МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ВЕБСЕРВІСУ ДЛЯ УПРАВЛІННЯ ФІНАНСАМИ НА ОСНОВІ REACT»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
В.Ю. Саранча
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Кривохата А.Г.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Саранчі Владиславу Юрійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебсервісу для управління фінансами на основі React

керівник роботи Кривохата Анастасія Григорівна, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 05.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Основні теоретичні відомості.
2. Проєктування вебсервісу.
3. Реалізація та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

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

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	07.04.2025	
2.	Збір вихідних даних.	09.05.2025	
3.	Обробка методичних та теоретичних	23.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	08.09.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	19.12.2025	

Студент _____ В.Ю. Саранча
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.Г. Кривохата
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка вебсервісу для управління фінансами на основі React»: 54 с., 32 рис., 1 табл., 11 джерел.

ВЕБСЕРВІС, БАЗА ДАНИХ, API, PYTHON, REACT, SPENDSPLIF, TYPESCRIPT.

Об’єктом роботи є процес організації та автоматизації фінансового обліку в онлайн-середовищі.

Предметом виступають методи, підходи та інструменти створення вебсервісу для ведення доходів, витрат, категорій, сховищ, фінансових цілей і аналітики з використанням TypeScript, React, серверної архітектури та бази даних PostgreSQL.

Мета роботи полягає у створенні вебсервісу, який забезпечує користувачу зручне середовище для обліку фінансових операцій, формування аналітики, роботи з кількома валютами та інтеграції банківських даних.

У процесі підготовки застосовано методи аналізу вимог, моделювання архітектури, проєктування інтерфейсу, застосування REST-підходу, побудову структури бази даних, а також тестування клієнтської й серверної частин.

У ході реалізації сформовано вимоги до функціоналу сервісу, спроєктовано його архітектуру, побудовано клієнтську частину на React та серверну частину з використанням TypeScript, налаштовано взаємодію з базою даних і реалізовано основні модулі системи: облік транзакцій, категорій, сховищ, аналітики, цілей і командної роботи. Проведено тестування ключових сценаріїв для перевірки коректності роботи сервісу.

У результаті створено вебсервіс, який дає змогу ефективно управляти фінансовими потоками, аналізувати витрати та доходи, формувати звіти, працювати в команді й інтегруватися з банківськими API. Розроблений підхід забезпечує надійну основу для подальшого розширення функціоналу та впровадження сервісу в індивідуальну або підприємницьку практику.

SUMMARY

Master’s qualifying paper “Development of a Web Service for Financial Management Using React”: 54 pages, 32 figures, 1 table, 11 references.

API, DATABASE, PYTHON, REACT, SPENDSPLIF, TYPESCRIPT, WEBSERVICE.

The object of the work is the process of organizing and automating financial accounting in an online environment.

The subject of the work includes methods, approaches, and tools for creating a web service for recording income, expenses, categories, accounts, financial goals, and analytics using TypeScript, React, server architecture, and a PostgreSQL database.

The aim of the work is to create a web service that provides users with a convenient environment for managing financial operations, generating analytics, working with multiple currencies, and integrating banking data.

The study applies methods of requirements analysis, architecture modeling, interface design, the REST approach, database structure development, as well as testing of the client-side and server-side components.

During the implementation, the functional requirements of the service were defined, its architecture was designed, the client side was developed using React, and the server side was implemented using TypeScript. Interaction with the database was configured, and the main system modules were developed: transaction management, categories, accounts, analytics, financial goals, and team collaboration. Testing of key scenarios was conducted to verify correctness and system performance.

As a result, a web service was created that enables effective financial flow management, analysis of expenses and income, report generation, team collaboration, and integration with banking APIs. The developed approach provides a solid foundation for further functional expansion and practical use in individual or entrepreneurial financial management.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 7
1 Огляд теоретичних властивостей та планування проєкту 9
1.1 TypeScript та його можливості..... 9
1.2 React та його можливості 10
1.3 Сучасні підходи до розробки вебсервісів..... 11
1.4 Функціональні та нефункціональні вимоги 15
1.5 Висновки до розділу 1 17
2 Проєктування вебсервісу..... 19
2.1 Опис предметної області 19
2.2 Вимоги до архітектури вебсервісу 22
2.3 Проєктування клієнтської частини на React 24
2.4 Проєктування серверної частини 26
2.5 Проєктування бази даних..... 28
2.6 Висновки до розділу 2 33
3 Реалізація та тестування 34
3.1 Реалізація клієнтської частини React 34
3.2 Реалізація серверної частини 38
3.3 Налаштування бази даних (PostgreSQL)..... 40
3.4 Інтеграція API та зовнішніх сервісів..... 42
3.5 Тестування функціоналу вебсервісу 43
3.6 Висновки до розділу 3 51
Висновки 53
Перелік посилань..... 54

ВСТУП

У сучасній цифровій економіці управління фінансами перетворюється на окремий напрям, що потребує швидких рішень, прозорих інструментів та зручного доступу до аналітики. Підприємці, фрилансери та невеликі команди дедалі частіше стикаються з потребою систематизувати інформацію про доходи й витрати, визначати фінансові цілі, прогнозувати рух коштів і працювати з даними, що постійно змінюються. Класичні інструменти, як-от електронні таблиці або записники, уже не справляються з обсягами інформації і не гарантують ні швидкості опрацювання даних, ні їхньої точності. Через це формується запит на вебсервіси, здатні поєднувати автоматизацію, наочність і гнучкість.

Зростання популярності односторінкових застосунків, розвиток типізованих мов програмування та перехід до інтерактивних інтерфейсів створили підґрунтя для появи нових фінансових платформ. TypeScript і React стали одними з інструментів, які дозволяють розробникам створювати масштабовані, стабільні та зручні вебзастосунки. Використання цих технологій дозволяє формувати передбачувану логіку роботи інтерфейсу, будувати модульні структури та забезпечувати високу швидкість взаємодії з даними.

Актуальність теми полягає у необхідності створення доступних і водночас технологічно досконалих рішень для управління фінансовими потоками. Сучасний бізнес працює у швидкому темпі, тому інструменти обліку повинні не лише відображати інформацію, а й допомагати ухвалювати обґрунтовані рішення на основі аналітики, прогнозування та автоматичного структурування даних. Вебсервіс SpendSplif орієнтований саме на це – він узагальнює фінансові показники, дозволяє додавати транзакції, формувати категорії, контролювати сховища, перевіряти прогрес фінансових цілей та спільно працювати над бюджетами в команді.

Метою роботи є розроблення вебсервісу для управління фінансами на основі React.

Для досягнення мети визначено такі завдання:

– розкрити предметну область та обґрунтувати вибір технологій;
– проаналізувати можливості TypeScript та React у контексті створення фінансового вебсервісу;
– сформувати функціональні та нефункціональні вимоги;
– спроєктувати архітектуру сервісу, моделі даних і структуру клієнтської та серверної частин;
– реалізувати основний функціонал вебсервісу;
– виконати тестування роботи системи й оцінити її відповідність вимогам.

Об’єктом роботи є процес створення вебсервісу для ведення фінансових операцій і аналітики. Предметом є методи, підходи й інструменти побудови клієнтської та серверної частин фінансового застосунку з використанням TypeScript, React та сучасних фреймворків бекенда.

Робота спирається на сучасні практики проєктування, що забезпечують масштабованість сервісу, передбачуваність коду та зручність майбутнього супроводу.

Результати роботи можуть бути основою для подальшого розширення системи і впровадження нових модулів.

Структура роботи включає вступ, три розділи, висновки та перелік використаних джерел. У першому розділі подано теоретичні засади та сформовано вимоги до вебсервісу. У другому розглянуто проєктування архітектури, клієнтської та серверної частин, а також бази даних. Третій розділ присвячено реалізації сервісу, інтеграції API та тестуванню. Завершує роботу розділ з узагальненими висновками.

1 ОГЛЯД ТЕОРЕТИЧНИХ ВЛАСТИВОСТЕЙ ТА ПЛАНУВАННЯ ПРОЄКТУ

1.1 TypeScript та його можливості

TypeScript посідає особливе місце серед сучасних інструментів фронтенд-розробки, оскільки поєднує гнучкість JavaScript із передбачуваністю статичної типізації. У розробці фінансових сервісів це має істотне значення: під час опрацювання транзакцій, підрахунку сум, роботи з валютами та взаємодії з API будь-яка дрібна помилка здатна призвести до некоректних розрахунків або збоїв у програмній логіці. Використання TypeScript дозволяє уникнути таких ситуацій завдяки суворішому контролю типів і зрозумілій структуризації даних [1].

Одна з ключових можливостей TypeScript – створення інтерфейсів та типів для даних, що циркулюють у застосунку. Наприклад, опис структури транзакції, категорії чи фінансового сховища забезпечує однаковий формат опрацювання на клієнтській і серверній частинах. У результаті зменшується кількість помилок, пов’язаних із неправильними назвами полів або різною структурою об’єктів. Це особливо корисно для великих команд, де важливо, щоб усі учасники взаємодіяли зі зрозумілою та стандартизованою моделлю даних [2].

TypeScript також полегшує рефакторинг. За умов постійного розширення функціоналу – додавання нових валют, нових типів аналітики, інтеграцій із банківськими сервісами – структура коду часто змінюється. Статична типізація дозволяє безпечно редагувати логіку й швидко знаходити ділянки, де зміна однієї функції вплинула на інші частини застосунку. Це пришвидшує розвиток сервісу й допомагає утримувати код у впорядкованому стані [4].

