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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення для контролю витрат сімейного бюджету на хобі та розваги»

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

_____ Олег СІКАЛО
(підпис)

Виконав: здобувач(ка) вищої освіти групи ПД-42

Олег СІКАЛО

Керівник: Тимур ДОВЖЕНКО
доц. каф. 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

Сікало Олегу Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка програмного забезпечення для контролю витрат сімейного бюджету на хобі та розваги»
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: огляд існуючих програмних рішень для фінансового обліку, нормативні документи з програмної інженерії, методичні матеріали з розробки інформаційних систем, технічні характеристики цільової платформи, програмні інтерфейси та інструменти для реалізації застосунку, вимоги до безпеки й конфіденційності даних.

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

1. Огляд та аналіз існуючих програмних рішень для обліку витрат на хобі і розваги в контексті ведення сімейного бюджету.
2. Проектування програмного забезпечення для контролю витрат на хобі та розваги в межах сімейного бюджету.

3. Програмна реалізація та опис функціонування застосунку для контролю витрат на хобі та розваги.
4. Тестування програмного забезпечення та оцінка його ефективності в умовах практичного використання.

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 методів і технологій обліку витрат	14.03–20.03.2025	
4	Проектування архітектури програмного забезпечення та інтерфейсу користувача	21.03–31.03.2025	
5	Програмна реалізація застосунку	01.04–20.04.2025	
6	Тестування застосунку	21.04–28.04.2025	
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	

Здобувач вищої освіти _____ Олег СІКАЛО
(підпис)

Керівник
кваліфікаційної роботи _____ Тимур ДОВЖЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 52 стор., 1 табл., 25 рис., 23джерел.

Мета роботи – розробка програмного забезпечення для контролю витрат на хобі та розваги в межах сімейного бюджету з можливістю візуального аналізу та гнучкого управління фінансовими даними.

Об'єкт дослідження – процес обліку та аналізу витрат у сфері дозвілля в межах сім'ї.

Предмет дослідження – програмне забезпечення для ведення обліку сімейного бюджету з акцентом на витрати на хобі та розваги.

Короткий зміст роботи: у роботі проведено аналіз сучасних програмних рішень для управління особистими та сімейними фінансами. Виокремлено основні переваги та недоліки популярних застосунків, таких як Money Lover, CoinKeeper, Monefy. Визначено функціональні та нефункціональні вимоги до майбутнього застосунку. Спроєктовано архітектуру системи на основі N-Layer підходу, реалізовано серверну частину за допомогою ASP.NET Core 8, клієнтську частину – з використанням React та TypeScript, збереження даних – на платформі SQL Server.

Програмне забезпечення підтримує створення та керування бюджетами, категоріями витрат, користувачами та сім'ями, а також відображає аналітику у вигляді графіків. Проведено функціональне тестування реалізованої системи, проаналізовано її ефективність, стабільність і зручність у практичному використанні.

Система може застосовуватись для покращення фінансової дисципліни в сім'ї, планування витрат на дозвілля та створення культури відповідального споживання.

КЛЮЧОВІ СЛОВА: СІМЕЙНИЙ БЮДЖЕТ, ВИТРАТИ, ХОБІ, ВЕБ-ЗАСТОСУНОК, АНАЛІТИКА, ASP.NET CORE, REACT, SQL SERVER

ЗМІСТ

ВСТУП

У сучасному динамічному світі питання раціонального використання фінансів є особливо актуальним як для окремих осіб, так і для сімей. Зростання вартості життя, інфляційні процеси, економічна нестабільність та постійний потік пропозицій на ринку розваг і дозвілля формують потребу у свідомому плануванні бюджету. Особливої уваги потребує сфера витрат на хобі та розваги, які, незважаючи на свою не обов'язковість, суттєво впливають на загальний фінансовий стан сім'ї. Часто саме в цій категорії відбуваються спонтанні витрати, які можуть виходити за рамки встановленого бюджету.

У зв'язку з цим важливим є впровадження цифрових інструментів, що дозволяють ефективно контролювати фінансові потоки, планувати витрати та аналізувати їхню структуру. Сучасні технології відкривають широкі можливості для створення веб-застосунків, здатних не лише фіксувати витрати, а й надавати аналітичну інформацію для прийняття обґрунтованих рішень. Такі системи сприяють підвищенню фінансової дисципліни, прозорості витрат та зміцненню фінансової відповідальності серед усіх членів сім'ї.

Актуальність теми обумовлена зростаючим попитом на прості у використанні, функціональні та безпечні рішення, які дозволяють контролювати витрати в окремих категоріях, зокрема в розділі хобі та дозвілля. Більшість наявних рішень або не мають вузької спеціалізації на цій сфері, або не підтримують сімейний режим з можливістю спільного управління бюджетами. Саме тому виникає потреба у створенні інструменту, який би враховував ці особливості та був орієнтований на реальні повсякденні потреби користувача.

1 ОГЛЯД ТА АНАЛІЗ ІСНУЮЧИХ ПРОГРАМНИХ РІШЕНЬ КОНТРОЛЮ ВИТРАТ СІМЕЙНОГО БЮДЖЕТУ

1.1 Аналіз предметної галузі

Предметна галузь охоплює сферу управління сімейними фінансами, а саме — облік, контроль та аналіз витрат, пов’язаних із хобі та розвагами. Це окрема підкатегорія у загальній тематиці фінансового планування, яка потребує особливої уваги через її непередбачуваність, емоційну природу та тенденцію до перевищення запланованого бюджету.

Хобі та розваги є важливою частиною життя сучасної людини, сприяють покращенню психоемоційного стану, розвитку особистості та зміцненню соціальних зв’язків у сім’ї. Водночас витрати на ці потреби часто є джерелом фінансового дисбалансу, особливо за відсутності системного підходу до планування та обліку. Нерідко такі витрати здійснюються імпульсивно або без попереднього аналізу фінансових можливостей, що може призводити до перевищення бюджету, накопичення боргів або обмеження в інших важливих сферах.

В умовах сучасного суспільства, коли більшість фінансових операцій здійснюється в цифровому вигляді, особливої актуальності набувають програмні засоби, що дозволяють ефективно управляти сімейним бюджетом, зокрема в частині необов’язкових, але важливих витрат — таких як дозвілля, подорожі, культурні заходи, хобі тощо. Наявні програмні рішення часто не мають окремої логіки для аналізу саме таких витрат, або обмежують користувача у гнучкості налаштувань, що знижує ефективність їх використання в реальних умовах.

Таким чином, розробка програмного забезпечення для контролю витрат сімейного бюджету на хобі та розваги є актуальним завданням, яке відповідає потребам широкого кола користувачів. Такий застосунок повинен забезпечити можливість формування бюджетів на визначені періоди, створення власних

категорій витрат, візуалізацію витрат, аналітику у вигляді діаграм, а також підтримку колективного управління фінансами сім'ї. Це дозволить підвищити фінансову дисципліну, знизити ризики перевитрат та забезпечити раціональне використання ресурсів без шкоди для якості життя.

1.2 Огляд існуючих програмних рішень

З метою виявлення переваг та недоліків актуальних програмних рішень для контролю сімейного бюджету, доцільно провести аналіз найбільш популярних і функціонально різноманітних застосунків у цій галузі. Для аналізу були обрані такі сервіси: Money Lover, CoinKeeper та Monefy. Критеріями відбору стали популярність серед користувачів, наявність базових функцій бюджетування, підтримка мобільних або веб-платформ, а також доступність у безкоштовному режимі.

Кожен із цих застосунків має власний підхід до ведення особистих або сімейних фінансів, і тому їх порівняння дозволяє більш обґрунтовано визначити необхідність та доцільність створення нового спеціалізованого веб-застосунку для контролю витрат саме на хобі та розваги. Особливу увагу під час аналізу приділено таким аспектам: підтримка кількох користувачів (сімейний доступ), можливість налаштування власних категорій витрат, створення бюджетів на конкретні періоди, візуалізація аналітики та наявність реклами чи платного функціоналу.

1.2.1 Money Lover

Money Lover — це популярний мобільний додаток для обліку особистих та сімейних фінансів, який надає користувачам широкі можливості для контролю доходів і витрат. Основна функціональність включає створення бюджетів, ведення транзакцій за категоріями, нагадування про борги, аналітику за допомогою діаграм і підтримку кількох гаманців. Додаток дозволяє встановлювати фінансові цілі та відстежувати прогрес до їх досягнення.

Однак, незважаючи на гнучкий функціонал, Money Lover має певні обмеження. Більшість розширених функцій (наприклад, експорт звітів, спільне використання бюджету, підтримка кількох пристроїв в реальному часі) доступні лише у преміум-версії. Також відсутня вузька спеціалізація на хобі та розвагах, а візуалізація даних не завжди інтуїтивно зрозуміла для нових користувачів.

На рисунку 1.1 зображено інтерфейс додатка.

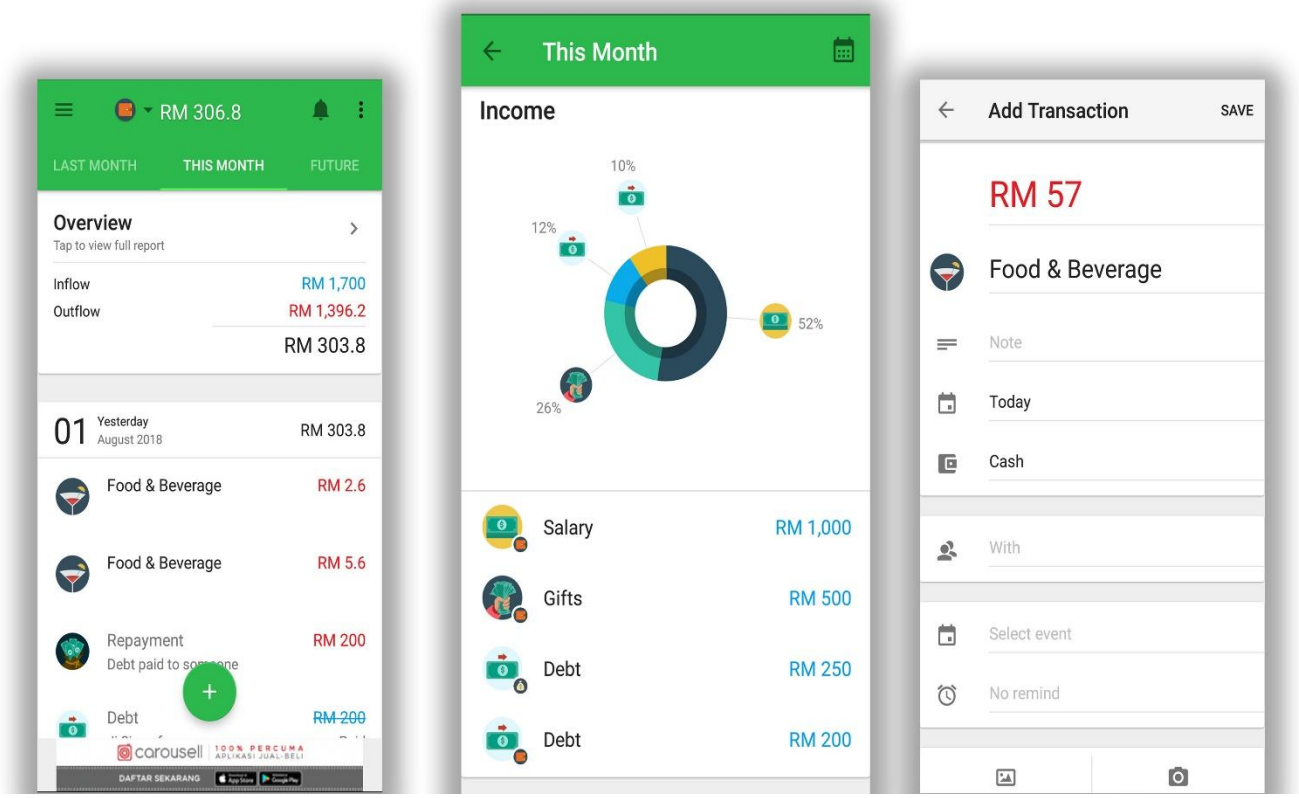


Рис. 1.1 Інтерфейс додатка Money Lover

1.2.2 CoinKeeper

CoinKeeper — це додаток, орієнтований на швидкий та зручний облік фінансів за допомогою інтерактивного інтерфейсу, де користувачі "перетягують" монети до відповідних категорій витрат. Такий підхід надає певну ігрову складову у процес планування бюджету, що робить застосунок привабливим для широкого кола користувачів.

Серед функціональних можливостей: встановлення місячного бюджету, аналітика витрат, категоризація операцій, нагадування про платежі, підтримка

кількох рахунків. CoinKeeper також дозволяє синхронізувати дані між пристроями, однак повний функціонал доступний лише у платній версії. Безкоштовна версія має обмеження за кількістю рахунків і доступних категорій.

Додаток не має спеціалізації для хобі та розваг, а відсутність гнучкого пошуку за різними параметрами (датою, категорією, бюджетом) зменшує ефективність аналізу витрат у вузьких сферах.

На рисунку 1.2 зображено інтерфейс додатка CoinKeeper.

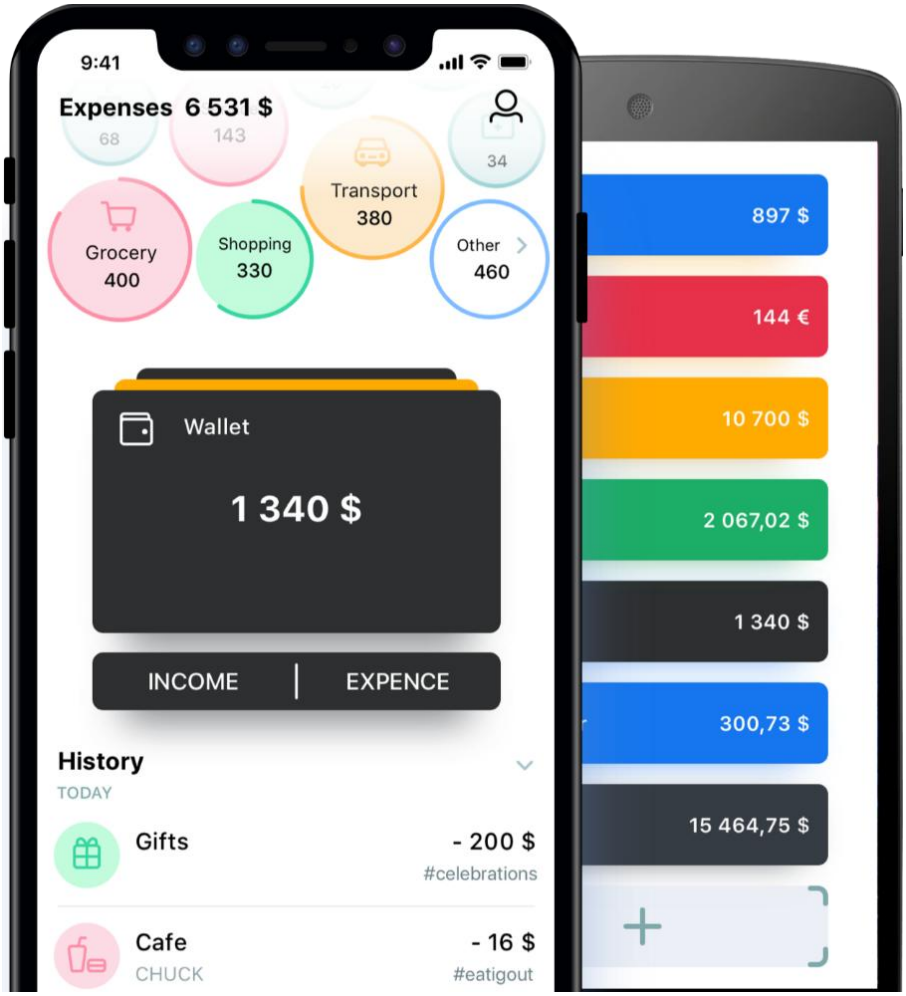


Рис. 1.2 Інтерфейс додатка CoinKeeper

1.2.3 Monefy

Monefy — один із найпростіших у використанні мобільних застосунків для обліку витрат. Основна перевага — мінімалістичний інтерфейс, який дозволяє

додавати транзакції в один клік, вибираючи категорію з яскравих піктограм. Додаток орієнтований на швидке щоденне використання без зайвої складності.

До основних функцій належать: ведення щоденних витрат, категоризація транзакцій, побудова простих кругових діаграм, можливість ведення кількох рахунків. Проте в безкоштовній версії відсутня можливість синхронізації з іншими пристроями, експорту даних, а також спільного користування бюджетом.

Monefy не дозволяє створювати власні категорії або працювати з кількома учасниками сім'ї, що є важливим для застосунку, орієнтованого на сімейне використання. Крім того, відсутні розширені функції аналітики або роботи з окремими бюджетами на певні періоди.

На рисунку 1.3 зображено інтерфейс додатка Monefy.

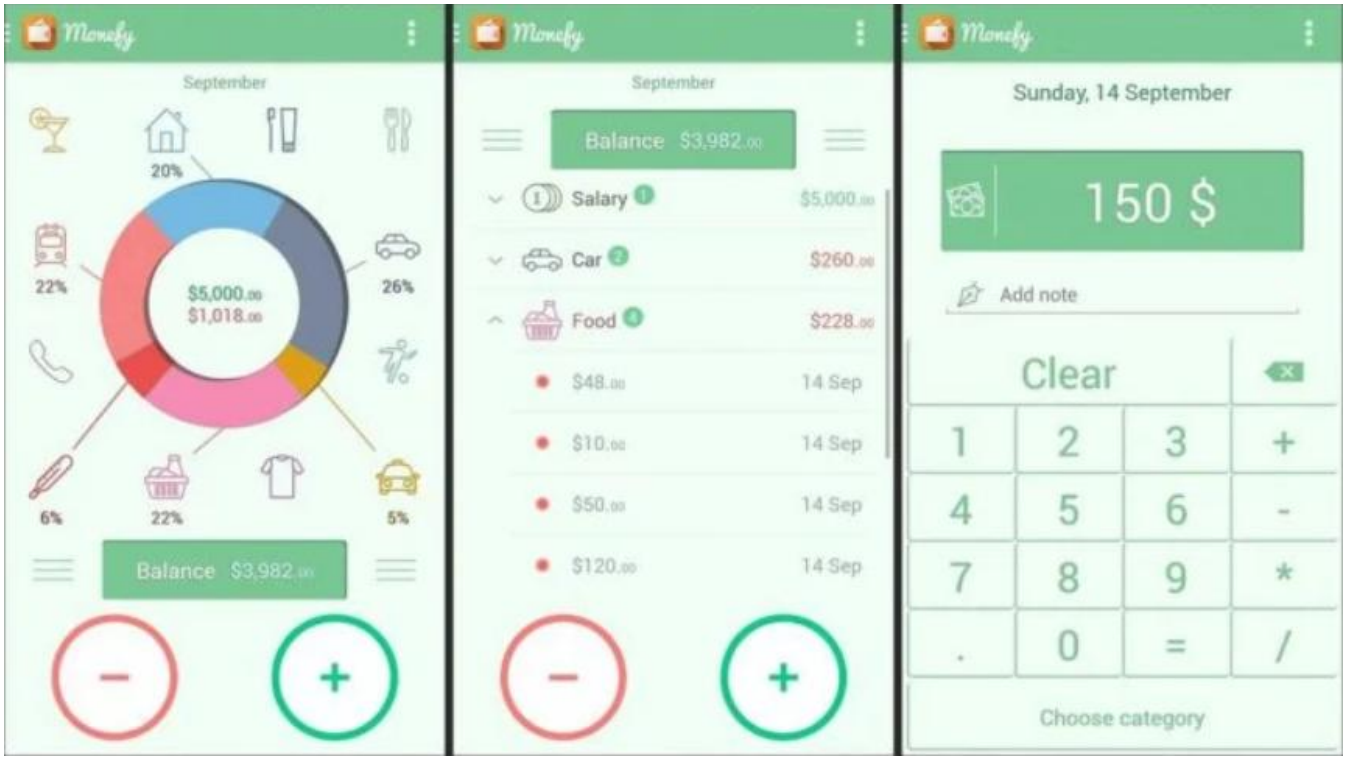


Рис. 1.3 Інтерфейс додатка Monefy

1.3 Порівняння існуючих програмних рішень

Проведено порівняльний аналіз функціональних можливостей популярних додатків для контролю витрат — Money Lover, CoinKeeper та Monefy — із

розроблюваним веб-застосунком. Метою порівняння є виявлення сильних і слабких сторін існуючих програмних рішень, а також обґрунтування доцільності створення нового програмного продукту, орієнтованого на зручний облік сімейних витрат саме на хобі та розваги.

Для об'єктивного порівняння були обрані такі ключові критерії:

- наявність повного функціоналу у безкоштовній версії;
- наявність реклами;
- можливість створення бюджетів;
- підтримка власних категорій витрат;
- пошук витрат за різними параметрами;
- візуалізація витрат (графіки, діаграми);
- підтримка сімейного доступу;
- зручність інтерфейсу;
- синхронізація між пристроями;
- спеціалізація на витратах на хобі та розваги;
- гнучка аналітика по бюджетах.

