

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Веб - додаток для керування особистими
фінансами»**

Виконав студент групи ПЗ-20-3
ІЛЬЧЕНКО Дмитро Валерійович

Керівник роботи:
ПОЧТОВІЮК Сергій Іванович

Консультант:
БАКАЛЮК Тетяна Анатоліївна

Рецензент:
ЛОБАНЧИКОВА Надія Миколаївна

Житомир – 2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Завідувач кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«26» лютого 2024 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ІЛЬЧЕНКО Дмитро Валерійович**

Керівник роботи: **ЛЕВЧЕНКО Антон Юрійович**

Тема роботи: **«Веб - додаток для керування особистими фінансами»**,
затверджена Наказом закладу вищої освіти від **«23» лютого 2024 р., №74с**

Вихідні дані для роботи: **створення рахунків, виконання операцій,
планові операції, статистика.**

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням
розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Вакалюк Т.А.	27.02.2023р.	27.02.2023р.
2	Вакалюк Т.А.	27.02.2023р.	27.02.2023р.
3	Вакалюк Т.А.	27.02.2023р.	27.02.2023р.

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з вебдодатку для керування фінансами та пояснювальної записки. Пояснювальна записка до випускної роботи містить 67 сторінок, 21 ілюстрацій та 15 таблиць.

Метою роботи є створення додатку для обліку фінансів з використанням сучасних технологій та шифрування для аутентифікації придбаних квитків та.

Архітектура вебдодатку керування фінансами побудована на основі сучасних вебтехнологій. Серверна частина реалізована за допомогою Node.js, що забезпечує високопродуктивний та масштабований бекенд. Фронтенд частина створена за допомогою React, що дозволяє будувати динамічні та інтерактивні інтерфейси користувача.

Вебдодаток забезпечує користувачам можливість легко створювати рахунки, виконувати операцію та категоризіція їх, одночасно гарантуючи захист їх особистих даних та безпеку транзакцій.

КЛЮЧОВІ СЛОВА: ВЕБДОДАТОК, ФІНАНСИ, NODE.JS, REACT, , АВТЕНТИФІКАЦІЯ.

					ІПЗ.КР.Б-121-24-ІПЗ			
Зм.	Арк.	№ докум.	Підпис	Дата	Веб - додаток для керування особистими фінансами	Літ.	Арк.	Аркушіів
Розроб.		Льченко Д.В.						
Керівник		Поштовюк С.І.					3	67
Рецензент		Лобанчикова Н.М				Житомирська політехніка, група ІПЗ-20-3		
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The bachelor's thesis consists of a web application for financial management and an explanatory note. The explanatory note contains 67 pages, 21 illustrations, and 15 tables. The goal of the work is to create an application for financial accounting using modern technologies and encryption for the authentication of acquired receipts.

The architecture of the financial management web application is built on modern web technologies. The server-side is implemented using Node.js, providing a high-performance and scalable backend. The frontend is created using React, allowing for the construction of dynamic and interactive user interfaces.

The web application enables users to easily create accounts, perform transactions, and categorize them, while simultaneously ensuring the protection of their personal data and the security of transactions.

KEYWORDS: WEB APPLICATION, FINANCE, NODE.JS, REACT, AUTHENTICATION.

					ИП3.КР.Б-121-24-И3	
						4

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ВЕБДОДАТКУ КЕРУВАННЯ ФІНАНСАМИ	9
1.1 Постановка задачі	9
1.2 Аналіз аналогів програмного продукту.....	11
1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення.....	18
Висновки до першого розділу.	21
РОЗДІЛ 2. ПРОЕКТУВАННЯ ВЕБДОДАТКУ ДЛЯ КЕРУВАННЯ ФІНАНСАМИ	22
2.1 Визначення варіантів використання	22
2.2 Розробка бази даних системи.....	29
2.3 Проектування та реалізація алгоритмів роботи системи.....	32
2.4 Реалізація функціоналу вебдодатку	33
Висновки до другого розділу.	43
РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ	44
3.1 Порядок встановлення та налаштування параметрів додатку	44
3.2 Структура інтерфейсу вебдодатку та порядок роботи.....	46
3.3 Тестування роботи вебдодатку	51
Висновки до третього розділу	53
ВИСНОВКИ.....	55
ДОДАТКИ.....	58
Додаток А	59

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – прикладний програмний інтерфейс

БД – база даних

ПЗ – програмне забезпечення

СУБД – система управління базами даних

					ІПЗ.КР.Б-121-24-ПЗ	
						6

ВСТУП

Актуальність теми полягає у тому, що у сучасному світі все більше людей усвідомлюють важливість управління своїми фінансами. Вебдодаток для фінансового керування може стати важливим інструментом для підвищення фінансової грамотності населення. Такий додаток дозволяє користувачам ефективно контролювати свої витрати, доходи і заощадження з будь-якого місця та у будь-який час, що робить процес управління фінансами значно зручнішим. Також веб-додаток може допомогти користувачам встановлювати та досягати фінансових цілей, таких як накопичення на покупку, зменшення боргів.

Метою випускної кваліфікаційної роботи бакалавра є створення сучасного вебдодатку для керування фінансами, що забезпечує зручність для усі категорій користувачів, а також забезпечить високий рівень безпеки та конфіденційності. Платформа повинна використовувати сучасні технології для автентифікації користувачів, надавати їм гарний функціонал та мати зрозумілий інтерфейс. Це дозволить користувачам легко та без проблем використовувати весь наданий їм функціонал додатку, а також забезпечить захист особистих даних.

Завданням випускної кваліфікаційної роботи бакалавра є розробка вебдодатку для керування особистими фінансами, що забезпечить високий рівень зручності для користувачів та допоможе їм у повсякденному житті. Вебдодаток має бути побудованим за допомогою використання сучасних технологій. Має бути спроектований та реалізований інтуїтивно зрозумілий інтерфейс для усіх категорій користувачів. Важливо реалізувати всі функції, такі як автентифікація, створення рахунків, категорій, операцій, запланованих операцій а також реалізувати зручний перегляд статистики для користувачів.

Об'єктом дослідження є створення вебдодатку для керування фінансами, що включає розробку ефективних та безпечних механізмів захисту особистих даних і операційних процесів, а також забезпечення інтуїтивно зрозумілого інтерфейсу для користувачів.

Предметом дослідження є розробка та реалізація вебдодатку для керування фінансами. Розробка механізмів відслідковування витрат та доходів,

					ІПЗ.КР.Б-121-24-ПЗ	
						7

використання сучасних технологій для захисту особистих даних та операційних процесів.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ВЕБДОДАТКУ КЕРУВАННЯ ФІНАНСАМИ

1.1 Постановка задачі

Метою кваліфікаційної роботи бакалавра є розробка вебдодатку для керування особистими фінансами, який надає користувачам зручний інструмент для ведення обліку їхніх доходів, витрат та зберігання фінансової інформації. Додаток буде розроблений з урахуванням простоти використання, ефективності та безпеки даних.

Для багатьох людей фінансове планування може бути важкою задачею. Відстеження витрат, планування майбутніх витрат та визначення ефективних стратегій збереження інколи може бути заплутаною і нудною процедурою. Тут на допомогу приходять додатки для керування особистими фінансами, які не лише роблять цей процес більш простим та зручним, але й можуть надати рекомендації для досягнення фінансової мети.

В цьому контексті розробка додатку для керування особистими фінансами є важливим завданням, спрямованим на надання користувачам ефективного та зручного інструменту для вирішення їхніх фінансових потреб. Такий додаток може стати невід'ємною частиною повсякденного життя, допомагаючи керувати фінансами, планувати витрати та досягати фінансових цілей.

Зосереджуючись на простоті використання, функціональності та безпеці даних, розроблений вебдодаток повинен бути простим в розумінні та керуванні для будь-якого користувача, незалежно від його рівня фінансової грамотності чи досвіду.

В результаті повинно бути створено вебдодаток на якому будь-який бажаючий може додати свої рахунки, категорії, переглядати історію витрат та доходів, а також переглядати статистику за різні періоди часу

Основними етапами розробки платформи є:

1. Розробка дизайну вебдодатку:

- розробка загальної концепції дизайну застосунку;

- створення компонентів інтерфейсу користувача;
 - проектування макетів сторінок;
2. Побудова архітектури клієнтської частини додатку:
 - встановлення основної структури клієнтського коду;
 - організація файлів, бібліотек і фреймворків на клієнтській стороні;
 - налагодження взаємодії між серверною та клієнтською частинами;
 3. Побудова архітектури серверної частини додатку:
 - налаштування сервера та робочого середовища;
 - розробка серверної логіки та обробки запитів;
 4. Розробка системи авторизації та реєстрації:
 - створення механізму автентифікації користувачів;
 - створення механізму для реалізації реєстрації нових користувачів;
 5. Розробка основного функціоналу:
 - інтеграція бекенду на клієнтській стороні;
 - створення та підтримка бази даних для зберігання інформації;
 6. Тестування додатку:
 - проведення функціонального тестування окремих частин додатку;
 - тестування графічного інтерфейсу;

Результатом виконання дій описаних в плані є створення вебдодатку для керування фінансами, користувачі якого зможуть легко та без проблем керувати особистими фінансами, переглядати історію витрат та статистику, а також вести кілька рахунків в різних валютах з їх конвертацією по останньому актуальному курсу. Для реалізації додатку було вирішено створити кілька основних модулів:

- Модуль користувача: містить в собі модель та методи для взаємодії з базою даних, а також контроллер для автентифікації та взаємодії з іншими модулями системи;
- Модуль рахунку: містить в собі модель, що визначає структуру даних для зберігання інформації про рахунки користувачів. Також включає а себе функції створення нових рахунків, отримання списку рахунків користувача;

- Модуль валюти: визначає структуру даних для зберігання інформації про валюту, таку як код валюти (наприклад, USD, EUR), назва валюти та обмінний курс;
- Модуль категорій: містить в собі модель категорії визначає структуру даних для зберігання інформації про категорії, функції для створення нових категорій, отримання списку категорій;
- Модуль операцій: визначає структуру даних для зберігання інформації про фінансові операції користувача. Також включає функції для створення нових операцій, отримання списку операцій користувача, фільтрації операцій за типом операції;
- Модуль планових операцій: містить модель, що визначає структуру даних для зберігання інформації про планові фінансові операції користувача. Це може включати такі дані, як назва операції, акаунт, до якого вона відноситься, категорія, сума, частота повторення операції (щотижня, щомісяця тощо);
- Модуль статистики: включає функції для обчислення різноманітних статистичних показників, таких як загальна сума доходів та витрат за певний період;

1.2 Аналіз аналогів програмного продукту

Для створення вебдодатку з керування фінансами потрібно детально дослідити конкурентний ринок, тому розглянемо та проаналізуємо найпопулярніші рішення на ринку. Це дасть нам розуміння потреб користувачів та визначити найбільш успішні рішення для додатку.

Одним з таких додатків є платформа You Need a Budget (YNAB) [\[1\]](#) (рис. 1.1).

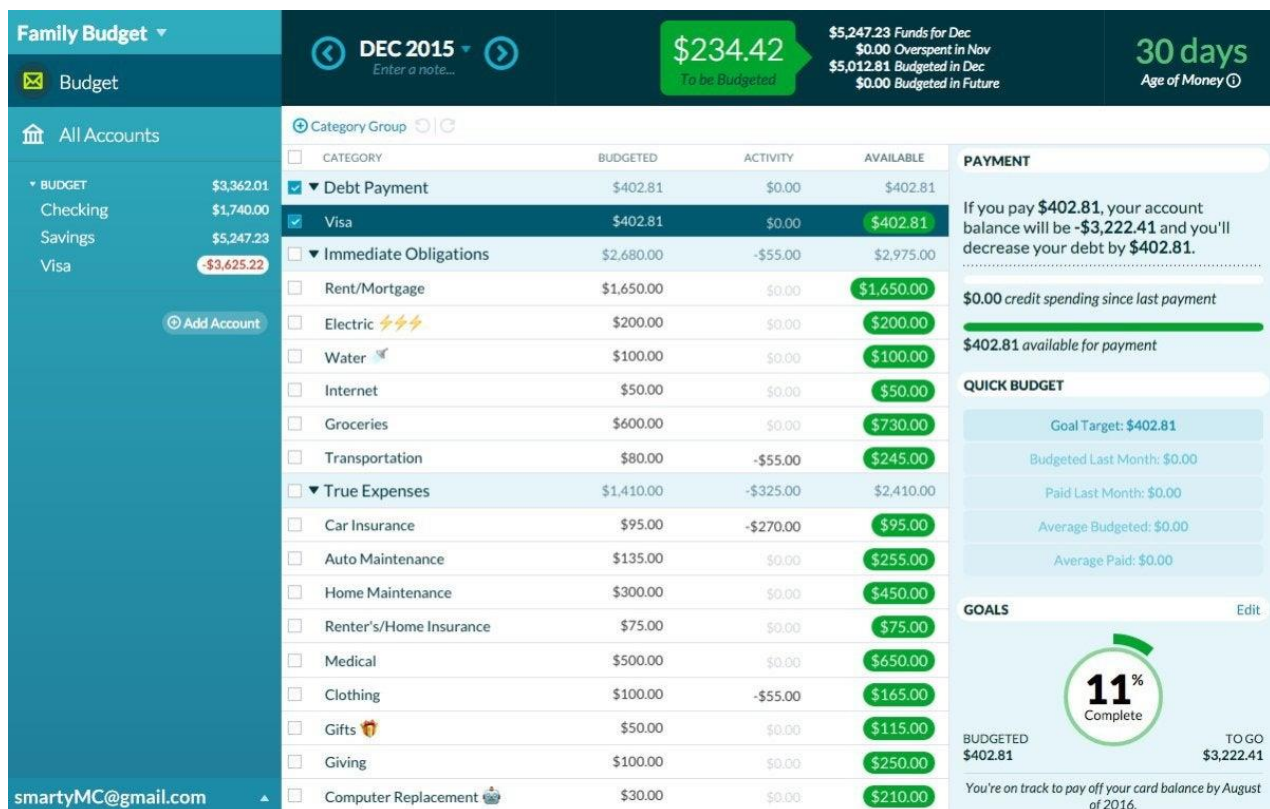


Рис. 1.1 Вебдодаток керування фінансами You Need a Budget [1]

Одна з найпопулярніших систем керування фінансами, що допомагає користувачам планувати бюджет, відстежувати витрати та досягати фінансових цілей

Основними перевагами You Need a Budget є:

- Фокус на жорсткому плануванні бюджету;
- Можливість відстеження категорій витрат;
- Прив'язка до фінансових цілей;
- Навчальні матеріали та підтримка;
- Висока надійність.

