МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Веб застосунок трекеру витрат з банківською інтеграцією та статистикою**

Виконав: студент 4 курсу, групи 6.1211-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
В. А. Мірзоян
(ініціали та прізвище)
Керівник асисент каф. ЕІС та ПЗ О. О. Жуков
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дісітел»
П. О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти _____перший (бакалавський)_____

Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)

Освітня програма ___Програмне забезпечення систем____
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т. В. Критська
“_01_” __березня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

_____Мірзояну Владиславу Арсеновичу_____
(прізвище, ім’я, по батькові)

1.Тема роботи Веб застосунок трекеру витрат з банківською інтеграцією та статистикою

керівник роботи _____Жуков Олександр Олексійович
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від 27.12.2024 № 2029-с_____

2. Строк подання студентом кваліфікаційної роботи_____02.06.2025_____

3. Вихідні дані бакалаврської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми створення веб застосунків;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- тестування програмної системи та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
_____слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.03.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз предметної області	01.03-10.03.25	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.03-12.03.25	виконано
3	Аналіз існуючих методів рішення	13.03-14.03.25	виконано
4	Аналіз методів розробки веб застосунків	15.03-20.03.25	виконано
5	Аналіз засобів розробки веб застосунків	21.03-26.03.25	виконано
6	Узгодження подальших дій з науковим керівником	27.03-28.03.25	виконано
7	Проєктування системи веб застосунка	29.03-05.04.25	виконано
8	Програмна реалізація веб застосунка	06.04-16.04.25	виконано
9	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	17.04-19.04.25	виконано
10	Реалізація користувацького інтерфейсу для веб застосунку	20.04-01.05.25	виконано
11	Тестування застосунку	02.05-7.05.25	виконано
12	Оформлення звіту	8.05-15.05.25	виконано
13	Оформлення презентації.	16.05-20.05.25	виконано

Студент _____ В. А. Мірзоян
(підпис) (прізвище та ініціали)

Керівник роботи _____ О. О. Жуков
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І. А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок – 79

Рисунки – 18

Таблиць – 1

Джерела – 11

Мірзоян В.А. Веб застосунок трекеру витрат з банківською інтеграцією та статистикою: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник О. О. Жуков. Запоріжжя : ЗНУ, 2025. 79 с.

Метою даного проєкту є розробка веб-додатку для відстеження витрат з банківською інтеграцією та статистикою. Проєкт побудований з використанням платформи ASP.NET Core та використовує API Monobank для автоматизованого отримання даних про транзакції.

Ключові функції програми включають категоризацію витрат, фільтрацію та сортування, візуалізацію статистики за допомогою графіків та зведень, а також можливість змінювати категорії. Додаток має адаптивний користувацький інтерфейс для забезпечення доступності як на настільних, так і на мобільних пристроях.

Робота досліджує проблему точної класифікації витрат на основі даних банківських транзакцій і демонструє, як технології ASP.NET Core можуть бути застосовані для створення масштабованих і зручних фінансових веб-додатків.

Ключові слова: *трекер витрат, банківська інтеграція, Monobank API, веб-додаток, фінансова статистика, ASP.NET Core, особисті фінанси, інтерфейс користувача*.

ABSTRACT

Pages – 79

Figures – 18

Tables – 1

Sources – 11

Mirzoian V.A. Web App to the Transaction Tracker with Banking Integration and Statistics: bachelor's thesis in specialty 121 “Software Engineering” / Scientific Supervisor O.O. Zhukov. Zaporizhzhia: ZNU, 2025. 79 p.

The aim of this project is to develop a web application for tracking expenses with banking integration and statistics. The project is built using the ASP.NET Core platform and uses the Monobank API to automate the receipt of transaction data.

Key features of the application include categorization of expenses, filtering and sorting, visualization of statistics using graphs and summaries, and the ability to change categories. The application has a responsive user interface to ensure accessibility on both desktop and mobile devices.

The paper explores the problem of accurate expense classification based on bank transaction data and demonstrates how ASP.NET Core technologies can be used to create scalable and user-friendly financial web applications.

Keywords: *expense tracker, banking integration, Monobank API, web application, financial statistics, ASP.NET Core, personal finance, user interface*.

ЗМІСТ

ВСТУП 7
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 14
1.1 Огляд літературних джерел 14
1.2 Огляд існуючих рішень 15
1.2.1 Власна аналітика витрат від Monobank 15
1.2.2 YNAB (You Need A Budget) 16
1.2.3 PocketGuard 17
1.2.4 Expensify 18
1.3 Постановка завдання 21
1.4 Висновки до розділу 1 22
2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ ВЕБЗАСТОСУНКУ 24
2.1 Огляд інструментів серверної розробки 24
2.2 Огляд інструментів розробки інтерфейсу користувача 26
2.3 Дослідження систем управління базами даних 28
2.4 Дослідження архітектурних стилів API 30
2.5 Висновки до розділу 2 32
3 РОЗРОБКА ВЕБ-ДОДАТКУ ТРЕКЕРУ ВИТРАТ 34
3.1 Опис предметної області 34
3.2 Архітектура системи 36
3.3 Функціональні вимоги системи 40
3.4 Вимоги до інтерфейсу 42
3.5 Реалізація 44
3.6 Тестування 71
3.7 Висновки до розділу 3 77
ВИСНОВКИ 78
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 79

ВСТУП

Актуальність теми

В епоху стрімкої та всепроникної цифрової трансформації управління особистими фінансами набуває все більшої ролі в житті кожної людини. Останні десятиліття значущих змін зазнала сфера фінансових технологій: мобільні платіжні системи, цифрові банки, блокчейн і криптовалюти. Усе це сильно змінило наші способи взаємодії зі своїми фінансами і вивело проблему відстеження власних витрат на принципово інший рівень. Незважаючи на це, багато людей продовжують покладатися на ручні методи відстеження фінансових витрат, які забирають багато часу і схильні до помилок. Існуючі ж цифрові рішення або також засновані на ручному введенні даних, або, маючи базову автоматизацію підрахунків, надають недостатній набір функцій в плані коригування та аналітики даних. Таким чином існує нагальна потреба у створенні цифрового рішення, яке змогло б поєднати в собі автоматизацію з орієнтованим на користувача дизайном і функціоналом.

Ця проблема особливо актуальна в Україні, де зростаючий глобальний попит на інструменти управління фінансами накладається на нестабільну економічну ситуацію. Політична нестабільність, коливання валютних курсів та інфляційний тиск роблять ефективне бюджетування не просто зручністю, але й необхідною запорукою стабільності українських домогосподарств і підприємців. Це в свою чергу посилює потребу українських споживачів в точних і доступних інструментах фінансового моніторингу, інтегрованих з місцевими цифровими фінансовими рішеннями.

Сьогодні серед сучасних фінансових сервісів в Україні особливу увагу привертає Monobank — мобільний банк, що завдяки своїм інноваційним підходам до обслуговування клієнтів здобув широку популярність. Його простий, але водночас потужний функціонал значно змінив уявлення українців про

банківські послуги та спростив щоденну взаємодію з фінансами. Водночас, незважаючи на очевидні переваги, функціональність Monobank переважно зосереджена на класичних банківських операціях. Користувачі отримують лише базові можливості з аналізу витрат, а відсутність гнучкого налаштування категорій і обмежені інструменти візуалізації даних не дозволяють повною мірою контролювати власний бюджет.

Цю проблему може вирішити окремий додаток, інтегрований з Monobank API. Такий застосунок автоматизуватиме отримання транзакцій у режимі реального часу, дозволяючи не тільки зручно відстежувати витрати, але й самостійно налаштовувати категорії, будувати деталізовану статистику та отримувати глибше розуміння фінансових звичок.

Зважаючи на потребу в доступності та зручності, для реалізації було обрано формат веб-додатку. Завдяки кросплатформності користувачі зможуть працювати з сервісом з будь-якого пристрою, не встановлюючи додаткового ПЗ. У якості серверної частини використовується ASP.NET Core, що забезпечує високу продуктивність і безпеку, необхідні при роботі з фінансовими даними. Інтерфейс побудовано на Razor Pages — це дозволяє створити інтуїтивно зрозумілий і адаптивний дизайн, а для роботи з базою даних застосовується Entity Framework Core у зв’язці з PostgreSQL — рішення, що добре підходить для масштабованих застосунків і складних запитів.

У результаті користувач отримає зручний інструмент для фінансового моніторингу, який поєднує автоматизацію, гнучкість і глибоку аналітику. Такий додаток зможе стати ефективною відповіддю на реальні потреби українського користувача та заповнити прогалину, яка досі залишається в екосистемі цифрових фінансових сервісів.

Мета дослідження

Метою даного дослідження є розробка веб-застосунку трекеру витрат з безшовною банківською інтеграцією та динамічною категоризацією. Цей до-

даток має подолати недоліки ручного відстеження витрат та обмежень існуючих цифрових інструментів через автоматизацію процесу заповнення даних та надання користувачам можливостей для самостійного налаштування правил категоризації витрат для досягнення більш індивідуальної звітності. Особливу увагу при розробці приділено забезпеченню високої продуктивності, масштабованості та безпеки, завдяки використанню сучасного технологічного стеку на основі ASP.NET Core.

Завдання дослідження

1. Аналіз літератури та існуючих рішень:
   - Дослідити наукові та освітні джерела, що описують процес розробки веб-додатків з використанням ASP.NET Core та додатків з банківською інтеграцією.
   - Дослідити існуючі застосунки трекерів витрат, зосередившись на визначені їх переваг та недоліків.
2. Створити архітектуру веб-додатку:
   - Розробити модульну архітектуру, яка передбачає розділення на рівні: презентаційний (UI), логіки бізнес-процесів (сервіси), доступу до даних (репозиторії), а також зовнішньої інтеграції (API-клієнти).
   - Забезпечити підтримку масштабованості: можливість додавання нових функцій без значного переписування коду.
3. Налаштувати банківську інтеграцію:
   - Ознайомитися з офіційною документацією Monobank API, включно з особливостями авторизації, форматами запитів та обмеженнями.
   - Реалізувати механізм авторизації користувача через токени доступу, їх зберігання у безпечному середовищі.
   - Налаштувати автоматичне оновлення транзакцій у фоновому режимі.
4. Реалізувати основні функціональні можливості для ефективного аналізу витрат:

- Створити систему динамічної категоризації витрат, яка дозволить користувачам гнучко налаштовувати власні категорії, змінювати їх та видаляти.
- Запровадити графіки та діаграми (наприклад, кругові, лінійні, гістограми), які в реальному часі відображатимуть розподіл витрат, динаміку змін і співвідношення категорій.
- Додати можливість пошуку та фільтрації транзакцій за категоріями.
- Забезпечити адаптацію інтерфейсу під мобільні пристрої, щоб користувач міг переглядати аналітику та редагувати витрати будь-де і будь-коли.

5. Виконати тестування та оптимізацію додатку:
   - Застосувати модульне тестування для основних компонентів логіки.
   - Провести інтеграційне тестування взаємодії з API Monobank та базою даних.
   - Провести UI-тести для перевірки зручності інтерфейсу та роботи з пристроями різних розмірів.

Об’єкт дослідження

Об'єктом дослідження є процес розробки веб-додатку трекеру витрат, з банківською інтеграцією для забезпечення автоматизованого відстеження витрат та налаштовуваної категоризаці. У дослідженні розглядаються як технічна реалізація веб-додатку, так і його функціональні аспекти. Особлива увага приділяється забезпеченню відповідності рішення сучасним вимогам щодо зручності використання та масштабованості, а також вирішенню проблем відстеження витрат в українському контексті.

Предмет дослідження

Предметом дослідження є вивчення методів та підходів для розробки веб-додатку автоматизованого трекеру витрат. Це включає в себе дослідження обраного технологічного стеку на основі ASP.NET Core, аналіз методів

інтеграції з банківськими API, зокрема API Monobank, дослідження методів реалізації категоризації витрат, що налаштовуються, вивчення основних принципів розробки інтерфейсу веб-застосунку для забезпечення простої та ефективної роботи користувача з ним.

Методи дослідження

Аналіз наукових та освітніх джерел, що стосуються сучасних трендів в сфері фінансових технологій, а також технологій розробки веб-додатків з використанням ASP.NET Core та додатків з банківською інтеграцією. Оцінка існуючих програмних рішень для виявлення їхніх особливостей, переваг, недоліків, якісної оцінки вимог споживачів.

Практичне значення одержаних результатів

Практична цінність цього дослідження полягає у спрощенні процесу відстеження витрат для українських користувачів. Розроблений веб-додаток трекеру витрат вирішує важливі потреби завдяки таким ключовим аспектам:

1. Завдяки безшовній інтеграції з банківським API користувачі зможуть автоматично відслідковувати свої витрати без необхідності вручну вводити дані. Це значно заощадить час і знизить кількість помилок, які часто виникають при традиційному способі ведення фінансів.
2. Гнучка система категоризації дозволить користувачам детальніше аналізувати свої фінанси, перетворюючи необроблені транзакції на зрозумілі та корисні звіти, що допоможуть приймати більш обдумані рішення щодо власного бюджету.
3. З технічної точки зору, проєкт демонструє ефективне використання сучасного технологічного стеку ASP.NET Core, що може стати корисним прикладом для розробників фінансових додатків, особливо тих, що працюють з банківською інтеграцією.

4. Важливо, що це рішення враховує особливості українського ринку і створене з урахуванням реалій економічної нестабільності, що робить його особливо актуальним і корисним для вітчизняних користувачів.

Окрім прямої користі для користувачів, проєкт робить свій внесок у розвиток фінансових технологій в Україні, показуючи, як можна успішно поєднати банківську інтеграцію та зручний дизайн у веб-застосунках, орієнтованих на потреби користувачів.

Глосарій

Веб-додаток — це програмне забезпечення, яке працює на сервері та до якого користувачі отримують доступ через веб-браузер. Воно забезпечує інтерактивність і дозволяє виконувати різноманітні дії онлайн.

Трекер витрат — це цифровий інструмент, який допомагає записувати, класифікувати та аналізувати ваші фінансові операції. Він допомагає краще розуміти ваші витрати, керувати бюджетом і приймати зважені рішення щодо грошей.

API (інтерфейс прикладного програмування) — це набір правил і інструментів, що дозволяють різним програмам «спілкуватися» між собою. Завдяки цьому трекер витрат може отримувати інформацію від зовнішніх банківських сервісів.

Monobank API — це сервіс від Monobank, який дає змогу автоматично отримувати дані про банківські транзакції та інформацію про рахунки без зайвих зусиль.

ASP.NET Core — це сучасний веб-фреймворк, який працює на різних платформах і допомагає розробляти потужні та масштабовані серверні додатки.

Razor Pages — частина ASP.NET Core, яка полегшує створення динамічних веб-сторінок, поєднуючи звичайний HTML з програмним кодом на C#.

Entity Framework Core — інструмент, який спрощує роботу з базою даних, дозволяючи програмістам працювати з даними через об’єкти, а не складні SQL-запити.

Синхронізація в режимі реального часу — це постійне оновлення інформації в додатку, яке гарантує, що ви завжди бачите найсвіжіші та точні дані про свої фінанси.

Інтерфейс користувача — це всі ті елементи, з якими взаємодієте ви у додатку. Хороший інтерфейс робить роботу з додатком зрозумілою, приємною та доступною.

Адаптивний дизайн — підхід, який дозволяє додатку однаково зручно працювати на будь-яких пристроях — від смартфонів до великих екранів комп’ютерів..

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Огляд літературних джерел

Швидкий розвиток фінансових технологій за останні роки суттєво змінив те, як ми керуємо своїми фінансами. З появою різних цифрових сервісів відстеження витрат стало простішим і зручнішим, а попит на автоматизовані та ефективні інструменти зростає. За прогнозами аналітиків компанії Grand View Research, світовий ринок програм для персональних фінансів активно росте: до 2030 року його обсяг може сягнути 1,59 мільярда доларів із середньорічним приростом понад 5%, що зумовлено широким використанням смартфонів і популярністю цифрових фінансових сервісів. Ще свіжіший звіт компанії The Business Research Company підкреслює, що з 2024 по 2033 роки цей ринок зростатиме навіть швидше — близько 8,5% на рік — завдяки поширенню цифрового банкінгу.

Ці тенденції відчутні і в Україні. Розвиток цифрових банків значно підвищив попит на прості та надійні інструменти для контролю особистих фінансів. Яскравим прикладом став Monobank — мобільний банк, що з 2017 року завоював значну частку українського ринку. За прогнозами одного із засновників, у 2024 році кількість активних користувачів Monobank сягне 8 мільйонів. Хоча додаток має базові можливості для відстеження витрат, йому бракує глибшого аналітичного функціоналу і гнучких налаштувань.

Загалом, на міжнародному ринку існує чимало міжнародних додатків для фінансового контролю, таких як YNAB, який допомагає планувати бюджет, або PocketGuard, що показує доступні кошти після сплати рахунків. Проте більшість з них орієнтовані на Північну Америку та Європу і не мають прямої інтеграції з українськими банками. Через це користувачі змушені вручну вводити дані або завантажувати файли, що створює додаткові незручності. Отже, на українському ринку є гостра потреба у локалізованих рішеннях, що враховують особливості національної банківської системи.

Для розробки такого рішення в цьому проєкті обрано сучасний технологічний стек на базі ASP.NET Core — кросплатформенного фреймворку, який поєднує високу продуктивність із можливістю масштабування, що важливо для фінансових сервісів з великим навантаженням. Razor Pages допомагають легко організувати навігацію і створювати динамічні сторінки, а Entity Framework Core значно спрощує роботу з базою даних, замінюючи складні SQL-запити на зручну роботу з об'єктами. Використання Monobank API дає можливість автоматично отримувати актуальні транзакції в режимі реального часу, зводячи до мінімуму ручне введення даних.

