Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук та інформаційних технологій
(повне найменування інституту, назва факультету(відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної циклової комісії))

Пояснювальна записка

до дипломної роботи

перший (бакалаврський)
(рівень вищої освіти)

на тему: Розроблення вебзастосунку для управління особистими фінансами

Виконав студент 4 курсу, групи КН-41
спеціальності: 122 "Комп'ютерні науки"
(шифр і назва напрямку підготовки спеціальності)

Пастернак Андрій Тарасович
(прізвище, ініціали)

Керівники Яркун В.І., Карашецький В.П.
(прізвище, ініціали)

Рецензент Часковський О.Г.
(прізвище, ініціали)

Львів-2025

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти перший (бакалаврський)
Спеціальність 122 "Комп'ютерні науки"

ЗАТВЕРДЖУЮ:
Завідувач кафедри КН
Борецька І.Б.
"10" червня 2025 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Пастернак Андрій Тарасович
(прізвище, ім'я, по батькові)

1.Тема бакалаврської роботи: Розроблення вебзастосунку для управління особистими фінансами
керівник роботи Яркун В.І. ст. викл., Карашецький В.П. к.т.н., доцент кафедри інженерії програмного забезпечення
затверджені наказом вищого навчального закладу від "15" 11 2024 р. № С-882
2. Термін подання студентом роботи 10 червня 2025р.
3. Вихідні дані до роботи Розробити вебзастосунок для ведення обліку витрат користувача. Даний вебзастосунок призначений для спільної роботи в реальному часі
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
Вступ
Стан проблемної області
Інформаційне забезпечення
Програмне та технічне забезпечення
Висновки
Список використаних джерел
Додатки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Підготовка матеріалу до доповіді.
6. Дата видачі завдання 18 листопада 2024р.

КАЛЕНДАРНИЙ ПЛАН

№, з/п	Етапи бакалаврської роботи	Термін виконання етапів роботи	Примітка
1.	Огляд літератури згідно досліджуваної теми. Збір необхідних матеріалів.	18.11.2025 р. 28.02.2025 р.	Виконано
2.	Огляд сучасного стану проблемної області. Виконання вхідного етапу технології	01.03.2025 р. 12.03.2025 р.	Виконано
3.	Реалізація головних функцій проєкту	16.03.2025 р. 29.03.2025 р.	Виконано
4.	Виконання етапу відлагодження проєкту	02.04.2025 р. 12.04.2025 р.	Виконано
5.	Оформлення висновків пояснювальної записки.	13.05.2025 р. 19.05.2025 р.	Виконано
6.	Оформлення записки до дипломного проєкту.	23.05.2025 р. 10.06.2025 р.	Виконано

Студент _____ (підпис) Пастернак А Т. (прізвище та ініціали)

Керівники роботи _____ (підпис) Яркун В.І. (прізвище та ініціали)

_____ (підпис) Карашецький В.П. (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 59 сторінок пояснювальної записки, 23 рисунка, 1 таблиця, 2 додатки.

У цій роботі представлено веб-застосунок Spendify для обліку особистих витрат, реалізований на базі ASP.NET Core, Angular та Elasticsearch. Проведено огляд існуючих рішень із персонального фінансового менеджменту та обґрунтовано вибір технологій, що забезпечують швидкий пошук за назвою, категорією, предметом і локацією. Застосунок працює безпосередньо в браузері й не вимагає складного налаштування чи додаткового ПЗ. Spendify дозволяє додавати, редагувати та видаляти записи про витрати, переглядати аналітичні діаграми й може використовуватися для особистого контролю витрат, малого бізнесу або навчальних проєктів.

Ключові слова: облік витрат, веб-застосунок, ASP.NET Core, Angular, Elasticsearch, діаграми витрат.

ABSTRACT

The thesis contains 59 pages of explanatory note, 23 figures, 1 table, 2 appendix.

This work presents Spendify, a web application for personal expense tracking built with ASP.NET Core, Angular, and Elasticsearch. The application runs directly in the browser without requiring complex setup or additional software. We review existing budget management solutions and justify the chosen tech stack for fast search by name, category, item or location. Spendify supports adding, editing, and deleting expense entries, provides analytical charts, and is suitable for individual users, small businesses, or educational projects.

Keywords: expense tracking, web application, ASP.NET Core, Angular, Elasticsearch.

ТЕХНІЧНЕ ЗАВДАННЯ

Розробити веб-застосунок для ведення обліку особистих витрат.

Реалізувати програмне забезпечення, де можна записувати редагувати і видаляти свої витрати, бачити статистичні графіки. Забезпечити можливість зручно створювати витрати з різним наповненням і вибором локації.

Система повинна мати зручний веб-інтерфейс і бути реалізованою з використанням ASP.NET Core, Angular. Можливість запуску всієї системи в браузері. У якості бекенда використати ASP.NET Core для реалізації REST-API та безпечної роботи з реляційною базою даних, фронтенд побудувати на Angular із компонентною архітектурою, а пошуковий індекс організувати за допомогою Elasticsearch.

Оформити пояснювальну записку відповідно до стандартів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface, інтерфейс програмування застосунків;

ASP.NET – Active Server Pages .NET, фреймворк для розробки веб-застосунків;

BE – Backend, серверна частина застосунку;

CI/CD – Continuous Integration / Continuous Deployment, безперервна інтеграція та доставлення;

CRUD – Create, Read, Update, Delete, базові операції з даними;

CSV – Comma-Separated Values, текстовий формат табличних даних;

DB – Database, база даних;

ES – Elasticsearch, пошуковий рушій на основі Lucene;

FE – Frontend, клієнтська частина застосунку;

HTTP/HTTPS – HyperText Transfer Protocol / HyperText Transfer Protocol Secure, протокол (захищений протокол) передачі гіпертексту;

ID – identifier, унікальний ідентифікатор.

JSON – JavaScript Object Notation, формат обміну даними;

JWT – JSON Web Token, стандарт безпечної передачі даних;

RDBMS – Relational Database Management System, система керування реляційною БД;

REST – Representational State Transfer, архітектурний стиль веб-сервісів;

UI – User Interface, користувацький інтерфейс;

UX – User Experience, досвід користувача;

ПЗ –програмне забезпечення.

ВСТУП

У сучасному інформаційному суспільстві питання ефективного контролю особистих фінансів набувають надзвичайної актуальності. Незважаючи на велику кількість доступних мобільних та десктопних рішень, багато з них відрізняються неналежною зручністю або надмірною складністю налаштування. Більшість існуючих сервісів спрямовані переважно на англомовний ринок та часто вимагають встановлення додаткового програмного забезпечення або володіння спеціалізованими знаннями для їхньої експлуатації. Таким чином, потреба в простому, інтуїтивно зрозумілому та доступному інструменті для обліку витрат залишається відкритою.

Актуальність дослідження. У сучасному інформаційному суспільстві все більша кількість користувачів стикається з необхідністю контролювати власні фінанси швидко та зручно. Існуючі рішення часто або надто складні у налаштуванні, або не пристосовані до українського ринку, що створює прогалину в доступності простого інструменту для обліку витрат.

Об’єкт дослідження. Інтерактивні веб-застосунки для персонального фінансового менеджменту, що працюють у браузері без додаткового встановлення ПЗ.

Предмет дослідження. Архітектура та реалізація компонентів клієнт-серверної системи Spendify на базі ASP.NET Core, Angular та Elasticsearch для зберігання, обробки та пошуку даних про витрати.

Мета роботи. Розробити та обґрунтувати дизайн веб-застосунку, який забезпечує інтуїтивний інтерфейс, швидкий пошук за назвою категорії, предмета та локації, а також візуалізацію аналітичних даних у вигляді діаграм динаміки витрат.

Новизна роботи. Впровадження Elasticsearch із діапазонним пошуком ±100 одиниць без втрати продуктивності, а також поєднання ASP.NET Core із компонентним Angular-фронтендом для досягнення принципу «zero-install» і адаптивності інтерфейсу будь-якого розміру екрана.

Практична значимість. Запропоноване рішення може бути застосоване як для особистого контролю витрат, так і в малій підприємницькій діяльності або освітніх проєктах, підвищуючи фінансову грамотність та спрощуючи аналіз особистих чи групових бюджетів.

Система може бути основою для подальшого розвитку у вигляді мобільного застосунку або сервісу з інтеграцією в існуючі цифрові бібліотеки.

РОЗДІЛ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1 Актуальність ведення обліку витрат

В сучасних реаліях, постійне зростання витрат через інфляцію, підвищення цін на нерухомість і комунальні послуги робить контроль особистих фінансів критично необхідним.

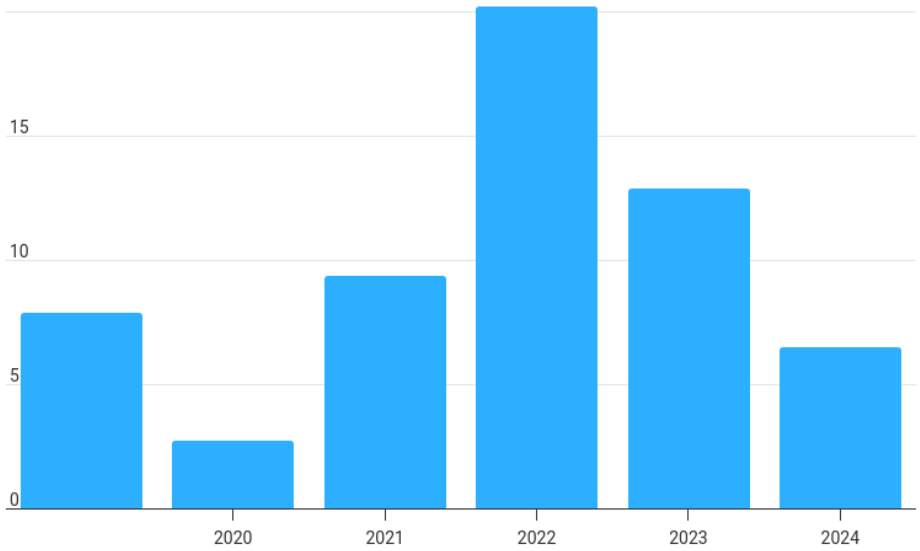


Рисунок 1.1 - Рівень інфляції в Україні (2019 – 2024)

Тому важливо застосовувати системний облік усіх фінансових операцій з таких причин:

1 **Економічно-соціальний вплив –** без системного обліку щоденних витрат, навіть незначні суми, як наприклад похід в кав'ярню, онлайн-підписки, транспорт – з часом можуть перетворитися в суттєві суми. Невміння управляти своїм бюджетом може поставити під загрозу фінансову стабільність особи або родини. "Головною метою цього проекту було надання користувачам програмного рішення для управління фінансами та відстеження особистих витрат та прибутків" [1].

2 **Самовпевненість в управлінні фінансами –** велика частка дорослих не використовує системи обліку особистих фінансів, вважаючи, що розуміють куди щомісяця «пропадає» значна частина доходів. Наслідки цього можуть бути різні, від заборгованостей, до кредитів і постійного стресу через

неможливість передбачити витрати. “Зроблено порівняльний опис і обрано функції, які найкраще підходять для зменшення особистих витрат, планування бюджету та розподілення коштів для особистих потреб для найбільш ефективного використання” [2].

3 **Позитивні наслідки управління коштами –** записування витрат на постійній основі виробляє усвідомлене ставлення до фінансів, від чого зменшується шанс імпульсивних покупок та мотивує витрачати гроші на пріоритетні цілі. За допомогою візуалізації витрат простіше зрозуміти прогалини, як наприклад походи в кафе чи непотрібні підписки.

4 **Економічні вигоди та перехід до цифрових рішень –** категоризація витрат допоможе виявити куди направлено найбільше поглинання ресурсів. Графіки та діаграми, наочно покажуть, куди саме користувач витрачає кошти. Звітність, яка дозволяє зручно переглядати і шукати потрібну транзакцію, що не скажеш про паперові журнали, які щороку поступаються місцем веб- та мобільним застосункам.

1.2 Огляд сучасних технологій для розробки

Клієнтська частина (Frontend)

- Angular – використовує модульну архітектуру, що чітко розподіляє відповідальність. Строга типізація TypeScript підтримує безпеку і зменшує кількість помилок при розробці
- PrimeNG – дуже великий вибір готових UI-компонентів (таблиці, діалоги, календарі, поля введення для форм) прискорює розробку інтерфейсу і забезпечує зручну і якісну стилізацію.
- ECharts – бібліотека для візуалізації даних, яка показує зручні графіки базуючись на великих об’ємах даних.

Серверна частина (Backend)

- ASP.NET Core (API) – швидкий, продуктивний крос-платформенний фреймворк для створення RESTful API. Користуючись ним

легко створювати потрібну логіку і маршрутизацію, обробку помилок і конфігурації.

- ASP.NET Identity + JWT – аутентифікація та авторизація використовуючи перевірені компоненти Microsoft гарантують безпеку в зберіганні облікових даних користувачів та видачу JWT токенів. За допомогою цього зручно інтегрувати зовнішні клієнти, наприклад мобільні застосунки
- Entity Framework Core – ORM інструмент для роботи з реляційною базою даних через моделі. Також підтримує міграції, що забезпечує зручну зміну схеми БД

Сховище даних та фонові сервіси

- MS SQL Server – надійна реляційна СУБД з сильними інструментами резервного копіювання, моніторингу та налаштування продуктивності. Є можливість користуватись транзакціями з ізоляцією для успішної роботи з фінансовими записами
- ElasticSearch – використовується для індексації та повнотекстового пошуку за витратами користувача. Шукати можна за локацією, категорією та предметами, які вказані у витраті. Відповідність знаходиться досить швидко завдяки механізму Inverted-index, що миттєво забезпечує відгук пошуку.
- SignalR – швидкий спосіб реалізувати real-time сповіщення у веб-застосунку, а також чат-підтримку без перевантаження сторінки.
- HangFire – фреймворк для створення фонових задач на .NET, можна використовувати для періодичної перевірки, відправки звіту на пошту, тощо.

1.3 Аналіз популярних бібліотек і фремворків

Останні роки розробка веб-застосунків розвивається у швидкому темпі. Тому вибір потребує співставлень різних фреймворків і бібліотек.

Серед фреймворків для клієнтської частини можна виділити Angular – всіма компаніями улюблений фреймворк від Google, який має в собі повне архітектурне рішення з чіткою структурою, на відміну від бібліотек таких як

React та Vue. Важливу роль відіграє посилена типізація надбудови TypeScript, а також модульність, форми, маршрутизації роблять Angular зручним вибором для розробки веб-додатків середніх і великих розмірів. У свою чергу, UI-бібліотеки, як PrimeNG та Angular Material, можуть значно скоротити час створення інтерфейсу, так як мають широку колекцію готових компонент. Крім цього їх можна додатково стилістично налаштовувати, що є вагомим фактором при розробці веб-застосунку.

Для побудови графіки та візуалізації даних на розгляд було декілька бібліотек, наприклад Chart.js, D3.js та ECharts. Остання була вибрана, через підтримку адаптивного рендерингу великих обсягів даних, продуктивність і гнучкість, звісно бібліотека може похвалитися широким набором графіків на любий смак.

На серверній частині найпопулярнішими є дві технології: ASP.NET Core та Node.js (з NestJS або Express). У цьому проекті було обрано ASP.NET Core, як більш зручне, стабільне та строго типізоване середовище, яке використовує C#. Варто зазначити ASP.NET Identity, який дає гнучку систему аутентифікації з можливістю інтегрування різних способів, наприклад JWT – токенів для захисту API-запитів, що є стандартом. Для доступу до реляційної бази даних на вибір є дві найпопулярніші ORM системи Entity Framework Core та Dapper. В цьому проекті було вибрано Entity Framework Core ORM систему, так як значно спрощує роботу з транзакціями, міграціями та запитами через LINQ.

На вибір реляційної бази даних було декілька варіантів: MySQL, MS SQL Server та PostgreSQL. Вибір попав на MS SQL Server, що забезпечує високу продуктивність, підтримку складних запитів за засоби для аналітики. Для реалізації пошуку було інтегровано Elasticsearch, що демонструє кращу продуктивність при повнотекстових запитах у порівнянні зі стандартними SQL-механізмами. Крім того, для реалізації механізмів реального часу було використано SignalR — бібліотеку від Microsoft для WebSocket-з’єднань, а для планування фонових завдань — HangFire, що дозволяє запускати задачі за розкладом без окремого сервісу чи крону.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Вимоги до вхідних даних

Для коректного функціонування системи Spendify необхідно забезпечити цілісність і достовірність кожного елементу вхідних даних, що безпосередньо впливає на точність подальших аналітичних розрахунків та відображення статистичних результатів. Передусім, всі вхідні дані мають бути належним чином структурувані, зберігати зв’язок із відповідними сутностями системи та ураховувати обмеження цілей цілісності (Integrity Constraints). Будь-яке відхилення або невідповідність у форматуванні вхідних полів призводить до втрати семантики даних і робить неможливою реалізацію пошукових запитів за допомогою Elasticsearch, а також достовірний розрахунок агрегованих показників витрат.