На основі вищевказаних параметрів було складено порівняльну таблицю 1.1.

Таблиця 1.1

Порівняння існуючих програм-аналогів

Критерій	Money Lover	CoinKeeper	Monefy	FunFinance
Безкоштовна версія з повним функціоналом	-	-	+	+
Реклама в інтерфейсі	-	+	+	-
Створення бюджетів	+	+	-	+
Власні категорії витрат	-	-	-	+
Пошук за категоріями, датою, періодом, бюджетом	-	-	-	+

Продовження таблиці 1.1

Порівняння існуючих програм-аналогів

Критерій	Money Lover	CoinKeeper	Monefy	FunFinance
Візуалізація витрат (діаграми)	+	+	-	+
Сімейний доступ	-	-	-	+
Керування учасниками	-	-	-	+
Синхронізація між пристроями	-	-	-	+
Орієнтація на витрати на хобі та розваги	-	-	-	+
Аналітика бюджету за періодами	-	-	-	+

Money Lover пропонує широкий функціонал, проте більшість можливостей, зокрема створення власних категорій, повноцінну аналітику та синхронізацію — доступні лише у платній версії. Реклама відсутня, але функціонал обмежено для безкоштовного користування.

CoinKeeper має привабливий інтерфейс, однак демонструє обмежені можливості щодо налаштувань, сімейного доступу та розширеного аналізу даних. Безкоштовна версія містить рекламу та обмежений функціонал.

Monefy орієнтований на швидкий облік витрат із простим інтерфейсом, однак не дозволяє створювати бюджети або шукати витрати за параметрами. Також не передбачено функцій для роботи з сімейним бюджетом.

Розроблюваний веб-застосунок враховує недоліки конкурентів та поєднує їхні сильні сторони. Він дозволяє гнучко управляти бюджетами, створювати власні категорії, виконувати пошук, здійснювати аналітику за періодами, а також підтримує безкоштовне використання без реклами. Основною перевагою є орієнтація саме на контроль витрат сімейного бюджету на хобі та розваги, що робить його унікальним на ринку.

2 ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ КОНТРОЛЮ ВИТРАТ НА ХОБІ ТА РОЗВАГИ

2.1 Визначення функціональних та нефункціональних вимог

Процес проєктування програмного забезпечення розпочинається з чіткого формулювання вимог до системи. Вимоги поділяються на функціональні — ті, що описують конкретну поведінку або функціональність, яку має реалізовувати система, та нефункціональні — що стосуються якості роботи системи, таких як зручність використання, продуктивність, безпека, масштабованість тощо.

У цьому розділі визначено ключові функціональні можливості, які має забезпечувати система для ефективного контролю витрат на хобі та розваги в межах сімейного бюджету. Також сформульовано нефункціональні вимоги, що визначають технічні та експлуатаційні характеристики, яким має відповідати програмний продукт.

2.1.1 Функціональні вимоги

Функціональні вимоги описують основні можливості системи, які визначають її поведінку з точки зору користувача. Вони є основою для реалізації бізнес-логіки застосунку та формують базу для архітектури та структури даних. Для системи контролю витрат сімейного бюджету на хобі та розваги були сформульовані такі ключові функціональні вимоги.

1. Аутентифікація та управління користувачами

1.1. Реєстрація та вхід:

Система повинна забезпечувати можливість створення облікового запису користувача, де він вказує ім’я, електронну пошту та пароль. Користувач має змогу входити в систему з використанням зареєстрованих облікових даних.

1.2. Управління профілем:

Користувач повинен мати доступ до свого профілю з можливістю перегляду та редагування персональних даних, зміни пароля, а також виходу з системи.

2. Управління сім’ями

2.1. Створення та редагування сім’ї:

Користувач має можливість створити сім’ю, редагувати її назву та опис, а також за необхідності видалити її (якщо він є адміністратором).

2.2.Членство в сім’ї:

Один користувач може входити до однієї або кількох сімей. Адміністратор має змогу надсилати запрошення іншим користувачам, переглядати статус запрошень, приймати або відхиляти запити, а також видаляти учасників.

3. Управління бюджетами

3.1.Створення та редагування бюджетів:

Користувач може створити новий бюджет, вказавши його назву, грошовий ліміт, початкову та кінцеву дату. Також передбачено редагування параметрів та можливість видалення бюджету.

3.2. Перегляд бюджетів:

Система повинна відображати список усіх бюджетів, що належать до сім’ї користувача, із зазначенням деталей: використана сума, залишок, прогрес виконання (візуалізований прогрес-бар). Бюджети, які перевищили встановлений ліміт або наближаються до нього, повинні бути візуально виділені.

4. Управління категоріями витрат

4.1. Створення та редагування категорій:

Користувач має змогу створювати власні категорії витрат, змінювати їх назви та видаляти ті категорії, які не мають пов’язаних транзакцій.

4.2.Перегляд категорій:

Інтерфейс повинен відображати список усіх доступних категорій із зазначенням загальної суми витрат у кожній з них.

5. Управління витратами

5.1.Додавання та редагування витрат:

Користувач повинен мати змогу додавати нові витрати, зазначаючи суму, опис, дату, обрану категорію та бюджет. Також має бути передбачена можливість редагування та видалення вже існуючих витрат.

5.2. Перегляд витрат:

Система повинна відображати повний список витрат з можливістю їх фільтрації за категоріями, датами або бюджетами. Також користувач повинен мати змогу сортувати витрати за сумою, датою або іншими параметрами.

6. Аналітика та звіти

6.1. Візуалізація даних:

Застосунок має забезпечувати графічне представлення витрат у вигляді діаграм: за категоріями, у часовому розрізі та з розподілом за учасниками сім'ї. Це дозволяє краще аналізувати структуру витрат та приймати обґрунтовані фінансові рішення.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики програмного забезпечення, що не стосуються безпосередньо його функціональності, але мають критичне значення для практичного використання, зручності, надійності та масштабованості системи. Для застосунку контролю витрат на хобі та розваги в межах сімейного бюджету сформульовано такі конкретні нефункціональні вимоги:

1. Продуктивність

Система повинна забезпечувати швидку реакцію на дії користувача. Завантаження основних сторінок не повинно перевищувати 2 секунд при середньому навантаженні. Запити до бази даних повинні оброблятися за менш ніж 500 мілісекунд для простих операцій читання. Візуалізація аналітики повинна генеруватися не довше ніж 1 секунду після вибору параметрів.

2. Масштабованість

Система повинна підтримувати зростання кількості користувачів та обсягів даних без зниження продуктивності. Архітектура застосунку має бути побудована таким чином, щоб у разі потреби забезпечити можливість горизонтального масштабування (наприклад, за рахунок переносу бази даних у хмару або додавання окремих сервісів).

3. Безпека

Застосунок повинен дотримуватися сучасних стандартів захисту даних:

- передача конфіденційної інформації має здійснюватися через захищене з'єднання HTTPS;
- доступ до сімейних бюджетів, витрат та аналітики повинен бути обмежений лише учасниками відповідної сім’ї;
- реалізовано перевірку прав доступу на рівні API.

2.2 Засоби реалізації

Розробка програмного забезпечення для контролю витрат на хобі та розваги в межах сімейного бюджету потребує використання сучасних, надійних та ефективних інструментів. Вибір технологій і засобів розробки безпосередньо впливає на функціональність, продуктивність, масштабованість та зручність використання програмного продукту.

Під час проектування та реалізації веб-застосунку були враховані вимоги до кросплатформенності, швидкодії, зручності візуального інтерфейсу, ефективного управління даними, можливості подальшого розширення функціоналу, а також підтримки сучасних стандартів розробки.

Нижче описано основні інструменти та середовища, використані під час реалізації проєкту, а також обґрунтовано їх вибір з урахуванням особливостей поставленого завдання.

2.2.1 ASP.NET Core 8

ASP.NET Core 8 — це сучасна, кросплатформенна, високопродуктивна веб-платформа з відкритим вихідним кодом, розроблена компанією Microsoft. Вона є продовженням класичного ASP.NET, проте значно перевершує його в гнучкості, модульності, продуктивності та можливостях масштабування. ASP.NET Core 8 є частиною платформи .NET 8 і забезпечує розробникам потужний інструментарій для створення веб-застосунків, RESTful API, мікросервісів, хмарних рішень тощо.

У межах цієї дипломної роботи ASP.NET Core 8 було обрано для реалізації серверної частини (бекенду) веб-застосунку, що забезпечує контроль витрат на хобі та розваги в межах сімейного бюджету. Цей вибір зумовлений низкою ключових переваг платформи які наведено нижче.

Кросплатформенність. Застосунок, розроблений на ASP.NET Core, може бути розгорнутий на різних операційних системах, зокрема Windows, Linux та macOS. Це надає гнучкість у виборі хостингу та розширює можливості для майбутнього масштабування.

Модульна архітектура. ASP.NET Core побудований як набір незалежних пакетів NuGet, що дозволяє використовувати лише необхідні компоненти, зменшуючи розмір застосунку та спрощуючи його підтримку.

Висока продуктивність. У порівнянні з попередніми версіями ASP.NET та багатьма конкурентними технологіями, ASP.NET Core забезпечує швидшу обробку HTTP-запитів, що критично важливо для веб-сервісів, які повинні обробляти велику кількість запитів за короткий проміжок часу.

Підтримка REST API. Фреймворк має вбудовані засоби для створення RESTful веб-служб, що дає змогу ефективно реалізувати логіку взаємодії між клієнтською частиною (React) та сервером, забезпечуючи надійну передачу даних.

Безпека. ASP.NET Core 8 підтримує сучасні механізми аутентифікації та авторизації, включаючи JWT-токени, OAuth 2.0, інтеграцію з зовнішніми провайдерами (Google, Microsoft, Facebook тощо), що дозволяє створити безпечне середовище для користувачів застосунку.

Інтеграція з іншими технологіями Microsoft. Платформа легко поєднується з такими інструментами як Entity Framework Core, Azure, SQL Server, що полегшує побудову комплексних бізнес-рішень.

Також важливо відзначити наявність Visual Studio 2022 як основного середовища розробки, яке надає інтегровану підтримку ASP.NET Core 8: інтелектуальні підказки, генерацію коду, запуск та налагодження проєктів, вбудовані шаблони та інструменти аналізу якості коду.

Таким чином, вибір ASP.NET Core 8 як основної серверної технології для даного проєкту є цілком виправданим: платформа забезпечує високу продуктивність, зручність розробки, гнучкість у розгортанні та підтримку сучасних стандартів веб-програмування, що є ключовими факторами для створення ефективного і надійного програмного забезпечення.

2.2.2 Entity Framework Core 8

Entity Framework Core 8 (EF Core 8) — це сучасна версія популярного об'єктно-реляційного ORM-фреймворку (Object-Relational Mapping), розробленого компанією Microsoft. EF Core є важливим компонентом екосистеми .NET, який забезпечує зручний спосіб роботи з базами даних через об'єктно-орієнтовану модель, дозволяючи розробникам уникати прямого написання SQL-запитів для більшості операцій.

У межах розробки дипломного веб-застосунку EF Core 8 було обрано як основний засіб для доступу до бази даних SQL Server. Це рішення є логічним продовженням вибору ASP.NET Core 8, оскільки обидві технології тісно інтегруються одна з одною та підтримують найкращі практики сучасного програмування.

Ключові переваги використання Entity Framework Core 8:

- зменшення кількості низькорівневого коду. Завдяки EF Core більшість операцій з базою даних (створення, читання, оновлення, видалення) виконуються за допомогою C#-коду без необхідності писати SQL-запити вручну.

- підтримка LINQ (Language Integrated Query). EF Core дозволяє здійснювати запити до бази даних за допомогою виразів LINQ, що підвищує читабельність і типобезпечність запитів.
- міграції (Migrations). За допомогою EF Core можливо поступово змінювати структуру бази даних у процесі розробки без втрати даних. Міграції є зручним засобом синхронізації моделі даних з реальною базою.
- підтримка складних моделей. EF Core підтримує успадкування, зв'язки типів "один до багатьох", "багато до багатьох", складні типи, атрибути валідації тощо, що дозволяє легко реалізувати логіку складної системи обліку витрат.
- платформонезалежність та оптимізація. EF Core 8 має покращену продуктивність у порівнянні з попередніми версіями, підтримує оптимізацію запитів та кешування, а також дозволяє використовувати різні постачальники баз даних.

Entity Framework Core 8 також чудово інтегрується з Visual Studio 2022, що спрощує управління міграціями, перегляд бази даних, автоматичне створення класів моделей тощо.

Зважаючи на необхідність побудови структурованої, гнучкої та масштабованої системи з тісною взаємодією з базою даних, використання EF Core 8 є доцільним і обґрунтованим вибором. Його переваги дозволяють значно пришвидшити процес розробки, підвищити якість коду та забезпечити надійну взаємодію з реляційною базою даних, що є ключовим елементом у реалізації функціональності контролю сімейного бюджету.

2.2.3 SQL Server

Microsoft SQL Server — це потужна, надійна та масштабована реляційна система управління базами даних (РСУБД), яка широко використовується у корпоративних та веб-рішеннях. Вона забезпечує зберігання, обробку та безпечне керування структурованими даними. У дипломному проєкті SQL Server використовується як основна платформа для зберігання всіх даних, пов’язаних із обліком витрат сімейного бюджету.

Причини вибору SQL Server:

- інтеграція з технологіями Microsoft. SQL Server ідеально поєднується з ASP.NET Core та Entity Framework Core, що дозволяє досягти високої продуктивності та стабільної роботи всього стеку.
- високий рівень безпеки. Система забезпечує розширене шифрування даних, контроль доступу та захист від SQL-ін'єкцій, що є критично важливим у застосунках із персональними фінансовими даними.
- підтримка складних запитів та транзакцій. SQL Server надає потужні інструменти для реалізації транзакцій, агрегатних запитів, з'єднань таблиць, що забезпечує гнучке керування фінансовими записами.

SQL Server є оптимальним вибором для даного проєкту, оскільки він дозволяє створити стабільну, ефективну та захищену систему зберігання даних. Його інтеграція з інструментами .NET-екосистеми, розвинені можливості адміністрування та підтримка аналітики роблять його надійною платформою для реалізації системи контролю сімейного бюджету.

2.2.4 SQL Server Management Studio

SQL Server Management Studio (SSMS) — це офіційне середовище для роботи з базами даних Microsoft SQL Server, яке забезпечує зручний графічний інтерфейс для адміністрування, написання запитів, створення таблиць, перевірки структури даних і виконання операцій із базою. У межах цього дипломного проєкту SSMS виконує роль головного інструменту для керування всіма аспектами бази даних, розробки її структури та тестування SQL-запитів.

Однією з головних переваг використання SSMS є наочність і простота у взаємодії з даними. Інструмент дозволяє створювати та змінювати таблиці, переглядати вміст бази, відлагоджувати запити та контролювати зв'язки між сутностями без потреби писати великий обсяг SQL-коду вручну. Це особливо важливо на етапі проєктування та налагодження бази даних.

Інтеграція SSMS із SQL Server забезпечує гнучкість у налаштуванні користувацьких прав доступу, безпеки та резервного копіювання. Крім того, він

дозволяє швидко виявляти помилки в логіці запитів та структури бази, що підвищує надійність застосунку.

Вибір SQL Server Management Studio є обґрунтованим, оскільки цей інструмент пропонує все необхідне для професійної роботи з SQL Server і є стандартом в багатьох комерційних і навчальних проєктах.

2.2.5 React TypeScript

React — це популярна JavaScript-бібліотека з відкритим кодом, розроблена компанією Meta (раніше Facebook), яка призначена для побудови інтерфейсів користувача на основі компонентного підходу. У поєднанні з TypeScript, що є строго типізованим надмножиною JavaScript, React забезпечує високу надійність, масштабованість та підтримуваність проєкту.

Було обрано React з TypeScript як основу для реалізації клієнтської частини вебзастосунку. Такий вибір пояснюється кількома важливими факторами.

По-перше, TypeScript додає статичну типізацію, що дозволяє уникати великої кількості помилок ще на етапі компіляції. Це значно спрощує розробку складних інтерфейсів і підвищує загальну якість коду. Крім того, підтримка автозаповнення, перевірка типів та документація безпосередньо в середовищі розробки (наприклад, Visual Studio Code) робить процес написання коду зручнішим і швидшим.

По-друге, React дозволяє реалізувати інтерфейс застосунку як набір ізольованих, повторно використовуваних компонентів. У контексті застосунку для контролю витрат на хобі та розваги це означає, що кожен функціональний блок — форма створення бюджету, візуалізація витрат, фільтри, діаграми, тощо — може бути реалізований окремим компонентом, що спрощує підтримку та масштабування застосунку в майбутньому.

По-третє, React має розвинену екосистему, яка включає численні бібліотеки для роботи зі станом (наприклад, Redux або Context API), маршрутизацією (React Router), формами, валідацією та інтеграцією з REST API. Це дозволяє зосередитися на бізнес-логіці проєкту, використовуючи перевірені рішення для типових задач.

Таким чином, поєднання React і TypeScript забезпечує потужну, сучасну та зручну платформу для створення швидкого, адаптивного та функціонального вебінтерфейсу. Цей вибір повністю відповідає вимогам дипломного проєкту, сприяє створенню якісного та зручного для користувача продукту і відповідає сучасним трендам у веброзробці.

2.2.6 React Bootstrap

React Bootstrap — це набір готових UI-компонентів, створених спеціально для інтеграції з React. Він базується на популярному CSS-фреймворку Bootstrap, але реалізований як React-компоненти, що дозволяє використовувати знайомі стилі та шаблони без необхідності взаємодії з HTML-класами вручну.

У проєкті React Bootstrap обрано для прискорення розробки інтерфейсу та забезпечення його адаптивності. Завдяки цьому інструменту легко створювати сучасні, зручні та стилістично узгоджені елементи інтерфейсу — такі як форми, таблиці, кнопки та навігаційні панелі. Крім того, компоненти автоматично адаптуються до різних розмірів екранів, що робить застосунок зручним як для десктопів, так і для мобільних пристроїв.

React Bootstrap дозволяє зосередитись на функціональності, мінімізуючи потребу в ручному стилізуванні, що робить його ефективним вибором для реалізації клієнтської частини застосунку.

2.2.7 Visual Studio 2022

Visual Studio 2022 — це потужне середовище розробки (IDE) від компанії Microsoft, яке забезпечує повний набір інструментів для створення, налагодження, тестування та розгортання програмного забезпечення. Воно підтримує широкий спектр мов програмування, зокрема C#, що є основною мовою на серверній частині застосунку.

У даному проєкті Visual Studio 2022 обрано як основне середовище для розробки бекенд-частини на платформі ASP.NET Core 8. Його використання дозволяє ефективно керувати структурою проєкту, зручно працювати з базою

даних за допомогою Entity Framework Core, а також легко застосовувати сучасні засоби відлагодження та аналізу коду.

Visual Studio надає інтегровану підтримку роботи з Git, генерації міграцій, перегляду структури бази даних та тестування API. Це значно спрощує процес розробки та дозволяє зосередитися на реалізації бізнес-логіки замість вирішення технічних складностей.

2.2.8 Visual Studio Code

Visual Studio Code (VS Code) — це легке, кросплатформене середовище розробки з відкритим кодом, створене компанією Microsoft. Воно підтримує велику кількість мов програмування, має розширювану архітектуру через плагіни та забезпечує швидкий і зручний процес написання коду.

У проєкті VS Code використовується переважно для розробки клієнтської частини застосунку на React з TypeScript. Завдяки розширенням для роботи з React, підтримці автоматичного форматування, перевірки типів, інтеграції з Git та терміналом, Visual Studio Code забезпечує комфортні умови для розробки фронтенду.

Гнучкість налаштувань, підтримка тем, швидкодія та велика кількість плагінів (наприклад, для перевірки синтаксису, автодоповнення JSX/TSX, навігації по проєкту) роблять VS Code зручним інструментом для роботи з сучасними вебтехнологіями.