Ще одна важлива перевага – підтримка сучасних можливостей ECMAScript. TypeScript компілюється у звичайний JavaScript, тому дозволяє

використовувати останні нововведення мови без ризику втратити сумісність із браузерами. Завдяки цьому полегшується впровадження нових підходів у розробці, таких як модульність, асинхронна обробка даних, робота з промісами або використання синтаксису async/await. Для вебсервісів, що активно працюють із зовнішніми API, це є суттєвою перевагою [2, 5].

Варто також виділити потужні інструменти для розробки. Інтеграція TypeScript із сучасними редакторами коду забезпечує підказки, автоматичне завершення, перевірку типів у режимі реального часу та швидкий пошук помилок. Це робить процес розробки комфортнішим і дозволяє зосередитися на логіці застосунку, а не на дрібних технічних нюансах [6].

Для фінансового сервісу, де важливі точність, стабільність і безпека, TypeScript виступає інструментом, що забезпечує передбачувану роботу застосунку, підтримує масштабування та допомагає уникати критичних помилок у процесі обробки фінансових даних. Він створює надійну основу, на якій будується клієнтська частина вебсервісу та формується стабільний і зручний користувацький досвід [1, 3].

1.2 React та його можливості

React посідає провідне місце серед інструментів для створення сучасних інтерфейсів, оскільки дозволяє будувати вебзастосунки, які швидко реагують на дії користувача та залишаються легкими у підтримці. Основою React є компонентний підхід, за якого інтерфейс формується з незалежних елементів, кожен із яких відповідає за окрему частину логіки та відображення. Такий підхід дозволяє структурувати фінансовий вебсервіс у вигляді модулів, зокрема панелі аналітики, списку транзакцій, форм додавання операцій та віджетів балансу. Це спрощує розвиток застосунку та зменшує ризик порушення роботи інших компонентів під час розширення функціоналу [4, 7].

Однією з ключових технічних особливостей React є використання віртуального DOM. Цей механізм дозволяє оптимізувати оновлення

інтерфейсу шляхом перерендерингу лише тих елементів, у яких відбулися зміни. Для фінансових сервісів це має особливе значення, оскільки дані про транзакції, баланси та аналітичні показники можуть змінюватися в режимі реального часу. Завдяки віртуальному DOM забезпечується висока продуктивність і плавність роботи інтерфейсу навіть за значного обсягу фінансових даних [1].

React підтримує гнучкі механізми керування станом застосунку. У межах окремих компонентів може використовуватися локальний стан, тоді як для складніших сценаріїв доступні контекстні механізми або зовнішні бібліотеки. Це дозволяє синхронізувати фінансові дані між різними частинами інтерфейсу. Наприклад, зміна балансу рахунку автоматично відображається у таблиці транзакцій, аналітичних графіках і зведеній панелі без необхідності ручного оновлення сторінки [4, 7].

Важливою складовою React є підтримка хуків, які дозволяють інкапсулювати логіку роботи з даними у вигляді повторно використовуваних функцій. Завдяки цьому операції завантаження даних з API, фільтрації транзакцій, сортування або обробки введення можуть бути винесені в окремі модулі та застосовані в різних компонентах. Для фінансового вебсервісу це спрощує роботу з зовнішніми сервісами, зокрема API валютних курсів, та підвищує читабельність і підтримуваність коду [7].

Окремої уваги заслуговує розвинена екосистема React. Велика кількість бібліотек і готових компонентів дозволяє швидко реалізувати графіки, таблиці, маршрутизацію сторінок, адаптивний дизайн та інші елементи інтерфейсу, що скорочує час розробки сервісу та дозволяє зосередитися на реалізації бізнес-логіки, а не базових елементів інтерфейсу [4].

1.3 Сучасні підходи до розробки вебсервісів

Розвиток вебтехнологій привів до появи нових підходів, які суттєво змінили спосіб створення та використання онлайн-платформ. Вебсервіси

перестали бути простими сторінками з обмеженим набором функцій – сьогодні вони нагадують повноцінні багаторівневі системи, здатні опрацьовувати великі обсяги даних, взаємодіяти з зовнішніми API, працювати з декількома видами клієнтів і забезпечувати високу швидкість відгуку. У сфері фінансових сервісів ці зміни відчуваються особливо: користувачі очікують миттєвої реакції інтерфейсу, стабільності та повної безпеки своїх даних. У результаті сучасні підходи до розробки вебзастосунків акцентують увагу на компонентності, масштабованості, модульності та уніфікації взаємодії між частинами системи [1].

Одним із ключових напрямів є перехід до архітектури SPA (Single Page Application). На відміну від класичних багатосторінкових застосунків, SPA завантажує основні ресурси один раз, а подальша робота відбувається за рахунок оновлення частин сторінки без повного перезавантаження. Це робить інтерфейс значно швидшим і плавнішим, що дуже важливо для сервісів, де користувач активно перемикається між розділами: переглядає баланси, фільтрує транзакції, аналізує графіки або додає нові операції. SPA також покращує мобільний досвід, оскільки забезпечує роботу на пристроях із повільнішим інтернетом і обмеженою продуктивністю [4].

Важливим підходом є компонентно-орієнтована розробка, що передбачає розбиття інтерфейсу на незалежні частини. Кожен компонент відповідає за свою функцію та має власну логіку, а зміни в одному місці не впливають на інші модулі. Для фінансових сервісів це особливо зручно: панелі з аналітикою, списки витрат, форми додавання операцій або елементи фільтрації стають окремими компонентами, які можна перевикористовувати, розширювати або замінювати без ризику порушити загальну структуру застосунку. Компонентний підхід також спрощує командну роботу, оскільки кожен розробник може зосередитися на своїй частині інтерфейсу [7].

Суттєвого поширення набув підхід API-first. Він передбачає, що в основі будь-якого сервісу лежить чітка, стандартизована API-специфікація, яка визначає спосіб взаємодії клієнтської частини із сервером. Надалі до цього API можуть підключатися інші клієнти, зокрема мобільні застосунки або сторонні

сервіси. API-first підхід забезпечує стабільність взаємодії та дозволяє змінювати серверну реалізацію без впливу на клієнтський інтерфейс, що є важливим для фінансових систем із великою кількістю інтеграцій [9].

У контексті сучасної розробки широко використовується архітектурний стиль REST. REST-орієнтовані API забезпечують логічний і зрозумілий обмін даними через стандартні HTTP-методи. Для фінансових сервісів це дозволяє уніфікувати операції отримання транзакцій, додавання нових записів, редагування категорій і роботи з балансами. REST спрощує виконання CRUD-операцій та забезпечує сумісність сервісу з різними клієнтськими платформами [1].

Окрему увагу варто приділити підходу modular design, який передбачає побудову системи з незалежних функціональних блоків. У такій архітектурі клієнтська частина, сервер, база даних, модуль аналітики або система авторизації можуть розвиватися автономно. Це особливо важливо для фінансових вебсервісів, які постійно розширюють функціонал: додають нові валюти, аналітичні звіти або інтеграції з банківськими системами [3].

У сучасній розробці активно застосовуються підходи DevOps та CI/CD, які забезпечують автоматизацію тестування, перевірки та розгортання застосунків. Для фінансових платформ це має критичне значення, оскільки оновлення повинні впроваджуватися без зупинки роботи сервісу, а кожна зміна проходити автоматичну перевірку. CI/CD дозволяє підтримувати стабільність системи та мінімізувати ризик помилок у продакшн-середовищі [2].

Ще одним важливим напрямом є інтеграція зовнішніх сервісів. У фінансових платформах це стосується підключення банківських API, сервісів конвертації валют або генерації звітів. Використання стандартів документування API спрощує інтеграцію нових модулів і забезпечує єдину структуру взаємодії між компонентами системи [9, 11].

Суттєвого поширення набули також підходи до оптимізації продуктивності, зокрема використання lazy loading, кешування та мемоізації.

Завантаження лише необхідних компонентів інтерфейсу дозволяє зменшити навантаження на систему та підвищити швидкість роботи вебсервісу. Для фінансових застосунків із великою кількістю статистики та графіків це є особливо актуальним [4].

У сучасних вебсервісах широко застосовується підхід Responsive Design, який гарантує коректне відображення інтерфейсу на різних пристроях. Це дозволяє користувачам працювати з фінансовим сервісом як з настільних комп’ютерів, так і з мобільних пристроїв, не втрачаючи зручності взаємодії [7].

У сукупності всі розглянуті підходи формують сучасну модель розробки вебсервісів, яка забезпечує масштабованість, стабільність і високу продуктивність. Для фінансового вебсервісу, де ключовими є точність, безпека та швидке реагування, застосування сучасних підходів є необхідною умовою якісної реалізації платформи [1, 2].

Усе більше значення отримує й підхід до складання зручного UI/UX. Інтерфейс має бути логічним, структурованим та простим: користувач повинен одразу розуміти, де знаходяться потрібні функції, як працювати з балансами та як фільтрувати транзакції.

Сучасні практики рекомендують дотримуватися мінімалізму, використовувати зрозумілі візуальні акценти та забезпечувати доступність інтерфейсу для всіх користувачів.

У сукупності всі ці підходи формують сучасний погляд на розробку вебсервісів. Вони забезпечують ефективність, масштабованість, надійність і високу швидкість роботи, а також створюють середовище, у якому вебзастосунки можуть розширюватися разом із потребами користувачів. Для фінансового вебсервісу, де важливі точність, безпека та швидке реагування, використання сучасних підходів є ключовою умовою якісної реалізації платформи.