Недоліками є:

- Складність у використанні;
- Вартість підписки;
- Потреба у регулярному оновленні бюджету.

Personal Capital [2] - це інструмент для управління фінансами та інвестиціями, який дозволяє користувачам відстежувати свої активи,

зобов'язання, доходи та витрати. Цей додаток є особливо корисним для тих, хто хоче мати більш детальну картину своїх інвестицій та загального фінансового стану.

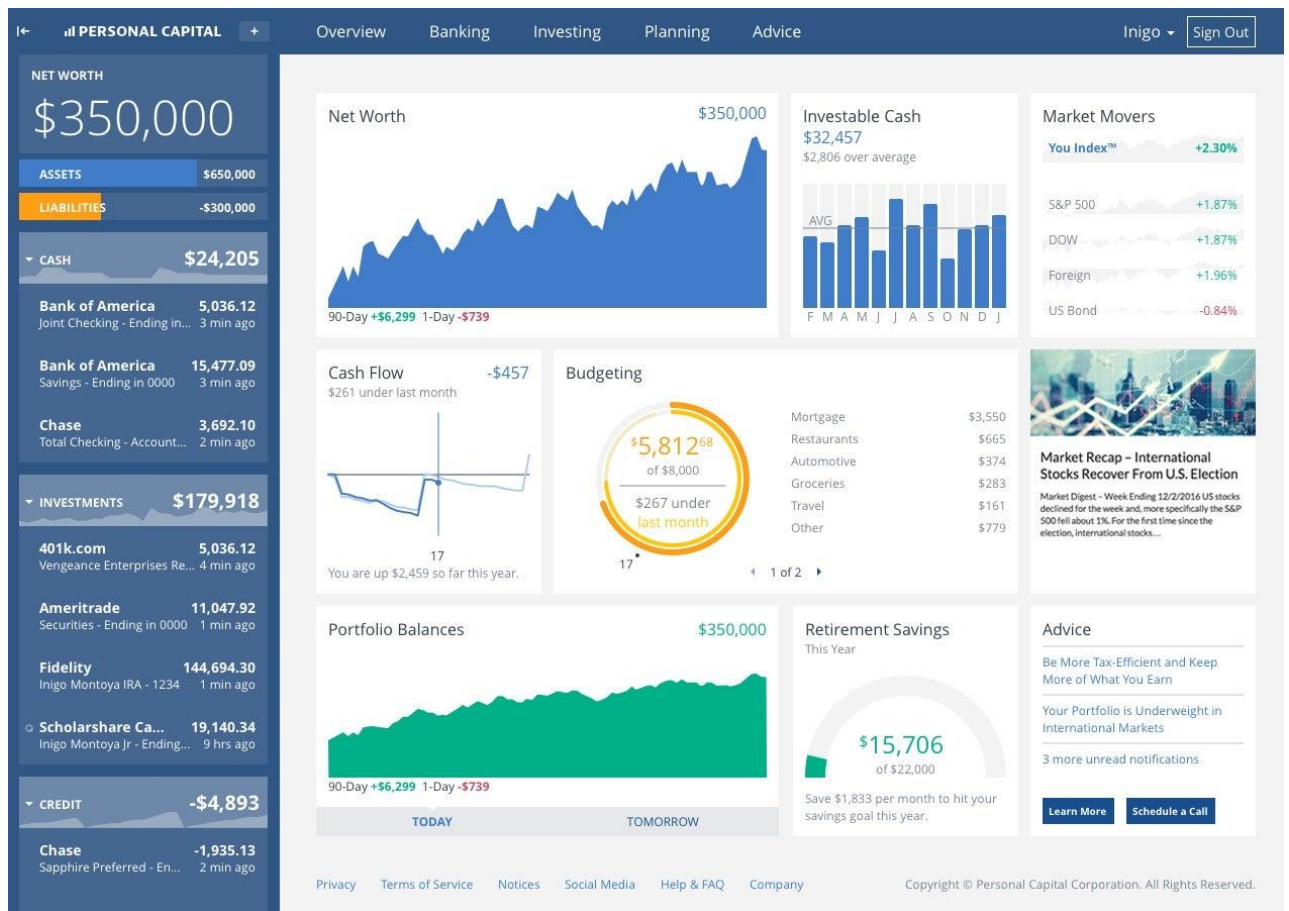


Рис. 1.2. Вебдодаток керування фінансами Personal Capital [2]

Основними перевагами Personal Capital є:

- Детальна фінансова аналітика;
- Можливість відстежувати різні фінансові активи на одній платформі;
- Безкоштовне використання.

Недоліками є:

- Складність у використанні для новачків;
- Фокус на інвестиції;
- Нав'язливі пропозиції.

PocketGuard[3] — це мобільний додаток для управління особистими фінансами, який допомагає користувачам контролювати свої витрати та бюджет. Основна мета додатка — зробити управління фінансами простішим та

ефективнішим, забезпечуючи користувачів інструментами для автоматичного відстеження витрат і доходів.

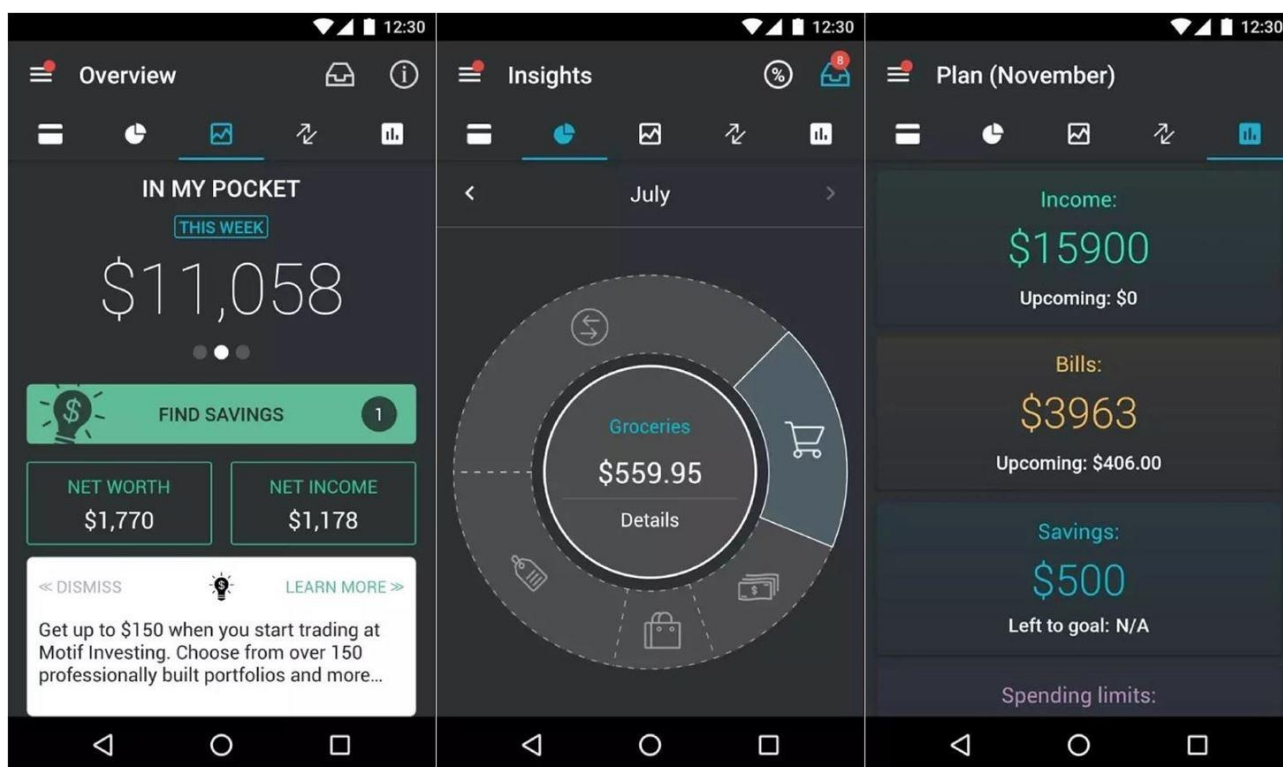


Рис. 1.3. Мобільний додаток керування фінансами Pocket Guard^[3]

Основні переваги Pocket Guard:

- Автоматичне відстеження витрат;
- Зручний інтерфейс.

Основні недоліки Pocket Guard:

- Обмежені можливості налаштування;
- Проблеми з синхронізацією;
- Відсутність деяких аналітичних функцій;
- Лише мобільна версія.

Money Lover^[4] є популярним додатком для керування особистими фінансами, який надає широкий спектр функцій для відстеження доходів і витрат. Цей додаток доступний як на мобільних пристроях, так і на веб-платформі, що робить його зручним для користувачів різних пристроїв.

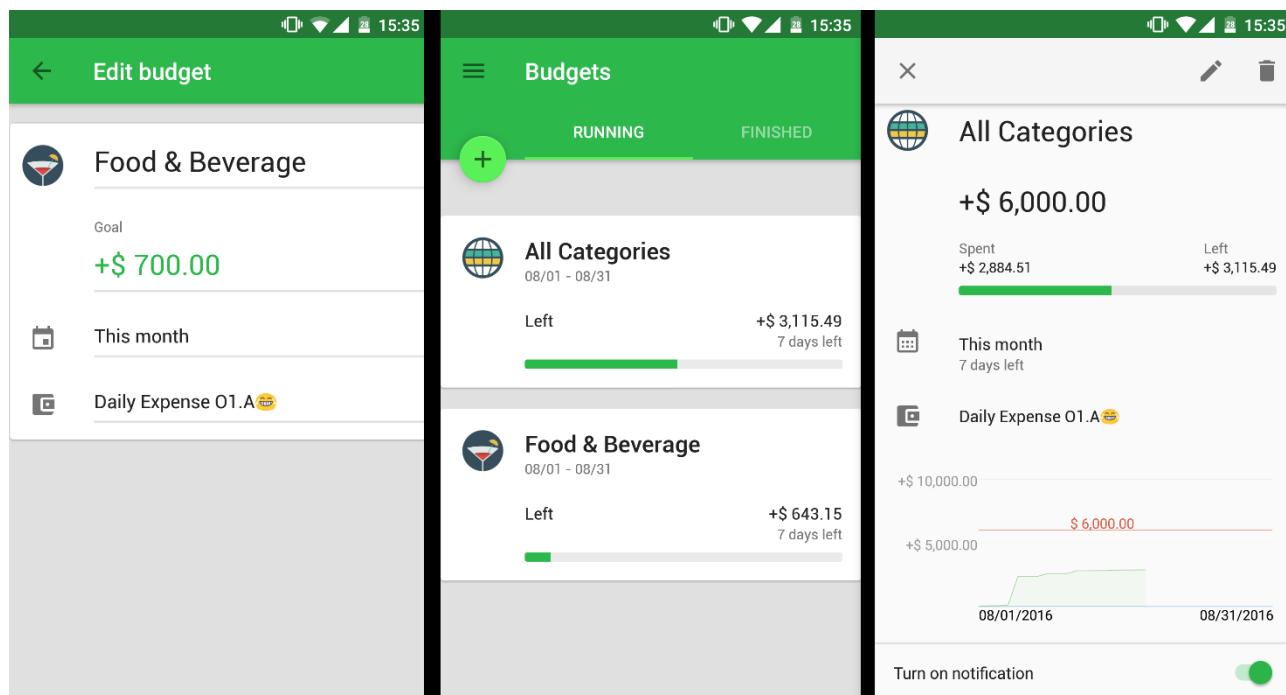


Рис. 1.4. Вебдодаток для керування особистими фінансами Money Lover^[4]

Основні переваги Money Lover:

- Інтуїтивно зрозумілий інтерфейс;
- Зручне планування бюджету;
- Підтримка багатьох валют.

Основні недоліки Money Lover:

- Реклама у безкоштовній версії;
- Обмежені функції без підписки;
- Іноколи проблеми з синхронізацією;

Goodbudget^[5] – це програма для управління особистими фінансами, яка використовує метод конвертів для бюджетування. Цей додаток дозволяє користувачам розподіляти свої доходи по віртуальних конвертах, кожен з яких представляє певну категорію витрат. Це допомагає організувати фінанси та контролювати витрати в межах встановлених бюджетів.

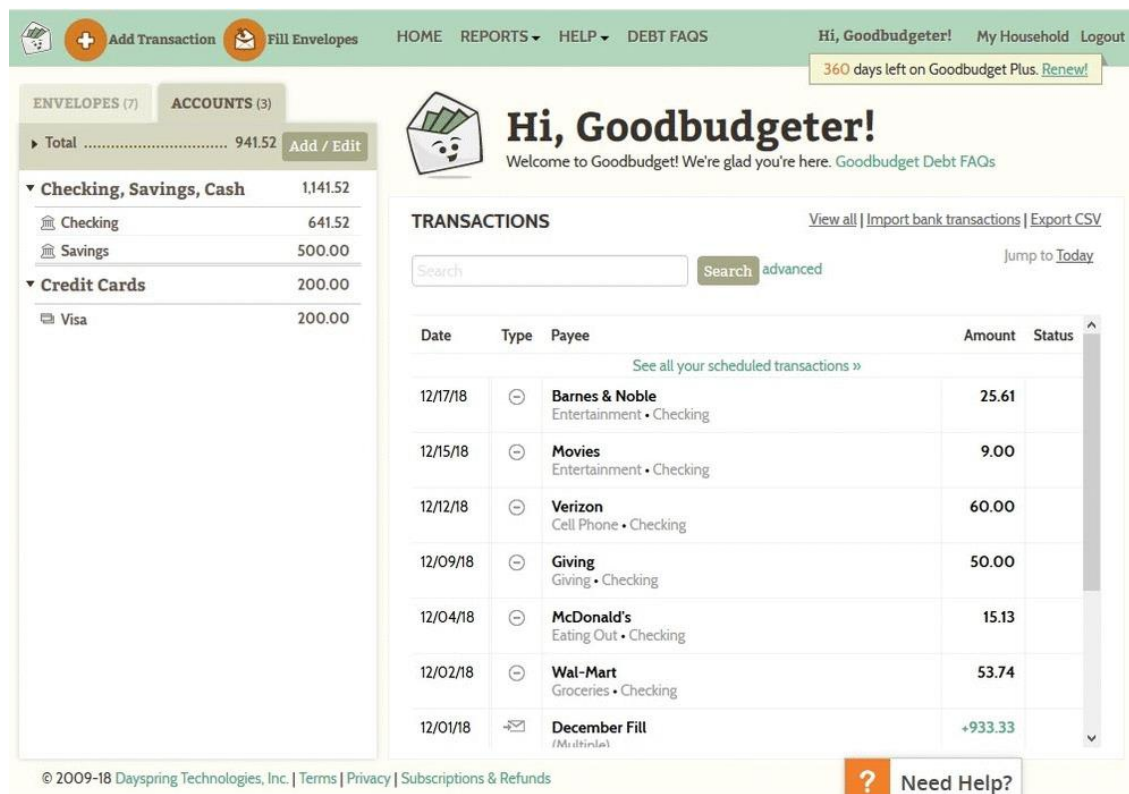


Рис. 1.5. Програма для управління особистими фінансам Goodbudget^[5]

Основними перевагами Goodbudget є:

- Легкий у використанні;
- Система конвертів допомагає контролювати витрати;
- Синхронізація з іншими користувачами.