2.3 Розробка архітектури

Для забезпечення чіткого розділення відповідальностей, зручності супроводу, тестування та масштабованості застосунку було обрано архітектурний підхід N-Layer (багатошарова архітектура). Такий підхід дозволяє ізолювати окремі аспекти системи один від одного, що сприяє покращенню структури коду, підвищує стабільність і спрощує розвиток програмного продукту. Загальна

архітектура програмного забезпечення поділяється на п’ять основних логічних шарів. Діаграма архітектури зображена на рисунку 2.1

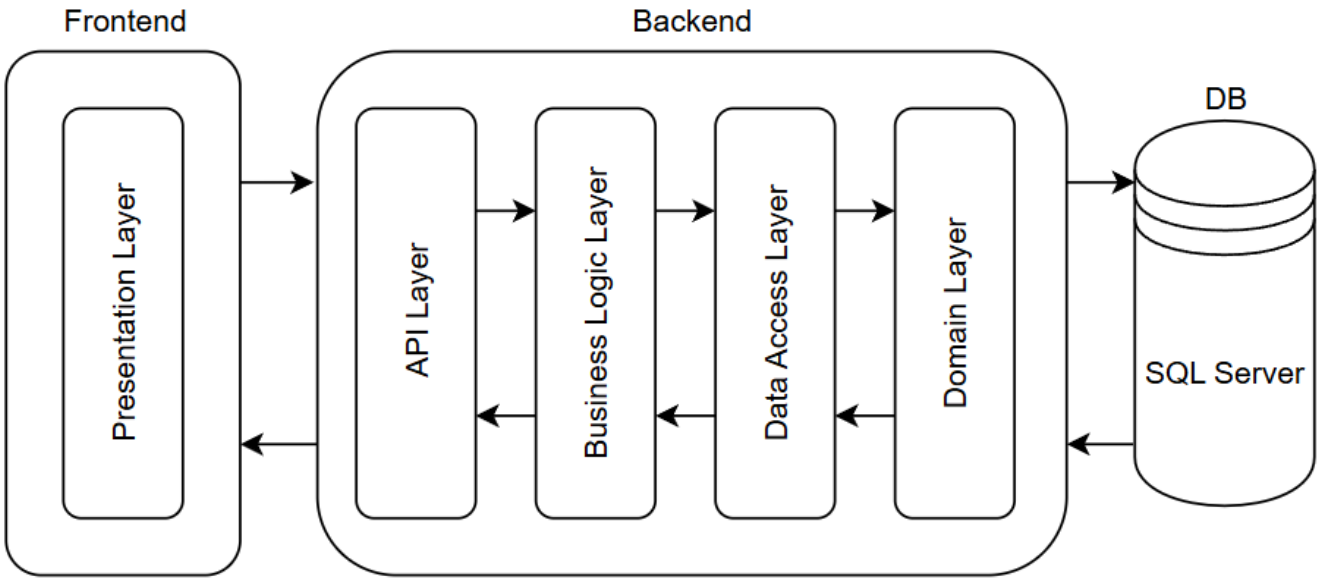


Рис 2.1 Архітектура програмного забезпечення

2.3.1 Presentation Layer

Шар представлення (Presentation Layer) відповідає за взаємодію кінцевого користувача із системою. Це саме той рівень, з яким безпосередньо працює користувач під час виконання всіх основних операцій — від реєстрації та входу до перегляду бюджетів, додавання витрат і аналізу статистики.

У розроблюваному проєкті цей шар реалізовано у вигляді вебінтерфейсу, створеного за допомогою бібліотеки React із використанням мови TypeScript. Такий вибір зумовлений потребою створити гнучкий, адаптивний та інтуїтивно зрозумілий інтерфейс, який би однаково добре працював на різних пристроях — від стаціонарних ПК до мобільних телефонів. Завдяки компонентному підходу React інтерфейс складається з окремих незалежних елементів, що відповідають за конкретні функціональні частини системи: форми, таблиці, діаграми, кнопки, списки тощо. Це дозволяє легко розширювати систему, повторно використовувати код та уникати дублювання логіки.

Інтерактивність інтерфейсу забезпечується за допомогою React Hooks та керування станом компонентів. Кожна дія користувача — наприклад, створення нового бюджету, додавання витрати або перегляд звіту — генерує HTTP-запит до серверної частини (через REST API), а отримані у відповідь дані використовуються для оновлення інтерфейсу в реальному часі без перезавантаження сторінки.

Для візуального оформлення інтерфейсу використовується React Bootstrap, що дозволяє дотримуватись сучасних принципів UI/UX дизайну без потреби ручного стилізування кожного елементу. Готові компоненти Bootstrap допомагають забезпечити єдиний стиль, відповідність адаптивній сітці, а також гарантують правильне відображення на різних розширеннях екранів.

2.3.2 API Layer

Шар API виступає посередником між клієнтською частиною та бізнес-логікою застосунку. Його основна функція — прийом запитів від клієнта, виклик відповідних сервісів, передача оброблених даних назад у вигляді HTTP-відповідей. API реалізовано з використанням ASP.NET Core Web API, що дозволяє створювати RESTful-інтерфейси з підтримкою маршрутизації, валідації даних і обробки помилок.

Контролери API не містять бізнес-логіки безпосередньо, а лише делегують виконання операцій відповідним сервісам.

2.3.3 Business Logic Layer

Шар бізнес-логіки реалізує всі правила, що визначають поведінку системи. Саме тут відбувається перевірка допустимості операцій, розрахунки, перевірки залежностей, обробка запитів користувача тощо. Наприклад, тут реалізується логіка: чи дозволено створити витрату в межах бюджету, чи можна видалити категорію, яка має пов’язані транзакції тощо.

Цей шар складається зі сервісів, які взаємодіють із репозиторіями, викликаються з API та використовують класи доменної моделі.

2.3.4 Data Access Layer

Цей шар відповідає за безпосередню роботу з базою даних. Він реалізує CRUD-операції через репозиторії, використовуючи Entity Framework Core 8 як ORM (Object-Relational Mapping). Таким чином, взаємодія з базою здійснюється через об’єктну модель, що підвищує читабельність та безпеку коду.

Репозиторії ізольовані від бізнес-логіки і не містять жодної логіки валідації або обробки даних — лише операції збереження, оновлення, отримання або видалення об’єктів.

2.3.5 Domain Layer

Цей шар містить класи, які описують предметну область системи: користувачі, сім’ї, бюджети, категорії витрат, транзакції тощо. Кожен клас моделі відображає сутність з реального світу та містить мінімальну логіку, пов’язану із внутрішньою цілісністю (наприклад, перевірку правильності даних у конструкторі).

Моделі використовуються як у бізнес-логіці, так і в шарі доступу до даних, що забезпечує єдину точку істини для структури даних у системі.

2.3.6 Database

База даних реалізована на основі Microsoft SQL Server. Структура бази даних відповідає сутностям доменної моделі. Всі зв’язки між таблицями визначено через зовнішні ключі, що забезпечує цілісність даних. Міграції для створення та оновлення структури бази даних реалізовано через механізм Entity Framework Core.

У базі зберігається інформація про облікові записи користувачів, запрошення до сімей, бюджети, витрати, категорії, а також статистичні дані, які використовуються в аналітиці.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ОПИС ФУНКЦІОНУВАННЯ ЗАСТОСУНКУ

3.1 Структура проєкту та основні компоненти

Програмна реалізація застосунку побудована відповідно до багатошарової архітектури (N-Layer Architecture), що забезпечує чітке розділення відповідальностей між логічними частинами системи. Такий підхід дозволяє досягти високої підтримуваності, масштабованості та прозорості коду, спрощує супровід і тестування системи.

Уся структура рішення згрупована в кілька основних логічних шарів, кожен з яких розміщений у відповідній папці проєкту FunFinance.

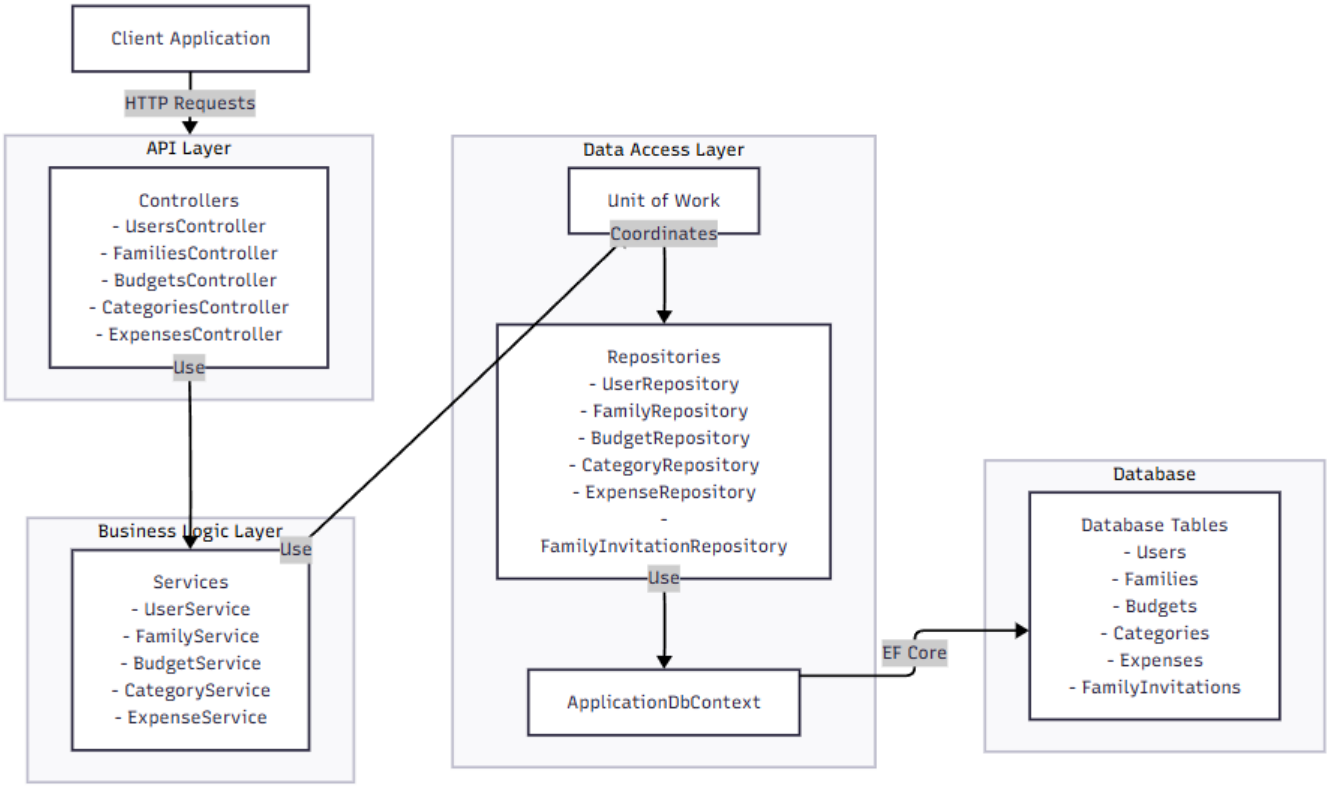


Рис. 3.1 Структура проєкту FunFinance

3.1.1 Models

Цей компонент містить класи, які представляють основні сутності предметної області. Вони відповідають за структуру даних, які використовуються в системі, та слугують основою для формування бази даних.

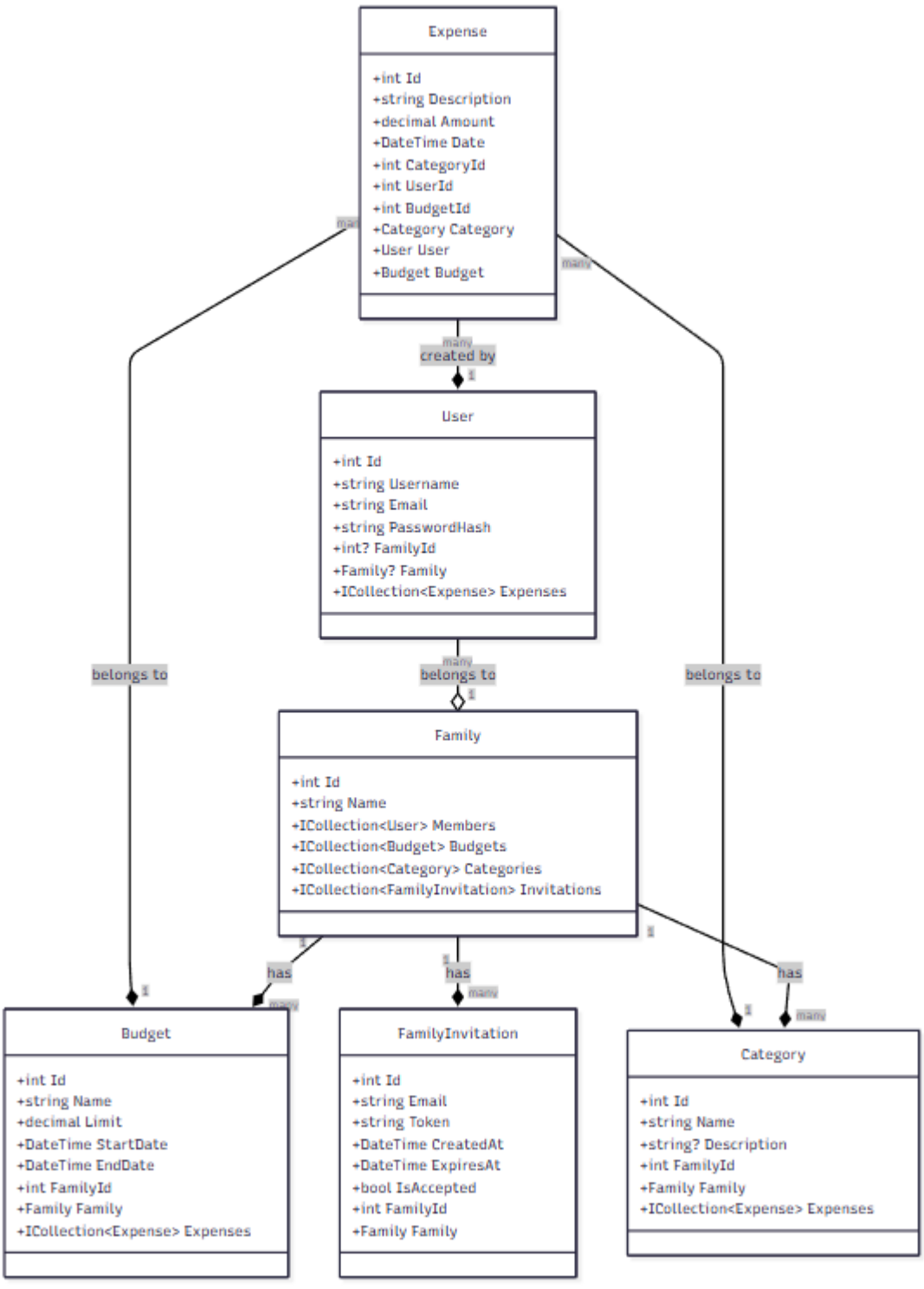


Рис. 3.2 Діаграма класів Domain Layer

Основні моделі:

User.cs — визначає дані користувача системи.

Family.cs — описує структуру сім’ї, до якої можуть належати користувачі.

Budget.cs — модель бюджету на певний період.

Category.cs — відображає категорію витрат.

Expense.cs — зберігає інформацію про окрему витрату.

FamilyInvitation.cs — представляє запрошення користувача до сім’ї.

3.1.2 Data

Папка Data відіграє ключову роль у реалізації доступу до бази даних. Вона містить класи, які безпосередньо відповідають за налаштування структури бази, її конфігурацію та взаємодію з нею на рівні об'єктно-реляційного відображення (ORM). Основним елементом цього компоненту є клас ApplicationDbContext.cs, який реалізує шаблон контексту бази даних у рамках технології Entity Framework Core 8.

Контекст бази даних (DbContext) є посередником між об'єктами програми (моделями) та самою базою даних. Він дозволяє виконувати операції створення, читання, оновлення та видалення даних (CRUD-операції), відображаючи таблиці у вигляді колекцій C#-об’єктів. У цьому контексті ApplicationDbContext містить публічні властивості типу DbSet<TEntity>, які відповідають за зберігання сутностей, таких як Users, Families, Budgets, Categories, Expenses, FamilyInvitations тощо. Кожна з них у базі даних буде представлена окремою таблицею.

Крім визначення таблиць, ApplicationDbContext також містить перевизначений метод OnModelCreating, в якому налаштовуються правила конфігурації зв’язків між сутностями — наприклад, типи зв’язків «один до багатьох» або «багато до багатьох», обмеження зовнішніх ключів, поведінка при видаленні пов’язаних записів (через DeleteBehavior), індекси, унікальність полів тощо.

3.1.3 Repositories

Цей компонент реалізує шаблон Repository, який ізолює логіку доступу до даних від бізнес-логіки. Кожен репозиторій відповідає за взаємодію з конкретною сутністю. Реалізовано також шаблон Unit of Work, який координує роботу кількох репозиторіїв в межах однієї транзакції.

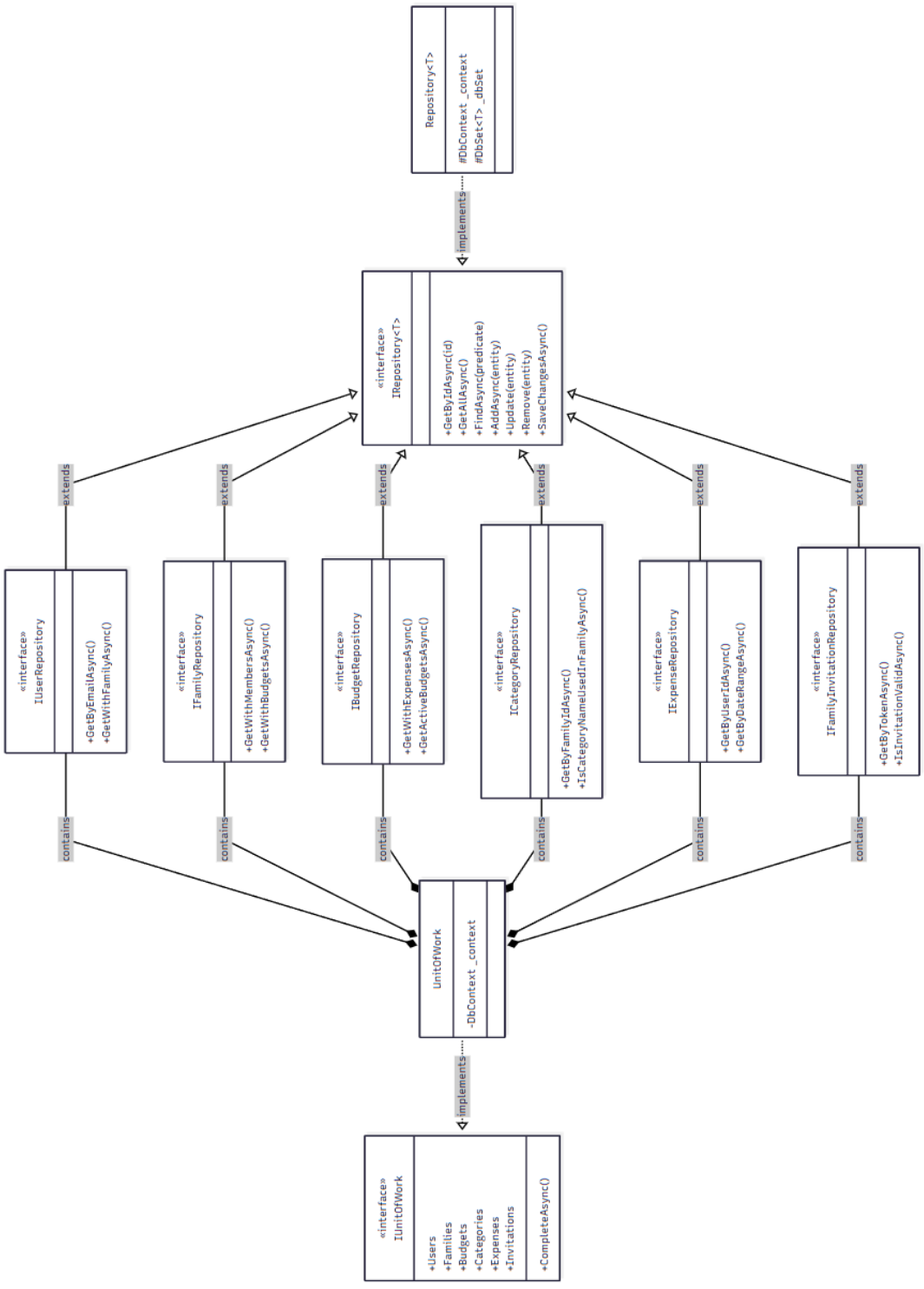


Рис.3.3 Діаграма класів репозиторіїв

3.1.4 Services

Шар сервісів містить бізнес-логіку системи. Тут реалізуються правила, перевірки та операції, які не належать ані до контролерів, ані до шарів доступу до даних.

