                {
                    Id = category.Id,
                    Name = category.Name,
                    Icon = category.Icon,
                    Type = category.Type,
                    Amount = amount,
                    Limit = activeLimit?.Amount,
                    PercentOfLimit = activeLimit != null ? CalculatePercentage(amount, activeLimit.Amount) : null,
                    TransactionsCount = categoryTransactions.Count
                };

                summaries.Add(summary);
            }

            return summaries;
        }

        public async Task<Dictionary<DateTime, decimal>> GetExpenseTrendAsync(int budgetId, DateTime startDate, DateTime endDate, string groupBy, int userId)
        {
            // Перевірка доступу
            if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
            {
                throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
            }
```

```
            var categories = await
_unitOfWork.Categories.GetByBudgetIdAsync(budgetId
);
            var expenseCategories =
categories.Where(c => c.Type ==
CategoryType.Expense).Select(c => c.Id).ToList();
            var transactions = await
_unitOfWork.Transactions.GetByDateRangeAsync(budg
etId, startDate, endDate);
            var expenseTransactions =
transactions.Where(t =>
expenseCategories.Contains(t.CategoryId));

            var trend = new Dictionary<DateTime,
decimal>();

            switch (groupBy.ToLower())
            {
                case "day":
                    trend = expenseTransactions
                        .GroupBy(t => t.Date.Date)
                        .ToDictionary(g => g.Key, g =>
g.Sum(t => t.Amount));
                    break;

                case "week":
                    trend = expenseTransactions
                        .GroupBy(t => t.Date.AddDays(-
(int)t.Date.DayOfWeek).Date)
                        .ToDictionary(g => g.Key, g =>
g.Sum(t => t.Amount));
                    break;

                case "month":
                    trend = expenseTransactions
                        .GroupBy(t => new
DateTime(t.Date.Year, t.Date.Month, 1))
                        .ToDictionary(g => g.Key, g =>
g.Sum(t => t.Amount));
                    break;

                case "year":
                    trend = expenseTransactions
                        .GroupBy(t => new
DateTime(t.Date.Year, 1, 1))
                        .ToDictionary(g => g.Key, g =>
g.Sum(t => t.Amount));
                    break;

                default:
                    throw new
ArgumentException("Невірний параметр групування.
Підтримується: day, week, month, year");
            }

            return trend;
        }

        public async Task<Dictionary<DateTime,
decimal>> GetIncomeTrendAsync(int budgetId,
DateTime startDate, DateTime endDate, string groupBy,
int userId)
        {
            // Перевірка доступу
            if (!await
_budgetService.HasUserAccessToBudgetAsync(budgetI
d, userId))
            {
                throw new
UnauthorizedAccessException("У вас немає доступу до
цього бюджету");
            }

            var categories = await
_unitOfWork.Categories.GetByBudgetIdAsync(budgetId
);
            var incomeCategories =
categories.Where(c => c.Type ==
CategoryType.Income).Select(c => c.Id).ToList();
            var transactions = await
_unitOfWork.Transactions.GetByDateRangeAsync(budg
etId, startDate, endDate);
```

```
            var incomeTransactions = transactions.Where(t => incomeCategories.Contains(t.CategoryId));

            var trend = new Dictionary<DateTime, decimal>();

            switch (groupBy.ToLower())
            {
                case "day":
                    trend = incomeTransactions
                        .GroupBy(t => t.Date.Date)
                        .ToDictionary(g => g.Key, g => g.Sum(t => t.Amount));
                    break;

                case "week":
                    trend = incomeTransactions
                        .GroupBy(t => t.Date.AddDays(-(int)t.Date.DayOfWeek).Date)
                        .ToDictionary(g => g.Key, g => g.Sum(t => t.Amount));
                    break;

                case "month":
                    trend = incomeTransactions
                        .GroupBy(t => new DateTime(t.Date.Year, t.Date.Month, 1))
                        .ToDictionary(g => g.Key, g => g.Sum(t => t.Amount));
                    break;

                case "year":
                    trend = incomeTransactions
                        .GroupBy(t => new DateTime(t.Date.Year, 1, 1))
                        .ToDictionary(g => g.Key, g => g.Sum(t => t.Amount));
                    break;