Недоліками є:

- Вимагає багато ручного введення даних;
- Обмежені можливості в безкоштовній версії.

Виходячи з наявних переваг та недоліків, складемо порівняльну таблицю вищезгаданих програмних аналогів (табл. 1.1).

Таблиця 1.1

Таблиця порівняння додатків для керування фінансами

Характеристика	YNAB	Personal Capital	Pocket Guard	Money Lover	Goodbudget
Простота використання	-	-	+	+	+

Характеристика	YNAB	Personal Capital	Pocket Guard	Money Lover	Goodbudget
Синхронізація з банками	+	+	+	-	-
Підписка	+	+	+	+	+
Кросплатформеність	+	+	-	-	-
Навчальні матеріали	+	+	-	-	-

Переглянувши різноманітні додатки для обліку фінансів, можна визначити кілька основних проблем, які виникають у процесі їх використання. Перш за все, існує проблема у простоті використання. Personal capital та YNAB є професійними програмами для досвідчених користувачів, через те новачки не одразу зрозуміють послідовність дій у цих додатках. Друга проблема стосується кросплатформеності. Деякі із додатків реалізовані лише для смартфонів а деякі лише для персональних комп'ютерів. Це може доставити складностей для користувачів різних категорій.

Додатковою проблемою може бути відсутність статистики та мала кількість категорій витрат та доходів для їх відстеження. Це може заплутати звичайного користувача та викликати певні проблеми при керуванні бюджетом.

Враховуючи інформацію отриману при дослідженні було прийнято рішення створити вебдодаток, який буде інтуїтивно зрозумітий для користувачів усіх вікових категорій. Також додаток буде мати інструменти та засоби для одночасного ведення кількох рахунків з різними валютами, створення власних категорій, відслідковування історії витрат, плановими операціями та зрозумілою статистикою за різні періоди часу.

1.3 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Вебдодаток буде побудований на клієнт-серверній архітектурі. Серверна частина використовуватиме сучасну платформу Node.js для реалізації API.

Node.js[\[6\]](#) — це середовище виконання JavaScript, відоме своєю однопоточною архітектурою з подіями, що забезпечує ефективність обробки великого обсягу одночасних з'єднань.

Основні переваги Node.js включають:

- Неблокуюче введення-виведення: Завдяки асинхронному програмуванню Node.js може продовжувати виконання коду під час очікування на завершення операцій введення-виведення, що забезпечує високу швидкодію, особливо для мережових операцій;
- Висока масштабованість: Node.js ідеально підходить для високонавантажених додатків, оскільки він ефективно обробляє асинхронні запити без затримок;
- Спільне використання коду: Використання JavaScript як на клієнтській, так і на серверній стороні спрощує розробку і зменшує навантаження на розробників;
- Спрощений веб-сервер: Node.js має вбудовану бібліотеку http для створення веб-серверів без необхідності використання сторонніх серверів.

Клієнтська частина нашого додатка буде реалізована за допомогою бібліотеки React.

React[\[7\]](#) — це відкрита JavaScript бібліотека, розроблена Facebook для побудови інтерфейсів користувача, яка відзначається високою динамічністю та інтерактивністю.

Основні переваги React включають:

- Ефективність: Використання віртуального DOM дозволяє оптимізувати рендеринг компонентів і забезпечує високу продуктивність додатків;

- Компонентна архітектура: Розробка додатків з малими, самодостатніми компонентами спрощує розробку, тестування і підтримку коду;
- Односторонній потік даних: Використання одностороннього потоку даних полегшує управління станом додатку і робить його більш передбачуваним;
- Розширюваність: Широкий вибір сторонніх бібліотек і інструментів дозволяє розширювати функціональність React.

У підсумку, Node.js і React є потужними інструментами для створення сучасних веб-додатків з високою швидкістю, масштабованістю і простотою у розробці. Їх комбінація дозволить нам ефективно реалізувати платформу з API на серверній стороні і динамічним інтерфейсом користувача на клієнтській.

MySQL [\[8\]](#) є популярною системою управління реляційними базами даних (RDBMS), яка використовується для зберігання, управління та доступу до даних. Вона була розроблена як відкрите програмне забезпечення і на сьогодні є однією з найпоширеніших баз даних у світі. Ось деякі її переваги:

- Відкритий вихідний код: MySQL є відкритим програмним забезпеченням, що дозволяє використовувати його безкоштовно і модифікувати під власні потреби.
- Швидкість і продуктивність: MySQL відомий своєю високою швидкістю і здатністю обробляти великі обсяги даних.
- Простота використання: MySQL має зрозумілий і простий у використанні інтерфейс, а також багато документації і ресурсів для розробників.
- Підтримка транзакцій: MySQL підтримує транзакції, що дозволяє забезпечувати цілісність даних і виконувати кілька операцій як єдину логічну одиницю.
- Гнучкість: MySQL підтримує різні типи даних і функції, що дозволяє адаптувати його під різні вимоги і завдання.
- Висока продуктивність: MySQL оптимізований для швидкого виконання запитів і може обробляти великі обсяги даних з високою швидкістю.

- **Безпека:** MySQL пропонує різні рівні безпеки для захисту даних, включаючи шифрування, контроль доступу і управління користувачами.
- **Кросплатформенність:** MySQL підтримує роботу на різних операційних системах, включаючи Windows, Linux, macOS, що робить його універсальним рішенням для різних середовищ.

MySQL є потужним і гнучким інструментом для зберігання і управління даними, що робить його популярним вибором для багатьох веб-додатків, включаючи системи управління особистими фінансами.

Для реалізації інтуїтивно зрозумілого інтерфейсу було вирішено використовувати TailwindCSS

Tailwind CSS[\[9\]](#) - це сучасна CSS-бібліотека, яка дозволяє швидко створювати адаптивні та красиві інтерфейси користувача. На відміну від традиційних CSS-фреймворків, таких як Bootstrap, Tailwind CSS використовує утилітарний підхід, де кожен клас відповідає за одну конкретну властивість стилю. Основні особливості Tailwind CSS:

- **Утилітарні класи:** Замість написання користувацького CSS для стилізації елементів, ви використовуєте класи, які визначають конкретні стилі. Наприклад, клас `bg-blue-500` встановлює синій фон, а клас `text-center` вирівнює текст по центру. Це дозволяє швидко будувати інтерфейси, зменшуючи кількість написаного CSS-коду.
- **Адаптивний дизайн:** Tailwind CSS підтримує створення адаптивних інтерфейсів за допомогою класів, що змінюють стиль залежно від розміру екрану. Наприклад, `md:bg-red-500` змінить фон на червоний при ширині екрану середнього розміру і більше.
- **Вбудовані плагіни:** Tailwind CSS має вбудовані плагіни для роботи з формами, типографією, анімаціями та іншими аспектами дизайну.

Висновки до першого розділу.

Проведено аналіз предметної області щоб визначити ключові вимоги до вебдодатку керування фінансами головною метою якої є відслідковування бюджету на кількох рахунках.

У процесі дослідження аналогічних продуктів було проаналізовано кілька додатків для керування фінансами з різним функціоналом, принципами роботи, та на різних платформах. Також були сформовані основні вимоги для власного додатку.

На основі цих вимог та критеріїв було обрано набір основних технологій та засобів для реалізації вебдодатку.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ВЕБДОДАТКУ ДЛЯ КЕРУВАННЯ ФІНАНСАМИ

2.1 Визначення варіантів використання

Вебдодаток для керування фінансаси надає інтерфейс, через який користувачі можуть додавати свої рахунки, створювати власні категорії витрат та доходів, переглядати їх, створювати планові операції, а також переглядати статистику за різний період часу, та порівнювати її.

Для наглядної взаємодії між елементами додатку було створено Діаграму варіантів використання (рис 2.1).

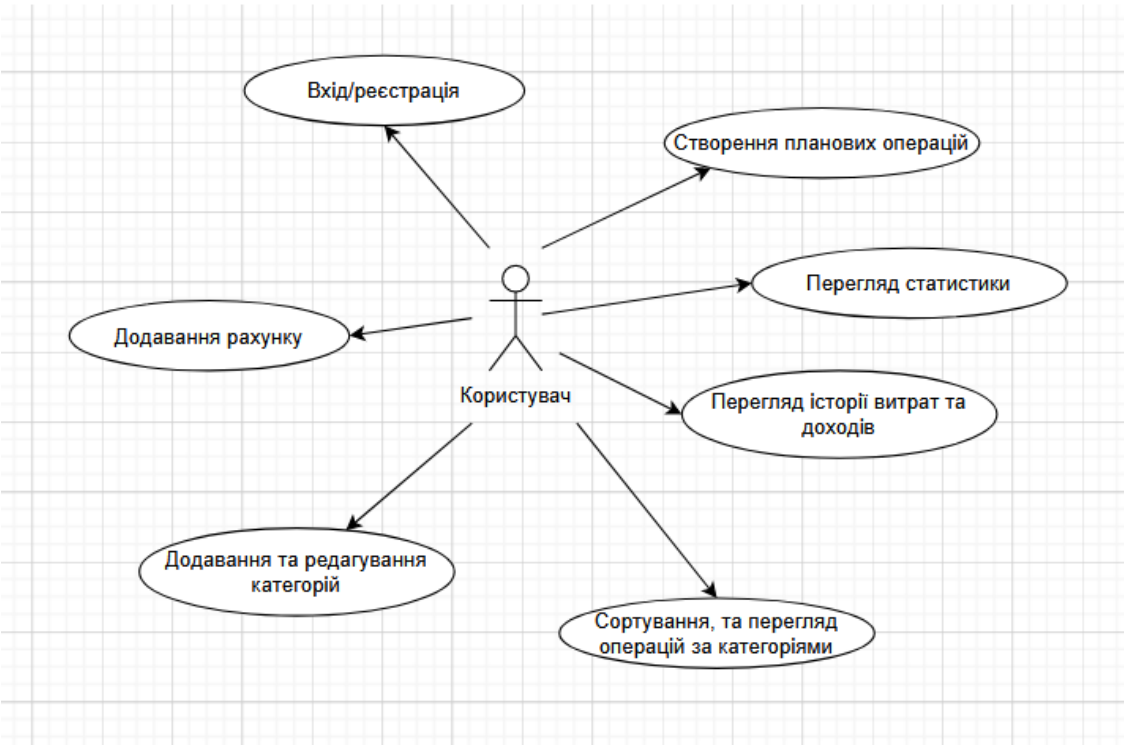


Рис. 2.1. Діаграма варіантів використання

Опис діаграми варіантів використання наведено в таблицях 2.1 – 2.7.

Таблиця 2.1

Опис прецеденту «Вхід/Реєстрація»

Назва варіанту використання	Вхід/реєстрація
Основні актори	Користувач

Продовження таблиці 2.1

Короткий опис	Користувачі можуть створювати обліковий запис, або автентифікуватися за логіном та паролем.
---------------	---

Таблиця 2.2

Опис прецеденту «Додавання рахунку»

Назва варіанту використання	Додавання рахунку
Основні актори	Користувач
Короткий опис	Користувачі можуть створювати свої рахунки, а також обирати валюту для них.

Таблиця 2.3

Опис прецеденту «Додавання та редагування категорій»

Назва варіанту використання	Додавання та редагування категорій
Основні актори	Користувач
Короткий опис	Користувачі можуть додавати власні категорії витрат, а також обирати їх тип (витрати або доходи).

Таблиця 2.4

Опис прецеденту «Сортування та перегляд операцій за категоріями»

Назва варіанту використання	Сортування та перегляд операцій за категоріями
Основні актори	Користувач
Короткий опис	Користувачі можуть відсортувати та переглядати всі свої операції та операції за певними обраними ними категоріями.

Таблиця 2.5

Опис прецеденту «Перегляд історії витрат та доходів»

Назва варіанту використання	Перегляд історії витрат та доходів
Основні актори	Користувач
Короткий опис	Користувачі можуть переглядати усю історію своїх фінансових операцій.

Таблиця 2.6

Опис прецеденту «Перегляд статистики»

Назва варіанту використання	Перегляд статистики
Основні актори	Користувач
Короткий опис	Користувачі зможуть переглядати статистику у вигляді графіків за різні періоди часу, та бачити різницю витрат та доходів.

Таблиця 2.7

Опис прецеденту «Аналітика та звітність»

Назва варіанту використання	Сторення планових операцій
Основні актори	Користувач
Короткий опис	Користувачі зможуть створювати планові операції, обирати категорії для них, частоту виконання, а також їх тип.

Основна ціль розробки додатку для керування фінансами полягає у створенні зручного онлайн-середовища, яке дозволяє користувачам ефективно управляти своїми фінансами. Додаток спрямований на забезпечення ефективної комунікації та взаємодії між користувачами та їхніми фінансовими даними, а також надання зручного і надійного інструменту для контролю та планування фінансів.

Основна мета полягає у сприянні фінансовій грамотності, плануванню бюджету, відстеженню витрат та доходів. Додаток надає можливість користувачам відстежувати свої фінансові операції, створювати рахунки, та планові операції. Завдяки цьому, основною метою є створення умов для успішного управління фінансами, що може мати значний вплив на їхнє фінансове благополуччя та стабільність.

