МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ СИСТЕМ ТА МЕРЕЖ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри

_____ Ігор Жуков
(підпис) (ПІБ)

« ___ » _____ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "МАГІСТР"
ЗА СПЕЦІАЛЬНІСТЮ 123 "КОМП'ЮТЕРНА ІНЖЕНЕРІЯ"

Тема: "Система для аналізу та прогнозування фінансових потоків"

Виконавець: студент групи КС–231М Кухар Єгор Ігорович

Керівник: кандидат технічних наук, доцент Кудренко Станіслава Олексіївна

Нормоконтролер: _____ Василь МАЛЯРЧУК

Засвідчую, що у магістерській роботі немає
запозичень праць інших авторів
без відповідних посилань

Студент _____ Кухар Є. І.

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук та технологій

Кафедра комп'ютерних систем та мереж

Спеціальність 123 "Комп'ютерна інженерія"

ЗАТВЕРДЖУЮ
Завідувач кафедри КСМ

_____ Ігор Жуков
(підпис) (ПІБ)

« ___» _____ 2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Кухара Єгора Ігоровича
(прізвище, ім'я, по батькові)

1. Тема роботи: "Система для аналізу та прогнозування фінансових потоків" затверджена наказом ректора від «10» травня 2023 р. №193/од

2. Термін виконання проєкту (роботи): з 02.10.2023 до 31.12.2023

3. Вихідні дані до роботи: розробити комп'ютерну систему для аналізу та прогнозування фінансових потоків

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):

1) Методи аналізу та прогнозування фінансових потоків

2) Технології розробки систем

3) Розробка системи

4) Тестування системи та приклад використання

5. Перелік обов’язкового графічного матеріалу:

Презентація *Power Point*

6. Календарний план–графік

№ п/п	Етапи виконання кваліфікаційної роботи	Термін виконання етапів	Примітка
1	Ознайомитись з постановкою задачі кваліфікаційної роботи	02.10.2023 – 06.10.2023	
2	Вивчити спеціальну літературу і технічну документацію	07.10.2023 – 09.10.2023	
3	Проаналізувати методи аналізу та прогнозування фінансових потоків	10.10.2023 – 19.10.2023	
4	Написати розділ 1	20.10.2023 – 25.10.2023	
5	Проаналізувати та порівняти технології для розробки систем	26.10.2023 – 30.10.2023	
6	Написати розділ 2	31.10.2023 – 10.11.2023	
7	Розробити систему	11.11.2023 – 20.11.2023	
8	Написати розділ 3	21.11.2023 – 30.11.2023	
9	Протестувати систему та описати приклад використання	01.12.2023– 06.12.2023	
10	Написати розділ 4	07.12.2023– 09.12.2023	
11	Оформити пояснювальну записку та	10.12.2023 –	

	пройти нормоконтроль	22.12.2023	
12	Підготувати презентаційний матеріал та захистити роботу	23.12.2023 – 31.12.2023	

7. Дата видачі завдання «02» жовтня 2023 р.

Керівник кваліфікаційної роботи Кудренко С. О.

(підпис)

Завдання прийняв до виконання Кухар Є. І.

(підпис студента)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи “Система для аналізу та прогнозування фінансових потоків” складається з 94 сторінок, включає 36 рисунків та базується на 20 використаних джерелах.

ФІНАНСОВІ ПОТОКИ, АНАЛІЗ, ПРОГНОЗУВАННЯ, КОМП'ЮТЕРНА СИСТЕМА

Об’єкт дослідження – процес аналізу та прогнозування фінансових потоків.

Предмет дослідження – розробка комп'ютерної системи для аналізу та прогнозування фінансових потоків.

Мета кваліфікаційної роботи – створення ефективної системи, яка надасть можливість користувачам аналізувати та прогнозувати фінансові потоки з метою прийняття обґрунтованих фінансових рішень.

Прогнози припущення щодо розвитку об’єкта – створення функціонального прототипу комп'ютерної системи для аналізу та прогнозування фінансових потоків.

Рекомендації до використання результатів: отримані результати можуть бути використані для розробки нових програмних засобів, спрямованих на управління та аналіз фінансових потоків.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

HTML – HyperText Markup Language (мова розмітки гіпертексту)

CSS – Cascading Style Sheets (каскадні таблиці стилів)

JS – JavaScript

SQL – Structured Query Language (мова структурованих запитів)

SPA – Single Page Application (односторінковий застосунок)

DOM – Document Object Model (модель об'єктів документу)

MVC – Model–View–Controller (модель–вид–контролер)

MVU – Model–View–Update (модель– вид–оновлення)

MVVM – Model–View–ViewModel (модель–вид–модель представлення)

MVP – Model–View–Presenter (модель–вид–представник)

EF – Entity Framework

ORM – Object–Relational Mapping (об’єктно–реляційне відображення)

LINQ – Language Integrated Query (мова інтегрованих запитів)

GUI – Graphical User Interface (графічний інтерфейс користувача)

SWT – Standard Widget Toolkit (стандартний набір віджетів)

WPF – Windows Presentation Foundation (фреймворк представлення віконних застосунків)

XAML – eXtensible Application Markup Language (розширювана мова розмітки додатків)

MAUI – Multi–platform App UI (мультиплатформний користувацький інтерфейс додатків)

SSL – Secure Sockets Layer (безпечний шар з'єднань)

JSON – JavaScript Object Notation (язик об'єктів *JavaScript*)

BSON – Binary JSON (двійковий *JSON*)

MSSQL – Microsoft SQL Server

T–SQL – Transact–SQL

DI – Dependency Injection (ін’єкція залежностей)

CLI – Command Line Interface (командний рядок)

IIS – Internet Information Services (інформаційні сервіси Інтернет)

CORS – Cross–Origin Resource Sharing (спільне використання ресурсів між джерелами)

XML – eXtensible Markup Language (мовний розширюваний маркувальний засіб)

JSX – JavaScript XML

СУБД – система управління базами даних

ВСТУП

З високою складністю сучасного економічного середовища особисті фінанси виростають в область, яка вимагає не тільки уваги, але й систематичного підходу до аналізу та прогнозування. В сучасному світі, де швидкі зміни, глобалізація та технологічний прогрес стають неот'ємною частиною нашого існування, збереження фінансової стабільності стає викликом, який потребує вдумливості та аналітичного підходу.

Особисті фінанси, як ніколи раніше, потребують ретельного управління та стратегічного планування. В даному контексті, аналіз фінансів стає ключовим елементом визначення фінансового здоров'я та прийняття обґрунтованих рішень. Проте, нарівні з традиційними методами, сучасні технології вносять суттєвий внесок у способи, якими ми можемо здійснювати аналіз та прогнозування своїх фінансів.

Особисті фінанси визначають не тільки нашу фінансову стабільність, але й забезпечують можливість реалізації життєвих цілей. Аналіз фінансового стану дозволяє визначити позитивні та негативні аспекти наших фінансів, виявити тенденції та уникнути можливих фінансових ризиків.

Аналіз особистих фінансів – це детальне вивчення фінансового стану і поведінки особистих фінансів. Він дозволяє виявити різні аспекти, такі як рівень доходу, витрати, заборгованості та активи. Цей процес може бути здійснений через вивчення банківських виписок, опитування витрат та розгляд бюджету.

Аналіз дозволяє визначити поточний фінансовий стан, виявити головні джерела доходів і основні витрати. Це стає фундаментом для подальшого фінансового планування.

Аналіз фінансів допомагає виявити області, де можна здійснити заощадження та оптимізацію витрат. Це може включати у себе уникнення непотрібних витрат та розробку стратегій розумного витрачання.

Аналіз допомагає визначити фінансові цілі та прийняти рішення, спрямовані на досягнення цих цілей. Це може бути стосовно освіти, житла, пенсійного забезпечення та інших аспектів життя.

Прогнозування фінансів – це визначення очікуваних фінансових результатів на майбутнє на основі історичних даних та теперішнього фінансового стану. Цей етап дозволяє створити план дій для досягнення фінансових цілей та уникнення можливих ризиків.

Прогнозування дозволяє розробити бюджет, який враховує майбутні доходи та витрати. Це створює рамки для ефективного управління фінансами.

Прогнозування включає в себе вибір стратегій інвестування, що допомагають збільшити фінансовий потенціал та забезпечити довгостроковий рост капіталу.

Прогнозування дозволяє створити фінансовий резерв для подолання непередбачених обставин, таких як втрата роботи, несподівані витрати чи економічна нестабільність.

Засоби аналізу та прогнозування особистих фінансів відіграють ключову роль у забезпеченні ефективного управління грошима, створенні стратегій бюджету та досягненні фінансових цілей.

Один із основних засобів аналізу – це періодичний огляд банківських виписок та фінансових звітів. Це дозволяє визначити джерела доходів, витрати та загальний фінансовий стан.

Створення бюджету є ефективним засобом аналізу фінансів. Визначення розподілу доходів на різні категорії допомагає контролювати витрати та планувати раціональне використання грошових ресурсів.

Детальний аналіз поточних витрат дозволяє виявити зайві витрати та здійснити корекції в бюджеті. Такий підхід сприяє збереженню фінансових ресурсів.

Прогнозування фінансів починається з визначення фінансових цілей. Це може бути економія для майбутніх витрат, покупка житла чи інвестування в освіту.

Стратегічне прогнозування включає в себе розробку плану інвестицій та стратегій фінансового росту. Інвестування може допомогти збільшити капітал та забезпечити фінансову стабільність у майбутньому.

Роль технологій у аналізі та прогнозуванні особистих фінансів стає визначальною, дозволяючи нам здійснювати більш ефективний та автоматизований контроль над нашими грошима.

Сучасні мобільні додатки та онлайн–платформи стали невід'ємною частиною фінансового керівництва. Мобільні додатки надають зручний інтерфейс для ведення бюджету, відстеження витрат та вивчення фінансової статистики. Онлайн–платформи дозволяють управляти банківськими рахунками та інвестиційними портфелями в реальному часі, що дозволяє швидше реагувати на фінансові події.

Застосування штучного інтелекту (ШІ) у фінансовій сфері відкриває нові перспективи для аналізу та прогнозування особистих фінансів. Системи на основі ШІ можуть аналізувати великі обсяги даних, враховувати попередні витрати та створювати персоналізовані рекомендації щодо оптимізації бюджету та інвестицій.

Технології дозволяють автоматизувати процес бюджетування та планування. Автоматичне відстеження витрат, автоматизовані платіжні системи та інші інноваційні рішення роблять ведення бюджету більш точним та зручним.

У контексті фінансів, де важливі особисті дані перебувають під загрозою, технології також відіграють ключову роль у забезпеченні кібербезпеки. Використання шифрування, двофакторної аутентифікації та інших методів допомагає захистити особисті фінансові дані користувачів.

Електронні платіжні системи та криптовалюти революціонізують спосіб, яким ми здійснюємо фінансові операції. Швидкі та безпечні електронні перекази, а також можливість інвестування у криптовалюти розширюють наші можливості у сфері фінансів.

Важливість систем аналізу та прогнозування особистих фінансів полягає в їхньому внеску у створення фінансової стабільності та забезпеченні раціонального використання ресурсів. Ці системи є невід’ємною частиною сучасного фінансового планування, надаючи інструменти для ефективного управління фінансами та досягнення поставлених цілей. Прикладами таких систем можуть стати:

– *Mint*. Це інноваційний фінансовий менеджер, створений для ефективного керування особистими фінансами. З його допомогою користувачі отримують

можливість бюджетувати та аналізувати витрати, відстежувати доходи та витрати. Mint дозволяє створювати детальні бюджети та отримувати аналіз витрат для збалансованого управління фінансами.

– *YNAB*. Відомий фінансовий інструмент, орієнтований на реальний час та філософію чотирьох категорій. Основні можливості включають визначення грошових потоків на обов'язкові витрати, заощадження, розваги та несподівані витрати.

– *PocketGuard*. Це додаток, який автоматизує процеси бюджетування та аналізу витрат для ефективного фінансового керування. *PocketGuard* розробляє автоматичний бюджет, щоб допомогти користувачам керувати своїми фінансами. Витрати автоматично категоризуються для кращого розуміння, де йдуть гроші. Користувачі можуть відслідковувати та оплачувати рахунки безпосередньо з додатку. Додатковий функціонал *PocketGuard* включає нагадування про майбутні платежі, рекомендації щодо збереження та інвестування грошей.

– *Personal Capital*. Ця система є комплексним фінансовим інструментом, спрямованим на інвестиції та пенсійне планування. Дозволяє користувачам відстежувати та аналізувати свої інвестиції та пенсійні вкладення. Податкове Планування: Забезпечує інструменти для податкового планування та оптимізації. Додатковий функціонал включає можливість встановлення фінансових цілей та завдань для досягнення конкретних цілей. *Personal Capital* інтегрується з банківськими системами для автоматичної синхронізації, а його доступність на різних платформах робить його зручним і доступним для використання в різних ситуаціях.

– *Wallet by BudgetBakers*. *Wallet* надає можливість створювати деталізовані бюджети та аналізувати витрати для ефективного фінансового планування. Автоматична синхронізація з банківськими рахунками для швидкого відстеження доходів та витрат. Підтримка різних валют для зручного використання у всьому світі. Wallet пропонує стратегії для заощадження грошей та досягнення фінансових цілей. Детальна аналітика та звіти про фінансовий стан для кращого розуміння. *Wallet by BudgetBakers* надає зручні інструменти для фінансового управління та

детального аналізу витрат, роблячи його популярним серед користувачів, що цінують ефективність та докладність в управлінні своїми фінансами.

Мета даної кваліфікаційної роботи полягає в дослідженні, розробці та впровадженні комп'ютерної системи, яка спрямована на оптимізацію фінансового аналізу та прогнозування.

Загальна мета дослідження:

Дослідити сучасний стан фінансового управління та розробити інноваційну систему для ефективного аналізу та прогнозування фінансових потоків.

Значущість дослідження:

Результати дослідження та розробки системи будуть внеском у галузі інформаційних технологій та фінансового менеджменту. Оптимізація процесів фінансового аналізу сприятиме підвищенню ефективності управління фінансами, забезпечуючи підприємствам та організаціям зручні та точні інструменти для прийняття стратегічних рішень.

Основні задачі дослідження:

– Детальний аналіз сучасних проблем фінансового управління.

– Розробка концепції та функціональних вимог до комп'ютерної системи.

– Впровадження технологічних рішень для оптимізації фінансового аналізу.

– Тестування та апробація розробленої системи на практиці.

РОЗДІЛ 1
МЕТОДИ АНАЛІЗУ ТА ПРОГНОЗУВАННЯ ФІНАНСОВИХ ПОТОКІВ

1.1. Аналіз фінансових звітів

Існують два типи особистих фінансових звітів:

– Особистий звіт про грошовий потік.

– Особистий балансовий звіт.

– Особистий звіт про грошовий потік.

– Особистий звіт про грошовий потік вимірює грошові надходження та витрати, щоб показати чистий грошовий потік протягом певного періоду часу. Грошові надходження включають наступне:

– Зарплата.

– Відсотки зі збережень.

– Дивіденди від інвестицій.

– Капітальний прибуток від продажу фінансових цінних паперів, таких як акції та облігації.

Грошові надходження також можуть включати гроші, отримані від продажу активів, таких як будинки чи автомобілі. По суті, грошові надходження складаються з усього, що приносить гроші.

Грошові витрати охоплюють усі витрати, незалежно від розміру. До грошових витрат входять такі типи витрат:

– Оренда або виплати за іпотеку.

– Платежі за комунальні послуги.

– Продукти харчування.

– Паливо.

– Розваги (книги, квитки в кіно, прийоми у ресторанах тощо).

Метою визначення грошових надходжень і витрат є визначення чистого грошового потоку. Чистий грошовий потік є результатом віднімання витрат від надходжень. Позитивний чистий грошовий потік означає, що людина заробила більше, ніж витратила, і у неї є залишок коштів з цього періоду. З іншого боку, негативний чистий грошовий потік показує, що людина витратила більше грошей, ніж заробила.

Балансовий звіт є другим типом особистого фінансового звіту. Особистий балансовий звіт надає загальний знімок багатства на певний період часу. Це підсумок активів (те, чим людина володіє), зобов'язань (те, що заборгувала) та чистого багатства (активи мінус зобов'язання).

Активи можна класифікувати на три основні категорії:

– Ліквідні активи: Ліквідні активи – це речі, якими володієте і які можна легко продати або перетворити на готівку, не втрачаючи вартості. Сюди входять розрахункові рахунки, рахунки грошового ринку, заощадження та готівка.

– Великі активи: Великі активи включають такі речі, як будинки, автомобілі, човни, мистецтво та меблі. Створюючи особистий балансовий звіт, потрібно переконатися, що враховується ринкова вартість цих предметів. Якщо складно знайти ринкову вартість, необхідно використовувати останні ціни продажу подібних предметів.

– Інвестиції: Інвестиції включають облігації, акції, інвестиційні фонди та нерухомість. Потрібно записувати інвестиції за їх поточними ринковими значеннями.

Зобов'язання – це те, що людина заборгувала. До зобов'язань входять поточні рахунки, платежі, які все ще винні за деякі активи, такі як автомобілі та будинки, баланси по кредитних картках та інші позики.

Чисте багатство – це різниця між тим, чим володієте і що заборгували. Ця цифра є показником багатства, оскільки вона відображає те, що залишається після того, як все зобов'язання було погашено. Якщо чисте багатство є негативним, це означає, що людина заборгувала більше, ніж володіє.

Є два способи збільшити чисте багатство: збільшити активи або зменшити зобов'язання. Збільшити активи можна за допомогою збільшення своїх готівкових кошти або підвищуючи вартість будь–якого активу, яким володієте. Збільшення чистого багатства є результатом ефективного управління активами та зобов'язаннями.

Обидва типи особистих фінансових звітів – особистий звіт про грошовий потік та особистий балансовий звіт – є корисними інструментами для керування фінансами. Вони надають вам ясність щодо вашого фінансового стану, допомагають планувати бюджет та приймати усвідомлені фінансові рішення.

Особисті фінансові звіти надають інструменти для контролю витрат і збільшення чистої вартості. Річ у тому, що особисті фінансові звіти не є просто двома окремими частинами інформації, але насправді вони працюють разом. Звіт про грошовий потік фактично може допомогти у досягненні збільшення чистої вартості. Якщо у людини є позитивний грошовий потік в певний період, то тоді можна використати ці гроші для придбання активів або погашення зобов'язань. Використання грошового потоку для збільшення чистої вартості є відмінним способом збільшення активів без збільшення зобов'язань або зменшення зобов'язань без збільшення активів.

Якщо у людини наразі грошовий потік є негативним або людина хоче збільшити позитивний грошовий потік, тоді єдиний спосіб цього досягти – це оцінити звички витрат і внести необхідні зміни. Використання особистих фінансових звітів для більш свідомого сприйняття звичок витрат і чистої вартості допоможе на шляху до більшої фінансової безпеки.

1.2. Аналіз витрат

Для того щоб покращити фінансову стійкість, потрібно проаналізувати поточну фінансову ситуацію. Слід уважно відстежувати всі витрати протягом місяця і подивитися, куди витрачається найбільше коштів. Хоча збереження може здатися

складним завданням, насправді це досить просто, якщо дізнатись, куди саме йдуть гроші.

Основні кроки для аналізу витрат це:

1) Записати всі покупки, які були протягом місяця. Необхідно вказати суму, яку витрачено, день і час.

2) Підрахувати фіксовані витрати. Фіксовані витрати не змінюються щомісяця. Зазвичай у людей фіксовані витрати включають наступне:

– Оренда або іпотека.

– Страхування.

– Платежі за автомобіль.

– Комунальні послуги.

– Погашення заборгованості.

3) Ретельніше розглянути не фіксовані витрати на розваги. Вони змінюються кожен місяць. Необхідно звернути увагу на те, на що витрачаються гроші. Потрібно розбити витрачені суми на категорії, наприклад:

– Продукти харчування.

– Відвідування ресторанів.

– Паливо.

– Одяг.

– Хобі/розваги.

4) Розглянути, коли відбуваються найбільші витрати. Потрібно подивитись на дні та часи, коли робиться більшість витрат. Чи відбуваються покупки імпульсивно відразу після роботи? Чи витрачається забагато грошей у вихідні?

Ці кроки допоможуть розібратися в витратах, та зрозуміти які витрати можна зменшити для покращення чистої вартості та чистого грошового потіку.

1.3. Створення бюджету

Створення бюджету, або ж бюджетування не завжди приносить задоволення, але це одна з найважливіших речей, які можна зробити для поліпшення фінансового

стану. Створення та використання бюджету – це щось, від чого можуть отримати користь всі. Бюджетування – це потужний процес, який допомагає розробити фінансовий план і розвивати фінансові навички та самодостатність.

Бюджетування дає контроль над тим, як спрямовувати гроші на те, що хочеться, але в розумних межах.

Бюджет – це план того, як будуть витрачатися та заощаджуватися доходи щомісяця. Бюджетування включає:

– Визначення пріорітетів та цілей.

– Створення бюджетного плану, який визначає приблизні щомісячні доходи та витрати.