1.4 Функціональні та нефункціональні вимоги

Під час розроблення вебсервісу управління фінансами важливо визначити, які саме можливості повинні бути доступні користувачу і які характеристики має забезпечувати система на технічному рівні. Чітке формулювання вимог дозволяє структурувати майбутній функціонал, оцінити складність проєкту, побудувати логічну архітектуру та уникнути непорозумінь під час подальшої реалізації. У сфері фінансових застосунків це особливо важливо, адже навіть незначні помилки або невраховані сценарії можуть вплинути на точність обрахунків, безпеку чи роботу з чутливими даними.

Функціональні вимоги описують конкретні дії, які вебсервіс має виконувати. Це стосується роботи з транзакціями, категоріями, сховищами, балансами, аналітикою, експортом даних і взаємодією з API. Усі ці процеси визначають основний набір можливостей, який формує користувацький досвід. Для фінансової системи ключовим є надання максимально прозорого обліку операцій та інструментів аналізу, що дозволяють користувачу оцінювати власні витрати, відслідковувати динаміку доходів і планувати фінансові цілі.

Нефункціональні вимоги, у свою чергу, описують властивості системи, що не пов’язані з безпосередніми діями, але визначають її якість: продуктивність, масштабованість, безпеку, зручність використання, надійність і доступність. Для будь-якого вебсервісу важливо забезпечити швидку обробку даних, стабільний інтерфейс і захист персональної інформації. У фінансовому застосунку ці вимоги набувають ще більшої ваги, оскільки йдеться про конфіденційні дані та взаємодію з банківськими сервісами.

Щоб забезпечити системність підходу та логічне структурування вимог, доцільно узагальнити їх у вигляді таблиці. Це створює наочну картину того, які можливості повинні бути реалізовані та які параметри визначають їх якість. Основні вимоги до вебсервісу SpendSplif наведено в таблиці 1.1.

Таблиця 1.1 – Функціональні та нефункціональні вимоги

Тип вимог	**Найважливіші пункти**	**Характеристика**
Функціональні	Облік доходів і витрат	Додавання, редагування, видалення транзакцій; прив’язка до категорій, коментарів і сховищ.
	Аналітика та графіки	Перегляд статистики за періодами, порівняння місяців, частки витрат, динаміка змін.
	Сховища та баланси	Підтримка кількох рахунків, автоматичне оновлення залишків, перерахунок валюти.
	Категорії операцій	Створення власних категорій, вибір зі стандартних шаблонів.
	Підключення банківського API	Автоматичне завантаження транзакцій, синхронізація історії операцій.
	Командна робота	Ролі, рівні доступу, спільне управління фінансами.
	Експорт та імпорт даних	Вивантаження в Excel, формування pdf-звітів, імпорт із файлів.
Нефункціональні	Безпека та конфіденційність	Шифрування, захист каналів передачі, безпечна авторизація (JWT/OAuth).
	Продуктивність	Швидке завантаження інтерфейсу, оптимізовані запити, кешування даних.
	Масштабованість	Можливість розширення модулів, підтримка великої кількості користувачів.
	Доступність	Коректна робота на різних пристроях, адаптивний дизайн, підтримка кількох мов.
	Надійність	Стійкість до помилок, стабільність при великих обсягах операцій.
	Юзабіліті	Інтуїтивний інтерфейс, зрозуміла навігація, мінімалістична структура.

Особливої уваги у фінансовому вебсервісі потребують вимоги безпеки. Шифрування даних, захист токенів доступу, правильна обробка помилок і уникнення вразливостей є необхідними умовами для роботи з чутливими даними. Будь-яка взаємодія з банківськими API потребує автентифікації, а всі запити повинні виконуватися через безпечні канали, що відповідають європейським стандартам захисту інформації.

Не менш важливими залишаються вимоги до зручності використання. У фінансовому сервісі користувач постійно працює з великими масивами даних: переглядає списки операцій, аналізує графіки, сортує транзакції за періодами

або категоріями. Тому інтерфейс має бути простим, логічним та продуманим. Застосування мінімалістичного дизайну, візуальних акцентів і адаптивної верстки забезпечує комфортну взаємодію з сервісом навіть на невеликих екранах.

Окремо варто відзначити значення продуктивності. Сервіс управління фінансами часто опрацьовує сотні або тисячі операцій, і навіть невелика затримка може знизити зручність користування. Для цього використовуються оптимізовані алгоритми фільтрації, кешування даних, lazy loading та оновлення компонентів лише за необхідності.

Важливе місце посідає й масштабованість. З часом кількість функцій може збільшуватися: з’являються нові типи звітів, розширені аналітичні інструменти, інтеграції з різними банками або більш глибокі механізми командної взаємодії. Якщо архітектура сервісу побудована правильно, кожен новий модуль може бути впроваджений без ризику порушити роботу вже існуючих частин. Це забезпечує довговічність проєкту та можливість подальшого розвитку.

Сформульовані функціональні та нефункціональні вимоги відіграють ключову роль у проєктуванні вебсервісу SpendSplif. Вони дозволяють перейти від загального бачення платформи до конкретного технічного плану, визначити майбутні модулі та спроєктувати архітектуру сервісу відповідно до потреб користувачів і бізнес-логіки. На наступному етапі вимоги стануть основою для побудови структури клієнтської частини, серверної логіки та бази даних.

1.5 Висновки до розділу 1

У першому розділі визначено ключові особливості предметної області та встановлено вимоги до майбутнього вебсервісу управління фінансами. Показано, що сучасні фінансові платформи повинні забезпечувати швидкий

доступ до даних, підтримку категорій, сховищ, аналітики і командної роботи, а також відповідати високим стандартам безпеки.

Розглянуті можливості TypeScript підтверджують доцільність використання статичної типізації для підвищення надійності й передбачуваності роботи застосунку. Аналіз React продемонстрував переваги компонентного підходу для створення гнучкого та продуктивного інтерфейсу, здатного працювати з великими масивами фінансової інформації.

Опис сучасних підходів до розробки вебсервісів окреслив технічні орієнтири – модульність, SPA-архітектуру, REST-взаємодію, адаптивність і застосування CI/CD. Визначені функціональні та нефункціональні вимоги стали основою для подальшого проєктування архітектури системи.

Отже, в даному розділі сформовано теоретичний фундамент вебсервісу SpendSplif та визначено напрямки подальшої роботи, що будуть розкриті у другому розділі.

2 ПРОЄКТУВАННЯ ВЕБСЕРВІСУ

2.1 Опис предметної області

Проєктування вебсервісу управління фінансами потребує чіткого розуміння предметної області, у межах якої функціонує система, а також ролей користувачів і типових сценаріїв їх взаємодії із сервісом. Предметна область даного проєкту охоплює процеси персонального та командного фінансового менеджменту, зокрема облік доходів і витрат, управління рахунками, категоризацію фінансових операцій, формування аналітичних звітів, а також взаємодію з зовнішніми фінансовими сервісами.

Ключовою особливістю предметної області є необхідність підтримки багатокористувацької роботи в межах спільного фінансового простору (Space). Такий простір об’єднує користувачів або учасників команди, надаючи їм доступ до спільних рахунків, категорій, транзакцій і фінансових цілей. При цьому важливою вимогою є гнучке керування ролями та правами доступу, що дозволяє розмежовувати відповідальність між учасниками команди.

Для формалізації вимог до системи та наочного відображення взаємодії користувачів із вебсервісом у процесі проєктування використано діаграму варіантів використання UML (див. рис. 2.1). Дана діаграма дозволяє систематизувати основні функції вебсервісу та визначити, які дії доступні різним типам користувачів.

На діаграмі виділено два основних актори: користувач та учасник команди. Користувач виступає базовим актором системи та має доступ до повного набору функцій, пов’язаних з управлінням власними фінансами та організацією спільної роботи. Учасник команди, у свою чергу, взаємодіє з системою в межах наданих йому прав доступу в конкретному фінансовому просторі.

Однією з базових функцій системи є реєстрація та автентифікація користувачів, що забезпечує ідентифікацію особи та захист персональних

фінансових даних. Після успішного входу користувач отримує можливість створювати або обирати фінансовий простір, у межах якого здійснюється подальша робота з рахунками та операціями.