Розробка такого фінансового додатку має низку викликів: необхідно забезпечити надійний захист з'єднання з банківськими сервісами, підтримувати швидку і коректну синхронізацію даних, а також створити інтерфейс, який буде зручним і зрозумілим для користувачів. Захист даних вимагає впровадження ефективних механізмів автентифікації, а робота з реальним часом — оптимізації, щоб інформація була актуальною і узгодженою. Водночас інтуїтивний дизайн інтерфейсу допоможе зробити користування додатком комфортним, не жертвуючи при цьому функціональністю.

Таким чином, враховуючи сучасні тенденції і особливості українського ринку, ми прагнемо створити практичний і дружній до користувача інструмент для відстеження витрат з глибокою банківською інтеграцією та розширеними можливостями аналізу фінансів.

1.2 Огляд існуючих рішень

1.2.1 Власна аналітика витрат від Monobank

Monobank – це провідний український цифровий банк, пропонує вбудовану функцію відстеження витрат у своєму мобільному додатку. Його інструменти автоматично класифікують транзакції за попередньо визначеними групами (наприклад, «Їжа», «Транспорт») та відображають базову статистику витрат за визначений період часу (див. Рис. 1).

Переваги:

- Прямий доступ до даних про транзакції в режимі реального часу без сторонньої інтеграції.
- Миттєві push-повідомлення про кожну транзакцію, що підвищує обізнаність про витрати.
- Безкоштовний для клієнтів Monobank.

Недоліки:

- Обмежена кастомізація: користувачі не можуть створювати, перейменовувати або об'єднувати категорії.
- Відсутність веб-інтерфейсу: доступний лише на мобільних пристроях, що обмежує доступність на десктопі.

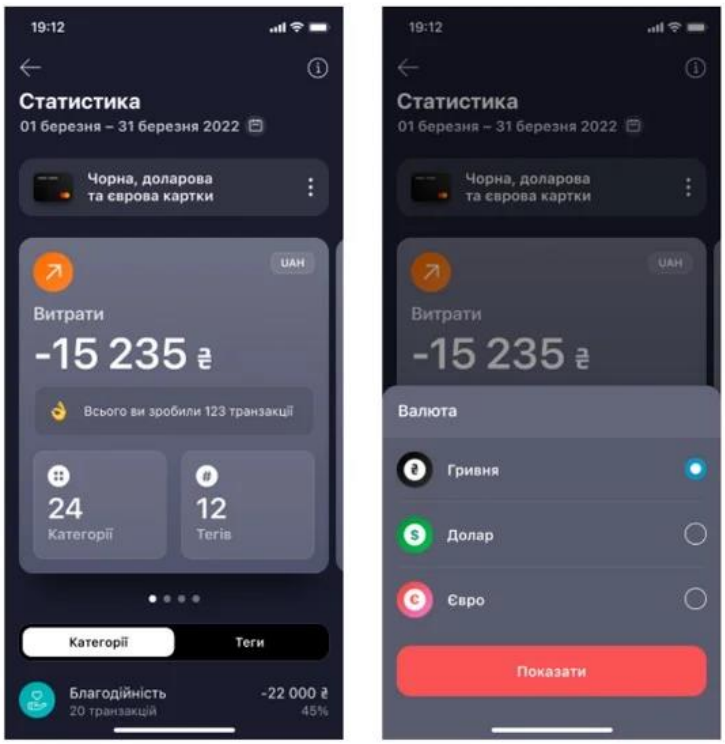


Рисунок 1 — *Інтерфейс Monobank.*

1.2.2 YNAB (You Need A Budget)

You Need a Budget (YNAB) – це додаток і веб-сервіс для управління особистими фінансами, заснований на методі нульового бюджетування.

YNAB допомагає користувачам планувати і відстежувати витрати, а також досягати конкретних фінансових цілей (див. Рис. 2).

Переваги:

- Категоризація постійних витрат на основі правил (наприклад, автоматичне сортування «Продукти»).
- Детальні звіти про чистий капітал, скорочення боргу та тенденції витрат.

Недоліки:

- Сервіс не має безкоштовної версії, а найдешевша місячна підписка коштує $14.99, що серйозно обмежує доступність.
- Відсутність інтеграції з українськими банками потребує ручного введення даних для українських користувачів.
- Стрімка крива навчання для користувачів, які не знайомі з бюджетуванням за методом нульового бюджетування.

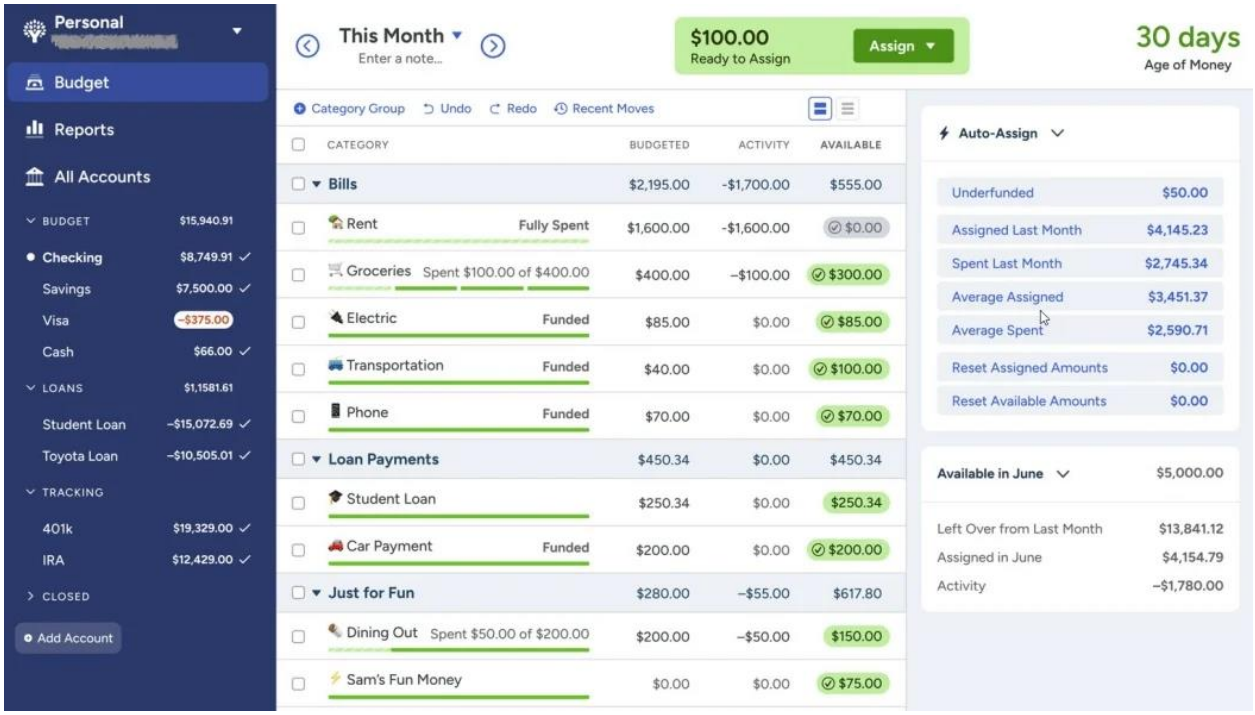


Рисунок 2 — *Інтерфейс YNAB.*

1.2.3 PocketGuard

PocketGuard - це зручний додаток для управління особистими фінансами, який допомагає користувачам контролювати щоденні витрати, розраховуючи,

скільки вільного доходу залишається після сплати рахунків і досягнення цілей заощаджень (див. Рис. 3).

Переваги:

- Функція «В кишені» показує доступні гроші на витрати з першого погляду.
- Простий та інтуїтивно зрозумілий користувацький інтерфейс, що підходить для початківців.
- Наявна можливість безкоштовного використання.

Недоліки:

- Відсутність інтеграції з українськими банками потребує ручного введення даних для українських користувачів.
- Безкоштовна версія програми має обмежений функціонал в контексті налаштовування власних категорій.
- Платні функції можуть бути недоступними для багатьох місцевих користувачів.

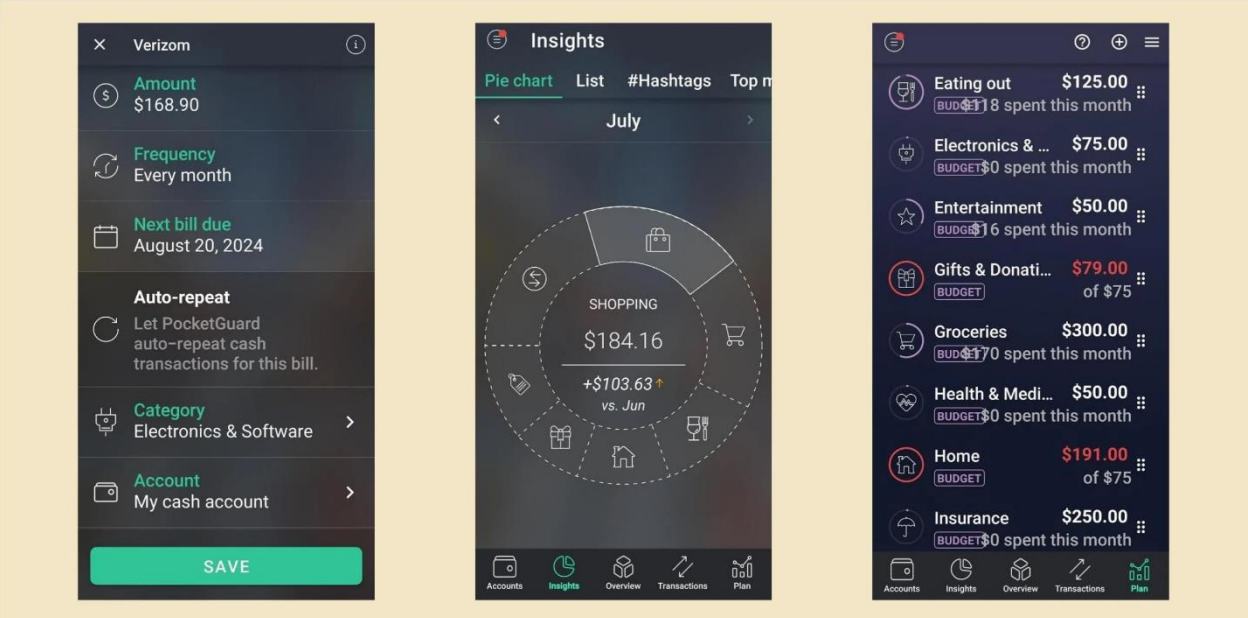


Рисунок 3 — *Інтерфейс PocketGuard.*

1.2.4 Expensify

Expensify - це універсальний інструмент, який спочатку призначався для звітування про бізнес-витрати, але також використовується для відстеження

особистих фінансів. Він підтримує сканування квитанцій, відстеження пробігу та автоматизовану звітність (див. Рис. 4).

Переваги:

- Сканування квитанцій на основі OCR з автоматичною категоризацією витрат.
- Багатокористувацька підтримка та синхронізація з хмарними пристроями.
- Інтеграція з бухгалтерськими інструментами, такими як QuickBooks та Xero.
- Наявна можливість безкоштовного використання.

Недоліки:

- Відсутність інтеграції з українськими банками потребує ручного введення даних для українських користувачів.
- Надто складний для персональних користувачів, які не зосереджені на управлінні бізнес-витратами.
- Вимагає підписки для доступу до більшості розширених функцій.

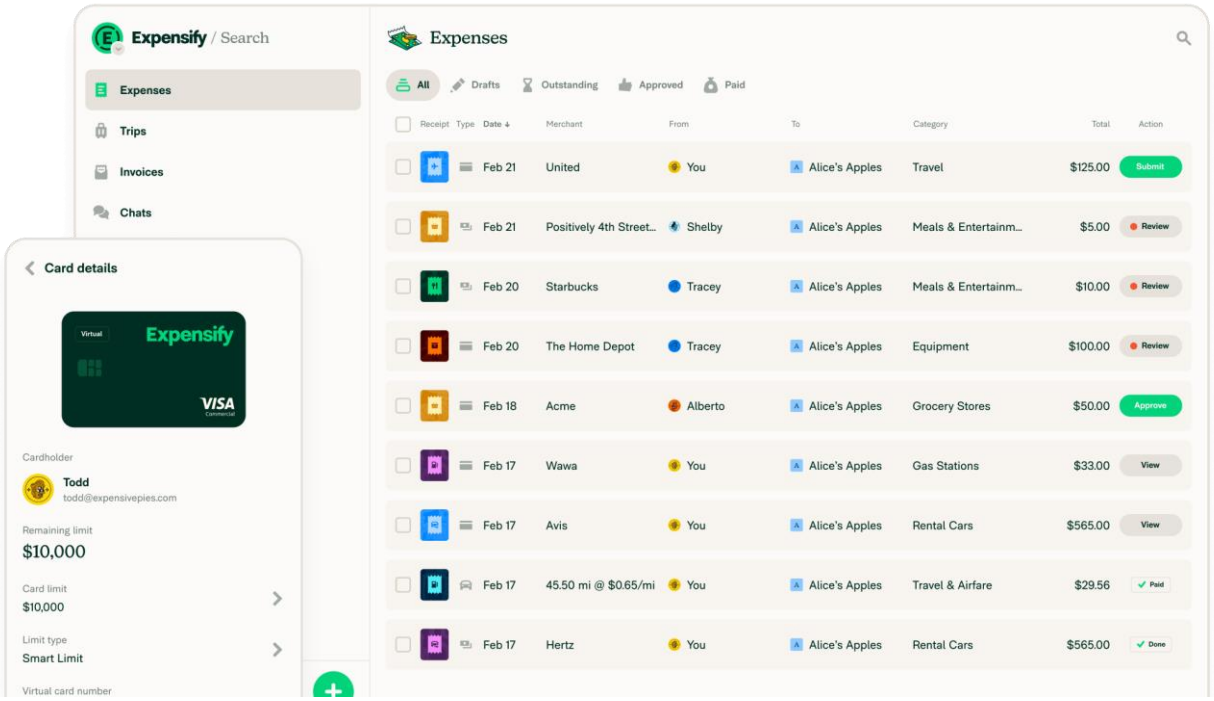


Рисунок 4 — *Інтерфейс Expensify.*

У таблиці 1 показано, як існуючі рішення відповідають найважливішим критеріям для українських користувачів:

Таблиця 1 *Порівняння характеристик Monobank, YNAB, PocketGuard та Expensify*

Критерії	Monobank	YNAB	PocketGuard	Expensify
Інтеграція з українськими банками	Використовує власні дані	Відсутня	Відсутня	Відсутня
Категорії, що налаштовуються	Відсутні	Наявні	Наявні лише в платній версії програми	Наявні лише в платній версії програми
Безкоштовна версія	Наявна, з обмеженим функціоналом	Відсутня	Наявна, з обмеженим функціоналом	Наявна, з обмеженим функціоналом
Підтримка української мови та валюти	Підтримуються	Повністю відсутня	Повністю відсутня	Підтримується тільки українська валюта

Проведений огляд існуючих рішень наочно продемонстрував, що більшість існуючих додатків для відстеження витрат не повністю відповідають потребам українських користувачів. Одна з головних проблем — відсутність інтеграції з місцевими банківськими системами, що ускладнює автоматичний облік фінансів. Крім того, багато популярних закордонних сервісів пропонують обмежений безкоштовний функціонал або взагалі не мають безкоштовних версій. Важливим бар’єром є також мовне питання: більшість таких додатків не підтримують українську мову і не адаптовані під національну валюту — гривню. Через це зростає потреба у доступному, простому у використанні та повністю локалізованому трекері витрат, який би працював у веб-браузері, мав приємний дизайн, гнучкі можливості та був доступний на різних пристроях.

1.3 Постановка завдання

Метою цього проєкту є створення зручного веб-трекера витрат з інтеграцією API Monobank широкими можливостями аналізу витрат. Такий інструмент допоможе заповнити існуючі прогалини на ринку, забезпечуючи автоматичну синхронізацію, надійний захист даних та практичні інструменти бюджетування, адаптовані для українських користувачів.

Етапи розробки:

1. Аналіз вимог:
   - Визначення функціональних вимог: інтеграція API Monobank в режимі реального часу, правила категоризації, що налаштовуються, інтерактивні дашборди.
2. Проектування системи:
   - Створення трирівневої архітектури для чіткого розподілу функцій і підвищення масштабованості:
     - Фронтенд: Razor Pages для створення гнучкого і адаптивного інтерфейсу, а також Chart.js для наочного відображення фінансових даних.
     - Бекенд: ASP.NET Core для обробки бізнес-логіки та безпечної роботи з API, а також Entity Framework Core для простого та ефективного доступу до даних.
     - База даних: структура на PostgreSQL для зберігання транзакцій, налаштувань користувачів та статистичних даних.
3. Реалізація серверної частини:
   - Аналіз вимог до безпеки, масштабованості та гнучкої обробки транзакцій.
   - Вибір архітектурного підходу Clean Architecture для розділення відповідальностей.
   - Проєктування шарів: контролери, сервіси, репозиторії, доменні моделі.

- Винесення інтеграції з Monobank API в окремі адаптери.
- Впровадження принципів інверсії залежностей і залежностей через інтерфейси.