– Відстеження вашого фактичного споживання та доходів.

– Внесення коригувань до плану.

Бюджетування допомагає:

– Контролювати гроші та забезпечити їх використання для задоволення потреб і досягнення поставлених цілей.

– Показати, куди йдуть гроші та зменшити марнотратні витрати.

– Покращити здатність сплатити всі рахунки та не вичерпати кошти протягом місяця.

– Звільнити гроші для погашення боргів.

– Зберігати гроші на необхідне та бажане.

– Зменшити стрес.

– Краще підготуватися до непередбачуваних ситуацій.

Приклад бюджету який фокусується на розподілення 50 відсотків грошей на необхідних речах, 30 відсотків на бажаних і 20 відсотків на збереженнях зображений на рис. 1.1.

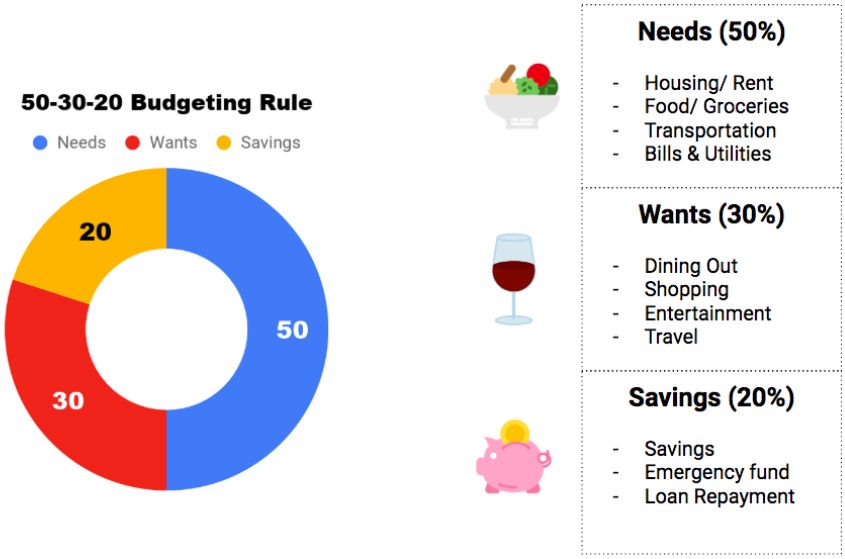

Рис.1.1. Приклад простого бюджет 50–30–20

1.4. Відносний аналіз

Відносний аналіз в персональних фінансах використовується для порівняння фінансових показників у різних періодах часу або відносно інших факторів. Цей аналіз дозволяє отримати більшу контекстуалізацію фінансової ситуації та зрозуміти зміни, що відбуваються в часі.

Основні методи відносного аналізу включають:

1. Темп зростання/спаду: Цей показник визначає, як швидко змінюється фінансовий показник відносно попереднього періоду. Це може бути виражено у відсотках або абсолютних значеннях. Наприклад, розраховуючи темп зростання доходів, порівняти поточний дохід з попереднім періодом, щоб визначити, наскільки значно збільшився дохід.

2. Відносна величина: Цей показник порівнює фінансовий показник з іншими факторами або заздалегідь встановленими метриками. Наприклад, можна порівняти відносну величину витрат на різні категорії, такі як їжа, житло, розваги, з метою виявлення основних факторів, які впливають на витрати.

3. Відношення та показники: Використовуючи співвідношення між різними фінансовими показниками, можна отримати більшу інформацію про фінансову ситуацію. Наприклад, відношення між прибутком та витратами (співвідношення доходів і витрат) може вказувати на фінансову стабільність або дисбаланс.

4. Графіки та візуалізація: Візуалізація фінансових даних у вигляді графіків або діаграм допомагає легше сприймати зміни в часі. Графіки можуть показувати тренди, сезонні зміни або порівнювати різні фінансові показники.

Відносний аналіз допомагає виявити тенденції, зміни та залежності у власних персональних фінансах. Він надає більше контексту та допомагає приймати обґрунтовані рішення щодо бюджетування, збереження, інвестування та планування майбутніх фінансових цілей.

1.5. Опис системи аналізу та прогнозування фінансових потоків

Система для аналізу та прогнозування фінансових потоків, заснована на бюджетуванні, аналізі витрат, аналізі грошових потоків та чистої вартості, є потужним інструментом для управління особистими фінансами. Основною метою цієї системи є надання користувачеві інформації та інструментів, необхідних для зрозуміння, контролю та оптимізації його фінансового стану.

Одним із ключових елементів системи є процес бюджетування. Вона дозволяє користувачеві створювати детальний план своїх доходів та витрат на певний період часу. Цей бюджет допомагає користувачеві усвідомити, які грошові потоки відбуваються в його особистих фінансах та як ці потоки впливають на його загальну фінансову ситуацію. Приклад діаграми для побудови необхідної інфраструктури в системі зображено на рис. 1.2.

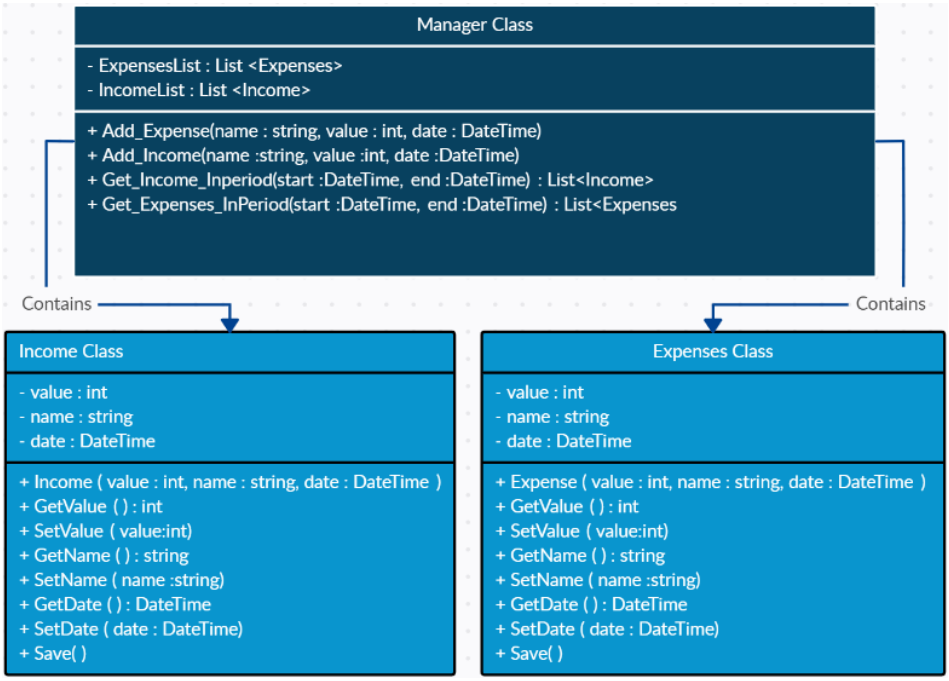

Рис.1.2. Класова діаграма для слідкування за бюджетом

Система також повинна мати можливість аналізувати витрати. Користувач може внести в систему інформацію про свої витрати, класифікувати їх за категоріями та періодами. Цей аналіз допомагає користувачеві бачити, куди йдуть його гроші і як ефективно керувати витратами.

Чиста вартість є ще одним важливим аспектом системи. Вона визначається як різниця між активами та зобов'язаннями користувача. Система відстежує зміни в чистій вартості з часом і надає користувачеві можливість оцінити його фінансовий прогрес та зміни в його майновому стані.

Також доволі корисним з точки зору мотивації користувача є відносний аналіз, адже якщо людина бачить, що кожного місяця в неї виходить збільшувати чисту вартість, то це мотивує її продовжувати.

Усі ці елементи системи для аналізу та прогнозування фінансових потоків сприяють збалансованому та ефективному управлінню особистими фінансами. Вона надає користувачу цінну інформацію та інструменти, необхідні для прийняття обґрунтованих фінансових рішень, планування майбутнього та досягнення фінансової стабільності та цілей

Висновки за розділом

У ході роботи були проаналізовані та досліджені методи для аналізу та прогнозування фінансових потоків. Був описаний аналіз фінансових звітів, який дозволяє отримати детальну інформацію про фінансовий стан людини. Також був описаний аналіз витрат, який допомагає виявити основні джерела витрат та розподілити їх на категорії для ефективного управління фінансами. Бюджетування було визначено як важливий інструмент для планування та контролю фінансів.

Досліджено також відносний аналіз, який дозволяє порівнювати фінансові показники з попередніми періодами або з іншими людьми для виявлення тенденцій та аналізу. Описано систему для аналізу та прогнозування фінансових потоків, яка має на меті забезпечити ефективне управління фінансами та досягнення поставлених фінансових цілей.

У результаті роботи була визначена необхідна функціональність системи для аналізу та прогнозування фінансових потоків, яка включає аналіз фінансових звітів, витрат, бюджетування та відносний аналіз. Ця система допоможе користувачам здійснювати ефективне управління фінансами, контролювати витрати, планувати бюджет та досягати фінансової стабільності та успіху.

Враховуючи важливість аналізу та прогнозування фінансових потоків у особистих фінансах, рекомендується використовувати систему, яка надає необхідні інструменти та функціональність для ефективного управління фінансами. Це допоможе досягти фінансової стабільності, забезпечити здоровий фінансовий стан та досягнути фінансових цілей.

РОЗДІЛ 2
ТЕХНОЛОГІЇ РОЗРОБКИ СИСТЕМ

2.1. Аналіз різних архітектурних підходів

Існують два основних архітектурних підходи – монолітна та розподілена архітектура.

Монолітна архітектура передбачає, що весь функціонал об'єднаний в одному програмному забезпеченні. Всі компоненти, такі як серверна логіка, база даних, інтерфейс користувача, розгортаються як єдиний блок. Приклад монолітної архітектури представлений на рис. 2.1.

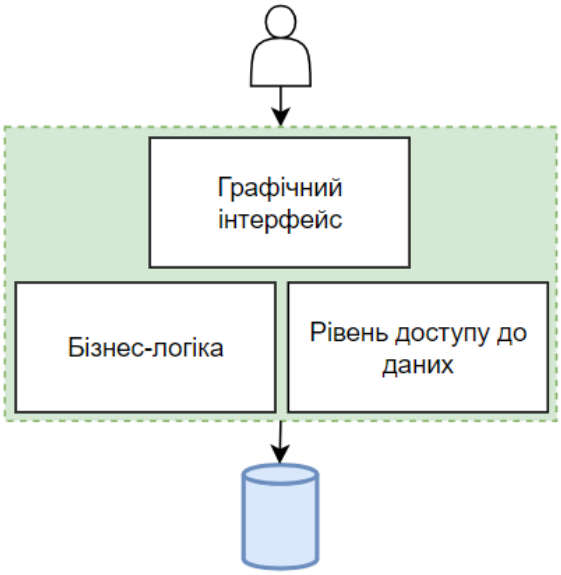

Рис.2.1. Монолітна архітектура

З переваг використання монолітної архітектури можна виділити простоту розробки та тестування, оскільки всі компоненти об'єднані, а також зменшення складності ведення конфігурації та розгортання.

З недоліків монолітної архітектури можна виділити обмежену масштабованість, оскільки росте складність з розширенням функціоналу, а також зменшену гнучкість та можливість використання різних технологій.

Розподілена архітектура передбачає розділення функціональності на окремі сервіси чи модулі, які можуть взаємодіяти між собою. Кожен сервіс може бути розгорнутий та масштабований незалежно. Приклад розподіленої архітектури представлено на рис. 2.2.

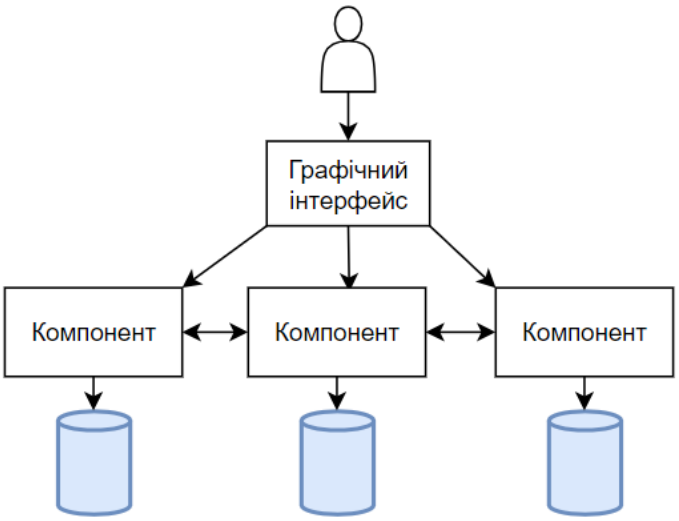

Рис.2.2. Розподілена архітектура

З переваг розподіленої архітектури можна виокремити збільшену гнучкість та можливість розвитку, оскільки сервіси можна додавати або модифікувати окремо, а також підвищену масштабованість, оскільки окремі компоненти можуть бути розгорнуті на різних серверах.

З недоліків використання розподіленої архітектури можна виокремити вищу складність управління та тестування через необхідність координації між сервісами і збільшену складність конфігурації та розгортання.

Монолітна архітектура може бути виправданою для простих систем, тоді як розподілена архітектура може забезпечити більшу гнучкість та масштабованість для складних застосунків.

Також для розробки застосунків є декілька ключових шаблонів, наприклад:

Модель–Вид–Контролер – це архітектурний шаблон, що розділяє истему на три основні компоненти – модель (бізнес–логіка та дані), вид (представлення інформації користувачеві) та контролер (управління потоком даних між моделлю та видом). Приклад цього архітектурного шаблону представлений на рис. 2.3.

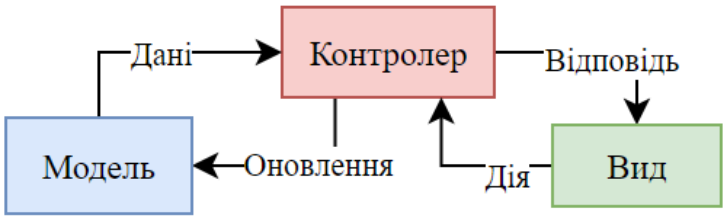

Рис.2.3. Шаблон Модель–Вид–Контролер

З переваг використання шаблону Модель–Вид–Контролер – чітке розділення обов'язків сприяє підтримці та розширенню коду, а також висока повторюваність та можливість використання компонентів у інших проектах.

Одним з недоліків шаблону Модель–Вид–Контролер є те, що може виникнути перенасиченість контролерів при великій кількості взаємодіючих компонентів.

Модель–Вид–Модель представлення – шаблон, що використовує три компоненти: модель (бізнес–логіка та дані), вид (представлення) та модель представлення (прокладка між моделлю та видом, що дозволяє відокремити логіку представлення від бізнес–логіки). Приклад шаблону представленний на рис. 2.4.

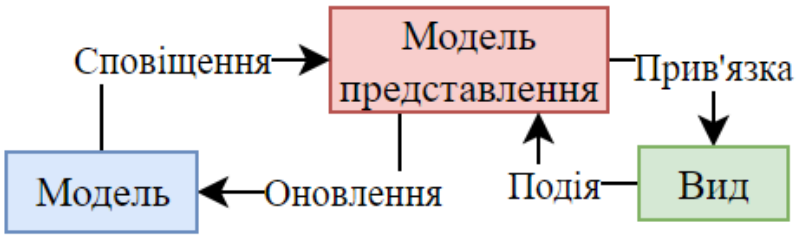

Рис.2.4. Шаблон Вид–Модель–Модель представлення

З переваг використання шаблону Вид–Модель–Модель представлення можна навести те, що даний шаблон забезпечує високий рівень абстракції та відокремлення коду і те що він підходить для розробки інтерактивних інтерфейсів.

Найбільшим недоліком шаблону *MVVM* є те, що він може стати складним у випадку великих проектів.

Модель–Вид–Оновлення – шаблон, спеціально адаптований для розробки користувацьких інтерфейсів та працює дуже ефективно в контексті односторінкових додатків. Модель представляє стан додатку, всі дані та інформація, яку бачить користувач на екрані зберігається у моделі. Вид відображає інформацію користувачеві, це представлення, яке може змінюватися відповідно до стану моделі.

Оновлення відповідає за те, які зміни потрібно внести до моделі при взаємодії користувача, відбувається відповідно до подій або введення користувача. Приклад шаблону представлено на рис. 2.5.

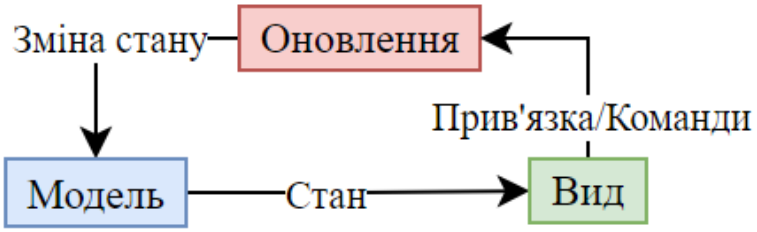

Рис.2.5. Шаблон Модель–Вид–Оновлення

З переваг використання шаблону Модель–Вид–Оновлення є простота та прозорість в логіці оновлення стану, декларативний підхід до опису інтерфейсу користувача.

Недоліком шаблону є те, що може виникнути складність при обробці більш складних станів.

Модель–Вид–Представник – шаблон архітектури програмного забезпечення, який розділяє відповідальності між компонентами програми на три основні частини: Модель, Вид та Представник. Цей шаблон забезпечує відокремлення бізнес–логіки від логіки представлення. Приклад шаблону представлений на рис. 2.6.

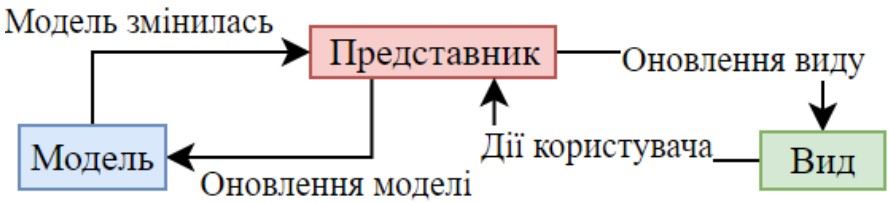

Рис.2.6. Модель–Вид–Представник

Модель відповідає за представлення даних та бізнес–логіки. Модель може включати в себе операції з отримання та зберігання даних.

Вид представляє візуальне відображення даних та інтерфейс користувача. Відповідає за взаємодію з користувачем та передачу дій Представнику.

Представник є посередником між Моделлю та Видом. Відповідає за обробку подій від користувача та взаємодію з Моделлю для отримання та оновлення даних. Представник також відповідає за оновлення Виду згідно зі змінами в Моделі.

З переваг використання шаблону Модель–Вид–Представник можна виділити:

– Легше тестування, оскільки бізнес–логіка та логіка представлення розділені.

– Відокремлення компонентів дозволяє покращити читабельність та розширюваність коду.

– Зручно для взаємодії з іншими шаблонами, такими як *Dependency Injection*.

З недоліків шаблону Модель–Вид–Представник можна виділити:

– Захоплення великою кількістю інтерфейсів та класів може робити код більш складним.

– Може вимагати більше часу для розробки порівняно з іншими шаблонами, такими як *MVVM*.

2.2. Аналіз технологій для серверної частини

Серед основних мов програмування які використовуються для розбробки додатків можна виділити *JavaScript, Java, C#, Python*. Кожна з перелічених мов також має так звані фреймворки, які допомагають в розробці.

Фреймворк (англ. "*framework*") — це структура або склад готових компонентів та інструментів, які визначають загальну архітектуру та надають базову функціональність для розробки програмного забезпечення або додатку. Фреймворк можна розглядати як набір правил, стандартів та інструкцій, які полегшують розробку та сприяють структурованості проекту.

Процес розробки за допомогою фреймворку значно полегшується завдяки наступним основним елементам:

1. Спільна архітектура: Фреймворк визначає загальну структуру проекту та розподіл обов'язків між його складовими частинами.

2. Готові компоненти: Включає набір готових до використання компонентів, які можна використовувати для створення функціональності.

3. Стандарти та правила: Встановлює основні стандарти та правила для написання коду, що сприяє однорідності та читабельності.

4. Інструментарій: Надає інструменти для автоматизації певних завдань, таких як збірка, тестування, розгортання тощо.

JavaScript, коли використовується на сервері через *Node.js*, набуває ім'я "*Node.js JavaScript*" або "*Node.js*". Розглянемо його основні характеристики.

– *Node.js* використовує асинхронну модель виконання, що робить його ефективним для обробки багатьох одночасних з'єднань.

– Забезпечує високу швидкість завдяки використанню двигуна *V8* від *Google*.

– *Node.js* має велику та активну спільноту розробників, що допомагає у вирішенні проблем та надає багато розширень.

– Не є найкращим вибором для важких обчислень, оскільки вони можуть блокувати один потік виконання.

– Велика вкладеність зворотніх викликів може вести до "*callback hell*" та ускладнювати код.