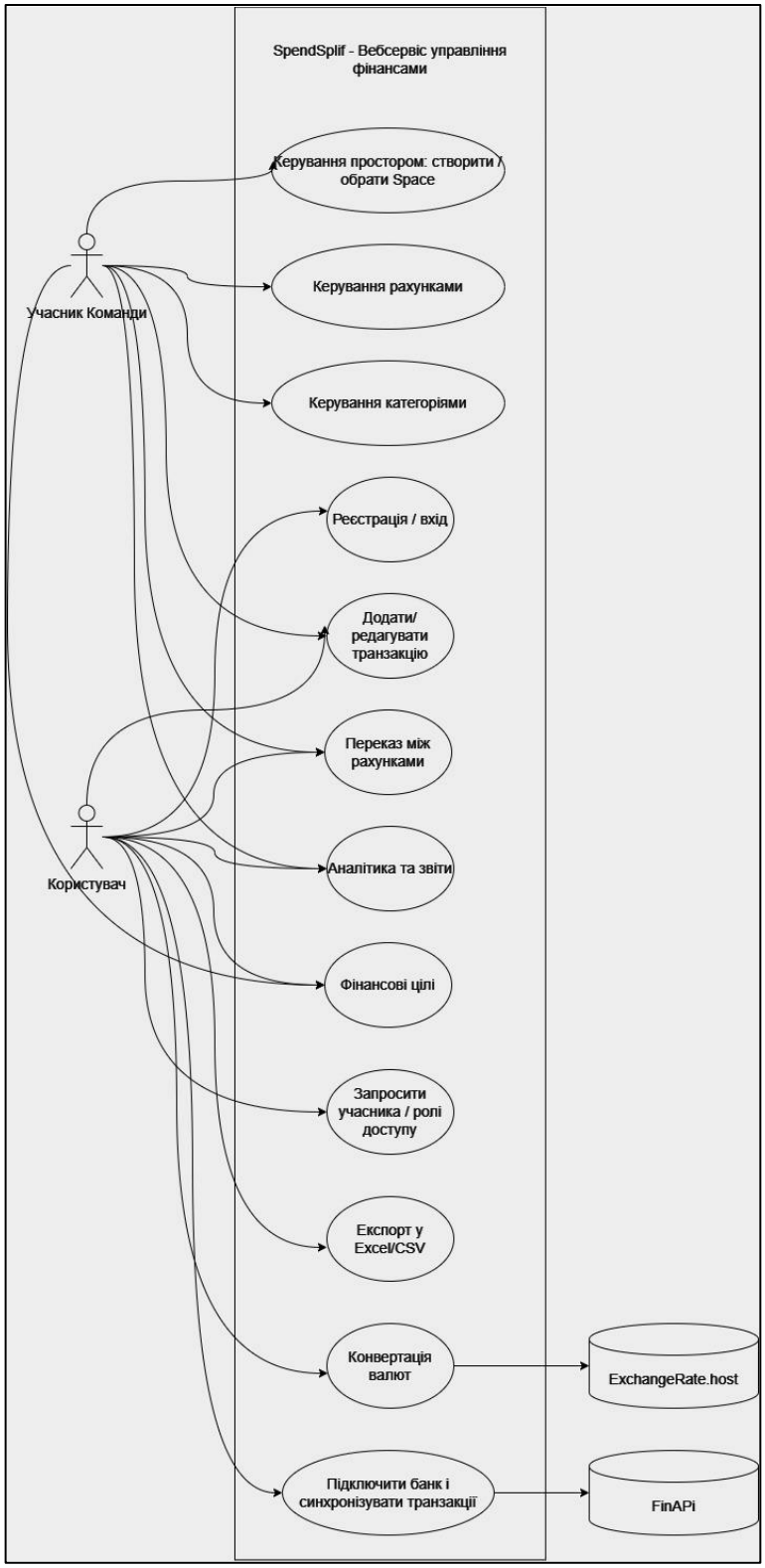


Рисунок 2.1 – Діаграма варіантів використання вебсервісу управління фінансами SpendSplif

Важливим функціональним блоком є керування рахунками, що включає створення, редагування та перегляд фінансових рахунків з підтримкою різних валют. У рамках предметної області рахунок розглядається як базова сутність, до якої прив’язуються фінансові операції та з якої формується загальний баланс користувача або команди.

Не менш значущою є функція керування категоріями, яка дозволяє структурувати доходи та витрати за логічними групами. Категоризація фінансових операцій є основою для подальшого формування аналітики, побудови звітів і візуалізації фінансових показників.

Центральним елементом взаємодії користувача із системою виступає додавання та редагування транзакцій. У межах цього сценарію користувач може фіксувати витрати та доходи, обирати відповідний рахунок і категорію, додавати коментарі та коригувати вже існуючі записи. Окремо передбачено сценарій переказу коштів між рахунками, що дозволяє коректно відображати внутрішні фінансові переміщення без спотворення загальної статистики.

Для підтримки процесу фінансового планування система передбачає функціонал формування фінансових цілей, які дають змогу встановлювати заплановані суми та відстежувати прогрес їх досягнення. Даний механізм є важливим інструментом підвищення фінансової дисципліни користувачів.

Окрему групу сценаріїв становлять аналітика та звітність, що забезпечують користувача узагальненою інформацією про фінансову діяльність за певний період. У рамках цих сценаріїв реалізовано побудову звітів, візуалізацію динаміки витрат і доходів, а також експорт даних у формати Excel та CSV для подальшого аналізу в зовнішніх інструментах.

Суттєвим розширенням функціональних можливостей системи є інтеграція з зовнішніми сервісами. Зокрема, використання сервісу ExchangeRate.host забезпечує конвертацію валют, що є критично важливим у разі роботи з мультивалютними рахунками. Крім того, передбачено підключення банківських установ через FinApi та механізми Open Banking, що дозволяє автоматично синхронізувати транзакції з реальними банківськими рахунками користувачів.

Таким чином, аналіз предметної області та побудова діаграми варіантів використання дозволили чітко визначити функціональні вимоги до вебсервісу управління фінансами. Отримані результати стали основою для подальшого архітектурного проєктування системи та розробки її клієнтської й серверної частин.

2.2 Вимоги до архітектури вебсервісу

Архітектура вебсервісу управління фінансами повинна забезпечувати надійність, масштабованість, безпеку обробки даних та зручність подальшого розвитку функціоналу. З огляду на специфіку предметної області, система має підтримувати багатокористувацьку роботу, обробку фінансових операцій у реальному часі, інтеграцію із зовнішніми сервісами та роботу з кількома типами сховищ даних. Тому в межах проєктування обрано клієнт-серверну архітектуру з чітким розділенням відповідальностей між компонентами.

Для візуалізації архітектурного рішення використано діаграму компонентів (див. рис. 2.2). Дана діаграма відображає основні складові системи, їх логічну структуру та взаємодію між собою.

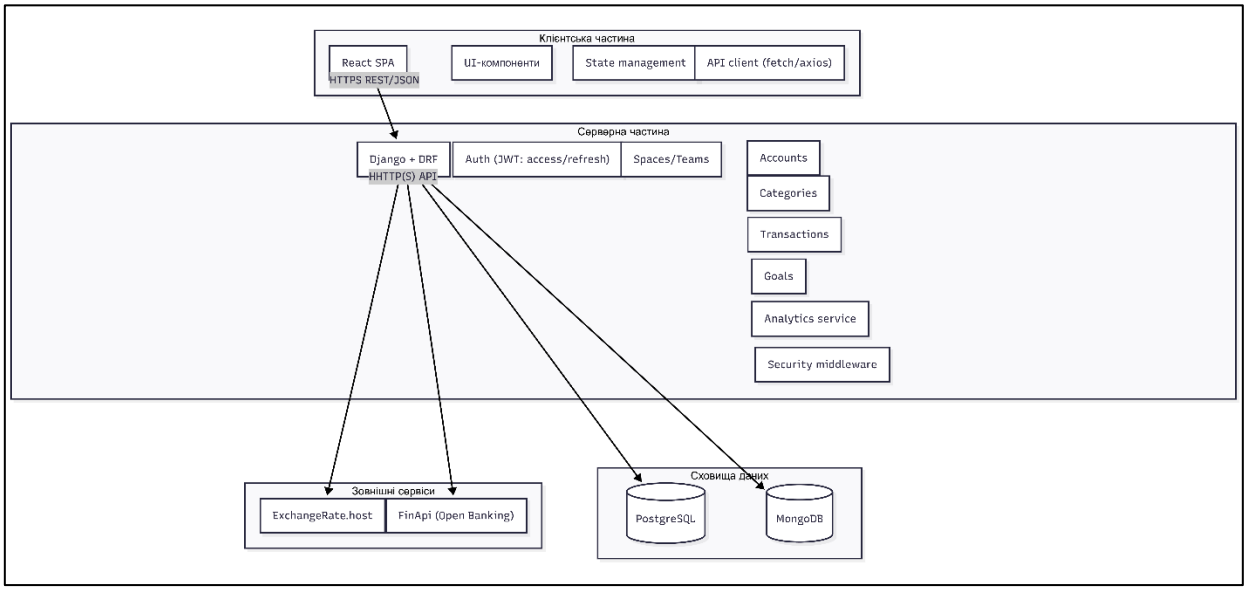


Рисунок 2.2 – Архітектура вебсервісу управління фінансами SpendSplif

Клієнтська частина вебсервісу реалізується у вигляді односторінкового застосунку (SPA) на основі бібліотеки React. Вона включає набір UI-компонентів, що відповідають за відображення інтерфейсу користувача, а також механізми управління станом застосунку. Для взаємодії з серверною частиною використовується API-клієнт, побудований на базі стандартних HTTP-запитів (fetch або axios), який забезпечує обмін даними у форматі JSON через захищене з’єднання HTTPS.

Серверна частина вебсервісу побудована з використанням Django та Django REST Framework і виступає центральним компонентом системи. Вона відповідає за обробку запитів від клієнта, реалізацію бізнес-логіки, контроль доступу та інтеграцію з базами даних і зовнішніми сервісами. В межах серверної архітектури виділено окремі логічні модулі, кожен з яких відповідає за певну функціональну область системи.

Ключовим елементом серверної частини є модуль автентифікації та авторизації, який реалізує механізм доступу на основі JWT-токенів із розділенням на access та refresh токени. Такий підхід дозволяє забезпечити безпечну роботу користувачів, контроль сесій та захист від несанкціонованого доступу. Окремо передбачено використання захисних middleware, що покривають типові вектори атак, зокрема CSRF, XSS, CORS та clickjacking.

Функціональні модулі серверної частини включають компоненти керування фінансовими просторами та командами, рахунками, категоріями, транзакціями та фінансовими цілями. Кожен із цих компонентів реалізує власний набір REST-ендпоінтів та працює незалежно, що відповідає принципам модульності та спрощує супровід і розширення системи. Окремий аналітичний сервіс відповідає за агрегацію фінансових даних і підготовку інформації для формування звітів.