4. Реалізація інтерфейсу користувача:
   - Визначення типових сценаріїв користувача для побудови структури інтерфейсу.
   - Обрання Razor Pages для взаємодії з API.
   - Вибір Chart.js для візуалізації даних у графіках.
   - Врахування адаптивності інтерфейсу для мобільних та десктопів.
5. Тестування та валідація:
   - Проведення модульного тестування всіх API-ендпоінтів і взаємодії з базою даних за допомогою Entity Framework Core для гарантії стабільності роботи.
   - Виконання інтеграційного тестування, щоб перевірити коректність роботи системи у цілому.

Результатом буде веб-застосунок, який не тільки автоматично відстежуватиме витрати клієнтів Monobank, зменшуючи помилки ручного введення, але й надаватиме зрозумілі, практичні інструменти для ефективного управління бюджетом. Високий рівень безпеки, підтримка української мови та адаптація до українського ринку зроблять цей трекер особливо корисним і затребуваним серед українських користувачів.

1.4 Висновки до розділу 1

1. Існуючим інструментам для відстеження витрат бракує адаптованих рішень для українських користувачів, зокрема інтеграції з Monobank у режимі реального часу.
2. Глобальні платформи (наприклад, Expensify, YNAB) пропонують розширену аналітику, але не мають достатньої локалізації для українського ринку.

3. Запропонований веб-додаток заповнює ці прогалини, поєднуючи автоматизовану банківську синхронізацію, категорії, що налаштовується, зрозумілий статстичний аналіз та адаптивність до українського ринку.
4. Сучасні технології (ASP.NET Core, PostgreSQL) забезпечують масштабованість, безпеку та швидкодію.
5. Проєкт сприяє підвищенню фінансової грамотності в Україні, поєднуючи світові фінтех-стандарти з регіональними потребами.

2 ДОСЛІДЖЕННЯ ПРОГРАМНИХ ЗАСОБІВ РЕАЛІЗАЦІЇ ВЕБЗАСТОСУНКУ

2.1 Огляд інструментів серверної розробки

Вибір фреймворку для серверної розробки — ключовий етап у створенні сучасного веб-додатку. У випадку з трекером витрат важливо забезпечити не лише хорошу продуктивність, але й надійний захист даних і зручність підтримки коду в майбутньому. Тому перед початком розробки було проведено порівняльний аналіз популярних інструментів для серверної розробки, щоб знайти найбільш придатне рішення для поставлених цілей:

1. ASP.NET Core — кросплатформений фреймворк з відкритим вихідним кодом, створений компанією Microsoft. Він вирізняється модульністю, високою продуктивністю та широкими можливостями для побудови надійних веб-сервісів. ASP.NET Core також має потужний набір інструментів, що дозволяє ефективно створювати як хмарні, так і корпоративні рішення.

   Переваги:

   - Висока продуктивність і масштабованість при великих навантаженнях.
   - Комплексні вбудовані функції безпеки (автентифікація, авторизація, захист даних).
   - Потужна інструментальна підтримка через Visual Studio та .NET CLI.
   - Безшовна інтеграція з Entity Framework Core для спрощення доступу до баз даних.
   - Крос-платформна сумісність.

   Недоліки:

   - Більш високе навантаження на систему під час виконання порівняно з іншими фреймворками.

- Обмеження щодо ліцензування деяких корпоративних функцій у Visual Studio.

2. Node.js — це середовище виконання JavaScript, створене на базі двигуна Chrome V8, яке ідеально підходить для розробки масштабованих мережевих додатків. Для спрощення створення веб-додатків і API у поєднанні з ним часто використовують легкий та гнучкий фреймворк Express.js, який допомагає швидко запускати серверні проєкти.

   Переваги:

   - Використовує JavaScript як у фронтенді, так і в бекенді, уніфікуючи технологічний стек.
   - Легке ядро з широкою екосистемою проміжного програмного забезпечення для швидкого створення прототипів.
   - Ефективний асинхронний ввід/вивід для обробки багатьох одночасних з'єднань.
   - Велика спільнота та велика кількість пакетів з відкритим вихідним кодом.

   Недоліки:

   - Вимагає ретельного управління асинхронним кодом, щоб уникнути зворотного виклику або складності обіцянок.
   - Функції безпеки повинні бути зібрані з декількох бібліотек, що збільшує накладні витрати на обслуговування.
   - Однопотоковість може вимагати додаткової конфігурації для завдань, що вимагають багато процесорних ресурсів.

3. Django — це потужний веб-фреймворк для мови Python, який орієнтований на швидку розробку та зрозумілий, продуманий дизайн. Django пропонує багато готових рішень "з коробки", таких як зручний ORM, панель адміністратора і вбудовані механізми захисту, що значно полегшує роботу розробника.

   Переваги:

- Фреймворк із «вбудованим набором» функцій — ORM, панеллю адміністратора та надійними механізмами безпеки.
- Швидка розробка завдяки акценту на конвенції замість складної конфігурації.
- Надійний захист від типових веб-загроз, таких як SQL-ін’єкції, XSS і CSRF.
- Велика спільнота та широкий вибір готових додатків і бібліотек.

Недоліки:

- Монолітна архітектура може виявитися надмірною для простих проектів.
- Продуктивність поступається скомпільованим фреймворкам при високих навантаженнях.
- Менш оптимальний вибір для мікросервісних рішень без додаткової адаптації.

Враховуючи проведений аналіз, для розробки серверної частини проєкту було обрано ASP.NET Core. Цей фреймворк демонструє відмінну продуктивність у тестах .NET, має потужні засоби безпеки і легко інтегрується з Entity Framework Core та PostgreSQL. Такий вибір повністю відповідає нашим вимогам до надійної та безпечної обробки фінансових даних, а також забезпечує масштабованість. Завдяки цьому ми отримуємо міцну основу для реалізації інтеграції з банком у режимі реального часу, складної бізнес-логіки і подальшого швидкого розвитку додатку.

2.2 Огляд інструментів розробки інтерфейсу користувача

Вибір ефективного фреймворку для розробки інтерфейсу користувача — ключовий крок для створення адаптивного та зручного інтерфейсу. Для цього проєкту було розглянуто три основні варіанти: Razor Pages з Chart.js, React і Angular.

1. Razor Pages — це серверний фреймворк у складі ASP.NET Core, який дозволяє швидко створювати сторінки. У поєднанні з Chart.js він дає змогу додавати інтерактивні графіки без складної SPA-архітектури.

   Переваги:

   - Безшовна інтеграція з бекендом ASP.NET Core та спільними моделями.
   - Спрощена модель розробки з мінімальним поділом між клієнтом і сервером.
   - Менша початкова складність та зменшений розмір пакету на стороні клієнта.
   - Вбудований рендеринг та маршрутизація без зовнішніх залежностей.

   Недоліки:

   - Знижена інтерактивність у порівнянні з SPA-фреймворками.
   - Повне перезавантаження сторінки при виконанні певних дій може вплинути на швидкість відгуку.
   - Обмежена екосистема для розширеного управління станом на стороні клієнта.

2. React — популярна JavaScript-бібліотека для створення інтерфейсів з компонентною архітектурою та ефективним оновленням віртуального DOM.

   Переваги:

   - Висока гнучкість у створенні динамічних, інтерактивних інтерфейсів.
   - Багата екосистема бібліотек (наприклад, Redux для управління станами, React Router для маршрутизації).
   - Потужна підтримка спільноти та велика документація.

   Недоліки:

   - Потребує інструментарію для збірки та клієнтського пакунку (наприклад, webpack).
   - Більш крута крива навчання для розробників, які не знайомі з сучасними JavaScript-фреймворками.

- Потенційні проблеми з продуктивністю, якщо не оптимізувати належним чином (надмірна кількість повторних рендерингів).

3. Angular — повноцінний TypeScript-фреймворк для створення масштабних SPA з усіма необхідними інструментами: маршрутизацією, формами, управлінням станом.

   Переваги:

   - Підхід «з коробки» з офіційними модулями для більшості потреб (HTTP, форми, маршрутизація).
   - Сильна інтеграція з TypeScript підвищує надійність коду та зручність супроводу.
   - Структура, що враховує думки, сприяє узгодженій архітектурі додатків.

   Недоліки:

   - Більший розмір пакунка порівняно з легшими фреймворками.
   - Крута крива навчання через широкий набір функцій та концепцій.
   - Накладні витрати для невеликих проектів, які не потребують повних можливостей SPA.

Враховуючи потреби проєкту — оновлення даних в реальному часі, багаті візуалізації та тісну інтеграцію з ASP.NET Core — було прийнято рішення використовувати Razor Pages для основного рендерингу і Chart.js для динамічних графіків. Такий підхід дозволяє поєднати простоту розробки, підтримку та зручність для користувачів.

2.3 Дослідження систем управління базами даних

Не менш важливим елементом програми для відстеження витрат є також система баз даних, яка повинна надійно зберігати великі обсяги фінансових операцій, забезпечуючи цілісність даних і високу продуктивність. Для вибору найбільш відповідної СУБД було проаналізовано три популярні реляційні бази даних: PostgreSQL, MySQL та SQLite.

1. PostgreSQL — сучасна, відкрита і надійна реляційна СУБД, що відповідає стандартам SQL. Вона підтримує складні запити, просунуті індекси та гарантує повну цілісність транзакцій (ACID). Це робить її ідеальним вибором для фінансових додатків.

   Переваги:

   - Повна відповідність стандарту ACID забезпечує цілісність транзакцій.
   - Підтримка розширених типів даних (JSONB, масиви) і збережених процедур.
   - Розширювана архітектура з багатим набором розширень (наприклад, PostGIS, pg_stat_statements).
   - Сильний контроль паралелізму за допомогою MVCC (Multi-Version Concurrency Control).

   Недоліки:

   - Дещо більше використання ресурсів у порівнянні з легкими базами даних.
   - Складність розширених функцій може вимагати більш глибокої експертизи для оптимізації.
2. MySQL — популярна і швидка СУБД з відкритим кодом, яка часто використовується для веб-додатків з простими операціями над даними.

   Переваги:

   - Висока швидкість читання і проста конфігурація.
   - Широка підтримка спільноти та комерційна підтримка (Oracle).
   - Легко налаштовується для простих випадків використання.

   Недоліки:

   - Обмежена підтримка складних запитів і розширених функцій SQL.
   - Непослідовна поведінка в різних механізмах зберігання (наприклад, MyISAM проти InnoDB).
   - Історично менший акцент на повну відповідність ACID до прийняття InnoDB.

3. SQLite — легка, безсерверна СУБД, що вбудовується безпосередньо у додаток. Ідеально підходить для невеликих проєктів, тестування або локального зберігання.

   Переваги:

   - Налаштування з нульовою конфігурацією та мінімальними накладними витратами.
   - Чудово підходить для сценаріїв розробки, тестування або локального кешування.
   - Малий розмір, що полегшує розгортання.

   Недоліки:

   - Не призначений для високонавантажених або великомасштабних багатокористувацьких середовищ.
   - Обмежена масштабованість у порівнянні з клієнт-серверними СКБД.

Після детального аналізу було обрано PostgreSQL для зберігання даних розроблюваного додатку. Її надійність, потужність у роботі зі складними запитами та можливість розширення найкраще відповідають вимогам безпечного й ефективного управління фінансовими даними. Крім того, розвинена екосистема інструментів сприяє зручній розробці і підтримці проекту..

2.4 Дослідження архітектурних стилів API

Вибір архітектурного стилю API суттєво впливає на те, наскільки гнучко, швидко і просто наш трекер витрат зможе взаємодіяти з банківськими сервісами. Ми розглянули три популярні підходи: REST, GraphQL та SOAP.

1. REST — це стиль побудови API, який використовує стандартні HTTP-методи (GET, POST, PUT, DELETE) для роботи з ресурсами, представленими у вигляді URL-адрес. REST наголошує на безстанній комунікації та уніфікованих інтерфейсах.

   Переваги:

- Простота і широке розповсюдження роблять REST API легкими для розуміння та інтеграції.
- Використовує методи HTTP (GET, POST, PUT, DELETE) для стандартизованих операцій.
- Відмінна підтримка кешування через заголовки HTTP, що підвищує продуктивність.
- Діагностика мови і платформи, з широким інструментарієм і документацією.

Недоліки:

- Можливе надмірне або недостатнє отримання даних, оскільки відповіді прив'язані до визначень ресурсів.
- Немає вбудованої схеми або системи типів, що вимагає додаткової документації (наприклад, OpenAPI) для ясності.

2. GraphQL — це мова запитів, що дає змогу клієнтам точно вказувати, які саме дані їм потрібні, і збирати інформацію з різних джерел одним запитом.

Переваги:

- Клієнти можуть вказати точні вимоги до даних, усуваючи надмірну вибірку.
- Єдина кінцева точка для всіх запитів зменшує мережеві накладні витрати.
- Строго типізована схема забезпечує чіткі контракти між клієнтом і сервером.

Недоліки:

- Додаткова складність у реалізації сервера та стратегій кешування.
- Крива навчання для розробників, незнайомих з концепціями GraphQL.
- Потенціал для вузьких місць у продуктивності, якщо запити не оптимізовані.

3. SOAP — це протокол, який працює з повідомленнями у форматі XML і має жорсткі стандарти, включаючи опис контрактів через WSDL. Він підтримує високий рівень безпеки та розширені функції для транзакцій.

   Переваги:

   - Вбудовані стандарти безпеки (WS-Security), транзакцій та надійності.
   - Надійна типізація та формальні контракти через WSDL забезпечують узгодженість.
   - Добре підходить для корпоративних середовищ, що вимагають суворих протоколів.

   Недоліки:

   - Багатослівний формат XML призводить до збільшення розміру повідомлень і накладних витрат на розбір.
   - Складні специфікації збільшують зусилля по впровадженню та підтримці.
   - Менш підходить для легких веб- або мобільних додатків.

Після врахування всіх факторів, було обрано REST як архітектурний стиль для API веб-додатку. Його простота, широке поширення та відмінна сумісність із HTTP і RESTful API Monobank роблять його найкращим варіантом для нашого трекера витрат. Безстанність REST і можливості кешування забезпечують швидкий і надійний обмін інформацією про фінансові транзакції.

2.5 Висновки до розділу 2

Проведене дослідження програмних інструментів допомогло сформувати оптимальний набір технологій для розроблюваного веб-додатку з відстеження витрат. Всі компоненти були ретельно підібрані, щоб забезпечити найбільш ефективну роботу з фінансовими даними, надійну інтеграцію з банком та якісний статистичний аналіз. Кожен елемент цього стеку доповнює інші, створюючи злагоджену і надійну систему.

Для серверної частини було обрано ASP.NET Core, який добре поєднує високу продуктивність і надійний захист від типових загроз, що особливо важливо для фінансових додатків. Взаємодія з базою даних через Entity Framework Core дозволить легко і безпечно працювати з даними, гарантуючи цілісність фінансових операцій.

На стороні клієнта буде використано поєднання Razor Pages та JavaScript, що дозволить створювати адаптивні й зручні інтерфейси. Такий підхід скоротить час розробки у порівнянні з альтернативами і дасть змогу плавно відображати динамічні графіки та статистику витрат у реальному часі.

В якості системи управління базами даних було обрано PostgreSQL завдяки її неперевершеній надійності у фінансовому контексті:

- Повна відповідність стандарту ACID для безпеки транзакцій.
- Розширене індексування для швидких статистичних запитів.
- Підтримка JSON для гнучких структур даних.
- Перевірена масштабованість за рахунок розбиття на розділи.

Архітектура REST API слугує комунікаційною основою, обраною для цього:

- Вбудована сумісність з банківськими API, такими як Monobank.
- Передбачувані характеристики продуктивності.
- Проста реалізація кешування.
- Широке знайомство з розробниками.

Після завершення дослідження доступних інструментів і вибору найефективніших технологій розпочнеться етап розробки. Він базуватиметься на обраному технологічному стеку, що дозволить створити надійне й зручне рішення для контролю витрат. Архітектура системи буде достатньо гнучкою, щоб легко адаптуватися до майбутніх змін і вдосконалень, але водночас забезпечуватиме стабільну роботу основних фінансових функцій уже з перших етапів.

3 РОЗРОБКА ВЕБ-ДОДАТКУ ТРЕКЕРУ ВИТРАТ

3.1 Опис предметної області

Предметна область цього дослідження лежить у площині управління особистими фінансами, з особливим акцентом на розробці та впровадженні веб-додатку для відстеження витрат, який інтегрується з банківськими системами для надання фінансової інформації та статистичного аналізу в реальному часі. У сучасному цифровому ландшафті, де рішення у сфері фінансових технологій (фінтех) стають незамінними, цей проект спрямований на задоволення зростаючого попиту на доступні та автоматизовані інструменти, які спрощують бюджетування та покращують процес прийняття фінансових рішень. Це особливо актуально для України, де фінтех-сектор стрімко розвивається завдяки впровадженню цифрових банківських послуг, таких як Monobank. Додаток має на меті розширити можливості користувачів, надаючи ефективну, зручну платформу для безперешкодного управління своїми фінансами.

Основною метою цього проекту є розробка веб-додатку, який використовує API Monobank для автоматизації відстеження витрат та надання користувачам дієвої фінансової інформації. Інтегруючись безпосередньо з банківським сервісом, додаток усуває необхідність ручного введення транзакцій, забезпечуючи користувачам точний і актуальний огляд їхньої фінансової діяльності. На додаток до відстеження в режимі реального часу, система пропонує такі функції, як категоризація витрат, що налаштовується, та інтерактивні статистичні інформаційні панелі, що дозволяють користувачам аналізувати свої витрати і приймати обґрунтовані фінансові рішення. Цей подвійний фокус на автоматизації та аналітиці позиціонує додаток як практичний інструмент для підвищення фінансової грамотності та незалежності.