По-перше, всі записи про витрати мають містити унікальний ідентифікатор користувача, до якого належить ця транзакція. Відсутність або некоректне значення цього поля негативно відображається на приватності даних і гарантії того, що кожен користувач бачить лише власні витрати. Використання одного загального індексу для декількох користувачів без чітко визначеного зв’язку між Expense і User призведе до порушення базових вимог безпеки та приватності. Вважаю, що зневага цим аспектом є фундаментальною помилкою, оскільки гарантування конфіденційності – обов’язкова складова будь-якого сервісу, що працює з персональними фінансовими даними.

Далі, кожен витрачений ресурс має супроводжуватися чітко визначеними атрибутами: датою транзакції, категорією, кількома пунктами витрат (ItemExpense), місцем придбання та сумою транзакції. Дата транзакції повинна мати формат ISO 8601 (рік-місяць-день) і потрапляти до індексу в єдиному часовому поясі – UTC, для уникнення будь-яких похибок при агрегації даних за часовими діапазонами. Я переконаний, що використання іншого формату без

попередньої конвертації призведе до некоректної агрегації – зокрема, місячних і річних звітів, що є ключовим елементом статистичного аналізу витрат.

Категорія витрати (Category Name) виступає важливим класифікатором, що дозволяє здійснювати групування та фільтрацію даних. Вхідні дані щодо категорії мають бути зведені до словника контрольованих термінів або таблиці у базі даних, яка містить перелік дозволених категорій та їх ієрархічні зв'язки (якщо такі передбачені). Вважаю, що відсутність контролю за цим полем призводить до утворення неуніфікованих категорій із синонімами та помилковими записами, що унеможливлює проведення якісного аналізу. Крім того, категорія повинна мати ідентифікатор, який передається у всі сервіси системи, щоб забезпечити коректне формування запитів до Elasticsearch та уникнути дублювання даних.

Кількість пунктів витрат (ItemExpense) є суттєвим елементом, оскільки кожен пункт (Item) може мати власний тип, ціну за одиницю та кількість. Вхідні дані щодо ItemExpense повинні включати унікальний ідентифікатор елемента, коректно зв'язаний із довідником Item (що містить назву, опис та інші атрибути), кількість (ціле число або десятковий формат з двома десятковими знаками) та ціну за одиницю витрати у гривнях. Неприпустимі будь-які від'ємні або нульові значення для цих полів; вважаю, що дозволяти такі дані – це грубе порушення самої суті обліку витрат. Кожна позиція має бути виражена в розрахунках із точністю до копійки, щоб виключити накопичення похибок при підсумуванні великих обсягів операцій.

Місце придбання (Location) має бути представлене у вигляді карти, що дозволить користувачу вказати де саме була зроблена покупка, або може вибрати зі списку збережених ним локацій. На мою думку, користувач має право вибирати де саме була зроблена покупка і це його відповідальність правильно вказувати локацію, щоб статистика коректно формувалась для нього.

Нарешті, сума витрати (Expense Amount) повинна формуватися на кількості і вартості пунктів витрат (Item). А ціна пункта витрати повинна мати числовий формат із фіксованою десятичною точкою (наприклад, decimal(10,2)),

що дозволяє зберегти необхідну точність у гривнях. Моя принципова позиція полягає в тому, що ігнорування контролю за розміром суми негативно впливає як на безпеку даних, так і на адекватність статистичного аналізу.

Узагальнюючи, вимоги до вхідних даних у проекті Spendify мають ґрунтуватися на чітко визначених стандартах форматування, суворих обмеженнях цілісності даних та контролі за довідниковими значеннями. Сприймаю будь-яку поблажливість до неуніфікованих або некоректно введених даних як неприйнятну, оскільки це підриває базову функціональність системи та унеможливлює реалізацію пошукових механізмів і достовірних статистичних звітів.

2.2 Структура вихідних та вхідних даних системи

Структура вхідних та вихідних даних у системі Spendify передбачає однорідну передачу інформації між клієнтським інтерфейсом і серверною частиною, де кожен елемент моделі представлений набором властивостей із визначеними обмеженнями. Вхідні дані, які надходять від користувача під час реєстрації, створення категорій, предметів або локацій, а також під час додавання витрат, повинні відповідати однозначно визначеним полям у відповідних таблицях бази даних. При цьому необхідно пам’ятати, що під час реєстрації значення полів Currency та Language не передаються користувачем, оскільки їм автоматично присвоюються дефолтні значення: Currency=0 (USD) та Language=0 (US). Ігнорування цього механізму вважав би серйозною помилкою дизайну, оскільки визначені за замовчуванням валюта та мова є критичними для подальшої коректної інтерпретації користувацького інтерфейсу і алгоритмів розрахунку вартості витрат.

Під час створення або редагування запису категорії (таблиця Categories) клієнтська частина передає серверу властивості Name, UserId, IsPublic, Color, Description та Icon. Значення поля Name обов’язково повинно бути унікальним у межах користувача, щоб уникнути дублювання найменувань і спрощувати

процес пошуку. Якщо категорію створює звичайний користувач, то поле IsPublic автоматично встановлюється у false, що гарантує її приватність та виключає можливість інших користувачів випадково або навмисно обрати цю категорію. Лише адміністратор може створити категорію із властивістю IsPublic=true, щоб зробити її доступною всім користувачам. Поле IsDeleted має логічний тип і його призначено для приховування категорій без їх фізичного видалення з бази даних; позначення IsDeleted=true означає, що категорія більше не відображається у виборі, але залишається для історії транзакцій та збереження цілісності даних.

Структура предметів витрати (таблиця Items) передбачає набір полів, що описують їх властивості: Name, Price, Description, Rating, UnitOfMeasure, Vendor, а також IsPublic, IsDeleted і UserId. Кожен запис Item пов’язаний із користувачем через UserId, окрім тих випадків, коли IsPublic=true — це дозволяє адміністратору поповнювати перелік доступних для всіх предметів, а звичайні користувачі можуть вибирати їх із єдиного «публічного» набору предметів. Поле Price має десятковий формат із двома знаками після коми і не може набувати нульового або від’ємного значення. Неприпустимість таких значень є визначальною вимогою, адже інакше виникають абсурдні чи шахрайські сценарії, коли «вартість» могла би бути негативною або просто незафіксованою. Строгість цього контролю є беззаперечною умовою достовірності облікових даних.

Зі свого боку, таблиця Locations моделює простір, у якому було здійснено покупку. Поля Name, Latitude, Longitude та Address використовуються для відображення і запам’ятовування локації. Користувач може вказати локацію за допомогою натискання на мапі, вручну ввести адресу або обрати одну зі збережених у власному довіднику. У разі створення нової локації властивість Save позначає, чи слід додати запис до довідника (Save=true) у заходах логіки роботи інтерфейсу, а безпосередньо у базі даних існує поле Save, що позначає, чи зберігається локація у переліку «улюблених» або «збережених» локацій користувача. Це може здатися зайвим дублюванням у дизайні, але, на мою

думку, чіткий поділ запиту «бути доступним для майбутніх виборів» і властивості «існувати одноразово лише для цієї транзакції» виправданий у складних сценаріях, коли користувачі працюють із великою кількістю точок на карті. Поле UserId у таблиці Locations гарантує, що локації створюються у приватному оточенні для користувача. Адміністратор не може додати публічно локацію.

Таблиця Expenses містить записи про окремі транзакції та має набір полів: Id, Date, UserId, CategoryId та LocationId. Поле Date у форматі ISO 8601 (YYYY-MM-DD) із збереженням у UTC відіграє ключову роль в обчисленні агрегованих показників витрат за часовими діапазонами. Недотримання цього стандарту чи введення дати в іншому форматі – неприйнятне, адже це призводить до спотворених інтервалів звітності, а відтак невірних висновків. UserId зв’язує кожну транзакцію з конкретним користувачем; ігнорувати цей зв’язок позначає нехтування безпекою й приватністю, що є непробачною помилкою на рівні архітектурного рішення. CategoryId і LocationId сцеплені зі своїми відповідними таблицями, отже будь-яке некоректне або неіснуюче значення у цих полях потребує валідації перед збереженням. При наявності єдиного індексу Elasticsearch для всіх витрат правильна передача цих ідентифікаторів є єдиним способом забезпечити коректний фільтр у запитах і виключити перетік інформації між обліками різних користувачів.

Ще одним критичним елементом є зв’язок сутностей «витрата–пункт витрати», що представлений таблицею ItemExpenses. Поля ItemId, ExpenseId і Quantity визначають, які саме предмети та в яких кількостях були включені до конкретної транзакції. ItemId відповідає запису з таблиці Items, у якому міститься інформація про ціну за одиницю, опис і таке інше. Quantity із форматом integer. Будь-яке порушення, наприклад передача нульового або від’ємного значення Quantity, є абсолютно неприпустимим і вважаю його показником серйозного дефекту бізнес-логіки, оскільки веде до хибних підрахунків.

Схемою вивідних даних можна вважати JSON-об’єкти, які сервіс повертає клієнту після обробки запиту. Наприклад, запит на отримання всіх категорій повертає масив об’єктів, у кожному з яких присутнє Id, Name, IsDeleted, UserId, IsPublic, Color, Description і Icon. Однак у відповіді мають бути виключені записи з IsDeleted=true, і, якщо йдеться про стандартного користувача, із загального переліку відфільтровуються ті категорії, у яких IsPublic=false і UserId не збігається з його ідентифікатором. Незнання або недотримання цього правила означало б неприпустиме порушення концепції захисту даних. Аналогічно, коли клієнт запитує список предметів (Items), повертається набір полів Id, Name, Price, IsPublic, IsDeleted, UserId, Description, Rating, UnitOfMeasure та Vendor, із виключенням тих записів, які марковані IsDeleted=true або належать іншому користувачу в разі IsPublic=false. У разі звернення до локацій (Locations) обов’язково включаються Id, Name, Latitude, Longitude, Address, Save і UserId, але навпаки.

Для сторінки витрат (Expenses) вихідні дані охоплюють Id, Date, UserId, CategoryId, LocationId, а також список пов’язаних ItemExpenses із полями ItemId і Quantity. Я вважаю, що забувати про включення інформації про кожен пункт витрати було б грубою помилкою, адже без цього стає неможливим детальне відображення збалансованого звіту та подальше виконання агрегованих розрахунків (наприклад, сума за категоріями чи середня вартість пункту). Зі сторони сервера дані передаються у вигляді вкладених об’єктів або мають відповідний механізм «join» у запиті до бази, щоб спростити клієнтський код і зменшити кількість запитів.

Підсумовуючи, вхідні дані системи представлені набором полів, що відображають стійкі зв’язки між сутностями Users, Categories, Items, Locations, Expenses та ItemExpenses, із суворою валідацією форматів і обов’язковим контролем полів IsPublic та IsDeleted для забезпечення приватності й коректного фільтрування. Вихідні дані формуються у вигляді структурованих JSON-об’єктів із сформованими відносинами між сутностями, що гарантує

повноцінне відображення інформації в інтерфейсі користувача та точність статистичних модулів.

2.3 Алгоритмічна структура та основи роботи Elasticsearch

У контексті системи Spendify Elasticsearch виконує роль ядра пошукового та аналітичного механізму, що забезпечує швидке й релевантне знаходження інформації про витрати користувачів. Насамперед слід розуміти, що Elasticsearch побудований на основі інвертованого індексу, який радикально відрізняється від традиційних реляційних підходів до пошуку. З математичної точки зору інвертований індекс у Elasticsearch формує своєрідну термін–документну матрицю, де кожен термін (слово або фраза) зіставлений із переліком документів, у яких він присутній. Такий підхід дозволяє значно зменшити час пошуку: замість повного сканування всіх документів система звертається безпосередньо до індексу за конкретним терміном, отримуючи список відповідних ідентифікаторів документів.

Для оцінювання релевантності знайдених документів Elasticsearch використовує механізми TF–IDF (Term Frequency–Inverse Document Frequency) у поєднанні з більш сучасною моделлю BM25. Простими словами, TF–IDF розраховує вагу терміну в документі як добуток двох компонентів: локальна частота терміну (TF), що відображає, скільки разів термін зустрічається у конкретному документі, та зворотна частота документа (IDF), що знижує вагу термінів, які занадто поширені в усій колекції документів. Прикладом того, чому TF–IDF варто застосовувати в Spendify, є пошук за назвою товару або категорії: якщо термін «кава» зустрічається в усіх документах (наприклад, усі витрати пов’язані з кафе), то його інформаційний вклад у ранжування хай і залишається, проте завдяки IDF його вплив буде зменшений порівняно з більш унікальними ключовими словами. Без цього механізму результати пошуку були б переповнені документами з популярними термінами, що знижує якість

підбору саме тих записів витрат, які найбільш відповідають запиту користувача.

Однак TF–IDF у «чистому» вигляді має недоліки, зокрема недостатньо точне нормування термінів у довгих документах. Саме тому Elasticsearch використовує модель BM25, яка поліпшує TF–IDF шляхом введення додаткових параметрів, відомих як k_1 та b. Параметр k_1 регулює насичення ваги терміну (наприклад, щоб надто часті вживання одного й того ж слова не надавали документу непропорційної переваги), а параметр b відповідає за нормування довжини документа, компенсуючи ситуацію, коли дуже короткі або дуже довгі записи переоцінюються системою. У Spendify кожен документ, що індексується в Elasticsearch, фактично є представленням транзакції користувача (ExpenseDocument), яка містить поля CategoryName, LocationName та ItemNames. Математично BM25 обчислює релевантність документа до пошукового запиту як суму ваг кожного терміну в запиті, скоригованих за частотою і довжиною документа. Це гарантує, що при пошуку за ключовими словами, фразами чи навіть неточними запитами (прив’язаними до властивості Fuzziness.Auto у MultiMatchQuery) найрелевантніші записи виявляються на вершинах результатів.

Важливо підкреслити, що налаштування індексу в Elasticsearch у Spendify здійснюється програмно через Nest-клієнт (як це видно в наведеному коді ElasticSearchService). Кожен документ формується методом MapToDocument, де обчислюється загальна ціна витрати з урахуванням конвертації валют, визначаються назви категорії, локації та набору товарів. Уточню: ігнорування необхідності налаштування полів з урахуванням правильних типів (наприклад, перевірки, що поле ItemNames не містить порожніх рядків) призводить до неточностей у побудові індексу. Себто сама процедура індексації мусить відбуватися з дотриманням вимог математичної коректності: будь-які відсутні чи неправильно конвертовані значення юзера можуть знизити кількість релевантних результатів або виникнуть «мертвяки» в індексі, які ніколи не будуть повернені через неспівпадіння типів.

Безпосередньо у процесі пошуку Elasticsearch будує bool-запит, який комбінує умови термінового фільтра (TermQuery за UserId), діапазону дат (DateRangeQuery для фільтрів типу «тиждень», «місяць» тощо) і текстового пошуку (MultiMatchQuery для полів CategoryName, ItemNames, LocationName). Саме тут алгоритм BM25 максимально проявляє себе: кожен отриманий документ оцінюється за ступенем відповідності введеному користувачем рядку (з урахуванням нечіткості через виставлений Fuzziness), з поправкою на те, наскільки часто термін зустрічається серед усіх документів користувача. На мою думку, звуження кола документів до окремого UserId у поєднанні з математично обґрунтованим BM25 є єдиним способом забезпечити високу швидкодію та відповідність результатів саме для конкретного споживача. Якщо ж допустити індексацію всіх витрат у єдиному контексті без поділу за користувачами, жодним чином не вдасться гарантувати приватність та релевантність.

Оскільки Elasticsearch використовує Lucene під капотом, слід розуміти, що під час індексації кожного терміну додатково враховується аналізатор (analyzer), який розбиває текст на токени, приводить усі символи до нижнього регістру, усуває стоп-слова тощо. З математичної точки зору це означає, що модель TF–IDF і BM25 працюють не з «голими» рядками, а з вже нормалізованими термінними траекторіями. У Spendify будь-яке недбале налаштування mapping для полів, скажімо, текстових полів «CategoryName» чи «ItemNames», призводить до втрати семантики й відповідно недоречних результатів пошуку. На мою думку, правильне визначення типів полів (keyword для точного фільтрування за UserId і Date для коректних DateRange-запитів) є критичним для того, щоб математичні моделі ранжування могли відпрацювати ефективно.

2.4 Організація фонового виконання задач із Hangfire

Hangfire забезпечує надійний механізм для відкладеного та періодичного виконання завдань, що значною мірою звільняє основний потік виконання веб-застосунку від додаткового навантаження й гарантує виконання критичних операцій вчасно. Hangfire ґрунтується на ідеї зберігання інформації про завдання (Jobs) у централізованій базі даних, що у нашому випадку реалізована за допомогою SQL Server. При створенні нового завдання система розраховує час його виконання: для одноразового видалення незавершених реєстрацій вона аналізує дату створення облікового запису й визначає, чи минуло більше ніж сім діб, а для періодичних операцій, зокрема розсилки щотижневих звітів усім активним користувачам у суботу, застосовується розклад у форматі Cron.