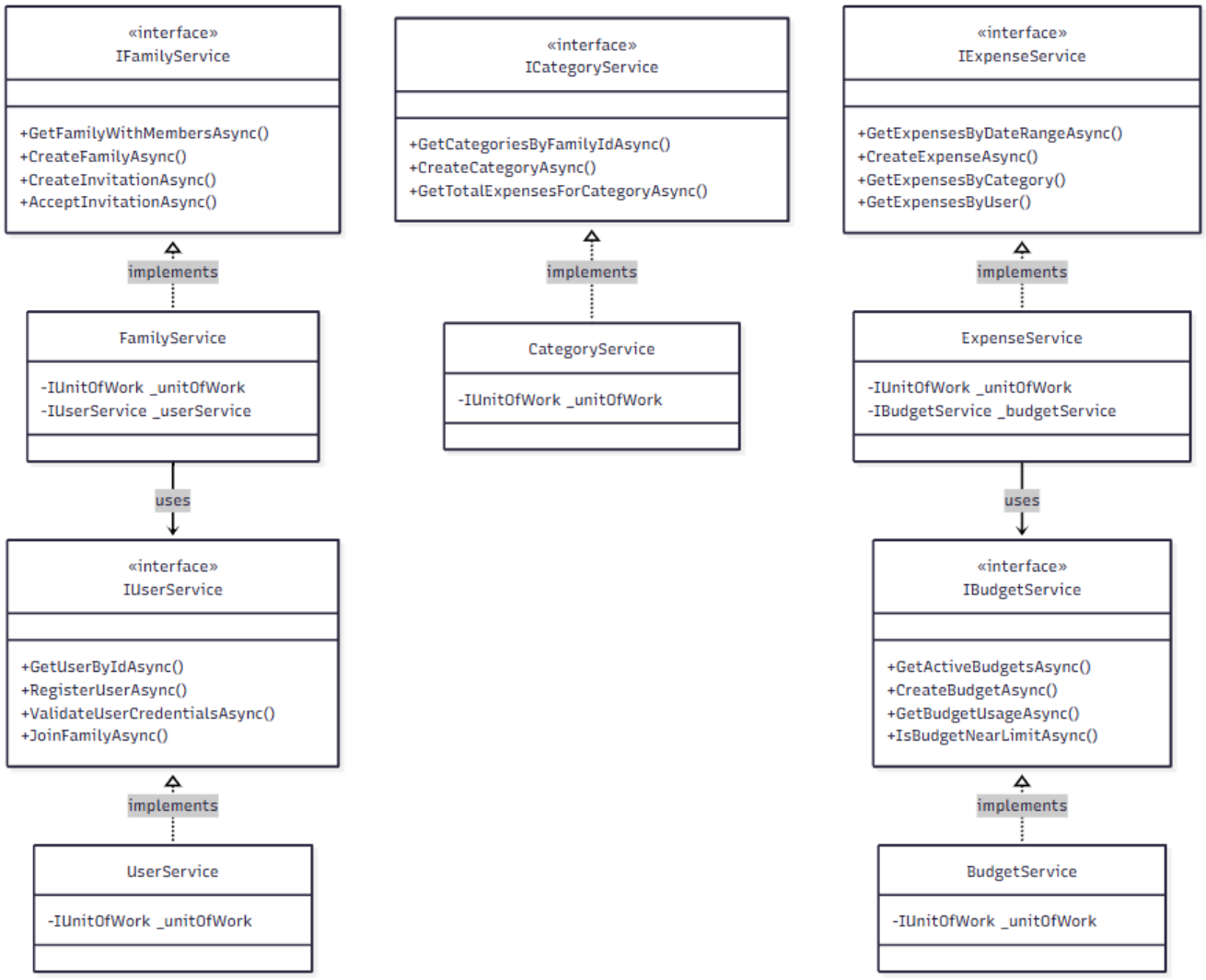


Рис. 3.4 Діаграма класів основних сервісів

Основні сервіси:

UserService.cs, FamilyService.cs, BudgetService.cs, CategoryService.cs, ExpenseService.cs.

Кожен сервіс має відповідний інтерфейс (IUserService, IFamilyService тощо), що дозволяє забезпечити слабке зв'язування між компонентами та спрощує тестування.

3.1.5 Controllers

Цей компонент реалізує API-контролери, які приймають HTTP-запити від клієнтської частини (фронтенду), викликають відповідні сервіси, а потім повертають відповіді у форматі JSON.

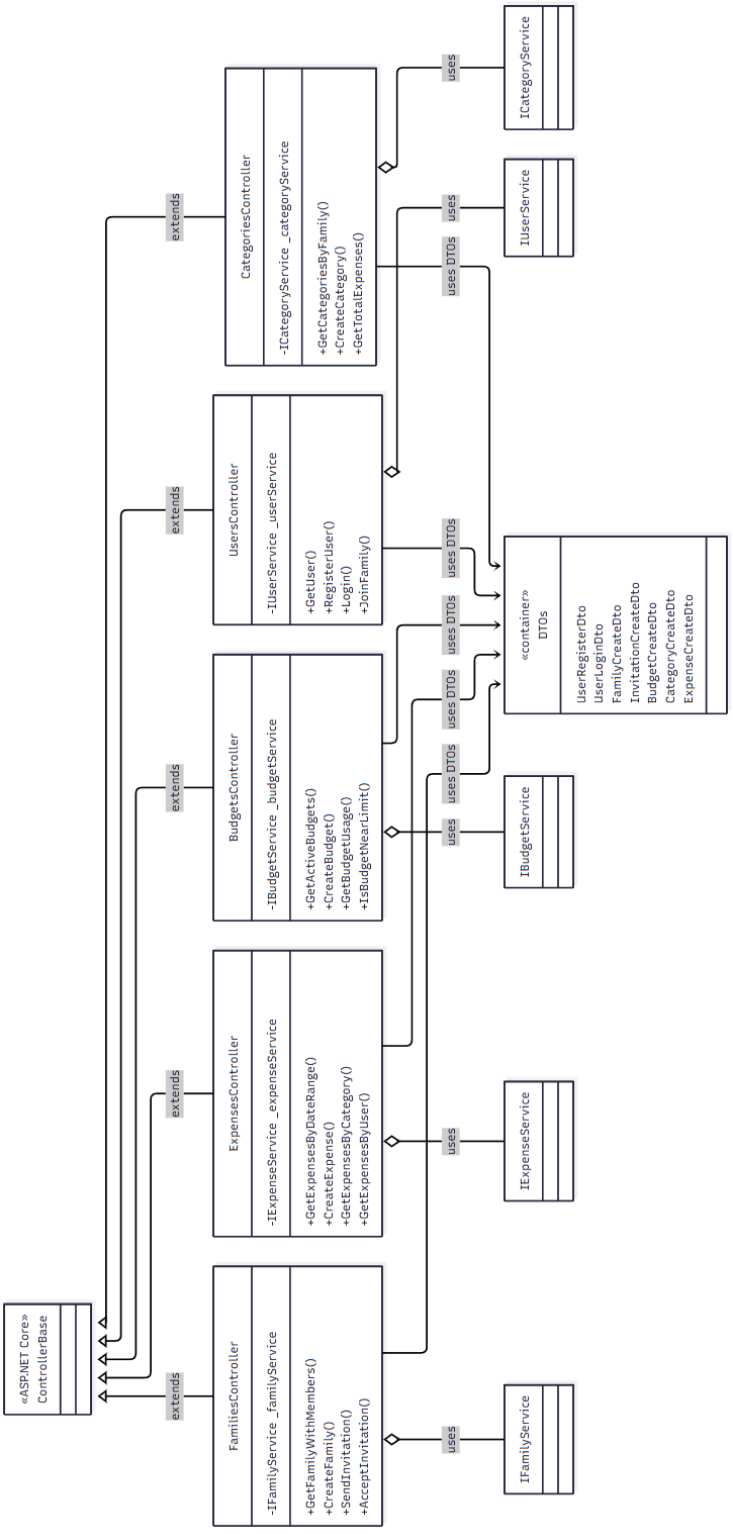


Рис. 3.5 Діаграма класів основних контролерів

3.2 Потік обробки запитів

Ключовим елементом архітектури є її вертикальний поділ на шари: презентаційний шар (контролери API), бізнес-шар (сервіси), шар доступу до даних (репозиторії) та шар зберігання даних. Така організація дозволяє розмежувати відповідальність компонентів і забезпечити високу згуртованість модулів при низькому зчепленні між ними. Кожен шар взаємодіє тільки з безпосередньо пов'язаними шарами через чітко визначені інтерфейси, що є реалізацією принципу інверсії залежностей (DIP).

У контексті управління сімейним бюджетом для хобі та розваг, взаємодія компонентів набуває особливої важливості, адже кожен запит користувача проходить через усі шари системи, забезпечуючи валідацію, обробку бізнес-логіки та збереження даних відповідно до функціональних вимог.

Обробка запитів користувача в системі FunFinance є основою її функціонування та відображає принципи, закладені в архітектуру. Життєвий цикл кожного запиту починається з моменту його надходження до API-ендпойнту і завершується формуванням відповіді клієнту. Протягом цього циклу запит проходить через різні шари архітектури, кожен з яких здійснює свій внесок у його обробку.

Коли користувач ініціює запит, наприклад, для створення нової витрати або перегляду бюджету, цей запит спочатку потрапляє до відповідного контролера. Контролери, представлені класами UsersController, FamiliesController, BudgetsController, CategoriesController та ExpensesController, слугують точками входу в систему та відповідають за первинну валідацію вхідних даних і маршрутизацію запитів до відповідних сервісів. Варто відзначити, що контролери оперують з даними через Data Transfer Objects (DTO), які абстрагують внутрішню структуру моделей від зовнішніх інтерфейсів, підвищуючи безпеку та гнучкість API.

Після успішної валідації даних, контролер передає запит до відповідного сервісу бізнес-логіки. Сервіси, представлені класами UserService, FamilyService,

BudgetService, CategoryService та ExpenseService, реалізують основну бізнес-логіку системи та виконують такі функції як реєстрація користувачів, управління бюджетами, аналіз витрат тощо. Кожен сервіс інкапсулює сукупність пов'язаних бізнес-операцій та правил, забезпечуючи консистентність та цілісність бізнес-процесів.

Важливо зауважити, що архітектура сервісного шару передбачає можливість взаємодії між різними сервісами. Наприклад, FamilyService використовує функціональність UserService для управління членами сім'ї, а ExpenseService звертається до BudgetService для перевірки лімітів бюджету. Ця взаємодія дозволяє організувати складні бізнес-процеси, зберігаючи при цьому модульність та розподіл відповідальності між компонентами.

Для доступу до даних сервіси використовують репозиторії через патерн Unit of Work, представлений класом UnitOfWork, який координує роботу конкретних репозиторіїв та забезпечує атомарність транзакцій. Репозиторії, такі як UserRepository, FamilyRepository та інші, абстрагують логіку доступу до даних, надаючи сервісам інтерфейс для виконання CRUD-операцій над відповідними сутностями. Кожен репозиторій є вузькоспеціалізованим компонентом, що працює з конкретним типом сутностей та реалізує методи для їх пошуку, фільтрації та маніпуляції.

Останній шар архітектури, представлений класом ApplicationDbContext, безпосередньо взаємодіє з базою даних через Entity Framework Core. Цей компонент відповідає за маппінг сутностей на таблиці бази даних, виконання запитів та керування транзакціями. ApplicationDbContext містить конфігурацію моделей даних та визначає відношення між ними, забезпечуючи цілісність даних на рівні бази даних.

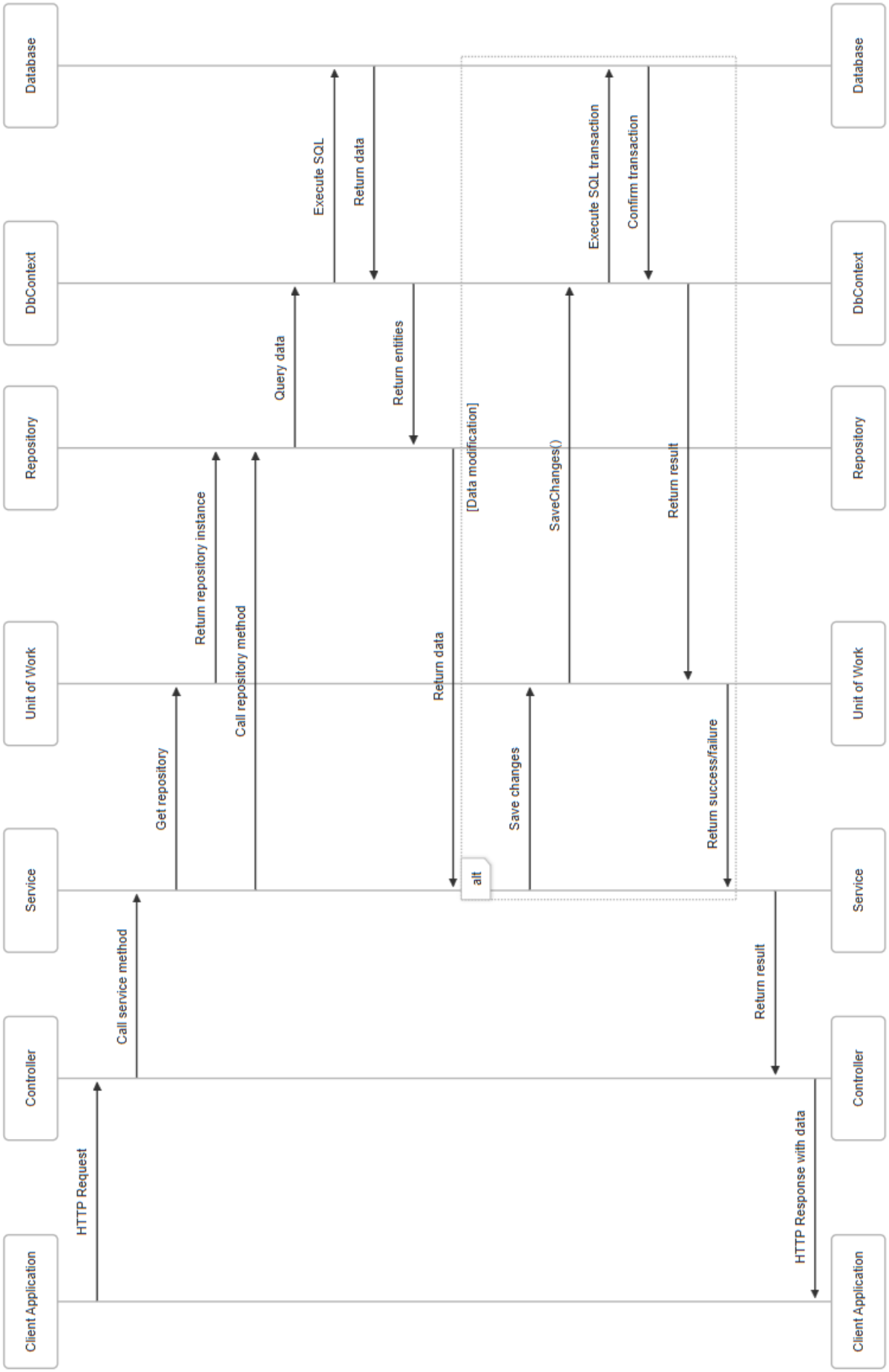


Рис. 3.6 Діаграма потоку обробки запиту

3.3 Розробка фронтенду

Клієнтська частина застосунку реалізована з використанням бібліотеки React у поєднанні з TypeScript та стилізована за допомогою React Bootstrap. Такий стек технологій забезпечує створення адаптивного, компонентного та зручного у користуванні інтерфейсу, який легко розширювати та підтримувати.

Фронтенд-архітектура додатку FunFinance базується на компонентному підході React, що дозволяє розділити інтерфейс на незалежні, повторно використовувані частини та працювати з ними окремо. Використання TypeScript замість звичайного JavaScript надає значні переваги в контексті типізації, що забезпечує раннє виявлення помилок на етапі розробки та покращує загальну надійність коду.

Структура клієнтської частини організована за принципом функціонального розподілу, де компоненти, сервіси та моделі згруповані за їхнім призначенням, а не за технічними характеристиками. Така організація підвищує зручність навігації по кодовій базі та сприяє кращому розумінню архітектури проєкту новими розробниками.

Для управління станом додатку використовується поєднання React Context API та локального стану компонентів. Цей підхід дозволяє ефективно ділитися даними між компонентами без надмірного ускладнення архітектури, яке могло б виникнути при використанні більш комплексних рішень для управління станом, таких як Redux.

Інтерфейс FunFinance розроблений з урахуванням принципів зручності використання (usability) та адаптивного дизайну, що дозволяє користувачам комфортно працювати з додатком як на настільних комп'ютерах, так і на мобільних пристроях.

Перший екран, з яким взаємодіє користувач — це форма входу/реєстрації. Дизайн форми мінімалістичний та інтуїтивно зрозумілий, що полегшує процес початку роботи з додатком.

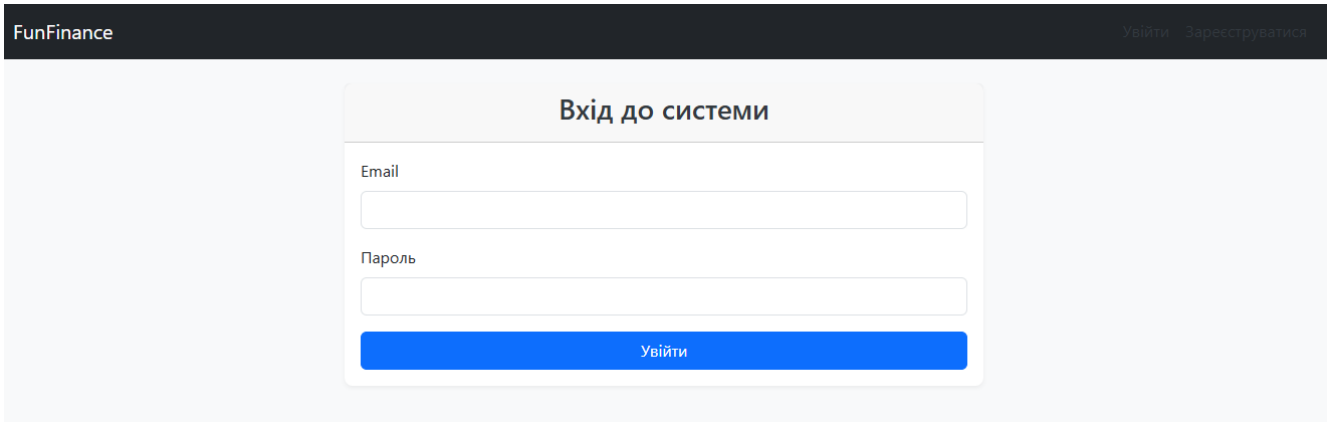


Рис. 3.7. Форма входу в систему

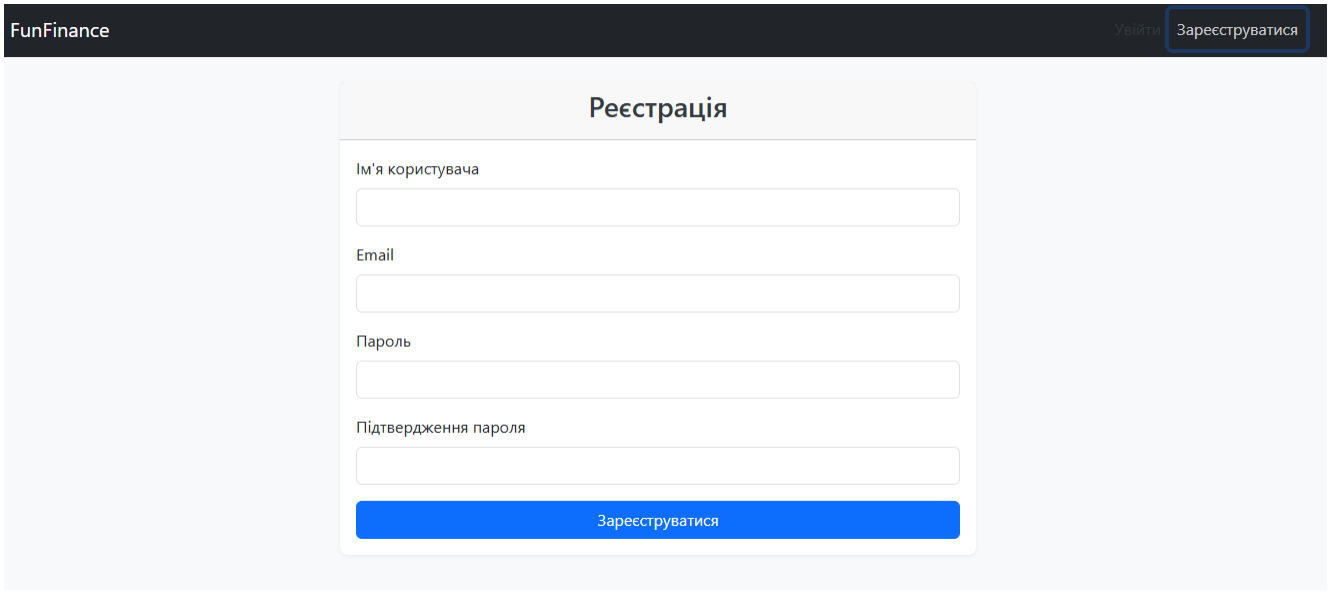


Рис. 3.8 Форма реєстрації в системі

Реалізація форми включає валідацію введених даних на стороні клієнта, що допомагає запобігти надсиланню некоректних даних на сервер та забезпечує миттєвий зворотний зв'язок для користувача.

Після успішної автентифікації користувач потрапляє на головну сторінку додатку.