Основні функціональні можливості додатку включають наступне:

- Синхронізація транзакцій в режимі реального часу: автоматично отримує та оновлює дані про транзакції з рахунку користувача в Monobank, забезпечуючи миттєвий фінансовий огляд.
- Налаштовувана категоризація: дозволяє користувачам визначати та налаштовувати категорії витрат відповідно до своїх уподобань, полегшуючи персоналізований фінансовий контроль.
- Інтерактивні аналітичні панелі: представляють фінансові дані за допомогою візуальних інструментів, таких як діаграми та графіки, дозволяючи користувачам виявляти тенденції та ефективно контролювати бюджети.

Технічна реалізація додатку базується на сучасному та надійному технологічному стеку. Бекенд розроблений з використанням ASP.NET Core, що забезпечує високу продуктивність, масштабованість і безпеку для управління даними в реальному часі та взаємодії з API. Фронтенд побудований за допомогою Razor Pages, що пропонує спрощену структуру для створення динамічних та адаптивних користувацьких інтерфейсів. Зберігання та управління даними здійснюється за допомогою PostgreSQL, доступ до якої здійснюється через Entity Framework Core, що забезпечує ефективне та безпечне рішення для роботи з базами даних. API Monobank слугує наріжним каменем для банківської інтеграції, дозволяючи додатку отримувати та обробляти дані про транзакції в режимі реального часу. Процес розробки також включає ретельне тестування та ітеративне вдосконалення для забезпечення надійності та задоволеності користувачів.

Ключовим акцентом цього проекту є його дизайн, орієнтований на користувача. Пріоритетом програми є інтуїтивно зрозумілий і доступний інтерфейс, що робить її придатною для користувачів з різним рівнем технічної підготовки. Такі функції, як адаптивний дизайн і проста навігація, підвищують зручність використання на різних пристроях, а заходи безпеки гарантують, що

користувачі можуть довіряти системі свої фінансові дані. Такий фокус на користувацькому досвіді відповідає ширшій меті - зробити управління фінансами доступним та ефективним для всіх.

У контексті зростаючої фінтех-екосистеми в Україні цей додаток має значну актуальність. Широке використання цифрових банківських послуг, таких як Monobank, відображає перехід до технологічних фінансових рішень. Пропонуючи відстеження в режимі реального часу та глибоку аналітику, додаток сприяє підвищенню фінансової грамотності та надає користувачам можливість впевнено керувати своїми фінансами. Таким чином, цей проект є значним кроком вперед у розробці практичних фінтех-інструментів, пристосованих до сучасних потреб.

3.2 Архітектура системи

Системна архітектура веб-додатку для відстеження витрат розроблена таким чином, щоб забезпечити надійне, безпечне та масштабоване рішення для управління особистими фінансами, яке легко інтегрується з банківськими системами для надання даних про транзакції в режимі реального часу та статистичної інформації. Використовуючи сучасні веб-технології, архітектура підтримує ефективну обробку даних, безпеку фінансових операцій та інтуїтивно зрозумілий користувацький інтерфейс, адаптований до українського фінтех-контексту, зокрема завдяки інтеграції з API Monobank. У цьому підрозділі описано основні компоненти, їхню взаємодію та загальний дизайн, підкреслюючи відповідність архітектури цілям проекту.

Архітектура складається з чотирьох ключових рівнів, кожен з яких базується на певних технологіях і виконує певні функції:

1. Клієнтський рівень (Frontend):
    - Технології: Razor Pages, HTML5, JavaScript, Tailwind CSS
    - Опис: клієнтський рівень слугує інтерфейсом користувача, забезпечуючи взаємодію з додатком через адаптивні та динамічні

веб-сторінки. Razor Pages, інтегрований з ASP.NET Core, полегшує розробку інтуїтивно зрозумілих представлень для відстеження витрат, управління категоріями та візуалізації статистики. Tailwind CSS покращує адаптивність інтерфейсу, забезпечуючи однакову роботу на десктопних і мобільних пристроях.

2. Серверний рівень (Backend):
   - Технології: ASP.NET Core, C#, шаблон MVC
   - Опис: серверний рівень керує основною логікою додатку, обробляє клієнтські запити та координує взаємодію з базою даних та зовнішніми API. Побудований на основі ASP.NET Core та з використанням патерну MVC (Model-View-Controller), цей рівень забезпечує модульний та підтримуваний підхід до обробки користувацьких операцій та генерації відповідей.
3. Рівень даних:
   - Технології: PostgreSQL, Entity Framework Core
   - Опис: рівень даних відповідає за зберігання та пошук даних програми, включаючи профілі користувачів, записи транзакцій та категоризацію витрат. PostgreSQL забезпечує надійну та масштабовану реляційну базу даних, тоді як Entity Framework Core (EF Core) спрощує доступ до даних за допомогою об'єктно-реляційного відображення, забезпечуючи ефективність та безпеку операцій з базою даних.
4. Зовнішній рівень API:
   - Технології: Monobank API
   - Опис: зовнішній рівень забезпечує зв'язок у режимі реального часу з банківськими послугами Monobank, дозволяючи додатку отримувати актуальні дані про транзакції. Він відіграє ключову роль в автоматизації відстеження витрат та наданні користувачам точної фінансової інформації.

На рисунку 5 зображено схему архітектури веб-додатку, яка ілюструє взаємодію основних компонентів системи — від клієнтського інтерфейсу до зовнішнього банківського API.

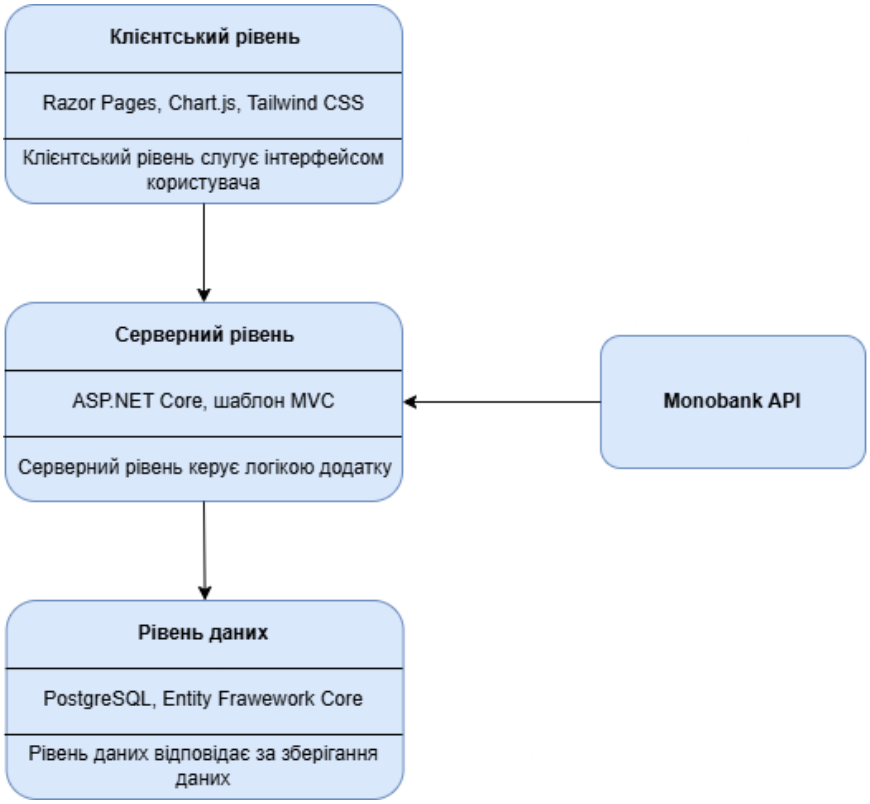


Рисунок 5 – *Схема архітектури веб-додатку.*

Компоненти системи взаємодіють через чітко визначений потік даних, забезпечуючи безперебійну функціональність і безпеку операцій:

- Зв'язок між клієнтом і сервером:

Клієнтський рівень ініціює HTTP-запити до серверного рівня, викликані діями користувача, такими як перегляд транзакцій або створення нових категорій. Ці запити обробляються контролерами ASP.NET Core, які організовують відповідь програми.

- Взаємодія між сервером і даними:

Контролери серверного рівня взаємодіють з рівнем даних через ядро Entity Framework Core, яке перетворює логіку програми в SQL-запити до

PostgreSQL. Ця взаємодія підтримує пошук, зберігання та оновлення даних, підтримуючи їх узгодженість та цілісність.

- Інтеграція Server-API:

Серверний рівень взаємодіє з API Monobank за допомогою захищених HTTPS-запитів. Дані про транзакції, отримані з API, обробляються і зберігаються в базі даних PostgreSQL, що дозволяє оновлювати фінансовий огляд користувача в режимі реального часу.

- Рендеринг відповідей:

Після обробки запитів серверний рівень повертає дані клієнту у форматі JSON. Потім Razor Pages динамічно перетворює ці дані у візуальні елементи, такі як списки транзакцій або аналітичні діаграми, для використання користувачем.

- Міркування щодо безпеки:

Архітектура надає пріоритет безпеці за допомогою декількох механізмів: HTTPS шифрує дані під час передачі, а параметризовані запити Entity Framework Core захищають від SQL-ін'єкцій.

Архітектура системи ефективно підтримує цілі програми для відстеження витрат, поєднуючи інтеграцію даних в режимі реального часу, безпечне управління фінансами та орієнтований на користувача дизайн. Використання Razor Pages і Tailwind CSS на клієнтському рівні забезпечує доступний і чуйний інтерфейс, а реалізація ASP.NET Core на серверному рівні гарантує ефективну і модульну обробку даних. Рівень даних на основі PostgreSQL та Entity Framework Core забезпечує надійну основу для управління фінансовими даними, а інтеграція Monobank API вигідно вирізняє додаток, автоматизуючи оновлення транзакцій. Ця архітектура не тільки відповідає поточним вимогам, але й пропонує масштабованість для майбутніх удосконалень, таких як додаткові банківські інтеграції або розширені статистичні функції, що робить її підходящою основою для сучасного фінтех-рішення.

3.3 Функціональні вимоги системи

Ці вимоги визначають ключові можливості системи, які зроблять користування додатком простим, безпечним та корисним для кожного, хто хоче краще контролювати свої фінанси. Головна мета — надати користувачам зручний інструмент із глибокою банківською інтеграцією та деталізованою статистикою витрат:

1. Реєстрація користувачів, автентифікація та управління профілем:
    - Реєстрація: користувачі зможуть легко створити власний профіль, вказуючи логін, пароль і свій Monobank API-ключ для отримання актуальних фінансових даних. Реєстрація повинна бути інтуїтивною та захищеною.
    - Вхід у систему: вхід до додатку проходитиме із застосуванням сучасних методів захисту, включно з хешуванням паролів, щоб гарантувати, що доступ має лише власник облікового запису.
    - Керування профілем: користувачі матимуть можливість оновлювати особисту інформацію (логін та пароль) і гнучко налаштовувати параметри свого облікового запису.
2. Відстеження та управління витратами:
    - Банківська інтеграція: система автоматично імпортуватиме транзакції через REST API Monobank, миттєво оновлюючи інформацію про витрати.
    - Гнучка категоризація витрат: для зручності аналізу користувачі зможуть сортувати витрати за попередньо встановленими категоріями, а також створювати власні категорії під свої потреби.
    - Ручне коригування: окрім автоматичного імпорту та категоризації витрат, користувачі матимуть можливість корегувати категорії витрат вручну.
3. Візуалізація даних та інформаційна панель:

- Інформаційна панель: на головній сторінці буде відображатися огляд основних фінансових метрик — загальна сума витрат, розподіл витрат по категоріям.
- Динамічні звіти і графіки: користувачі зможуть глибше аналізувати свої фінанси через інтерактивні діаграми, які відображатимуть порівняння між різними категоріями та періодами.

4. Безпека та конфіденційність даних:
   - Захист даних: вся чутлива інформація (паролі, API-ключі) хешуватиметься або шифруватиметься під час передачі і зберігання, щоб уникнути несанкціонованого доступу.
   - Контроль доступу: використання токенів JWT забезпечить надійну автентифікацію і авторизацію, гарантуватиме, що кожен користувач бачить тільки свої дані.

На рисунку 6 подано діаграму варіантів використання (use case), яка демонструє основні дії, що доступні користувачеві у взаємодії з додатком:

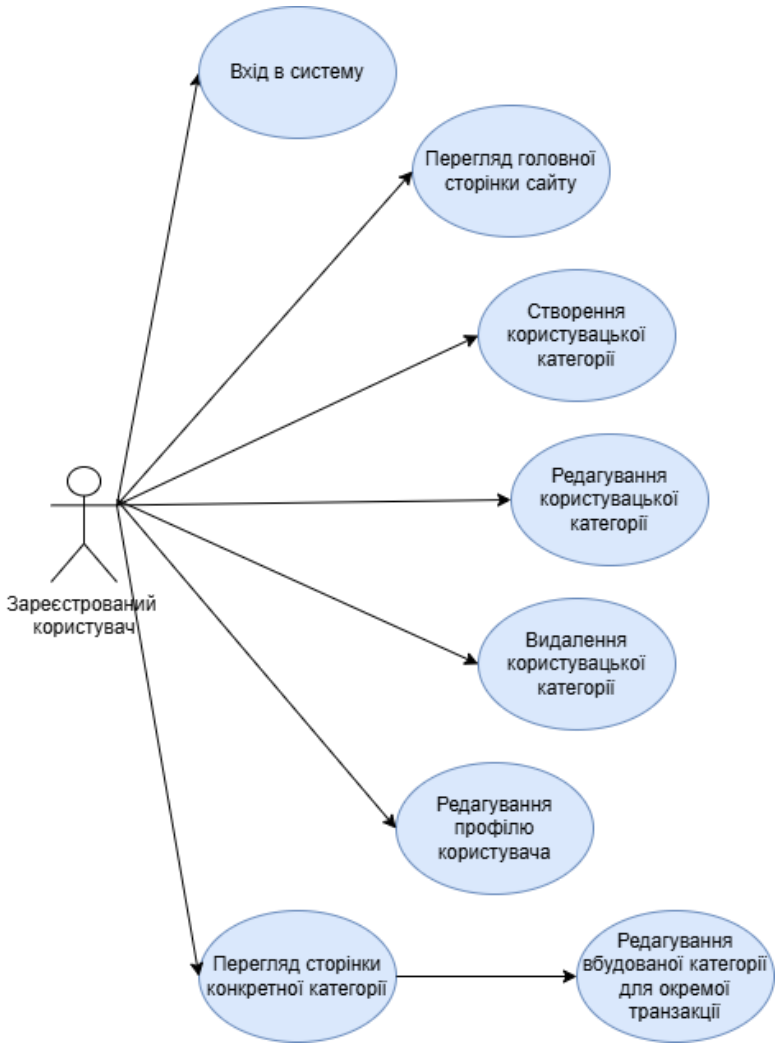


Рисунок 6 – *Use Case діаграма для взаємодії користувача.*

Завдяки цим можливостям веб-додаток стане не лише зручним інструментом для щоденного контролю бюджету, але й надійним помічником у плануванні фінансів. Користувачі зможуть легко аналізувати свої витрати, приймати свідомі рішення та краще розуміти свої фінансові звички, що сприятиме підвищенню їхнього фінансового добробуту.

3.4 Вимоги до інтерфейсу

Користувацький інтерфейс (UI) веб-додатку для відстеження витрат повинен відповідати ряду вимог, які забезпечують оптимальний досвід для користувачів, що керують своїми особистими фінансами. Ці вимоги визначають пріоритети зручності, естетики та доступності, а також

відповідають технічним вимогам інтеграції з банківськими послугами в режимі реального часу та статистичного аналізу. Нижче наведено основні вимоги до інтерфейсу, кожна з яких покликана підвищити задоволеність користувачів та підтримати основні функції додатку:

- Простота використання: інтерфейс повинен бути інтуїтивно зрозумілим і простим, що дозволить користувачам швидко отримати доступ до ключових функцій, таких як перегляд транзакцій, управління категоріями та аналітичні панелі.
- Естетика та привабливість: сучасний, професійний дизайн має бути використаний для залучення користувачів та передачі надійності.
- Адаптивність: інтерфейс повинен бути повністю адаптивним, забезпечуючи безперебійну функціональність на різних пристроях, включаючи настільні комп'ютери, планшети та смартфони.
- Навігація: навігація повинна бути зрозумілою та доступною, з добре структурованою системою меню, яка дозволить користувачам легко перемикатися між такими розділами, як списки транзакцій, налаштування категорій та статистичні панелі.
- Користувацький досвід: інтерфейс повинен забезпечувати інтерактивний та захоплюючий досвід завдяки використанню адаптивних елементів, таких як кнопки, форми та візуалізації даних (наприклад, діаграми та графіки).
- Доступність: додаток повинен відповідати стандартам веб-доступності для забезпечення інклюзивності для користувачів з обмеженими можливостями.
- Швидкість та продуктивність: інтерфейс повинен бути оптимізований для швидкого завантаження сторінок і швидкої взаємодії, мінімізуючи затримки для підтримки залученості користувачів.
- Відображення інформації: фінансові дані, включаючи деталі транзакцій та статистичну інформацію, повинні бути представлені в чіткій, структурованій формі.

Загалом, ці вимоги до інтерфейсу відіграватимуть ключову роль у створенні зручного та зрозумілого для користувача веб-додатку. Вони допоможуть зробити не лише сам процес відстеження витрат простим і логічним, але й забезпечать комфортне користування додатком у повсякденному житті.