                default:
                    throw new ArgumentException("Невірний параметр групування. Підтримується: day, week, month, year");
            }

            return trend;
        }

        public async Task<Dictionary<string, object>> CompareBudgetWithActualAsync(int budgetId, DateTime startDate, DateTime endDate, int userId)
        {
            // Перевірка доступу
            if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
            {
                throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
            }

            var categories = await _unitOfWork.Categories.GetByBudgetIdAsync(budgetId);
            var transactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, startDate, endDate);
            var budgetLimits = await _unitOfWork.BudgetLimits.GetByBudgetIdAsync(budgetId);

            var result = new Dictionary<string, object>();
            var categoryComparisons = new List<object>();

            foreach (var category in categories.Where(c => c.Type == CategoryType.Expense))
            {
```

```
                var actual = transactions.Where(t => t.CategoryId == category.Id).Sum(t => t.Amount);
                var limits = budgetLimits.Where(l => l.CategoryId == category.Id &&
                                                l.StartDate <= endDate &&
                                                l.EndDate >= startDate);
                var budgeted = limits.Sum(l => l.Amount);

                categoryComparisons.Add(new
                {
                    CategoryName = category.Name,
                    Budgeted = budgeted,
                    Actual = actual,
                    Difference = budgeted - actual,
                    PercentageUsed = budgeted > 0 ? CalculatePercentage(actual, budgeted) : 0
                });
            }

            result["categories"] = categoryComparisons;
            result["totalBudgeted"] = categoryComparisons.Sum(c => ((dynamic)c).Budgeted);
            result["totalActual"] = categoryComparisons.Sum(c => ((dynamic)c).Actual);
            result["totalDifference"] = (decimal)result["totalBudgeted"] - (decimal)result["totalActual"];

            return result;
        }

        public async Task<Dictionary<string, decimal>> ForecastExpensesAsync(int budgetId, int months, int userId)
        {
            // Перевірка доступу
            if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
            {
                throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
            }

            var endDate = DateTime.UtcNow;
            var startDate = endDate.AddMonths(-3); // Аналізуємо останні 3 місяці для прогнозу

            var categories = await _unitOfWork.Categories.GetByBudgetIdAsync(budgetId);
            var expenseCategories = categories.Where(c => c.Type == CategoryType.Expense).ToList();
            var transactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, startDate, endDate);

            var forecast = new Dictionary<string, decimal>();

            foreach (var category in expenseCategories)
            {
                var categoryTransactions = transactions.Where(t => t.CategoryId == category.Id).ToList();
                if (!categoryTransactions.Any())
                {
                    forecast[category.Name] = 0;
                    continue;
                }

                // Обчислюємо середній місячний витрат
                var monthlyExpenses = categoryTransactions
```

```
                .GroupBy(t => new { t.Date.Year, t.Date.Month })
                .Select(g => g.Sum(t => t.Amount))
                .ToList();

            var averageMonthly = monthlyExpenses.Any() ? monthlyExpenses.Average() : 0;
            forecast[category.Name] = averageMonthly * months;
        }

        forecast["Total"] = forecast.Values.Sum();
        return forecast;
    }

    public async Task<Dictionary<string, decimal>> AnalyzeTopExpensesAsync(int budgetId, DateTime startDate, DateTime endDate, int limit, int userId)
    {
        // Перевірка доступу
        if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
        {
            throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
        }

        var categories = await _unitOfWork.Categories.GetByBudgetIdAsync(budgetId);
        var expenseCategories = categories.Where(c => c.Type == CategoryType.Expense).ToList();
        var transactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, startDate, endDate);

        var topExpenses = new Dictionary<string, decimal>();

        foreach (var category in expenseCategories)
        {
            var categoryTotal = transactions
                .Where(t => t.CategoryId == category.Id)
                .Sum(t => t.Amount);

            if (categoryTotal > 0)
            {
                topExpenses[category.Name] = categoryTotal;
            }
        }

        // Повертаємо топ N категорій за витратами
        return topExpenses
            .OrderByDescending(e => e.Value)
            .Take(limit)
            .ToDictionary(e => e.Key, e => e.Value);
    }

    public async Task<Dictionary<string, decimal>> AnalyzeTopIncomesAsync(int budgetId, DateTime startDate, DateTime endDate, int limit, int userId)
    {
        // Перевірка доступу
        if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
        {
            throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
        }
```