З точки зору інформаційного забезпечення, кожне завдання формує набір метаданих, у якому вказуються ключові атрибути: унікальний ідентифікатор користувача або групи користувачів, дата реєстрації чи критерії відбору даних для звіту, часовий пояс Europe/Kyiv та необхідні параметри для генерації вмісту повідомлення. Ці дані конвертуються в єдиний запис у таблиці Hangfire.Worker, яка містить інформацію про тип завдання, параметри виклику методу, а також часову відмітку наступного очікуваного запуску. У результаті в будь-який момент можна отримати ґрунтовну інформацію про стан кожної одиничної або періодичної задачі: чи вона вже виконувалась, чи відбувалася спроба повторного запуску після помилки, а також скільки разів відбулося виконання впродовж останнього тижня.

Математичне забезпечення цієї функціональності полягає у розрахунку інтервалів і умов триггерів. Для одноразового видалення користувачів, які протягом семи днів не завершили процес реєстрації, алгоритм виконує обчислення різниці між поточною датою (отриманою від системного годинника в UTC, а потім скоригованою під Europe/Kyiv) і датою створення облікового запису. Якщо результат перевищує 7×24 годин, Hangfire ініціює видалення цього запису. У випадку щотижневих звітів для розсилки в суботу

використовується Cron-вираз, який визначає точний час запуску (наприклад, «0 9 * * 6» означає виконання рівно о 09:00 щосуботи за місцевим часом). Це означає, що система заздалегідь обчислює наступну дату та годину виконання, ґрунтуючись на календарних алгоритмах із урахуванням календарних змін (перехід на літній/зимовий час).

Крім того, виклик методів, що відповідають за позначення тикетів у саппорті як «розв'язаних», також відбувається у фоновому режимі за фіксованим інтервалом. Кожні кілька годин Hangfire перевіряє журнал переписки: якщо останнє повідомлення в чаті було від адміністратора, і після цього не надійшло жодного нового запиту від користувача протягом встановленого проміжку часу (наприклад, 48 годин), завдання автоматично оновлює статус тикета у базі даних. Якщо ж ніхто з адміністраторів не відповів у межах визначеного терміну, то генерується повідомлення для адміністратора із проханням звернути увагу на невирішені запити. У математичному сенсі це також зводиться до обчислення різниці між часовими мітками останніх повідомлень і поточним часом, щоб визначити, чи необхідно виконувати дію.

Зберігання та стан кожного фонового завдання відстежуються у кількох таблицях Hangfire—Job, Hangfire—State, Hangfire—Counter, що дозволяє оцінити продуктивність основного потоку, виконати аудит історії запусків, а також здійснювати повторні спроби (Retries) у разі помилок. Ці механізми контролюють, щоб у разі збою підключення до бази даних або тимчасової неготовності зовнішніх сервісів (наприклад, служби відправки електронної пошти) Hangfire автоматично перераховував час наступної спроби, враховуючи коефіцієнт зростання інтервалу між спробами (exponential backoff). Такий підхід запобігає заторів у черзі завдань і забезпечує належний рівень відмовостійкості системи.

Узагальнення процесу фонового виконання задач із Hangfire у проекті Spendify демонструє поєднання інформаційної моделі (збереження стану завдань у базі, зберігання метаданих) та математичних алгоритмів (розрахунок часових інтервалів, тригерів та стратегій повторного запуску). Без цієї

складової автоматичного керування важливими повторюваними операціями система не могла б своєчасно видаляти некоректні дані, надсилати регулярні звіти та підтримувати актуальний стан тикетів у саппорті, що у підсумку негативно відобразилося б на якості обслуговування користувачів і загальній продуктивності Spendify.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Архітектура проєкту

3.1.1 Огляд загальної архітектури

Система «Spendify» реалізована за багатошаровою клієнт-серверною архітектурою, що гарантує чітке розмежування відповідальності між компонентами, полегшує підтримку та масштабування рішення.

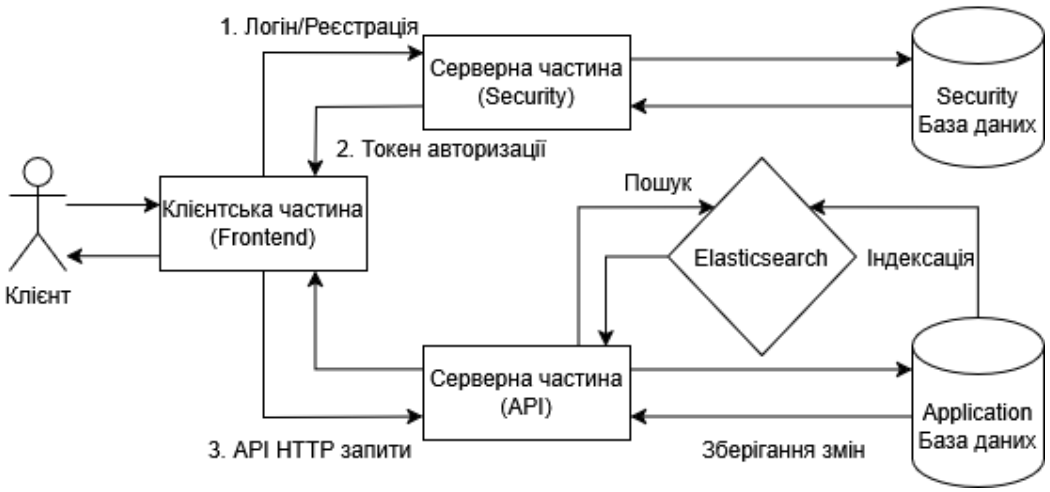


Рисунок 3.1 – Загальна архітектура системи

3.1.2 Презентаційний шар (Frontend)

Клієнтська частина забезпечує взаємодії з користувачем та відображення інтерфейсу веб-застосунка.

У випадку Spendify були вибрані відносно останні технології, а саме: Angular 17+, TypeScript, HTML5, CSS3.

Основні можливості для користувача:

- **Сторінки авторизації і логіка –** користувач може увійти в аккаунт або створити новий аккаунт, пройти перевірку підтвердження електронної пошти і після цього буде безпечно авторизований у веб-застосунку.
- **Інформаційна Панель, Витрати, Каталог –** дають змогу користувачу зручно створити витрату, наповнити предметами з глобального списку, або створивши самому вручну, вибрати категорію і вписати локацію (або вказати на карті), де була зроблена витрата.

- **Налаштування –** в цій сторінці користувач може змінити пароль, відредагувати аккаунт, видалити свій аккаунт, а також якщо є запитання, то може звернутись в підтримку надіславши тікет адміністратору з описом проблеми і в режимі реального часу в чаті проконсультуватись з адміністратором веб-застосунку.
- **Панель керування -** панель адміністрування, в якій швидко можна глянути загальну статистику у веб-застосунку, створити нові категорії, предмети
- **Статистика –** сторінка загальної статистики в якій є можливість переглядати динаміку витрати в часових рамках, а також анонімно дивитись, які у користувачів найпопулярніші предмети і ранжування середньої вартості витрат.
- **Публічний каталог –** окрема сторінка для адміністратора, щоб редагувати публічні предмети і категорії, додавати нові чи приховувати існуючі.

Взаємодія з бекендом відбувається через HTTP-запити до RESTful API на прописані ендпоінти.

3.1.3 Серверна частина (Backend)

Backend Spendify побудований за багатошаровою архітектурою (рис.3.2) із чотирма основними проектами:

1. **WebAPI**

- Відповідає за прийом і маршрутизацію HTTP-запитів (RESTfulAPI).
- Містить контролери, які делегують обробку бізнес-логіці у шар BL.
- Налаштування middleware для аутентифікації/авторизації (JWT), логування, глобальної обробки помилок та CORS.

2. **BL (Business Logic Layer)**

- Взаємодіє із EST.DAL для збереження/отримання даних та із EST.Domain для роботи з доменними сутностями.
- Запускає фонові завдання через Hangfire (наприклад: агрегація щоденних звітів).

- Підтримує SignalR-хаби для оповіщення клієнта в реальному часі (оновлення статистики, push-повідомлення).

3. **DAL (Data Access Layer)**

- Відповідає за взаємодію з реляційною базою даних через Entity Framework Core 8.0:
  - Конфігурація контексту, міграцій, налаштування зв’язків між сутностями.
  - Реалізація патерну «Repository» та «Unit of Work» для атомарних транзакцій.
- Підтримує додаткові сховища (наприклад, кеш Redis або індексування в Elasticsearch 8.x).

4. **Domain**

- Містить доменні моделі (Expense, Category, Location, User тощо) та бізнес-валідатори.
- Описує контракти (інтерфейси сервісів і репозиторіїв), специфікації для пошуку/фільтрації даних.
- Використовується як єдина «мовна платформа» (Ubiquitous Language) усіх шарів.

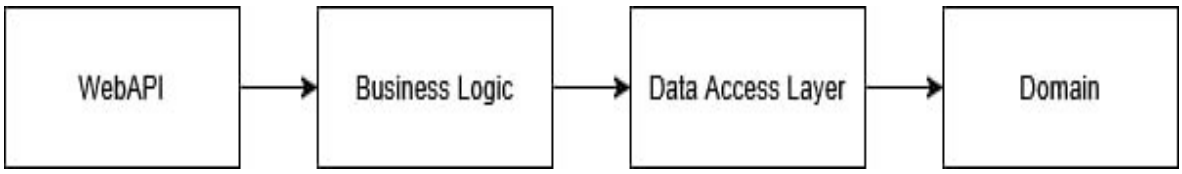


Рисунок 3.2 – Структура залежності проектів

Використані технології:

- платформа: ASP.NET Core 8.0, C#;
- ORM: Entity Framework Core 8.0;
- пошук та аналітика: Elasticsearch 8.x;
- фонова обробка: Hangfire;
- реального часу: SignalR;

- документування API: Swagger / OpenAPI.

REST-endpoints:

- GET — повернення даних (наприклад, останніх 10 витрат);
- POST — створення ресурсів (нового запису витрати);
- PUT/PATCH — оновлення ресурсів (редагування категорії, суми);
- DELETE — видалення ресурсів (видалення витрати або елемента списку).

3.1.4 Рівень зберігання даних (Persistence Layer)

Для забезпечення надійного та ефективного збереження даних, а також швидкого пошуку, у проєкті використовується реляційна база даних Microsoft SQL Server 2019, що містить основні таблиці для користувачів, витрат, категорій, товарів, локацій та зв'язків між ними (рси.3.3). Взаємодія даних реалізована через відношення один-до-багатьох і багато-до-багатьох, зокрема між користувачами та витратами, категоріями та витратами, локаціями та витратами, а також між товарами та витратами через проміжну таблицю. Для пошуку застосовано Elasticsearch версії 8.x, який може працювати як у локальному кластері, так і у Elastic Cloud. Індексація даних здійснюється за допомогою індексів, що формуються за користувачем, з відповідним мапінгом полів, включаючи назви, суми, категорії, товари, локації та дати створення записів. Управління міграціями бази даних проводиться через EF Core Migrations.

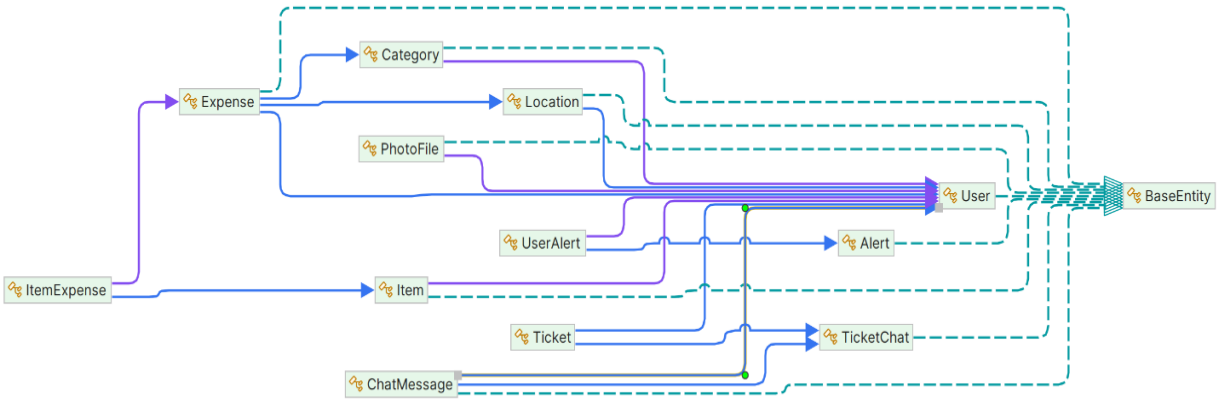


Рисунок 3.3 – Структура залежності класів/таблиць

3.2 Програмна реалізація

У цьому розділі описані інструменти та підходи, використані при розробці клієнт-серверної частини Spendify, а також ключові етапи побудови й налаштування проєкту.

3.2.1 Налаштування середовища розробки

1. **Інтегровані середовища (IDE) та редактори коду**

- **Backend**: Visual Studio 2022 (Enterprise) або JetBrains Rider;
- **Frontend**: Visual Studio Code або JetBrains WebStorm.

2. **Контроль версій**

- Git з віддаленим репозиторієм на GitHub (branch-based workflow, pull requests).

3. **Конфігураційні файли**

- appsettings.Development.json / appsettings.Production.json – налаштування підключень до бази даних, Elasticsearch, параметри логування;
- angular.json, tsconfig.json, package.json — конфігурація Angular-проєкту;

3.2.2 Реалізація бекенду (ASP.NET Core Web API)

Проєкт Web API створюється на основі готового шаблону ASP.NET Core. Після генерації початкового набору файлів, наприклад, за допомогою команди

```
dotnet new webapi --name Spendify.Api
```

Або як показано на Рисунку 3.4 створити за допомогою IDE.

Для забезпечення чіткого розмежування відповідальності система розбивається на окремі проєкти-шари:

- **Spendify.Api** – контролери й конфігурація HTTP-конвеєра;
- **Spendify.BL** – бізнес-логіка та сервіси;
- **Spendify.DAL** – доступ до бази даних;
- **Spendify.Domain** – доменні моделі та інтерфейси.

Таке мультипроєктне рішення гарантує інкапсуляцію кожного рівня абстракції та спрощує подальшу підтримку і тестування.

Модуль авторизації може бути розгорнутий в межах основного API-проєкту або виокремлений у самостійний сервіс (як у Spendify). Окремий модуль аутентифікації дозволяє реалізувати єдину точку входу для різних клієнтських платформ (мобільних, веб чи десктопних) і спрощує інтеграцію з зовнішніми провайдерами (Google, Facebook тощо). Такий підхід рекомендований для масштабованих корпоративних рішень, де одна централізована служба авторизації обслуговує численні клієнтські додатки.

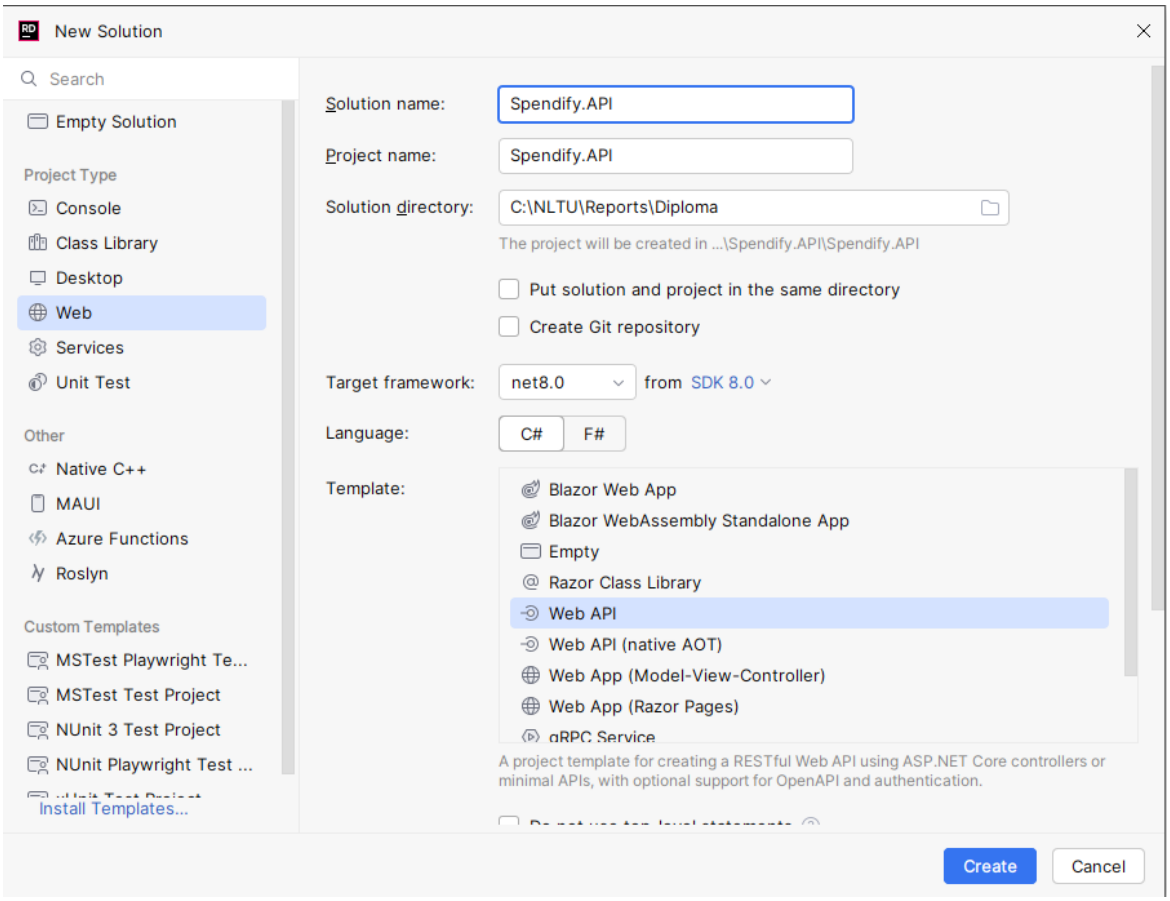


Рисунок 3.4 – Приклад створення проекту з готовим темплейтом

У ролі доменних сутностей виступають класи User, Expense, Category, Item, Location та зв’язуюча сутність ItemExpense. Кожна модель описує набір властивостей із відповідними зовнішніми ключами та навігаційними властивостями для Entity Framework Core.