3.5 Реалізація

Реалізація веб-додатку для відстеження витрат розроблена з використанням модульної, багаторівневої архітектури, що забезпечує розподіл завдань, масштабованість та зручність обслуговування. Система складається з трьох основних проектів: ExpenseTracker.Data для збереження даних, ExpenseTracker.Web для користувацького інтерфейсу та ExpenseTracker.API для бізнес-логіки та зовнішніх інтеграцій. Побудований на ASP.NET Core, додаток використовує Entity Framework Core для доступу до даних, Razor Pages для динамічного рендерингу інтерфейсу та RESTful API для безперешкодного зв'язку із зовнішніми сервісами, такими як Monobank API. Така структура полегшує незалежну розробку, тестування та розгортання кожного модуля, зберігаючи при цьому уніфіковану систему завдяки чітко визначеним інтерфейсам та ін'єкції залежностей. PostgreSQL слугує реляційною базою даних, обраною завдяки своїй надійності та підтримці складних запитів, тоді як клієнтські розширення, такі як Chart.js, забезпечують інтерактивність та оновлення в реальному часі.

Модульний дизайн проекту відображається в його структурі каталогів, де кожен модуль відповідає певній ролі:

1. ExpenseTracker.Data (рівень даних): керує взаємодією з базами даних та еволюцією схем.
    - Migrations/: зберігає файли міграції EF Core для відстеження та поступового застосування змін схеми. Такий підхід гарантує, що база даних розвивається разом з додатком без втрати даних.

- Model/: визначає класи сутностей (User, Transaction, Category, TransactionCategory, CategoryRelationship.) з властивостями та навігаційними властивостями для зв'язків.
- AppDbContext.cs: централізує конфігурацію бази даних за допомогою DbContext EF Core. Він використовує Fluent API для визначення зв'язків, встановлення обмежень і налаштування рядка з'єднання з PostgreSQL. Ця абстракція спрощує доступ до даних і забезпечує узгодженість у всьому додатку.

2. ExpenseTracker.Web (презентаційний рівень): надає інтуїтивно зрозумілий, чуйний користувацький інтерфейс.
   - wwwroot/: розміщує користувацькі таблиці стилів для адаптивного дизайну та тем, клієнтські скрипти для інтерактивності, включаючи маніпуляції з DOM, іконки та брендингові ресурси, зовнішні бібліотеки, такі як Chart.js для візуалізації.
   - Pages/: містить сторінки Login.cshtml і Register.cshtml з моделями сторінок, які обробляють автентифікацію користувачів за допомогою викликів API до UserController, а також сторінка MainPage.cshtml, що відображає транзакції та графіки, та реалізує створення, редагування, та видалення кастомізованих категорій за допомогою викликів API до TransactionController та CategoryController.
   - Program.cs: конфігурує веб-хост, конвеєр проміжного програмного забезпечення та ін'єкцію залежностей для таких сервісів, як автентифікація.
3. ExpenseTracker.API (рівень інтеграції та бізнес-логіки): централізує бізнес-логіку та зовнішні інтеграції.
   - Controllers/: реалізує RESTful API контролери, що керують категоріями, імпортують транзакції, обробляють транзакції, включаючи посторінкове виведення, керують автентифікацією.
   - DTO/: містить класи, які зіставляють сутності з відповідями API, зменшуючи ризик витоку даних.

- Interfaces/: містить інтерфейси репозиторіїв для забезпечення ізоляції бізнес-логіки від реалізації доступу до даних.
- Repositories/: реалізує доступ до даних, ізолюючи іншу частину програми від ORM.
- Sevices/: реалізує автоматизований імпорт транзакцій в MonobankService та автоматичную категоризацію транзакцій в TransactionCategorizationService.
- Program.cs: конфігурує хост API, CORS та ін'єкцію залежностей.

Рисунок 7 відображає структуру розробленого рішення.

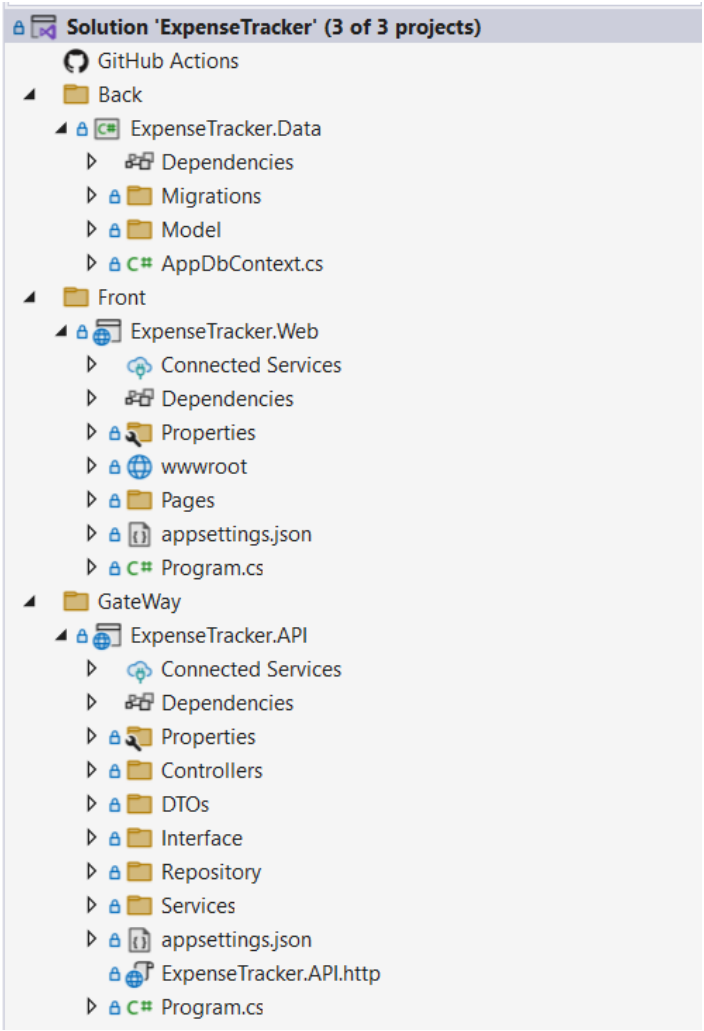


Рисунок 7 – *Структура веб-додатку трекеру витрат.*

Модель даних розроблена для гнучкості та продуктивності, з п’ятьма основними сутностями, сконфігурованими в AppDbContext:

- User: зберігає дані облікового запису користувача. Вона слугує батьківською таблицею для специфічних для користувача транзакцій і категорій.

Лістинг *1 — Модель User.*

```
public class User
{
    public Guid Id { get; set; } = new Guid();
    [Required]
    public string Login { get; set; }
    public string PasswordHash { get; set; }
    public string Token { get; set; }
}
```

- Transactions: записує деталі витрат. Пов'язується з категоріями через таблицю TransactionCategories.

Лістинг *2 — Модель Transaction.*

```
public class Transaction
{
    public Guid Id { get; set; }
    public Guid UserId { get; set; }
    public User User { get; set; }
    [Required]
    public string Description { get; set; }
    public decimal Amount { get; set; }
    public Guid CategoryId { get; set; } // Base (built-in) cate-
gory
    public Category Category { get; set; }
    private DateTime _date;
    public DateTime Date
    {
        get => _date;
        set => _date = value.Kind == DateTimeKind.Unspecified ?
value.ToUniversalTime() : value;
    }
    public int? MccCode { get; set; }
    public string TransactionType { get; set; }
    public bool IsManuallyCategorized { get; set; }
    public List<TransactionCategory> TransactionCategories { get;
set; } = new List<TransactionCategory>();
}
```

- Categories: керує класифікацією витрат. Підтримує створення користувацьких категорій.

Лістинг *3 — Модель Category.*

```
public class Category
{
    public Guid Id { get; set; }
    [Required]
    public string Name { get; set; }
    public string MccCodes { get; set; }
    public Guid? UserId { get; set; } // Null for built-in,
user ID for custom
    public bool IsBuiltIn { get; set; } // True for built-in
categories
    [NotMapped]
    public int[] MccCodesArray
    {
        get => string.IsNullOrEmpty(MccCodes) ? Ar-
ray.Empty<int>() : JsonSerializer.Deserialize<int[]>(MccCodes);
        set => MccCodes = JsonSerializer.Serialize(value);
    }
}
```

- TransactionCategories: сприяє встановленню зв'язків «багато-до-багатьох» між транзакціями і категоріями.

Лістинг *4 — Модель TransactionCategories.*

```
 public class TransactionCategory
{
    public Guid TransactionId { get; set; }
    public Guid CategoryId { get; set; }
    public bool IsBaseCategory { get; set; }
    public Transaction Transaction { get; set; }
    public Category Category { get; set; }
}
```

- CategoryRelationships: визначає ієрархічні структури категорій, уможливлюючи зв'язки «батько-дочка».

Лістинг *5 — Модель CategoryRelationships.*

```
public class CategoryRelationship
{
    public Guid CustomCategoryId { get; set; }
    public Guid BaseCategoryId { get; set; }
    public Category CustomCategory { get; set; }
    public Category BaseCategory { get; set; }
}
```

Конфігурації Fluent API в AppDbContext забезпечують цілісність даних. Складений індекс на Categories забезпечує унікальні назви категорій для кожного користувача, а індекси на Transactions.Date та TransactionCategories.CategoryId оптимізують продуктивність запитів.

Рисунок 8 відображає зв'язки між сутностями розробленої бази даних.

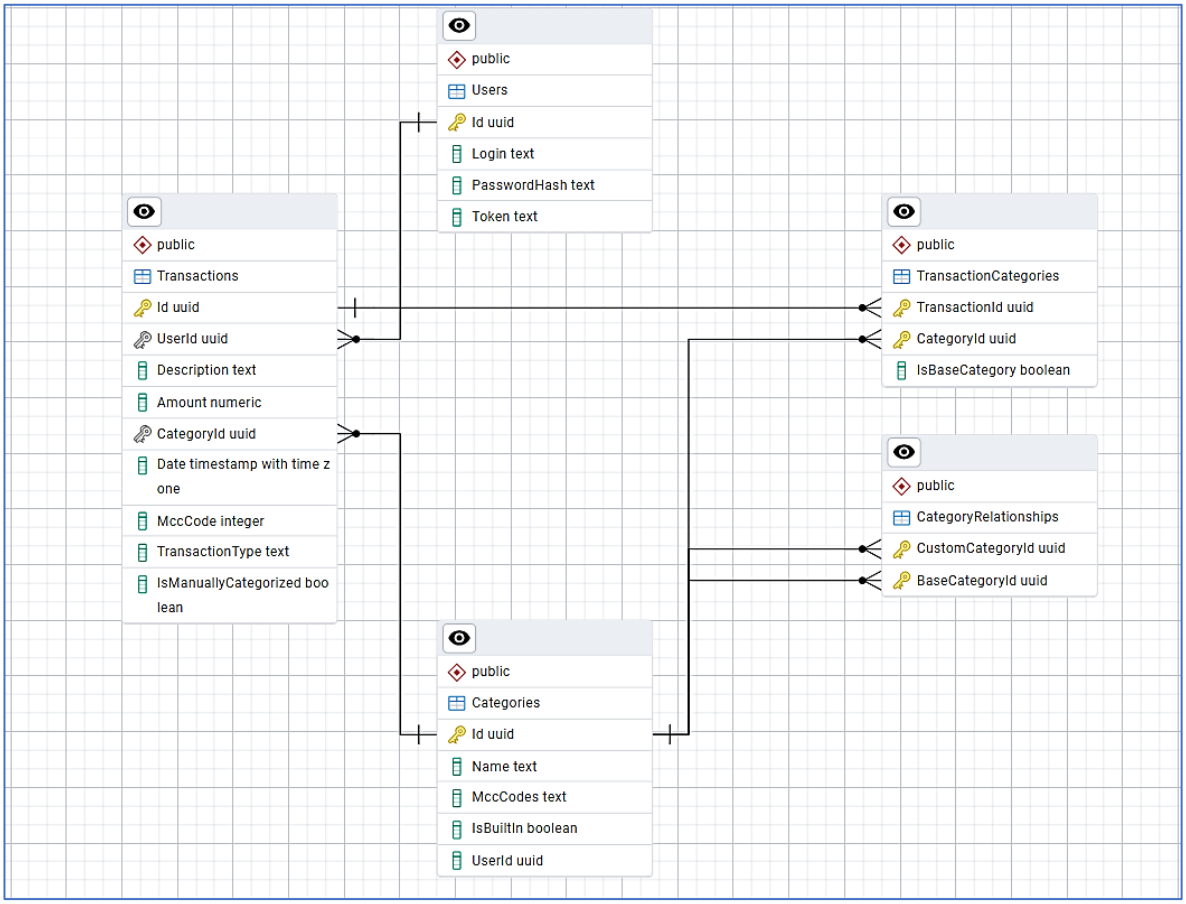


Рисунок 8 – *Діаграма зв'язків між сутностями.*

Рівень доступу до даних в ExpenseTracker.API/Repository використовує патерн репозиторіїв для інкапсуляції операцій EF Core:

- TransactionRepository: реалізує ITransactionRepository з методами, що відповідають за отримання транзакцій з бази даних для конкретного користувача за певний проміжок часу, отримання транзакцій з бази даних для конкретної категорії, створення та видалення зв'язків транзакцій з користувацькою категорією, редагування базової категорії, до якої належить транзакція.

Лістинг 6 демонструє метод UpdateCategoryForTransactionsAsync, який реалізує створення зв’язку між певною кількістю транзакцій, що належить базовій категорії або категоріям та створюваної користувачем категорії.

Лістинг *6 — Метод UpdateCategoryForTransactionsAsync.*

```
public async Task UpdateCategoryForTransactionsAsync(Guid
categoryId, List<Guid> transactionIds)
{
    if (transactionIds == null || !transactionIds.Any())
    {
        throw new ArgumentException("Transaction IDs list cannot
be empty.", nameof(transactionIds));
    }

    if (categoryId == Guid.Empty)
    {
        throw new ArgumentException("Category ID cannot be
empty.", nameof(categoryId));
    }

    using var transaction = await
_context.Database.BeginTransactionAsync();
    try
    {
        var categoryExists = await
_context.Categories.AnyAsync(c => c.Id == categoryId);
        if (!categoryExists)
        {
            throw new InvalidOperationException($"Category with
ID {categoryId} does not exist.");
        }

        await _context.TransactionCategories
            .Where(tc =>
transactionIds.Contains(tc.TransactionId) && tc.IsBaseCategory)
            .ExecuteUpdateAsync(s => s
                .SetProperty(tc => tc.CategoryId, categoryId));

        await _context.Transactions
            .Where(t => transactionIds.Contains(t.Id))
            .ExecuteUpdateAsync(s => s
                .SetProperty(t => t.CategoryId, categoryId));

        var currentCustomCategories = await
_context.TransactionCategories
            .Where(tc =>
transactionIds.Contains(tc.TransactionId) && !tc.IsBaseCategory)
            .Select(tc => new { tc.TransactionId, tc.CategoryId
})
```

```
            .ToListAsync();

        var validCustomCategoryIds = await
_context.CategoryRelationships
            .Where(cr => cr.BaseCategoryId == categoryId)
            .Select(cr => cr.CustomCategoryId)
            .ToListAsync();

        var customCategoriesToRemove = currentCustomCategories
            .Where(tc =>
!validCustomCategoryIds.Contains(tc.CategoryId))
            .ToList();

        if (customCategoriesToRemove.Any())
        {
            await _context.TransactionCategories
                .Where(tc =>
transactionIds.Contains(tc.TransactionId) &&
                               !tc.IsBaseCategory &&

!validCustomCategoryIds.Contains(tc.CategoryId))
                .ExecuteDeleteAsync();
        }

        var newMappings = new List<TransactionCategory>();
        foreach (var transactionId in transactionIds)
        {
            foreach (var customCategoryId in
validCustomCategoryIds)
            {
                var existing = currentCustomCategories
                    .Any(tc => tc.TransactionId == transactionId
&& tc.CategoryId == customCategoryId);
                if (!existing)
                {
                    newMappings.Add(new TransactionCategory
                    {
                        TransactionId = transactionId,
                        CategoryId = customCategoryId,
                        IsBaseCategory = false
                    });
                }
            }
        }

        if (newMappings.Any())
        {
            await
_context.TransactionCategories.AddRangeAsync(newMappings);
        }
        await _context.SaveChangesAsync();
        await transaction.CommitAsync();
    }
```

```
    catch (Exception ex)
    {
        await transaction.RollbackAsync();
        throw;
    }
}
```

- CategoryRepository: реалізує ICategoryRepository, з методами, що повертають резервну категорію для некатегоризованих транзакцій, створюють та видаляють зв’язок між користувацькою та базовою категорією, повертають всі наянві категорії для певного користувача, повертає всі транзакції, що належать певній категорії.

Лістинг 7 демонструє метод AddRelationshipAsync, який реалізує створення зв’язку між користувацькою категорією та вбудованою базовою категорією з автоматичним записом MCC кодів базової категорії користувацькій для синхронізації майбутніх транзакцій з базової категорії зі створеної користувачем категорії.