```
            var categories = await
_unitOfWork.Categories.GetByBudgetIdAsync(budgetId
);
            var incomeCategories =
categories.Where(c => c.Type ==
CategoryType.Income).ToList();
            var transactions = await
_unitOfWork.Transactions.GetByDateRangeAsync(budg
etId, startDate, endDate);

            var topIncomes = new Dictionary<string,
decimal>();

            foreach (var category in
incomeCategories)
            {
                var categoryTotal = transactions
                    .Where(t => t.CategoryId ==
category.Id)
                    .Sum(t => t.Amount);

                if (categoryTotal > 0)
                {
                    topIncomes[category.Name] =
categoryTotal;
                }
            }

            // Повертаємо топ N категорій за
доходами
            return topIncomes
                .OrderByDescending(i => i.Value)
                .Take(limit)
                .ToDictionary(i => i.Key, i =>
i.Value);
        }

        public async Task<Dictionary<string,
decimal>> AnalyzeExpensesByUserAsync(int budgetId,
DateTime startDate, DateTime endDate, int userId)
        {
            // Перевірка доступу
            if (!await
_budgetService.HasUserAccessToBudgetAsync(budgetI
d, userId))
            {
                throw new
UnauthorizedAccessException("У вас немає доступу до
цього бюджету");
            }

            var categories = await
_unitOfWork.Categories.GetByBudgetIdAsync(budgetId
);
            var expenseCategories =
categories.Where(c => c.Type ==
CategoryType.Expense).Select(c => c.Id).ToList();
            var transactions = await
_unitOfWork.Transactions.GetByDateRangeAsync(budg
etId, startDate, endDate);
            var expenseTransactions =
transactions.Where(t =>
expenseCategories.Contains(t.CategoryId));

            var expensesByUser =
expenseTransactions
                .GroupBy(t => t.CreatedByUserId)
                .Select(g => new
                {
                    UserId = g.Key,
                    Total = g.Sum(t => t.Amount)
                })
                .ToList();

            var result = new Dictionary<string,
decimal>();

            foreach (var expense in expensesByUser)
            {
                var user = await
_unitOfWork.Users.GetByIdAsync(expense.UserId);
                if (user != null)
                {
```

```
                result[$"{user.FirstName} {user.LastName}"] = expense.Total;
            }
        }

        return result;
    }

    public async Task<Dictionary<string, object>> AnalyzeExpenseChangesAsync(int budgetId, DateTime startDate, DateTime endDate, int userId)
    {
        // Перевірка доступу
        if (!await _budgetService.HasUserAccessToBudgetAsync(budgetId, userId))
        {
            throw new UnauthorizedAccessException("У вас немає доступу до цього бюджету");
        }

        var categories = await _unitOfWork.Categories.GetByBudgetIdAsync(budgetId);
        var expenseCategories = categories.Where(c => c.Type == CategoryType.Expense).ToList();
        var transactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, startDate, endDate);

        // Порівнюємо з попереднім періодом
        var periodLength = (endDate - startDate).Days;
        var previousEndDate = startDate.AddDays(-1);
        var previousStartDate = previousEndDate.AddDays(-periodLength);
        var previousTransactions = await _unitOfWork.Transactions.GetByDateRangeAsync(budgetId, previousStartDate, previousEndDate);

        var result = new Dictionary<string, object>();
        var categoryChanges = new List<object>();

        foreach (var category in expenseCategories)
        {
            var currentPeriodTotal = transactions
                .Where(t => t.CategoryId == category.Id)
                .Sum(t => t.Amount);

            var previousPeriodTotal = previousTransactions
                .Where(t => t.CategoryId == category.Id)
                .Sum(t => t.Amount);

            var change = currentPeriodTotal - previousPeriodTotal;
            var changePercent = previousPeriodTotal > 0
                ? CalculatePercentage(change, previousPeriodTotal)
                : (currentPeriodTotal > 0 ? 100 : 0);

            categoryChanges.Add(new
            {
                CategoryName = category.Name,
                CurrentPeriod = currentPeriodTotal,
                PreviousPeriod = previousPeriodTotal,
                Change = change,
                ChangePercent = changePercent
            });
        }