Найпопулярніші фреймворки для серверної частини на базі *JavaScript*:

Express.js – це легковаговий фреймворк для розробки веб–застосунків на мові програмування *JavaScript*, спеціально створений для забезпечення зручної роботи з серверною частиною додатків. Він базується на *Node.js*, який використовується для виконання *JavaScript* на стороні сервера.

Основні риси та можливості *Express.js*:

– Легкість використання: *Express* забезпечує простий і зрозумілий *API* для роботи з *HTTP*–запитами та відповідями, що дозволяє розробникам легко створювати серверні додатки.

– *Middleware*: Одна з ключових концепцій *Express* – це *middleware*. *Middleware* – це функції, які мають доступ до об'єктів запиту та відповіді, і можуть викликатися послідовно під час обробки запиту. Це дає можливість розробникам ланцюговано обробляти запити.

– Маршрутизація: Express дозволяє легко визначати шляхи (*routes*), що вказує на те, як обробляти різні запити *HTTP* для конкретних *URL*.

– Шаблонізація: В *Express* легко використовувати шаблонізатори для обробки *HTML*–сторінок. Популярні шаблонізатори, такі як *EJS* або *Handlebars*, можуть використовуватися з *Express*.

– Статичні файли: *Express* забезпечує можливість легко обслуговувати статичні файли, такі як *CSS, JavaScript* або зображення.

– Можливості мережі: В *Express* вбудовані функції для створення серверів, обробки мережевих запитів, роботи з параметрами запиту і відправлення відповідей клієнту.

– Розширюваність: Через велику кількість додаткових бібліотек та модулів, які можна використовувати разом з *Express*, цей фреймворк легко розширюється.

Koa.js – це легковаговий та елегантний фреймворк для розробки веб–застосунків на мові програмування *JavaScript*. Він є наступником фреймворку *Express.js* та створений тим же розробником. Однією з головних особливостей *Koa.js* є використання асинхронного програмування, що дозволяє легко та ефективно робити з обіцяннями (*Promises*) та *async*/*await*.

Основні характеристики та переваги *Koa.js*:

– Асинхронність: Основним принципом *Koa.js* є використання асинхронного програмування. Весь код Koa.js базується на обіцяннях (*Promises*) та *async*/*await*, що спрощує роботу з асинхронним кодом.

– *Middleware*: Як і в *Express*, *Koa* використовує концепцію *middleware*. Проте, *Koa* використовує менше *middleware* за замовчуванням, надаючи розробникам більше контролю над тим, як обробляються запити та відповіді.

– Контекст (*Context*): Koa використовує об'єкт контексту (*context*), який надає доступ до об'єктів запиту та відповіді, а також інших корисних функцій, таких як логування та обробка помилок.

– Маршрутизація: В *Koa* також присутня система маршрутизації, яка дозволяє визначати шляхи та обробники для обробки запитів.

– Підтримка *Generators*: В останніх версіях *JavaScript* та *Node.js, Koa* використовує *Generators* для роботи з асинхронним кодом. Це спрощує написання коду порівняно з *callback*–функціями.

– Надійність та читабельність: Код на *Koa* часто вважається більш читабельним та елегантним, завдяки використанню асинхронних функцій та генераторів.

– Легкість використання: Код *Koa.js* дуже компактний та легкий для розуміння. Він пропонує простий і зрозумілий *API.*

Nest.js – це фреймворк для розробки серверних застосунків на мові програмування *TypeScript* та *JavaScript*. Він базується на архітектурних принципах та структурі *Angular*, що робить його ідеальним вибором для тих, хто вже використовує *Angular* або хоче створити сучасний та масштабований серверний застосунок.

Основні характеристики та переваги *Nest.js*:

– Модульність: *Nest.js* побудований на основі модульної структури, що сприяє поділу застосунку на логічні модулі та спрощує його розширення.

– *Express* та *Fastify*: *Nest.js* використовує *Express* або *Fastify* як веб–фреймворки для обробки *HTTP*–запитів, що дозволяє розробникам використовувати всі можливості цих фреймворків.

– *Dependency Injection (DI)*: *Nest.js* активно використовує принцип ін'єкції залежностей для зменшення зв'язку між компонентами, що робить код більш тестованішим та підтримуваним.

– *TypeScript*: Оскільки *Nest.js* написаний на *TypeScript*, він надає статичну типізацію та інші переваги *TypeScript*, такі як автодопомога та перевірка помилок під час розробки.

– *CLI* (*Command Line Interface*): *Nest.js* має власний інтерфейс командного рядка, який спрощує створення та управління проектами.

– *WebSocket* та *Microservices*: *Nest.js* підтримує вбудовану підтримку для роботи з *WebSocket* та може використовуватися для розробки мікросервісних застосунків.

– Підтримка *GraphQL* та *REST API*: *Nest.js* надає можливість створення як *REST API*, так і *GraphQL* застосунків.

– *Active Community*: У *Nest.js* є активна та зростаюча спільнота розробників, що дозволяє швидко отримувати відповіді на питання та спільно вдосконалювати фреймворк.

Hapi.js – це фреймворк для розробки серверних застосунків на мові *JavaScript* та *Node.js*. Він відомий своєю простотою використання, гнучкістю та ефективністю. *Hapi.js* зосереджений на забезпеченні простоти створення веб–серверів та *API*, і водночас, він надає потужні можливості для розробки різноманітних застосунків.

Основні характеристики та переваги *Hapi.js*:

– Маршрутизація: *Hapi.js* надає зручний механізм для визначення маршрутів, що робить його легким у використанні та налаштуванні.

– Конфігурація: Фреймворк активно використовує конфігураційні об'єкти, що дозволяє зручно налаштовувати різні аспекти застосунку.

– Плагіни: *Hapi.js* дозволяє використовувати плагіни для розширення функціональності застосунку та полегшення обслуговування коду.

– Валідація: Можливості валідації дозволяють забезпечити правильність та цілісність даних, що надходять на сервер.

– Відомий управління життєвим циклом: *Hapi.js* надає зручний інтерфейс для обробки запитів та подій у життєвому циклі застосунку.

– Підтримка *WebSocket*: *Hapi.js* дозволяє створювати застосунки, які використовують технологію *WebSocket* для реального часу обміну даними.

C# є мовою програмування, розробленою компанією *Microsoft*, та широко використовується для розробки різноманітних додатків, включаючи серверні. Ось основні характеристики цієї мови:

– Інтеграція з *.NET*: *C#* інтегрується з *.NET Framework* та *.NET Core*, що робить його потужним та розширюваним.

– Сучасні можливості: Підтримка сучасних мовних конструкцій, таких як *async/await*, *LINQ* та інші.

– Широкі бібліотеки: Має широкий набір бібліотек та фреймворків для розробки різних застосунків.

– Обмежена переносимість: Хоча *.NET Core* зробив кроки для поліпшення переносимості, *C#* все ще пов'язаний з екосистемою *Microsoft*.

– Залежність від інструментарію *Microsoft*: Одна з основних мов для розробки під платформу *Microsoft*.

Найпопулярніші фреймворки для серверної частини на базі *C#*:

ASP.NET Core – це відкритий високопродуктивний фреймворк для розробки веб–застосунків, який розробляється *Microsoft*. Цей фреймворк є модульним, кросплатформеним і забезпечує високу продуктивність та широкий набір функцій для розробників.

Основні характеристики та переваги *ASP.NET Core*:

– Кросплатформеність: *ASP.NET Core* підтримує роботу на різних платформах, таких як *Windows, macOS* і *Linux*, що надає розробникам вибір операційної системи для розгортання своїх застосунків.

– Висока продуктивність: Фреймворк оптимізований для високої продуктивності та завантаження сторінок за мінімальний час.

– Модульність: Розробники можуть використовувати тільки ті частини фреймворку, які їм потрібні, що робить застосунки більш легкими та ефективними.

– Інтеграція зі сторонніми інструментами: Підтримка інструментів та сторонніх бібліотек, таких як *Entity Framework* для роботи з базами даних, *SignalR* для реального часу, *Swagger* для створення документації *API* та інші.

– Підтримка мікросервісної архітектури: *ASP.NET Core* ідеально підходить для розробки мікросервісів, забезпечуючи зручний механізм для розділення функціональності застосунків на окремі компоненти.

– Інструменти для тестування: Вбудовані засоби для тестування, включаючи тести одиниць, інтеграційні тести та засоби автоматизації тестування.

– Висока безпека: Засоби вбудованої безпеки, такі як *Identity* для аутентифікації та авторизації, що роблять застосунки стійкими до атак.

– Вбудована підтримка облачних служб: Інтеграція з *Azure* та іншими хмарами для легкого розгортання та масштабування застосунків.

– *Open Source*: *ASP.NET Core* є відкритим проектом з активним спільнотовим розвитком та підтримкою.

Entity Framework (*EF*) – це *ORM* (*Object–Relational Mapping*) фреймворк для платформи *.NET*, розроблений *Microsoft*. *ORM* дозволяє розробникам взаємодіяти з базою даних через об'єктно–орієнтовану модель, замість традиційного *SQL*–коду. *Entity Framework* дозволяє використовувати об'єктно–орієнтовані типи в мовах *.NET* для взаємодії з базою даних, що полегшує розробку та забезпечує більш високий рівень абстракції.

Основні характеристики *Entity Framework*:

– *Code–First* і *Database–First* підходи: *EF* підтримує *Code–First* (розробка бази даних на основі об'єктного коду) та *Database–First* (генерація моделі об'єктів з існуючої бази даних) підходи.

– *LINQ* (*Language Integrated Query*): *EF* інтегрується з мовою запитів *LINQ*, що дозволяє розробникам виразно та типобезпечно створювати запити до бази даних, використовуючи стандартні конструкції мови.

– Підтримка різних типів баз даних: *EF* може працювати з різними системами управління базами даних (*MS SQL Server*, *MySQL*, *PostgreSQL*, *SQLite*, тощо).

– *Code–First Migrations*: Міграції дозволяють автоматично оновлювати базу даних при змінах в об'єктному коді.

– *Lazy Loading* та *Eager Loading*: *Lazy Loading* дозволяє завантажувати дані з бази даних тільки при їх фактичному використанні, що зменшує навантаження. *Eager Loading* дозволяє завантажити всі необхідні дані одночасно.

– Підтримка транзакцій: *EF* забезпечує механізми для роботи з транзакціями в базі даних.

– Підтримка конвенцій та атрибутів: *EF* надає можливість конфігурувати модель об'єктів через конвенції (за замовчуванням) або за допомогою атрибутів.

– Інструменти для тестування: *EF* надає засоби для тестування, які дозволяють розробникам ефективно тестувати код, пов'язаний з базою даних.

Dapper – це мікро–*ORM* (*Object–Relational Mapping*) фреймворк для мови програмування *.NET*. Він створений для спрощення роботи з базами даних,

забезпечуючи високу продуктивність та ефективність взаємодії з базами даних. Основна ідея *Dapper* полягає в тому, щоб надати легкий та швидкий спосіб виконання запитів *SQL* та відображення результатів в об'єктно–орієнтовані структури даних.

Основні риси *Dapper*:

– Легкість використання: *Dapper* надає простий та зрозумілий *API* для виконання *SQL*–запитів та мапіння результатів в об'єкти.

– Висока продуктивність: Однією з ключових особливостей *Dapper* є висока продуктивність завдяки мінімальному набору операцій мапінгу та відсутності надмірної абстракції.

– Розширюваність: Ви можете використовувати *Dapper* разом з будь–якою базою даних, яка підтримує стандартні *ADO.NET* підключення.

– Підтримка мультимапінгу: *Dapper* дозволяє використовувати один *SQL*–запит для запитання та мапінгу декількох результатів.

– Параметризовані запити: Запити можуть бути параметризовані для уникнення *SQL*–ін'єкцій.

– Підтримка процедур бази даних: Можна використовувати *Dapper* для виклику збережених процедур бази даних.

NancyFX – це легкий та гнучкий веб–фреймворк для мови програмування *.NET*, призначений для створення веб–додатків та служб. *NancyFX* надає простий та елегантний спосіб розробки веб–додатків, виступаючи альтернативою до інших більших фреймворків, таких як *ASP.NET MVC*.

Основні особливості та переваги *NancyFX*:

– Легкість використання: *NancyFX* має простий та інтуїтивно зрозумілий синтаксис, що дозволяє розробникам швидко створювати веб–додатки без надмірних конфігурацій та складних налаштувань.

– Модульність: Додатки *NancyFX* будуються з використанням модулів, які є самодостатніми компонентами з обмеженим контекстом. Це полегшує структуру коду та робить його більш читабельним і підтримуваним.

– Розширюваність: *NancyFX* дозволяє використовувати власні розширення та плагіни для розширення функціоналу за необхідності. Це дозволяє додавати нові можливості та фічі, не втрачаючи простоту.

– Вбудована підтримка виглядів: *NancyFX* підтримує різні механізми для рендерингу виглядів, включаючи шаблони *Razor* та інші.

– Вбудована обробка маршрутів: *NancyFX* автоматично обробляє маршрути, що полегшує визначення шляхів та параметрів у вашому коді.

– Самодостатність: *NancyFX* не вимагає використання серверів, таких як *Internet Information Services* (*IIS*). Ви можете запускати *NancyFX* додатки на будь–якому сервері, що підтримує *.NET*.

Python — це високорівнева, інтерпретована мова програмування, яка здобула популярність завдяки своїй простоті та зручності використання. Ось основні характеристики мови та найпопулярніших фреймворків для серверної частини:

– Простота синтаксису: Синтаксис *Python* лаконічний та читабельний, що полегшує розробку та обслуговування коду.

– Широке застосування: *Python* використовується для веб–розробки, обробки даних, штучного інтелекту, наукових досліджень та багатьох інших галузей.

– Велика громадська спільнота: *Python* має активну та велику спільноту розробників.

– Швидкодія: *Python* може бути менш ефективним у виконанні великих обчислень порівняно з деякими іншими мовами.

Найпопулярніші фреймворки для серверної частини на базі *Python*:

Django – це відкритий веб–фреймворк для розробки веб–додатків на мові програмування *Python*. Розроблений для спрощення створення високоякісних веб–додатків, *Django* включає в себе вбудовані інструменти для вирішення багатьох типових завдань та відповідає за виконання кращих практик веб–розробки.

Основні особливості та характеристики *Django*:

– *MVC* архітектура: *Django* використовує архітектурний шаблон *Model–View–Controller* (*MVC*), але у *Django* він відомий як *Model–View–Template* (*MVT*). Це полегшує розробку та розподіл обов'язків між різними компонентами додатку.

– *ORM* (*Object–Relational Mapping*): *Django* має вбудований *ORM*, який дозволяє вам взаємодіяти з базою даних, використовуючи об'єкти *Python*, замість *SQL*–запитів. Це полегшує роботу з базою даних та забезпечує переносимість коду між різними системами управління базами даних (СУБД).

– Адміністративний інтерфейс: *Django* надає автоматично створюваний адміністративний інтерфейс, що дозволяє легко управляти даними вашого додатку. Адмін–панель можна налаштовувати та розширювати відповідно до потреб проєкту.

– Інтегрована система аутентифікації та авторизації: *Django* надає готові засоби для реалізації аутентифікації та авторизації користувачів, включаючи збереження паролів, сеанси та доступ до ресурсів.

– Шаблонізатор *Django*: *Django* використовує власний мовний шаблонізатор, який дозволяє вбудовувати *Python*–код безпосередньо в *HTML*–шаблони.

– *URL*–маршрутизація: *Django* використовує просту та зручну систему *URL*–маршрутизації, що дозволяє визначати, як обробляти різні *URL*–запити.

– Розширені можливості роботи з формами: *Django* забезпечує високорівневі засоби для створення та обробки веб–форм, включаючи перевірку валідності даних.

– Широкий спектр вбудованих бібліотек: *Django* постачається з багатьма вбудованими бібліотеками та модулями для роботи з електронною поштою, зображеннями, кешуванням, *API* та іншими.

Flask – це легкий веб–фреймворк для мови програмування *Python*, розроблений з урахуванням простоти та гнучкості. *Flask* дозволяє швидко створювати веб–додатки та *API*, забезпечуючи основні засоби для роботи з *HTTP*–запитами та відповідями.

Основні особливості та характеристики *Flask*:

– Легкість використання: *Flask* славиться своєю простотою та лаконічністю. Він не нав'язує складних конвенцій та структур, що дозволяє розробникам мати більше свободи в організації свого коду.

– Мінімальні вимоги: *Flask* вимагає мінімальних вкладень для початку роботи. Його можна встановити однією командою, і для створення додатку не потрібно багато конфігураційних файлів.

– Розширюваність: Хоча *Flask* поставляється з базовим набором функціоналу, він дозволяє розробникам легко підключати додаткові бібліотеки і розширення для вирішення конкретних завдань.

– Вбудований сервер розвитку: *Flask* має вбудований сервер розвитку, що робить його ідеальним для швидкого розгортання та тестування веб–додатків.

– Вигляди та маршрутизація: *Flask* використовує концепцію виглядів (*views*) для обробки *HTTP*–запитів. Маршрутизація визначається за допомогою декораторів, що робить процес визначення шляхів доступу зрозумілим та лаконічним.

– Шаблонізатор *Jinja2*: *Flask* використовує *Jinja2* для обробки *HTML*–шаблонів. Це дозволяє використовувати змінні, умови та цикли прямо в *HTML*–коді.

– Робота з формами: *Flask* надає інструменти для створення та обробки веб–форм, спрощуючи взаємодію з користувачами.

– Розширені можливості для роботи з *HTTP*: *Flask* дозволяє легко обробляти різні види *HTTP*–запитів, включаючи *GET, POST, PUT* та *DELETE*.

FastAPI – це модерний, швидкий (за рахунок використання вбудованого автоматичного документування та генерації *OpenAPI*), фреймворк для створення веб–додатків та *API* на мові програмування *Python*. Він спеціально розроблений для ефективного створення високопродуктивних *RESTful API*.

Основні особливості та характеристики *FastAPI*:

– Автоматичне документування: *FastAPI* генерує документацію *OpenAPI* та *ReDoc* автоматично на основі анотацій типів даних та параметрів функцій. Це полегшує розробку, тестування та розгортання *API*.

– Валідація типів даних: *FastAPI* використовує *Pydantic* для визначення та перевірки типів даних. Це дозволяє автоматично генерувати схеми валідації та документації для введення та виведення даних.

– *Starlette*: *FastAPI* ґрунтується на фреймворку *Starlette*, який відомий своєю високою швидкістю та асинхронністю. Це робить *FastAPI* ефективним в обробці великої кількості одночасних підключень.

– Асинхронність: *FastAPI* підтримує асинхронні запити, що дозволяє розробляти асинхронні веб–додатки та *API* для взаємодії з іншими асинхронними системами.

– Захист від відомих атак: *FastAPI* вбудовує захист від різних видів атак, таких як впровадження коду та перехоплення з'єднань.

– Зручна робота зі статичними файлами: *FastAPI* дозволяє легко обслуговувати статичні файли, такі як стилі, зображення та інші ресурси.

– Відмінна продуктивність: *FastAPI* пропонує високий рівень продуктивності завдяки компіляції та оптимізації коду.

– Вбудована підтримка *CORS*: Підтримка *Cross–Origin Resource Sharing* (*CORS*) дозволяє обробляти запити від клієнтів на різних доменах.

Tornado – це відомий фреймворк для розробки веб–додатків та *API* на мові програмування *Python*. Одна з ключових особливостей *Tornado* – це використання асинхронного програмування, що дозволяє легко обробляти багато одночасних з'єднань без блокування виконання коду. Нижче наведено деякі характеристики та особливості *Tornado*:

– Асинхронне програмування: *Tornado* базується на подійному циклі та використовує концепції асинхронного програмування для обробки багатьох одночасних підключень. Це робить його ефективним для великих навантажень та високоодноразових додатків.

– Вбудована підтримка *WebSocket*: *Tornado* надає вбудовану підтримку протоколу *WebSocket*, що дозволяє легко розробляти веб–додатки з реальним часом та обміном даними в режимі реального часу.

– Обробка *HTTP*–запитів: *Tornado* може бути використаний для обробки *HTTP*–запитів, і він надає функціонал для визначення маршрутів та обробки запитів власноруч.

– Шаблони та статичні файли: *Tornado* підтримує використання шаблонів для генерації *HTML*–сторінок та може обслуговувати статичні файли, такі як стилі та зображення.

– Швидкодія: Хоча *Tornado* не є найшвидшим фреймворком для веб–розробки, він демонструє хорошу продуктивність завдяки асинхронному підходу та ефективному використанню ресурсів.

– Розширюваність: *Tornado* розширюється за допомогою різних бібліотек та модулів, що дозволяє розробникам вибирати та додавати необхідний функціонал.

– Сервер *Comet*: *Tornado* може слугувати в якості сервера *Comet* для побудови високопродуктивних веб–застосунків з обміном даними в режимі реального часу.