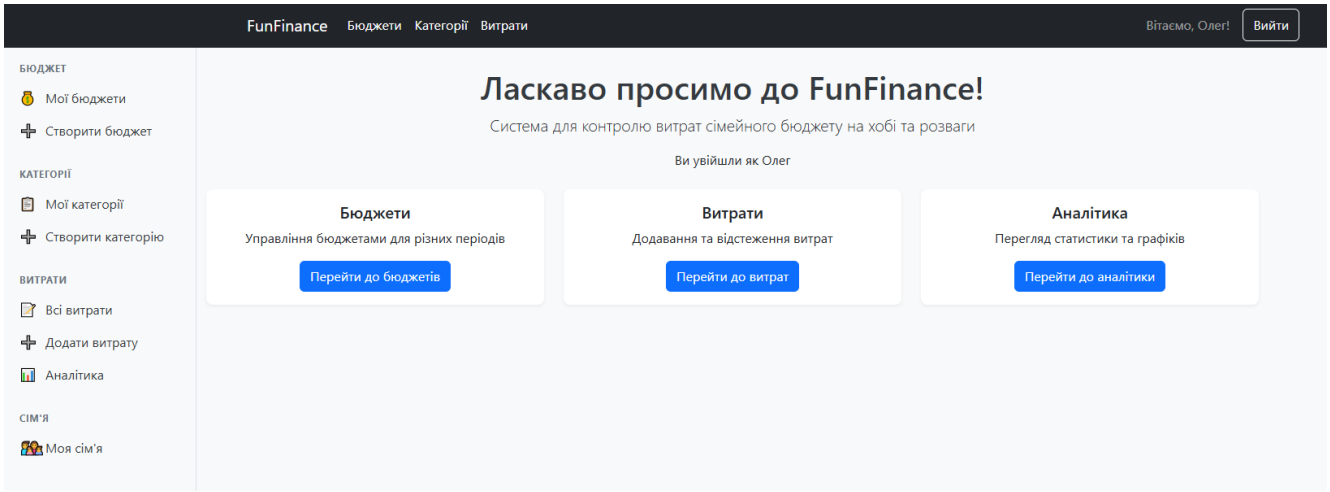


Рис. 3.9. Головна сторінка додатку

Навігаційна панель, розташована зліва (на десктопних пристроях) або зверху (на мобільних пристроях), забезпечує зручний доступ до всіх основних функцій додатку. Дизайн навігації реалізований з використанням компонентів React Bootstrap, що гарантує консистентність зовнішнього вигляду та поведінки на різних пристроях

Для ефективного обліку витрат у сімейному контексті, було розроблено функціонал створення та управління сімейними групами. Користувач може створити сім'ю, запросити інших користувачів приєднатися та спільно керувати бюджетами.

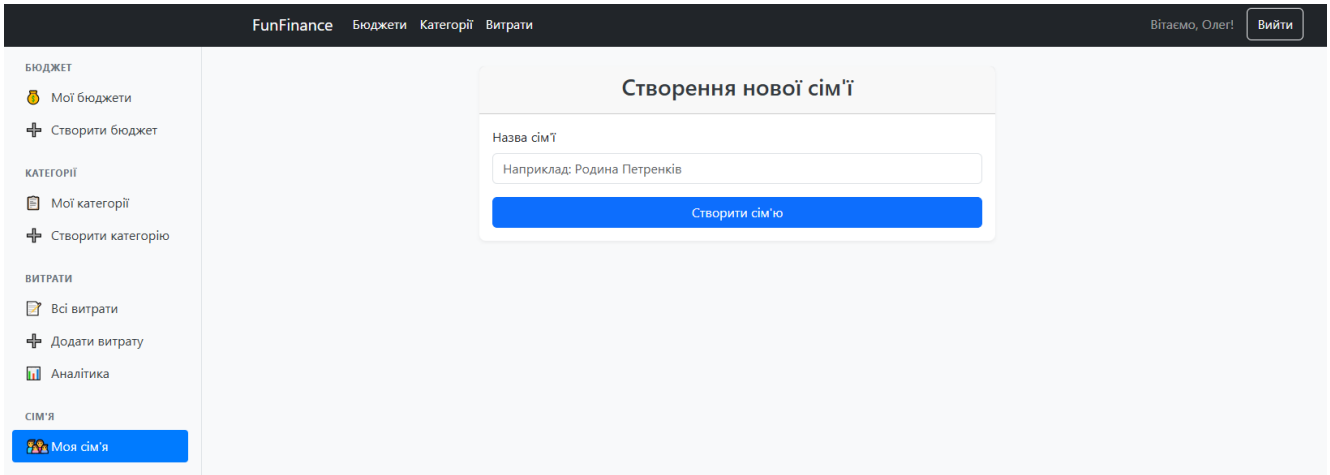


Рис. 3.10. Форма створення сім’ї

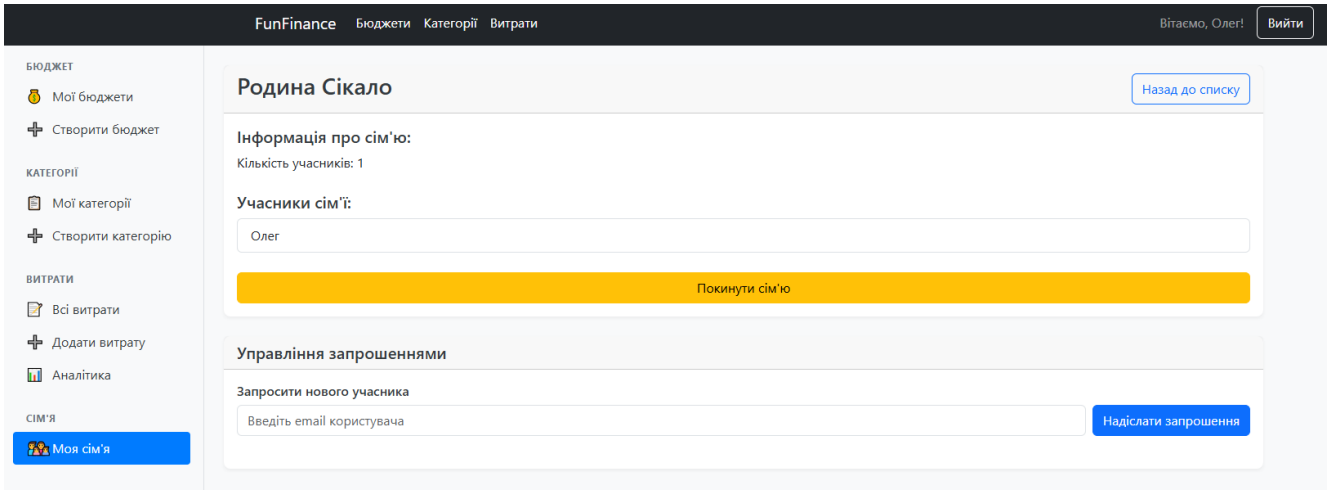


Рис. 3.11 Сторінка перегляду учасників сім’ї

Процес запрошення нових членів реалізований через систему запрошень, де користувач вводить email адресу запрошуваної особи, і система генерує унікальне посилання для приєднання.

Центральною функцією додатку є створення та управління бюджетами на хобі та розваги. Інтерфейс дозволяє користувачу створювати бюджети на певний період, встановлювати ліміти витрат та відстежувати поточне використання коштів.

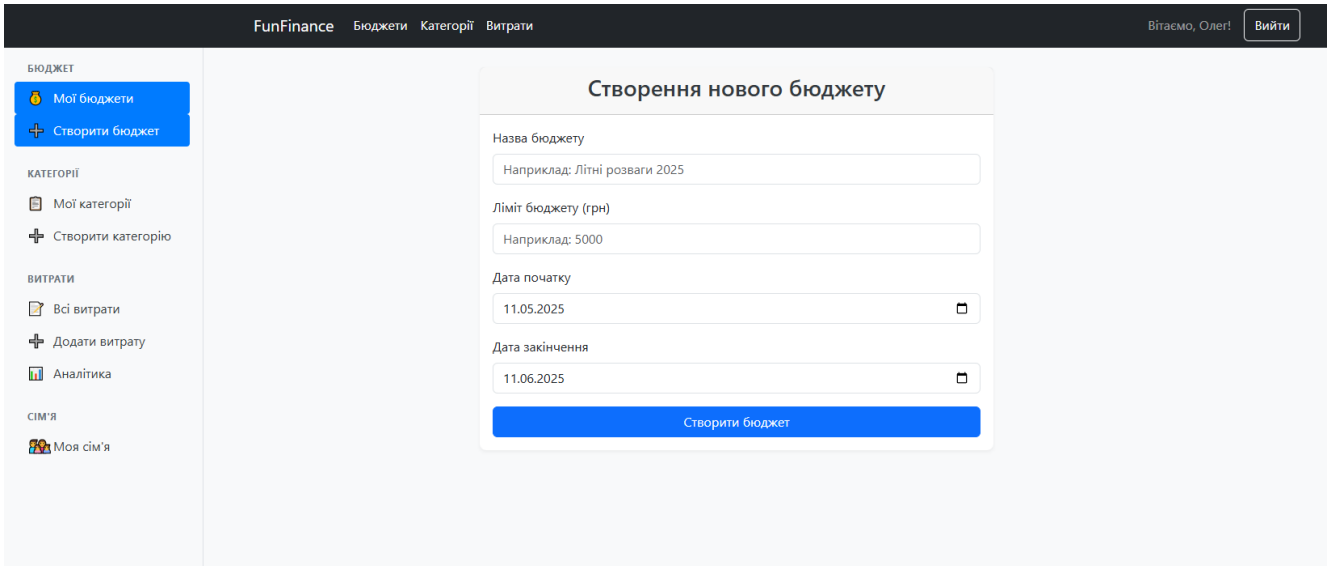


Рис. 3.12 Форма створення бюджету

Кожен бюджет візуалізується у вигляді картки, що містить ключову інформацію: назву, період дії, встановлений ліміт, поточне використання та прогрес у вигляді прогрес-бару.

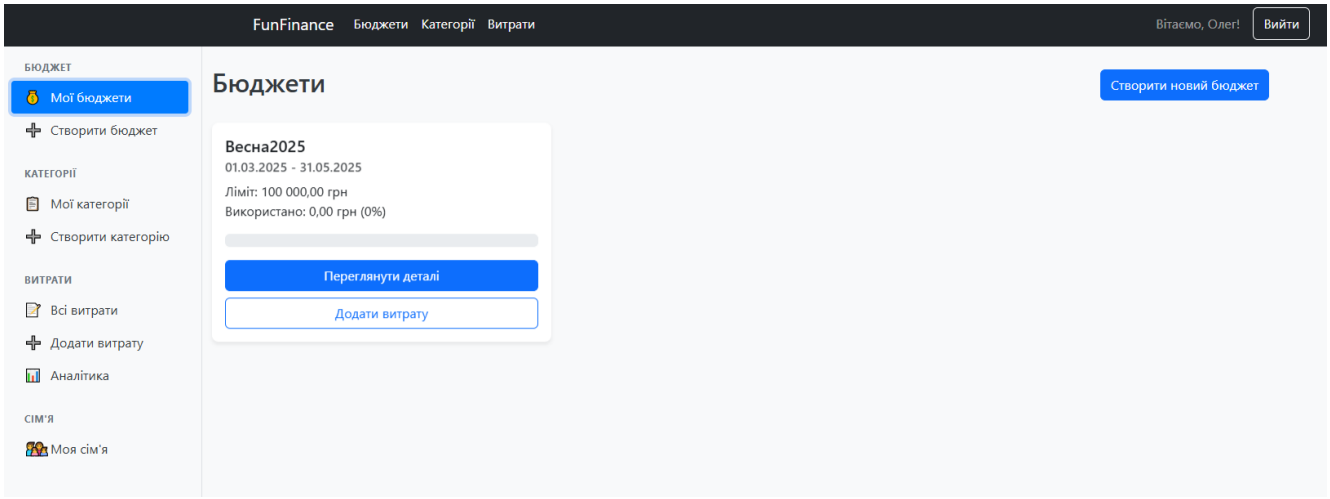


Рис. 3.13 Сторінка бюджетів

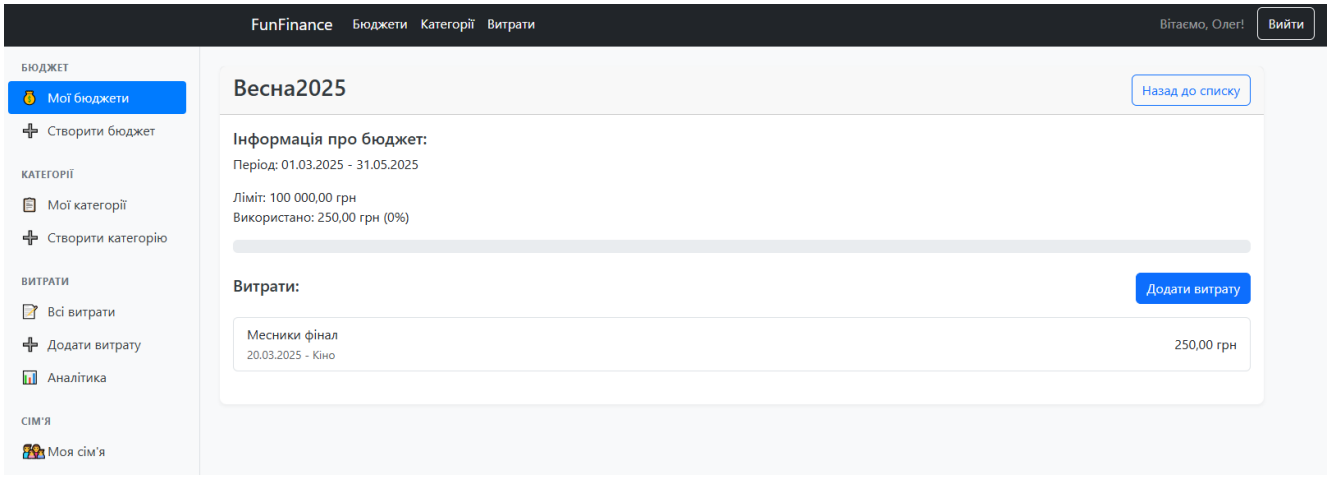


Рис. 3.14 Сторінка інформації бюджету

Для структурованого обліку витрат, у системі передбачено функціонал створення та управління категоріями витрат. Користувач може створювати власні категорії, які відповідають його специфічним потребам у хобі та розвагах.

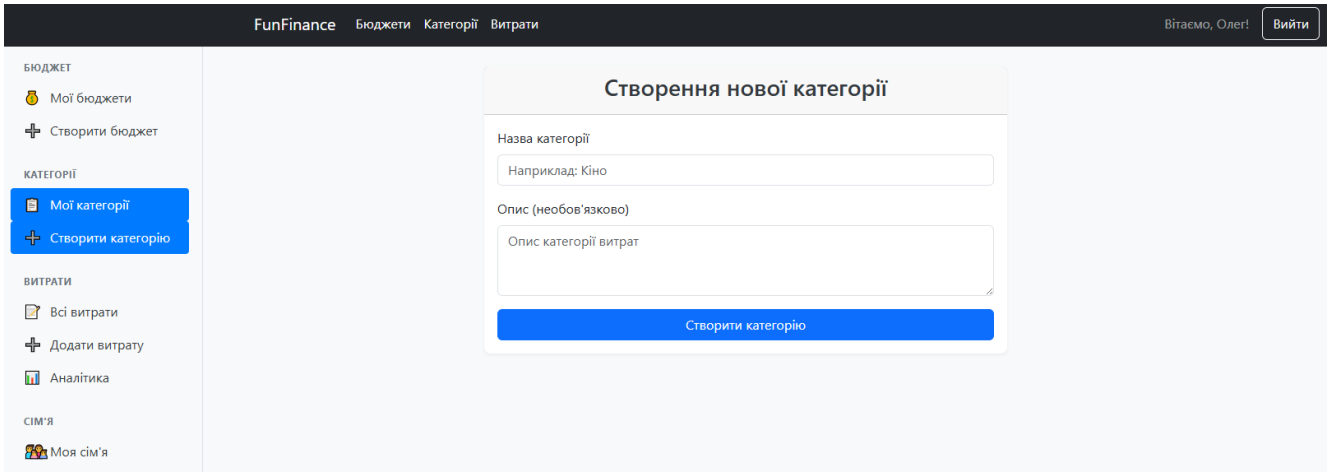


Рис 3.15 Форма створення категорії

Категорії відображаються як інтерактивні елементи, які можна обирати при додаванні нових витрат. Для кращої візуальної диференціації, кожна категорія має свій колір, що також використовується при побудові діаграм розподілу витрат.

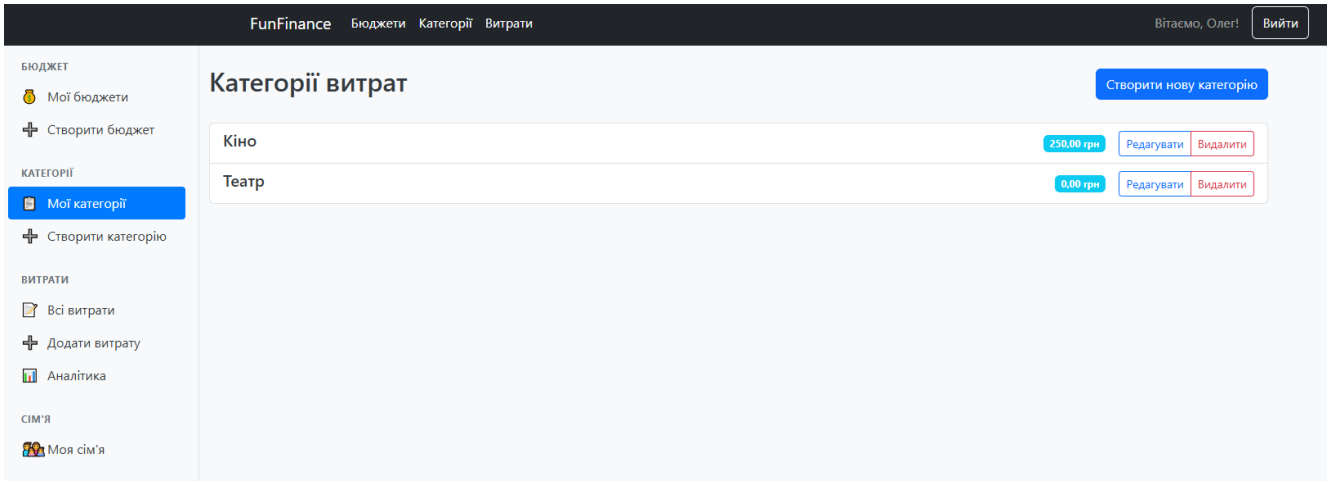


Рис. 3.16 Сторінка категорій витрат

Ключовою функцією додатку є облік витрат на хобі та розваги. Інтерфейс дозволяє користувачу легко додавати нові витрати, вказуючи суму, опис, категорію та дату.