        result["categoryChanges"] = categoryChanges;
```

```
                result["totalCurrentPeriod"] =
transactions
                    .Where(t => expenseCategories.Any(c
=> c.Id == t.CategoryId))
                    .Sum(t => t.Amount);
                result["totalPreviousPeriod"] =
previousTransactions
                    .Where(t => expenseCategories.Any(c
=> c.Id == t.CategoryId))
                    .Sum(t => t.Amount);
                result["totalChange"] =
(decimal)result["totalCurrentPeriod"] -
(decimal)result["totalPreviousPeriod"];
                result["totalChangePercent"] =
(decimal)result["totalPreviousPeriod"] > 0
                    ?
CalculatePercentage((decimal)result["totalChange"],
(decimal)result["totalPreviousPeriod"])
                    :
((decimal)result["totalCurrentPeriod"] > 0 ? 100 : 0);

                return result;
            }

            #region Helper Methods

            private decimal
CalculatePercentage(decimal value, decimal total)
            {
                if (total == 0) return 0;
                return Math.Round((value / total) * 100,
2);
            }

            #endregion
        }
    }
    using
FamilyBudgeter.API.BLL.DTOs.AuthDTOs;
    using FamilyBudgeter.API.BLL.Interfaces;
    using FamilyBudgeter.API.DAL.Interfaces;
    using FamilyBudgeter.API.Domain.Entities;
    using Microsoft.IdentityModel.Tokens;
    using System.IdentityModel.Tokens.Jwt;
    using System.Security.Claims;
    using System.Security.Cryptography;
    using System.Text;

    namespace FamilyBudgeter.API.BLL.Services
    {
        public class AuthService : IAuthService
        {
            private readonly IUnitOfWork
_unitOfWork;
            private readonly IConfiguration
_configuration;

            public AuthService(IUnitOfWork
unitOfWork, IConfiguration configuration)
            {
                _unitOfWork = unitOfWork;
                _configuration = configuration;
            }

            public async Task<AuthResponseDto>
RegisterAsync(UserRegistrationDto registrationDto)
            {
                try
                {
                    // Перевірка чи існує користувач з
такою електронною поштою
                    var existingUser = await
_unitOfWork.Users.GetByEmailAsync(registrationDto.E
mail);
                    if (existingUser != null)
                    {
                        return new AuthResponseDto
                        {
                            Success = false,
                            Message = "Користувач з такою
електронною поштою вже існує"
                        };
                    }
```

```
            // Створення нового користувача
            var user = new User
            {
                Email = registrationDto.Email,
                PasswordHash = HashPassword(registrationDto.Password),
                FirstName = registrationDto.FirstName,
                LastName = registrationDto.LastName
            };

            await _unitOfWork.Users.AddAsync(user);
            await _unitOfWork.SaveChangesAsync();

            // Генерація токену
            var token = GenerateJwtToken(user);
            var refreshToken = GenerateRefreshToken();

            // Повернення результату
            return new AuthResponseDto
            {
                Success = true,
                Token = token,
                RefreshToken = refreshToken,
                Expiration = DateTime.UtcNow.AddMinutes(Convert.ToDouble(_configuration["JwtSettings:ExpiryInMinutes"] ?? "60")),
                User = new UserDto
                {
                    Id = user.Id,
                    Email = user.Email,
                    FirstName = user.FirstName,
                    LastName = user.LastName
                }
            };
        }
        catch (Exception ex)
        {
            return new AuthResponseDto
            {
                Success = false,
                Message = $"Помилка при реєстрації: {ex.Message}"
            };
        }
    }

    public async Task<AuthResponseDto> LoginAsync(UserLoginDto loginDto)
    {
        try
        {
            // Пошук користувача за електронною поштою
            var user = await _unitOfWork.Users.GetByEmailAsync(loginDto.Email);
            if (user == null)
            {
                return new AuthResponseDto
                {
                    Success = false,
                    Message = "Користувач з такою електронною поштою не знайдений"
                };
            }

            // Перевірка пароля
            if (!VerifyPassword(loginDto.Password, user.PasswordHash))
            {
                return new AuthResponseDto
                {
                    Success = false,
                    Message = "Невірний пароль"
                };
            }