Приклад класу Expense:

```
public class Expense : BaseEntity
{
    35 usages
    public DateTime Date { get; set; }
    14 usages
    public Guid UserId { get; set; }
    4 usages
    public User User { get; set; }
    13 usages
    public Guid CategoryId { get; set; }
    23 usages
    public Category Category { get; set; }
    32 usages
    public List<ItemExpense> ItemExpenses { get; set; }
    7 usages
    public Guid LocationId { get; set; }
    14 usages
    public Location Location { get; set; }
}
```

Для налаштування контексту доступу до даних для Spendify:

1. **Оголошення контексту**

- Клас ExpensesContext наслідується від DbContext та вміщує властивості DbSet<TEntity> для всіх доменних сутностей (Users, Expenses, Categories, Items, Locations, ItemExpenses).
- Завдяки цьому EF Core розуміє, які таблиці необхідно синхронізувати з моделями.

2. **Налаштування моделі**

- У методі OnModelCreating(ModelBuilder modelBuilder) прописано складний (composite) первинний ключ для сутності ItemExpense та конфігуровано зв’язки «один-до-багатьох» між Item ↔ ItemExpenses і Expense ↔ ItemExpenses.
- Така конфігурація гарантує коректну побудову проміжної таблиці для зв’язків витрат із кількома товарами.

3. **Рядок підключення**

У файлі appsettings.json секція визначає параметри підключення до SQL Server – назву сервера, ім’я бази даних та режим автентифікації.

```
"ConnectionStrings": {
  "db": "Server=localhost;Database=ExpensesTrackingSystemDB;Trusted_Connection=True;"
},
```

Приклад конфігурації ExpensesContext’а:

```
public class ExpensesContext : DbContext
{
    18 usages
    public DbSet<User> Users { get; set; }
    17 usages
    public DbSet<Item> Items { get; set; }
    25 usages
    public DbSet<Expense> Expenses { get; set; }
    21 usages
    public DbSet<Category> Categories { get; set; }
    17 usages
    public DbSet<Location> Locations { get; set; }
    5 usages
    public DbSet<ItemExpense> ItemExpenses { get; set; }
    4 usages
    public DbSet<PhotoFile> PhotoFiles { get; set; }
    4 usages
    public DbSet<Alert> Alerts { get; set; }
    5 usages
    public DbSet<UserAlert> UserAlerts { get; set; }
    6 usages
    public DbSet<Ticket> Tickets { get; set; }
    2 usages
    public DbSet<TicketChat> TicketChats { get; set; }
    1 usage
    public DbSet<ChatMessage> ChatMessages { get; set; }

    ◇ Andrii Pasternak
    public ExpensesContext()
    {
    }
◇ Andrii Pasternak
public ExpensesContext(DbContextOptions<ExpensesContext> options) : base(options)
{
}
◇ Andrii Pasternak
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new CategoryConfiguration());
    modelBuilder.ApplyConfiguration(new ItemConfiguration());
    modelBuilder.ApplyConfiguration(new ItemExpenseConfiguration());
    modelBuilder.ApplyConfiguration(new UserConfiguration());
    modelBuilder.ApplyConfiguration(new ExpenseConfiguration());
    modelBuilder.ApplyConfiguration(new LocationConfiguration());
    modelBuilder.ApplyConfiguration(new PhotoFileConfiguration());
    modelBuilder.ApplyConfiguration(new AlertConfiguration());
    modelBuilder.ApplyConfiguration(new UserAlertConfiguration());
    modelBuilder.ApplyConfiguration(new TicketConfiguration());
    modelBuilder.ApplyConfiguration(new TicketChatConfiguration());
    modelBuilder.ApplyConfiguration(new ChatMessageConfiguration());

    base.OnModelCreating(modelBuilder);
}
```

При конфігурації Program.cs потрібно налаштувати такі ключові моменти:

1. **Реєстрація DbContext** із SQL Server.
2. **Інтеграція Elasticsearch** через Nest-клієнт (IElasticClient).
3. **Впровадження залежностей (DI)** для сервісів:

- IUserService, IExpenseService, IElasticsearchService та інші.

4. **JWT-автентифікація та авторизація** зі зчитуванням ключа, видавця та аудиторії з конфігурації.

5. **Middleware pipeline**: UseAuthentication(), UseAuthorization(), MapControllers().

Ключовим хотів би виділити логіку інтеграції Elasticsearch, яка використовувалась для пошуку витрат за назвою предметів, категорії чи локації. У межах BL (Business Logic Layer) інтеграція Elasticsearch реалізована через окремий сервіс IElasticSearchService і його конкретну реалізацію ElasticSearchService. Повний код реалізації цього сервісу наведено в Додатку А.

Основні моменти, які варто згадати в цьому підрозділі:

1. **Інтерфейс та методи сервісу**

- BulkIndexExpensesAsync(List<Expense> expenses, …) — масове індексування списку витрат. Використовує BulkDescriptor, куди для кожного об’єкта Expense виконується перетворення у внутрішню модель ExpenseDocument (метод MapToDocument). У разі помилки запиту кидається виняток;
- CreateExpenseIndexAsync(PaginationExpenseItemsDTO expense, Guid userId, …) — індексування однієї нової витрати, наприклад після її створення у БД. Поля документа формуються напряму з DTO (ціна, дата, категорія, локація, перелік товарів, та списки підпольових полів: CategoryName, ItemNames, LocationName);
- UpdateExpenseIndexAsync (Guid userId, Guid id, PaginationExpenseItemsDTO expense, …) — оновлення існуючого документа в індексі. Оновлюються лише ті поля, які могли змінитися (ціна, дата, назви полів для пошуку). Використовується client.UpdateAsync<ExpenseDocument>();
- DeleteExpenseAsync(Guid expenseId) — видалення запису з індексу за його Id;
- SearchExpensesAsync(Guid userId, PaginationFilter filter, …) — основний метод пошуку. Він повертає ElasticSearchResponse, який містить

перелік знайдених DTO (PaginationExpenseItemsDTO) із відсортованими та відфільтрованими результатами, а також загальну кількість документів для пагінації.

2. **Мапінг доменної моделі у документ для Elasticsearch**

- Метод MapToDocument(Expense expense) бере оригінальний об’єкт Expense (з усіма зв’язками: категорія, локація, список ItemExpenses) і конвертує його у ExpenseDocument;
- Поля Category, Location, Items (детальна інформація по кожному товару) передаються як вкладені об’єкти (DTO), а додатково формуються текстві представлення CategoryName, ItemNames (список назв товарів) та LocationName. Це дозволяє виконувати пошук саме за назвою категорії, назвою товару чи назвою локації.

3. **Побудова запиту для пошуку (BuildBaseQuery)**

- Спочатку створюється TermQuery для фільтрації по полю UserId, щоб користувач бачив лише свої витрати;
- Якщо у пагінаційному фільтрі вказана дата-фільтрація за тиждень, місяць чи три місяці, то додається DateRangeQuery з відповідними межами;
- Якщо поле SearchQuery не порожнє, будується MultiMatchQuery із автофаззінгом (Fuzziness.Auto) по полям CategoryName, ItemNames та LocationName. Це дає змогу шукати витрати за частковою назвою категорії, товару чи локації;
- Всі ці підзапити об’єднуються у BoolQuery з секціями Must (обов’язкові умови) та Filter (фільтрація за датою). У результаті ми отримуємо такі критерії пошуку: «тільки мої витрати», «за вказаним текстом» і «за потрібним проміжком дат».

4. **Сортування й пагінація**

- Якщо користувач не вказує власний стовпець для сортування, результати віддаються за спаданням дати (SortOrder.Descending). Інакше можна сортувати за будь-яким доступним полем (у даному коді показано лише варіант із датою, але легко розширити);

- Для пагінації використовується .Skip(filter.PageNumber * filter.PageSize) та .Take(filter.PageSize), а в кінці результатів враховується загальна кількість документів через TrackTotalHits().

5. **Використання у контролері** – у методі GetUserExpensesPagination (HTTP GET users) викликається elasticService.SearchExpensesAsync(...), передається унікальний UserId (з currentUserService), та сам об’єкт PaginationFilter (містить параметри сторінки, розмір, текст пошуку й фільтр за період). Після отримання ElasticSearchResponse результати конвертуються в оновлений курс валют (якщо потрібно) через expenseService.RecalculateCurrency(...), а далі формуються у стандартний PagedResponse<List<PaginationExpenseItemsDTO>>, який повертається клієнту.

Коротко про ключову роль Elasticsearch у BL:

- Після створення/оновлення/видалення витрат у БД викликаються відповідні методи сервісу для індексування чи видалення документа у Elastic;
- Під час запиту на список витрат BL звертається безпосередньо до Elasticsearch, не завантажуючи всі записи з бази, а повертаючи вже відфільтровані та відсортовані результати;
- Завдяки поєднанню MultiMatchQuery з полями CategoryName, ItemNames, LocationName користувач може шукати витрати за назвою категорії, назвою товару чи локації, включно з нечітким пошуком (фаззінг);
- Додаткові фільтри за періодом (“week”, “month”, “threeMonths”) реалізовані за допомогою DateRangeQuery, що дає змогу швидко обмежувати результати за потрібними діапазонами дат.

Ця логіка розміщена у BL-шарі, щоб відділити «чисто» бізнес-правила пошуку й індексування від контролеру (API-level), де просто виконується запит і повернення результатів у необхідному форматі. Також така архітектура полегшує тестування: можна замокати IElasticSearchService і перевіряти, що контролер правильно обробляє відповідь, не заглиблюючись у деталі Elasticsearch.

3.2.3 Реалізація фронтенду (Angular)

Для створення клієнтської частини Spendify було використано фреймворк Angular (версія 17+). Який використовує TypeScript, що є безпечною надбудовою над JavaScript'ом і дуже популярним у великих комерційних проектах. Angular також надає потужний набір інструментів – CLI для швидкого генерування компонентів, сервісів і модулів, а також вбудовану підтримку AOT-компіляції і tree-shaking для оптимізації розміру фінального бандлу. Використання RxJS у поєднанні з реактивними формами забезпечує гнучке та ефективне управління потоками даних і станом застосунку. Завдяки модульній архітектурі та ленивому завантаженню (lazy-loading) сторінок, фронтенд Spendify залишається масштабованим і підтримує швидке завантаження навіть при зростанні функціональності. Проєкт генерується за допомогою Angular CLI:

```
ng new spendify-app --routing --style=scss
```

де вмикається модуль маршрутизації (--routing) і використовується SCSS для стилізації.

Далі структура проєкту виглядає як показано на Рисунку 3.5.

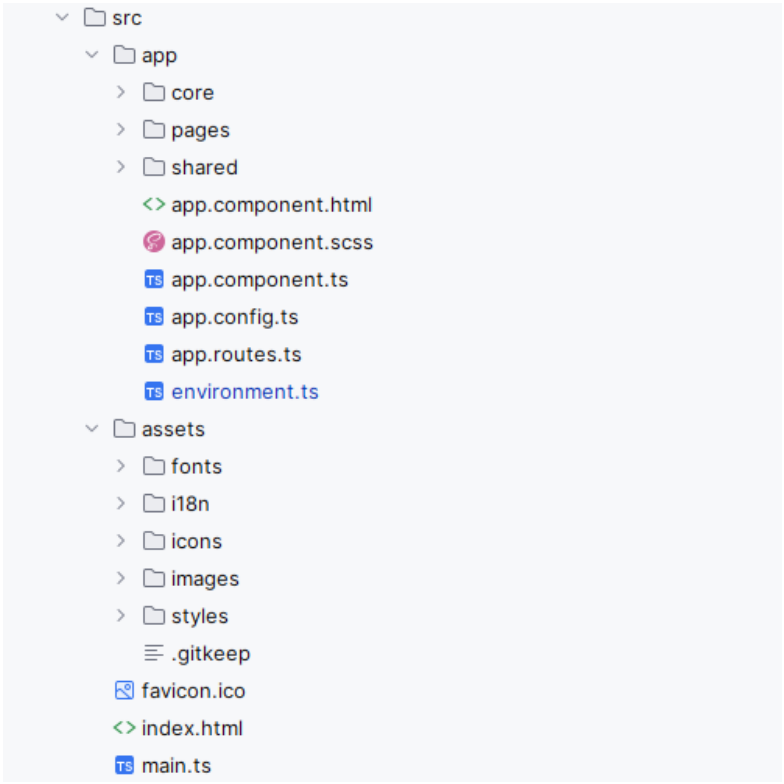


Рисунок 3.5 – Структура клієнтської частини (Frontend) проекту

3.2.4 Технічні вимоги до обладнання

Таблиця 3.1. Технічні вимоги до обладнання.

Середовище	**CPU**	**RAM**	**Диск**	**OS**
Продакшн	4-ядерний Intel Xeon (або AMD еквівалент)	≥ 8 GB (рекомендовано 16 GB)	SSD: ≥ 120 GB ОС + ≥ 250 GB БД	Ubuntu 22.04 LTS або Windows Server 2019
Локальна машина	Intel i5 (8-е покоління) або вище	≥ 8 GB (рекомендовано 16 GB)	SSD: ≥ 256 GB	Windows 10/11 або сучасна Linux

3.3 Інтерфейс

У цьому розділі розглянеться інтерфейс користувача, який зайшов на сайт Spendify. Адміністраторський функціонал буде наведений у вигляді скріншотів у Додатку Б.

Коли користувач вперше заходить на сайт Spendify, перед ним відкривається головна сторінка з привабливим заголовком і коротким описом можливостей сервісу. Справа зверху розташовані кнопки «Увійти» та «Зареєструватися», що забезпечують швидкий доступ до авторизації та реєстрації. Також на сторінці є посилання на соціальні мережі (рис.3.6).

Рисунок 3.6 – Головна сторінка Spendify

Після натискання кнопки «Зареєструватися» користувач переходить на форму реєстрації, де йому пропонується ввести адресу електронної пошти, придумати надійний пароль та підтвердити згоду з умовами використання. Поля форми чітко промарковані, а кнопка підтвердження активується тільки після заповнення всіх обов’язкових полів (рис.3.7).

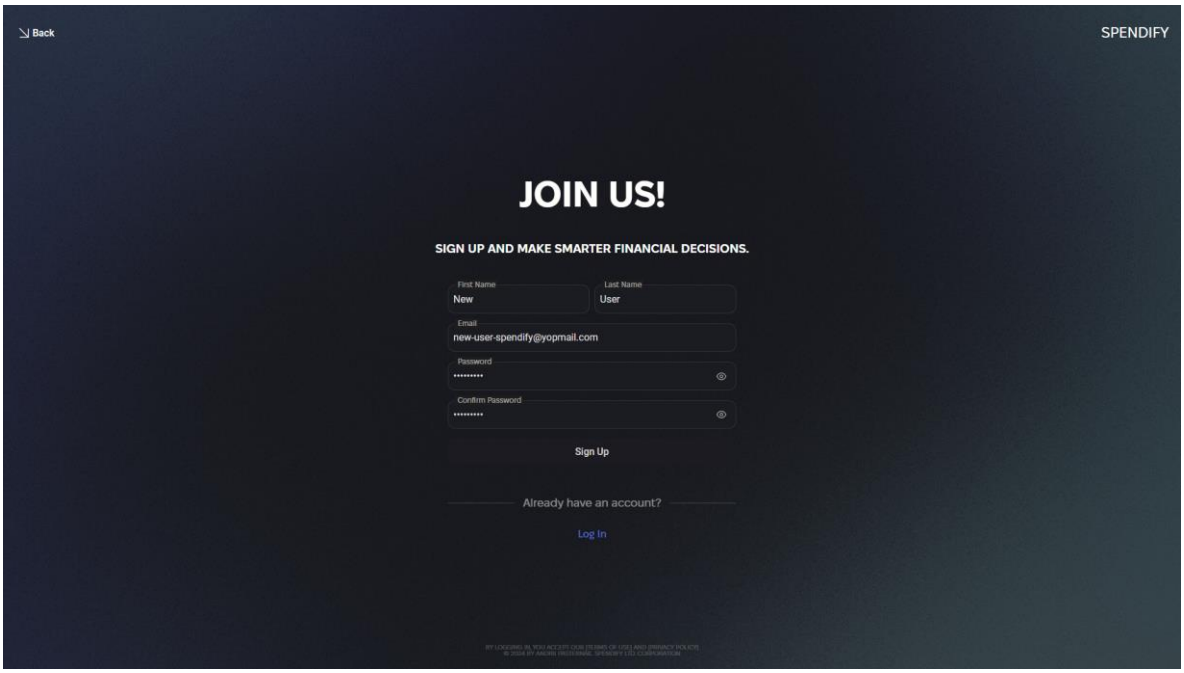


Рисунок 3.7 – Сторінка реєстрації, форма для реєстрації

Після відправлення даних система повідомляє користувачу про необхідність підтвердити поштову адресу – на вказану електронну пошту надсилається лист із посиланням активації (рис.3.8).