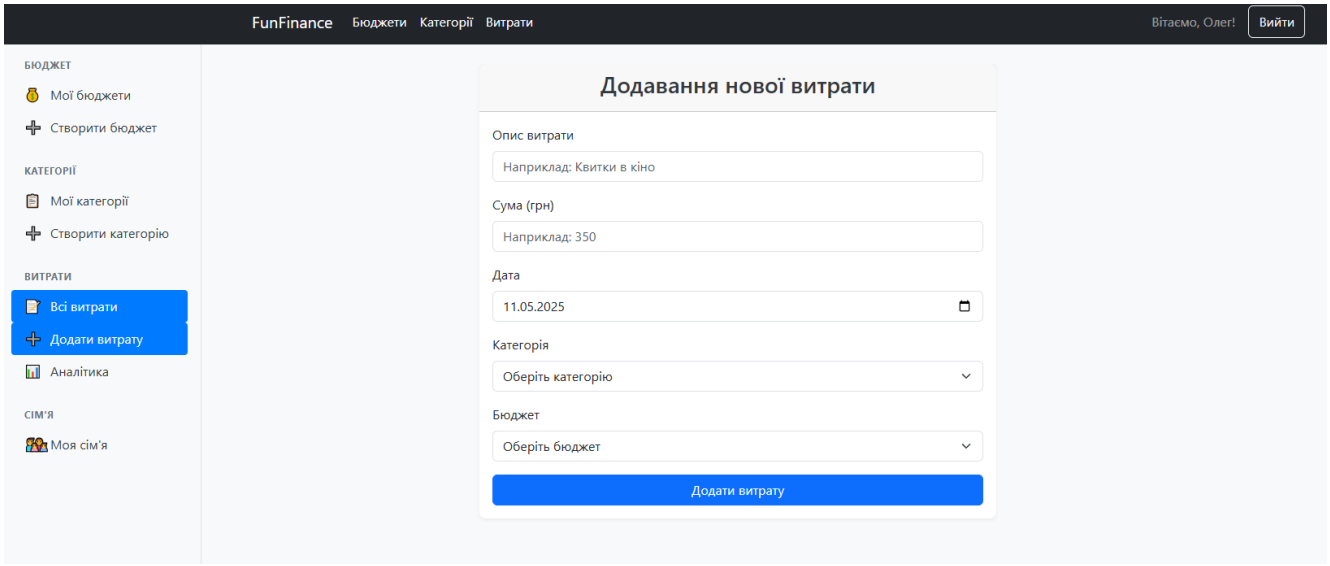


Рис. 3.17 Форма створення витрати

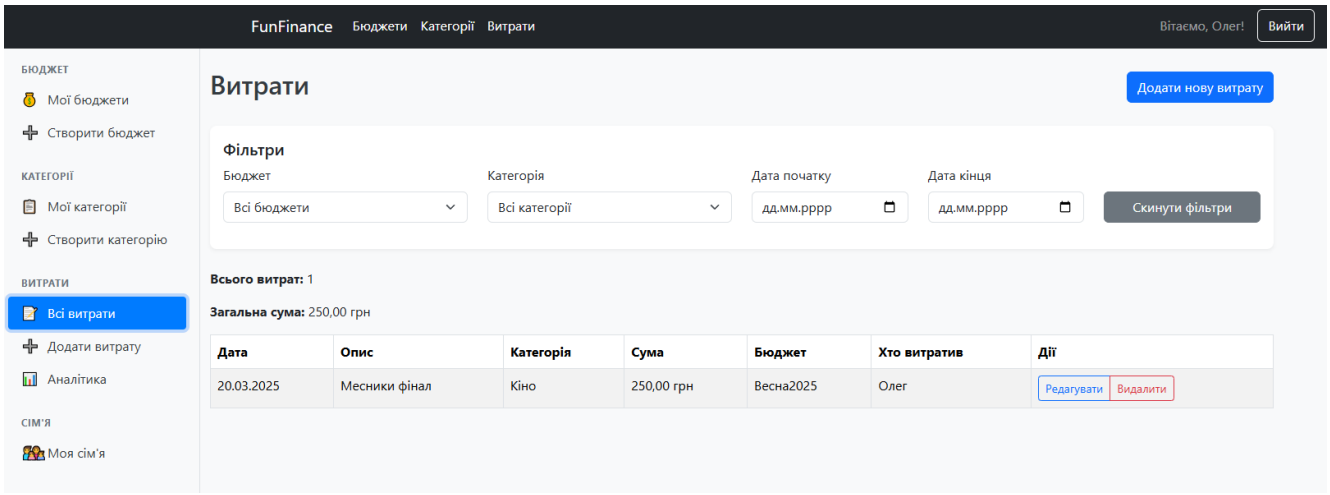


Рис. 3.18 Сторінка витрат

Для аналізу витрат розроблено набір інтерактивних візуалізацій, які допомагають користувачам зрозуміти структуру їхніх витрат та виявити тренди. Використано бібліотеку Recharts для створення діаграм, які відображають розподіл витрат за категоріями, динаміку витрат за часом та порівняння витрат між членами сім'ї.

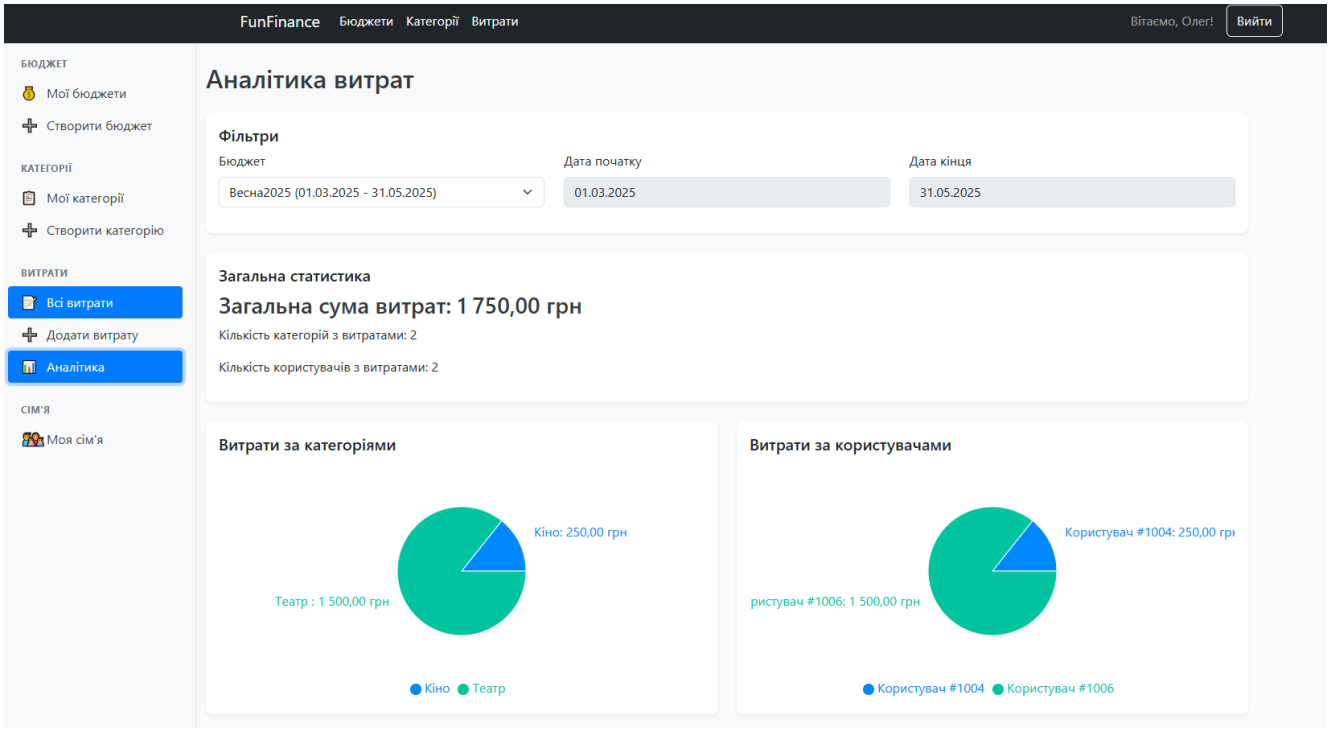


Рис. 3.19 Сторінка аналітики витрат

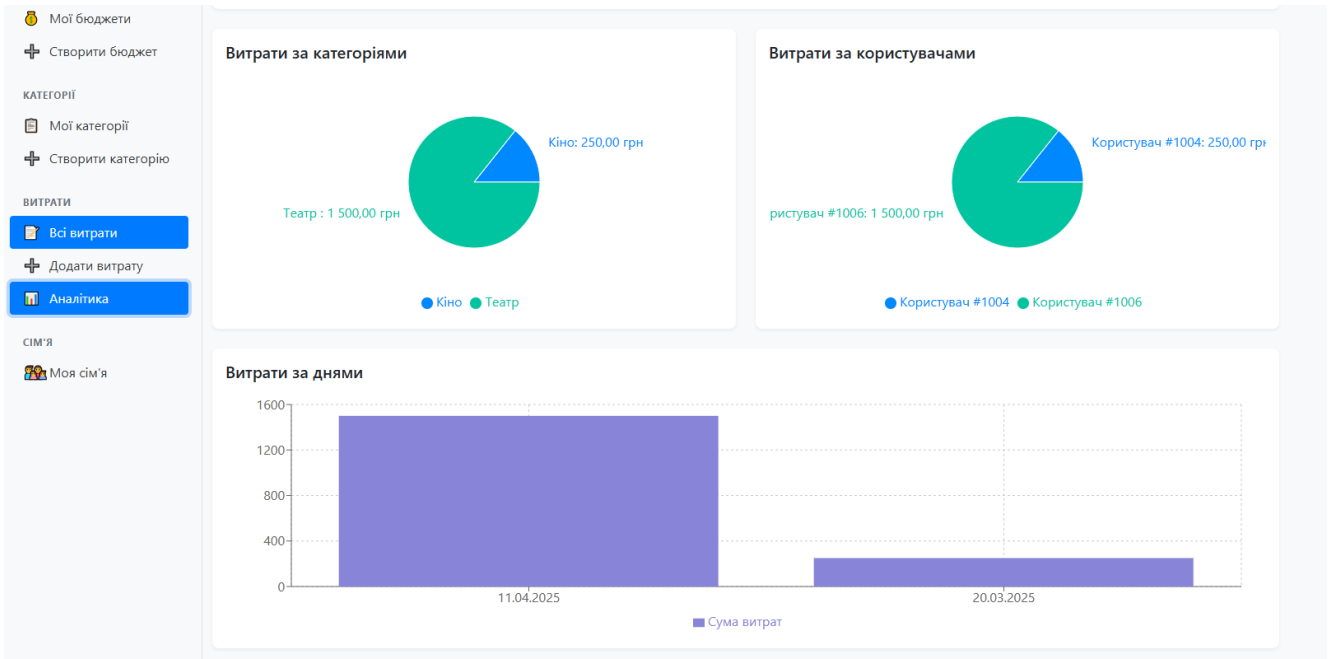


Рис. 3.20 Сторінка аналітики витрат

Особлива увага приділена реалізації адаптивного дизайну, який забезпечує оптимальне відображення інтерфейсу на пристроях з різними розмірами екрану. Використання сіткової системи React Bootstrap спрощує створення гнучких макетів, які автоматично адаптуються до розміру екрану.

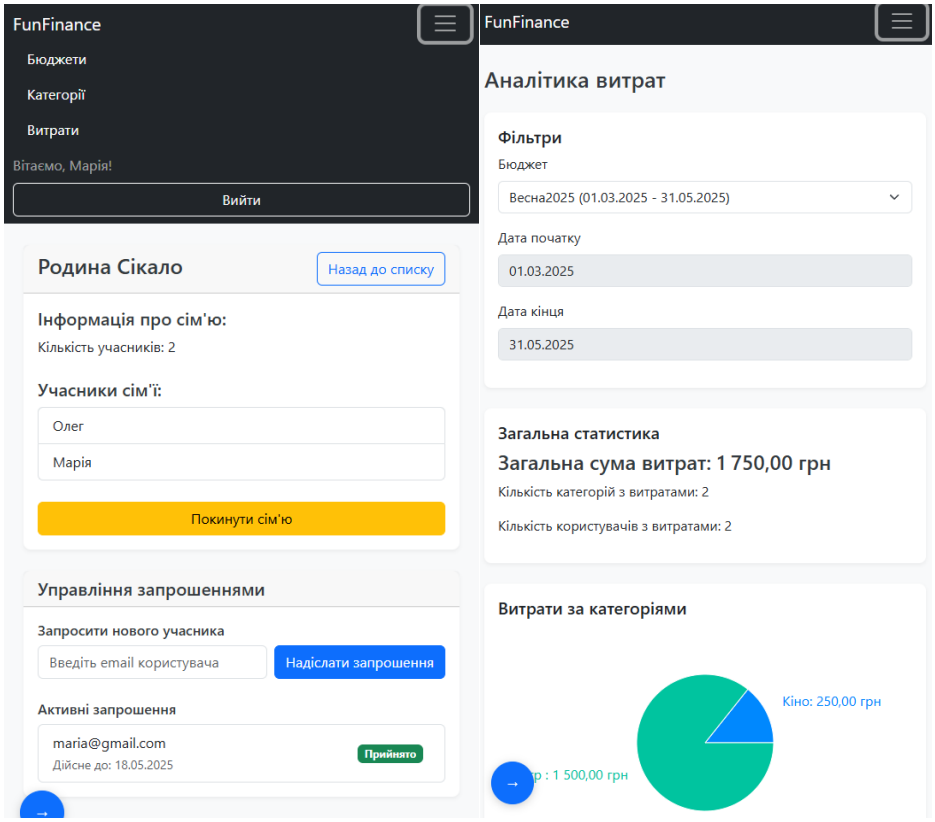


Рис. 3.21 Демонстрація адаптивності інтерфейсу

Розроблений фронтенд для системи FunFinance забезпечує інтуїтивно зрозумілий, адаптивний та функціональний інтерфейс для обліку та аналізу витрат на хобі та розваги. Використання сучасних технологій та архітектурних підходів дозволило створити продукт, який відповідає сучасним стандартам веб-розробки.

Обраний стек технологій (React, TypeScript, React Bootstrap) надає оптимальний баланс між продуктивністю розробки та якістю кінцевого продукту. Компонентний підхід React дозволяє легко масштабувати та розширювати функціональність інтерфейсу, а використання TypeScript підвищує надійність коду та полегшує його підтримку.

Особлива увага до користувацького досвіду, реалізація адаптивного дизайну та фокус на продуктивності роблять додаток зручним для користувачів різного рівня технічної підготовки та пристроїв з різними характеристиками. Система візуалізації даних у вигляді інтерактивних діаграм та графіків надає користувачам потужні інструменти для аналізу власних витрат та оптимізації сімейного бюджету на розваги.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОЦІНКА ЙОГО ЕФЕКТИВНОСТІ В УМОВАХ ПРАКТИЧНОГО ВИКОРИСТАННЯ

4.1 Методологія тестування

Після завершення етапу розробки програмного забезпечення важливою та обов'язковою частиною життєвого циклу інформаційної системи є її тестування та оцінка ефективності. Саме цей етап дозволяє не лише переконатися у коректності реалізації заявлених функціональних можливостей, а й оцінити, наскільки система є надійною, продуктивною та зручною для кінцевого користувача.

Метою тестування є перевірка відповідності розробленої системи до первинно сформульованих вимог, зокрема як функціональних, так і нефункціональних. Важливо встановити, чи справляється система з передбаченими сценаріями використання, чи не виникають критичні помилки, чи коректно обробляються виняткові ситуації, а також чи не порушується логіка взаємодії між компонентами.

Оцінка ефективності, у свою чергу, охоплює аналіз таких характеристик як швидкодія, масштабованість, ергономіка інтерфейсу, безпека, стабільність функціонування у типовому середовищі розгортання. Застосунок має не лише виконувати покладені на нього завдання, а й робити це швидко, передбачувано, без збоїв та із задовільним рівнем зручності для користувача.

Також у рамках тестування перевіряється наскільки логічно організований інтерфейс, чи зручно користувачу здійснювати основні дії (наприклад, додати витрату, створити бюджет, змінити категорію), як система реагує на помилки введення, відсутність інтернет-з'єднання або інші нештатні ситуації. Це дозволяє виявити потенційні недоліки ще до впровадження системи в реальне середовище.

Таким чином, тестування є невід'ємною складовою процесу розробки, що забезпечує якість, надійність і практичну цінність програмного забезпечення. Саме завдяки тестуванню вдається гарантувати, що розроблений застосунок не лише

функціонує згідно з технічним завданням, а й є дійсно корисним, зручним і ефективним інструментом для обліку витрат на хобі та розваги в межах сімейного бюджету.

Для перевірки якості розробленого застосунку було використано ручне функціональне тестування, що охоплює основні сценарії взаємодії користувача із системою. Тестування проводилося з урахуванням функціональних вимог, визначених у розділі 2.1.1.

Було протестовано:

- аутентифікацію користувачів (реєстрація, вхід, відновлення паролю);
- керування сім'єю (створення, запрошення, видалення користувачів);
- створення та редагування бюджетів;
- керування категоріями витрат;
- додавання, редагування, видалення витрат;
- відображення графіків аналітики.

Кожен тестовий сценарій оцінювався за критеріями: правильність виконання, реакція системи на помилки, швидкість обробки запитів, коректність відображення в інтерфейсі.

4.2 Результати тестування

Під час тестування було виявлено декілька незначних помилок, пов'язаних із валідацією форм (наприклад, неправильна обробка порожніх полів при створенні бюджету), які були виправлені в процесі налагодження.

У результаті проведеного тестування було перевірено всі основні функціональні блоки застосунку відповідно до вимог, визначених на етапі проєктування. Основна увага приділялась стабільності роботи системи, правильності обробки вхідних даних, відповідності отриманих результатів очікуваним, а також реакції на типові помилки користувача.

Під час тестування сценаріїв взаємодії з інтерфейсом (включаючи реєстрацію, вхід, створення бюджету, додавання витрат, перегляд аналітики тощо)

система продемонструвала стабільну поведінку. Усі основні функції були виконані без критичних збоїв. Виявлені незначні помилки, як-от некоректна обробка порожніх полів при створенні нової категорії або неправильне повідомлення про помилку під час запиту з неіснуючим ідентифікатором, були усунуті на етапі налагодження.

Система також коректно реагувала на недопустимі дії користувача, наприклад, спробу додати витрату без вибраного бюджету або за межами дозволеного діапазону дат. У таких випадках користувач отримував інформативне повідомлення про помилку, а дані не записувалися у базу. Особливу увагу було приділено перевірці аутентифікації: механізм входу з неправильними обліковими даними працював надійно, не розкриваючи сторонній особі деталей про існування акаунтів.

Продуктивність системи також показала задовільні результати. Запити до API виконувалися швидко, у середньому протягом 300–400 мс при роботі в локальному середовищі. Інтерфейс оновлювався миттєво, а графіки аналітики будувались без помітних затримок. Усі дії користувача відслідковувались логами, що дозволяло швидко локалізувати потенційні помилки або нестандартні ситуації.

Таким чином, результати тестування свідчать про те, що програмне забезпечення працює стабільно, передбачувано і відповідає сформульованим вимогам. Застосунок готовий до використання в реальному середовищі за умови подальшого розгортання на сервері з відповідною інфраструктурою.

4.3 Оцінка ефективності системи

Ефективність програмного забезпечення оцінювалася за наступними критеріями:

Зручність використання — інтерфейс є інтуїтивно зрозумілим, адаптивним, з мінімальною кількістю дій для виконання основних операцій. Усі ключові функції доступні за 2–3 кліки.

Швидкодія — система обробляє запити та оновлює дані майже миттєво, що особливо важливо при роботі з бюджетами та витратами в режимі реального часу.

Точність — усі дані бюджету, витрат і категорій обчислюються та відображаються правильно. Відсутні розбіжності між введеними й відображеними значеннями.

Масштабованість — завдяки застосованій архітектурі система може бути легко адаптована для обробки більшої кількості користувачів і додаткових функцій без потреби в глобальних змінах структури.

Безпека — система не допускає несанкціонованого доступу до приватних даних. Паролі зберігаються у хешованому вигляді, передача даних здійснюється через HTTPS.

Система продемонструвала високу ефективність у виконанні поставлених завдань. Завдяки використанню сучасних технологій (ASP.NET Core, React, Entity Framework Core) та багатошарової архітектури вдалося досягти чіткого розділення відповідальностей між компонентами, що позитивно вплинуло на швидкодію та стабільність роботи. Навіть при моделюванні активної взаємодії кількох користувачів одночасно, застосунок не демонстрував критичних збоїв чи затримок. Операції з додавання, редагування та фільтрації даних виконувались оперативно, без відчутного навантаження на систему. Застосунок є адаптивним, легко масштабованим та зручним у використанні, що підтверджує його придатність для впровадження у реальних умовах.

Результати тестування підтвердили, що реалізоване програмне забезпечення відповідає функціональним і нефункціональним вимогам, визначеним на етапі проєктування. Система є стабільною, продуктивною та зручною для використання. Застосунок успішно виконує поставлену мету — забезпечення ефективного контролю витрат сімейного бюджету на хобі та розваги.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було реалізовано повний цикл розробки веб-застосунку для контролю витрат сімейного бюджету з фокусом на хобі та розваги. Проведено всебічний аналіз предметної галузі, вивчено існуючі програмні рішення, визначено їхні сильні та слабкі сторони, що дозволило сформувати обґрунтовані вимоги до майбутнього продукту.

На основі сформульованих функціональних та нефункціональних вимог було спроєктовано архітектуру системи з використанням підходу N-Layer, що забезпечує розділення відповідальностей і спрощує підтримку та масштабування застосунку. Було реалізовано серверну частину на платформі ASP.NET Core 8, клієнтську частину з використанням React та TypeScript, а для зберігання даних використано SQL Server. Особливу увагу приділено зручності інтерфейсу та візуалізації фінансових даних у вигляді інтерактивних графіків і діаграм.

Розроблений застосунок підтримує основні сценарії взаємодії: створення облікового запису, формування сімей, додавання бюджетів, категорій, витрат, а також перегляд аналітики. Система дозволяє ефективно відстежувати витрати, виявляти перевищення бюджетів та приймати обґрунтовані фінансові рішення в межах сімейного планування.

Проведене тестування показало стабільність роботи програмного забезпечення, високу швидкодію, відповідність заявленим функціональним вимогам і зручність використання інтерфейсу. Визначено, що застосунок придатний до використання в реальних умовах і має потенціал для подальшого розширення, зокрема шляхом впровадження планування доходів, спільного використання категорій, персоналізації рекомендацій або інтеграції з банківськими API.

Таким чином, розроблене програмне забезпечення відповідає сучасним вимогам до систем обліку фінансів, забезпечує зручний та наочний контроль витрат на дозвілля та може бути корисним інструментом для формування фінансової грамотності в рамках сім’ї.

Апробація результатів дослідження:

1.Сікало О.С., Довженко Т.П., Розробка програмного забезпечення для контролю витрат сімейного бюджету на хобі та розваги, VI Всеукраїнська науково-технічна конференція(Застосування програмного забезпечення в інформаційнійно-комунікаційних технологіях), Київ, Україна, 4 квітня 2025 року , с. 76-78.