Для зберігання даних у системі використовується підхід із застосуванням двох типів сховищ. Реляційна база даних PostgreSQL призначена для зберігання структурованих фінансових даних, користувачів, рахунків і транзакцій. Документоорієнтована база даних MongoDB

використовується для збереження допоміжних та аналітичних даних, що дозволяє гнучко працювати зі складними структурами та історією змін.

Окреме місце в архітектурі займає інтеграція із зовнішніми сервісами. Взаємодія з сервісом ExchangeRate.host забезпечує актуальну конвертацію валют для мультивалютних рахунків, а підключення до FinApi реалізує доступ до банківських даних через механізми Open Banking. Обмін інформацією з зовнішніми сервісами здійснюється через захищені HTTP(S) API.

У результаті спроєктована архітектура вебсервісу відповідає сучасним вимогам до вебзастосунків даного класу, забезпечує чітке розділення відповідальностей між компонентами та створює надійну основу для подальшої реалізації й масштабування системи.

2.3 Проєктування клієнтської частини на React

Проєктування клієнтської частини вебсервісу управління фінансами спрямоване на забезпечення зручної взаємодії користувача із системою, логічної навігації між функціональними модулями та масштабованої структури інтерфейсу. З урахуванням вимог до інтерактивності та швидкодії було обрано підхід односторінкового застосунку (Single Page Application), що дозволяє мінімізувати перезавантаження сторінок і забезпечити плавний користувацький досвід.

Клієнтська частина спроєктована на основі компонентної архітектури React, де інтерфейс формується з незалежних, повторно використовуваних компонентів. Для візуалізації структури клієнтського застосунку наведено діаграму компонентів клієнтської частини (див. рис. 2.3).

На верхньому рівні ієрархії розташований кореневий компонент App, який ініціалізує застосунок та відповідає за глобальні налаштування. Основною функцією цього компонента є підключення механізмів маршрутизації та передача управління до компонента макета інтерфейсу.

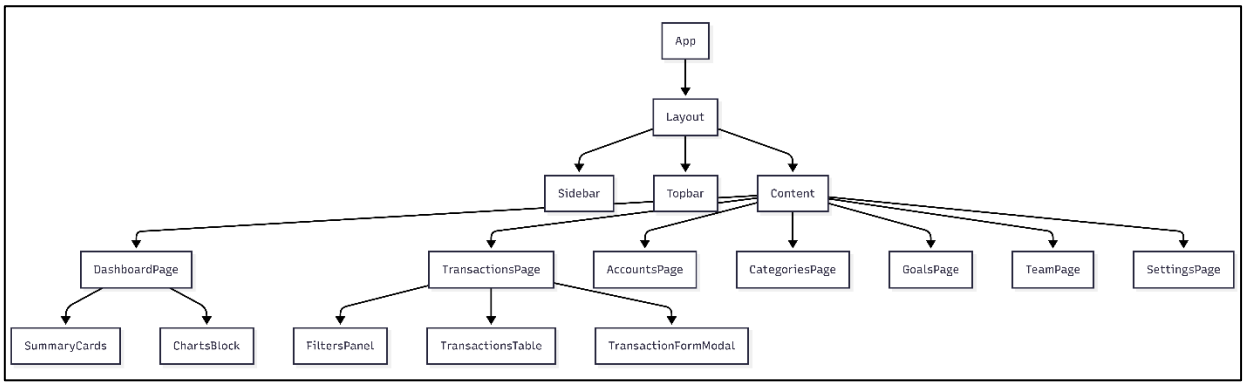


Рисунок 2.3 – Структура компонентів клієнтської частини вебсервісу SpendSplif

Компонент Layout виконує роль каркаса користувацького інтерфейсу та забезпечує єдину структуру відображення для всіх сторінок застосунку. Він об’єднує три ключові елементи інтерфейсу: бічну навігаційну панель (Sidebar), верхню панель керування (Topbar) та основну область відображення контенту (Content). Такий підхід дозволяє уніфікувати зовнішній вигляд застосунку та спростити керування навігацією.

Компонент Sidebar призначений для швидкого переходу між основними розділами системи, зокрема панеллю аналітики, рахунками, транзакціями, категоріями, фінансовими цілями та налаштуваннями. Topbar, у свою чергу, використовується для відображення контекстної інформації, елементів керування простором, профілем користувача та допоміжних дій. Компонент Content виступає контейнером для динамічного відображення сторінок залежно від обраного маршруту.

У межах області контенту спроєктовано окремі сторінкові компоненти, кожен з яких відповідає за певний функціональний блок системи. Так, DashboardPage використовується для відображення зведеної інформації про фінансовий стан користувача. Вона включає вкладені компоненти SummaryCards, що відображають ключові показники, та ChartsBlock, який відповідає за графічну візуалізацію даних.

Компонент TransactionsPage реалізує сценарії роботи з фінансовими операціями та включає панель фільтрації (FiltersPanel), таблицю транзакцій

(TransactionsTable) і модальне вікно для створення або редагування операцій (TransactionFormModal). Така декомпозиція дозволяє ізолювати логіку окремих підфункцій і спрощує підтримку коду.

Окремі сторінки AccountsPage, CategoriesPage, GoalsPage, TeamPage та SettingsPage відповідають за керування рахунками, категоріями, фінансовими цілями, учасниками команди та налаштуваннями системи відповідно. Кожна з цих сторінок розглядається як самостійний функціональний модуль, що може розвиватися незалежно від інших частин клієнтського застосунку.

У результаті запропонована структура клієнтської частини забезпечує чітке розділення відповідальностей між компонентами, підвищує читабельність коду та створює основу для масштабування інтерфейсу без необхідності суттєвих архітектурних змін.

Спроєктована компонентна модель є логічним продовженням загальної архітектури вебсервісу та узгоджується з вимогами, сформованими у попередніх підрозділах.

2.4 Проєктування серверної частини

Серверна частина вебсервісу управління фінансами виконує ключову роль у забезпеченні бізнес-логіки системи, збереженні та обробці фінансових даних, контролі доступу користувачів і взаємодії з зовнішніми сервісами. У процесі проєктування серверної архітектури основну увагу приділено модульності, безпеці, масштабованості та можливості незалежного розвитку окремих функціональних компонентів.

Для реалізації серверної частини обрано стек технологій Python, Django та Django REST Framework, що дозволяє будувати REST-орієнтовані API з чітким розділенням відповідальностей між рівнями системи. Архітектурне рішення серверної частини представлено у вигляді діаграми компонентів (див. рис. 2.4).

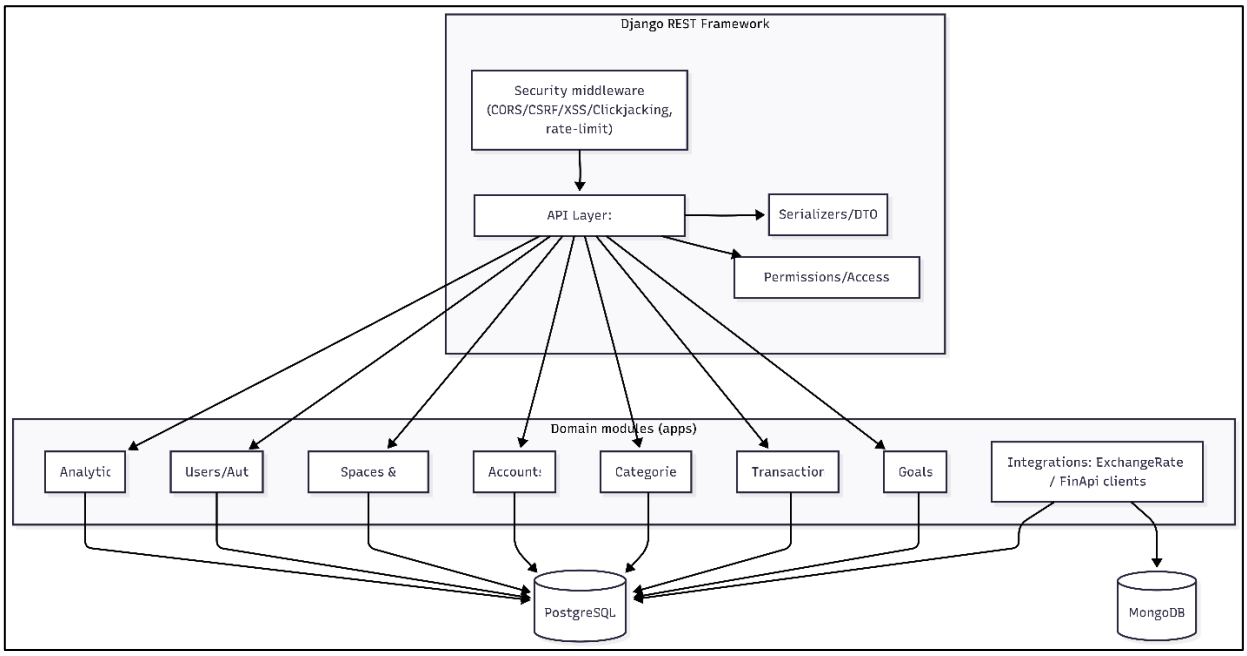


Рисунок 2.4 – Архітектура серверної частини вебсервісу на основі Django REST Framework

Верхній рівень серверної архітектури представлений набором захисних middleware, які обробляють кожен HTTP-запит до моменту його потрапляння в прикладну логіку. Даний рівень відповідає за реалізацію базових механізмів безпеки, зокрема обмеження кількості запитів, контроль міждоменних запитів, захист від CSRF, XSS та clickjacking-атак. Такий підхід дозволяє централізовано забезпечити захист системи незалежно від конкретних бізнес-модулів.