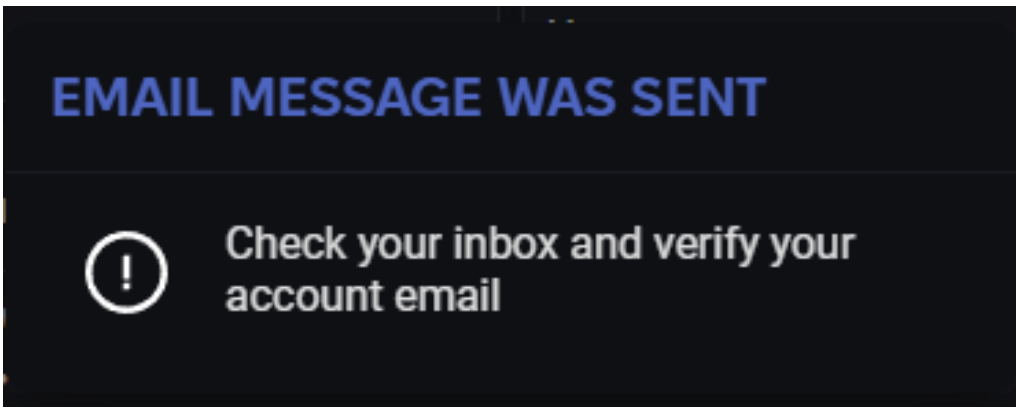


Рисунок 3.8 – Сповіщення про необхідність підтвердження аккаунта

Тільки після переходу за цим посиланням обліковий запис вважається підтвердженим, і користувач може успішно увійти в систему (рис.3.9).

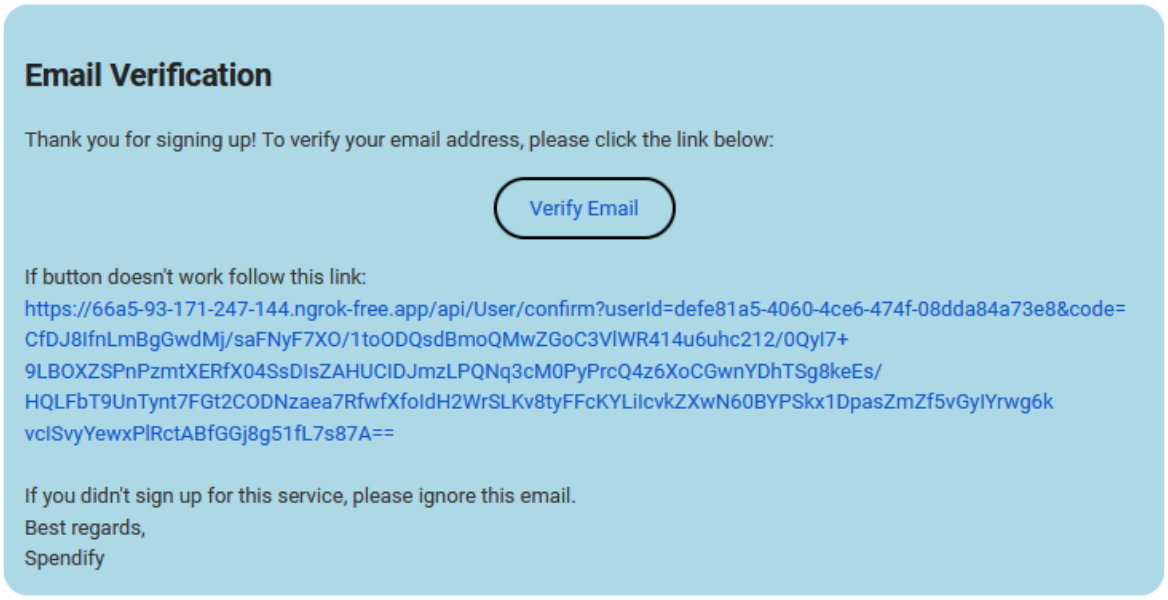


Рисунок 3.9 – Е-мейл з проханням перейти по посиланню

Після того як користувач перейде по посиланню, тоді йому покажуть, що він успішно верифікувався (рис.3.10).

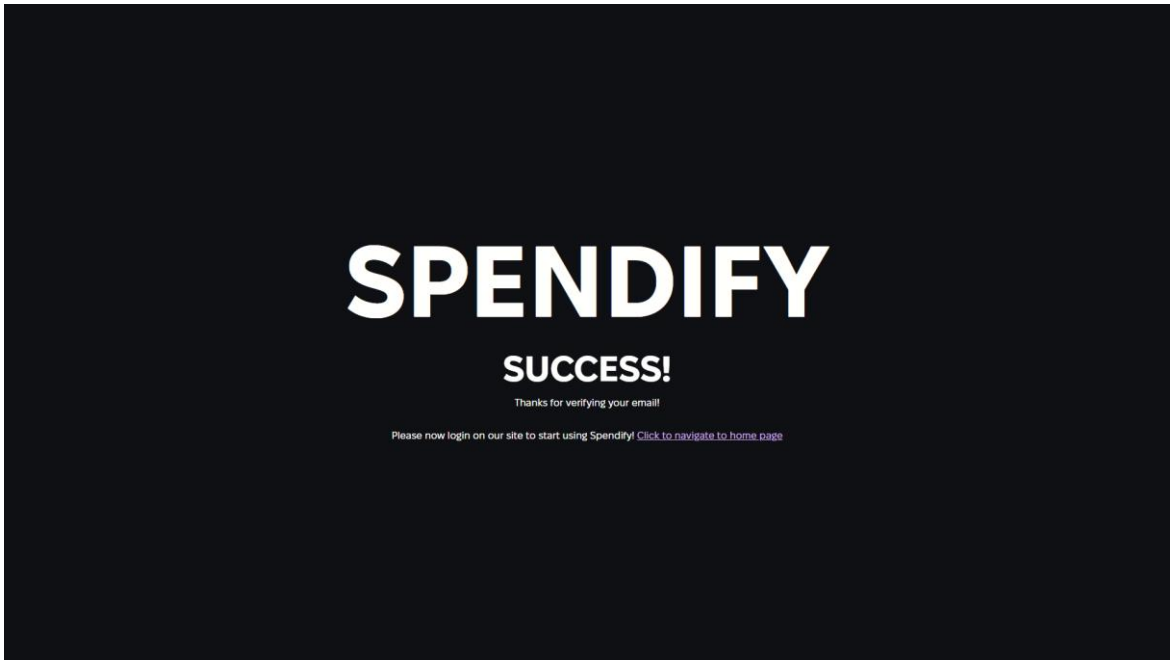


Рисунок 3.10 – Успішна реєстрація для користувача

Після успішної реєстрації, або якщо у користувача вже є аккаунт, він обирає «Увійти» та бачить форму авторизації із полями для введення електронної пошти та пароля. При введенні коректних даних та натисканні кнопки «Увійти» користувач автоматично перенаправляється на основну сторінку – Expenses (рис.3.11).

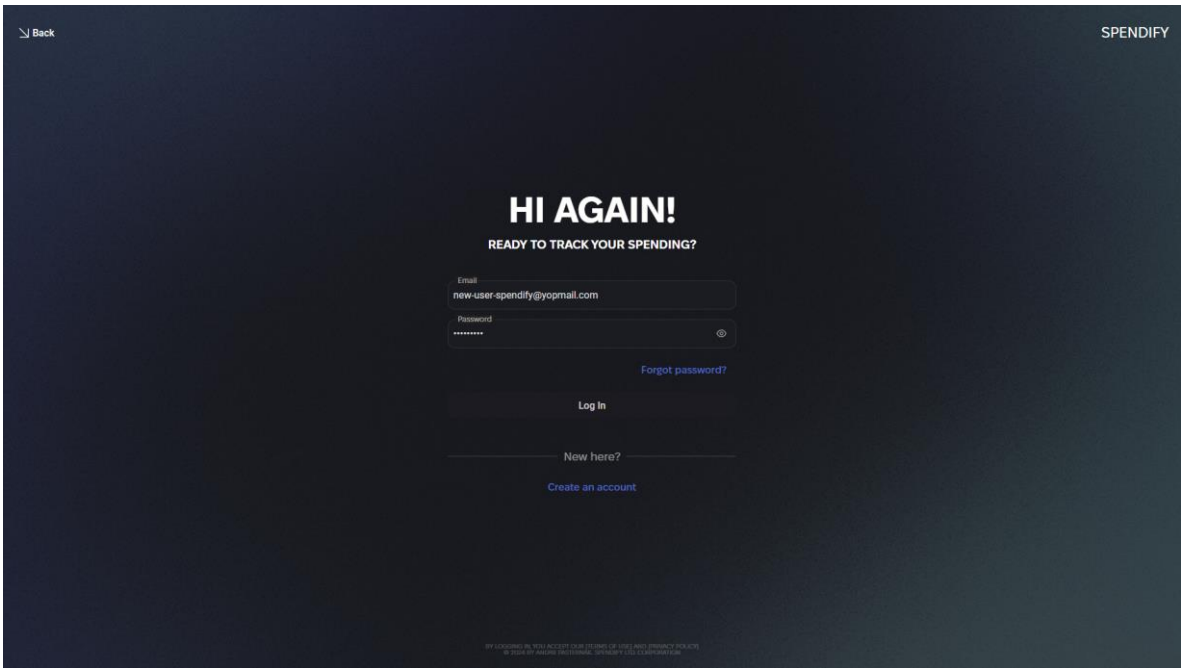


Рисунок 3.11 – Сторінка логіну

На Expenses зібрані ключові елементи для роботи з витратами: кнопка «Нова витрата», загальний баланс і зведені графіки. Саме звідси користувач

може розпочати створення та редагування записів про свої фінансові операції. Інтерфейс інтуїтивно зрозумілий і дозволяє оперативно перейти до будь-якої функції управління бюджетом (рис.3.12).

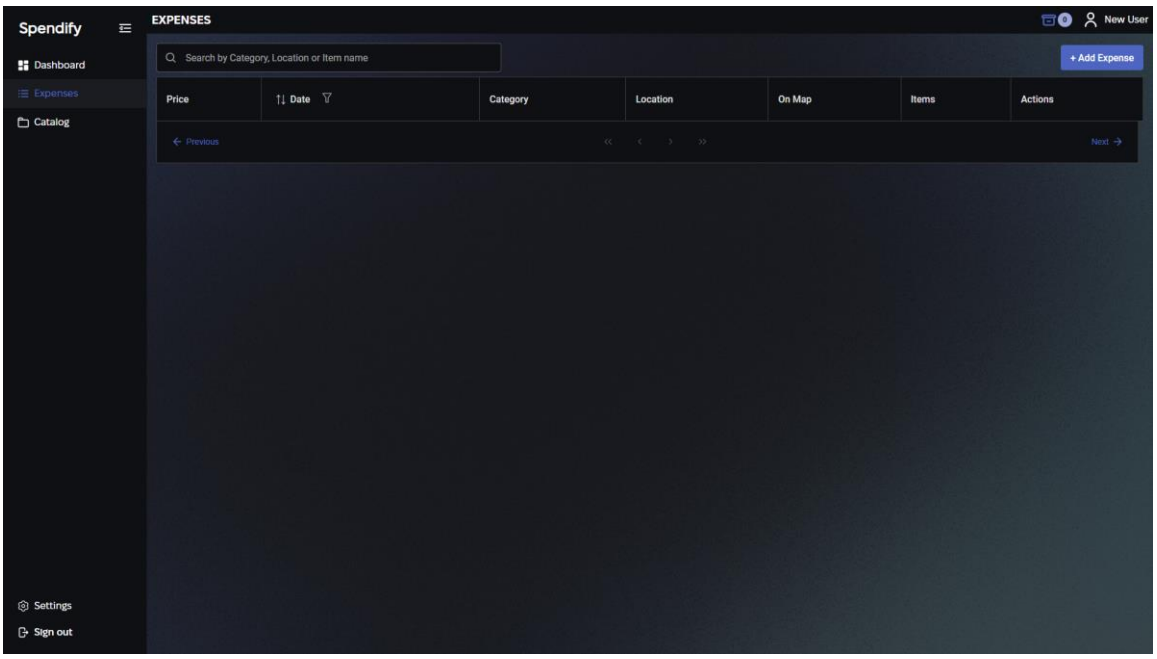


Рисунок 3.12 – Сторінка витрат користувача

На цій сторінці він може створити нову витрату або використовуючи пошук шукати по назві предмета, категорії чи локації витрати, які були ним зроблені (рис.3.13).

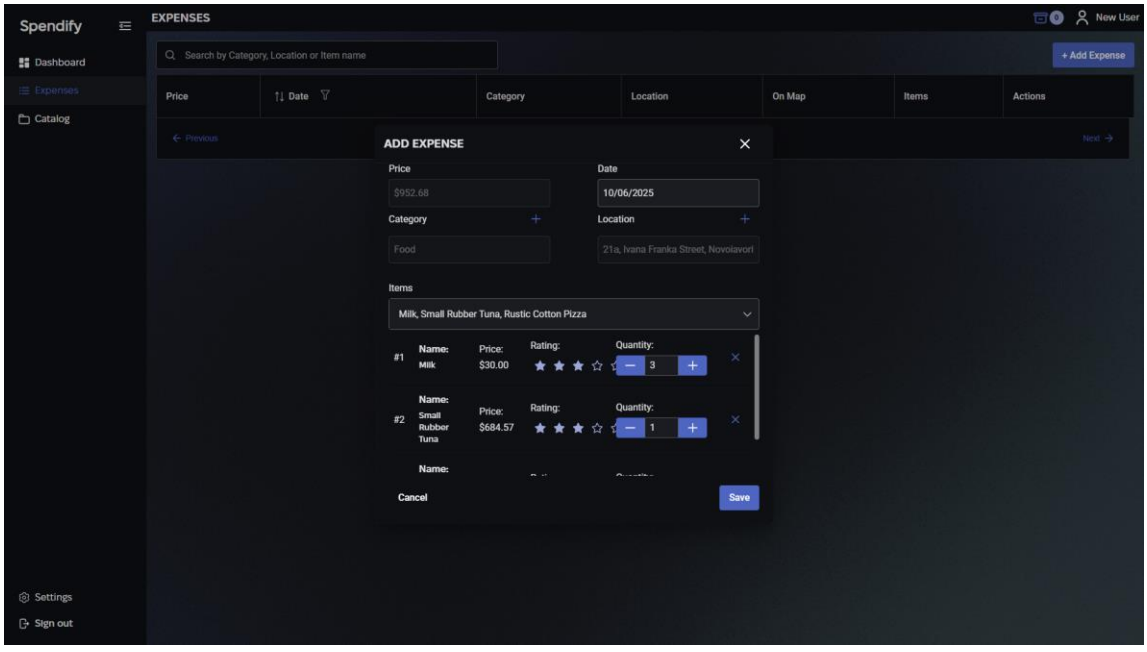


Рисунок 3.13 – Форма створення витрати

На сторінці «Каталог» користувач бачить набір карток трьох типів – Предмети, Категорії та Локації, які доступні виключно йому в особистому

просторі. Кожна картка містить назву об’єкта, короткий опис або іконку, а також індикатор статусу (відкрита чи прихована) (рис.3.14).