Лістинг *7 — Метод AddRelationshipAsync.*

```
public async Task AddRelationshipAsync(Guid customCategoryId,
Guid baseCategoryId)
{
    var customCategory = await
_context.Categories.FindAsync(customCategoryId);
    if (customCategory == null)
    {
        throw new InvalidOperationException($"Custom category
{customCategoryId} not found.");
    }

    var baseCategory = await
_context.Categories.FindAsync(baseCategoryId);
    if (baseCategory == null || !baseCategory.IsBuiltIn)
    {
        throw new InvalidOperationException($"Base category
{baseCategoryId} is invalid.");
    }

    var relationship = new CategoryRelationship
    {
        CustomCategoryId = customCategoryId,
        BaseCategoryId = baseCategoryId
    };
    await _context.CategoryRelationships.AddAsync(relationship);
```

```
    var customMccCodes = customCategory.MccCodesArray.ToList();
    var baseMccCodes = baseCategory.MccCodesArray.ToList();

customMccCodes.AddRange(baseMccCodes.Except(customMccCodes));
    customCategory.MccCodesArray = customMccCodes.ToArray();
    _context.Categories.Update(customCategory);

    await _context.SaveChangesAsync();

    var baseTransactions = await _context.TransactionCategories
        .Where(tc => tc.CategoryId == baseCategoryId &&
tc.Transaction.UserId == customCategory.UserId)
        .Select(tc => tc.Transaction)
        .Distinct()
        .ToListAsync();

    if (baseTransactions.Any())
    {
        var transactionCategories = baseTransactions.Select(t =>
new TransactionCategory
        {
            TransactionId = t.Id,
            CategoryId = customCategoryId,
            IsBaseCategory = false
        }).ToList();


_context.TransactionCategories.AddRange(transactionCategories);
        await _context.SaveChangesAsync();
    }
}
```

- UserRepository: реалізує IUserRepository з методами, що реалізують автентифікацію користувача при вході в систему та редагування користувачем свого логіну та паролю.

Лістинг 8 демонструє метод UpdateUserCredentialsAsync, який реалізує оновлення логіну та/або паролю для користувача з валідацією введених користувачем даних.

Лістинг *8 — Метод UpdateUserCredentialsAsync.*

```
        public async Task<bool> UpdateUserCredentialsAsync(Guid
userId, string? newLogin, string? newPassword)
        {
            var user = await _context.Users.FindAsync(userId);
            if (user == null)
                return false;
```

```
            if (!string.IsNullOrWhiteSpace(newLogin) && newLogin
!= user.Login)
            {
                if (await _context.Users.AnyAsync(u => u.Login
== newLogin && u.Id != userId))
                    return false;
                user.Login = newLogin;
            }

            if (!string.IsNullOrWhiteSpace(newPassword))
            {
                user.PasswordHash =
BCrypt.Net.BCrypt.HashPassword(newPassword);
            }

            if (!string.IsNullOrWhiteSpace(newLogin) ||
!string.IsNullOrWhiteSpace(newPassword))
            {
                await _context.SaveChangesAsync();
            }

            return true;
        }
```

- GenericRepository: забезпечує багаторазові CRUD-операції для всіх сутностей, зменшуючи дублювання коду. Демонструється в лістингу 9.

Лістинг *9 — Репозиторій GenericRepository.*

```
public class GenericRepository<T> : IGenericRepository<T> where
T : class
{
    protected readonly AppDbContext _context;
    protected readonly DbSet<T> _dbSet;

    public GenericRepository(AppDbContext context)
    {
        _context = context;
        _dbSet = context.Set<T>();
    }

    public async Task<IEnumerable<T>> GetAllAsync()
    {
        return await _dbSet.ToListAsync();
    }

    public async Task<T> GetByIdAsync(Guid id)
    {
        return await _dbSet.FindAsync(id);
    }
```

```
    public async Task<T> GetByNameAsync(string title)
    {
        return await _dbSet.FirstOrDefaultAsync(entity =>
EF.Property<string>(entity, "Name") == title);
    }

    public async Task AddAsync(T entity)
    {
        await _dbSet.AddAsync(entity);
        await _context.SaveChangesAsync();
    }

    public async Task UpdateAsync(T entity)
    {
        _dbSet.Update(entity);
        await _context.SaveChangesAsync();
    }

    public async Task DeleteAsync(Guid id)
    {
        var entity = await _dbSet.FindAsync(id);
        if (entity != null)
        {
            _dbSet.Remove(entity);
            await _context.SaveChangesAsync();
        }
    }

    public async Task<IEnumerable<T>>
FindAsync(Expression<Func<T, bool>> predicate)
    {
        return await _dbSet.Where(predicate).ToListAsync();
    }
}
```

ExpenseTracker.API/Controllers реалізує RESTful API:

- CategoryController: керує категоріями за допомогою методів, що повертають всі категорії користувача, відфільтровані за UserId. створюють визначені користувачем категорії з можливістю створити її на основі вбудованих категорій, редагують визначені користувачем категорії з можливістю змінити її ім’я та змінити приналежні базові категорії та видаляють створені користувачем категорії, каскадно до TransactionCategory.

Лістинг 10 демонструє метод UpdateCategory, який реалізує редагування створеної користувачем категорії, виконує зміну назви користувацької категорії, керує її зв’язками з базовими категоріями, оновлює MCC-коди та коригує категорії пов’язаних транзакцій.

Лістинг *10 — Контролер UpdateCategory.*

```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateCategory(Guid id, [From-
Body] UpdateCategoryDto updateCategoryDto)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var category = await _categoryRepository.GetByIdAsync(id);
    if (category == null)
    {
        return NotFound();
    }

    var userId = Guid.Parse(User.FindFirst(ClaimTypes.NameI-
dentifier)?.Value);
    if (category.IsBuiltIn || category.UserId != userId)
    {
        return Forbid("Cannot modify built-in or other user's
categories.");
    }

    var defaultCategory = await _categoryRepository.GetDe-
faultCategoryAsync();
    if (defaultCategory == null)
    {
        return StatusCode(500, "Default category 'Інше' not
found.");
    }
    var defaultCategoryId = defaultCategory.Id;

    category.Name = updateCategoryDto.Name;
    await _categoryRepository.UpdateAsync(category);

    var currentBaseCategoryIds = await _categoryRepository.Get-
BaseCategoryIdsAsync(id);
    var newBaseCategoryIds = updateCategoryDto.BaseCategoryIds
?? new List<Guid>();

    foreach (var baseCategoryId in newBaseCategoryIds)
    {
```

```
        var baseCategory = await _categoryRepository.GetBy-
IdAsync(baseCategoryId);
        if (baseCategory == null || !baseCategory.IsBuiltIn)
        {
            return BadRequest($"Invalid base category ID:
{baseCategoryId}");
        }
    }

    var baseCategoriesToRemove = currentBaseCategoryIds.Ex-
cept(newBaseCategoryIds).ToList();
    foreach (var baseCategoryId in baseCategoriesToRemove)
    {
        await _categoryRepository.RemoveRelationshipAsync(id,
baseCategoryId);
    }

    var baseCategoriesToAdd = newBaseCategoryIds.Except(current-
BaseCategoryIds).ToList();
    foreach (var baseCategoryId in baseCategoriesToAdd)
    {
        await _categoryRepository.AddRelationshipAsync(id,
baseCategoryId);
    }

    var allMccCodes = new List<int>();
    foreach (var baseId in newBaseCategoryIds)
    {
        var baseCategory = await _categoryRepository.GetBy-
IdAsync(baseId);
        allMccCodes.AddRange(baseCategory.MccCodesArray);
    }
    category.MccCodesArray = allMccCodes.Distinct().ToArray();
    await _categoryRepository.UpdateAsync(category);

    var transactionIds = await _categoryRepository.GetTransac-
tionIdsByCategoryIdAsync(id, userId);
    if (transactionIds.Any())
    {
        foreach (var baseId in baseCategoriesToRemove)
        {
            var baseTransactionIds = await _categoryReposi-
tory.GetTransactionIdsByCategoryIdAsync(baseId, userId);
            var transactionsToRemove = transactionIds.Inter-
sect(baseTransactionIds).ToList();
            if (transactionsToRemove.Any())
            {
                await _transactionRepository.RemoveTransaction-
CategoriesAsync(id, transactionsToRemove);
            }
        }

        foreach (var baseId in baseCategoriesToAdd)
```

```
        {
            var baseTransactionIds = await _categoryReposi-
tory.GetTransactionIdsByCategoryIdAsync(baseId, userId);
            var transactionsToAdd = baseTransactionIds.Ex-
cept(transactionIds).ToList();
            if (transactionsToAdd.Any())
            {
                var newCategories = transactionsToAdd.Select(tId
=> new TransactionCategory
                {
                    TransactionId = tId,
                    CategoryId = id,
                    IsBaseCategory = false
                }).ToList();
                await _transactionRepository.AddTransactionCate-
goriesAsync(newCategories);
            }
        }
        await _transactionRepository.ReassignOrphanedTransac-
tionsAsync(transactionIds, defaultCategoryId);
    }

    var categoryDto = new CategoryDto
    {
        Id = category.Id,
        Name = category.Name,
        MccCodes = category.MccCodesArray,
        IsBuiltIn = category.IsBuiltIn,
        UserId = category.UserId
    };

    return Ok(categoryDto);
}
```

- TransactionController: обробляє транзакції за допомогою методів, що отримують транзакції разом з їх категоріями та оновлюють відображення транзакцій відповідно до їх категорій, перевіряючи вхідні дані за допомогою ModelState.

Лістинг 11 демонструє метод UpdateTransactionCategories, який реалізує оновлення базової категорії для вибраної транзакції, перевіряючи валідність введених даних, належність транзакції користувачу та обмежуючи призначення лише до вбудованих категорій.

Лістинг *11 — Метод UpdateTransactionCategories.*

```
[HttpPost("update-category")]
```

```
    public async Task<IActionResult> UpdateTransactionCatego-
ries([FromBody] UpdateTransactionCategoryRequest request)
    {
        if (request.TransactionIds == null || !request.Transac-
tionIds.Any())
        {
            return BadRequest("Transaction IDs list cannot be
empty.");
        }

        if (request.CategoryId == Guid.Empty)
        {
            return BadRequest("Category ID cannot be empty.");
        }

        var category = await _categoryRepository.GetBy-
IdAsync(request.CategoryId);
        if (category != null && !category.IsBuiltIn)
        {
            return BadRequest("Base category must be built-
in.");
        }

        var currentUserId = Guid.Parse(User.Find-
First(ClaimTypes.NameIdentifier)?.Value);
        var invalidUserTransactions = await _transactionReposi-
tory.FindAsync(t => request.TransactionIds.Contains(t.Id) &&
t.UserId != currentUserId);
        if (invalidUserTransactions.Any())
        {
            return Forbid();
        }

        await _transactionRepository.UpdateCategoryForTransac-
tionsAsync(request.CategoryId, request.TransactionIds);
        return NoContent();
    }
}
```

- MonobankController: забезпечує імпорт транзакцій користувача через MonobankService.

Лістинг 12 демонструє метод GetMonobankTransactions, який реалізує отримання транзакцій з Monobank для вказаного користувача за останні 30 днів. Метод перевіряє наявність користувача та його токена, розшифровує токен та викликає MonobankService сервіс для отримання транзакцій.

Лістинг *12 — Метод GetMonobankTransactions.*

```
[HttpGet("transactions/{userId}")]
```

```
public async Task<ActionResult<List<TransactionDto>>> GetMono-
bankTransactions(Guid userId)
{
    try
    {
        var user = await _userRepository.GetByIdAsync(userId);
        if (user == null)
        {
            return NotFound("Користувач не знайдений.");
        }
        if (string.IsNullOrEmpty(user.Token))
        {
            return BadRequest("У користувача відсутній API-ключ
Monobank.");
        }

        string decryptedToken;
        try
        {
            decryptedToken = _protector.Unprotect(user.Token);
        }
        catch (Exception ex)
        {
            return BadRequest("Помилка під час розшифрування
токена.");
        }

        var now = DateTimeOffset.UtcNow;
        var fromTimestamp = now.AddDays(-30).ToUnixTimeSec-
onds();
        var toTimestamp = now.ToUnixTimeSeconds();

        var transactions = await _monobankService.GetTransac-
tionsAsync(user.Id, decryptedToken, fromTimestamp, toTimestamp);
        var transactionDtos = transactions.Select(t => new
TransactionDto
        {
            Id = t.Id,
            Description = t.Description,
            Amount = t.Amount,
            Date = t.Date,
            CategoryId = t.CategoryId,
            CategoryName = t.Category?.Name,
            UserId = t.UserId,
            TransactionType = t.TransactionType,
            MccCode = t.MccCode,
            IsManuallyCategorized = t.IsManuallyCategorized,
            TransactionCategories = t.TransactionCategories.Se-
lect(tc => new TransactionCategoryDto
            {
                CategoryId = tc.CategoryId,
                CategoryName = tc.Category?.Name,
                IsBaseCategory = tc.IsBaseCategory
```

```
            }).ToList()
        }).ToList();
        return Ok(transactionDtos);
    }
    catch (Exception ex)
    {
        return StatusCode(500, $"Помилка під час отримання
транзакцій: {ex.Message}");
    }
}
```

- UserController: забезпечує реєстрацію, вхід, вихід, оновлення облікових даних і отримання поточного користувача. Працює з JWT для автентифікації та шифрує API-токени Monobank.

Лістинг 13 демонструє метод UpdateUserCredentials, який дозволяє авторизованому користувачу змінити логін і/або пароль після перевірки поточного пароля та валідації нових даних. У разі зміни логіна — генерується новий JWT-токен.

Лістинг *13 — Метод UpdateUserCredentials.*

```
[Authorize]
[HttpPut("user")]
public async Task<IActionResult> UpdateUserCredentials([From-
Body] UpdateUserCredentialsRequest request)
{
    var userId = User.FindFirst(ClaimTypes.NameIdenti-
fier)?.Value;
    if (userId == null)
    {
        return Unauthorized(new { message = "Користувач не
авторизований" });
    }

    var user = await _userRepository.GetBy-
IdAsync(Guid.Parse(userId));
    if (user == null)
    {
        return NotFound(new { message = "Користувача не
знайдено" });
    }

    if (!BCrypt.Net.BCrypt.Verify(request.CurrentPassword,
user.PasswordHash))
    {
        return BadRequest(new { message = "Невірний поточний
пароль" });
    }
```

```
    if (string.IsNullOrWhiteSpace(request.NewLogin) &&
string.IsNullOrWhiteSpace(request.NewPassword))
    {
        return BadRequest(new { message = "Потрібно вказати
новий логін або пароль" });
    }

    if (!string.IsNullOrWhiteSpace(request.NewPassword) && re-
quest.NewPassword != request.ConfirmNewPassword)
    {
        return BadRequest(new { message = "Новий пароль і
підтвердження не співпадають" });
    }

    var success = await _userRepository.UpdateUserCreden-
tialsAsync(Guid.Parse(userId), request.NewLogin, request.New-
Password);
    if (!success)
    {
        return BadRequest(new { message = "Не вдалося оновити
дані. Логін може бути вже зайнятий." });
    }

    if (!string.IsNullOrWhiteSpace(request.NewLogin))
    {
        var newToken = GenerateJwtToken(user);
        return Ok(new { message = "Логін і/або пароль успішно
оновлено", token = newToken });
    }

    return Ok(new { message = "Пароль успішно оновлено" });
}
```

Задля зменшення розміру корисного навантаження в програмі широко використовуються DTO (Data Transfer Object), що виключають непотрібні або конфіденційні поля, які не повинні передаватися через API. Це підвищує безпеку і також дозволяє оптимізувати обсяг інформації, що передається між різними частинами програми, зменшуючи навантаження на мережу та пришвидшуючи обмін даними.

Сервіси в ExpenseTracker.API/Services інкапсулюють логіку домену:

- TransactionCategorizationService: реалізує призначення категорій на основі кодів MCC. Словник зіставляє MCC коди з MCC кодами вбудованих категорій, що зберігається в базі даних або призначає категорію за замовчуванням в разі неспівпадіння.

Лістинг *14 — Сервіс TransactionCategorizationService.*

```
    public class TransactionCategorizationService
    {
        private readonly ICategoryRepository _categoryReposi-
tory;
        private readonly ILogger<TransactionCategorization-
Service> _logger;

        public TransactionCategorizationService(
            ICategoryRepository categoryRepository,
            ILogger<TransactionCategorizationService> logger)
        {
            _categoryRepository = categoryRepository ?? throw
new ArgumentNullException(nameof(categoryRepository));
            _logger = logger;
        }

        public async Task<Guid> CategorizeTransactionAsync(int?
mccCode)
        {
            _logger.LogDebug("Categorizing transaction with MCC
{MccCode}", mccCode);

            if (!mccCode.HasValue || mccCode == 0)
            {
                _logger.LogDebug("MCC is null or 0, using de-
fault category");
                return (await _categoryRepository.GetDefaultCat-
egoryAsync())?.Id ?? throw new InvalidOperationException("De-
fault category not found.");
            }

            var categories = await _categoryRepository.GetAllA-
sync(null); // Built-in only
            var matchingCategory = categories.FirstOrDefault(c
=>
                !string.IsNullOrEmpty(c.MccCodes) &&
c.MccCodesArray.Contains(mccCode.Value));

            if (matchingCategory != null)
            {
                _logger.LogDebug("Found matching category {Cate-
goryId} for MCC {MccCode}", matchingCategory.Id, mccCode);
                return matchingCategory.Id;
            }

            _logger.LogDebug("No matching category for MCC
{MccCode}, using default", mccCode);
            return (await _categoryRepository.GetDefaultCate-
goryAsync())?.Id ?? throw new InvalidOperationException("Default
category not found.");
```

```
        }
}
```

- MonobankService: реалізує отримання банківських транзакцій. Демонструється в лістингу 15.