2.3. Аналіз технологій для клієнтської частини

Під час створення фронтенду (клієнтської частини) веб–додатка, віддається перевага використанню тих же мов програмування, що і для серверної (бекенд) частини або загальної логіки додатка. Це означає, що мова програмування для розробки клієнтської частини співпадає з тією, яка використовується у бекенді. Цей підхід сприяє уніфікації розробки та полегшує спільне управління кодовою базою. Розробники можуть використовувати спільні інструменти, бібліотеки та фреймворки на обох рівнях додатка, що сприяє консистентності та взаємодії між клієнтом та сервером.

Найпопулярніші фреймворки та бібліотеки для розробки клієнтської частини на *JavaScript*:

React – це бібліотека *JavaScript* для розробки інтерфейсів користувача, яка зосереджена на створенні односторінкових додатків та компонентів інтерфейсу. Розроблена *Facebook*, вона широко використовується в індустрії для створення динамічних та інтерактивних веб–додатків.

Основні риси *React*:

– Компонентна архітектура: *React* спроектований навколо концепції компонентів, які є відокремленими та повторно використовуваними елементами інтерфейсу. Кожен компонент може мати свою логіку, стиль та стан.

– Віртуальний *DOM*: *React* використовує віртуальний *DOM* для ефективного оновлення інтерфейсу. Зміни відбуваються спочатку в віртуальному *DOM*, і тільки потім виконуються мінімально необхідні зміни в реальному *DOM*, що підвищує продуктивність.

– *JSX*: *React* використовує *JSX* (*JavaScript XML*) – розширення синтаксису *JavaScript*, яке надає можливість описувати структуру інтерфейсу, схожу на розмітку *HTML* чи *XML*, прямо в коді *JavaScript*.

– Односторінкові додатки: *React* часто використовується для створення односторінкових додатків (*SPA*), де сторінка не перезавантажується при взаємодії користувача, а зміни відбуваються динамічно.

– Екосистема та спільнота: *React* має широку екосистему, яка включає в себе багато корисних бібліотек і інструментів, таких як *Redux* для управління станом додатка, *React Router* для навігації, тощо.

React забезпечує ефективний та зручний спосіб розробки інтерфейсів, і його популярність продовжує зростати завдяки простоті використання та потужному функціоналу.

Angular – це фреймворк для розробки веб–додатків, розроблений командою *Angular* у *Google*. Він надає повноцінні інструменти для створення як простих веб–сайтів, так і складних односторінкових додатків (*SPA*). Основною метою *Angular* є полегшення розробки та тестування таких додатків шляхом надання комплексного набору функцій та інструментів.

Основні характеристики *Angular*:

– Компонентна архітектура: *Angular* базується на ідеї компонентів, які є основними будівельними блоками додатку. Кожен компонент включає в себе розмітку, логіку та стилі.

– Двостороннє зв'язування даних: *Angular* надає можливість автоматично оновлювати інтерфейс користувача при зміні даних в моделі, а також змінювати дані в моделі при зміні введення користувача.

– Залежності та ін'єкції залежностей: *Angular* використовує систему ін'єкції залежностей для управління залежностями та створення компонентів з чітко визначеними інтерфейсами.

– Маршрутизація: *Angular* має вбудовану систему маршрутизації, яка дозволяє створювати *SPA* з різними сторінками та відстежувати стан додатка.

– Тестування: *Angular* підтримує різні підходи до тестування, включаючи юніт–тести та *e2e*–тести.

– Розширена екосистема: *Angular* має широку екосистему, що включає в себе такі інструменти, як *Angular CLI* для швидкої розробки, *Angular Material* для готових *UI*–компонентів, і ін.

Angular створений для того, щоб забезпечити структурований та ефективний підхід до розробки веб–додатків і підтримується активною спільнотою розробників.

Vue.js – це прогресивний фреймворк для розробки користувацьких інтерфейсів, який широко використовується для створення веб–додатків та односторінкових додатків (*SPA*). Однією з ключових особливостей *Vue.js* є його легкість використання та гнучкість, що робить його привабливим для розробників різного досвіду.

Основні характеристики *Vue.js*:

– Простота використання: *Vue.js* відзначається легкістю вивчення та використання. Його можна включити в існуючий проект або використовувати для створення проекту "з нуля".

– Двостороннє зв'язування даних: *Vue.js* підтримує двостороннє зв'язування даних, що дозволяє автоматично оновлювати відображення даних при їхньому зміні.

– Компонентна архітектура: Як і в інших сучасних фреймворках, *Vue.js* використовує компоненти для структуризації коду та покращення повторного використання.

– Директиви: *Vue.js* має вбудовані директиви, такі як *v–if, v–for*, і *v–bind*, які роблять маніпулювання *DOM* більш зручним та декларативним.

– Модульність: Фреймворк дозволяє використовувати лише ті частини, які необхідні для конкретного проекту, що сприяє швидкому старту та оптимізації додатків.

– Екосистема: *Vue.js* має активну спільноту розробників і багато розширень, бібліотек та інструментів, таких як *Vue Router* для роботи з маршрутами та *Vuex* для управління станом додатку.

Vue.js є відмінним вибором для тих, хто цінує простоту та ефективність у розробці веб–додатків.

jQuery – це широко використовувана бібліотека *JavaScript*, яка спрощує взаємодію з *HTML*–документами, обробку подій, анімацію та взаємодію з сервером за допомогою *AJAX*. Заснована в 2006 році, *jQuery* допомагає розробникам робити веб–розробку більш ефективною та зменшує кількість коду, який потрібно писати для досягнення певного ефекту.

Основні характеристики *jQuery*:

– Легкість використання: *jQuery* спрощує вибірку та маніпулювання *DOM*–елементами, роблячи код коротшим та більш зрозумілим.

– Взаємодія з *AJAX*: *jQuery* надає простий і зручний спосіб взаємодії з сервером за допомогою техніки *AJAX*, що дозволяє асинхронно оновлювати частини сторінки без перезавантаження всього документа.

– Анімація: Бібліотека надає можливості для легкої реалізації анімацій, які можуть використовуватися для створення візуальних ефектів.

– Робота з подіями: *jQuery* пропонує зручний інтерфейс для обробки подій, таких як клік, подвійний клік, зміна стану тощо, що полегшує роботу з користувацькими взаємодіями.

– Розширені можливості: Велика кількість розширень та плагінів для *jQuery* дозволяє розробникам легко розширювати його функціонал.

– Кросбраузерність: *jQuery* розроблено з урахуванням кросбраузерності, що означає, що код, написаний на *jQuery*, працює однаково на різних веб–переглядачах.

Хоча сучасні фреймворки, такі як *React*, *Angular* та *Vue.js*, набули великої популярності, *jQuery* залишається корисним інструментом для простих задач та для тих, хто продовжує підтримувати старі проекти.

Найпопулярніші фреймворки для розробки клієнтської частини на *Java*:

JavaFX – це платформа для розробки багатоплатформових десктопних та вбудованих (*embedded*) застосунків на мові програмування *Java*. Цей фреймворк надає розробникам інструменти для створення графічних інтерфейсів користувача (*GUI*), відтворення мультимедіа, анімації та інших інтерактивних елементів.

Основні характеристики та особливості *JavaFX*:

– Графічний двигун: *JavaFX* використовує сучасний графічний двигун, що дозволяє створювати зручні та естетичні інтерфейси з високою продуктивністю.

– *FXML* та *Scene Builder*: Існує можливість описувати інтерфейс користувача за допомогою мови розмітки *FXML*, а також використовувати інструмент *Scene Builder* для візуального створення *GUI*.

– Мультимедіа та анімація: *JavaFX* підтримує відтворення аудіо та відео, а також дозволяє створювати різні види анімацій, що робить його ідеальним для розробки мультимедійних застосунків.

– Багатофункціональні компоненти: *JavaFX* надає широкий набір готових компонентів для побудови інтерфейсу, таких як кнопки, тексти, таблиці, графіки тощо.

– Модульність: З *Java* 9 та впровадженням модульності в *Java*, *JavaFX* став доступним як окремий модуль, що дозволяє використовувати його як невеличку частину або повністю залежно від потреб.

– Вбудовані взаємодії з *Java*: *JavaFX* інтегрований з іншими компонентами платформи Java, що спрощує роботу з іншими *Java*–бібліотеками та фреймворками.

– Підтримка *CSS*: Використання *Cascading Style Sheets* (*CSS*) для оформлення інтерфейсу дозволяє розробникам легко визначати зовнішній вигляд компонентів.

JavaFX є сучасним та потужним інструментом для розробки десктопних застосунків на мові програмування *Java*, забезпечуючи розробникам зручний інструментарій для створення візуально привабливих та функціональних додатків.

SWT (*Standard Widget Toolkit*) – це набір бібліотек та інструментів для розробки графічного інтерфейсу користувача (*GUI*) в мові програмування *Java*. Він був розроблений та підтримується *Eclipse Foundation* і став важливою частиною інфраструктури для розробки різноманітних застосунків на *Java*.

Основні характеристики та особливості *SWT*:

– Вбудоване в *Java*: *SWT* входить в комплект стандартної поставки *Java*, що дозволяє використовувати його безпосередньо без додаткового встановлення або конфігурування.

– Нативний вигляд і взаємодія: Основна ідея *SWT* – це використання нативних компонентів відповідних операційних систем (*Windows*, *macOS*, *Linux*) для досягнення більшої швидкодії та природного вигляду застосунку на кожній платформі.

– Низькорівневий доступ: *SWT* надає низькорівневий доступ до нативних функцій, що дозволяє розробникам більше контролювати взаємодію з операційною системою та нативними бібліотеками.

– Мале використання ресурсів: *SWT* оптимізований для мінімізації використання ресурсів системи, забезпечуючи високу продуктивність та ефективність.

– Широкий вибір компонентів: *SWT* містить в собі різноманітні вбудовані компоненти, такі як кнопки, тексти, таблиці, дерева, які можна використовувати для побудови різноманітних інтерфейсів.

– Підтримка захисту від уразливостей: З врахуванням особливостей низькорівневого доступу до нативних функцій, *SWT* надає можливості для реалізації заходів безпеки та захисту від потенційних уразливостей.

– Платформозалежність: Оскільки *SWT* використовує нативні компоненти, код *SWT* може бути трохи платформозалежним, але це дозволяє досягти кращого інтеграції та взаємодії на різних операційних системах.

SWT є потужним інструментом для розробки графічного інтерфейсу в середовищі Java, забезпечуючи розробникам зручний спосіб створення швидких та

ефективних десктопних застосунків з високою продуктивністю та природним виглядом на різних платформах.

Java Swing – це набір бібліотек та класів для розробки графічних інтерфейсів користувача (*GUI*) для програм, написаних на мові програмування *Java*. Використання Java Swing дозволяє створювати кросплатформенні та незалежні від платформи графічні інтерфейси для різних типів додатків, включаючи стандартні десктопні програми, інструменти розробки та інші застосунки.

Основні характеристики та особливості *Java Swing*:

– Кросплатформенність: Завдяки використанню власних елементів управління, *Java Swing* забезпечує кросплатформенність, що означає, що програми можуть працювати на будь–якій підтримуваній платформі без змін вихідного коду.

– Легкість використання: Java Swing надає велику кількість готових компонентів для створення різноманітних елементів інтерфейсу, таких як кнопки, поля введення, таблиці тощо.

– Можливості настроювання: Розширені можливості настроювання зовнішнього вигляду та поведінки компонентів, що дозволяє створювати індивідуальний та естетичний дизайн.

– Подійно–орієнтована модель програмування: Заснована на подіях (*event–driven*) модель програмування спрощує обробку подій та взаємодію з користувачем.

– Мультипоточність: Підтримка мультипоточності для забезпечення різних операцій, що виконуються паралельно, наприклад, обробка подій та оновлення інтерфейсу.

– Підтримка перетягування та опускання (*drag and drop*): Забезпечує можливість перетягувати та відпускати елементи інтерфейсу.

– Графічні та текстові компоненти: Включає в себе різноманітні графічні елементи, такі як графічні кнопки, текстові області, зображення тощо.

– Підтримка моделі даних: Можливість легко прив'язувати дані до компонентів інтерфейсу.

Java Swing використовується для розробки класичних десктопних застосунків та є однією з популярних технологій для створення графічних інтерфейсів в світі *Java*–розробки.

Найпопулярніші фреймворки для розробки клієнтської частини на *C#*:

Windows Presentation Foundation (*WPF*) – це технологія розробки графічних інтерфейсів користувача (*GUI*) в середовищі *.NET Framework*, що надає розширені можливості для створення сучасних та зручних десктопних застосунків для операційних систем *Windows*. Основними характеристиками *WPF* є використання векторної графіки, подійно–орієнтована модель програмування та розширені засоби стилізації і анімації.

Основні компоненти та особливості *WPF*:

– *XAML* (*eXtensible Application Markup Language*): *WPF* використовує мову опису *XAML* для розмітки інтерфейсу. *XAML* дозволяє визначати елементи інтерфейсу, їх властивості та взаємодію за допомогою декларативного синтаксису.

– Векторна графіка: *WPF* базується на векторній графіці, що дозволяє створювати гнучкі та масштабовані інтерфейси, які адаптуються до різних розмірів екранів.

– Подійно–орієнтована модель програмування: *WPF* використовує модель подій для обробки взаємодії з користувачем. Це дозволяє легко реагувати на події, такі як натискання кнопок чи переміщення миші.

– Шаблони і стилі: *WPF* надає потужні інструменти для стилізації та оформлення інтерфейсу. Можна визначати власні стилі для елементів та використовувати шаблони для кастомізації вигляду.

– Анімація: *WPF* підтримує створення різноманітних анімацій для зроблення інтерфейсу більш динамічним та привабливим для користувача.

– *Data Binding*: Вбудована підтримка прив'язки даних дозволяє легко зв'язувати елементи інтерфейсу з джерелами даних, що спрощує роботу з великим обсягом інформації.

– Система макетів: *Grid, StackPanel* та інші контейнери дозволяють створювати гнучкі та комплексні макети для розміщення елементів.

– *3D*–графіка: *WPF* підтримує роботу з тривимірною графікою, що дозволяє створювати додатки з елементами *3D*–графіки.

WPF є важливою технологією для розробки *Windows*–додатків, особливо тих, які вимагають сучасний та привабливий інтерфейс користувача.

.NET MAUI (*Multi–platform App UI*) – це мультиплатформений фреймворк для розробки мобільних, десктопних та вбудованих додатків, який є частиною екосистеми *.NET*. *MAUI* надає зручний і консистентний спосіб створення додатків для різних платформ, таких як *Android, iOS, macOS, Windows* та інші.

Основні характеристики та особливості *.NET MAUI*:

– Мультиплатформеність: Розробники можуть використовувати *.NET MAUI* для створення додатків, які працюють на різних платформах, не переписуючи код для кожної окремо.

– Спільний код: *.NET MAUI* використовує спільний код для логіки додатка, що дозволяє зберігати спільний функціонал для всіх платформ.

– *XAML*: Розмітка інтерфейсу користувача в *.NET MAUI* виконується за допомогою мови опису *XAML*, що дозволяє легко розділяти дизайн від логіки додатка.

– Однорідний інтерфейс: *MAUI* надає стандартизовані елементи управління для створення однорідного інтерфейсу користувача на різних платформах.

– Підтримка графіки та анімацій: *MAUI* використовує сучасні засоби для роботи з графікою та анімаціями, що робить додатки привабливими та ефективними.

– Інструментарій *Visual Studio*: *.NET MAUI* повністю інтегрований з середовищем розробки *Visual Studio*, що дозволяє розробникам ефективно створювати та налагоджувати додатки.

– Гнучкість конфігурації: Можливість конфігурування додатків для різних платформ, враховуючи їхні особливості.

.NET MAUI спрощує розробку кросплатформених додатків, забезпечуючи швидкий та ефективний спосіб створення додатків для різних пристроїв та операційних систем.

Найпопулярніші фреймворки для розробки клієнтської частини на *Python*:

PyQt – це набір засобів для розробки графічних інтерфейсів користувача (*GUI*) в мові програмування *Python*. *PyQt* використовує *Qt* – потужну бібліотеку для створення *GUI*, що надає широкі можливості для розробників.

Основні характеристики *PyQt*:

– *Qt Integration*: *PyQt* надає повний доступ до функціоналу *Qt*, що включає в себе віджети для створення вікон, кнопок, полів введення та інших елементів інтерфейсу.

– Кросплатформенність: *PyQt* дозволяє створювати кросплатформенні додатки, які можуть працювати на різних операційних системах, таких як *Windows, macOS* та *Linux*.

– *PyQt Designer*: Це графічний інструмент для швидкого створення *GUI*. Розробники можуть візуально розміщати та налаштовувати віджети, що спрощує процес створення інтерфейсу.

– Сигнали та слоти: *PyQt* використовує механізм сигналів та слотів, який спрощує обробку подій та взаємодію об'єктів в *GUI*.

– Широкі можливості налаштувань: Розширені можливості конфігурації та стилізації елементів інтерфейсу, що дозволяє створювати додатки з індивідуальним дизайном.

– Ліцензія *GPL* або комерційна ліцензія: *PyQt* доступний під ліцензією *GPL*, але також є можливість придбати комерційну ліцензію для проектів з закритим кодом.

PyQt використовується для створення різноманітних додатків, від простих інструментів до складних програм з графічним інтерфейсом. Завдяки своїй потужній функціональності та можливостям, *PyQt* є популярним вибором серед розробників *Python* для створення *GUI*–додатків.

Kivy – це відкрите програмне забезпечення для розробки мультимедійних додатків з відкритим кодом. Він спеціалізується на розробці мобільних додатків, особливо тих, які мають мультимедійні та сенсорні інтерфейси. Основною метою *Kivy* є забезпечення простої та ефективної розробки кросплатформених додатків для різних пристроїв.

Основні особливості *Kivy*:

– Кросплатформенність: *Kivy* дозволяє створювати додатки, які можуть працювати на різних операційних системах, включаючи *Android, iOS, Windows, Linux* і *macOS*.

– Декларативний мовний опис інтерфейсу: *Kivy* використовує декларативні мови опису мови *Python* та *KV* для визначення користувальницького інтерфейсу та взаємодії.

– Відмінна підтримка графіки: Краща підтримка графіки в *Kivy* дозволяє створювати додатки з анімаціями та графічною привабливістю.

Kivy добре підходить для розробки ігор, плеєрів, освітніх програм та інших додатків, що вимагають мультимедійних та тачскрін–інтерфейсів. Висока кросплатформенність робить *Kivy* популярним вибором для розробки додатків для мобільних пристроїв та планшетів.

2.4. Аналіз баз даних

Існує кілька типів баз даних, і кожен з них має свої переваги та недоліки. Найпоширеніші типи баз даних включають реляційні та нереляційні бази даних.

Реляційні бази даних є однією з найпоширеніших та ефективних форм організації та зберігання даних. Основним принципом реляційних баз даних є використання таблиць для організації даних у вигляді рядків та стовпців, де кожен рядок представляє конкретний запис, а кожен стовпець – конкретний атрибут чи поле даних. Реляційні бази даних є основою багатьох сучасних систем управління базами даних і забезпечують надійний та консистентний доступ до даних для різних застосунків та веб–систем.

Реляційні бази даних зазвичай використовуються в сферах, де важлива точність та цілісність даних, таких як банківська справа чи системи управління клієнтами.

Нереляційні бази даних є альтернативою реляційним системам управління базами даних і використовуються для зберігання та обробки даних в середовищах,

де реляційні моделі можуть бути обмеженими або неефективними. Основна відмінність між ними полягає в тому, що нереляційні бази даних не використовують традиційні таблиці та *SQL*–мову для управління даними. Нереляційні бази даних знаходять широке застосування в різних областях, зокрема в Інтернеті речей, великих даних (*Big Data*), аналізі соціальних мереж, реального часу та інших сценаріях, де потрібно ефективно працювати з різноманітними типами даних та масштабувати інфраструктуру. Приклад нереляційних баз даних представлений на рис. 2.7.

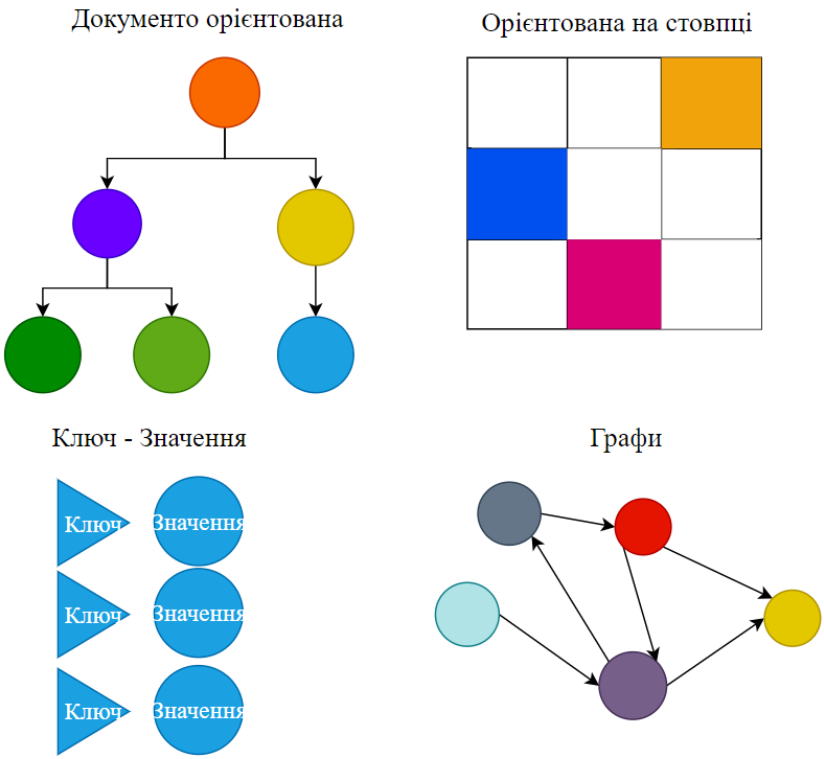

Рис.2.7. Типи нереляційних баз даних

Після теоретичного розгляду та опису різних типів баз даних, наступним кроком є детальний аналіз конкретних баз даних.