            // Генерація токену
            var token = GenerateJwtToken(user);
```

```
                var refreshToken = GenerateRefreshToken();

                // Повернення результату
                return new AuthResponseDto
                {
                    Success = true,
                    Token = token,
                    RefreshToken = refreshToken,
                    Expiration = DateTime.UtcNow.AddMinutes(Convert.ToDouble(_configuration["JwtSettings:ExpiryInMinutes"] ?? "60")),
                    User = new UserDto
                    {
                        Id = user.Id,
                        Email = user.Email,
                        FirstName = user.FirstName,
                        LastName = user.LastName
                    }
                };
            }
            catch (Exception ex)
            {
                return new AuthResponseDto
                {
                    Success = false,
                    Message = $"Помилка при авторизації: {ex.Message}"
                };
            }
        }

        public async Task<AuthResponseDto> RefreshTokenAsync(string token, string refreshToken)
        {
            try
            {
                var principal = GetPrincipalFromExpiredToken(token);
                if (principal == null)
                {
                    return new AuthResponseDto
                    {
                        Success = false,
                        Message = "Недійсний або прострочений токен"
                    };
                }

                var userIdClaim = principal.FindFirst(ClaimTypes.NameIdentifier);
                if (userIdClaim == null || !int.TryParse(userIdClaim.Value, out int userId))
                {
                    return new AuthResponseDto
                    {
                        Success = false,
                        Message = "Недійсний токен"
                    };
                }

                var user = await _unitOfWork.Users.GetByIdAsync(userId);
                if (user == null)
                {
                    return new AuthResponseDto
                    {
                        Success = false,
                        Message = "Користувач не знайдений"
                    };
                }

                var newToken = GenerateJwtToken(user);
                var newRefreshToken = GenerateRefreshToken();

                return new AuthResponseDto
                {
                    Success = true,
                    Token = newToken,
                    RefreshToken = newRefreshToken,
```

```
                Expiration =
DateTime.UtcNow.AddMinutes(Convert.ToDouble(_con
figuration["JwtSettings:ExpiryInMinutes"] ?? "60")),
                User = new UserDto
                {
                  Id = user.Id,
                  Email = user.Email,
                  FirstName = user.FirstName,
                  LastName = user.LastName
                }
              };
            }
            catch (Exception ex)
            {
              return new AuthResponseDto
              {
                Success = false,
                Message = $"Помилка при
оновленні токену: {ex.Message}"
              };
            }
          }

          public async Task<UserDto>
GetCurrentUserAsync(int userId)
          {
            var user = await
_unitOfWork.Users.GetByIdAsync(userId);
            if (user == null)
            {
              throw new
KeyNotFoundException("Користувач не знайдений");
            }

            return new UserDto
            {
              Id = user.Id,
              Email = user.Email,
              FirstName = user.FirstName,
              LastName = user.LastName
            };
          }

          public async Task<UserDto>
UpdateUserAsync(int userId, UserDto userDto)
          {
            var user = await
_unitOfWork.Users.GetByIdAsync(userId);
            if (user == null)
            {
              throw new
KeyNotFoundException("Користувач не знайдений");
            }

            // Перевірка, чи змінилася електронна
пошта
            if (user.Email != userDto.Email)
            {
              // Перевірка, чи існує користувач з
такою електронною поштою
              var existingUser = await
_unitOfWork.Users.GetByEmailAsync(userDto.Email);
              if (existingUser != null &&
existingUser.Id != userId)
              {
                throw new
InvalidOperationException("Користувач з такою
електронною поштою вже існує");
              }

              user.Email = userDto.Email;
            }

            user.FirstName = userDto.FirstName;
            user.LastName = userDto.LastName;
            user.UpdatedAt = DateTime.UtcNow;

            _unitOfWork.Users.Update(user);
            await
_unitOfWork.SaveChangesAsync();

            return userDto;
          }
```

```
        public async Task<bool> ChangePasswordAsync(int userId, string currentPassword, string newPassword)
        {
            var user = await _unitOfWork.Users.GetByIdAsync(userId);
            if (user == null)
            {
                throw new KeyNotFoundException("Користувач не знайдений");
            }