Наступним рівнем є API-рівень, представлений контролерами та ViewSet-компонентами Django REST Framework. Саме на цьому рівні відбувається приймання запитів від клієнтської частини, перевірка прав доступу та маршрутизація запитів до відповідних доменних модулів. Для стандартизації обміну даними використовується формат JSON, а всі API-виклики здійснюються через захищене з’єднання HTTPS.

Для перетворення внутрішніх моделей даних у формат, придатний для передачі клієнту, використовується шар Serializer/DTO. Він забезпечує валідацію вхідних даних, контроль типів і структури запитів, а також формування узгоджених відповідей API. Окремо виділено модуль керування

доступом і правами, який реалізує перевірку ролей користувачів та обмежує доступ до ресурсів у межах фінансових просторів.

Основу серверної частини складають доменні модулі, кожен з яких відповідає за окрему функціональну область системи. Зокрема, виділено модулі автентифікації та користувачів, керування просторами та ролями, рахунками, категоріями, транзакціями, фінансовими цілями та аналітикою. Такий поділ відповідає принципам предметно-орієнтованого проєктування та дозволяє ізолювати бізнес-логіку окремих підсистем.

Модуль інтеграцій відповідає за взаємодію з зовнішніми API. Через нього реалізується доступ до сервісу ExchangeRate.host для отримання актуальних курсів валют, а також до FinApi, який забезпечує підключення банківських рахунків користувачів у межах Open Banking. Взаємодія з зовнішніми сервісами винесена в окремий компонент з метою зменшення зв’язності системи та спрощення супроводу коду.

Для зберігання даних у серверній частині використовується комбінація реляційної та документоорієнтованої баз даних. PostgreSQL застосовується для зберігання основних фінансових і користувацьких даних, тоді як MongoDB використовується для аналітичної інформації та допоміжних структур. Такий підхід дозволяє оптимально поєднати строгість реляційної моделі з гнучкістю документоорієнтованого сховища.

У результаті спроєктована серверна архітектура забезпечує чітке розділення рівнів системи, високий рівень безпеки та можливість подальшого масштабування вебсервісу. Запропоноване рішення є логічним продовженням загальної архітектури системи та створює надійну основу для реалізації функціональних вимог, визначених у попередніх підрозділах.

2.5 Проєктування бази даних

Проєктування бази даних є одним із ключових етапів створення вебсервісу управління фінансами, оскільки саме структура даних визначає

цілісність фінансової інформації, коректність бізнес-логіки та можливість масштабування системи. З огляду на складність предметної області та наявність декількох функціональних підсистем, модель даних була декомпозована на окремі логічні блоки, кожен з яких представлено у вигляді самостійної діаграми.

Такий підхід дозволяє зберегти читабельність схем, чітко відобразити зв'язки між сутностями та уникнути перевантаження однієї узагальненої діаграми, що особливо важливо для складних багатосутнісних інформаційних систем. Декомпозиція моделі даних на логічні підсистеми забезпечує кращу структурованість, спрощує аналіз предметної області та підвищує зрозумілість архітектурних рішень як на етапі проєктування, так і під час подальшої реалізації.

Основою системи є користувачі та робочі простори (Space), у межах яких здійснюється фінансовий облік. Користувач може бути учасником декількох просторів із різними ролями та рівнями доступу. Для цього використовується проміжна сутність SPACE_MEMBER_PERMISSIONS, яка зберігає роль користувача та набір дозволів (див. рис. 2.5).

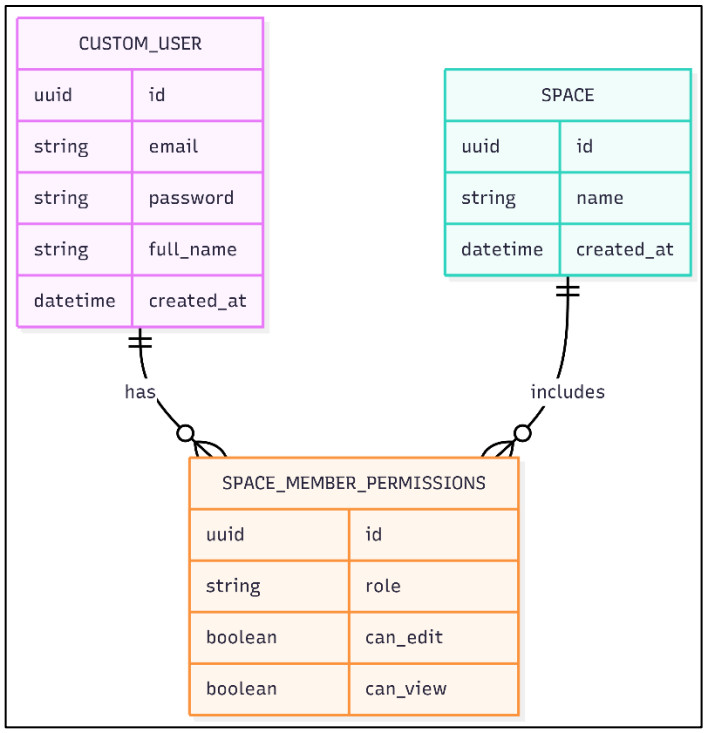


Рисунок 2.5 – Модель користувачів, просторів та прав доступу

Запропонована модель забезпечує гнучке управління доступом, підтримує командну роботу та дозволяє розмежовувати права редагування й перегляду фінансових даних.

Фінансові операції в системі організовані навколо рахунків та категорій, які належать до конкретного простору. Кожна транзакція пов’язана з рахунком і категорією, що дозволяє здійснювати деталізований облік доходів і витрат (див. рис. 2.6).

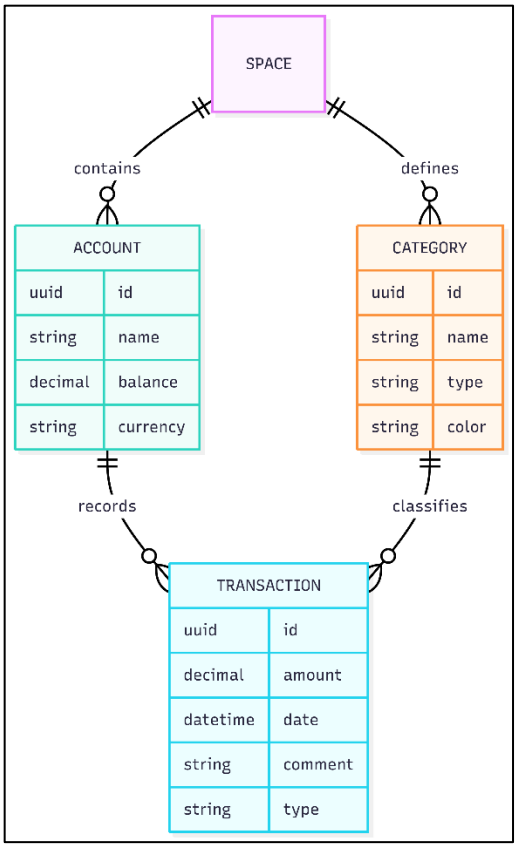


Рисунок 2.6 – Модель рахунків, категорій та транзакцій

Така структура дозволяє класифікувати транзакції за типом, валютами та часовими періодами, що є необхідною умовою для формування аналітики та звітів.

Для підтримки внутрішніх фінансових операцій у системі реалізовано окрему модель переказів між рахунками. Сутність TRANSFER дозволяє фіксувати рух коштів між двома рахунками з урахуванням суми та дати операції (див. рис. 2.7).

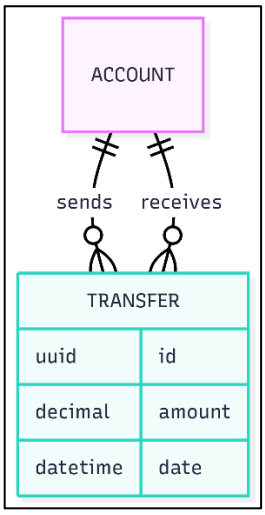


Рисунок 2.7 – Модель переказів між рахунками

Виділення переказів в окрему сутність дозволяє коректно обробляти такі операції без спотворення статистики доходів і витрат.

Для реалізації функціоналу фінансового планування в системі передбачено сутність GOAL, яка дозволяє користувачам задавати цільові суми, відстежувати поточний прогрес та контролювати строки досягнення цілей (див. рис. 2.8).

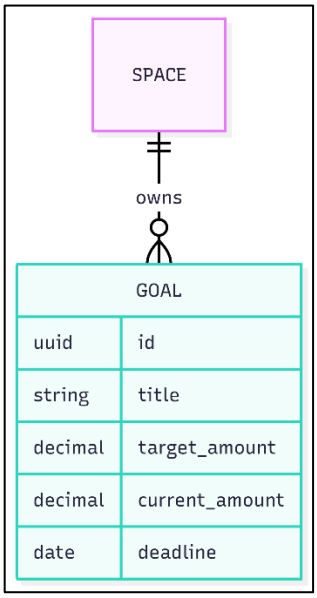


Рисунок 2.8 – Модель фінансових цілей

Цілі прив’язуються до простору, що дозволяє використовувати їх як у персональному, так і в командному фінансовому плануванні.

Для автоматизації отримання банківських даних реалізовано модель інтеграції з зовнішніми банківськими сервісами через Open Banking. Сутність BANK_CONNECTION відповідає за зв’язок простору з банком, а BANK_ACCOUNT – за конкретні рахунки, отримані з зовнішнього API (див. рис. 2.9).