MySQL – це система управління базами даних (СУБД), яка використовується для зберігання та управління структурованою інформацією. Одна з найпопулярніших відкритих реляційних баз даних, *MySQL*, була розроблена компанією *MySQL AB*, а зараз належить корпорації *Oracle*.

MySQL знайшов широке застосування у веб–розробці, системах управління контентом, електронної комерції та інших областях завдяки своїм можливостям, ефективності та доступності.

Типові використання:

– *MySQL* використовується для зберігання та обробки даних у різних сценаріях, включаючи веб–розробку, бізнес–застосунки, мобільні додатки та інші.

– Підтримка транзакцій робить його популярним в тих галузях, де важлива цілісність даних, наприклад, у банківській сфері.

PostgreSQL є потужною та розширюваною системою управління базами даних (СУБД) з акцентом на розширені можливості та дотримання стандартів *SQL*. Долаючи обмеження традиційних реляційних баз даних, *PostgreSQL* надає багато функцій, які роблять його привабливим для широкого кола застосувань.

PostgreSQL широко використовується для великих та складних систем управління даними, де вимагається розширені можливості, стабільність та надійність.

Типові використання:

– *PostgreSQL* широко використовується у різних галузях, включаючи веб–розробку, геоінформаційні системи, бізнес–аналітику та багато іншого.

– Висока рівень стандартності *SQL* робить його зручним для використання для тих, хто вже знайомий з *SQL*.

MongoDB є нереляційною (*NoSQL*) системою управління базами даних, яка базується на документ–орієнтованій моделі даних. Основними особливостями *MongoDB* є гнучкість у роботі з даними та можливість працювати з великими обсягами структурованих та неструктурованих даних.

MongoDB широко використовується в сучасних веб–додатках, аналітиці даних, а також проектах, де необхідно працювати з великими обсягами неструктурованих даних.

Типові використання:

– *MongoDB* часто використовується для зберігання та обробки великих обсягів документоорієнтованих даних.

– Добре підходить для сценаріїв, де необхідно забезпечити гнучкість сховища даних та де модель даних може часто змінюватися.

Microsoft SQL *Server* (*MSSQL*) – це реляційна система управління базами даних (*RDBMS*), розроблена компанією Microsoft. Вона надає потужні можливості для зберігання та обробки структурованих даних, і використовується в широкому спектрі застосувань, від корпоративних систем до веб–додатків.

Microsoft SQL Server застосовується в різноманітних областях, включаючи корпоративні інформаційні системи, банківську сферу, системи управління виробництвом та багато інших.

Типове використання:

– *Microsoft SQL Server* широко використовується у корпоративних середовищах для зберігання та управління реляційними даними.

– Застосовується в різних галузях, включаючи фінанси, логістику, телекомунікації та інші.

SQLite – це легка, компактна та самодостатня вбудовувана система управління базами даних (СУБД), яка не вимагає окремого сервера та працює в режимі вбудованого додатка.

SQLite знаходить застосування в широкому спектрі проектів, включаючи мобільні додатки, настільні програми, вбудовані системи та інші випадки, де важлива простота та ефективність.

Типове використання:

– *SQLite* широко використовується в мобільних додатках, вбудованих системах, браузерах та інших легких програмах.

– Його легкість та вбудовуваність роблять його ідеальним для сценаріїв, де потрібна невелика та проста база даних.

2.5. Переваги і недоліки обраних технологій

Після детального аналізу різноманітних архітектурних підходів та шаблонів було обрано монолітну архітектуру та шаблон Модель–Вид–Оновлення.

Вибір монолітної архітектури обумовлений легшою розробкою та відсутністю недоліків для малих додатків. В майбутньому можна провести перегляд коду і розбити моноліт на розподілену системи, таким чином можна отримати найкраще від обох архітектур в різні періоди життя додатку.

Вибір шаблону Модель–Вид–Оновлення обумовлений технологією розробки додатка, адже зазвичай різні фреймворки використовують різні архітектурні шаблони, наприклад *Windows Forms Presentation* (*WPF*) використовує Модель–Вид–Представлення моделі, а *ASP.NET Core* використовує Модель–Вид–Контролер.

В даному ж варіанті було обрано *.NET MAUI Blazor Hybrid*, який поєднує в собі кращі характеристики для крос–платформенної розробки. Він в свою чергу використовує шаблон Модель–Вид–Оновлення. *MAUI* було обрано через зручність використання загальний код для клієнтської і серверної частини, а також через крос–платформеність, яка дозволяє писати загальний код для різних операційних систем, але з можливістю використання функцій контректної системи. *Blazor Hybrid* був обраний через те, що він дозволяє використовувати веб–технології, такі як *Razor* та компоненти. Приклад як працює крос–платформеність в *.NET MAUI* зображено на рис. 2.8.

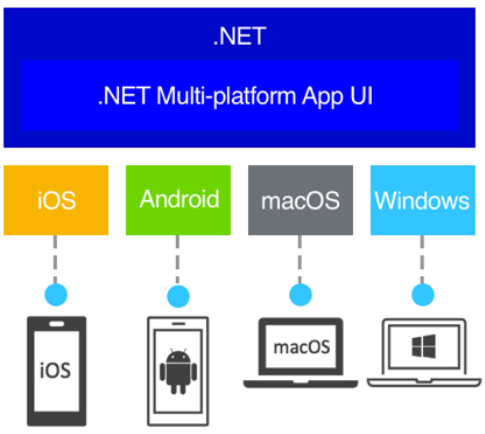

Рис.2.8. Структура крос–платформеності в *.NET MAUI*

Також однією з головних переваг виристання саме *Blazor* є можливість використання *JavaScript* на клієнтській стороні, через те, що графічний інтерфейс

будується за допомогою *HTML, CSS* та *JavaScript*. В додаток до цього можна використати *Bootstrap*, який дозволить швидше розроблювати сучасні графічні інтерфейси.

Bootstrap є відкритим фреймворком для розробки веб–сайтів та веб–додатків. Він містить набір інструментів для швидкої та ефективної розробки інтерфейсу, що включає в себе стилі, компоненти та *JavaScript*–розширення.

Обрання *SQLite* та *Entity Framework Core* обґрунтовано прагматичним підходом до розробки крос–платформених мобільних додатків. *SQLite* забезпечує легкість та ефективність для вбудованої бази даних, особливо в мобільних додатках, а *Entity Framework Core* використовується для спрощення взаємодії з базою даних через *ORM*. Такий стек технологій дозволяє оптимально використовувати можливості кожної технології з урахуванням вимог та завдань проекту.

Висновки за розділом

У даному розділі проведено докладний аналіз технологічних аспектів розробки системи, спрямованої на аналіз та прогнозування фінансових потоків. Поглиблене вивчення основних складових, таких як архітектура, серверна та клієнтська частини, бази даних, а також вибір мов програмування та фреймворків, дозволило здійснити обґрунтований вибір технологічного стеку.

Під час порівняння архітектурних підходів було вирішено вибрати монолітну архітектуру з метою спрощення процесів розгортання та підтримки системи. Вибір *.NET MAUI Blazor Hybrid* був зумовлений бажанням використовувати веб–технології для розробки мобільного додатка, забезпечуючи тим самим уніфікацію технологічного стеку на обох сторонах системи – клієнтській та серверній.

Технологія *Entity Framework Core* була обрана як інструмент для спрощення взаємодії з базою даних, забезпечуючи зручний та консистентний доступ до фінансової інформації. *Bootstrap* вибрано для створення інтуїтивно зрозумілого та естетичного інтерфейсу, забезпечуючи єдинообразний дизайн для різних пристроїв та полегшуючи взаємодію з додатком на різних платформах.

Однією з ключових виборів була реляційна база даних *SQLite*, обрана для ефективного зберігання фінансових даних на пристроях користувачів, забезпечуючи при цьому зручний та ефективний доступ до інформації.

Таким чином, було обрано технології, що дозволяють ефективно використовувати спільний код між платформами. Розглянуті технології надають гнучкість у розробці та підтримці фінансового додатка. Обрані технології підходять для створення зручного та ефективного інструменту для аналізу та прогнозування фінансових потоків.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ

3.1. Розробка структур даних

Розробка структур даних є ключовою складовою будь–якого програмного забезпечення, яке працює з базами даних. У світі *.NET* розробки одним із потужних інструментів для цього є *Entity Framework Core* (*EF Core*). *EF Core* дозволяє розробникам зручно взаємодіяти з базами даних, використовуючи об'єктно–реляційну модель.

Один з підходів до розробки структур даних – *Code First*. Замість того, щоб починати з опису бази даних, розробник спочатку визначає класи, які представлятимуть дані, а потім *EF Core* автоматично створює схему бази даних на основі цих класів. Цей метод дозволяє зосередитися на логіці програми, а структура бази даних "за кулісами" буде автоматично синхронізуватися.

Основні етапи роботи за підходом *Code First* в *Entity Framework Core*:

Створення класів: Розробник визначає класи, які представлять таблиці в базі даних.

```
public class Expense
{
public int Id { get; set; }
public string Description { get; set; }
public decimal Amount { get; set; }
}
```

Додавання атрибутів: Розробник може додавати атрибути або конфігураційні файли для налаштування деталей, таких як ключі, зв'язки та інші.

```
public class Expense
{
```

```
    [Key]
    public int Id { get; set; }
    public string Description { get; set; }
    public decimal Amount { get; set; }
}
```

Створення контексту даних: Розробник створює контекст даних, який унаслідований від *DbContext* і включає властивості для *DbSet* класів.

```
public class ApplicationDbContext : DbContext
{
    public DbSet<Expense> Expenses { get; set; }
}
```

Конфігурація підключення: У контексті даних розробник конфігурує підключення до бази даних.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
optionsBuilder.UseSqlServer("Server=(localdb)\\mssqllocaldb;Database=ExpenseTrackerDb;Trusted_Connection=True;");
}
```

Створення бази даних: При запуску програми *Entity Framework Core* автоматично створить базу даних на основі класів та їх атрибутів.

Застосування змін: Під час змін в класах або контексті даних розробник викликає команду для застосування змін до бази даних.

```
using (var context = new ApplicationDbContext())
{
    context.Database.Migrate();
}
```

Переваги *Code First*:

– Швидкість розробки: Швидкий старт розробки без потреби в написанні *SQL*–запитів для створення бази даних.

– Гнучкість: Зміни в базі даних легко робити шляхом модифікації класів та їх атрибутів.

Недоліки *Code First*:

– Менше контролю: Можливо втратити деякий контроль над структурою бази даних, особливо в складних випадках.

– Не завжди оптимально: В окремих випадках автоматичне створення структури може не бути оптимальним вирішенням.

У випадку *Database First*, процес відбувається в зворотному порядку. Розробник починає з існуючої бази даних, і *Entity Framework Core* генерує класи на основі цієї бази даних. Це корисно, коли вже існує готова база даних, і вам потрібно створити модель даних, яка буде використовуватися в програмі.

Ключові етапи використання *Database First*:

Створення бази даних: База даних створюється або використовується з існуючої.

Визначення моделі: За допомогою інструментів *Entity Framework* (зокрема, команди *Scaffold–DbContext* в *.NET Core*), визначається модель, яка буде відображати структуру бази даних. В *.NET Core* можна використовувати команду *Scaffold–DbContext* з пакетом *Entity Framework* для визначення моделі даних.

Ця команда генерує модель даних у каталозі *Models* на основі існуючої бази даних *Microsoft SQL Server*.

Генерація класів: *Entity Framework* автоматично генерує класи для таблиць та контекст бази даних.

Використання згенерованих класів: Розробник може використовувати згенеровані класи для взаємодії з базою даних через *LINQ*–запити.

Переваги *Database First*:

– Робота з існуючим: Ідеально підходить для роботи з існуючими базами даних.

– Швидкий старт: Швидкий старт роботи, особливо, коли потрібно взаємодіяти з базою даних, яка вже існує.

Більший контроль: Забезпечує більший контроль над структурою бази даних.

– Недоліки *Database First*:

– Менша гнучкість: Менша гнучкість порівняно з *Code First* при створенні нових проектів.

Складніші зміни: Зміни в структурі бази даних можуть бути складнішими.

Підхід *Model First* означає створення моделі даних за допомогою спеціального інструмента (наприклад, *Entity Framework Designer* в *Visual Studio*) без прив'язки до конкретної бази даних. Потім на основі цієї моделі *EF Core* створить або оновить базу даних.

Entity Framework Designer – це графічний інтерфейс для визначення моделі даних в *Model First*. Він включений в *Visual Studio* та дозволяє створювати та редагувати модель даних, перетягуючи та розміщуючи таблиці та зв'язки.

Спосіб роботи з *Entity Framework Designer*:

1. Додавання таблиць: Можна обрати таблиці та поля, які потрібно додати до моделі.

2. Визначення зв'язків: Можна встановити взаємозв'язки між таблицями за допомогою відносин.

3. Генерація *SQL*–Скриптів: Після визначення моделі можна генерувати SQL–скрипти для створення бази даних.

Також Visual Studio можна використовувати команду "*Generate Database*" для генерації бази даних на основі визначеної моделі. Ця команда створює *SQL*–скрипти для створення бази даних та генерує файли згідно визначеної моделі.

Ключові етапи використання *Model First*:

1. Створення моделі: Розробник визначає модель даних за допомогою графічного інтерфейсу (наприклад, *Entity Framework Designer*) або мови моделювання (*XML* або *EDMX* файл).

2. Генерація бази даних: *Entity Framework* генерує скрипти для створення бази даних на основі визначеної моделі.

3. Створення класів: *Entity Framework* автоматично генерує класи для взаємодії з базою даних на основі визначеної моделі.

4. Використання згенерованих класів: Розробник можете використовувати згенеровані класи для взаємодії з базою даних через *LINQ*–запити.

Переваги *Model First*:

– Швидкий прототип: Ідеально підходить для швидкого створення прототипів та експериментів.

– Графічний інтерфейс: Використання графічного інтерфейсу дозволяє візуально моделювати структуру даних.

Недоліки *Model First*:

– Обмежена гнучкість: Може бути менше гнучким для складних структур даних.

– Генерація *SQL*–Скриптів: Генеровані *SQL*–скрипти можуть бути складні для налаштування та оптимізації.

Розробка системи "з нуля" передбачає вибір оптимального методу створення структур даних. У даному випадку *Code First* є найбільш підходящим варіантом. Цей підхід дозволяє створювати модель даних за допомогою коду програми, без необхідності вручну визначати схему бази даних. Такий метод є ефективним і простим для використання, сприяючи швидкому початку розробки та легкій модифікації структури даних в майбутньому.

Структура бази даних необхідна для роботи додатку представлена на рис. 3.1.

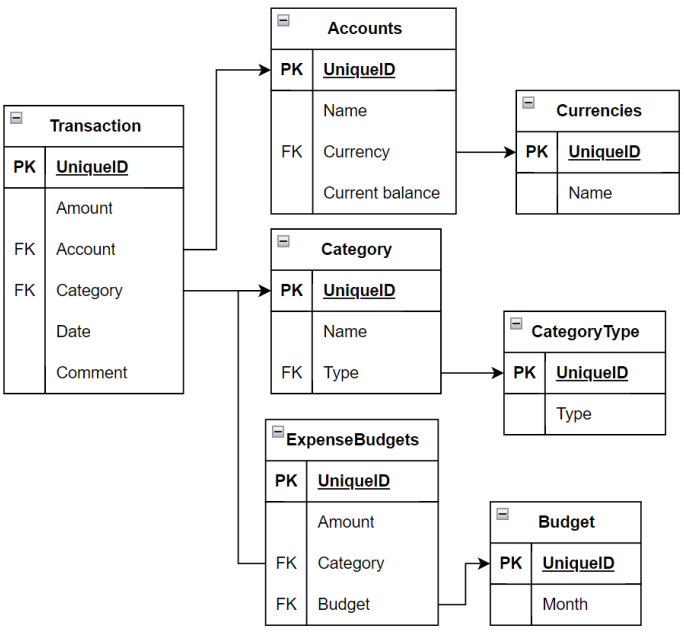

Рис.3.1. Структура бази даних

На даній структурі представлені транзакції, дохід та витрати зі своїми окремими категоріями, рахунки з прив’язкою до валюти та бюджет на витрати. Для того щоб створити таку структуру через підхід Code First без необхідності дублювати деякі поля необхідно створити один базовий клас, який допоможе зменшити повторюваність. Приклад такого класу зображено на рис. 3.2.

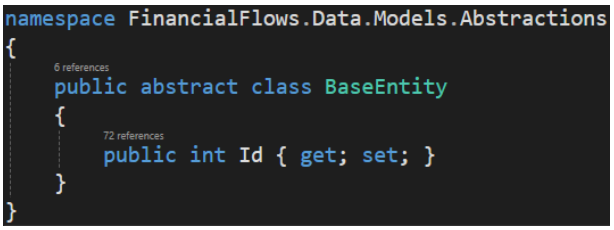

Рис.3.2. Базовий абстрактний клас

Цей код представляє абстрактний клас, які можуть бути використані для моделювання основних сутностей у системі бюджетування.

– *BaseEntity* (Основний клас сутності):

Властивість *Id*: Представляє унікальний ідентифікатор для кожної сутності.

Цей клас можна успадковувати для подальшого розширення та конкретизації в конкретних класах.

Використання абстрактних класів дозволяє створювати загальні шаблони для об'єктів, які мають спільні характеристики, і уникати повторюваності коду.

Після створення базового класу можна почати створювати інші моделі, наприклад класи *Account* і *Currency*. Приклад цих класів зображений на рис. 3.3.

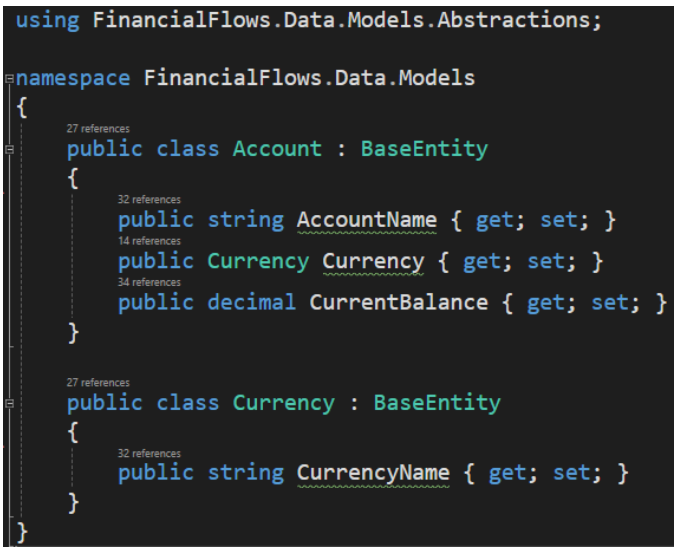

Рис.3.3. Клас для валют та різних акаунтів

Account class:

– *AccountName*: Властивість, що представляє ім'я облікового запису.

– *Currency*: Властивість, яка вказує на валюту облікового запису (використовується клас *Currency*).

– *CurrentBalance*: Властивість, що вказує на поточний баланс облікового запису.

Currency class:

– *CurrencyName*: Властивість, що представляє ім'я валюти.

Маючи ці моделі можна буде додати свій банківський рахунок, вказати його баланс, а також валюту в якій він був випущений.

Далі необхідно додати модель для категорій транзакцій, а також самих транзакцій. Приклад таких класів зображені на рис. 3.4 та рис. 3.5.