            // Перевірка поточного пароля
            if (!VerifyPassword(currentPassword, user.PasswordHash))
            {
                throw new InvalidOperationException("Поточний пароль невірний");
            }

            // Оновлення пароля
            user.PasswordHash = HashPassword(newPassword);
            user.UpdatedAt = DateTime.UtcNow;

            _unitOfWork.Users.Update(user);
            await _unitOfWork.SaveChangesAsync();

            return true;
        }

        #region Helper Methods

        private string HashPassword(string password)
        {
            using var sha256 = SHA256.Create();
            var hashedBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(password));
            return Convert.ToBase64String(hashedBytes);
        }

        private bool VerifyPassword(string password, string passwordHash)
        {
            // Для спрощення використаємо простий хеш-алгоритм
            var hashedPassword = HashPassword(password);
            return hashedPassword == passwordHash;
        }

        private string GenerateJwtToken(User user)
        {
            var jwtSettings = _configuration.GetSection("JwtSettings");
            var key = Encoding.ASCII.GetBytes(jwtSettings["Secret"] ?? "defaultsecretkey12345678901234567890");

            var tokenHandler = new JwtSecurityTokenHandler();
            var tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(new[]
                {
                    new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
                    new Claim(ClaimTypes.Email, user.Email),
                    new Claim(ClaimTypes.GivenName, user.FirstName),
                    new Claim(ClaimTypes.Surname, user.LastName)
                }),
                Expires = DateTime.UtcNow.AddMinutes(Convert.ToDouble(jwtSettings["ExpiryInMinutes"] ?? "60")),
```

```
            SigningCredentials = new
SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature),
            Issuer = jwtSettings["Issuer"],
            Audience = jwtSettings["Audience"]
          };

          var token =
tokenHandler.CreateToken(tokenDescriptor);
          return tokenHandler.WriteToken(token);
        }

        private string GenerateRefreshToken()
        {
          var randomNumber = new byte[32];
          using var rng =
RandomNumberGenerator.Create();
          rng.GetBytes(randomNumber);
          return
Convert.ToBase64String(randomNumber);
        }

        private ClaimsPrincipal?
GetPrincipalFromExpiredToken(string token)
        {
          var jwtSettings =
_configuration.GetSection("JwtSettings");
          var key =
Encoding.ASCII.GetBytes(jwtSettings["Secret"] ??
"defaultsecretkey12345678901234567890");

          var tokenValidationParameters = new
TokenValidationParameters
          {
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new
SymmetricSecurityKey(key),
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidIssuer = jwtSettings["Issuer"],
            ValidAudience =
jwtSettings["Audience"],
            ValidateLifetime = false //
Дозволяємо прострочені токени
          };

          var tokenHandler = new
JwtSecurityTokenHandler();
          SecurityToken securityToken;
          var principal =
tokenHandler.ValidateToken(token,
tokenValidationParameters, out securityToken);

          if (securityToken is not
JwtSecurityToken jwtSecurityToken ||

!jwtSecurityToken.Header.Alg.Equals(SecurityAlgorith
ms.HmacSha256,
StringComparison.InvariantCultureIgnoreCase))
          {
            return null;
          }

          return principal;
        }

        #endregion
      }
    }
    using FamilyBudgeter.API.Domain.Entities;
    using Microsoft.AspNetCore.Mvc;
    using System.Security.Claims;

    namespace FamilyBudgeter.API.Controllers
    {
      [ApiController]
      [Route("api/[controller]")]
      public abstract class BaseApiController :
ControllerBase
      {
        protected int GetCurrentUserId()
        {
          var userIdClaim =
User.FindFirst(ClaimTypes.NameIdentifier);
```

```
            if (userIdClaim == null || !int.TryParse(userIdClaim.Value, out int userId))
            {
                throw new UnauthorizedAccessException("Користувач не автентифікований");
            }
            return userId;
        }

        protected IActionResult HandleException(Exception ex)
        {
            if (ex is KeyNotFoundException)
                return NotFound(new { message = ex.Message });

            if (ex is UnauthorizedAccessException)
                return Unauthorized(new { message = ex.Message });

            if (ex is InvalidOperationException)
                return BadRequest(new { message = ex.Message });

            if (ex is ArgumentException)
                return BadRequest(new { message = ex.Message });

            return StatusCode(500, new { message = "Виникла внутрішня помилка сервера" });
        }
    }
}
using FamilyBudgeter.API.BLL.Interfaces;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace FamilyBudgeter.API.Controllers
{
    [Authorize]
    public class AnalysisController : BaseApiController
    {
        private readonly IAnalysisService _analysisService;

        public AnalysisController(IAnalysisService analysisService)
        {
            _analysisService = analysisService;
        }