Лістинг *15 — Сервіс MonobankService.*

```
public class MonobankService
{
    private readonly HttpClient _httpClient;
    private readonly ITransactionRepository
_transactionRepository;
    private readonly ICategoryRepository _categoryRepository;
    private readonly TransactionCategorizationService
_categorizationService;
    private readonly ILogger<MonobankService> _logger;
    private const string BaseUrl = "https://api.monobank.ua";

    public MonobankService(
        HttpClient httpClient,
        ITransactionRepository transactionRepository,
        ICategoryRepository categoryRepository,
        TransactionCategorizationService categorizationService,
        ILogger<MonobankService> logger)
    {
        _httpClient = httpClient;
        _transactionRepository = transactionRepository;
        _categoryRepository = categoryRepository;
        _categorizationService = categorizationService;
        _logger = logger;
    }

    public async Task<IEnumerable<Transaction>> GetTransac-
tionsAsync(Guid userId, string token, long fromTimestamp, long
toTimestamp)
    {
        _logger.LogInformation("Fetching transactions for user
{UserId} from {FromTimestamp} to {ToTimestamp}", userId,
fromTimestamp, toTimestamp);

        try
        {
            if (string.IsNullOrEmpty(token))
            {
                _logger.LogError("Token is empty for user
{UserId}", userId);
                throw new ArgumentException("Monobank token can-
not be empty.");
            }
```

```
            var request = new HttpRequestMessage(HttpMethod.Get,
$"{BaseUrl}/personal/statement/0/{fromTimestamp}/{to-
Timestamp}");
            request.Headers.Add("X-Token", token);
            _logger.LogDebug("Sending request to {Url}", re-
quest.RequestUri);

            var response = await _httpClient.SendAsync(request);
            if (!response.IsSuccessStatusCode)
            {
                var errorContent = await
response.Content.ReadAsStringAsync();
                _logger.LogError("Monobank API error: {Status-
Code} - {Error}", response.StatusCode, errorContent);
                throw new HttpRequestException($"Monobank API
request failed: {errorContent}");
            }

            var content = await response.Content.ReadAsStrin-
gAsync();
            _logger.LogDebug("Received response: {ContentLength}
chars", content.Length);

            var monobankTransactions = JsonSerializer.Deserial-
ize<List<MonobankTransaction>>(content, new JsonSerializerOp-
tions
            {
                PropertyNameCaseInsensitive = true
            });

            if (monobankTransactions == null || !monobankTrans-
actions.Any())
            {
                _logger.LogInformation("No transactions found
for user {UserId}", userId);
                return new List<Transaction>();
            }

            _logger.LogInformation("Deserialized {Count} trans-
actions", monobankTransactions.Count);

            var fromDate = DateTimeOffset.FromUnixTimeSec-
onds(fromTimestamp).UtcDateTime;
            var toDate = DateTimeOffset.FromUnixTimeSeconds(to-
Timestamp).UtcDateTime;
            var existingTransactions = await _transactionReposi-
tory.GetTransactionsByUserAndDateAsync(userId, fromDate, to-
Date);
            _logger.LogDebug("Found {Count} existing transac-
tions", existingTransactions.Count);

            var transactionsToSave = new List<Transaction>();
```

```
            foreach (var t in monobankTransactions)
            {
                var transactionDate = DateTimeOffset.FromUnix-
TimeSeconds(t.Time).UtcDateTime;

                var existingTransaction = existingTransac-
tions.FirstOrDefault(tx =>
                    Math.Abs(tx.Amount - (t.Amount / 100m)) <
0.01m &&
                    Math.Abs((tx.Date - transactionDate).Total-
Seconds) < 1 &&
                    tx.TransactionType == (t.Amount < 0 ? "Ex-
pense" : "Income"));

                if (existingTransaction != null)
                {
                    _logger.LogDebug("Skipping existing transac-
tion: {Description}", t.Description);
                    continue;
                }

                var transaction = new Transaction
                {
                    Id = Guid.NewGuid(),
                    UserId = userId,
                    Description = t.Description ?? "",
                    Amount = t.Amount / 100m,
                    Date = transactionDate,
                    MccCode = t.MccCode,
                    TransactionType = t.Amount < 0 ? "Expense" :
"Income",
                    IsManuallyCategorized = false,
                    CategoryId = await _categorization-
Service.CategorizeTransactionAsync(t.MccCode)
                };

                transactionsToSave.Add(transaction);
            }

            if (transactionsToSave.Any())
            {
                _logger.LogInformation("Saving {Count} new
transactions", transactionsToSave.Count);
                await _transactionRepository.AddRan-
geAsync(transactionsToSave);
                _logger.LogInformation("Saved transactions suc-
cessfully");
            }

            return existingTransactions.Concat(transac-
tionsToSave).ToList();
        }
        catch (ArgumentException ex)
```

```
        {
            _logger.LogError(ex, "Validation error");
            throw;
        }
        catch (HttpRequestException ex)
        {
            _logger.LogError(ex, "Monobank API error");
            throw;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Unexpected error");
            throw;
        }
    }
}
```

Інтерфейс користувача в ExpenseTracker.Web/Pages використовує Razor Pages для динамічного рендерингу:

- MainPage.cshtml: отримує дані про транзакції для користувача, відображає деталізовану статистику витрат та доходів розподілену на різні категорії або на конкретні транзакції для обраної категорії. Реалізує редагування базових категорій для кожної окремої транзакції, створення нових користувацьких категорій, редагування користувацьких категорій, видалення користувацьких категорій та редагування логіну та паролю користувача.

Лістинг 16 демонструє метод OnPostDeleteCategoryAsync, який відповідає за видалення користувацьких категорій. Він виконує автентифікацію користувача через JWT-токен та надсилає HTTP-запит до API на видалення певної категорії за її ID.

Лістинг *16 — Метод OnGetAsync.*

```
        public async Task<IActionResult>
OnPostDeleteCategoryAsync(string categoryId)
        {
            var jwt = Request.Cookies["jwt"];
            if (string.IsNullOrEmpty(jwt))
            {
                return RedirectToPage("/Auth/Login");
            }

            if (!Guid.TryParse(categoryId, out var id))
```

```
            {
                CategoryDeleteError = "Некоректний ідентифікатор
категорії.";
                return Page();
            }

            var client =
_httpClientFactory.CreateClient("ExpenseTrackerApi");
            client.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", jwt);

            try
            {
                var response = await
client.DeleteAsync($"api/category/{id}");
                if (response.IsSuccessStatusCode)
                {
                    return RedirectToPage();
                }
                var responseBody = await
response.Content.ReadAsStringAsync();
                switch ((int)response.StatusCode)
                {
                    case 404:
                        CategoryDeleteError = "Категорію не
знайдено.";
                        break;
                    case 403:
                        CategoryDeleteError = "Неможливо вида-
лити базову категорію або категорію іншого користувача.";
                        break;
                    case 500:
                        CategoryDeleteError = "Базова категорія
'Інше' не знайдена.";
                        break;
                    default:
                        CategoryDeleteError = "Помилка сервера
при видаленні категорії.";
                        break;
                }
            }
            catch (Exception ex)
            {
                CategoryDeleteError = "Помилка при з'єднанні з
сервером.";
            }

            return Page();
        }
```

- Login.cshtml: Обробляє введенні облікові дані та виконує вхід у систему, зберігаючи JWT у файлах cookie.

Обробка введенних даних та вхід в систему реалізується за допомогою методу OnPostAsync, що демонструється в лістингу 17.

Лістинг *17 — Метод OnPostAsync.*

```
        public async Task<IActionResult> OnPostAsync()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    try
    {
        var loginData = new { Login, Password };
        var client =
_httpClientFactory.CreateClient("ExpenseTrackerApi");
        var response = await
client.PostAsJsonAsync("api/auth/login", loginData);

        if (response.IsSuccessStatusCode)
        {
            var authResponse = await
response.Content.ReadFromJsonAsync<AuthResponse>();
            if (authResponse?.Token != null)
            {
                Response.Cookies.Append("jwt",
authResponse.Token, new CookieOptions
                {
                    HttpOnly = true,
                    Secure = true,
                    SameSite = SameSiteMode.Strict,
                    Path = "/",
                    Expires = DateTimeOffset.UtcNow.AddDays(1)
                });
                return RedirectToPage("/MainPage");
            }
            ErrorMessage = "Отримано некоректну відповідь від
сервера.";
            return Page();
        }
        else
        {
            var errorResponse = await
response.Content.ReadFromJsonAsync<ErrorResponse>();
            ErrorMessage = errorResponse?.Message ?? "Невірний
логін або пароль.";
            ModelState.AddModelError(string.Empty,
ErrorMessage);
            return Page();
        }
```

```
    }
    catch (Exception ex)
    {
        ErrorMessage = $"Помилка: {ex.Message}";
        ModelState.AddModelError(string.Empty, ErrorMessage);
        return Page();
    }
}
```

- Register.cshtml: Створює нові облікові записи, з перевіркою коректності введених даних через запит до API в UserController.

Створення нових облікових записів реалізується за допомогою методу OnPostAsync, що демонструється в лістингу 18.

Лістинг *18 — Метод OnPostAsync.*

```
public async Task<IActionResult> OnPostAsync()
 {
     if (!ModelState.IsValid)
     {
         return Page();
     }

     try
     {
         var registerRequest = new { Login, Password, Token };
         var                          client                          =
_httpClientFactory.CreateClient(«ExpenseTrackerApi»);
         var        response        =        await        cli-
ent.PostAsJsonAsync(«api/auth/register», registerRequest);

         if (response.IsSuccessStatusCode)
         {
             return RedirectToPage(«/Auth/Login»);
         }
         else
         {
             var  errorResponse  =  await  response.Content.Read-
FromJsonAsync<ErrorResponse>();
             ErrorMessage  =  errorResponse?.Message  ??  «Помилка
реєстрації. Перевірте введені дані.»;
             ModelState.AddModelError(string.Empty,          Er-
rorMessage);
             return Page();
         }
     }
     catch (Exception ex)
     {
         ErrorMessage = $"Помилка: {ex.Message}»;
         ModelState.AddModelError(string.Empty, ErrorMessage);
```

```
            return Page();
        }
    }
```

3.6 Тестування

У веб-додатку «Трекер витрат» користувачі керують своїми особистими фінансами, відстежуючи щоденні витрати за допомогою різних видів графічного відображення витрат та доходів, змінюючи категорії транзакцій відповідно до потреб користувачів та створюючи користувацькы категорії для полегшення відстеження витрат та доходів за певними групами базових категорій.

Спочатку користувачі взаємодіють з додатком через інтерфейс реєстрації, де вони задають свій логін, пароль та API-ключ від Monobank для автоматизованого імпорту транзакцій (див. Рис. 9).

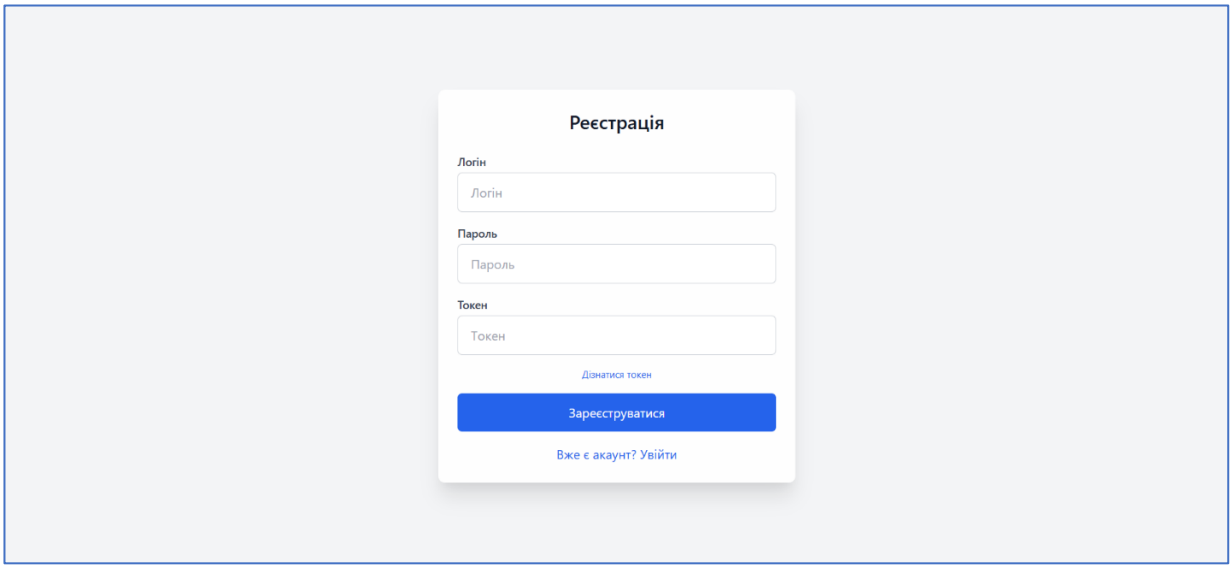


Рисунок 9 – *Інтерфейс реєстрації.*

Після успішної реєстрації користувачі потрапляють на сторінку сайту з інтерфейсом входу, де вони вводять свій логін та пароль (див. Рис. 10). Успішна автентифікація надає доступ до головної сторінки.

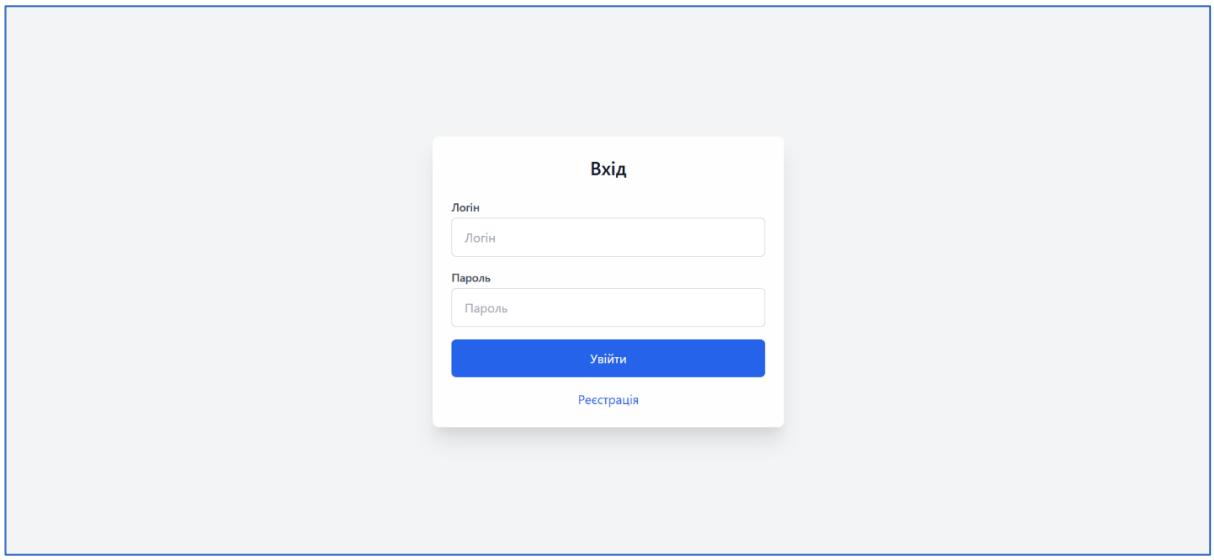


Рисунок 10 – *Інтерфейс входу.*

Після успішної автентифікації користувач потрапляє на головну сторінку сайту, що надає огляд фінансових даних та інтерфейс для керування відстеженням особистих фінансів. На рисунку 11 показана верхня частина головної сторінки з графіками витрат і надходжень розподілених за категоріями, кнопками для перемикання між різними видами графіків, списком категорій витрат та доходів, кнопкою виходу та кнопкою редагування профілю користувача.

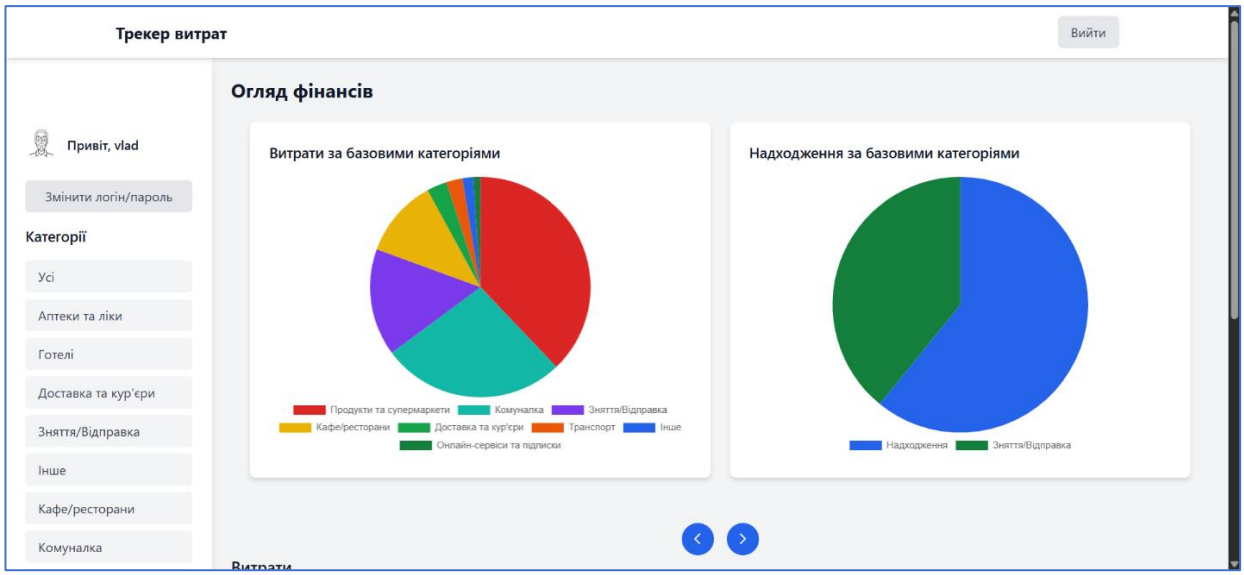


Рисунок 11 – *Інтерфейс верхньої частини головної сторінки.*

На рисунку 12 показана нижня частина головної сторінки з текстовим представленням витрат та надходжень, розподілених за вбудованими категоріями, кнопкою створення користувацьких категорій та створеними користувацькими категоріями, що виділяються специфічним кольором.