Користь для користувачів

- **Фінансова грамотність:** Користувачі зможуть краще розуміти свої фінансові потоки, що сприяє підвищенню фінансової грамотності;
- **Контроль витрат:** Завдяки постійному відстеженню витрат користувачі зможуть уникати непотрібних витрат та економити кошти;

					ІПЗ.КР.Б-121-24-ПЗ	
						24

- **Стабільність:** Ефективне управління фінансами сприяє загальному фінансовому благополуччю та стабільності користувачів;

Таким чином, додаток для керування фінансами створює необхідні умови для ефективного управління фінансовими ресурсами, що має значний позитивний вплив на фінансове здоров'я та стабільність користувачів.

Вимоги користувачів:

Внутрішній користувач:

1. Можливість відслідковування різних рахунків з різними валютами.
2. Перегляд статистики витрат та доходів, а також їх аналітика.
3. Можливість редагування операцій.
4. Можливість планування витрат та оформлення «підписки на щоденні, щотижневі, щомісячні та щорічні платежі.
5. Можливість створювати власні категорії витрат.

Функціональні вимоги:

1. Система реєстрації та авторизації для користувачів.
2. Можливість відслідковування різних рахунків з різними валютами.
3. Постійне оновлення актуального курсу валют, та їх конвертація.
4. Перегляд статистики витрат та доходів, а також їх аналітика.
5. Можливість редагування операцій.
6. Можливість планування витрат та оформлення «підписки на щоденні, щотижневі, щомісячні та щорічні платежі.
7. Можливість створювати власні категорії витрат.

Не функціональні вимоги:

1. Сприйняття:

- час, що потрібен для розуміння додатку – до 5 хвилин;
- час відповіді додатку на дії користувача не більше 3 секунд;
- інтерфейс додатку повинен бути інтуїтивно зрозумілим та зручним;

2. Надійність:

- вебдодаток має бути доступний протягом усієї доби;

3. Продуктивність:

- вебдодаток має витримувати користування багатьма користувачами одночасно;
- вебдодаток повинен забезпечувати швидку обробку запиту та завантаження сторінок;

Системні вимоги:

1. Вимоги до середовища виконання: платформа повинна задовільняти мінімальні апаратні вимоги:
 - 1.5 Гб оперативної пам'яті;
 - процесор з тактовою частотою не нижче 1.2 GHz;
 - вихід в мережу інтернет;
2. Вимоги до серверу інформаційної системи:
 - підтримка сервером технологій NodeJS;

Аналіз функціональних вимог допоміг виділити наступні сутності, а саме User, Account, Currency, OperationCategory, Operation, PlanedOperation, Statistic. Вони забезпечать програмну реалізацію додатку (рис. 2.2).

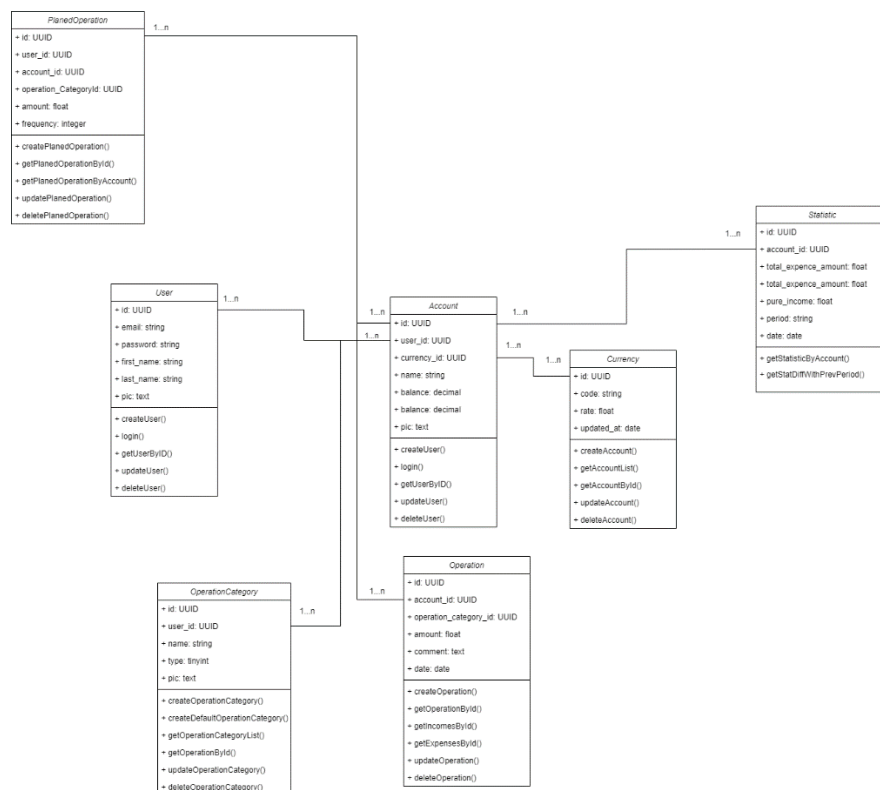


Рис. 2.2 Діаграма класів

User – клас який відповідає за користувача:

- getUserById() – відповідає за отримання користувача за ідентифікатором;
- updateUser() – відповідає за оновлення інформації користувача;
- deleteUser() – відповідає за видалення користувача;
- login() – відповідає за авторизацію;

Account – клас який відповідає за рахунки:

- createAccount() – створення рахунку;
- getAccountList() – відповідає за отримання всіх рахунків користувача;
- getAccountById() – відповідає за отримання рахунку за ідентифікатором;
- updateAccount() – відповідає за оновлення рахунку;
- deleteAccount() – відповідає за видалення рахунку;

Operation – клас який відповідає за операції:

- createOperation() – відповідає за створення операції;
- getOperationById() – відповідає за отримання категорії за ідентифікатором;
- getOperationByAccount() – відповідає за отримання операцій за рахунком;
- getIncomesByAccount() – відповідає за отримання прибутків за рахунком;
- getExpensesByAccount() – відповідає за отримання витрат за рахунком;
- updateOperation() – відповідає за оновлення операцій;
- deleteOperation() – відповідає за видалення операції;

OperationCategory – клас який відповідає за категорії операцій:

- createDefaultOperationCategory() – відповідає за створення стандартних категорій;

- createOperationOperation() – відповідає за створення операцій;
- getOperationCategoryById() – відповідає за отримання категорії за ідентифікатором;
- getOperationCategoriesList() – відповідає за отримання списку всіх категорій;
- updateOperationCategory() – відповідає за оновлення операцій;
- deleteOperationCategory() – відповідає за видалення категорії.

PlannedOperation – клас що відповідає за планові операції:

- createPlannedOperation() – відповідає за створення планових операцій;
- getPlannedOperationById() – відповідає за отримання планових операцій за ідентифікатором;
- getPlannedOperationByAccount() – відповідає за отримання планових операцій за рахунком;
- getPlannedIncomesByAccount() – відповідає за отримання планових доходів за рахунком;
- getPlannedExpensesByAccount() – відповідає за отримання планових витрат за рахунком;
- updatePlannedOperation() – відповідає за оновлення планових операцій;
- deletePlannedOperation() – відповідає за видалення планових операцій.

Currency – клас що відповідає за валюту:

- updateCurrencyRate() – відповідає за оновлення курсу валют;
- getCurrencies() – відповідає за отримання валют.

Statistic – клас що відповідає за статистику:

- getStatisticByAccount() – відповідає за отримання статистики;
- getStatDiffWithPrevPeriod() – відповідає за порівняння з попередніми періодами.

2.2 Розробка бази даних системи

Для зберігання файлів та інформації у додатку використовується MySQL, система управління базами даних, яка базується на реляційній моделі. Для даної системи було побудовано таку схеми бази даних (рис. 2.3).

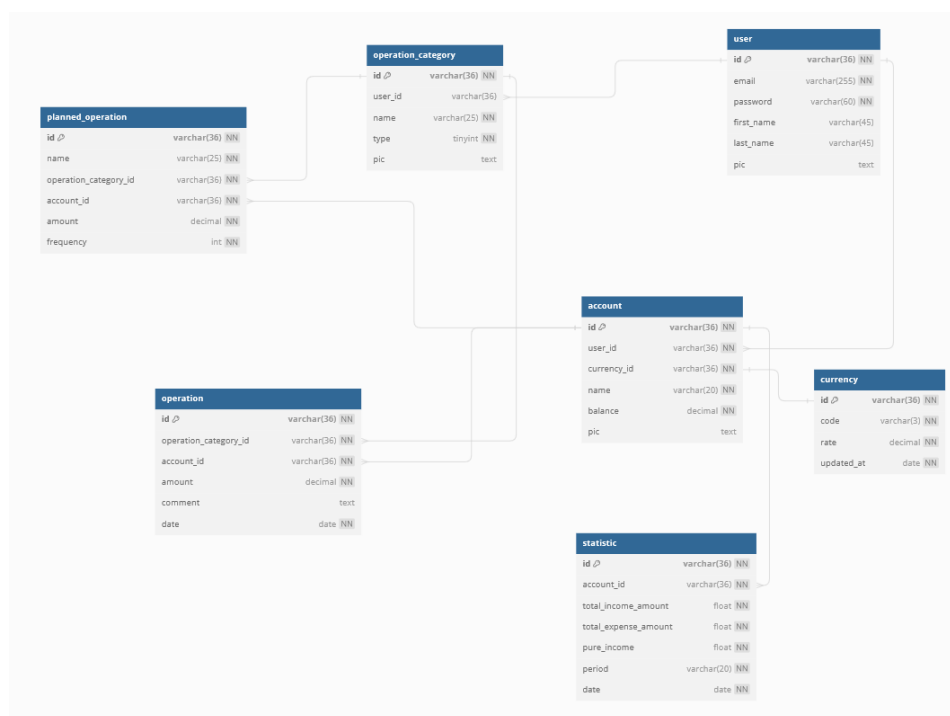


Рис. 2.3 Діаграма бази даних.

На діаграмі вище (рис. 2.4) зображення структура вебдодатку для керування фінансами. Дана таблиця складається з семи таблиць:

1. User – таблиця, яка зберігає данні користувачів.
2. Account – таблиця, яка зберігає дані про рахунки.
3. Currency – таблиця, яка зберігає інформацію про валюти.
4. Operation – таблиця, яка зберігає всі операції користувача.
5. Operation_category – таблиця, що зберігає користувацькі категорії.
6. Planned_operation – таблиця, що зберігає інформацію про планові операції.
7. Statistic – таблиця, що зберігає інформацію про статистику.

Опис кожної з таблиць бази даних наведений у таблицях 2.8 – 2.12.

Таблиця 2.8

Опис таблиці user

user		
Поле	Тип	Призначення
id	varcher	Ідентифікатор
email	varchar	Email користувача
password	varchar	Пароль користувача
firstName	varchar	Ім'я
lastName	varchar	Фамілія
pic	text	Email, що відображається

Таблиця 2.9

Опис таблиці account

account		
Поле	Тип	Призначення
id	varchar	Ідентифікатор
user_id	varchar	Ідентифікатор користувача
currency_id	varchar	Ідентифікатор валюти
name	varchar	Назва рахунку
balance	decimal	Баланс
pic	text	Аватар рахунку

Таблиця 2.10

Опис таблиці currency

currency		
Поле	Тип	Призначення
id	varchar	Ідентифікатор
code	varchar	Код валюти
rate	decimal	Курс
updated_at	date	Час оновлення курсу

Таблиця 2.11

Опис таблиці operation_category

operation_category		
Поле	Тип	Призначення
id	varchar	Ідентифікатор
user_id	varchar	Ідентифікатор користувача
name	varchar	Назва операції
type	tinyint	Тип операції
pic	text	Аватар категорії

Таблиця 2.12

Опис таблиці operation

operation		
Поле	Тип	Призначення
id	varchar	Ідентифікатор
operation_category_id	varchar	Ідентифікатор категорії
account_id	varchar	Ідентифікатор користувача
amount	decimal	Сума операції
comment	text	Коментар
date	date	Дата операції

Таблиця 2.13

Опис таблиці statistic

statistic		
Поле	Тип	Призначення
id	varchar	Ідентифікатор
account_id	varchar	Ідентифікатор рахунку
total_income_amount	float	Загальні прибутки
total_expense_amount	float	Загальні витрати
pure_income	float	Різниця
period	varchar	Період

Продовження таблиці 2.13

statistic		
Поле	Тип	Дата
date	date	Ідентифікатор

Таблиця 2.14

Опис таблиці planned_operation

statistic		
Поле	Тип	Призначення
id	varchar	Ідентифікатор
name	varchar	Назва
operation_category_id	varchar	Ідентифікатор категорії
account_id	varchar	Ідентифікатор рахунку
amount	decimal	Сума
frequency	int	Частота виконання

Дана бази даних, яка забезпечує високу ефективність, що є важливим аспектом в забезпеченні швидкодії та надійності системи. Також важливим є регулярне періодичне виконання завдань з обслуговування бази даних, таких як реорганізація індексів, оновлення статистики та очищення, це допоможе підтримувати високу ефективність роботи бази даних.

2.3 Проектування та реалізація алгоритмів роботи системи

Вебдодаток є інтуїтивно зрозумілим та простим у використанні для звичайних користувачів, це допомагає користувачам використовувати усі функції додатку без проблем та складностей (рис.2.4).

Авторизація користувача:

- після авторизації користувач потрапляє на головну сторінку додатку;
- на головній сторінці користувач може створити рахунок та обрати валюту для нього;
- також користувач на головній сторінці може додавати свої фінансові

операції;

- на головній сторінці користувач також може створити категорію витрат, або обрати вже наявну у списку;
- перейшовши на сторінку планових операцій користувач може додати операцію та період виконання для неї, а також її тип та категорію;
- На сторінці статистики користувач може переглянути статистику за різні періоди часу та побачити загальний прибуток та витрати



Рис. 2.4 Діаграма активності

2.4 Реалізація функціоналу вебдодатку

Розглянемо основний функціонал додатку для керування фінансами. Для взаємодії з базою даних MySQL, було використано Sequelize.

Sequelize - це об'єктно-реляційний мапер (ORM) для Node.js, який дозволяє взаємодіяти з реляційними базами даних, такими як MySQL, PostgreSQL, SQLite і MSSQL, за допомогою JavaScript-об'єктів. Sequelize дозволяє представляти таблиці бази даних як класи JavaScript (моделі). Це дозволяє розробникам працювати з даними у структурованому, об'єктно-орієнтованому стилі, що зменшує необхідність у прямому маніпулюванні SQL-запитами. Sequelize підтримує різні Системи Управління Базами Даних (СУБД), що дозволяє розробникам легко переносити додатки між різними СУБД або використовувати різні бази даних для різних середовищ (наприклад, розробка, тестування, продукція).