```
using FinancialFlows.Data.Models.Abstractions;

namespace FinancialFlows.Data.Models
{
    28 references
    public class Category : BaseEntity
    {
        36 references
        public string CategoryName { get; set; }
        34 references
        public CategoryType CategoryType { get; set; }
    }
    15 references
    public enum CategoryType
    {
        Income,
        Expense
    }
}
```

Рис.3.4. Клас для категорій

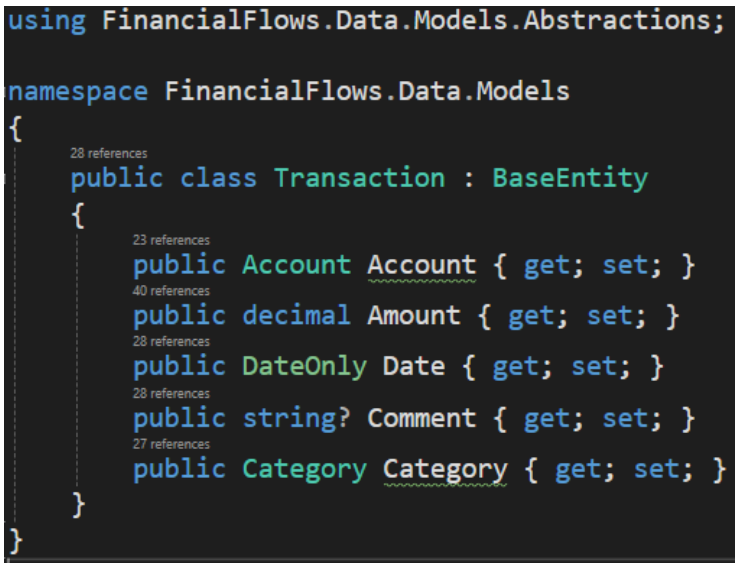

Рис.3.5. Клас для транзакцій

Опис класу для категорій:

– *CategoryName*: Властивість, що представляє ім'я категорії.

– *CategoryType*: Властивість, що вказує на тип категорії (дохід чи витрати).

CategoryType enum:

– *Income* та *Expense*: Перелік типів категорій.

Опис класу:

– *Account*: Властивість, що вказує на обліковий запис, пов'язаний з транзакцією (використовується клас *Account*).

– *Amount*: Властивість, що вказує на суму транзакції.

– *Date*: Властивість, яка вказує на дату транзакції.

– *Comment*: Властивість, що містить коментар (може бути null).

– *Category*: Властивість, яка вказує на категорію транзакції (використовується клас *Category*).

Наостанок необхідно додати модель для бюджетування. Вона складається з двох класів, один з яких має в собі список з об’єктами іншого класу. Приклад класів для бюджетування зображені на рис. 3.6.

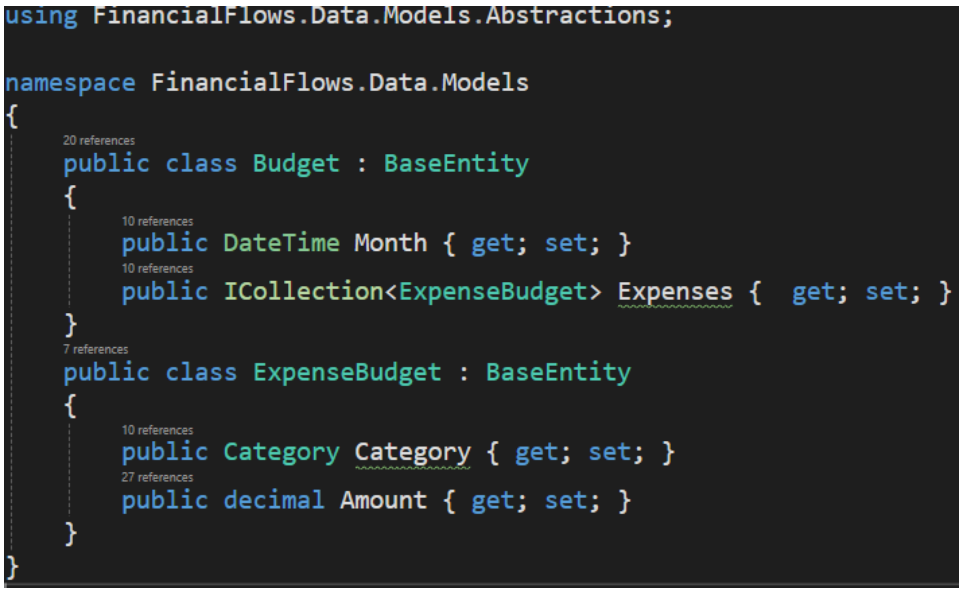

Рис.3.6. Класи для бюджетування.

Опис класів:

Budget class:

– *Month*: Властивість, яка вказує на місяць бюджету.

– *Expenses*: Колекція видатків (використовується клас *ExpenseBudget*).

ExpenseBudget class:

– *Category*: Властивість, яка вказує на категорію витрат (використовується клас *Category*).

– *Amount*: Властивість, що вказує на суму видатків у даній категорії.

Наступним кроком необхідно створити клас який визначає контекст бази даних *Entity Framework Core* для системи фінансових потоків. Цей клас дозволяє взаємодіяти з базою даних *SQLite* для збереження та витягування об'єктів фінансових потоків. *DbSet* визначає таблиці бази даних, а *OnConfiguring* налаштовує параметри підключення до бази даних *SQLite*. Приклад такого класу наведений на рис. 3.7

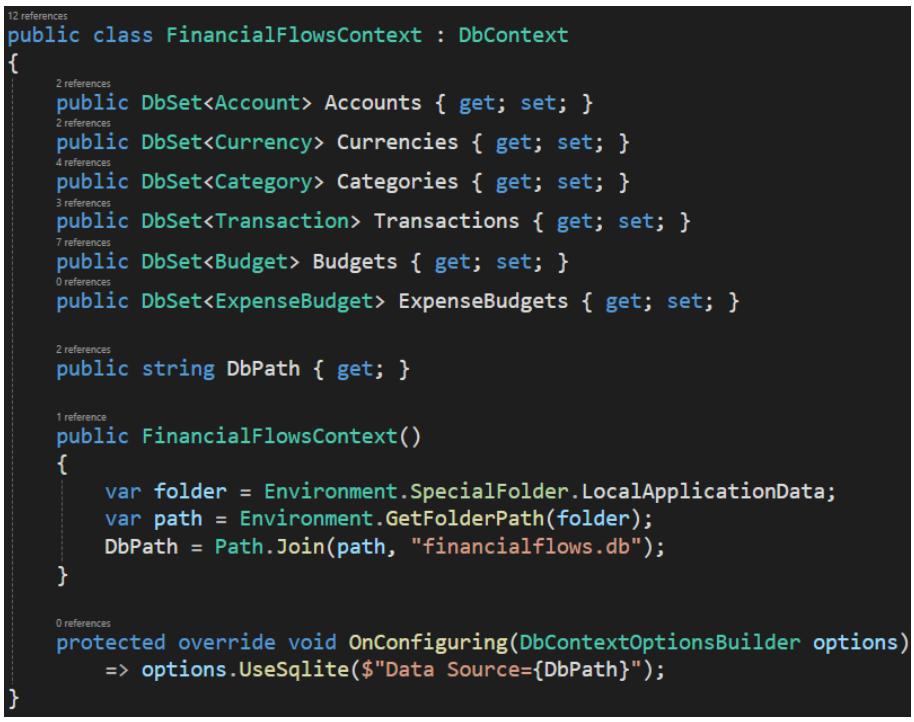

Рис.3.7. Клас з контекстом бази даних

Цей клас *FinancialFlowsContext* є підкласом *DbContext* з *Entity Framework*, і він визначає структуру та конфігурацію бази даних для збереження фінансових даних. Структура класу:

Accounts, *Currencies*, *Categories*, *Transactions*, *Budgets*, та *ExpenseBudgets* – це властивості типу *DbSet<>*, які представляють таблиці бази даних для об'єктів типів *Account*, *Currency*, *Category*, *Transaction*, *Budget*, та *ExpenseBudget* відповідно.

DbPath – це властивість, яка містить шлях до файлу бази даних. Використовується для забезпечення локального збереження бази даних у спеціальній папці користувача (*LocalApplicationData*).

Конструктор встановлює значення для *DbPath*, обираючи шлях до папки локальних даних користувача та об'єднуючи його з ім'ям файлу бази даних ("*financialflows.db*").

Перевизначений метод *OnConfiguring* встановлює опції конфігурації для контексту бази даних. У цьому випадку, використовується *SQLite* база даних, і з'єднання вказується через рядок підключення з використанням *UseSqlite*.

Цей клас слугує точкою входу для здійснення операцій з базою даних, таких як зчитування, запис та оновлення фінансових об'єктів. Він також визначає, як база даних буде налаштована та де знаходитиметься файл бази даних на диску.

Таким чином, рівень доступу до даних майже готовий залишається лише запустити декілька команд щоб створити міграцію та саму базу даних. Спочатку треба створити початкову міграцію за допомогою наступної команди:

dotnet ef migration add InitialCreate

Міграції в *Entity Framework Core* – це механізм, який надає автоматизований спосіб зміни схеми бази даних під час розвитку проекту. Міграції дозволяють вам визначати, яким чином схема бази даних має змінитися відносно поточного стану моделі даних.

Основні концепції пов'язані з міграціями:

1. Модель даних: Це ваша об'єктна модель даних, яку ви визначаєте у своєму коді. Міграції використовують цю модель для порівняння з поточним станом бази даних.

2. Схема бази даних: Це фактична структура бази даних, яка включає таблиці, зв'язки, індекси тощо.

3. Міграційні файли: Це файли, що створюються і керуються *Entity Framework Core*. Кожен файл міграції містить операції для зміни бази даних відносно попереднього стану.

4. Додавання змін: Додавання нових об'єктів (таблиць, стовпців тощо) до схеми бази даних.

5. Видалення змін: Видалення об'єктів з схеми бази даних.

6. Оновлення змін: Модифікація об'єктів в схемі бази даних.

Після створення або зміни моделі даних вам слід створити нову міграцію. Міграція створюється на основі порівняння поточної моделі та схеми бази даних. Після цього ви можете застосувати міграцію, щоб оновити базу даних.

Використання міграцій полегшує розвиток проекту, оскільки вони дозволяють вам ефективно зберігати та відстежувати зміни в схемі бази даних разом із змінами у вашому коді.

Після цього можна створити або оновити базу даних за допомогою команди наведеної нижче:

dotnet ef database update

Після цієї команди буде створена або оновлена база даних. Приклад бази даних яка буде створена з переліченими класами наведена на рис. 3.8.

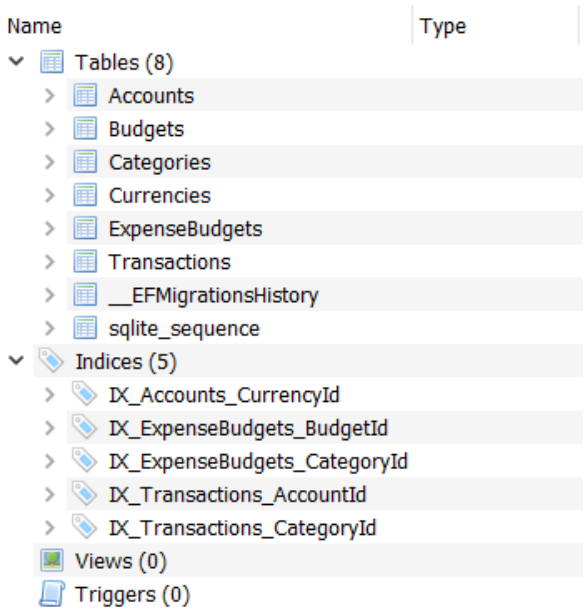

Рис.3.8. Приклад згенерованої бази даних

3.2. Розробка бізнес логіки

Бізнес–логіка для данного фінансового додатку має на меті взаємодію з фінансовими даними користувачів та забезпечення ефективного управління фінансами. Основні функції та принципи бізнес–логіки можуть виглядати наступним чином:

Можливість користувачам додавати нові облікові записи

– Розроблення функціоналу, який надає користувачам зручний інтерфейс для створення нових облікових записів.

– Форма або інтерактивний елемент, який дозволяє введення необхідної інформації для нового облікового запису.

Встановлення початкового балансу та валюти для облікового запису:

– Включення полів або параметрів для введення початкового балансу та валюти при створенні нового облікового запису.

– Можливість вибору валюти з вже наявних у системі або додавання нової валюти.

Можливість користувачам видаляти облікові записи:

– Додавання функціоналу для видалення облікового запису з бази даних.

– Перед видаленням може здійснюватися підтвердження або виведення попередження для уникнення випадкового видалення.

Можливість користувачам оновлювати інформацію про обліковий запис:

– Наявність інтерфейсу для редагування даних облікового запису, таких як назва та валюта.

– Відображення поточних значень облікового запису у формі для зручного оновлення.

Ці функціональні можливості дозволяють користувачам ефективно взаємодіяти з обліковими записами, додавати нові, оновлювати існуючі та видаляти ті, які більше не потрібні. Організація інтерфейсу повинна бути зрозумілою та зручною для користувача, а функціонал повинен забезпечувати правильне взаємодію з базою даних та зберігати консистентність фінансової інформації. Приклад бізнес–логіки для керування обліковими записами зображений на рис. 3.9.

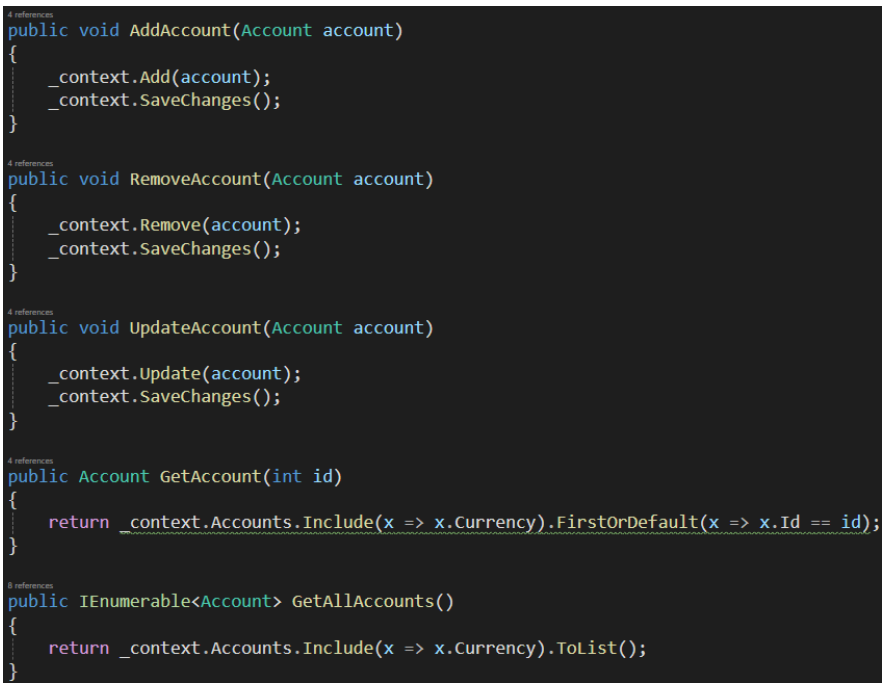

Рис.3.9. Контролер для керування обліковими записами

Автоматичне створення бюджетів для поточного та наступного місяців:

– В реалізації вбудованого механізму для автоматичного створення бюджетів для поточного та наступного місяців в разі їх відсутності.

– Врахування інтерфейсу для введення користувачем додаткових параметрів, які можуть впливати на автоматичне створення бюджету.

Забезпечення користувачів можливістю визначати обсяг бюджету для кожної категорії витрат:

– Розроблення інтерактивного інтерфейсу для введення користувачем обсягу бюджету для окремих категорій витрат.

– Збереження цієї інформації в базі даних для подальшого використання та відстеження.

Забезпечення можливості користувачам змінювати обсяг бюджету для кожної категорії витрат:

– Розроблення інтерфейсу для оновлення обсягу бюджету для конкретної категорії витрат.

– Забезпечення коректного відображення та підтримки змін в базі даних після оновлення.

Ця система керування бюджетами має на меті надати користувачам зручний та ефективний інструмент для визначення, створення, редагування та видалення бюджетів. Автоматизація процесів, які пов'язані з автоматичним створенням та видаленням бюджетів, спрощує взаємодію користувача з системою, а можливості налаштування обсягу бюджету для кожної категорії дозволяють гнучко адаптувати систему до індивідуальних фінансових потреб користувача. Приклад бізнес–логіки для автоматичного створення бюджетів зображений на рис. 3.10.

```
2 references
private void EnsureBudgetForMonth(DateTime date)
{
    var existingBudget = _context.Budgets
        .Where(x => x.Month.Month == date.Month && x.Month.Year == date.Year)
        .FirstOrDefault();

    if (existingBudget == null)
    {
        var categories = _context.Categories
            .Where(x => x.CategoryType == CategoryType.Expense)
            .ToList();

        var expenseBudgets = categories.Select(category =>
            new ExpenseBudget { Amount = 0m, Category = category }
        ).ToList();

        var budget = new Budget
        {
            Month = date,
            Expenses = expenseBudgets
        };

        _context.Budgets.Add(budget);
        _context.SaveChanges();
    }
}
```

Рис.3.10. Автоматичне створення бюджетів

Забезпечення можливості користувачам додавати нові категорії:

– Розроблення інтерфейсу або форми, що дозволяє користувачам вводити необхідну інформацію для створення нової категорії витрат.

–Включення необхідних полів, таких як назва категорії, тип (*Income* або *Expense*) тощо.

Автоматичне оновлення всіх існуючих бюджетів для включення нової категорії:

– Після додавання нової категорії забезпечення механізму, що автоматично оновлює всі існуючі бюджети, включаючи нову категорію.

– Зміни в бюджетах повинні відображати додану категорію та розподіл коштів між існуючими категоріями.

Можливість видалення категорії, якщо вона більше не використовується:

– Розроблення інтерфейсу або механізму для видалення категорії.

– Після видалення категорії забезпечення оновлення всіх бюджетів для відображення змін в розподілі коштів.

Забезпечення можливості користувачам змінювати інформацію про категорію:

– Наявність інтерфейсу для редагування даних про категорію, таких як назва чи тип.

– Після змін в інформації про категорію, забезпечення оновлення всіх бюджетів для відображення змін.

– Врахування можливості зміни типу категорії (Income або Expense) та відображення цих змін у всіх відповідних частинах системи.

Ці функціональності дозволяють користувачам ефективно управляти категоріями витрат, додавати нові, видаляти ті, які більше не потрібні, та оновлювати інформацію для існуючих категорій. Автоматичне оновлення бюджетів забезпечує консистентність інформації та допомагає уникнути помилок при розподілі коштів між різними категоріями. Приклад бізнес–логіки для додавання категорій разом з оновленням бюджетів зображений на рис. 3.11.

```
4 references
public void AddCategory(Category category)
{
    var budgets = _context.Budgets.Include(x => x.Expenses).ThenInclude(x => x.Category).ToList();
    foreach (var budget in budgets)
    {
        budget.Expenses.Add(new ExpenseBudget() { Category = category, Amount = 0m });
    }
    _context.Add(category);
    _context.SaveChanges();
}
```

Рис.3.11. Метод для додавання категорій

Можливість користувачам додавати нові валюти для розрахунків:

– Створення інтерфейсу або форми, що дозволяє користувачам вводити інформацію для додавання нової валюти.

– Включення необхідних полів, таких як назва валюти, скорочення, обмінний курс тощо.

–Забезпечення унікальності ідентифікаторів для кожної валюти.

Можливість видалення валюти, якщо вона більше не потрібна:

– Розроблення механізму або інтерфейсу для видалення валюти з системи.

Забезпечення можливості користувачам змінювати інформацію про валюту:

– Введення можливості редагування інформації про валюту, такої як обмінний курс, назва чи скорочення.

– Автоматичне оновлення всіх транзакцій та облікових записів, які використовують дану валюту, для відображення змін.

Ці функціональності дозволяють користувачам додавати, видаляти та оновлювати інформацію про валюти в системі. Можливість внесення змін у валюту та її параметри допомагає користувачам підтримувати актуальну інформацію для фінансових розрахунків. Автоматичне оновлення транзакцій та облікових записів, пов'язаних з валютою, сприяє уникненню розбіжностей у фінансовій інформації. Приклад бізнес–логіки для керування валютами зображений на рис. 3.12.

```
4 references
public void AddCurrency(Currency currency)
{
    _context.Add(currency);
    _context.SaveChanges();
}

4 references
public void RemoveCurrency(Currency currency)
{
    _context.Remove(currency);
    _context.SaveChanges();
}

4 references
public void UpdateCurrency(Currency currency)
{
    _context.Update(currency);
    _context.SaveChanges();
}

4 references
public Currency GetCurrency(int currencyId)
{
    return _context.Currencies.Find(currencyId);
}

8 references
public IEnumerable<Currency> GetAllCurrencies()
{
    return _context.Currencies.ToList();
}
```

Рис.3.12. Контроллер для керуванням валютами

Забезпечення можливості користувачам додавати нові транзакції для обліку доходів та витрат:

– Створення інтерфейсу для введення деталей транзакції, таких як сума, категорія, коментар, дата тощо.

– Врахування можливості обратного перегляду історії транзакцій.

Автоматичне оновлення балансу облікового запису та відповідного бюджету:

– Автоматичне визначення впливу транзакції на баланс облікового запису (зменшення або збільшення).

– Інтеграція з бюджетами для автоматичного відстеження витрат та доходів у відповідних категоріях.

Можливість видалення транзакції та відновлення попереднього балансу облікового запису та бюджету:

– Розроблення інтерфейсу для видалення транзакції та відновлення попереднього стану фінансів.

– Забезпечення логіки, яка автоматично коригує баланс облікового запису та бюджету після видалення.

Забезпечення можливості користувачам змінювати інформацію про транзакцію, враховуючи зміни в балансі облікового запису та бюджеті:

– Розроблення інтерфейсу для редагування деталей транзакції.

– Автоматичне коригування балансу облікового запису та бюджету при змінах.

Ці функціональності дозволяють користувачам легко та ефективно вести облік своїх фінансів, реєструючи та керуючи транзакціями. Автоматичне оновлення балансу та інтеграція з бюджетами забезпечують точність та зручність в управлінні фінансами. Приклад бізнес–логіки для автоматичного коригування балансу облікового запису зображений на рис. 3.13.