2.Сікало О.С., Довженко Т.П., Безпека у розробці програмного забезпечення для контролю витрат: актуальні загрози та підходи до захисту, Збірник тез XII Всеукраїнської науково-практичної конференції молодих учених, Київ, Україна, 15 травня 2025 року, с. 325-326.

ПЕРЕЛІК ПОСИЛАНЬ

1. Simple way to manage personal finances. Money Lover: веб-сайт. URL: https://moneylover.me/ (дата звернення: 03.03.2025)
2. Take back control of your money. Monefy: веб-сайт. URL: https://monefy.com/ (дата звернення: 03.03.2025)
3. 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)
4. REST API як спосіб спілкування компонент веб-додатків. Foxminded: веб-сайт. URL: https://foxminded.ua/shcho-take-rest-api/ (дата звернення: 05.03.2025)
5. Introduction to JSON Web Tokens. JWT Debugger: веб-сайт. URL: https://jwt.io/introduction (дата звернення: 05.03.2025)
6. Entity Framework Core. Learn.microsoft: веб-сайт. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 06.03.2025)
7. EF Core Migrations. Learn Entity Framework Core: веб-сайт. URL: https://www.learnentityframeworkcore.com/migrations (дата звернення: 06.03.2025)
8. Microsoft SQL Server. Microsoft: веб-сайт. URL: https://www.microsoft.com/en-us/sql-server (дата звернення: 07.03.2025)
9. React The library for web and native user interfaces. React: веб-сайт. URL: https://react.dev/ (дата звернення: 07.03.2025)
10. 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)
11. 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)

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

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для контролю витрат сімейного бюджету на хобі та розваги

Виконав студент 4 курсу
групи ПД-42
Сікало Олег Сергійович
Керівник роботи
Доцент кафедри, кандидат технічних наук Довженко Тимур Павлович

Київ – 2025

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: оптимізація контролю витрат в межах сімейного бюджету на хобі, та розваги, за допомогою програмного забезпечення, з можливістю візуального управління фінансовими даним.

Об’єкт дослідження: процес обліку та аналізу витрат у сфері дозвілля в межах сім’ї.

Предмет дослідження: програмне забезпечення для ведення обліку сімейного бюджету з акцентом на витрати на хобі та розваги.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Проаналізувати сучасні програмні рішення для управління фінансами.
2. Визначити функціональні та нефункціональні вимоги до майбутнього застосунку.
3. Спроектувати архітектуру системи, розробити серверну та клієнтську частину.
4. Провести тестування.

АНАЛІЗ АНАЛОГІВ

	Спеціалізація для хобі та розваг	Створення власних категорій	Орієнтація на кількох користувачів
FanFinance	+	+	+
Money Lover	-	+	+
CoinKeeper	-	-	+
Monefy	+	+	-

ФУНКЦІОНАЛЬНІ ВИМОГИ ДО ДОДАТКУ

1. Аутентифікація та управління користувачами (Реєстрація та вхід / Управління профілем);
2. Управління сім'ями (Створення та редагування сім'ї / Членство в сім'ї);
3. Управління бюджетами (Створення та редагування бюджетів / Перегляд бюджетів);
4. Управління категоріями витрат (Створення та редагування категорій / Перегляд категорій);
5. Управління витратами (Додавання та редагування витрат / Перегляд витрат);
6. Візуалізація аналітики(Візуалізація даних);

НЕФУНКЦІОНАЛЬНІ ВИМОГИ ДО ДОДАТКУ

1. Продуктивність (Завантаження основних сторінок не повинно перевищувати 2 секунд);
2. Масштабованість(Система повинна підтримувати зростання кількості користувачів та обсягів даних без зниження продуктивності);
3. Безпека(Застосунок повинен дотримуватися сучасних стандартів захисту даних);

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

Visual Studio Code

React TypeScript

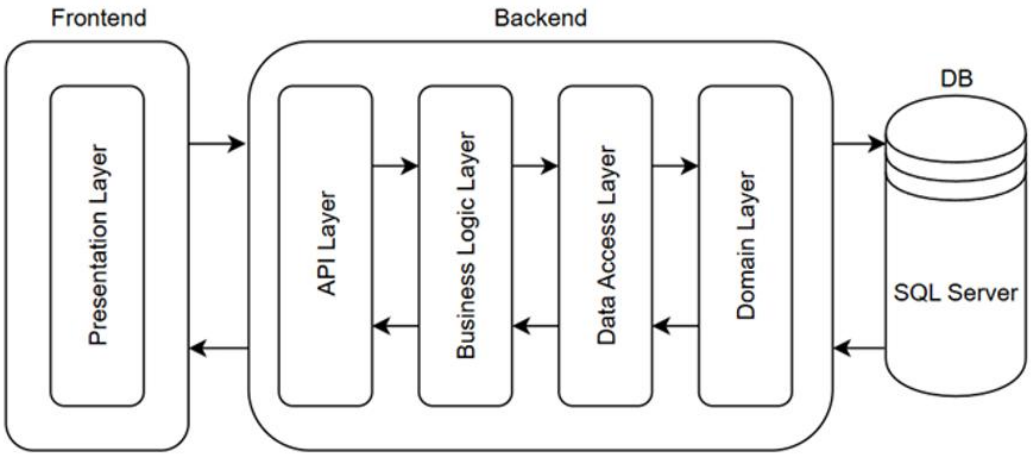
- ✓ **ASP.NET Core 8**
- ✓ **Entity Framework Core 8**

React Bootstrap

Visual Studio 2022

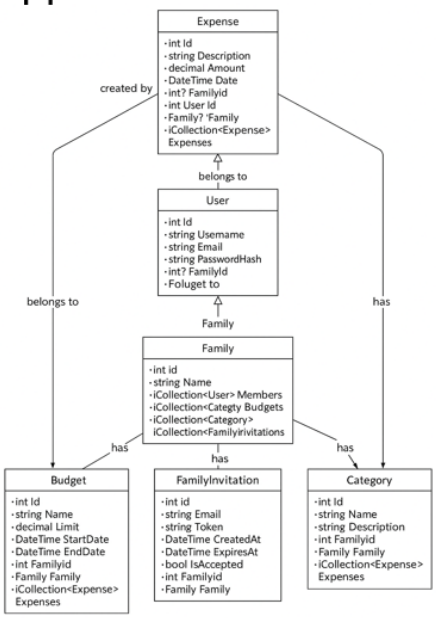
6

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ



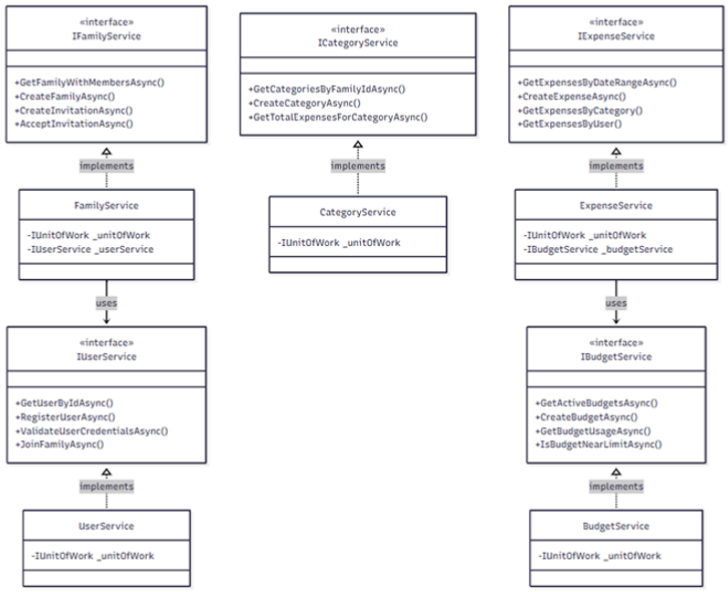
7

ДІАГРАМА ПРЕДМЕТНОЇ ГАЛУЗІ DOMAIN LAYER



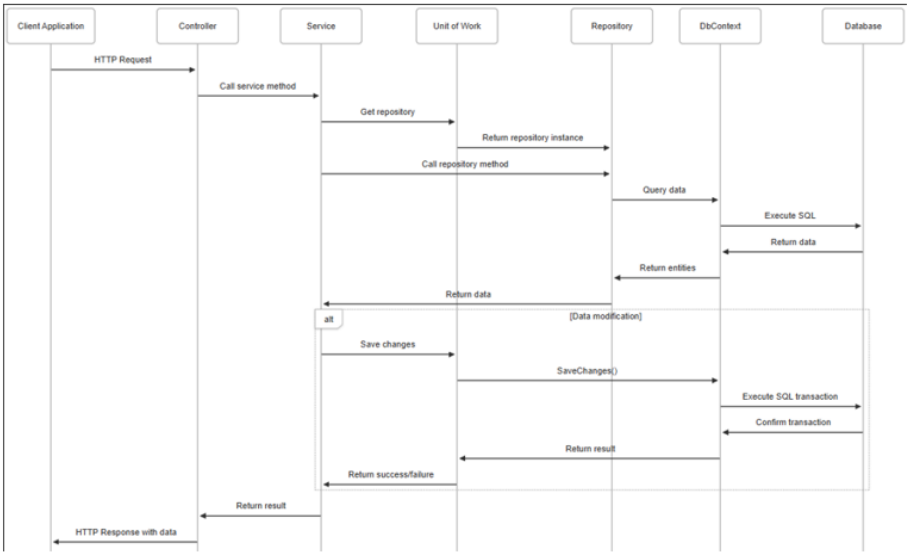
8

ДІАГРАМА КЛАСІВ ОСНОВНИХ СЕРВІСІВ



9

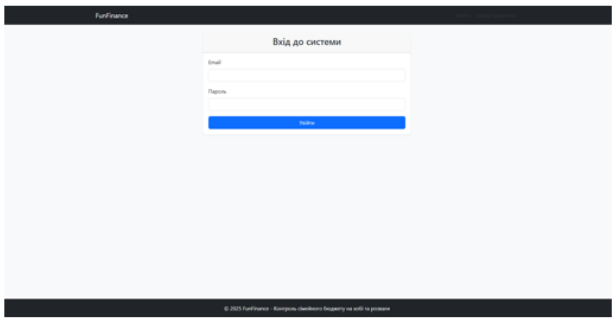
ДІАГРАМА ПОТОКУ ОБРОБКИ ЗАПИТУ



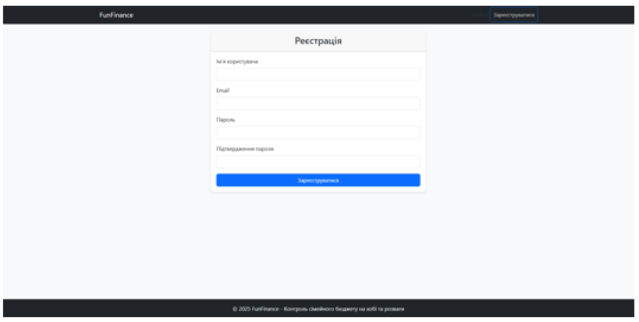
10

ЕКРАННІ ФОРМИ

ФОРМА ВХОДУ В СИСТЕМУ

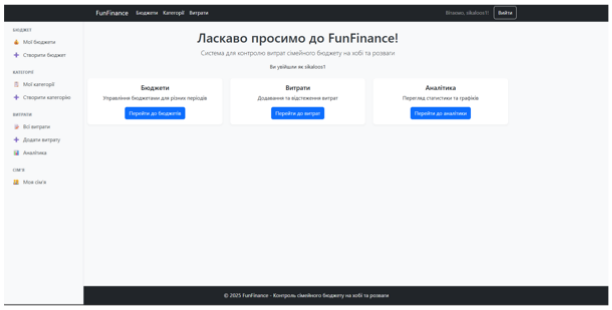


ФОРМА РЕЄСТРАЦІЇ В СИСТЕМІ

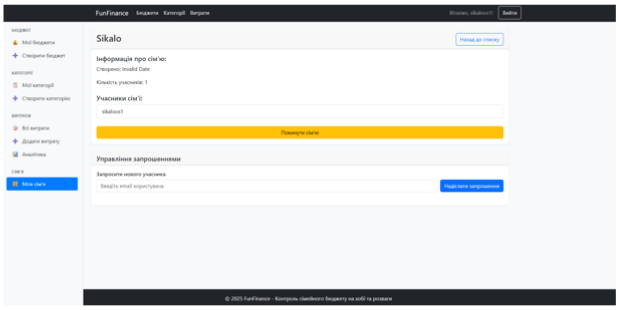


11

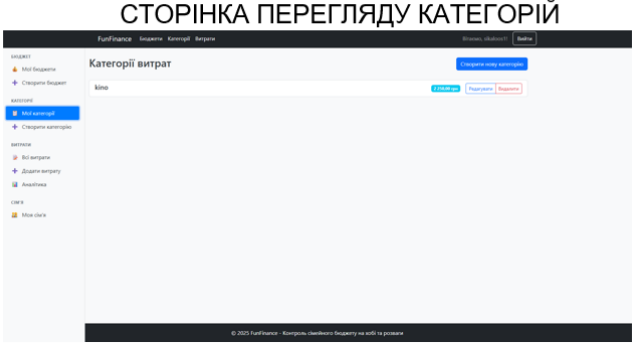
ГОЛОВНА СТОРІНКА ДОДАТКУ



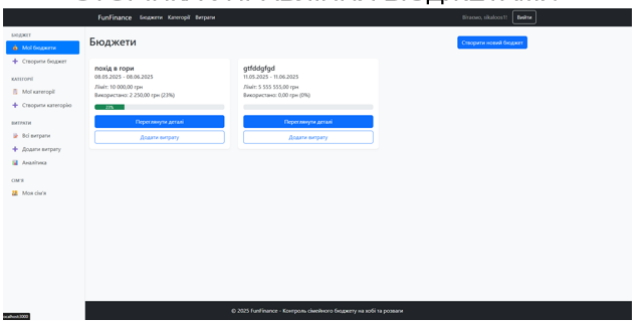
СТОРІНКА ПЕРЕГЛЯДУ УЧАСНИКІВ СІМ'Ї



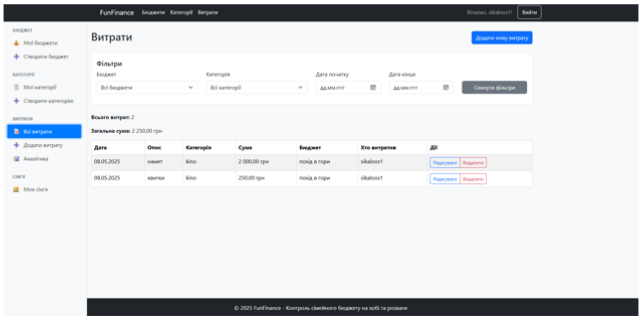
СТОРІНКА ПЕРЕГЛЯДУ КАТЕГОРІЙ



СТОРІНКА УПРАВЛІННЯ БЮДЖЕТАМИ



СТОРІНКА АНАЛІЗУ ВИТРАТ



13

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Сікало О.С., Довженко Т.П., Розробка програмного забезпечення для контролю витрат сімейного бюджету на хобі та розваги, VI Всеукраїнська науково-технічна конференція(Застосування програмного забезпечення в інформаційнійно-комунікаційних технологіях), Київ, Україна, 4 квітня 2025 року , с. 76-78.
2. Сікало О.С., Довженко Т.П., Безпека у розробці програмного забезпечення для контролю витрат: актуальні загрози та підходи до захисту, Збірник тез XII Всеукраїнської науково-практичної конференції молодих учених, Київ, Україна, 15 травня 2025 року, с. 325-326.