Для доступу до головної сторінки потрібно авторизуватися або зареєструватися. Для реалізації автентифікації використовується jswebtoken. Це дозволяє уникнути необхідності зберігати сесии на сервері, що робить систему більш масштабованою.

JSON Web Token (JWT) — це відкритий стандарт (RFC 7519), який визначає компактний і самодостатній спосіб передачі інформації між сторонами у вигляді JSON-об'єкта. Ця інформація може бути перевірена та довірена, оскільки вона підписана за допомогою криптографічних алгоритмів.

Фрагмент коду User.js:

```
module.exports.login = async (req, res) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ where: { email } });

    if (!user) {
      return res.status(401).json({ error: "Invalid email or password" });
    }

    const isMatch = bcryptjs.compareSync(password, user.password);

    if (!isMatch) {
      return res.status(401).json({ error: "Invalid email or password" });
    }

    const payload = { id: user.id, email: user.email };
    const token = jwt.sign(payload, process.env.SECRET || "secret", {
      expiresIn: "1h",
    });

    res.status(200).json({ token });
  } catch (error) {
    console.error("Error during login:", error);
  }
}
```

```

    res.status(500).json({ error: "Internal Server Error" });
  }
};

```

Реєстрація користувача:

1. Для реєстрації користувачу потрібно ввести свої дані(ім'я, прізвище, електронна пошта, пароль).
2. Після успішної реєстрації, сервер генерує JWT, який містить зашифровану інформацію про користувача.
3. JWT відправляється користувачеві та зберігається на клієнтській стороні додатку.

Переваги використання JWT для авторизації:

1. JWT дозволяє масштабувати додатки горизонтально, оскільки немає необхідності зберігати сесії на сервері. Кожен запит містить всю необхідну інформацію для аутентифікації, що робить додаток більш ефективним при обробці великої кількості користувачів.
2. JWT підписані за допомогою секретного ключа або асиметричної криптографії (наприклад, RSA), що гарантує цілісність і автентичність даних. Це означає, що дані в токени не можуть бути змінені без виявлення.
3. JWT може передаватися через URL, параметри POST або HTTP-заголовки, що робить його дуже гнучким і зручним для використання в різних сценаріях.

Після входу, користувач може створити рахунки, вказати їх назву(name), обрати валюту для них(currency_id), вказати баланс рахунку(balance) та обрати аватар для рахунку(pic).

Фрагмент коду user/controller/index.js для створення користувача:

```

module.exports.createAccount = async (req, res) => {
  try {
    const { name, currency_id, balance, pic } = req.body;
    const user_id = req.user.id;

    if (!(user_id && name && balance && currency_id)) {
      res.status(400).json({ error: "Bad Request" });
    }

    const account = await Account.create({

```

```

        user_id,

        currency_id,
        name,
        balance,
        pic
    });

    res.status(201).json(account);

} catch (e) {
    res.status(500).json({ error: e.message });
}
};

```

Для реалізації функції оновлення актуального курсу валют було використано бібліотеку `node – crones`, задля реалізації оновлення курсу кожні 10 хвилин.

Node – crones — це бібліотека для Node.js, яка дозволяє вам виконувати завдання за розкладом, використовуючи `cron`-вирази. Використання `cron`-завдань (`crons`) у Node.js має кілька переваг, особливо коли потрібно автоматизувати виконання завдань у певний час або через певні проміжки часу. Ось основні переваги:

1. Автоматизація завдань: `Cron`-завдання дозволяють автоматизувати регулярні завдання.
2. Гнучкість: `Cron`-завдання можуть бути налаштовані на виконання з будь-якою потрібною періодичністю: щохвилини, щогодини, щодня, щотижня, щомісяця тощо.
3. Паралельне виконання: Node.js за своєю природою підтримує асинхронне виконання коду, що дозволяє паралельно виконувати кілька `cron`-завдань без блокування основного потоку програми.
4. Інтеграція з іншими модулями: `Cron`-завдання можуть бути легко інтегровані з іншими модулями та бібліотеками Node.js, такими як модулі для роботи з базами даних, HTTP-клієнтами (наприклад, `axios`).

Лістинг `scheduler.js`:

```

cron.schedule("* / 10 * * * *", async () => {
    try {
        console.log("Running updateCurrencyRates...");
        await controller.updateCurrencyRates();
    }
});

```

					ІПЗ.КР.Б-121-24-ПЗ	36

```

    } catch (error) {
      throw error;
    }
  });

```

Завдяки функції вказаній вище кожні 10 хвилин викликається функція оновлення курсу. Дана функція реалізована завдяки бібліотеки `axios`. Функція виконує GET запит і повертає об'єкт з курсами валют, таких як доллар, євро та гривня.

axios – це популярна бібліотека для здійснення HTTP-запитів у Node.js та браузерах. Вона забезпечує простий і зрозумілий API для роботи з асинхронними запитами.

Лістинг `currencyService.js`:

```

require("dotenv").config();
const axios = require("axios");

const API_URL = "https://v6.exchangerate-
api.com/v6/25294766cffa4c9e9260642c/latest/USD";

const fetchCurrencyRates = async () => {
  try {
    const response = await axios.get(API_URL);

    return {
      USD: response.data.conversion_rates.USD,
      EUR: response.data.conversion_rates.EUR,
      UAH: response.data.conversion_rates.UAH
    };
  } catch (error) {
    throw error;
  }
};

module.exports = fetchCurrencyRates;

```

Після створення рахунків користувач може перемикатися між ними, для того щоб керувати фінансами одразу на декількох рахунках. Для цього було реалізовано функцію для отримання всіх рахунків користувача.

Фрагмент коду `account/controller/index.js`:

```

module.exports.getAccountList = async (req, res) => {
  try {
    const user_id = req.user.id;

    const accounts = await Account.findAll({
      where: {
        user_id: user_id
      }
    });
  }
};

```

```

        res.status(200).json(accounts);
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
};

```

Після того як у користувача є один або кілька рахунків, він може виконувати операцію для того щоб керувати фінансами. Для цього користувачу необхідно додати операцію, та обрати категорію для неї. Для створення категорії йому необхідно вказати назву(name), тип категорії(type) та обрати аватар для неї(pic).

Фрагмент коду account/controller/index.js:

```

module.exports.createOperationCategory = async (req, res) => {
    try {
        const { name, type, pic } = req.body;
        const user_id = req.user.id;

        if (!(name !== undefined && type !== undefined)) {
            return res.status(400).json({ error: "Bad Request" });
        }

        const operationCategory = await OperationCategory.create({
            user_id,
            name,
            type,
            pic,
        });

        res.status(201).json(operationCategory);
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
};

```

Після створення рахунку, категорії, користувач може виконувати операції на своєму рахунку. Всі операції відображаються в таблиці, також їх можна сортувати за категоріями та типом операцій задля зручності користувачів. Це все допоможе новим користувачам в швидкому розумінні додатку. Для створення операції користувачам потрібно бути автентифікованим вказати категорію(operation_category_id), суму(amount) та не обов'язковий коментар(comment).

Фрагмент коду operation/controller/index.js:

```

module.exports.createOperation = async (req, res) => {
    try {
        const { account_id, operation_category_id, amount, comment } = req.body;
        const user_id = req.user.id;

```

					ІПЗ.КР.Б-121-24-ПЗ	38

```

    if (!(account_id && operation_category_id && amount)) {
        return res.status(400).json({ error: "Bad Request" });
    }

    const account = await Account.findByIdPk(account_id);

    if (account.user_id !== user_id) {
        res.status(404).json({ error: "Account Not Found" });
    }

    const operation = await Operation.create({
        account_id,
        operation_category_id,
        amount,
        comment
    });

    res.status(201).json(operation);
} catch (e) {
    res.status(500).json({ error: e.message });
}
};

```

Для реалізації функції планових операцій знову було використано бібліотеку node – cron, було вирішено зробити чотири періоди для виконання планових операцій, такі як щоденні, щотижневі, щомісячні та щорічні.

Фрагмент коду PlannedOperationCrons.js:

```

const perDayCron = () => {
    cron.schedule("0 0 * * *", async () => {
        await executeAsyncMethods(dailyMethods);
    });
};

const perWeekCron = () => {
    cron.schedule("0 0 * * 0", async () => {
        await executeAsyncMethods(weeklyMethods);
    });
};

const perMonthCron = () => {
    cron.schedule("0 0 1 * *", async () => {
        await executeAsyncMethods(monthlyMethods);
    });
};

const perYearCron = () => {
    cron.schedule("0 0 1 1 *", async () => {
        await executeAsyncMethods(yearlyMethods);
    });
};

```

Завдяки кронам було зроблено основний функціонал для реалізації планових операцій. Це спростило структуру коду та оптимізувало його. Було реалізовано контролер для створення запланованої операції для конкретного

рахунку користувача. Він виконує кілька кроків: перевірка введених даних, перевірка доступу користувача до рахунку, створення запланованої операції та обробка помилок.

Фрагмент коду `planned_operation/controller/index.js`:

```
module.exports.createPlannedOperation= async (req, res) => {
  try {
    const { name, account_id, operation_category_id, amount, frequency } =
req.body;
    const user_id = req.user.id;

    if (!(name && account_id && operation_category_id && amount &&
frequency)) {
      return res.status(400).json({ error: "Bad Request" });
    }

    const account = await Account.findByPk(account_id);

    if (account.user_id !== user_id) {
      res.status(404).json({ error: "Account Not Found" });
    }

    const plannedOperation = await PlannedOperation.create({
      name,
      account_id,
      operation_category_id,
      amount,
      frequency
    });

    res.status(201).json(plannedOperation);

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};
```

Також в вебдодатку для зручності користувачів була реалізована функція статистики, на ній користувачі можуть на графіках отримати інформацію про їх витрати та доходи, за різні періоди часу. Для цього було створенно фільтр для запитів до бази даних, що дозволяє робити вибірку даних за різними періодами часу (день, тиждень, місяць, рік або весь час).

Фрагмент коду `dateFilter.js`, для вибірки за день:

```
const getDateFilter = (period, today, fieldName = "date") =>{
  let dateFilter = {};

  switch (period) {
    case 'day':
      dateFilter = {
        [fieldName]: {
          [Op.between]: [startOfDay(today), endOfDay(today)]
        }
      }
    }
};
```

```
};
break;
```

Фрагмент коду dateFilter.js, для вибірки за тиждень:

```
case 'week':
  dateFilter = {
    [fieldName]: {
      [Op.between]: [startOfWeek(today), endOfWeek(today)]
    }
  };
break;
```

Фрагмент коду dateFilter.js, для вибірки за місяць:

```
case 'month':
  dateFilter = {
    [fieldName]: {
      [Op.between]: [startOfMonth(today), endOfMonth(today)]
    }
  };
break;
```

Фрагмент коду dateFilter.js, для вибірки за рік:

```
case 'year':
  dateFilter = {
    [fieldName]: {
      [Op.between]: [startOfYear(today), endOfYear(today)]
    }
  };
break;
```

Фрагмент коду dateFilter.js, для вибірки за весь час:

```
case 'all':
  dateFilter = {
  }
break;
```

Для отримання статистики було реалізовано контролер для отримання статистичних даних за конкретним рахунком користувача. Він використовує асинхронний синтаксис для обробки запитів і виконання запитів до бази даних.

Фрагмент коду statistic/controller/index.js:

```
module.exports.getStatisticByAccount = async (req, res) => {
  try {
    const account_id = req.params.account_id;
    const {period} = req.query;
    const user_id = req.user.id;

    const account = await Account.findByPk(account_id);

    if(account.user_id !== user_id){
      res.status(404).send("Account Not Found!");
    }

    const statistic = await createStatisticForAccount(account, period);
```

```

res.status(201).json(statistic);

} catch (e) {
res.status(500).json({ error: e.message });
}

```

У підсумку було створено діаграму компонентів веб – додатку для керування фінансами(рис. 2.5) та визначено головні модулі додатку.

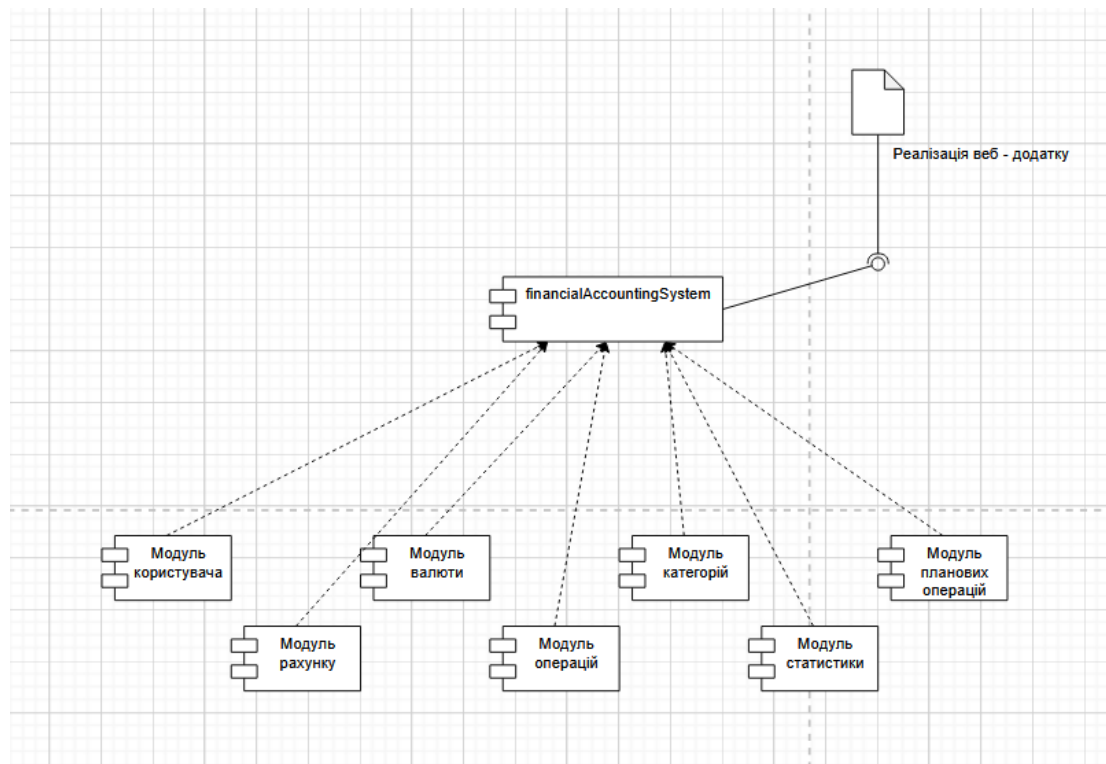


Рис. 2.16. Діаграма компонентів

На діаграмі визначені такі елементи:

- модуль користувача – модуль завдяки якому користувач може реєструватися та автентифікуватися у вебдодатку;
- модуль рахунку – завдяки цьому модулю користувачі можуть створювати свої рахунки та керувати ними;
- модуль валюти – забезпечує користувача актуальним курсом валют;
- модуль операцій – надає користувачу можливість користувачу вести керування рахунком, та виконувати операції на ньому;
- модуль категорій – забезпечує користувача можливістю створювати операції та категоризувати операції;

– модуль статистики – допомагає користувачам отримувати статистику за різні періоди часу та порівнювати її.