```
2 references
private void UpdateAccountBalanceOnAdd(Transaction transaction)
{
    var account = transaction.Account;
    var amount = GetTransactionAmount(transaction);

    account.CurrentBalance += amount;
    _context.Update(account);
}
2 references
private void UpdateAccountBalanceOnRemove(Transaction transaction)
{
    var account = transaction.Account;
    var amount = GetTransactionAmount(transaction);

    account.CurrentBalance -= amount;
    _context.Update(account);
}
2 references
private decimal GetTransactionAmount(Transaction transaction)
{
    return transaction.Category.CategoryType == CategoryType.Expense
        ? transaction.Amount * -1m
        : transaction.Amount;
}
```

Рис.3.13. Автоматичне коригування балансів

Всі вищезазначені компоненти є складовою проекту *Core*, що визначає бізнес–логіку системи. Однак, для полноцінної реалізації функціоналу та взаємодії з користувачем, також необхідно додати частину з вищезазначеної логіки в проект *Client*. В проекті *Client* буде розміщено інтерфейс користувача, реалізацію взаємодії з сервером, а також логіку відображення та обробки даних. Це дозволить забезпечити повноцінний та зручний інтерфейс для користувачів, а також взаємодію з серверною частиною системи. Зміст файлу *MauiProgram.cs* зображений на рис. 3.14.

```
public static class MauiProgram
{
    4 references
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            });
        builder.Services.AddMauiBlazorWebView();
        var db = new FinancialFlowsContext();
        db.Database.EnsureCreated();
        builder.Services.AddSingleton(new AccountController(db));
        builder.Services.AddSingleton(new BudgetController(db));
        builder.Services.AddSingleton(new CurrencyController(db));
        builder.Services.AddSingleton(new CategoryController(db));
        builder.Services.AddSingleton(new TransactionController(db));
#if DEBUG
        builder.Services.AddBlazorWebViewDeveloperTools();
        builder.Logging.AddDebug();
#endif
        return builder.Build();
    }
}
```

Рис.3.14. Код файлу *MauiProgram.cs*

В функції *CreateMauiApp* у файлі *MauiProgram.cs* здійснюється налаштування основних компонентів Maui додатка. Для роботи з базою даних використовується об'єкт *FinancialFlowsContext*, який ініціалізується та перевіряється на наявність структур бази даних.

Для взаємодії із контролерами системи фінансового обліку – *AccountController*, *BudgetController*, *CurrencyController*, *CategoryController* та *TransactionController* – вони реєструються в службовій колекції *MauiApp*, щоб бути доступними для використання в усьому додатку.

Важливо відзначити, що такий підхід до реєстрації служб використовує паттерн *Dependency Injection*, що спрощує управління залежностями в програмі та забезпечує легку розширюваність коду. Контролери вбудовують у себе логіку обробки даних і взаємодії з базою даних, тим самим дозволяючи здійснювати різноманітні операції, такі як додавання, оновлення та видалення облікових записів, бюджетів, категорій, валют та транзакцій.

Розгалуження "*#if DEBUG*" дозволяє додавати інструменти розробки у відлагоджувальному режимі, що полегшує вивчення та виправлення помилок. Такий підхід є стандартною практикою в розробці для забезпечення зручності відладки.

Усі ці компоненти інтегруються в *Maui* додаток, який створюється та будується за допомогою створеного білдера, що дозволяє отримати готовий до запуску *Maui* додаток з налаштованими сервісами та залежностями.

Клієнтська логіка, яка визначає основні операції над об'єктами, часто реалізується в стилі *CRUD*, що складається з чотирьох основних операцій над об'єктом: створення (*Create*), читання (*Read*), оновлення (*Update*) та видалення (*Delete*). Розглянемо кожну з цих операцій більш детально:

Create (Створення):

– Для здійснення створення нового об'єкта зазвичай реалізується окрема сторінка або форма.

– Користувач вводить необхідні дані, такі як ім'я, параметри чи інші властивості.

– За допомогою цієї форми створюється новий об'єкт, який додається до системи.

Read (Читання):

– Читання даних здійснюється через сторінки чи інші інтерфейсні елементи, що відображають інформацію про існуючі об'єкти.

– Інформація може бути відображена у вигляді списків, таблиць чи інших елементів інтерфейсу.

Update (Оновлення):

– Сторінки для оновлення дозволяють користувачам вносити зміни в існуючі об'єкти.

– Вони можуть містити форми, подібні до тих, що використовуються для створення, але заповнені ініціальними даними об'єкта, який слід оновити.

– Зміни зберігаються у системі після підтвердження користувачем.

Delete (Видалення):

– Операція видалення дозволяє користувачам видаляти об'єкти, які вже існують у системі.

– Зазвичай це реалізується за допомогою окремих сторінок або підтверджувальних діалогів, щоб уникнути випадкового видалення.

З метою зручності користувачів створюються окремі сторінки для кожного типу об'єкту, а також окремі сторінки для виконання операцій модифікації та додавання нових об'єктів. Це дозволяє ефективно взаємодіяти з системою та забезпечує інтуїтивно зрозумілий інтерфейс для користувачів.

Беручи до уваги моделі, представлені в даному проекті, можна розглядати створення 5 сторінок для кожного з контролерів з метою надання користувачеві повного доступу до записів бази даних в конкретних таблицях і можливості видалення конкретних записів. Також необхідно розробити сторінки для додавання та модифікації існуючих записів для кожного контролера. Структура сторінок може бути організована наступним чином:

Страниці для облікових записів (*/accounts*):

–*/accounts*: Сторінка, що відображає всі записи облікових записів з бази даних.

– */accounts/add*: Сторінка для додавання нового облікового запису.

– */accounts/edit/{id}*: Сторінка для редагування існуючого облікового запису за конкретним ідентифікатором.

Страниці для бюджетів (*/budgets*):

– */budgets*: Сторінка, що відображає всі записи бюджетів з бази даних.

– */budgets/edit/{id}*: Сторінка для редагування існуючого бюджету за конкретним ідентифікатором.

Страниці для категорій (*/categories*):

– */categories*: Сторінка, що відображає всі записи категорій з бази даних.

– */categories/add*: Сторінка для додавання нової категорії.

– */categories/edit/{id}*: Сторінка для редагування існуючої категорії за конкретним ідентифікатором.

Страниці для валют (*/currencies*):

– */currencies*: Сторінка, що відображає всі записи валют з бази даних.

– */currencies/add*: Сторінка для додавання нової валюти.

– */currencies/edit/{id}*: Сторінка для редагування існуючої валюти за конкретним ідентифікатором.

Страниці для транзакцій (*/transactions*):

– */transactions*: Сторінка, що відображає всі записи транзакцій з бази даних.

– */transactions/add*: Сторінка для додавання нової транзакції.

– */transactions/edit/{id}*: Сторінка для редагування існуючої транзакції за конкретним ідентифікатором.

Така структура сторінок забезпечить зручний і логічний інтерфейс для користувачів і дозволить їм легко взаємодіяти з різними частинами системи.

Створені сторінки для додавання та модифікації можуть бути об'єднані за принципом, що визначає тип операції в залежності від наявності переданого ідентифікатора. Якщо ідентифікатор не передано, це вказує на операцію додавання нової транзакції. У випадку, коли ідентифікатор присутній, здійснюється витягування об'єкта з бази даних для подальшої модифікації. Принцип роботи цієї логіки ілюстрований на рис. 3.15 та рис. 3.16.

```
@page "/accounts/add"
@page "/accounts/edit/{accountID:int}"
@using FinancialFlows.Core.Controllers
@inject NavigationManager Navigation
@inject CurrencyController currencyController
@inject AccountController accountController
```

Рис.3.15. Використання однієї сторінки для двух маршрутів

```
[Parameter]
public int accountID { get; set; }
protected FinancialFlows.Data.Models.Account account = new();
protected override async Task OnParametersSetAsync()
{
    if (accountID != 0)
    {
        account = accountController.GetAccount(accountID);
    }
}
protected async Task SaveAccount()
{
    if (account.Id != 0)
    {
        accountController.UpdateAccount(account);
    }
    else
    {
        accountController.AddAccount(account);
    }
    Cancel();
}
```

Рис.3.16. Використання різних операцій залежно від облікового запису

Додавання облікового запису *(/accounts/add)*:

– Якщо ідентифікатор не передано, користувачу доступна сторінка для додавання нового облікового запису.

Модифікація облікового запису *(/accounts/edit/{id})*:

– Якщо передано ідентифікатор, система витягує об'єкт облікового запису з бази даних за цим ідентифікатором.

– Витягнутий об'єкт використовується для заповнення полів на сторінці для подальшої модифікації.

Такий підхід до об'єднання сторінок для додавання та модифікації дозволяє системі динамічно визначати тип операції на основі введених користувачем даних та оптимізує взаємодію з інтерфейсом.

Принцип, що описаний вище, може бути застосований до всіх типів об'єктів, де присутні операції додавання та модифікації. Цей підхід до створення сторінок

спрощує розробку та управління інтерфейсом, роблячи його більш гнучким і зручним для користувача.

Для типів об'єктів, які мають залежності від інших (наприклад, обліковий запис, який посилається на валюту), необхідно додатково реалізувати можливість вибору залежних об'єктів. На прикладі облікового запису, який має залежність від валюти, важливо створити механізм вибору валют. Процес вибору валюти зображений на рис. 3.17.

```
protected FinancialFlows.Data.Models.Currency[] currencies;
protected override Task OnInitializedAsync()
{
    currencies = currencyController.GetAllCurrencies().ToArray();
    return base.OnInitializedAsync();
}
private void SelectCurrency(ChangeEventArgs e) {
    account.Currency = currencies.Where(ao => ao.Id.ToString() == e.Value.ToString()).First();
}
public void Cancel()
{
    Navigation.NavigateTo("/accounts");
}
```

Рис.3.17. Процес отримання списку валют для вибору.

Вибір облікового запису для модифікації *(/accounts/edit/{id}*):

– При виборі опції модифікації облікового запису, система витягує об'єкт облікового запису з бази даних.

Вибір валюти для облікового запису:

– Створення можливості вибору валюти, на яку посилається обліковий запис.

– Відображення доступних варіантів валют для вибору користувачем.

Збереження вибраної валюти:

–Збереження вибраної користувачем валюти для облікового запису.

Цей підхід дозволяє забезпечити користувача зручним та інтуїтивно зрозумілим інтерфейсом для вибору залежних об'єктів та їхньої модифікації.

Після успішного збереження, модифікації або відміни дії, користувача повинно перенаправляти на основну сторінку для відповідного контролера. Ця основна сторінка повинна надавати користувачеві повний перегляд всіх записів в конкретній таблиці, а також має вбудовані можливості для додавання нових записів та модифікації існуючих. Приклад логіки на такій сторінці зображено на рис. 3.18.

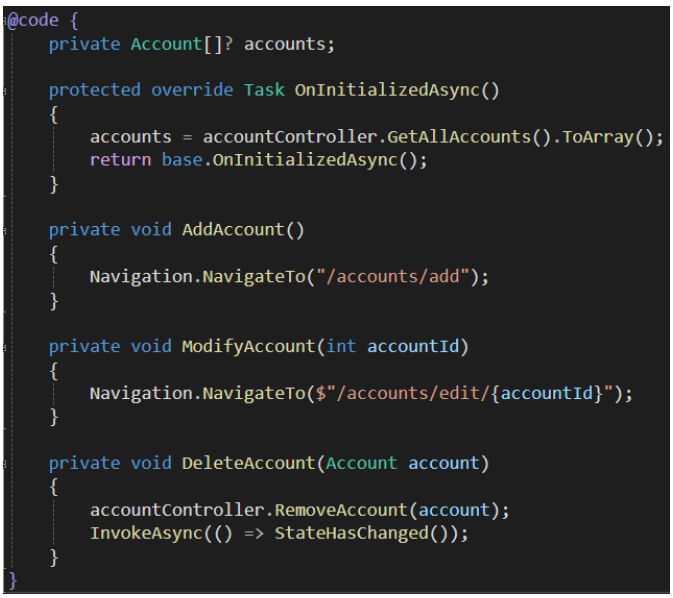

Рис.3.18. Логіка головної сторінки для облікових записів

3.3. Розробка графічного інтерфейсу користувача

Розробка графічного інтерфейсу користувача (*GUI*) є невід'ємною та критичною частиною процесу створення сучасних програм та додатків. Цей аспект визначає спосіб, яким користувач взаємодіє з програмою, маючи на меті забезпечити йому зручність, інтуїтивність та ефективність в користуванні. У даному випадку використання *Bootstrap* спрощує інтеграцію зовнішніх елементів дизайну, забезпечуючи легкість та швидкість розробки.

Bootstrap дозволяє легко використовувати наявні картинки та стилі за допомогою атрибутів class, що спрощує вбудовування графічних елементів на сторінках. Це надає можливість створювати естетично приємний та сучасний вигляд інтерфейсу.

Основна структура інтерфейсу включає п'ять сторінок, що дозволяє користувачам взаємодіяти з різними функціональними частинами системи. Для полегшення навігації та розпізнавання кожної сторінки, до навігаційного меню додані індивідуальні картинки, які відображають тематику кожної сторінки. Наприклад, для сторінки, що відображає валюту, можна обрати найпоширенішу валюту, таку як долар. Це не лише додає естетику до інтерфейсу, але й полегшує

розпізнавання користувачем різних розділів системи. Приклад навігаційного меню зображено на рис. 3.19.

```
<div class="nav-item px-3">
    <NavLink class="nav-link" href="transactions" Match="NavLinkMatch.All">
        <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16" fill=
            <path d="M8 10a2 2 0 1 0 0-4 2 2 0 0 0 0 4" />
            <path d="M0 4a1 1 0 0 1 1-1h14a1 1 0 0 1 1 1v8a1 1 0 0 1-1 1H1a1
        </svg> Transactions
    </NavLink>
</div>
<div class="nav-item px-3">
    <NavLink class="nav-link" href="accounts">
        <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16" fill=
            <path d="M12.136.326A1.5 1.5 0 0 1 14 1.78V3h.5A1.5 1.5 0 0 1 16
        </svg> Accounts
    </NavLink>
```

Рис.3.19. Навігаційне меню

Отже, процес навігації в інтерфейсі користувача здійснюється за допомогою елементів меню, які дозволяють користувачеві швидко та ефективно переходити між різними розділами системи. При натисканні на конкретний пункт меню користувач повинен автоматично перенаправлятися на відповідну сторінку, що забезпечує зручний доступ до функціональності, яку він бажає використовувати.

Головні сторінки для кожного контролера мають відображати усі записи, пов'язані з відповідною тематикою. Це дозволяє користувачеві переглядати та управляти інформацією шляхом відображення всіх доступних даних на одній сторінці. Крім того, цей підхід спрощує інтерфейс для користувача, оскільки всі головні сторінки контролерів мають аналогічну структуру та вигляд, що сприяє їхній однаковості та інтуїтивному користуванню. Структура головної сторінки на прикладі сторінки для облікових записів зображена на рис. 3.20.

```
<button class="btn btn-success" @onclick="() => AddAccount()">Add account</button>
<table class="table">
    <thead>
        <tr>
            <th>Name</th>
            <th>Balance</th>
            <th>Currency</th>
            <th>Action</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var i in accounts)
        {
            <tr>
                <td>@i.AccountName</td>
                <td>@i.CurrentBalance</td>
                <td>@i.Currency?.CurrencyName</td>
                <td>
                    <button class="btn btn-primary" @onclick="() => ModifyAccount(i.Id)">Modify</button>
                    <button class="btn btn-danger" @onclick="() => DeleteAccount(i)">Delete</button>
                </td>
            </tr>
        }
    </tbody>
</table>
```

Рис.3.20. Головна сторінка для облікових записів

Після після того, як користувач вибрав опцію для модифікації існуючого запису або додавання нового, його автоматично перенаправить на відповідну сторінку, спеціально призначену для цього типу операцій. Структура цих сторінок аналогічна, що дозволяє користувачу швидко освоїти інтерфейс та легко розуміти, як використовувати їхні можливості.

Наприклад, розглянемо сторінку для додавання та модифікації облікових записів. Ця сторінка надає користувачеві можливість вводити необхідні дані та параметри для нового облікового запису або вносити зміни в існуючий. Компоненти інтерфейсу легко розрізнити та зрозуміти, а керування опціями введення дозволяє ефективно взаємодіяти з системою. Приклад такої сторінки представлено на рис. 3.21.