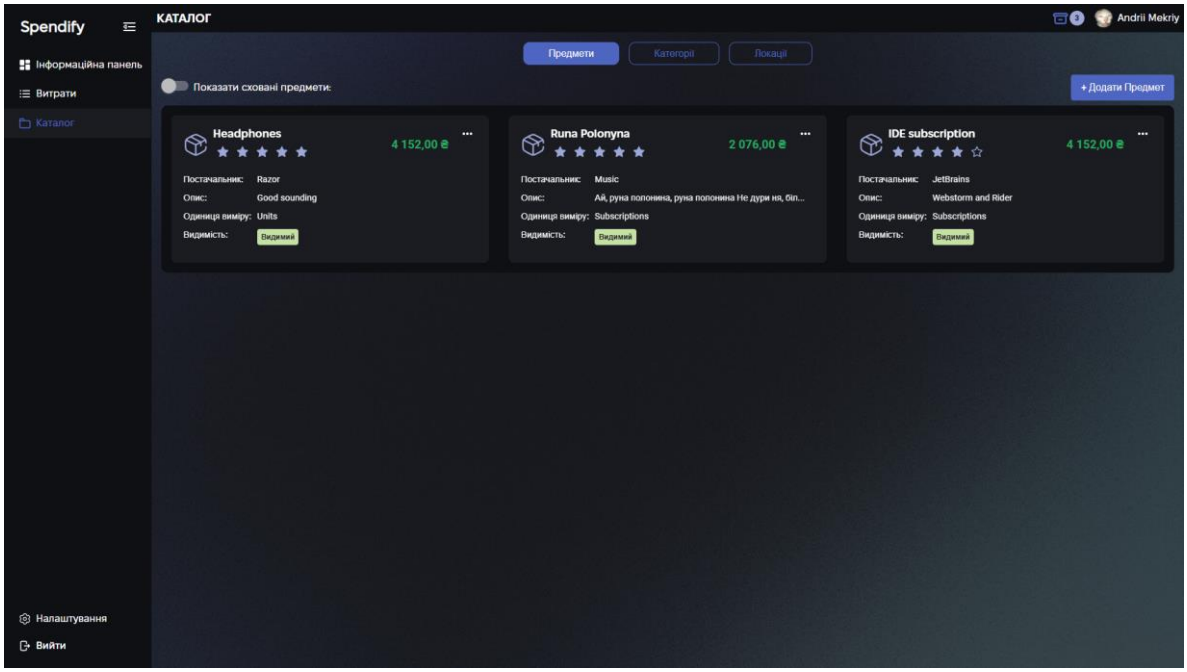


Рисунок 3.14 – Сторінка каталог

Для додавання нового елемента достатньо натиснути кнопку «Додати» у верхній частині відповідного розділу – після чого відкривається спливаюча форма з полями для заповнення назви, опису та вибору пов’язаних атрибутів (рис.3.15).

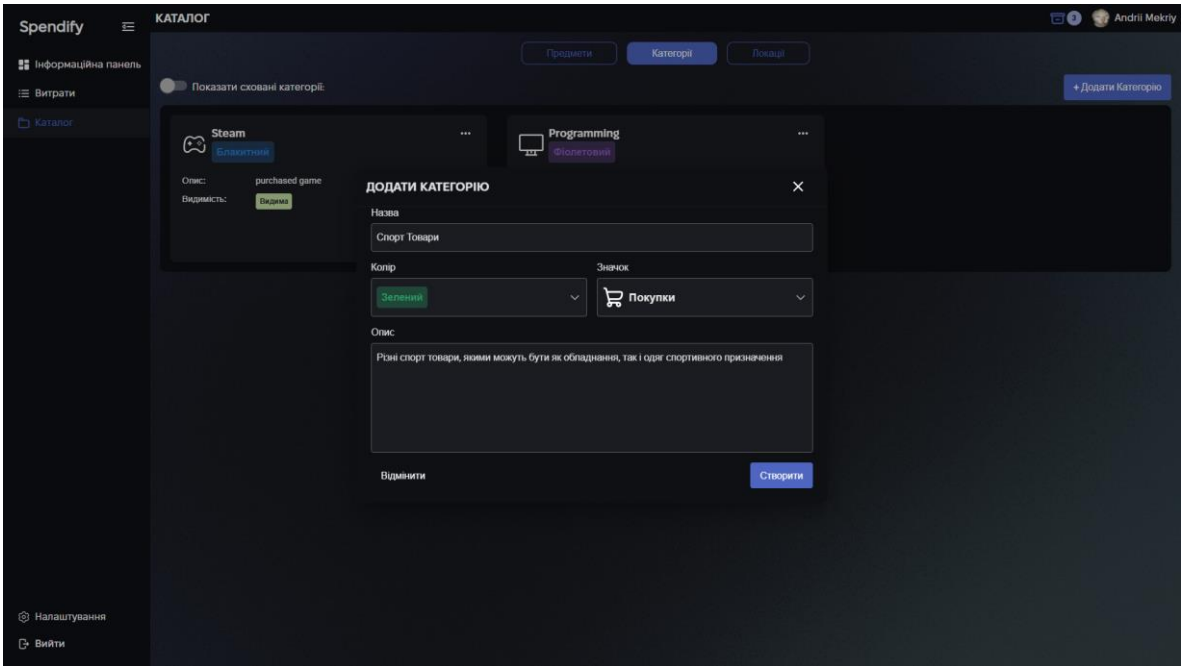


Рисунок 3.15 – Створення категорії

Редагування існуючих записів відбувається безпосередньо на картці: при наведенні курсора на неї з’являються іконки «Редагувати» та «Приховати/Показати». Перший відкриває форму зі збереженими даними для зміни, другий миттєво змінює видимість елемента в загальному переліку (рис.3.16).

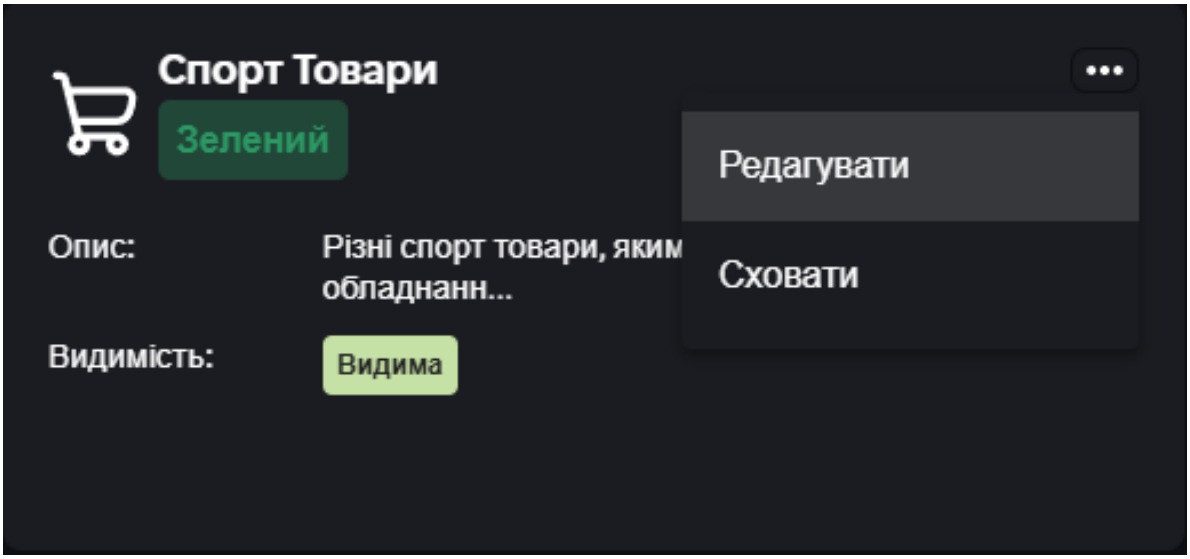


Рисунок 3.16 – Кнопки редагувати і сховати

Приховані елементи залишаються доступними в окремому фільтрі «Приховані», до якого можна швидко перейти за допомогою вкладки у верхньому меню сторінки. Тут користувач може за потреби повернути будь-яку картку до загального каталогу або остаточно видалити її (рис.3.17).

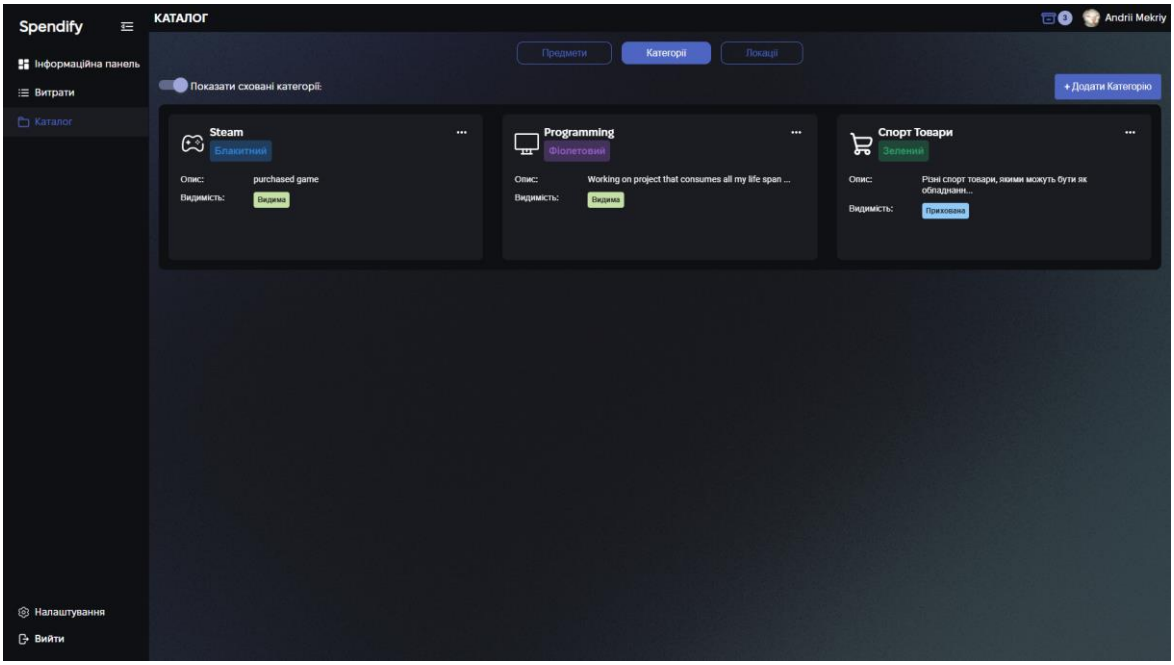


Рисунок 3.17 –Показує приховані категорії

Загалом інтерфейс «Каталогу» спроектований таким чином, щоб забезпечити максимальну гнучкість у керуванні переліками предметів, категорій та локацій, зберігаючи при цьому простоту й інтуїтивність процесу створення, редагування та організації даних.

Перейшовши в Інформаційну панель(Dashboard), користувач бачить сторінку, яка відображає поточний стан його фінансів у зручному огляді. У верхній частині сторінки розташовано кругову діаграму з розподілом витрат за категоріями та таблицю нещодавніх транзакцій із зазначенням суми, дати, категорії та місця оплати. Нижче користувачу пропонуються швидкі дії для додавання нової витрати, предмета, категорії або локації, що дозволяє миттєво розширювати свої списки без переходу на інші сторінки. У самому низу розміщено секцію місячного огляду, де відображаються зведені графіки для поточного й попереднього місяців, а також подальша деталізація витрат за категоріями. Такий дизайн забезпечує користувачеві оперативний доступ до ключових показників і дозволяє швидко реагувати на зміни фінансового стану (рис.3.18).

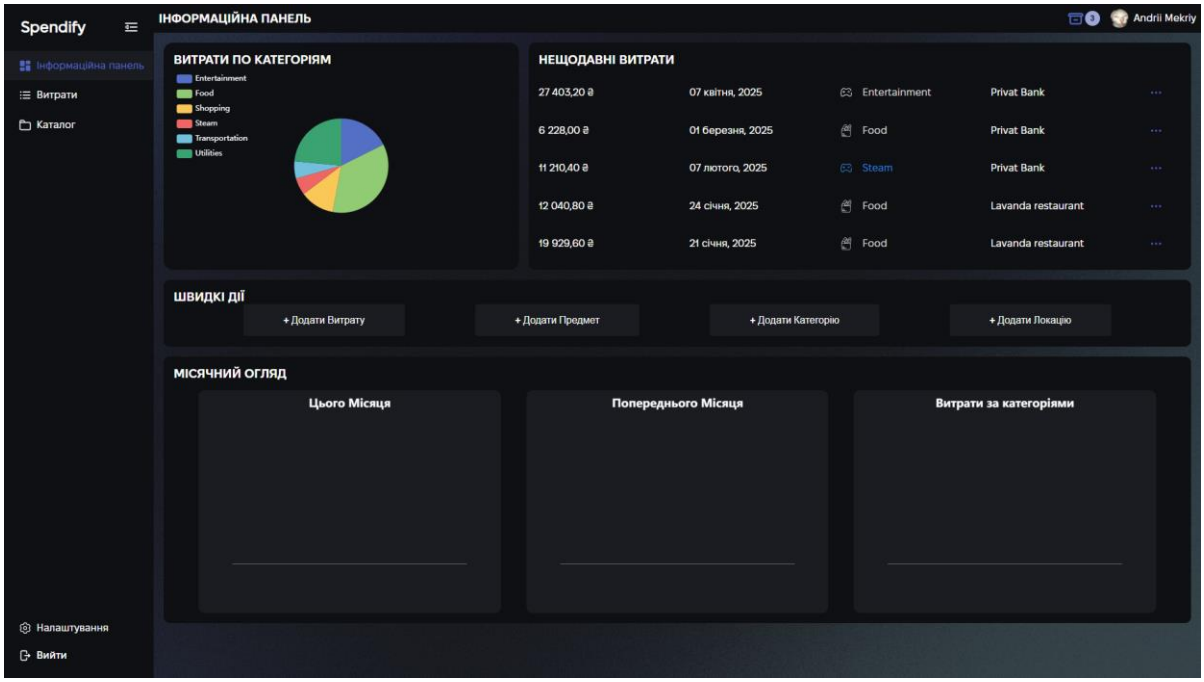


Рисунок 3.18 – Інформаційна панель

На сторінці «Налаштування» вгорі розташовано перемикач вкладок «Налаштування» та «Підтримка», який дозволяє швидко переходити між персональними параметрами та списком звернень до служби підтримки. У

розділі «Налаштування» відображається аватар користувача, повне ім’я й адреса електронної пошти. Нижче знаходиться панель редагування профілю з полями для імені та прізвища, кнопками «Редагувати», «Змінити пароль» і «Видалити» акаунт. Після натискання «Редагувати» поля стають доступними для зміни, і з’являється кнопка «Зберегти».

При зміні паролю, користувач підтверджує зміну паролю через пошту і коли він це зробить його переадресовує на сторінку зміни паролю де він зможе змінити пароль на новий (рис.3.19).

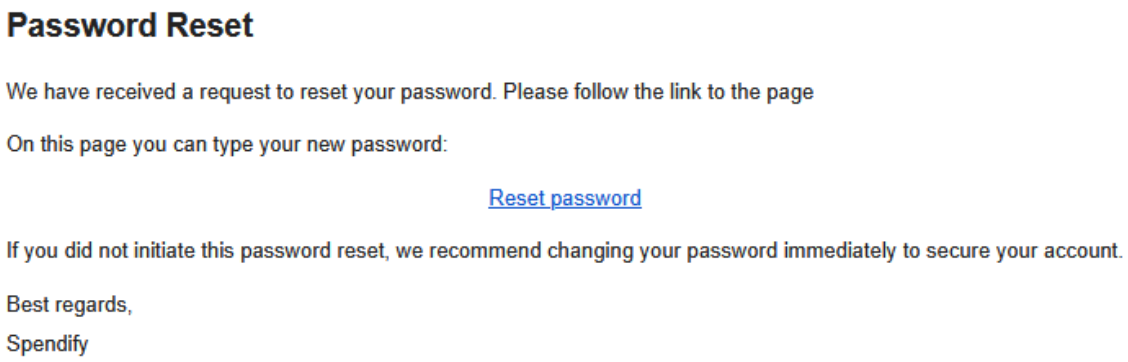


Рисунок 3.19 – Лист зі зміною паролю

Трохи нижче розміщено налаштування інтерфейсу та валюти: два випадаючі списки дозволяють обрати мову відображення (наприклад, українська) та основну валюту (UAH – гривня), натисканням кнопки «Оновити» застосовуються нові параметри. В самому низу блоку знаходиться секція для зміни фотографії профілю – користувач може обрати файл через кнопку «Choose» та завантажити його (рис.3.20).