– модуль планових операцій – модуль, що надає можливість користувачам планувати фінансові операції, обирати категорії для них та їх тип.

Висновки до другого розділу.

Під час проектування вебдодатку для керування фінансами було проведено аналіз всіх сценаріїв використання програмного продукту відповідно до вимог програмного забезпечення. На основі цього аналізу була розроблена структура платформи, що включає моделювання структури проекту та взаємодії між його складовими частинами.

Також була спроектована база даних для зберігання на серверній частині додатку, таких як дані користувачів, категорії, операції та статистики.

Було описано алгоритм роботи з додатком та описані послідовні пункти варіантів його використання.

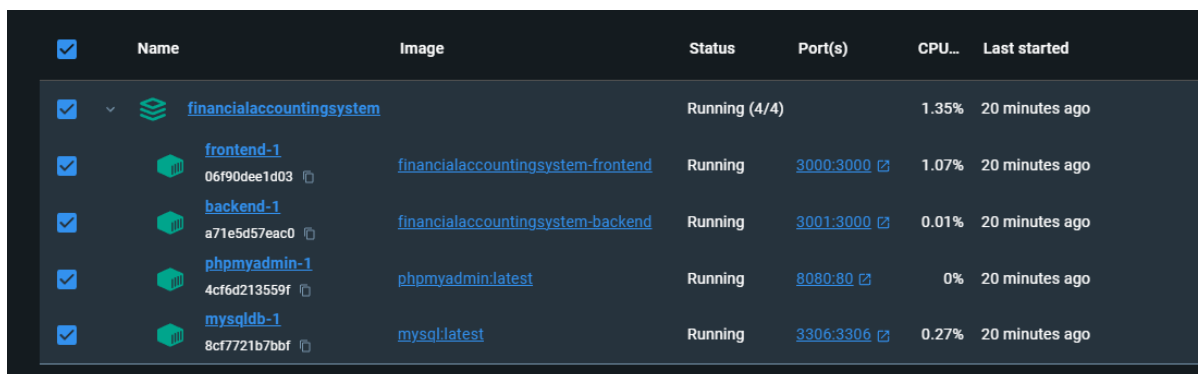
Також був наведений опис основного функціоналу додатку – це керування рахунками користувача, також були розглянуті основні функції по взаємодії користувача із вебдодатком.

РОЗДІЛ 3. ОПИС РОБОТИ З ПРОГРАМНИМ ДОДАТКОМ ТА ЙОГО ТЕСТУВАННЯ

3.1 Порядок встановлення та налаштування параметрів додатку

Для розгортання вебдодатку використовується Docker Compose.

Docker Compose — це інструмент для оркестрації багатоконтейнерних застосунків у середовищі Docker. Docker Compose надає простий інтерфейс для запуску та управління контейнерами з командного рядка. Docker Compose дозволяє вам визначати необхідність для кожного сервісу у вашому застосунку, наприклад, базу даних, серверний додаток та фронтенд, і налаштовувати їх взаємодію. Docker Compose підтримує можливість масштабування сервісів, що дозволяє вам визначати кількість контейнерів для кожного сервісу і автоматично керувати їх розгортанням.



☒	Name	Image	Status	Port(s)	CPU...	Last started
☒	 financialaccountingsystem		Running (4/4)		1.35%	20 minutes ago
☒	 frontend-1 06f90dee1d03	financialaccountingsystem-frontend	Running	3000:3000	1.07%	20 minutes ago
☒	 backend-1 a71e5d57eac0	financialaccountingsystem-backend	Running	3001:3000	0.01%	20 minutes ago
☒	 phpmyadmin-1 4cf6d213559f	phpmyadmin:latest	Running	8080:80	0%	20 minutes ago
☒	 mysqlbd-1 8cf7721b7bbf	mysql:latest	Running	3306:3306	0.27%	20 minutes ago

Рис. 3.1 Розгорнуті контейнери в вебдодатку керування фінансами

На рисунку 3.1 зображено розгорнуті контейнери вебдодатку для керування фінансами. Вебдодаток містить в своїй структурі 4 контейнера: frontend – 1, backend – 1, myphpadmin – 1, mysqlbd – 1.

Опис контейнерів:

1. Контейнер frontend – 1 розгорнутий на портах 3000:3000. Містить в собі програмне забезпечення, яке відповідає за відображення та взаємодію користувача з вебдодатком. В клієнтській частині використовується React.js для рендерингу та взаємодії з додатком. Також він містить в собі

бібліотеку Chart.js для реалізації графіків статистики та статичні ресурси для відображення користувацьких аватарів.

2. Контейнер backend – 1 розгорнутий на портах 3001:3000. Містить в собі конфігурацію серверу, конфігураційні файли для підключення до бази даних MySQL для зберігання користувацької інформації. Має логіку, яка визначає, які дії повинні бути виконані при отриманні запитів від клієнтів, яка частина даних повинна бути взята з бази даних або інших сервісів, які повинні бути виконані для відповіді на запити. А також логування задля моніторингу діяльності та працездатності серверу.
3. Контейнер phpmyadmin – 1 розгорнутий на портах 8080:80. На цьому порту розгорнутий веб – інтерфейс для керування та адміністрування базою даних MySQL. Це було реалізовано задля зручної взаємодії з базою даних без необхідності використання командного рядку.
4. Контейнер mysql – 1 розгорнутий на портах 3306:3306. На даному порту запущений сервер бази даних MySQL. Ця база даних містить в собі усі дані про користувача, його рахунки, операції та інші.

Для полегшеного розгортання системи було реалізовано файл rebuilt&run.bat. Цей файл потрібен для розгортання контейнерів в Docker, завантаження усіх пакетів, запуском лише одного файлу без окремого прописування команд в командному рядку.

Лістинг rebuilt&run.sh:

```
#!/bin/bash
echo "stop and remove containers"
docker rm -f $(docker ps -aq)

echo "run containers"
docker compose build
docker compose up -d
```

Лістинг rebuilt&run.ps1:

```
Write-Host "stop and remove containers"
Invoke-Expression -Command "docker rm -f $(docker ps -aq) "

Write-Host "run containers"
Invoke-Expression -Command "docker compose build"
Invoke-Expression -Command "docker-compose up -d"
```

Read-Host "Press Enter to continue..."

Лістинг rebuild&run.bat:

```
powershell.exe -ExecutionPolicy Bypass -File ".\rebuild&run.ps1"
```

3.2 Структура інтерфейсу вебдодатку та порядок роботи

Для того, щоб розпочати роботу з додатком необхідно мати девайс із інтернет – з’єднанням та браузером. Інтерфейс вебдодатку адаптивний, тому ним можна користуватись як зі смартфона так і з персонального ком’ютера не відчуючи дискомфорт.

Коли користувач відкриває додаток, він опиняється на головному екрані, який пропонує користувачу автентифікуватись(рис. 3.2).

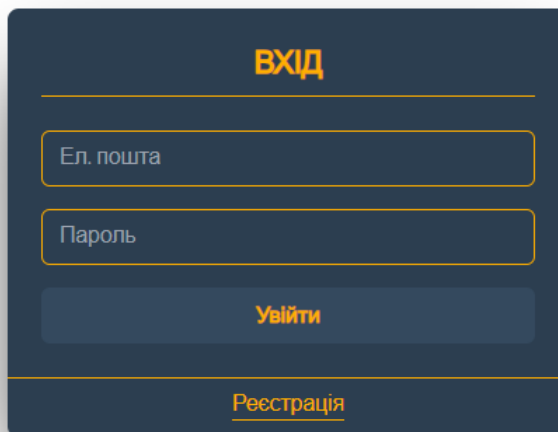


Рис. 3.2 Форма для автентифікації в системі

Якщо користувач не зареєстрований, він може перейти на вікно реєстрації(рис 3.3).

РЕЄСТРАЦІЯ

Ім'я

Прізвище

Ел. пошта

Пароль

Підтвердіть пароль

Зареєструватися

[Вхід](#)

Рис. 3.3 Форма для реєстрації в системі

Після успішної реєстрації система сповіщає про це користувача повідомленням(рис. 3.4).

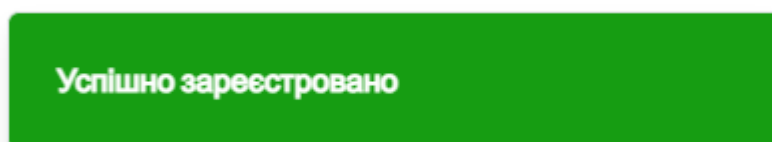


Рис. 3.4 Повідомлення про успішну реєстрацію

Коли користувач автентифікується в системі, він перейде на головну сторінку додатку, де відображаються рахунки, операції, сторінка планових операцій та статистики(рис. 3.5).

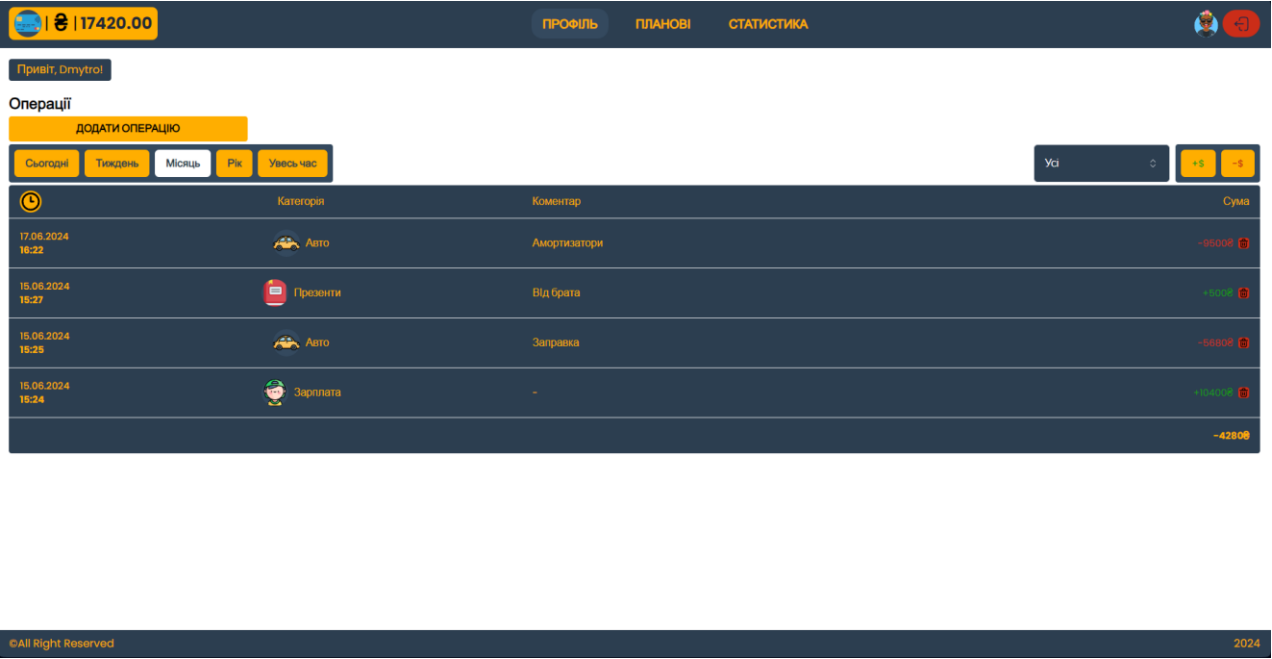


Рис. 3.6 Головна сторінка додатку

На головній сторінці користувач може створювати рахунки, операції та категорії витрат або доходів(рис. 3.7 – рис. 3.9).

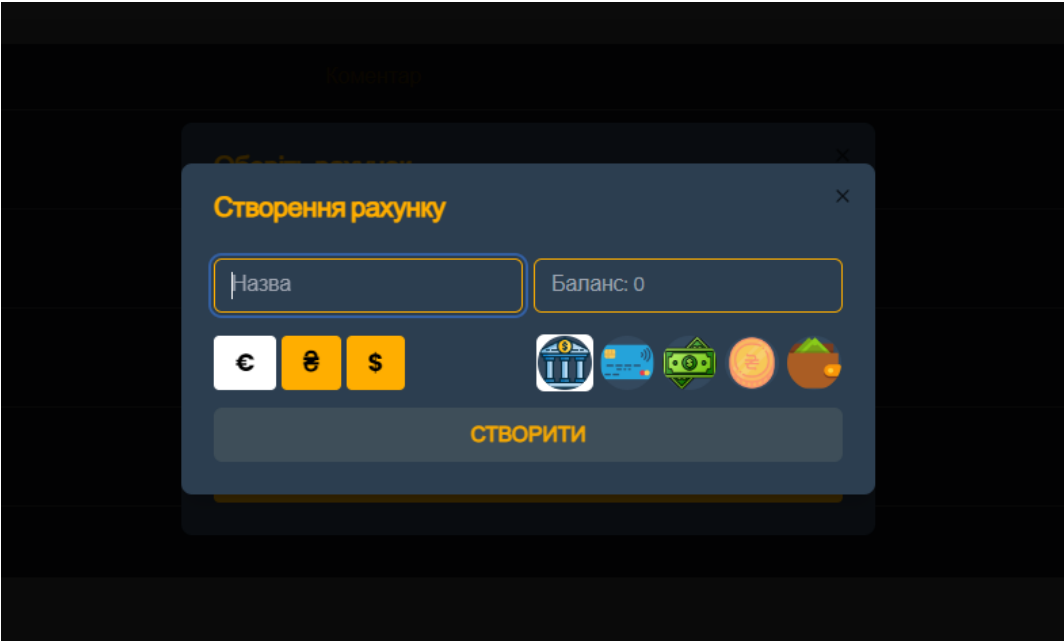


Рис. 3.7 Форма створення рахунку

Створення операції

Сума

Коментар(не обов'язково)

Зарплата



Презенти



Транспорт



Перекази



Котик



Квартира



Авто



Квартира



Книги



Подарунки



СТВОРИТИ

Додати категорію

Рис. 3.8 Форма створення операції

Створення категорії

Назва

+

-



СТВОРИТИ

Рис. 3.9 Форма створення категорії

Далі користувач може перейти на сторінку планових операцій(рис. 3.10), де він може створити планову операцію, обрати її тип, суму та частоту виконання(рис. 3.11).