```
<EditForm Model="@account" OnValidSubmit="SaveAccount">
    <DataAnnotationsValidator />
    <div class="mb-3">
        <label for="AccountName" class="form-label">Name</label>
        <div class="col-md-4">
            <InputText class="form-control" @bind-Value="account.AccountName" />
        </div>
        <ValidationMessage For="@(() => account.AccountName)" />
    </div>
    <div class="mb-3">
        <label for="CurrentBalance" class="form-label">Balance</label>
        <div class="col-md-4">
            <InputNumber class="form-control" @bind-Value="account.CurrentBalance" />
        </div>
        <ValidationMessage For="@(() => account.CurrentBalance)" />
    </div>
    <div class="mb-3">
        <label for="Currency" class="form-label">Currency</label>
        <div class="col-md-4">
            <select class="form-control" @onchange="SelectCurrency">
                <option selected hidden>Select currency</option>
                @foreach (var currency in currencies)
                {
                    <option value="@currency.Id">@currency.CurrencyName</Option>
                }
            </select>
        </div>
        <ValidationMessage For="@(() => account.Currency)" />
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-primary">Save</button>
        <button class="btn btn-light" @onclick="Cancel">Cancel</button>
    </div>
</EditForm>
```

Рис.3.21. Сторінка для додавання та модифікації облікових записів

Додавши Розробивши всі необхідні сторінки для функціональності додатка, можна перейти до процесу компіляції для різних платформ, таких як *Windows*, *Android*, *iOS* та *Tizen*. Цей процес здійснюється автоматично платформою *Maui*, не вимагаючи від розробника додаткових зусиль. Таким чином, створивши лише один

додаток, можна отримати готові застосунки, оптимізовані для використання на конкретних платформах.

Приклади графічного інтерфейсу для додатка, адаптованого під платформи *Windows* та *Android*, наведено на рис. 3.22 та рис. 3.23 відповідно. Зображені екрани відображають інтерфейс додатка, що відповідає стандартам та вимогам для відповідних операційних систем. Такий уніфікований підхід спрощує розгортання та експлуатацію додатка на різних пристроях та платформах.

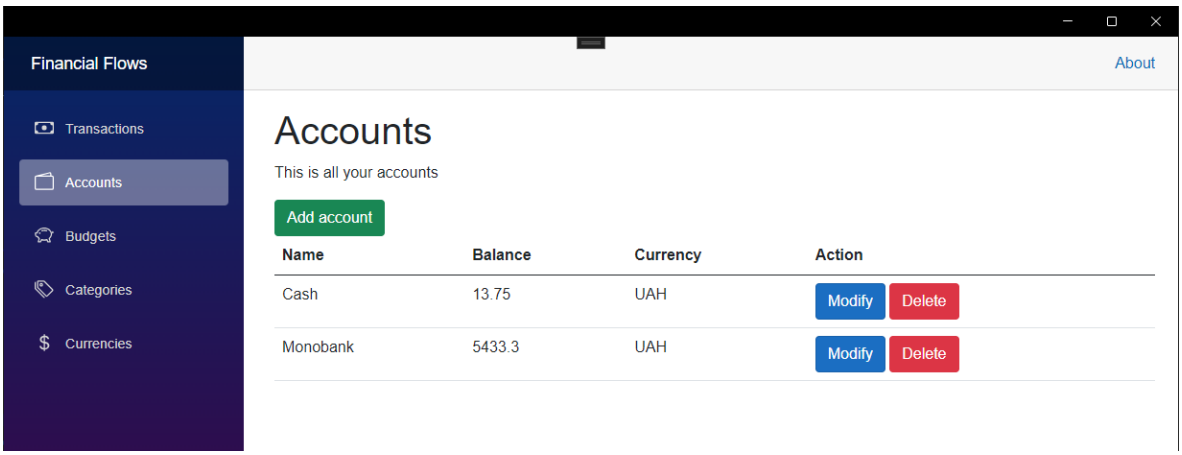

Рис.3.22. Графічний інтерфейс додатку під платформу *Windows*

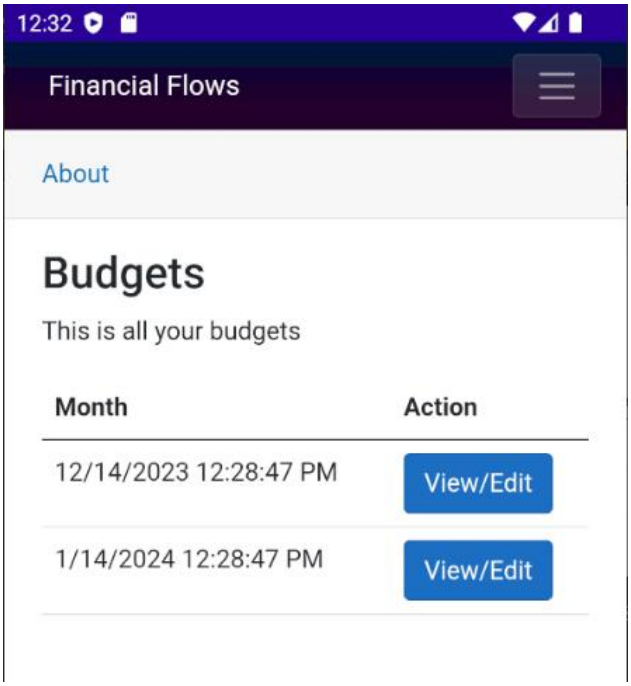

Рис.3.23. Графічний інтерфейс додатку під платформу *Android*

Висновки за розділом

У даному розділі розглядався процес розробки системи, який включав три ключові аспекти: розробку структур даних, бізнес–логіки та графічного інтерфейсу користувача.

Було детально розглянуто створення необхідних базових сутностей, таких як транзакції, облікові записи, бюджети, категорії та валюти. Проектування бази даних та вибір відповідних типів даних дало змогу створити ефективну та оптимізовану структуру для зберігання інформації.

Після створення сутностей була описана реалізація функціоналу для керування транзакціями, обліковими записами, бюджетами, категоріями та валютами. Використання контролерів дозволило створити логіку, яка забезпечує коректну роботу системи з фінансовими даними, враховуючи різні сценарії взаємодії.

В кінці розділу було висвітлено процес створення графічного інтерфейсу за допомогою *Maui* та *Bootstrap*. Було наведено приклади основних сторінок, таких як сторінки для перегляду, додавання та модифікації записів, а також навігаційного меню. Використання уніфікованого інтерфейсу для різних платформ спрощує процес розгортання та використання додатка на різних пристроях.

У цілому, було систематизувано та висвітлено ключові етапи розробки фінансового додатка, що дозволяє користувачам ефективно вести облік своїх фінансів та планувати витрати.

РОЗДІЛ 4
ТЕСТУВАННЯ СИСТЕМИ ТА ПРИКЛАД ВИКОРИСТАННЯ

4.1. Модульне тестування програмного коду

Модульне тестування програмного коду — це практика в програмуванні, яка передбачає використання тестових випадків для перевірки окремих модулів або компонентів програмного коду з метою визначення їхньої правильності та коректності роботи. Основна ідея полягає в тому, щоб перевірити, чи веде кожен окремий компонент коду (найчастіше, це може бути функція, метод, клас або модуль) до очікуваного результату.

Основні аспекти модульного тестування:

1. Поділ на модулі:

– Модуль: Невеликий ізольований компонент програмного коду, що має конкретну функціональність.

– Тестовий випадок: Набір вхідних даних, параметрів та очікуваних результатів для перевірки правильності роботи модуля.

2. Створення тестових випадків:

– Розроблення набору тестових випадків для охоплення різних сценаріїв використання модуля.

– Включення випадків для обробки коректних та виключних ситуацій.

3. Автоматизація тестів:

– Використання автоматизаційних інструментів для автоматичного виконання тестів.

– Забезпечення швидкого та ефективного виконання тестів під час розробки.

4. Залежності та ізоляція:

– Тести повинні бути незалежними та ізольованими від інших тестів та компонентів.

– Забезпечення стабільності коду при внесенні змін.

5. Пошук та виправлення помилок:

– Використання тестів для виявлення та виправлення помилок та дефектів у коді.

6. Підтримка розширення:

– Забезпечення можливості додавання та редагування тестів при змінах у коді.

7. Інтеграція в CI/CD:

– Включення модульних тестів до процесу неперервної інтеграції та розгортання для автоматичного виконання під час розробки.

Приклад тестування логіки додатку та запуск усіх тестів на компонент наведено в рис. 4.1 та рис. 4.2.

```
[Fact]
0 references
public void AddTransactionTest_WhenAmountIsNotZero_UpdateUnderlyingAccount()
{
    var dbContext = new FinancialFlowsContext();
    var transactionController = new TransactionController(dbContext);
    var currency = new Currency()
    {
        CurrencyName = "UAH"
    };
    var account = new Account()
    {
        AccountName = "Test",
        CurrentBalance = 2000m,
        Currency = currency
    };
    var transaction = new Transaction()
    {
        Account = account,
        Amount = 2000
    };
    transactionController.AddTransaction(transaction);
    Assert.Equal(0m, dbContext.Accounts.Find(account).CurrentBalance);
}
```

Рис.4.1. Модульний тест для метода *AddTransaction*

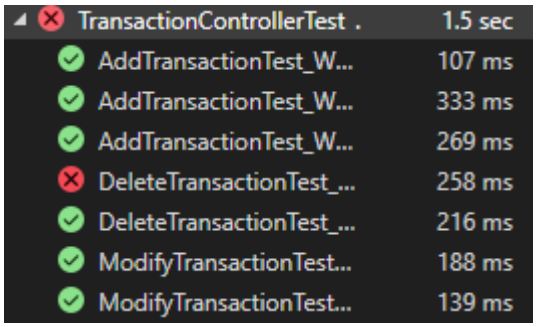

Рис.4.2. Тестування компоненту *TransactionController*

4.2. Ручне тестування

Ручне тестування — це процес перевірки функціональності, відповідності вимогам та коректності роботи програми чи системи за допомогою вручного введення вхідних даних та аналізу вихідних результатів. В даному методі тестування людина, яка називається тестувальником чи тестером, виконує роль кінцевого користувача та перевіряє, як система веде себе в реальних умовах.

Для ефективного тестування програми, важливо перевірити її реакцію на некоректні введені дані. Наприклад, можна провести тест, введення тексту в поле, яке очікує числове значення, або, навпаки, введення числа в поле, призначене для текстового вводу. Це допомагає визначити, чи програма адекватно обробляє такі ситуації та виводить відповідні повідомлення про помилки.

Особливу увагу слід звернути на вибір об'єктів з випадаючих списків. Тестування має включати перевірку, чи неможливо внести неіснуючий запис у випадаючий список. Також, у випадку відсутності залежних об'єктів, важливо перевірити, чи неможливо зберегти форму без вибору залежності, тобто чи коректно програма обробляє ситуації, коли деякі поля є обов'язковими для заповнення.

Приклад такого тестування можна зобразити на рис. 4.3, де демонструється введення некоректного значення та відповідна реакція програми на цю ситуацію. Такі тести допомагають виявити потенційні проблеми та покращити користувацький досвід взаємодії з програмою.

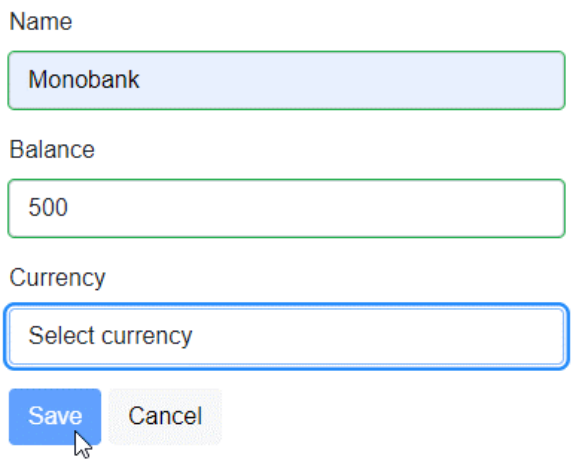

Рис.4.3. Кнопка *Save* не активна поки не заповнені всі необхідні поля

4.3. Приклад використання системи аналізу та прогнозування фінансових потоків

Приклад використання системи для аналізу та прогнозування фінансових потоків може бути надзвичайно корисним для приватних осіб у керуванні їх фінансами. Розглянемо сценарій використання системи для приватного користувача.

Сценарій використання: Особистий Бюджет та Фінансове Планування

1. Додавання фінансових об'єктів:

– Користувач створює облікові записи для різних фінансових об'єктів (банківські рахунки, інвестиції, готівка тощо).

– Встановлює початкові баланси та валюти для кожного облікового запису.

2. Створення бюджетів:

– Користувач визначає місячні бюджети для різних категорій витрат (продукти, житло, розваги).

– Автоматично створюються бюджети для поточного та наступного місяців.

3. Запис транзакцій:

– Користувач додає транзакції для відстеження доходів та витрат.

– Система автоматично відслідковує баланс облікових записів та оновлює відповідні бюджети.

4. Аналіз та звітність:

– Користувач використовує систему для аналізу своїх фінансів, перегляду звітів та графіків витрат і доходів.

– Отримує підсумкову інформацію про стан фінансів за певний період.

5. Прогнозування та оптимізація:

– Система, використовуючи історичні дані, допомагає користувачу прогнозувати майбутні фінансові стани.

6. Повідомлення та нагадування:

– Система надсилає повідомлення про наближення до ліміту бюджету, невідповідності витрат до плану та інші важливі події.

Цей сценарій дозволяє користувачеві ефективно керувати своїми фінансами, максимально використовуючи можливості системи для аналізу, прогнозування та оптимізації фінансових потоків.

Висновки за розділом

У цьому розділі кваліфікаційної роботи було проведено комплексне тестування розробленої системи для аналізу та прогнозування фінансових потоків. Процес тестування охопив кілька ключових аспектів, включаючи модульне тестування програмного коду, ручне тестування та приклад використання системи в реальних умовах.

Були створені модульні тести для різних контролерів системи. Цей етап тестування виявився надзвичайно важливим для виявлення та усунення помилок на ранніх етапах розробки. Запуск тестів призвів до виявлення деяких помилок, що були внесені в програмний код. Цей процес визначив важливі аспекти стабільності та надійності системи.

Описано процес ручного тестування, який допомагає перевірити коректність та зручність інтерфейсу системи, а також правильність виконання основних функцій. Ручне тестування включало в себе сценарії використання, перевірку відповідності функціональних вимог та забезпечення коректного взаємодії з користувачем.

Проілюстровано приклад використання розробленої системи для аналізу та прогнозування фінансових потоків. Цей етап дозволив вирішити, як система впроваджується в реальному бізнес–сценарії та як вона сприяє прийняттю обґрунтованих фінансових рішень.

Результати тестування дозволяють зробити висновок, що розроблена система ефективна та готова до використання в реальних умовах. Виявлені та усунені помилки під час модульного тестування свідчать про високу якість програмного

коду. Ручне тестування визначило, що інтерфейс користувача є зручним та інтуїтивно зрозумілим. Приклад використання демонструє, як система може впроваджуватися для покращення фінансового аналізу та прийняття стратегічних рішень.

ВИСНОВКИ

У рамках цієї кваліфікаційної роботи була проведена аналіз та розробка системи для проведення аналізу та прогнозування фінансових потоків. Виконаний ретельний аналіз використаних технологій та порівняння з кращими наявними альтернативами.

У першому розділі було представлено ключові інструменти та методи для аналізу та прогнозування особистих фінансових потоків. Дослідження включало аналіз фінансових звітів, витрат, бюджетування, відносин та розробку системи для ефективного управління фінансами.

Проведений аналіз фінансових звітів надав можливість глибше дослідити фінансовий стан, використовуючи систематизований підхід. Цей метод дозволяє визначити ключові фінансові показники, що становлять основу для подальших стратегічних рішень.

Структурований аналіз витрат допомагає ідентифікувати головні джерела витрат та розподіляти їх ефективно за різними категоріями. Це дозволяє користувачам свідомо контролювати свої витрати та уникати фінансових ризиків через системний підхід до управління фінансами.

Бюджетування визначено як ключовий інструмент для планування та контролю фінансів. Регулярне використання бюджету допомагає користувачам утримувати фінансові показники під контролем та дотримуватися стратегії досягнення фінансових цілей за допомогою управлінської дисципліни.

Відносний аналіз відчиняє можливість детального порівняння фінансових показників, враховуючи їхні зміни у відповіді на періодичність часу або порівняння з іншими учасниками системи. Цей аспект є ключовим для виявлення тенденцій та

аналізу результатів у контексті динаміки змін, що сприяє обґрунтованому прийняттю стратегічних рішень. Розроблена система визначає свою мету у наданні зручних та ефективних інструментів для аналізу та прогнозування фінансових потоків. Це включає функціонал для проведення аналізу різних аспектів, дозволяючи користувачеві активно управляти своєю фінансовою сферою.

Другий розділ дипломної роботи присвячений докладному вивченню та аналізу сучасних технологій, які були задіяні у процесі розробки системи для аналізу та прогнозування фінансових потоків. Проходячи огляд різних аспектів, від архітектурних підходів до конкретних інструментів розробки, було виявлено, що обрані технології належним чином відповідають завданням проекту та сприяють створенню ефективної системи.

Було проведений аналіз різних архітектурних підходів, і в результаті обрано монолітну архітектуру. Цей вибір був обґрунтований спрощенням розгортання та обслуговування системи.

Велика увага була приділена ретельному аналізу технологій, спрямованих на реалізацію серверної та клієнтської частини системи. В результаті цього аналізу було обрано *.NET MAUI Blazor Hybrid* в якості ключового інструменту, який вдало поєднує в собі можливості веб–технологій та розробки мобільних додатків. Крім того, *Entity Framework Core* був ідентифікований як високоефективний інструмент для роботи з базою даних, що забезпечує комфортний доступ до фінансових потоків.

В процесі створення інтерфейсу користувача був використаний *Bootstrap* з метою створення зручного та естетичного дизайну. Використання цього інструменту дозволило реалізувати єдинообразний інтерфейс для різних пристроїв, спрощуючи процес розгортання та використання додатка.

У проведеному аналізі баз даних було визначено, що реляційна база даних *SQLite* є оптимальним вибором для ефективного зберігання та управління фінансовою інформацією. Цей вибір базується на її високій продуктивності та надійності в контексті забезпечення надійного зберігання фінансових даних системи.В третьому розділі був описаний комплексний процес розробки системи для аналізу та прогнозування фінансових потоків. Вивчаючи ключові аспекти, такі

як розробка структур даних, бізнес–логіки та графічного інтерфейсу користувача, було виявлено, що створення ефективної та зручної системи вимагає інтегрованого підходу до кожного з цих етапів.

На першому етапі розгляду створення структур даних було проведено детальне вивчення необхідних базових сутностей, таких як транзакції, облікові записи, бюджети, категорії та валюти. Розробка оптимізованої структури бази даних була ключовою для забезпечення ефективного зберігання та обробки фінансової інформації.

У другому етапі описувалась реалізація функціоналу для керування транзакціями, обліковими записами, бюджетами, категоріями та валютами. Застосування контролерів дозволило створити логіку, що забезпечує коректну роботу системи з фінансовими даними в різних сценаріях взаємодії.

На завершальному етапі розглядався процес створення графічного інтерфейсу за допомогою Maui та Bootstrap. Відобразивши основні сторінки для перегляду, додавання та модифікації записів, а також навігаційне меню, був створений інтерфейс, що сприяє зручній навігації та взаємодії з системою.

Комплексне тестування системи виявилося ключовим етапом в оцінці її ефективності та готовності до практичного використання. Процес тестування включав у себе модульне тестування програмного коду, ручне тестування та використання системи в реальних умовах.

Створені модульні тести для різних контролерів системи виявились ефективними виявленням та усуненням помилок на ранніх етапах розробки. Запуск тестів дозволив виявити та виправити деякі помилки в програмному коді, що свідчить про високий стандарт його якості.

Описаний процес ручного тестування допоміг перевірити коректність та зручність інтерфейсу системи, а також правильність виконання основних функцій. Ручне тестування включало в себе сценарії використання, перевірку відповідності функціональних вимог та забезпечення коректної взаємодії з користувачем.

Проілюстрований приклад використання системи для аналізу та прогнозування фінансових потоків дозволив вирішити, як система впроваджується в

реальному бізнес–сценарії та як вона сприяє прийняттю обґрунтованих фінансових рішень.

У ході розробки системи для аналізу та прогнозування фінансових потоків були вирішені ключові завдання, пов'язані із забезпеченням користувачам ефективного інструментарію для управління особистими фінансами. Основна мета полягала в створенні функціональної та легко використовуваної системи, яка дозволить користувачам здійснювати аналіз та прогнозування фінансових потоків з легкістю.

Незважаючи на видиму простоту системи, наданої її інтерфейсом, важливість забезпечення стабільної роботи та ефективного аналізу фінансових даних не може бути переоцінена. Результати тестування підтвердили високу якість програмного коду, а також зручність інтерфейсу для користувачів різного рівня досвіду.

Отже, розроблена система для аналізу та прогнозування фінансових потоків представляє собою не просто інструмент для ведення обліку фінансів, але і потужний інструмент для прийняття обґрунтованих фінансових рішень, що сприяє досягненню фінансової стабільності та успіху.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Комп'ютерна інженерія: методичні рекомендації до виконання дипломних проєктів для студентів освітньо–кваліфікаційного рівня "Бакалавр" напряму підготовки 6.050102 "Комп'ютерна інженерія" / Уклад.: І.А. Жуков, М.М. Проценко – К.: НАУ, 2015. – 36 с.

2. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Національного авіаційного університету. – К.: НАУ, 2017. 63 с.

3. ДСТУ 2392–94 Інформація та документація. Базові поняття.

4. *Investopedia: What Is Personal Finance, and Why Is It Important?* [Електронний ресурс] – Режим доступу до ресурсу: *https://www.investopedia.com/terms/p/personalfinance.asp*

5. *Kevin Lano Financial Software Engineering (Undergraduate Topics in Computer Science), / Howard Haughton 2019*, 105c

6. *C# for Financial Markets, 1st ed. 2013, Daniel J. Duffy, Andrea Germani*

7. *Microsoft.* (2023). *"Entity Framework Core Documentation."* [Електронний ресурс] – Режим доступу до ресурсу: *https://docs.microsoft.com/en–us/ef/*

8. *Microsoft.* (2023). "*ASP.NET Blazor Documentation."* [Електронний ресурс] – Режим доступу до ресурсу: *https://learn.microsoft.com/en–us/aspnet/core/blazor*

9. *Microsoft*. (2023) *"C# documentation"* [Електронний ресурс] – Режим доступу до ресурсу: *https://learn.microsoft.com/en–us/dotnet/csharp/*

10. *Microsoft*. (2023) *".NET MAUI documentation"* [Електронний ресурс] – Режим доступу до ресурсу: *https://learn.microsoft.com/en–us/dotnet/maui/what–is–maui?view=net–maui–8.0*

11. *Microsoft*. (2023) *"ASP.NET Core Blazor Hybrid documentation"* [Електронний ресурс] – Режим доступу до ресурсу: *https://learn.microsoft.com/en–us/aspnet/core/blazor/hybrid/?view=aspnetcore–8.0*

12. Ковальчук, О. С. Розвиток інтернет–технологій / О. С. Ковальчук, І. П. Іванов ; Національний університет інформаційних технологій. – Київ: Видавництво НУІТ, 2015. – 180 с.

13. Сміт, Д. Історія комп'ютерних технологій / Д. Сміт ; Університет техніки та технологій. – Нью–Йорк: Технічне видавництво, 2022. – 250 с. : іл. – Бібліографія: с. 220–240.

14. *SQL* чи *NoSQL* – ось в чому питання [Електронний ресурс] – Режим доступу до ресурсу: *https://alternativescience.net/programming/242–sql–chy–nosql–os–v–chomu–pytannya/*

15. *What is Bootstrap: A Beginner's Guide* [Електронний ресурс] – Режим доступу до ресурсу: *https://careerfoundry.com/en/blog/web–development/what–is–bootstrap–a–beginners–guide*

16. Архітектура програмного забезпечення: все що треба знати [Електронний ресурс] – Режим доступу до ресурсу: *https://wezom.com.ua/ua/blog/arhitektura–programmnogo–obespecheniya*

17. *HTML: HyperText Markup Language* [Електронний ресурс] – Режим доступу до ресурсу: *https://developer.mozilla.org/en–US/docs/Web/HTML*

18. *Web APIs* [Електронний ресурс] – Режим доступу до ресурсу: *https://developer.mozilla.org/en–US/docs/Web/API*

19. *Elixir and Phoenix can do it all!* [Електронний ресурс] – Режим доступу до ресурсу: *https://fly.io/phoenix–files/elixir–and–phoenix–can–do–it–all*

20. Мікросервісна архітектура [Електронний ресурс] – Режим доступу до ресурсу: *https://medium.com/@IvanZmerzlyi/microservices–architecture–461687045b3d*