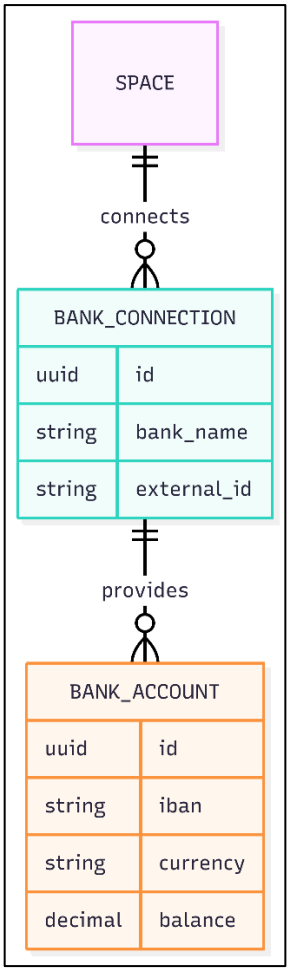


Рисунок 2.9 – Модель інтеграції з банківськими сервісами

Такий підхід дозволяє синхронізувати транзакції з банківськими рахунками та забезпечує актуальність фінансових даних у системі.

Запропонована структура бази даних забезпечує логічну цілісність, масштабованість і підтримку всіх ключових функцій вебсервісу управління фінансами. Декомпозиція моделі на окремі діаграми підвищує наочність проєктних рішень і спрощує подальший розвиток системи, зокрема розширення аналічного та інтеграційного функціоналу.

2.6 Висновки до розділу 2

У другому розділі виконано проєктування вебсервісу управління фінансами, визначено його архітектуру, основні компоненти та зв’язки між ними. Сформовано логічну модель системи, що охоплює клієнтську частину, серверну логіку та базу даних.

Проєктування клієнтської частини на React дозволило окреслити структуру інтерфейсу та сценарії взаємодії користувача з фінансовими даними. Серверна частина на основі Django та Django REST Framework забезпечує реалізацію бізнес-логіки, контроль доступу й інтеграцію зовнішніх сервісів.

Структура бази даних орієнтована на надійне зберігання фінансової інформації, підтримку аналітики та масштабованість системи. Запроєктовані рішення створюють основу для переходу до етапу реалізації та тестування вебсервісу, що розглядається у наступному розділі.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Реалізація клієнтської частини React

Клієнтська частина вебсервісу реалізована у вигляді односторінкового застосунку з використанням компонентного підходу. Основна увага під час розробки була приділена зручності користування, адаптивності інтерфейсу та чіткому розділенню логіки відображення і роботи з даними. Такий підхід дозволяє забезпечити масштабованість клієнтської частини та спростити її подальший розвиток.

Інтерфейс вебзастосунку автоматично адаптується до різних розмірів екрану та масштабу браузера. Це особливо важливо з огляду на використання сервісу як на настільних комп’ютерах, так і на ноутбуках з різною роздільною здатністю.

На рисунку 3.1 та 3.2 продемонстровано приклади адаптації головної сторінки застосунку при зміні масштабу інтерфейсу. Видно, що всі ключові елементи залишаються доступними, зберігають коректне позиціонування та читабельність, а навігація не втрачає зручності.

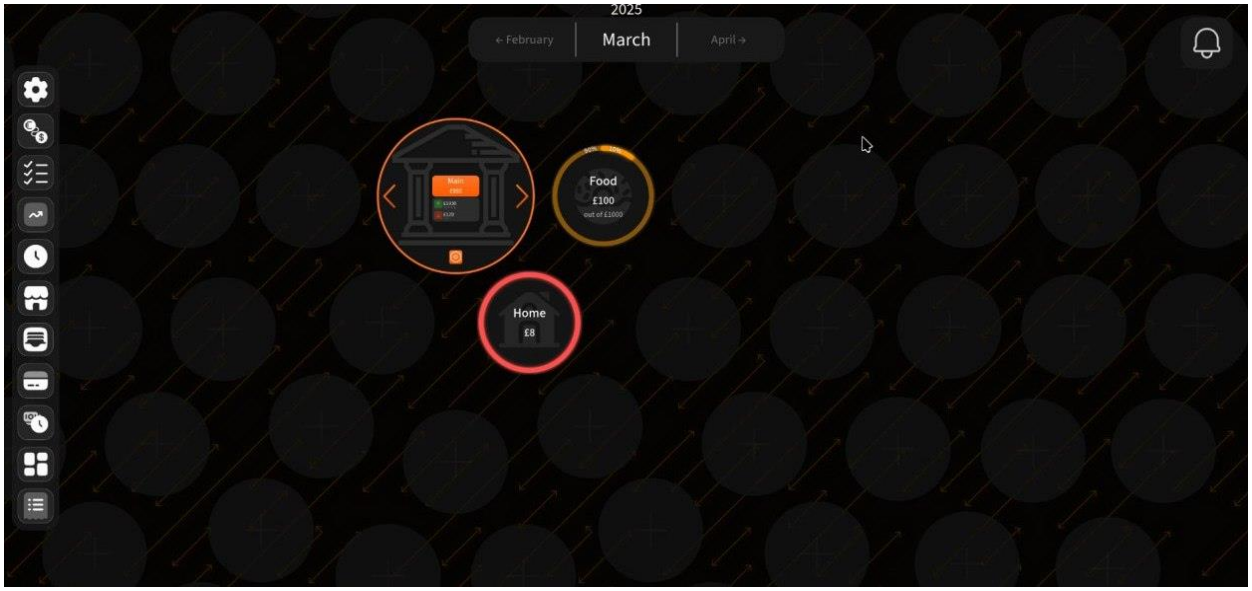


Рисунок 3.1 – Приклад адаптивності інтерфейсу при стандартному масштабі

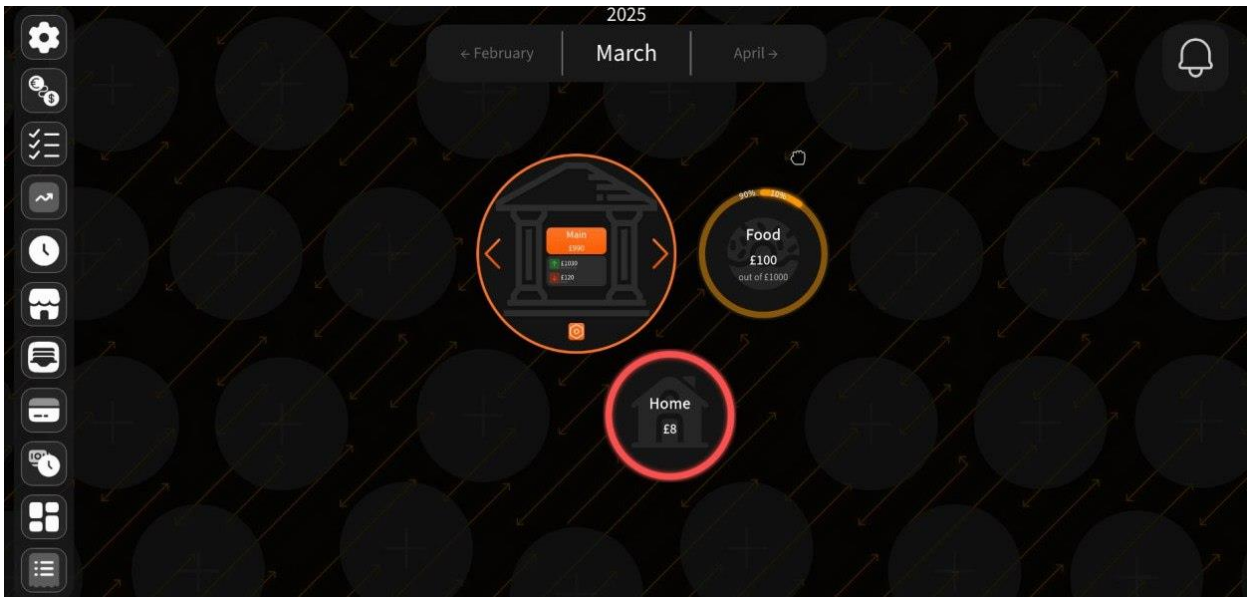


Рисунок 3.2 – Приклад адаптивності інтерфейсу при зміненому масштабі

Центральним елементом клієнтської частини є головна сторінка вебзастосунку, яка відображає фінансову структуру обраного простору у вигляді візуальних блоків рахунків і категорій. На рисунку 3.3 наведено загальний вигляд головної сторінки, де користувач може переглядати поточні баланси, переходити між періодами та взаємодіяти з фінансовими об’єктами.