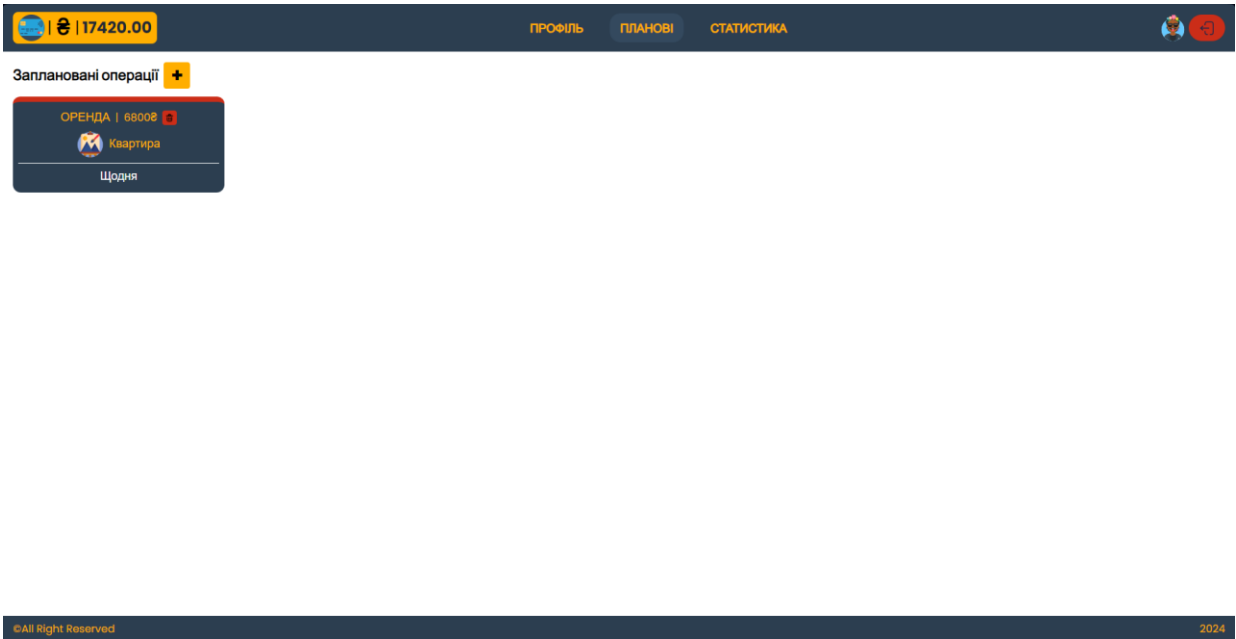


Рис. 3.10 Екран планових операцій

Створення планової операції

Назва

Сума

Щодня

Щотижня

Щомісяця

Щороку

Зарплата

Презенти

Транспорт

Перекази

Котик

Квартира

Авто

Квартира

Книги

Подарунки

СТВОРИТИ

Додати категорію

Рис. 3.11 Форма створення планової операції

Далі користувач може перейти на сторінку статистики(рис. 3.12) та дослідити її. Статистика представлення за допомогою інтуїтивно зрозумілого графіку та на ній присутні загальні витрати та доходи, а також порівняння з попереднім періодом.

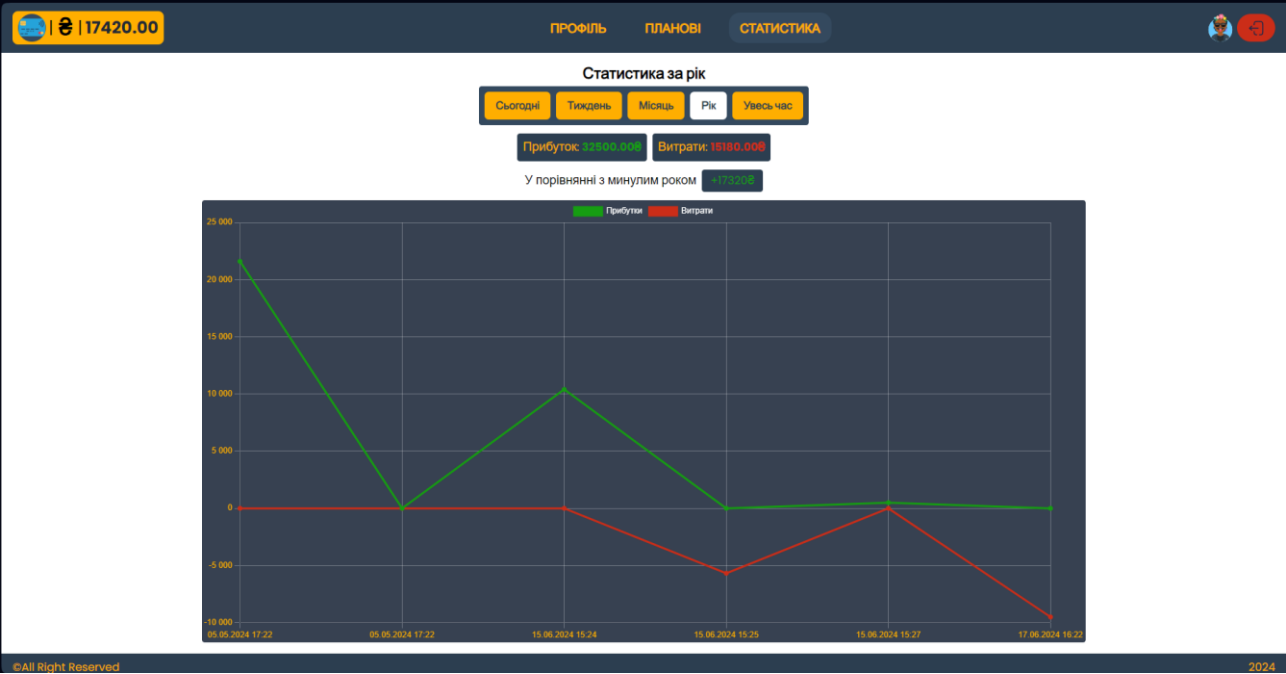


Рис. 3.12 Екран відображення статистики

3.3 Тестування роботи вебдодатку

Для тестування вебдодатку було обрано мануальне тестування для того щоб перевірити правильність роботи вебдодатку.

1. Перевірка реєстрація користувача.

Тестові випадок:

- заповнення форми правильними даними(ім'я, прізвище, електронна адреса, пароль).

Очікуваний результат:

- успішна реєстрація.

Результат тестування:

- успішна реєстрація

2. Автентифікація в додатку.

Тестовий випадок:

- заповнення форми входу вже зареєстрованими даними(логін та пароль);

Очікуваний результат:

- успішна автентифікація;

Результат тестування:

- успішна автентифікація;

3. Валідація форм.

Тестовий випадок:

- заповнення форми реєстрації не коректними даними (не правильний формат електронної форми, не співпадаючі паролі);
- заповнення форми автентифікації не коректними даними (хибна пошта або пароль);

Очікуваний результат

- відображення про помилку реєстрації;
- відображення про помилку автентифікації;

Результат тестування:

- відображення про помилку реєстрації;
- відображення про помилку автентифікації;

4. Перевірка обов'язкових полів форми реєстрації.

Тестовий випадок:

- відправити форму без заповнення обов'язкових полів;
- заповнити лише частину обов'язкових полів та спробувати відправити форму;

Очікуваний результат:

- відображення про помилку реєстрації;

Результат тестування:

- відображення про помилку реєстрації;

5. Перевірка унікальності:

Тестовий випадок:

- спробувати зареєструватися з email або ім'ям користувача, яке вже існує в системі;

Очікуваний результат:

- відображення про помилку реєстрації;

Результат тестування:

- відображення про помилку реєстрації;

6. Тестування інтерфейсу:

Тестовий випадок:

- натискання на кожну кнопку на сторінці (створення рахунку, категорії, операції, перехід на сторінку статистики та планових операцій);

Очікуваний результат:

- успішний перехід по сторінках додатку;

Результат тестування:

- успішний перехід по сторінках додатку;

Висновки до третього розділу

У даному розділі було детально розглянуто процес роботи з програмним додатком, а також проведено мануальне тестування його функціональних можливостей. Було описано основні аспекти взаємодії користувача з додатком, починаючи з розгортання контейнерів, і закінчуючи виконанням конкретних задач, передбачених функціоналом додатка.

Особливу увагу було приділено формі реєстрації користувачів, оскільки вона є критично важливою частиною будь-якого сучасного вебдодатку. Проведені тестування охопили різні сценарії використання, включаючи перевірку обов'язкових полів, валідацію форматів введених даних. Результати тестування показали, що додаток відповідає заявленим вимогам та здатен ефективно обробляти користувацькі дані.

Узагальнюючи результати, можна зазначити, що програмний додаток успішно пройшов усі етапи тестування та готовий до використання кінцевими

					ІПЗ.КР.Б-121-24-ПЗ	
						53

користувачами. Він забезпечує необхідну функціональність, відповідає сучасним стандартам якості та безпеки, а також надає користувачам зручний та інтуїтивно зрозумілий інтерфейс для виконання необхідних дій

ВИСНОВКИ

У процесі виконання бакалаврської кваліфікаційної роботи було спроектовано та реалізовано вебдодаток для управління фінансами. Було проведено аналіз предметної області та порівняння різних платформ для керування бюджетом. На основі цього аналізу були визначені основні вимоги до власного вебдодатку. Під час дослідження було розроблено архітектуру, обрано підходи та основні технології для реалізації вебдодатку. Проектування платформи включало аналіз варіантів використання програмного продукту та проектування структури, включаючи моделювання взаємодії між складовими частинами. Також було спроектовано базу даних для зберігання клієнтської інформації.

Було описано алгоритм його роботи та варіанти його використання. Розглянуто основний функціонал, що включає в себе реєстрацію користувача, автентифікацію, створення рахунків, операцій як звичайних так і планових, категорій та перегляд статистики.

Також було детально описано розгортання вебдодатку для керування особистими фінансами, описано взаємодію структурних елементів системи та наведено детальний опис інтерфейсу користувача. Також продемонстровано роботу вебдодатку та можливості взаємодії між ним та користувачем. Було проведено мануальне тестування для перевірки працездатності додатку та виявлення помилок в його роботі.

Вебдодаток має інтуїтивно зрозумілий інтерфейс, що надає можливість користувачам без проблем користуватись ним, створювати власні рахунки, категорії, операції та переглядати статистику.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. You Need a Budget [Електронний ресурс] – Режим доступу до ресурсу: <https://www.ynab.com/>
2. Personal Capital [Електронний ресурс] – Режим доступу до ресурсу: <https://www.empower.com/>
3. Pocket Guard [Електронний ресурс] – Режим доступу до ресурсу: <https://pocketguard.com/>
4. Money Lover [Електронний ресурс] – Режим доступу до ресурсу: <https://note.moneylover.me>
5. Goodbudget [Електронний ресурс] – Режим доступу до ресурсу: <https://goodbudget.com/>
6. Node.js [Електронний ресурс] – Режим доступу до ресурсу: <https://nodejs.org/en>
7. React [Електронний ресурс] - Режим доступу до ресурсу: <https://react.dev/>
8. MySQL [Електронний ресурс] - Режим доступу до ресурсу: <https://www.mysql.com/>
9. TailwindCSS [Електронний ресурс] – Режим доступу до ресурсу: <https://tailwindcss.com/>
10. Draw.IO [Електронний ресурс] – Режим доступу до ресурсу: <https://app.diagrams.net/>
11. DBDiagram [Електронний ресурс] – Режим доступу до ресурсу: <https://dbdiagram.io/home>
12. Sequelize [Електронний ресурс] – Режим доступу до ресурсу: <https://sequelize.org/>
13. JWT [Електронний ресурс] - Режим доступу до ресурсу: <https://jwt.io/>
14. Node - cron [Електронний ресурс] - Режим доступу до ресурсу: <https://www.npmjs.com/package/node-cron>
15. Axios [Електронний ресурс] - Режим доступу до ресурсу: <https://axios-http.com/uk/docs/intro>

- 16.Docker Compose [Электронный ресурс] - Режим доступа до ресурсу:
<https://docs.docker.com/>
- 17.Chart.js [Электронный ресурс] - Режим доступа до ресурсу:
<https://www.chartjs.org/docs/latest/>
- 18.Passport.js [Электронный ресурс] - Режим доступа до ресурсу:
<https://www.passportjs.org/>
- 19.useState [Электронный ресурс] - Режим доступа до ресурсу:
<https://react.dev/reference/react/useState>
- 20.Hooks [Электронный ресурс] - Режим доступа до ресурсу:
<https://legacy.reactjs.org/docs/hooks-intro.html>
- 21.Forms [Электронный ресурс] - Режим доступа до ресурсу:
<https://legacy.reactjs.org/docs/forms.html>
- 22.Render Props [Электронный ресурс] - Режим доступа до ресурсу:
<https://legacy.reactjs.org/docs/render-props.html>

ДОДАТКИ

					ІІЗ.КР.Б-121-24-ІІЗ	
						58

Лістинг файлу user/controller/index.js

```
const Account = require("../../models/Account");

module.exports.createAccount = async (req, res) => {
  try {
    const { name, currency_id, balance, pic } = req.body;
    const user_id = req.user.id;

    if (!(user_id && name && balance && currency_id)) {
      res.status(400).json({ error: "Bad Request" });
    }

    const account = await Account.create({
      user_id,
      currency_id,
      name,
      balance,
      pic
    });

    res.status(201).json(account);

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.getAccountList = async (req, res) => {
  try {
    const user_id = req.user.id;

    const accounts = await Account.findAll({
      where: {
        user_id: user_id
      }
    });

    res.status(200).json(accounts);

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.getAccountById = async (req, res) => {
  try {
    const accountId = req.params.id;
    const userId = req.user.id;

    const account = await Account.findPk(accountId);

    if (account.user_id !== userId) {
      res.status(404).json({ error: "Not found" });
    }

    res.status(200).json(account);

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};
```

```

    }
  };

module.exports.updateAccount = async (req, res) => {
  try {
    const accountId = req.params.id;
    const userId = req.user.id;

    const account = await Account.findByIdPk(accountId);

    if (account.user_id !== userId) {
      res.status(404).json({ error: "Not found" });
    }

    if (!account) {
      res.status(404).json({ error: "Not found" });
    }

    const { name, currency_id, balance, pic } = req.body;

    account.name = name ? name : account.name;
    account.currency_id = currency_id ? currency_id : account.currency_id;
    account.balance = balance ? balance : account.balance;
    account.pic = pic ? pic : account.pic;

    await account.save();

    res.status(200).json(account);

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.deleteAccount = async (req, res) => {
  try {
    const accountId = req.params.id;
    const userId = req.user.id;

    const account = await Account.findByIdPk(accountId);

    if (account.user_id !== userId) {
      res.status(404).json({ error: "Not found" });
    }

    if (!account) {
      res.status(404).json({ error: "Not found" });
    }

    await account.destroy();
    res.status(204).send();

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