ВИСНОВКИ

1. Проаналізовано сучасні програмні рішення для управління фінансами.
2. Визначено функціональні та нефункціональні вимоги до майбутнього застосунку.
3. Спроєктовано архітектуру системи на основі N-Layer підходу, реалізовано серверну частину за допомогою ASP.NET Core 8, клієнтську частину – з використанням React та TypeScript, збереження даних – на платформі SQL Server.
4. Проведено функціональне тестування реалізованої системи, проаналізовано її ефективність, стабільність і зручність у практичному використанні.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

```
using System.ComponentModel.DataAnnotations;

namespace FunFinance.API.Models
{
    public class Budget
    {
        public int Id { get; set; }

        [Required, StringLength(50)]
        public required string Name { get; set; }

        [Required]
        public decimal Limit { get; set; }

        public DateTime StartDate { get; set; }
        public DateTime EndDate { get; set; }

        public int FamilyId { get; set; }

        public required virtual Family Family { get; set; }
        public virtual ICollection<Expense> Expenses { get; set; } = new List<Expense>();
    }
}

using System.ComponentModel.DataAnnotations;

namespace FunFinance.API.Models
{
    public class Category
    {
        public int Id { get; set; }

        [Required, StringLength(50)]
        public required string Name { get; set; }

        [StringLength(200)]
        public string? Description { get; set; }

        public int FamilyId { get; set; }

        public required virtual Family Family { get; set; }
        public virtual ICollection<Expense> Expenses { get; set; } = new List<Expense>();
    }
}

using System.ComponentModel.DataAnnotations;

namespace FunFinance.API.Models
{
    public class Expense
    {
        public int Id { get; set; }

        [Required, StringLength(100)]
        public required string Description { get; set; }

        [Required]
        public decimal Amount { get; set; }

        [Required]
        public DateTime Date { get; set; }

        public int CategoryId { get; set; }
        public int UserId { get; set; }
        public int BudgetId { get; set; }

        public required virtual Category Category { get; set; }
```

```
        public required virtual User User { get; set;
}
        public required virtual Budget Budget { get;
set; }
    }
}
using
System.ComponentModel.DataAnnotations;

namespace FunFinance.API.Models
{
    public class Family
    {
        public int Id { get; set; }

        [Required, StringLength(50)]
        public required string Name { get; set; }

        public virtual ICollection<User> Members
{ get; set; } = new List<User>();
        public virtual ICollection<Budget> Budgets
{ get; set; } = new List<Budget>();
        public virtual ICollection<Category>
Categories { get; set; } = new List<Category>();
        public virtual
ICollection<FamilyInvitation> Invitations { get; set; } =
new List<FamilyInvitation>();
    }
}

using
System.ComponentModel.DataAnnotations;

namespace FunFinance.API.Models
{
    public class FamilyInvitation
    {
        public int Id { get; set; }

        [Required, EmailAddress,
StringLength(100)]
        public required string Email { get; set; }

        [Required]
        public required string Token { get; set; }

        public DateTime CreatedAt { get; set; } =
DateTime.UtcNow;

        public DateTime ExpiresAt { get; set; }

        public bool IsAccepted { get; set; } = false;

        public int FamilyId { get; set; }

        public required virtual Family Family { get;
set; }
    }
}
using
System.ComponentModel.DataAnnotations;

namespace FunFinance.API.Models
{
    public class User
    {
        public int Id { get; set; }

        [Required, StringLength(50)]
        public required string Username { get; set;
}

        [Required, EmailAddress,
StringLength(100)]
        public required string Email { get; set; }

        [Required]
        public required string PasswordHash { get;
set; }

        public int? FamilyId { get; set; }
        public virtual Family? Family { get; set; }
```

```
        public virtual ICollection<Expense>
Expenses { get; set; } = new List<Expense>();
    }
}
using FunFinance.API.Models;
using FunFinance.API.Repositories;

namespace FunFinance.API.Services
{
    public class BudgetService : IBudgetService
    {
        private readonly IUnitOfWork
_unitOfWork;

        public BudgetService(IUnitOfWork
unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public async Task<IEnumerable<Budget>>
GetAllBudgetsAsync()
        {
            return await
_unitOfWork.Budgets.GetAllAsync();
        }

        public async Task<Budget?>
GetBudgetByIdAsync(int id)
        {
            return await
_unitOfWork.Budgets.GetByIdAsync(id);
        }

        public async Task<Budget?>
GetBudgetWithExpensesAsync(int id)
        {
            return await
_unitOfWork.Budgets.GetWithExpensesAsync(id);
        }

        public async Task<IEnumerable<Budget>>
GetBudgetsByFamilyIdAsync(int familyId)
        {
            return await
_unitOfWork.Budgets.GetByFamilyIdAsync(familyId);
        }

        public async Task<IEnumerable<Budget>>
GetActiveBudgetsAsync(int familyId)
        {
            return await
_unitOfWork.Budgets.GetActiveBudgetsAsync(familyId
);
        }

        public async Task<Budget>
CreateBudgetAsync(string name, decimal limit,
DateTime startDate, DateTime endDate, int familyId)
        {
            if (limit <= 0)
                throw new
ArgumentException("Ліміт бюджету повинен бути
більшим за нуль");

            if (startDate >= endDate)
                throw new ArgumentException("Дата
початку повинна бути раніше дати закінчення");

            var family = await
_unitOfWork.Families.GetByIdAsync(familyId);
            if (family == null)
                throw new
ArgumentException("Сім'ю не знайдено");

            var budget = new Budget
            {
                Name = name,
                Limit = limit,
                StartDate = startDate,
                EndDate = endDate,
                FamilyId = familyId,
                Family = family
```

```
            };

            await
_unitOfWork.Budgets.AddAsync(budget);
            await _unitOfWork.CompleteAsync();

            return budget;
        }

        public async Task<bool>
UpdateBudgetAsync(Budget budget)
        {
            _unitOfWork.Budgets.Update(budget);
            return                              await
_unitOfWork.CompleteAsync();
        }

        public async Task<bool>
DeleteBudgetAsync(int id)
        {
            var budget = await
_unitOfWork.Budgets.GetByIdAsync(id);
            if (budget == null)
                return false;

            _unitOfWork.Budgets.Remove(budget);
            return                              await
_unitOfWork.CompleteAsync();
        }

        public async Task<decimal>
GetBudgetUsageAsync(int budgetId)
        {
            return                              await
_unitOfWork.Budgets.GetTotalExpensesAsync(budgetId
);
        }

        public async Task<decimal>
GetRemainingBudgetAsync(int budgetId)
        {
            var budget = await
_unitOfWork.Budgets.GetByIdAsync(budgetId);
            if (budget == null)
                throw                           new
ArgumentException("Бюджет не знайдено");

            var totalExpenses = await
_unitOfWork.Budgets.GetTotalExpensesAsync(budgetId
);
            return budget.Limit - totalExpenses;
        }

        public async Task<bool>
IsBudgetExceededAsync(int budgetId)
        {
            var budget = await
_unitOfWork.Budgets.GetByIdAsync(budgetId);
            if (budget == null)
                throw                           new
ArgumentException("Бюджет не знайдено");

            var totalExpenses = await
_unitOfWork.Budgets.GetTotalExpensesAsync(budgetId
);
            return totalExpenses > budget.Limit;
        }

        public async Task<bool>
IsBudgetNearLimitAsync(int budgetId, decimal
thresholdPercentage = 80)
        {
            var budget = await
_unitOfWork.Budgets.GetByIdAsync(budgetId);
            if (budget == null)
                throw                           new
ArgumentException("Бюджет не знайдено");

            var totalExpenses = await
_unitOfWork.Budgets.GetTotalExpensesAsync(budgetId
);
            var usagePercentage = (totalExpenses /
budget.Limit) * 100;
```

```
            return usagePercentage >= thresholdPercentage && usagePercentage < 100;
        }
    }
}

using FunFinance.API.Models;
using FunFinance.API.Repositories;
using System.Security.Cryptography;
using System.Text;

namespace FunFinance.API.Services
{
    public class UserService : IUserService
    {
        private readonly IUnitOfWork _unitOfWork;

        public UserService(IUnitOfWork unitOfWork)
        {
            _unitOfWork = unitOfWork;
        }

        public async Task<IEnumerable<User>> GetAllUsersAsync()
        {
            return await _unitOfWork.Users.GetAllAsync();
        }

        public async Task<User?> GetUserByIdAsync(int id)
        {
            return await _unitOfWork.Users.GetByIdAsync(id);
        }

        public async Task<User?> GetUserByEmailAsync(string email)
        {
            return await _unitOfWork.Users.GetByEmailAsync(email);
        }

        public async Task<User> RegisterUserAsync(string username, string email, string password)
        {
            // Перевірка чи не зайняті email та username
            if (await EmailExistsAsync(email))
                throw new InvalidOperationException("Користувач з таким email вже існує");

            if (await UsernameExistsAsync(username))
                throw new InvalidOperationException("Користувач з таким логіном вже існує");

            // Хешування пароля
            var passwordHash = HashPassword(password);

            // Створення нового користувача
            var user = new User
            {
                Username = username,
                Email = email,
                PasswordHash = passwordHash,
            };

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

            return user;
        }
```

```
        public async Task<bool> ValidateUserCredentialsAsync(string email, string password)
        {
            var user = await _unitOfWork.Users.GetByEmailAsync(email);

            if (user == null)
                return false;

            var passwordHash = HashPassword(password);
            return user.PasswordHash == passwordHash;
        }

        public async Task<bool> UpdateUserAsync(User user)
        {
            _unitOfWork.Users.Update(user);
            return await _unitOfWork.CompleteAsync();
        }

        public async Task<bool> DeleteUserAsync(int id)
        {
            var user = await _unitOfWork.Users.GetByIdAsync(id);
            if (user == null)
                return false;

            _unitOfWork.Users.Remove(user);
            return await _unitOfWork.CompleteAsync();
        }

        public async Task<bool> JoinFamilyAsync(int userId, int familyId)
        {
            var user = await _unitOfWork.Users.GetByIdAsync(userId);
            if (user == null)
                return false;

            var family = await _unitOfWork.Families.GetByIdAsync(familyId);
            if (family == null)
                return false;

            user.FamilyId = familyId;
            user.Family = family;

            return await _unitOfWork.CompleteAsync();
        }

        public async Task<bool> LeaveFamilyAsync(int userId)
        {
            var user = await _unitOfWork.Users.GetByIdAsync(userId);
            if (user == null)
                return false;

            user.FamilyId = null;
            user.Family = null;

            return await _unitOfWork.CompleteAsync();
        }

        public async Task<bool> EmailExistsAsync(string email)
        {
            return await _unitOfWork.Users.EmailExistsAsync(email);
        }

        public async Task<bool> UsernameExistsAsync(string username)
        {
            return await _unitOfWork.Users.UsernameExistsAsync(username);
```

```
        }

        private string HashPassword(string password)
        {
            using (var sha256 = SHA256.Create())
            {
                var hashedBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(password));
                return BitConverter.ToString(hashedBytes).Replace("-", "").ToLower();
            }
        }
    }
}
using FunFinance.API.Models;
using FunFinance.API.Services;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace FunFinance.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class UsersController : ControllerBase
    {
        private readonly IUserService _userService;

        public UsersController(IUserService userService)
        {
            _userService = userService;
        }

        // GET: api/Users
        [HttpGet]
        public async Task<ActionResult<IEnumerable<User>>> GetUsers()
        {
            var users = await _userService.GetAllUsersAsync();
            return Ok(users);
        }

        // GET: api/Users/5
        [HttpGet("{id}")]
        public async Task<ActionResult<User>> GetUser(int id)
        {
            var user = await _userService.GetUserByIdAsync(id);

            if (user == null)
                return NotFound();

            return Ok(user);
        }

        // POST: api/Users/Register
        [HttpPost("Register")]
        public async Task<ActionResult<User>> RegisterUser([FromBody] UserRegisterDto registerDto)
        {
            try
            {
                var user = await _userService.RegisterUserAsync(
                    registerDto.Username,
                    registerDto.Email,
                    registerDto.Password);

                return CreatedAtAction(nameof(GetUser), new { id = user.Id }, user);
            }
            catch (InvalidOperationException ex)
            {
                return BadRequest(ex.Message);
```

```
        }
    }

    // POST: api/Users/Login
    [HttpPost("Login")]
    public async Task<ActionResult> Login([FromBody] UserLoginDto loginDto)
    {
        var isValid = await _userService.ValidateUserCredentialsAsync(
            loginDto.Email,
            loginDto.Password);

        if (!isValid)
            return Unauthorized("Невірний email або пароль");

        var user = await _userService.GetUserByEmailAsync(loginDto.Email);

        if (user == null)
            return Unauthorized("Користувача не знайдено");

        // В реальному додатку тут ми б генерували JWT токен
        return Ok(new { message = "Вхід успішний", userId = user.Id });
    }

    // PUT: api/Users/5
    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateUser(int id, [FromBody] UserUpdateDto updateDto)
    {
        var user = await _userService.GetUserByIdAsync(id);
        if (user == null)
            return NotFound();

        // Оновлюємо тільки дозволені поля
        user.Username = updateDto.Username ?? user.Username;

        // В реальному додатку тут би була перевірка на права доступу

        var success = await _userService.UpdateUserAsync(user);
        if (!success)
            return BadRequest("Не вдалося оновити користувача");

        return NoContent();
    }

    // DELETE: api/Users/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteUser(int id)
    {
        var success = await _userService.DeleteUserAsync(id);
        if (!success)
            return NotFound();

        return NoContent();
    }

    // POST: api/Users/5/JoinFamily/10

    [HttpPost("{userId}/JoinFamily/{familyId}")]
    public async Task<IActionResult> JoinFamily(int userId, int familyId)
    {
        var success = await _userService.JoinFamilyAsync(userId, familyId);
        if (!success)
            return BadRequest("Не вдалося приєднатися до сім'ї");

        return NoContent();
    }
```

```
        // POST: api/Users/5/LeaveFamily
        [HttpPost("{userId}/LeaveFamily")]
        public async Task<IActionResult> LeaveFamily(int userId)
        {
            var success = await _userService.LeaveFamilyAsync(userId);
            if (!success)
                return BadRequest("Не вдалося вийти з сім'ї");

            return NoContent();
        }
    }

    // DTO класи для передачі даних
    public class UserRegisterDto
    {
        public required string Username { get; set; }
        public required string Email { get; set; }
        public required string Password { get; set; }
    }

    public class UserLoginDto
    {
        public required string Email { get; set; }
        public required string Password { get; set; }
    }

    public class UserUpdateDto
    {
        public required string Username { get; set; }
    }
}
using FunFinance.API.Models;
using FunFinance.API.Services;
using Microsoft.AspNetCore.Mvc;

namespace FunFinance.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BudgetsController : ControllerBase
    {
        private readonly IBudgetService _budgetService;

        public BudgetsController(IBudgetService budgetService)
        {
            _budgetService = budgetService;
        }

        // GET: api/Budgets
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Budget>>> GetBudgets()
        {
            var budgets = await _budgetService.GetAllBudgetsAsync();
            return Ok(budgets);
        }

        // GET: api/Budgets/5
        [HttpGet("{id}")]
        public async Task<ActionResult<Budget>> GetBudget(int id)
        {
            var budget = await _budgetService.GetBudgetByIdAsync(id);

            if (budget == null)
                return NotFound();

            return Ok(budget);
        }

        // GET: api/Budgets/5/WithExpenses
        [HttpGet("{id}/WithExpenses")]
```

```
        public async Task<ActionResult<Budget>> GetBudgetWithExpenses(int id)
        {
            var budget = await _budgetService.GetBudgetWithExpensesAsync(id);

            if (budget == null)
                return NotFound();

            return Ok(budget);
        }

        // GET: api/Budgets/ByFamily/5
        [HttpGet("ByFamily/{familyId}")]
        public async Task<ActionResult<IEnumerable<Budget>>> GetBudgetsByFamily(int familyId)
        {
            var budgets = await _budgetService.GetBudgetsByFamilyIdAsync(familyId);
            return Ok(budgets);
        }

        // GET: api/Budgets/Active/5
        [HttpGet("Active/{familyId}")]
        public async Task<ActionResult<IEnumerable<Budget>>> GetActiveBudgets(int familyId)
        {
            var budgets = await _budgetService.GetActiveBudgetsAsync(familyId);
            return Ok(budgets);
        }

        // POST: api/Budgets
        [HttpPost]
        public async Task<ActionResult<Budget>> CreateBudget([FromBody] BudgetCreateDto createDto)
        {
            try
            {
                var budget = await _budgetService.CreateBudgetAsync(
                    createDto.Name,
                    createDto.Limit,
                    createDto.StartDate,
                    createDto.EndDate,
                    createDto.FamilyId);

                return CreatedAtAction(nameof(GetBudget), new { id = budget.Id }, budget);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(ex.Message);
            }
        }

        // PUT: api/Budgets/5
        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateBudget(int id, [FromBody] BudgetUpdateDto updateDto)
        {
            var budget = await _budgetService.GetBudgetByIdAsync(id);
            if (budget == null)
                return NotFound();

            budget.Name = updateDto.Name ?? budget.Name;

            if (updateDto.Limit.HasValue)
                budget.Limit = updateDto.Limit.Value;

            if (updateDto.StartDate.HasValue)
                budget.StartDate = updateDto.StartDate.Value;

            if (updateDto.EndDate.HasValue)
```

```
            budget.EndDate = updateDto.EndDate.Value;

            var success = await _budgetService.UpdateBudgetAsync(budget);
            if (!success)
                return BadRequest("Не вдалося оновити бюджет");

            return NoContent();
        }

        // DELETE: api/Budgets/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteBudget(int id)
        {
            var success = await _budgetService.DeleteBudgetAsync(id);
            if (!success)
                return NotFound();

            return NoContent();
        }

        // GET: api/Budgets/5/Usage
        [HttpGet("{id}/Usage")]
        public async Task<ActionResult<decimal>> GetBudgetUsage(int id)
        {
            try
            {
                var usage = await _budgetService.GetBudgetUsageAsync(id);
                return Ok(usage);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(ex.Message);
            }
        }

        // GET: api/Budgets/5/Remaining
        [HttpGet("{id}/Remaining")]
        public async Task<ActionResult<decimal>> GetRemainingBudget(int id)
        {
            try
            {
                var remaining = await _budgetService.GetRemainingBudgetAsync(id);
                return Ok(remaining);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(ex.Message);
            }
        }

        // GET: api/Budgets/5/IsExceeded
        [HttpGet("{id}/IsExceeded")]
        public async Task<ActionResult<bool>> IsBudgetExceeded(int id)
        {
            try
            {
                var isExceeded = await _budgetService.IsBudgetExceededAsync(id);
                return Ok(isExceeded);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(ex.Message);
            }
        }

        // GET: api/Budgets/5/IsNearLimit
        [HttpGet("{id}/IsNearLimit")]
        public async Task<ActionResult<bool>> IsBudgetNearLimit(int id, [FromQuery] decimal threshold = 80)
        {
            try
```

```
            {
                var isNearLimit = await _budgetService.IsBudgetNearLimitAsync(id, threshold);
                return Ok(isNearLimit);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(ex.Message);
            }
        }
    }

    // DTO класи для передачі даних
    public class BudgetCreateDto
    {
        public required string Name { get; set; }
        public decimal Limit { get; set; }
        public DateTime StartDate { get; set; }
        public DateTime EndDate { get; set; }
        public int FamilyId { get; set; }
    }

    public class BudgetUpdateDto
    {
        public required string Name { get; set; }
        public decimal? Limit { get; set; }
        public DateTime? StartDate { get; set; }
        public DateTime? EndDate { get; set; }
    }
}
using FunFinance.API.Models;
using FunFinance.API.Services;
using Microsoft.AspNetCore.Mvc;

namespace FunFinance.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ExpensesController : ControllerBase
    {
        private readonly IExpenseService _expenseService;

        public ExpensesController(IExpenseService expenseService)
        {
            _expenseService = expenseService;
        }

        // GET: api/Expenses
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Expense>>> GetExpenses()
        {
            var expenses = await _expenseService.GetAllExpensesAsync();
            return Ok(expenses);
        }

        // GET: api/Expenses/5
        [HttpGet("{id}")]
        public async Task<ActionResult<Expense>> GetExpense(int id)
        {
            var expense = await _expenseService.GetExpenseByIdAsync(id);

            if (expense == null)
                return NotFound();

            return Ok(expense);
        }

        // GET: api/Expenses/ByUser/5
        [HttpGet("ByUser/{userId}")]
        public async Task<ActionResult<IEnumerable<Expense>>> GetExpensesByUser(int userId)
        {
            var expenses = await _expenseService.GetExpensesByUserIdAsync(userId);
```

```
            return Ok(expenses);
        }

        // GET: api/Expenses/ByBudget/5
        [HttpGet("ByBudget/{budgetId}")]
        public async Task<ActionResult<IEnumerable<Expense>>> GetExpensesByBudget(int budgetId)
        {
            var expenses = await _expenseService.GetExpensesByBudgetIdAsync(budgetId);
            return Ok(expenses);
        }

        // GET: api/Expenses/ByCategory/5
        [HttpGet("ByCategory/{categoryId}")]
        public async Task<ActionResult<IEnumerable<Expense>>> GetExpensesByCategory(int categoryId)
        {
            var expenses = await _expenseService.GetExpensesByCategoryIdAsync(categoryId);
            return Ok(expenses);
        }

        // GET: api/Expenses/ByFamily/5
        [HttpGet("ByFamily/{familyId}")]
        public async Task<ActionResult<IEnumerable<Expense>>> GetExpensesByFamily(int familyId)
        {
            var expenses = await _expenseService.GetExpensesByFamilyIdAsync(familyId);
            return Ok(expenses);
        }

        // GET: api/Expenses/ByDateRange
        [HttpGet("ByDateRange")]
        public async Task<ActionResult<IEnumerable<Expense>>> GetExpensesByDateRange(
            [FromQuery] int familyId,
            [FromQuery] DateTime startDate,
            [FromQuery] DateTime endDate)
        {
            var expenses = await _expenseService.GetExpensesByDateRangeAsync(familyId, startDate, endDate);
            return Ok(expenses);
        }

        // POST: api/Expenses
        [HttpPost]
        public async Task<ActionResult<Expense>> CreateExpense([FromBody] ExpenseCreateDto createDto)
        {
            try
            {
                var expense = await _expenseService.CreateExpenseAsync(
                    createDto.Description,
                    createDto.Amount,
                    createDto.Date,
                    createDto.CategoryId,
                    createDto.UserId,
                    createDto.BudgetId);

                return CreatedAtAction(nameof(GetExpense), new { id = expense.Id }, expense);
            }
            catch (ArgumentException ex)
            {
                return BadRequest(ex.Message);
            }
        }

        // PUT: api/Expenses/5
```

```
        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateExpense(int id, [FromBody] ExpenseUpdateDto updateDto)
        {
            var expense = await _expenseService.GetExpenseByIdAsync(id);
            if (expense == null)
                return NotFound();

            if (!string.IsNullOrEmpty(updateDto.Description))
                expense.Description = updateDto.Description;

            if (updateDto.Amount.HasValue)
                expense.Amount = updateDto.Amount.Value;

            if (updateDto.Date.HasValue)
                expense.Date = updateDto.Date.Value;

            if (updateDto.CategoryId.HasValue)
                expense.CategoryId = updateDto.CategoryId.Value;

            var success = await _expenseService.UpdateExpenseAsync(expense);
            if (!success)
                return BadRequest("Не вдалося оновити витрату");

            return NoContent();
        }

        // DELETE: api/Expenses/5
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteExpense(int id)
        {
            var success = await _expenseService.DeleteExpenseAsync(id);
            if (!success)
                return NotFound();

            return NoContent();
        }

        // GET: api/Expenses/TotalByUser
        [HttpGet("TotalByUser")]
        public async Task<ActionResult<decimal>> GetTotalExpensesByUser(
            [FromQuery] int userId,
            [FromQuery] DateTime startDate,
            [FromQuery] DateTime endDate)
        {
            var total = await _expenseService.GetTotalExpensesByUserAsync(userId, startDate, endDate);
            return Ok(total);
        }

        // GET: api/Expenses/TotalByCategory
        [HttpGet("TotalByCategory")]
        public async Task<ActionResult<decimal>> GetTotalExpensesByCategory(
            [FromQuery] int categoryId,
            [FromQuery] DateTime startDate,
            [FromQuery] DateTime endDate)
        {
            var total = await _expenseService.GetTotalExpensesByCategoryAsync(categoryId, startDate, endDate);
            return Ok(total);
        }

        // GET: api/Expenses/GroupByCategory
        [HttpGet("GroupByCategory")]
```

```
        public async Task<ActionResult<IDictionary<int, decimal>>> GetExpensesByCategory(
            [FromQuery] int familyId,
            [FromQuery] DateTime startDate,
            [FromQuery] DateTime endDate)
        {
            var groupedExpenses = await _expenseService.GetExpensesByCategory(familyId, startDate, endDate);
            return Ok(groupedExpenses);
        }

        // GET: api/Expenses/GroupByUser
        [HttpGet("GroupByUser")]
        public async Task<ActionResult<IDictionary<int, decimal>>> GetExpensesByUser(
            [FromQuery] int familyId,
            [FromQuery] DateTime startDate,
            [FromQuery] DateTime endDate)
        {
            var groupedExpenses = await _expenseService.GetExpensesByUser(familyId, startDate, endDate);
            return Ok(groupedExpenses);
        }

        // GET: api/Expenses/GroupByDay
        [HttpGet("GroupByDay")]
        public async Task<ActionResult<IDictionary<DateTime, decimal>>> GetExpensesByDay(
            [FromQuery] int familyId,
            [FromQuery] DateTime startDate,
            [FromQuery] DateTime endDate)
        {
            var groupedExpenses = await _expenseService.GetExpensesByDay(familyId, startDate, endDate);
            return Ok(groupedExpenses);
        }
    }

    // DTO класи для передачі даних
    public class ExpenseCreateDto
    {
        public required string Description { get; set; }
        public decimal Amount { get; set; }
        public DateTime Date { get; set; }
        public int CategoryId { get; set; }
        public int UserId { get; set; }
        public int BudgetId { get; set; }
    }

    public class ExpenseUpdateDto
    {
        public required string Description { get; set; }
        public decimal? Amount { get; set; }
        public DateTime? Date { get; set; }
        public int? CategoryId { get; set; }
    }
}
```