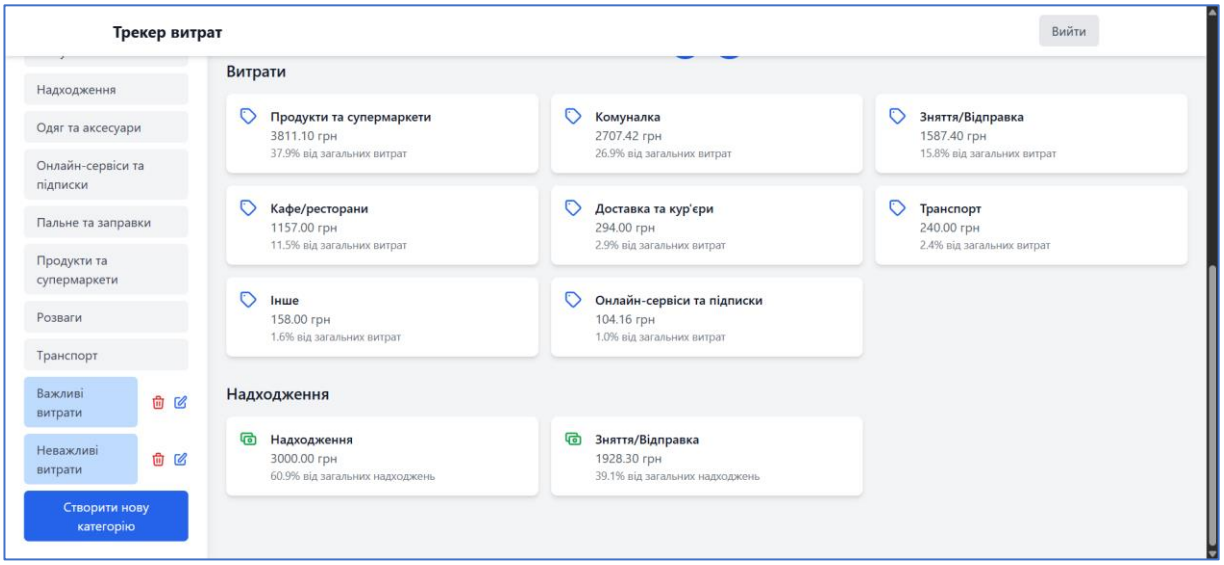


Рисунок 12 – *Інтерфейс нижньої частини головної сторінки.*

Перейшовши на сторінку конкретної категорії, користувачі можуть переглянути детальні графіки витрат і надходежнь, сегментовані за описами транзакцій, а також посторінковий список транзакцій з їх описом. На Рисунку 13 показана верхня частина сторінки категорії з графіками витрат і/або надходжень розподілених за описом транзакцій, з доданими кнопками для перемикання між різними видами графіків за необхідністю.



Рисунок 13 – *Інтерфейс верхньої частини сторінки категорії.*

На Рисунку 14 показана нижня частина сторінки категорії зі списком транзакцій за категорією з детальним описом транзакції та випадаючим меню для зміни базової категорії, до якої належить транзакція.

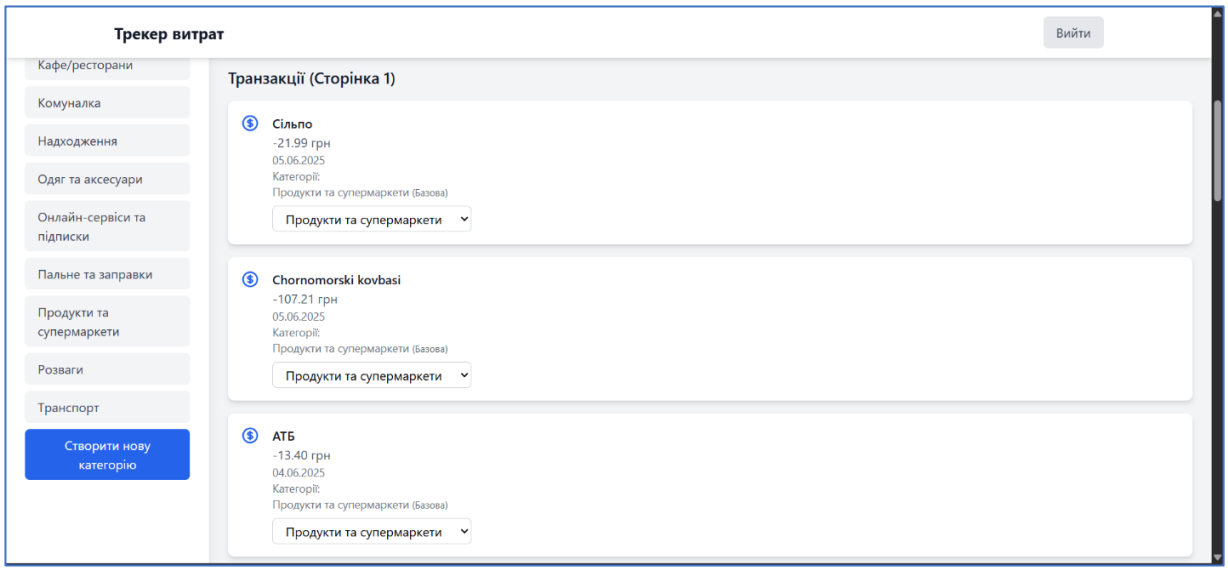


Рисунок 14 – *Інтерфейс нижньої частини сторінки категорії.*

Натиснувши на кнопку «Створити нову категорію» користувач відкриває модальне вікно з формою створення користувацької категорії з можливістю задати її назву та обрати вбудовані категорії, транзакції з яких мають бути включені до створюваної категорії (див. Рис. 15).

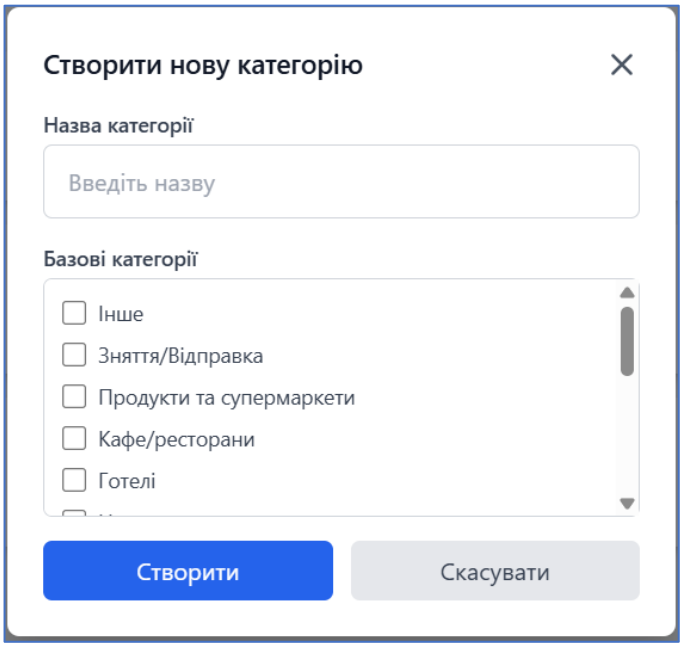


Рисунок 15 – *Модальне вікно з формою для створення категорії.*

Натиснувши на блакитну іконку-кнопку біля користувацької категорії користувач відкриває модальне вікно з формою редагування користувацької категорії (див Рис. 16). У модальному вікні буде виведена поточна назва категорії та поточні вбудовані категорії, транзакції з яких враховуються до користувацької категорії. Користувач може змінити назву категорії та вбудовані категорії, транзації з яких імпортуються до користувацької.

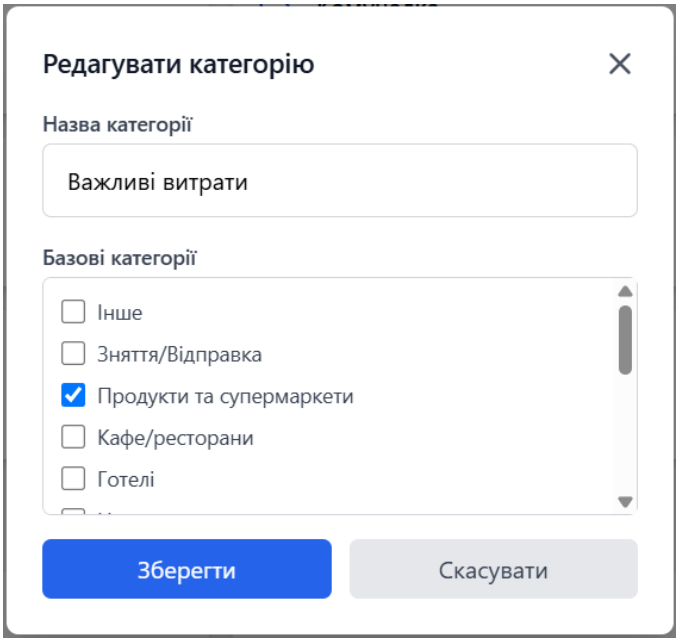


Рисунок 16 – *Модальне вікно з формою для редагування категорії.*

Натиснувши на червону іконку-кнопку біля користувацької категорії користувач відкриває модальне вікно з формою видалення користувацької категорії (див. Рис. 17). У модальному вікні буде виведена назва категорії.

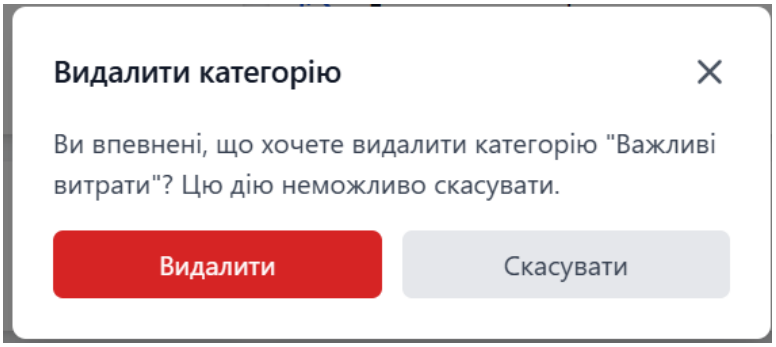


Рисунок 17 – *Модальне вікно з формою для видалення категорії.*

Натиснувши на кнопку «Змінити логін/пароль» користувач відкриває модальне вікно з формою редагування профілю користувача з можливістю задати новий логін та/або пароль (див. Рис. 18). Для затвердження змін до профілю користувач зобов’язаний ввести поточний пароль.

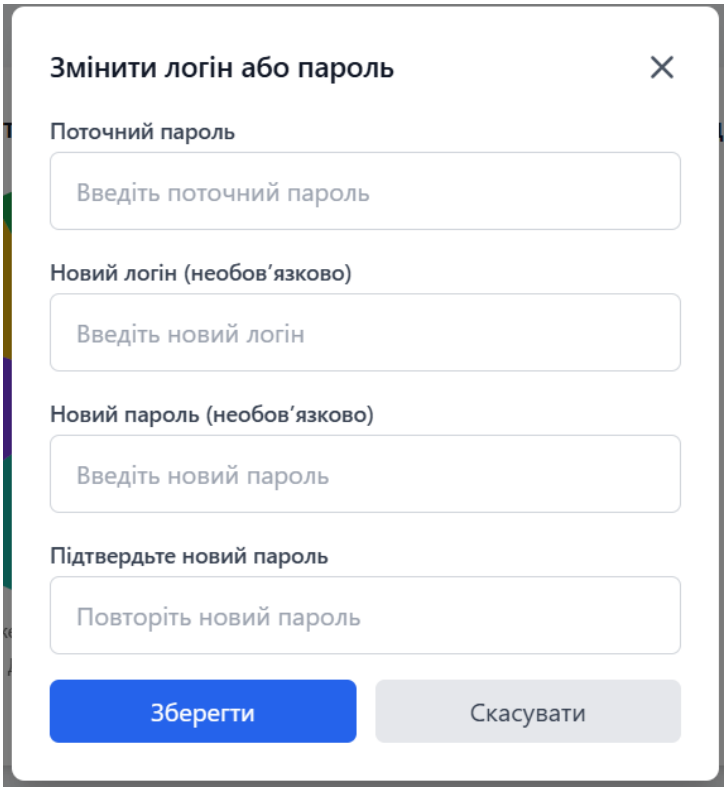


Рисунок 18 – *Модальне вікно з формою для редагування профілю.*

3.7 Висновки до розділу 3

У межах розробки веб-застосунку для відстеження витрат було реалізовано повний життєвий цикл створення програмного продукту.

Інтеграція з API Monobank забезпечила автоматичне отримання транзакцій у режимі реального часу, що дозволило оперативно відображати актуальну фінансову інформацію.

Архітектура системи була побудована з урахуванням ключових вимог — гнучкості, надійності та безпеки. Завдяки використанню сучасного стеку технологій вдалося забезпечити злагоджену взаємодію між усіма компонентами та простоту масштабування проєкту в майбутньому.

Особливу увагу приділено зручності користувацького інтерфейсу. Інтуїтивна навігація, адаптивний дизайн і графічне представлення даних дозволяють легко взаємодіяти із системою як на комп’ютерах, так і на мобільних пристроях.

Фінальне тестування охоплювало всі критичні аспекти — функціональність, безпеку та продуктивність. Результати підтвердили стабільну роботу системи, її відповідність вимогам та готовність до повноцінного використання.

У підсумку вдалося створити сучасний та практичний веб-застосунок, який поєднує автоматичну банківську інтеграцію з аналітичними інструментами в єдиній платформі.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було спроєктовано та реалізовано веб застосунок трекеру витрат з банківською інтеграцією та статистикою, який допомагає користувачам ефективно контролювати власні фінанси. Метою проєкту було створення зручного, безпечного та адаптивного інструменту для щоденного обліку витрат, з можливістю гнучкої категоризації та інтерактивної візуалізації даних.

Розробка почалася з аналізу предметної області та вивчення ринку — було виявлено, що існуючі сервіси, зокрема власний застосунок від Monobank, YNAB, PocketGuard, мають обмеження у кастомізації та відображенні актуальних даних. Це визначило основні функції майбутнього додатку: синхронізація транзакцій у реальному часі завдяки інтеграції Monobank API, можливість ручного коригування категорій для транзакцій та створення власних категорій, а також наочна подача фінансової статистики.

Технічну реалізацію побудовано на ASP.NET Core з Razor Pages, базу даних реалізовано на PostgreSQL з використанням Entity Framework Core. Tailwind CSS та Chart.js були використані для розробки приємного користувацького інтерфейсу та інтерактивних графіків.

Система спроєктована за трирівневою архітектурою з акцентом на масштабованість, зручність і безпеку. Реалізовано автентифікацію з JWT, автоматичний імпорт транзакцій, кастомізовані категорії та захист чутливої інформації.

Під час розробки було подолано низку технічних викликів, зокрема обмеження швидкості API, забезпечення узгодженості даних та оптимізацію інтерфейсу. Завдяки тестуванню і поступовій розробці вдалося досягти стабільної та чуйної роботи додатку.

Веб застосунок трекеру витрат з банківською інтеграцією та статистикою став надійним рішенням для персонального фінансового планування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Personal Finance Software Market Size & Share Report 2030: https://www.grandviewresearch.com/industry-analysis/personal-finance-software-market-report (дата звернення 08.06.2025 р).

2. Personal Finance Software Market Report 2025 - Opportunities and Outlook By 2034: https://www.thebusinessresearchcompany.com/report/personal-finance-software-global-market-report (дата звернення 08.06.2025 р).

3. Overview of ASP.NET Core: https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-9.0 (дата звернення 08.06.2025 р).

4. Exploring Your First Razor Page Application: https://codesignal.com/learn/courses/introduction-to-razor-pages/lessons/exploring-your-first-razor-page-application (дата звернення 08.06.2025 р).

5. A Beginner’s Guide to Entity Framework Core (EF Core) https://medium.com/@ravipatel.it/a-beginners-guide-to-entity-framework-core-ef-core-5cde48fc7f7a (дата звернення 08.06.2025 р).

6. How Are APIs Used In The FinTech And Banking Sectors? https://ideausher.com/blog/use-of-apis-in-fintech-guide/ (дата звернення 08.06.2025 р).

7. Pro ASP.NET Core 7, Tenth Edition 10th Edition, by Adam Freeman, Manning, 2023.

8. ASP.NET Core in Action, Third Edition 3rd Edition, by Andrew Lock, Manning, 2023.

9. Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series), by Robert Martin, Pearson, 2017.

10. C# 10 and .NET 6 – Modern Cross-Platform Development: Build apps, websites, and services with ASP.NET Core 6, Blazor, and EF Core 6 using Visual Studio 2022 and Visual Studio Code, Sixth Edition 6th Edition, by Mark J. Price, Packt Publishing, 2021.

11. ASP.NET Core Razor Pages in Action, by Mike Brind, Manning, 2022.

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Мірзоян Владислав Арсенович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти se121-07@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Веб застосунок трекеру витрат з банківською інтеграцією та статистикою»** відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України
«Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет- системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 17.06.2025 Підпис Мірзоян Владислав Арсенович
(студент)

Дата 18.06.2025 Підпис Жуков Олександр Олексійович
(науковий керівник)