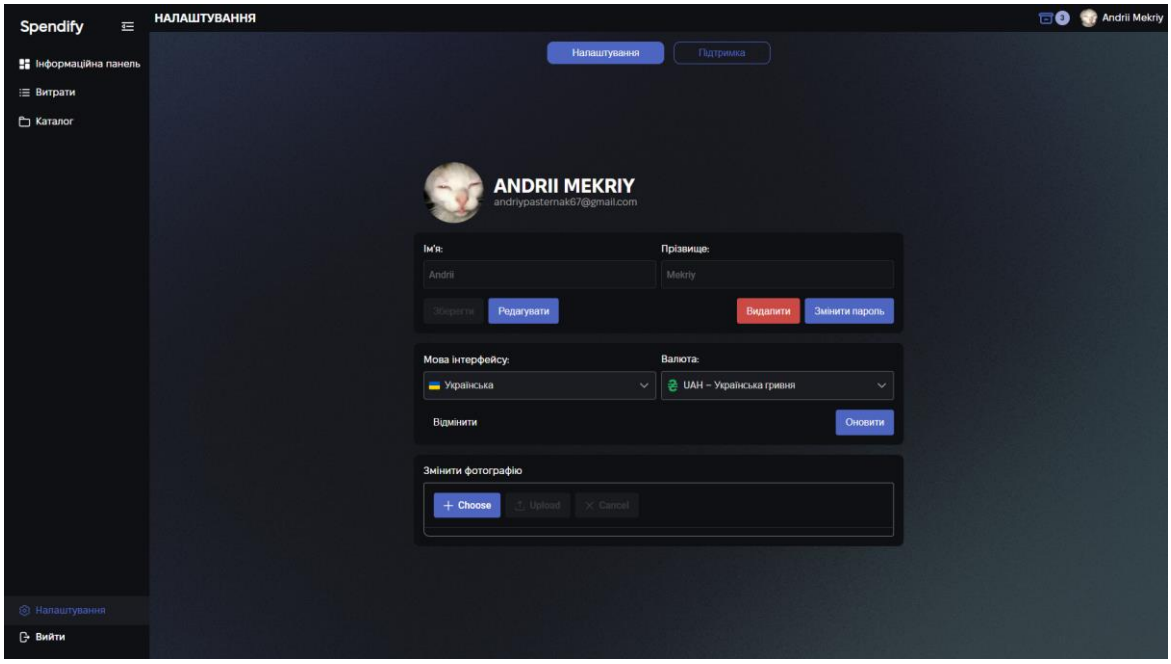


Рисунок 3.20 – Сторінка налаштування

Після переходу на вкладку «Підтримка» користувачу відкривається перелік його звернень у вигляді карток (рис.3.21).

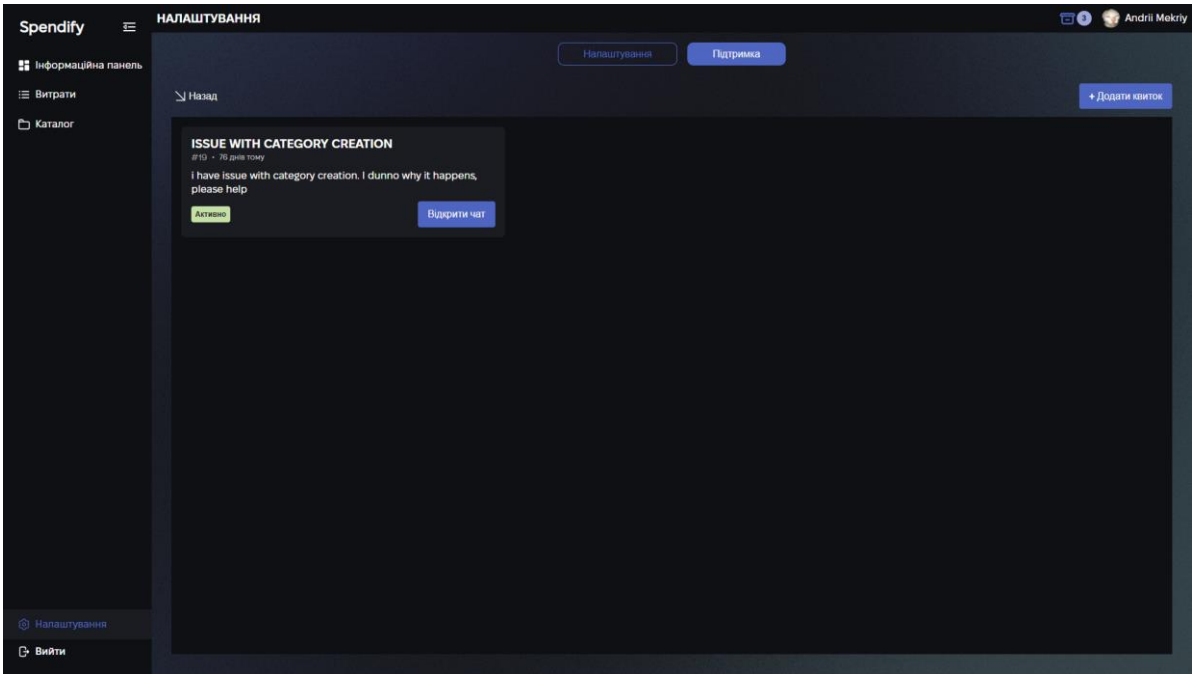


Рисунок 3.21 – Сторінка підтримки

Кожна картка містить заголовок запиту, його номер, час створення, короткий опис та статус (наприклад, «Активно» або «Вирішено»), а також кнопку «Відкрити чат» для перегляду обговорення з підтримкою. Над списком розташована кнопка «+ Додати квиток», яка відкриває форму створення нового

звернення. Інтерфейс виконаний у єдиному стилі з іншими розділами Spendify, що забезпечує цілісність користувацького досвіду (рис.3.22).

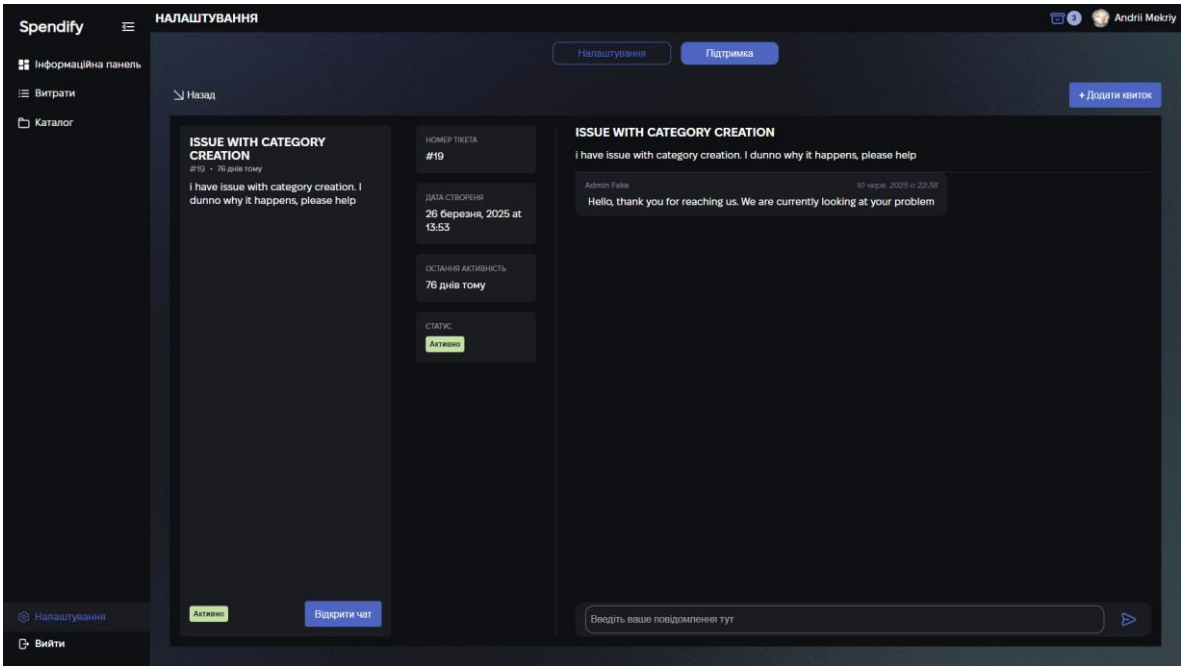


Рисунок 3.22 – Відкритий чат тікета підтримки

ВИСНОВОК

У процесі виконання дипломної роботи було реалізовано вебзастосунок Spendify для обліку особистих витрат. Система охоплює повний цикл взаємодії користувача — від реєстрації та автентифікації до введення й аналізу витрат. Особливу увагу приділено зручності інтерфейсу, приватності даних і гнучкому управлінню категоріями, предметами та локаціями витрат.

Розроблений функціонал дозволяє користувачеві ефективно відстежувати фінансову активність, отримувати статистичні огляди та візуалізації, що сприяє підвищенню обізнаності щодо особистого бюджету. Окремий розділ налаштувань забезпечує можливість адаптації інтерфейсу та профілю під індивідуальні потреби, а вбудована підтримка користувачів дозволяє оперативно вирішувати проблеми в роботі сервісу. Таким чином, реалізована система є прикладом сучасного застосунку для ведення особистої фінансової аналітики, який поєднує в собі функціональність, доступність та інтуїтивно зрозумілий інтерфейс.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сітайло А. С. Вебзастосунок для розширеного обліку особистого фінансового стану користувача : дипломний проєкт бакалавра : 121 Інженерія програмного забезпечення / наук. кер. Заболотня Т. М. – Київ, 2024. – 123 с. – URL: https://ela.kpi.ua/items/19411227-d6be-495b-b01d-e99503115adc
2. Сірокомський М. С. Мобільний додаток для обліку особистих витрат : дипломний проєкт бакалавра : Комп'ютерна інженерія / наук. кер. Павлов В. Г. – Київ, 2022. – 64 с. – URL: https://ela.kpi.ua/server/api/core/bitstreams/a7a69af9-d1ef-4bad-9c71-ae4b16b44de2/content
3. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. – Addison-Wesley, 1994.
4. Fowler M. Patterns of Enterprise Application Architecture. – Addison-Wesley, 2002.
5. Freeman E., Robson E. Head First Design Patterns. – O'Reilly Media, 2021.
6. MDN Web Docs. Офіційна документація HTML, CSS, JavaScript [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/ – Назва з екрана.
7. Angular Documentation. Офіційна документація фреймворку Angular [Електронний ресурс]. – Режим доступу: https://angular.dev/ – Назва з екрана.
8. OWASP Foundation. Top 10 Web Application Security Risks [Електронний ресурс]. – Режим доступу: https://owasp.org/www-project-top-ten/ – Назва з екрана.
9. Двірничук К. В., Вацек Д. О. Веб-програмування та веб-дизайн : навч. посіб. – Чернівці : Чернівецький національний університет імені Юрія Федьковича, 2022. – 471 с.
10. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures (REST) : doctoral dissertation. – University of California, 2000.

ДОДАТКИ

ДОДАТОК А

IElasticSearchService.cs

```
using EST.Domain.DTOs;
using EST.Domain.Helpers;
using EST.Domain.Models;
using EST.Domain.Pagination;

namespace EST.BL.Interfaces;

5 usages  1 inheritor  Andrii Pasternak
public interface IElasticSearchService
{
    1 usage  1 implementation  Andrii Pasternak
    public Task BulkIndexExpensesAsync(List<Expense> expenses, CancellationToken token = default);
    1 usage  1 implementation  Andrii Pasternak
    public Task DeleteAllElasticExpenses();
    1 usage  1 implementation  Andrii Pasternak
    public Task DeleteExpenseAsync(Guid expenseId);

    1 implementation  Andrii Pasternak
    public Task CreateExpenseIndexAsync(
        PaginationExpenseItemsDTO expense,
        Guid userId,
        CancellationToken token = default);

    3 usages  1 implementation  Andrii Pasternak
    public Task UpdateExpenseIndexAsync(
        PaginationExpenseItemsDTO result,
        Guid userId,
        CancellationToken token);

    1 usage  1 implementation  Andrii Pasternak
    Task<ElasticSearchResponse> SearchExpensesAsync(
        Guid userId,
        PaginationFilter filter,
        CancellationToken token);
}
```

ElasticSearchService.cs

```
using EST.BL.Interfaces;
using EST.Domain.DTOs;
using EST.Domain.ElasticSearchDTOs;
using EST.Domain.Helpers;
using EST.Domain.Models;
using EST.Domain.Pagination;
using Nest;

namespace EST.BL.Services;

1 usage  Andrii Pasternak *
public class ElasticSearchService(IElasticClient client, ICurrencyConversionService currencyConversionService)
    : IElasticSearchService
{
```

Метод BulkIndexExpensesAsync

```
◇ 0+1 usages  Andrii Pasternak
public async Task BulkIndexExpensesAsync(List<Expense> expenses, CancellationToken token = default)
{
    var bulkDescriptor = new BulkDescriptor();

    foreach (var document in expenses.Select(MapToDocument))
    {
        bulkDescriptor.Index<ExpenseDocument>(op :BulkIndexDescriptor<ExpenseDocument> => op
            .Id( document.Id)
            .Document(document)
        );
    }

    var bulkResponse = await client.BulkAsync(bulkDescriptor, token);
    if (!bulkResponse.IsValid)
    {
        throw new Exception("Failed to bulk index expenses");
    }
}
```

Метод DeleteAllElasticExpenses

```
◇ 0+1 usages  Andrii Pasternak *
public async Task DeleteAllElasticExpenses()
{
    var response = await client.DeleteByQueryAsync<ExpenseDocument>(q :DeleteByQueryDescriptor<ExpenseDocument>
        => q.Query(rq :QueryContainerDescriptor<ExpenseDocument> => rq.MatchAll())); // Task<DeleteByQueryResponse>
    if (!response.IsValid)
    {
        throw new Exception($"Failed to delete expenses from Elasticsearch: {response.ServerError?.Error.Reason}");
    }
}
```

Метод CreateExpenseIndexAsync

```
◇ Andrii Pasternak
public async Task CreateExpenseIndexAsync(PaginationExpenseItemsDTO expense, Guid userId,
    CancellationToken token = default)
{
    var document = new ExpenseDocument
    {
        Id = expense.Id,
        Price = expense.Price,
        Date = expense.Date,
        UserId = userId,
        Category = expense.Category,
        Location = expense.Location,
        Items = expense.Items,

        CategoryName = expense.Category.Name,
        LocationName = expense.Location.Name,
        ItemNames = expense.Items // List<ItemDTO>
            .Select(i :ItemDTO => i.Name)
            .Where(name => !string.IsNullOrWhiteSpace(name)) // IEnumerable<string>
            .ToList() // List<string>
    };

    var response = await client.IndexDocumentAsync(document, token);
    if (!response.IsValid)
        throw new Exception("Failed to index expense");
}
```

Метод MapToDocument

```
◇  1 usage  Andrii Pasternak *
private ExpenseDocument MapToDocument(Expense expense)
{
    return new ExpenseDocument
    {
        Id = expense.Id,
        Price = currencyConversionService.Convert( amountInUsd: expense.ItemExpenses.Sum(ie
            => ie.Quantity * ie.Item.Price)),
        Date = expense.Date,
        UserId = expense.UserId,
        Category = new CategoryDTO
        {
            Id = expense.Category.Id,
            Name = expense.Category.Name,
            Description = expense.Category.Description,
            Color = expense.Category.Color,
            Icon = expense.Category.Icon,
            IsDeleted = expense.Category.IsDeleted
        },
        Location = new DropdownLocationDTO
        {
            Id = expense.Location.Id,
            Name = expense.Location.Name,
            Address = expense.Location.Address,
            Latitude = expense.Location.Latitude,
            Longitude = expense.Location.Longitude,
            Save = expense.Location.Save
        },
        Items = expense.ItemExpenses.Select(ie => new ItemDTO
        {
            Id = ie.Item.Id,
            Name = ie.Item.Name,
            Price = ie.Item.Price,
            Rating = ie.Item.Rating,
            Description = ie.Item.Description,
            Vendor = ie.Item.Vendor,
            UnitOfMeasure = ie.Item.UnitOfMeasure,
            IsDeleted = ie.Item.IsDeleted,
            Quantity = ie.Quantity
        }).ToList(),
        CategoryName = expense.Category.Name,
        ItemNames = expense.ItemExpenses // List<ItemExpense>
            .Select(ie => ie.Item.Name)
            .Where(name => !string.IsNullOrEmpty(name)) // IEnumerable<string>
            .ToList(),
        LocationName = expense.Location.Name
    };
}
```

Метод UpdateExpenseIndexAsync

```
0+3 usages  Andrii Pasternak *
public async Task UpdateExpenseIndexAsync(
    PaginationExpenseItemsDTO expense,
    Guid userId,
    CancellationToken token = default)
{
    var document = new ExpenseDocument
    {
        Id = expense.Id,
        Price = expense.Price,
        Date = expense.Date,
        UserId = userId,
        Items = expense.Items,

        CategoryName = expense.Category?.Name ?? "",
        ItemNames = expense.Items? // List<ItemDTO>
            .Select(ie :ItemDTO => ie.Name)
            .Where(name => !string.IsNullOrEmpty(name)) // IEnumerable<string>
            .ToList() ?? [],
        LocationName = expense.Location?.Name ?? ""
    };
    await client.UpdateAsync<ExpenseDocument>( expense.Id, u :UpdateDescriptor<ExpenseDocument,ExpenseDocument> => u.Doc(document), token);
}
```

Метод DeleteExpenseAsync

```
0+1 usages  Andrii Pasternak
public async Task DeleteExpenseAsync(Guid expenseId)
{
    var response = await client.DeleteAsync<ExpenseDocument>( expenseId);
    if (!response.IsValid)
        throw new Exception("Failed to delete expense from index");
}
```