```

Лістинг файлу operation/controller/index.js

```

const Operation = require("../../models/Operation");
const Account = require("../../models/Account");
const OperationCategory = require("../../models/OperationCategory");
const getDateFilter = require("../../utils/dateFilter");

```

```

module.exports.createOperation = async (req, res) => {
  try {
    const { account_id, operation_category_id, amount, comment } = req.body;
    const user_id = req.user.id;

    if (!(account_id && operation_category_id && amount)) {
      return res.status(400).json({ error: "Bad Request" });
    }

    const account = await Account.findByIdPk(account_id);

    if (account.user_id !== user_id) {
      res.status(404).json({ error: "Account Not Found" });
    }

    const operation = await Operation.create({
      account_id,
      operation_category_id,
      amount,
      comment
    });

    res.status(201).json(operation);
  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.getOperationById = async (req, res) => {
  try {
    const operationId = req.params.id;
    const userId = req.user.id;

    const operation = await Operation.findByIdPk(operationId);

    if (!operation) {
      return res.status(404).json({ error: "Operation Not Found" });
    }

    const account = await Account.findByIdPk(operation.account_id);

    if (account.user_id !== userId) {
      return res.status(404).json({ error: "Operation Not Found" });
    }

    res.status(200).json(operation);
  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.getOperationsByAccount = async (req, res) => {
  try {
    const accountId = req.params.accountId;
    const userId = req.user.id;

    const { period, category_id } = req.query;

    const account = await Account.findByIdPk(accountId);
    if (!account || account.user_id !== userId) {
      return res.status(404).json({ error: "Account Not Found" });
    }
  }
};

```

```

const category = await OperationCategory.findByPk(category_id);

if (category?.user_id !== userId && category?.user_id !== null &&
category?.user_id !== undefined) {
  return res.status(404).json({ error: "Category Not Found" });
}

const dateFilter = getDateFilter(period, new Date());

const categoryFilter =
category_id?{operation_category_id:category_id}:{};

const operations = await Operation.findAll({
  where: {
    account_id: accountId,
    ...categoryFilter,
    ...dateFilter
  },
  include: [
    {
      model: OperationCategory,
      as: "operation_category"
    }
  ]
});

res.status(200).json(operations);
} catch (e) {
  res.status(500).json({ error: e.message });
}
};

module.exports.getIncomesByAccount = async (req, res) => {
  try {
    const accountId = req.params.id;
    const userId = req.user.id;

    const { period } = req.query;

    const account = await Account.findByPk(accountId);

    if (!account) {
      return res.status(404).json({ error: "Account Not Found" });
    }

    if (account.user_id !== userId) {
      return res.status(404).json({ error: "Account Not Found" });
    }

    const dateFilter = getDateFilter(period, new Date());

    const operations = await Operation.findAll({
      where: {
        account_id: accountId,
        ...dateFilter
      },
      include: [
        {
          model: OperationCategory,
          as: "operation_category",
          where: {
            type: 1

```

```

        }

    }

    });

    res.status(200).json(operations);
} catch (e) {
    res.status(500).json({ error: e.message });
}
};

module.exports.getExpensesByAccount = async (req, res) => {
    try {
        const accountId = req.params.id;
        const userId = req.user.id;

        const { period } = req.query;

        const account = await Account.findByPk(accountId);

        if (!account) {
            return res.status(404).json({ error: "Account Not Found" });
        }

        if (account.user_id !== userId) {
            return res.status(404).json({ error: "Account Not Found" });
        }

        const dateFilter = getDateFilter(period, new Date());

        const operations = await Operation.findAll({
            where: {
                account_id: accountId,
                ...dateFilter
            },
            include: [
                {
                    model: OperationCategory,
                    as: "operation_category",
                    where: {
                        type: 0
                    }
                }
            ]
        });

        res.status(200).json(operations);
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
};

module.exports.updateOperation = async (req, res) => {
    try {
        const operationId = req.params.id;
        const userId = req.user.id;

        const operation = await Operation.findByPk(operationId);

        if (!operation) {
            return res.status(404).json({ error: "Operation Not Found" });
        }
    }
};

```

```

    const account = await Account.findByPk(operation.account_id);

    if (account.user_id !== userId) {
        return res.status(404).json({ error: "Operation Not Found" });
    }

    const { account_id, operation_category_id, amount, comment } = req.body;

    operation.account_id = account_id || operation.account_id;
    operation.operation_category_id = operation_category_id ||
operation.operation_category_id;
    operation.amount = amount || operation.amount;
    operation.comment = comment || operation.comment;
    operation.date = new Date();

    await operation.save();

    res.status(200).json(operation);
} catch (e) {
    res.status(500).json({ error: e.message });
}
};

module.exports.deleteOperation = async (req, res) => {
    try {
        const operationId = req.params.id;
        const userId = req.user.id;

        const operation = await Operation.findByPk(operationId);

        if (!operation) {
            return res.status(404).json({ error: "Operation Not Found" });
        }

        const account = await Account.findByPk(operation.account_id);

        if (account.user_id !== userId) {
            return res.status(404).json({ error: "Operation Not Found" });
        }

        await operation.destroy();
        res.status(204).send();
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
};

```

Лістинг файлу operation_category/controller/index.js

```

const OperationCategory = require("../../models/OperationCategory");
const User = require("../../models/User");
const { Op } = require("sequelize");

/** todo delete this */
module.exports.createDefaultOperationCategory = async (req, res) =>{
    try {
        const { name, type, pic } = req.body;

        if (!(name!== undefined && type!== undefined)) {
            return res.status(400).json({ error: "Bad Request" });
        }
    }

```

```

const operationCategory = await OperationCategory.create({
    name,
    type,
    pic,
});

    res.status(201).json(operationCategory);
} catch (e) {
    res.status(500).json({ error: e.message });
}
}
/** end of deleting */

module.exports.createOperationCategory = async (req, res) => {
    try {
        const { name, type, pic } = req.body;
        const user_id = req.user.id;

        if (!(name !== undefined && type !== undefined)) {
            return res.status(400).json({ error: "Bad Request" });
        }

        const operationCategory = await OperationCategory.create({
            user_id,
            name,
            type,
            pic,
        });

        res.status(201).json(operationCategory);
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
};

module.exports.getOperationCategoryById = async (req, res) => {
    try {
        const operationCategoryId = req.params.id;
        const userId = req.user.id;

        const operationCategory = await
OperationCategory.findByPk(operationCategoryId);

        if (!operationCategory) {
            return res.status(404).json({ error: "Operation category not found"
});
        }

        if (operationCategory.user_id !== userId) {
            return res.status(404).json({ error: "Operation Category Not Found"
});
        }

        res.status(200).json(operationCategory);
    } catch (e) {
        res.status(500).json({ error: e.message });
    }
};

module.exports.getOperationCategoriesList = async (req, res) => {
    try {
        const userId = req.user.id;

```

```

const operationCategories = await OperationCategory.findAll({
  where: {
    [Op.or]: [
      { user_id: userId },
      { user_id: null }
    ]
  }
});

res.status(200).json(operationCategories);
} catch (e) {
  res.status(500).json({ error: e.message });
}
};

module.exports.updateOperationCategory = async (req, res) => {
  try {
    const operationCategoryId = req.params.id;
    const userId = req.user.id;

    const operationCategory = await
OperationCategory.findByPk(operationCategoryId);

    if (!operationCategory) {
      return res.status(404).json({ error: "Operation category not found"
});
    }

    if (operationCategory.user_id !== userId) {
      return res.status(404).json({ error: "Operation Category Not Found"
});
    }

    const { name, type, pic } = req.body;

    operationCategory.name = name || operationCategory.name;
    operationCategory.type = type || operationCategory.type;
    operationCategory.pic = pic || operationCategory.pic;

    await operationCategory.save();

    res.status(200).json(operationCategory);
  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.deleteOperationCategory = async (req, res) => {
  try {
    const operationCategoryId = req.params.id;
    const userId = req.user.id;

    const operationCategory = await
OperationCategory.findByPk(operationCategoryId);

    if (!operationCategory) {
      return res.status(404).json({ error: "Operation category not found"
});
    }

    if (operationCategory.user_id !== userId) {
      return res.status(404).json({ error: "Operation Category Not Found"
});
    }
  }
};

```

```

    }

    await operationCategory.destroy();
    res.status(204).send();
  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

```

Лістинг файлу statistic/controller/index.js

```

const { Statistic, createStatisticForAccount, createPrevPeriodStatistic } =
require("../.../models/Statistic");
const Account = require("../.../models/Account");

module.exports.getStatisticByAccount = async (req, res) => {
  try {
    const account_id = req.params.account_id;
    const {period} = req.query;
    const user_id = req.user.id;

    const account = await Account.findByPk(account_id);

    if(account.user_id !== user_id){
      res.status(404).send("Account Not Found!");
    }

    const statistic = await createStatisticForAccount(account, period);

    res.status(201).json(statistic);

  } catch (e) {
    res.status(500).json({ error: e.message });
  }
};

module.exports.getStatDiffWithPrevPeriod = async (req, res) => {
  try{
    const account_id = req.params.account_id;
    const {period} = req.query;
    const user_id = req.user.id;

    const account = await Account.findByPk(account_id);

    if(account.user_id !== user_id){
      res.status(404).send("Account Not Found!");
    }

    const statistic = await createStatisticForAccount(account, period);
    const prevPeriodStat = await createPrevPeriodStatistic(account, period);

    let total_income_diff_percent, total_expense_diff_percent,
    pure_income_percent;

    if(prevPeriodStat.total_income_amount !== 0) {
      total_income_diff_percent = statistic.total_income_amount * 100 /
prevPeriodStat.total_income_amount;
    } else{
      total_income_diff_percent = statistic.total_income_amount * 100;
    }
    if(prevPeriodStat.total_expense_amount !== 0) {
      total_expense_diff_percent = statistic.total_expense_amount * 100 /
prevPeriodStat.total_expense_amount;
    } else{

```

```

        total_expense_diff_percent = statistic.total_expense_amount * 100;
    }
    if (prevPeriodStat.pure_income != 0) {
        pure_income_percent = statistic.pure_income * 100 /
prevPeriodStat.pure_income;

    } else {
        pure_income_percent = statistic.pure_income * 100;
    }

    res.status(200).json({
        account_id: account_id,
        total_income_diff_amount: statistic.total_income_amount -
prevPeriodStat.total_income_amount,
        total_income_diff_percent: total_income_diff_percent.toFixed(2),
        total_expense_diff_amount: statistic.total_expense_amount -
prevPeriodStat.total_expense_amount,
        total_expense_diff_percent: total_expense_diff_percent.toFixed(2),
        pure_diff_income: statistic.pure_income -
prevPeriodStat.total_expense_amount,
        pure_income_diff_percent: pure_income_percent.toFixed(2),
        period: period
    });

} catch (e) {
    res.status(500).json({error: e.message});
}
}

```

Лістинг файлу PlanedOperationCrones

```

const cron = require("node-cron");
const { removeMethodFromArray } = require("../utils/removeMethodFromArray");

const dailyMethods = [];
const weeklyMethods = [];
const monthlyMethods = [];
const yearlyMethods = [];

const subscribeToCron = (method, frequency) => {
    switch (frequency) {
        case 1:
            dailyMethods.push(method);
            break;
        case 10:
            weeklyMethods.push(method);
            break;
        case 100:
            monthlyMethods.push(method);
            break;
        case 1000:
            yearlyMethods.push(method);
            break;
        default:
            throw new Error("Invalid frequency");
    }
};

const unsubscribeFromCron = (method, frequency) => {
    switch (frequency) {

```

```

        case 1:
            removeMethodFromArray(dailyMethods, method);
            break;
        case 10:
            removeMethodFromArray(weeklyMethods, method);
            break;
        case 100:
            removeMethodFromArray(monthlyMethods, method);
            break;

        case 1000:
            removeMethodFromArray(yearlyMethods, method);
            break;
        default:
            throw new Error("Invalid frequency");
    }
};

const executeAsyncMethods = async (methods) => {
    for (const method of methods) {
        try {
            await method();
        } catch (error) {
            console.error(`Error executing method: ${error.message}`);
        }
    }
};

const perDayCron = () => {
    cron.schedule("0 0 * * *", async () => {
        await executeAsyncMethods(dailyMethods);
    });
};

const perWeekCron = () => {
    cron.schedule("0 0 * * 0", async () => {
        await executeAsyncMethods(weeklyMethods);
    });
};

const perMonthCron = () => {
    cron.schedule("0 0 1 * *", async () => {
        await executeAsyncMethods(monthlyMethods);
    });
};

const perYearCron = () => {
    cron.schedule("0 0 1 1 *", async () => {
        await executeAsyncMethods(yearlyMethods);
    });
};

const startPlannedOperationCrons = () => {
    perDayCron();
    perWeekCron();
    perMonthCron();
    perYearCron();
};

module.exports = { subscribeToCron, startPlannedOperationCrons,
unsubscribeFromCron };

```

Лістинг файлу currencyService.js

```
require("dotenv").config();
const axios = require("axios");

const API_URL = "https://v6.exchangerate-
api.com/v6/25294766cffa4c9e9260642c/latest/USD";

const fetchCurrencyRates = async () => {
  try {
    const response = await axios.get(API_URL);

    return {
      USD: response.data.conversion_rates.USD,
      EUR: response.data.conversion_rates.EUR,
      UAH: response.data.conversion_rates.UAH
    };
  } catch (error) {
    throw error;
  }
};

module.exports = fetchCurrencyRates;
```

					ІПЗ.КР.Б-121-24-ІЗ	
						70