        [HttpGet("budget/{budgetId}/summary")]
        public async Task<IActionResult> GetBudgetSummary(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate)
        {
            try
            {
                var userId = GetCurrentUserId();
                var summary = await _analysisService.GetBudgetSummaryAsync(budgetId, startDate, endDate, userId);
                return Ok(summary);
            }
            catch (Exception ex)
            {
                return HandleException(ex);
            }
        }

        [HttpGet("budget/{budgetId}/category-summaries")]
        public async Task<IActionResult> GetCategorySummaries(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate)
        {
            try
            {
                var userId = GetCurrentUserId();
```

```
            var summaries = await _analysisService.GetCategorySummariesAsync(budgetId, startDate, endDate, userId);
            return Ok(summaries);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/expense-trend")]
    public async Task<IActionResult> GetExpenseTrend(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate, [FromQuery] string groupBy = "month")
    {
        try
        {
            var userId = GetCurrentUserId();
            var trend = await _analysisService.GetExpenseTrendAsync(budgetId, startDate, endDate, groupBy, userId);
            return Ok(trend);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/income-trend")]
    public async Task<IActionResult> GetIncomeTrend(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate, [FromQuery] string groupBy = "month")
    {
        try
        {
            var userId = GetCurrentUserId();
            var trend = await _analysisService.GetIncomeTrendAsync(budgetId, startDate, endDate, groupBy, userId);
            return Ok(trend);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/compare-with-actual")]
    public async Task<IActionResult> CompareBudgetWithActual(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate)
    {
        try
        {
            var userId = GetCurrentUserId();
            var comparison = await _analysisService.CompareBudgetWithActualAsync(budgetId, startDate, endDate, userId);
            return Ok(comparison);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/forecast")]
    public async Task<IActionResult> ForecastExpenses(int budgetId, [FromQuery] int months = 3)
    {
        try
        {
            var userId = GetCurrentUserId();
            var forecast = await _analysisService.ForecastExpensesAsync(budgetId, months, userId);
```

```
            return Ok(forecast);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/top-expenses")]
    public async Task<IActionResult> AnalyzeTopExpenses(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate, [FromQuery] int limit = 10)
    {
        try
        {
            var userId = GetCurrentUserId();
            var topExpenses = await _analysisService.AnalyzeTopExpensesAsync(budgetId, startDate, endDate, limit, userId);
            return Ok(topExpenses);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/top-incomes")]
    public async Task<IActionResult> AnalyzeTopIncomes(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate, [FromQuery] int limit = 10)
    {
        try
        {
            var userId = GetCurrentUserId();
            var topIncomes = await _analysisService.AnalyzeTopIncomesAsync(budgetId, startDate, endDate, limit, userId);
            return Ok(topIncomes);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/expenses-by-user")]
    public async Task<IActionResult> AnalyzeExpensesByUser(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate)
    {
        try
        {
            var userId = GetCurrentUserId();
            var expensesByUser = await _analysisService.AnalyzeExpensesByUserAsync(budgetId, startDate, endDate, userId);
            return Ok(expensesByUser);
        }
        catch (Exception ex)
        {
            return HandleException(ex);
        }
    }

    [HttpGet("budget/{budgetId}/expense-changes")]
    public async Task<IActionResult> AnalyzeExpenseChanges(int budgetId, [FromQuery] DateTime startDate, [FromQuery] DateTime endDate)
    {
        try
        {
            var userId = GetCurrentUserId();
            var changes = await _analysisService.AnalyzeExpenseChangesAsync(budgetId, startDate, endDate, userId);
            return Ok(changes);
        }
```

```
catch (Exception ex)                                    }
{                                                     }
   return HandleException(ex);                    }
}
```