Метод SearchExpenseAsync

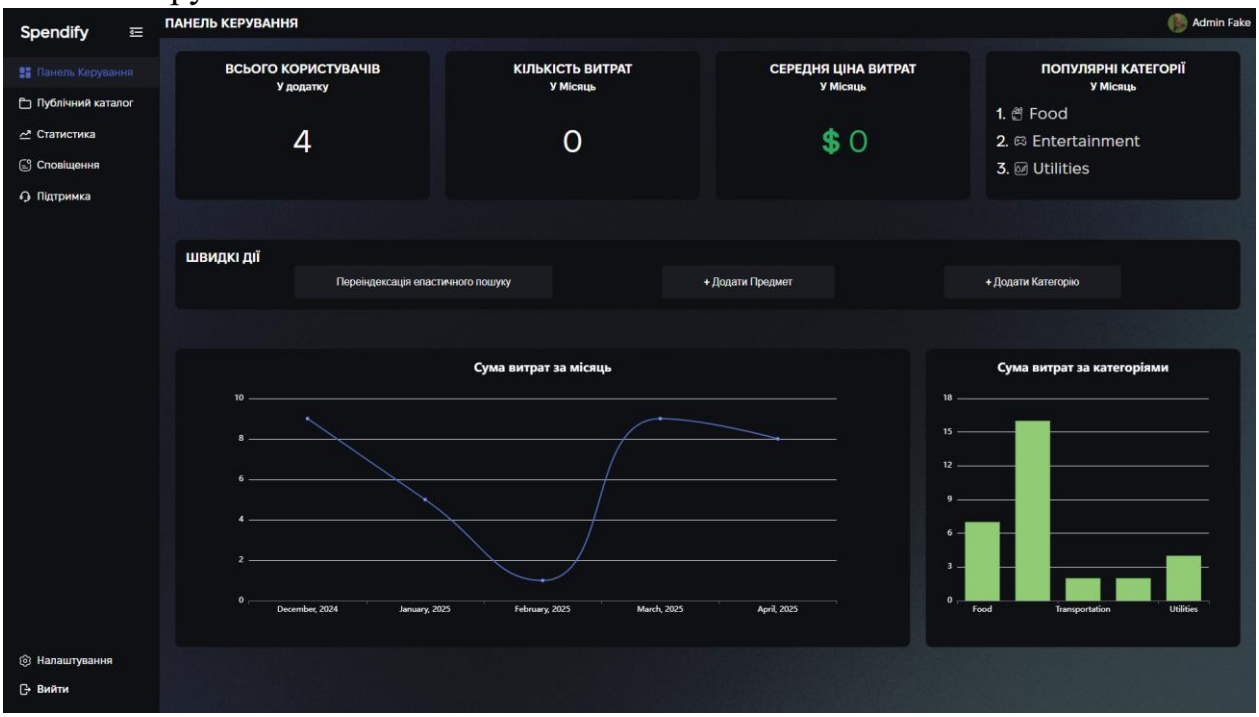
```
0+1 usages  Andrii Pasternak *
public async Task<ElasticSearchResponse> SearchExpensesAsync(
    Guid userId,
    PaginationFilter filter,
    CancellationToken token)
{
    var searchDescriptor = new SearchDescriptor<ExpenseDocument>()
        .Index( "expenses")
        .Query(q :QueryContainerDescriptor<ExpenseDocument> => BuildBaseQuery(userId,
            request: filter.SearchQuery, filter.Filter));

    if (string.IsNullOrWhiteSpace(filter.SortColumn))
    {
        searchDescriptor = searchDescriptor.Sort(s :SortDescriptor<ExpenseDocument> => s.Field(f :ExpenseDocument
            => f.Date, SortOrder.Descending)); // SearchDescriptor<ExpenseDocument>
    }
    else
    {
        var sortField = filter.SortColumn.ToLower() switch
        {
            "date" => Infer.Field<ExpenseDocument>( path: f :ExpenseDocument => f.Date),
            _ => Infer.Field<ExpenseDocument>( path: f :ExpenseDocument => f.Date)
        };

        var order = (!string.IsNullOrWhiteSpace(filter.SortDirection) && filter.SortDirection.ToLower() == "asc")
            ? SortOrder.Ascending
            : SortOrder.Descending;

        searchDescriptor = searchDescriptor.Sort(s :SortDescriptor<ExpenseDocument> => s.Field(sortField, order));
    }

    searchDescriptor = searchDescriptor
        .Skip(filter.PageNumber * filter.PageSize)
        .Take(filter.PageSize)
        .TrackTotalHits(); // SearchDescriptor<ExpenseDocument>

    var response = await client.SearchAsync<ExpenseDocument>(searchDescriptor, token);

    if (!response.IsValid)
    {
        throw new Exception("Failed to retrieve expenses from Elasticsearch.");
    }

    var expenses :List<PaginationExpenseItemsDTO> = response.Documents.Select(doc :ExpenseDocument
        => new PaginationExpenseItemsDTO
    {
        Id = doc.Id,
        Price = doc.Price,
        Date = doc.Date,
        Category = doc.Category,
        Location = doc.Location,
        Items = doc.Items
    }).ToList();

    return new ElasticSearchResponse
    {
        Expenses = expenses,
        TotalCount = response.Total,
        PageSize = filter.PageSize,
        PageNumber = filter.PageNumber
    };
}
```

Метод BuildBaseQuery

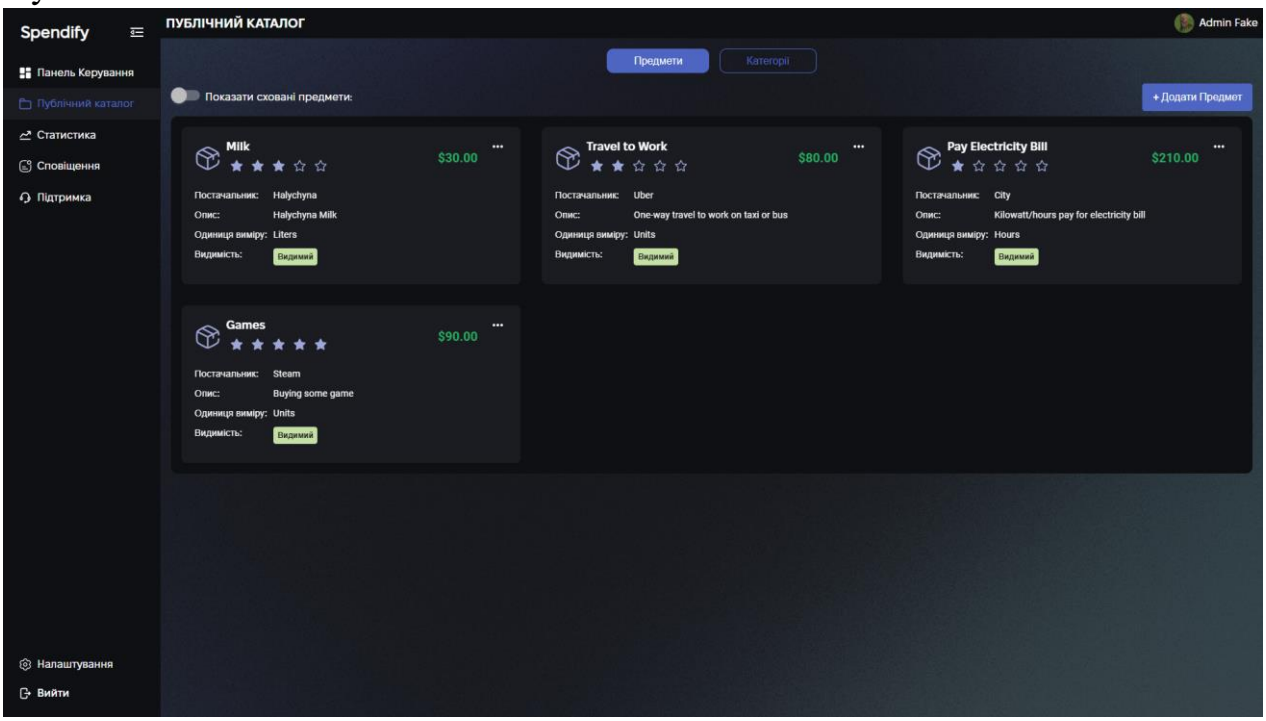
```
1 usage  Andrii Pasternak *
private QueryContainer BuildBaseQuery(Guid userId, string request, string filter)
{
    QueryContainer userFilter = new TermQuery
    {
        Field = Infer.Field<ExpenseDocument>( path: f :ExpenseDocument => f.UserId.Suffix("keyword")),
        Value = userId
    };

    QueryContainer dateFilter = null;
    var now :DateTime = DateTime.UtcNow;
    switch (filter)
    {
        case "week":
        {
            var startOfWeek :DateTime = now.Date.AddDays(-(int)now.DayOfWeek + (int)DayOfWeek.Monday);
            var endOfWeek :DateTime = startOfWeek.AddDays(6).AddHours(23).AddMinutes(59).AddSeconds(59);
            dateFilter = new DateRangeQuery
            {
                Field = Infer.Field<ExpenseDocument>( path: f :ExpenseDocument => f.Date),
                GreaterThanOrEqualTo = startOfWeek,
                LessThanOrEqualTo = endOfWeek
            };
            break;
        }
        case "month":
        {
            var startOfMonth = new DateTime(now.Year, now.Month, day: 1);
            var endOfMonth :DateTime = startOfMonth.AddMonths(1).AddSeconds(-1);
            dateFilter = new DateRangeQuery
            {
                Field = Infer.Field<ExpenseDocument>( path: f :ExpenseDocument => f.Date),
                GreaterThanOrEqualTo = startOfMonth,
                LessThanOrEqualTo = endOfMonth
            };
            break;
        }
        case "threeMonths":
        {
            var startOfMonth = new DateTime(now.Year, now.Month, day: 1);
            var firstOfThreeMonths :DateTime = startOfMonth.AddMonths(-2);
            var endOfMonth :DateTime = startOfMonth.AddMonths(1).AddSeconds(-1);
            dateFilter = new DateRangeQuery
            {
                Field = Infer.Field<ExpenseDocument>( path: f :ExpenseDocument => f.Date),
                GreaterThanOrEqualTo = firstOfThreeMonths,
                LessThanOrEqualTo = endOfMonth
            };
            break;
        }
    }

    QueryContainer textQuery = null;
    if (!string.IsNullOrWhiteSpace(request))
    {
        textQuery = new MultiMatchQuery
        {
            Query = request,
            Fields = Infer.Fields<ExpenseDocument>(
                f :ExpenseDocument => f.CategoryName,
                f :ExpenseDocument => f.ItemNames,
                f :ExpenseDocument => f.LocationName),
            Fuzziness = Fuzziness.Auto,
        };
    }

    var mustList = new List<QueryContainer> { userFilter };
    if (textQuery != null)
    {
        mustList.Add(textQuery);
    }

    var boolQuery = new BoolQuery
    {
        Must = mustList,
        Filter = dateFilter != null ? new List<QueryContainer> { dateFilter } : null
    };

    return boolQuery;
}
```

ДОДАТОК Б

Панель Керування

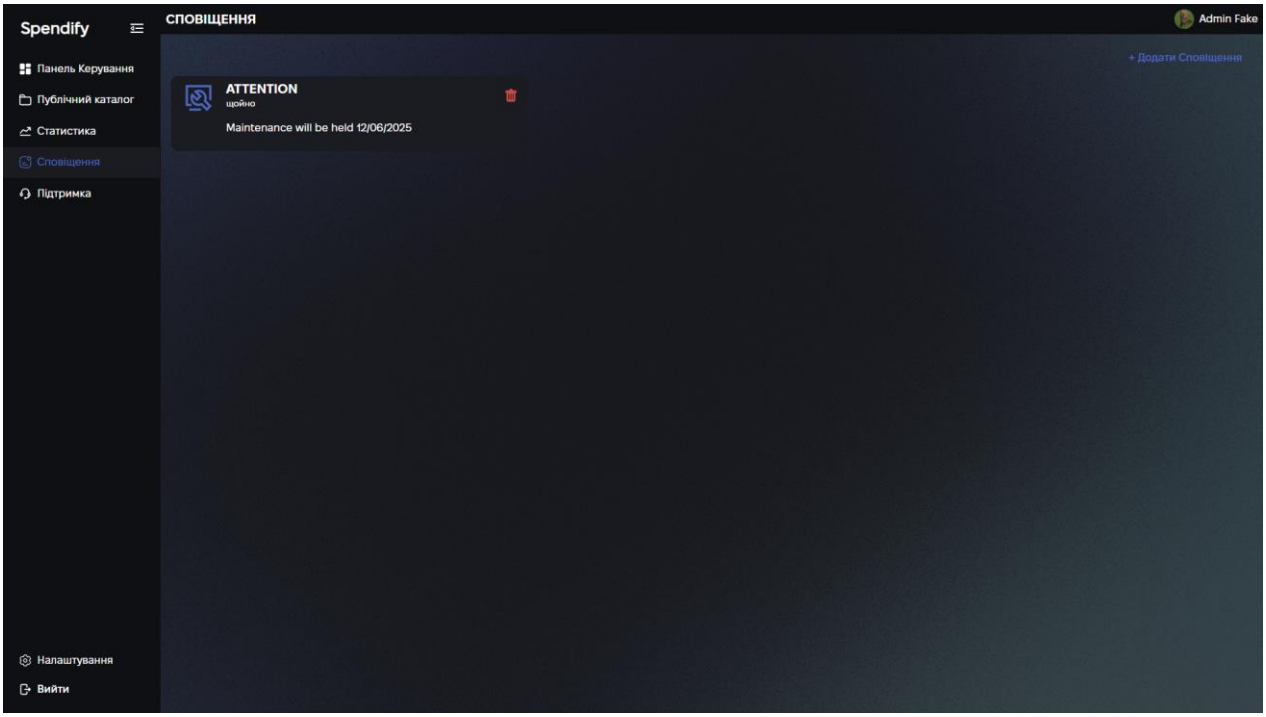


Публічний Каталог

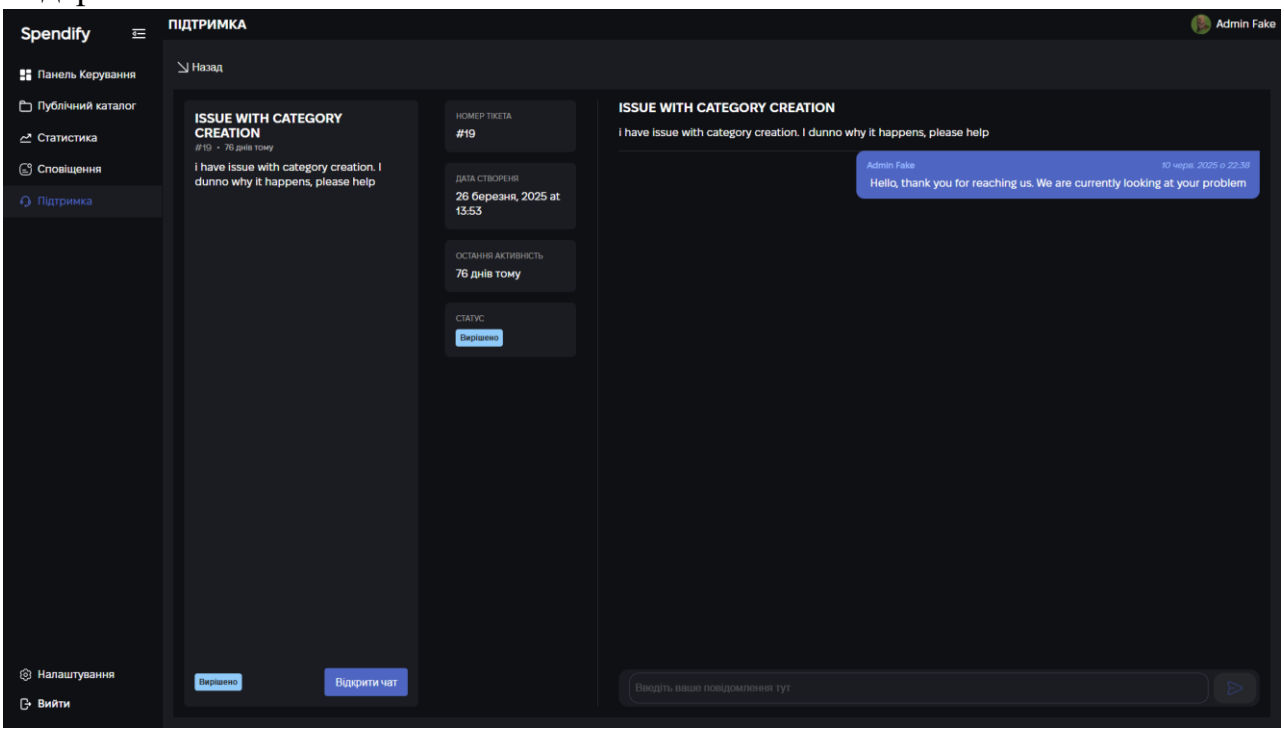


Статистика

Сповіщення



Підтримка



Налаштування