Рисунок 3.3 – Головна сторінка вебзастосунку управління фінансами

Для забезпечення динамічної взаємодії з даними у клієнтській частині використано асинхронні запити до серверного API, а також механізми

управління станом застосунку. На рисунках 3.4 та 3.5 наведено фрагменти програмного коду, що відповідають за завантаження активного простору, обробку станів завантаження, взаємодію з банківськими підключеннями та оновлення інтерфейсу без перезавантаження сторінки.Зокрема, реалізовано логіку обробки асинхронних запитів, керування локальним станом, реагування на зміни параметрів маршруту та коректне відображення модальних вікон. Такий підхід дозволяє забезпечити плавну роботу інтерфейсу та позитивний користувацький досвід навіть при складних сценаріях взаємодії.



```
        } else {
          SpaceAPI.isBankActionRequired(spaceId)
            .then((res) => {
              if (res.status === 204) {
              } else if (res.status === 200) {
                setData(res.data);
                setShowingCustomAlert(true);
              }
            })
            .catch((error) => {
              console.error("Error fetching spaces:", error);
            });
        }
      })
      .catch((error) => {
        console.error("Error fetching spaces:", error);
      });
  }, []);

  useEffect(() => {
    if (id) {
      navigate("/home", { state: { spaceId: DOMPurify.sanitize(id) } });
    }
  }, [id, navigate]);

  const handleUpdateClick = () => {
    if (!data) return;

    setIsLoading(true);

    SpaceAPI.updateConnection(spaceId, {
      bank_connection_id: data.connection_id,
    })
      .then((res) => {
        if (res) {
          window.location.href = res.url;
        }
      })
      .catch((error) => {
        setIsLoading(false);
        console.error("Error fetching spaces:", error);
      });
  };

  return (
    <>
      {isLoading && <Loader />}
      <div className={classes.main}>
        <Expenses />
        <Overlay />
        {showingCustomAlert && data && (
          <Modal
            setActive={
              setShowingCustomAlert as Dispatch<SetStateAction<boolean>>
            }
          >
            <CustomAlertUpdateBank
              data={data}
              handleUpdateClick={handleUpdateClick}
              setActive={setShowingCustomAlert}
            />
          </Modal>
        )}
      </div>
    </>
  );
};

export default HomePage;
```

Рисунок 3.4 – Фрагмент коду клієнтської логіки головної сторінки

```
interface IData {
  bank: string;
  connection: string;
  connection_id: number;
}

const HomePage = () => {
  const navigate = useNavigate();
  const { id } = useParams();
  const { t } = useTranslation();
  const [showingCustomAlert, setShowingCustomAlert] = useState(false);
  const [data, setData] = useState<IData | null>(null);
  const [isLoading, setIsLoading] = useState(true);
  const spaceId = getSpaceId();

  useEffect(() => {
    const handleSetLoading = (loading: boolean) => {
      if (loading || (accounts && categories)) {
        setIsLoading(loading);
      }
    };

    eventEmitter.on("setLoading", handleSetLoading);

    return () => {
      eventEmitter.off("setLoading", handleSetLoading);
    };
  }, []);

  const { accounts, totalBalance, setAccounts, setTotalBalance, categories } =
    useStore();

  const { data: space } = useQuery({
    queryKey: ["get active space"],
    queryFn: () => SpaceAPI.getActiveSpace(),
  });

  // to disable the loader after the back button press
  useEffect(() => {
    if (accounts && categories) {
      setIsLoading(false);
    }
  }, [accounts, categories]);

  useEffect(() => {
    if (space) {
      localStorage.setItem("space", JSON.stringify(sanitizeObject(space)));
      // after logining through google - accounts and totalBalance - undefined
      if (!accounts || !totalBalance) {
        setIsLoading(true);
        AccountAPI.ViewAccounts(space.id)
          .then((res) => {
            setAccounts(res.data.Account);
            setTotalBalance(res.data.TotalBalance);
          })
          .finally(() => setIsLoading(false));
      }
    }
  }, [space]);

  useEffect(() => {
    spaceId &&
      SpaceAPI.spaceStatus(spaceId)
        .then((res) => {
          if (!res.active) {
            navigate("/admin-panel/");
            darktalk.alert(t("admin.service"), t("admin.yougottapay"));
          } else {
```

Рисунок 3.5 – Обробка стану інтерфейсу та взаємодії з API

Таким чином, клієнтська частина вебсервісу реалізована з урахуванням сучасних вимог до адаптивності, інтерактивності та зручності використання. Використані підходи дозволяють ефективно відображати фінансові дані, забезпечують швидку реакцію інтерфейсу на дії користувача та створюють надійну основу для подальшого розвитку функціоналу.

3.2 Реалізація серверної частини

Серверна частина вебсервісу SpendSplif реалізована з використанням бекенд-фреймворку Django та розширення Django REST Framework, які забезпечують побудову REST-орієнтованого API, виконання бізнес-логіки, автентифікацію користувачів і контроль доступу до ресурсів системи. Обраний підхід дозволяє створювати масштабовану та надійну серверну архітектуру з чітким розподілом функціональних обов'язків.

Архітектура серверної частини побудована за модульним принципом, відповідно до якого кожна предметна підсистема реалізована у вигляді окремого застосунку. Така структура спрощує супровід коду, полегшує тестування та дозволяє розширювати функціонал без впливу на інші компоненти системи.

Функціональність, пов'язана з користувачами, зосереджена в окремому серверному модулі, який відповідає за реєстрацію, отримання профілю та обробку запитів, що потребують автентифікації. Для реалізації API використано клас-орієнтовані представлення Django REST Framework, що дозволяють описувати логіку обробки HTTP-запитів у структурованому та декларативному вигляді.

На рисунку 3.6 наведено приклад реалізації серверних класів, які відповідають за створення нового користувача та отримання даних профілю поточного користувача.

```
class CustomUserRegistrationView(generics.CreateAPIView):
    queryset = CustomUser.objects.all()
    serializer_class = CustomUserSerializer
    permission_classes = (permissions.AllowAny,)


class UserProfileView(generics.RetrieveAPIView):
    serializer_class = CustomUserSerializer

    def get_object(self):
        return self.request.user
```

Рисунок 3.6 – Реалізація серверних API для реєстрації та профілю користувача

Доступ до відповідних ендпоінтів контролюється за допомогою permission-класів, що забезпечує захист персональних даних.

Окремий серверний модуль призначений для роботи з категоріями витрат. Він містить логіку створення, перевірки та обмеження доступу до категорій у межах фінансового простору. Кожен запит проходить перевірку прав користувача відповідно до його ролі та рівня доступу.

На рисунку 3.7 показано приклад реалізації серверного класу створення категорії, в якому додатково реалізовано перевірки бізнес-обмежень, зокрема максимально допустимої кількості категорій залежно від типу доступу користувача. Такий підхід дозволяє централізовано реалізувати правила системи без дублювання логіки на клієнтській стороні.

```python

class CreateCategory(generics.CreateAPIView):
    serializer_class = CategorySerializer
    permission_classes = (IsSpaceMember & (IsSpaceOwner | CanCreateCategories),)

    def create(self, request, *args, **kwargs):
        space_pk = self.kwargs.get('space_pk')
        space = get_object_or_404(Space, pk=space_pk)

        user_categories_counter = Category.objects.filter(father_space=space).count()
        highest_role = self.request.user.roles[0]
        if user_categories_counter >= 126 and highest_role == "business_lic":
            return Response("Error: you can't create more than 126 categories because your role is Business license", status=status.HTTP_422_UNPROCESSABLE_ENTITY)
        elif user_categories_counter >= 126 and highest_role == "business_plan" :
            return Response("Error: you can't create more than 126 categories because your role is Business plan", status=status.HTTP_422_UNPROCESSABLE_ENTITY)
        elif user_categories_counter >= 3 and highest_role == "free":
            return Response("Error: you can't create more than 3 categories because your role is free", status=status.HTTP_422_UNPROCESSABLE_ENTITY)

        data = request.data.copy()
        data['father_space'] = space.pk
        data['spent'] = 0

        serializer = self.get_serializer(data=data)
        serializer.is_valid(raise_exception=True)
        self.perform_create(serializer)

        headers = self.get_success_headers(serializer.data)
        return Response(serializer.data, status=status.HTTP_201_CREATED, headers=headers)
```

Рисунок 3.7 – Серверна логіка створення категорій з перевіркою доступу

Для зберігання інформації використовується об’єктно-реляційна модель даних. Кожна сутність системи описується у вигляді окремої моделі з чітко визначеними атрибутами та зв’язками. Це забезпечує цілісність даних і спрощує реалізацію складних взаємозв’язків між об’єктами.

На рисунку 3.8 наведено приклад реалізації серверної моделі категорії, яка містить основні атрибути та зв’язок із фінансовим простором. Використання вбудованих валідаторів і типізованих полів дозволяє забезпечити коректність даних ще на рівні серверної логіки.

```python
from django.db import models
from colorfield.fields import ColorField

from backend.apps.category.constants import Icons

from backend.apps.space.models import Space

from django.core.validators import MaxValueValidator


class Category(models.Model):
    title = models.CharField(max_length=24)
    spent = models.DecimalField(max_digits=12, decimal_places=2, validators=[MaxValueValidator(99000000000)])
    limit = models.DecimalField(max_digits=20, decimal_places=2, null=True, validators=[MaxValueValidator(99000000000)])
    color = ColorField(format="hex")
    icon = models.CharField(choices=Icons.choices)
    father_space = models.ForeignKey(Space, verbose_name='father_space', on_delete=models.CASCADE)

    def __str__(self):
        return self.title
```

Рисунок 3.8 – Модель Category у серверній частині

Таким чином, серверна частина вебсервісу реалізована у вигляді набору незалежних модулів, що взаємодіють через стандартизовані API-інтерфейси. Така архітектура забезпечує гнучкість, безпеку та створює передумови для подальшого розвитку і масштабування системи.

3.3 Налаштування бази даних (PostgreSQL)

База даних вебсервісу реалізована на основі реляційної системи керування базами даних PostgreSQL. Формування структури бази даних здійснюється за допомогою об’єктно-реляційного відображення (ORM), що надається фреймворком Django. Такий підхід дозволяє описувати структуру таблиць, зв’язки між ними та обмеження цілісності безпосередньо на рівні програмного коду, що значно спрощує супровід і розвиток проєкту.

Основними елементами взаємодії з базою даних є Django-моделі, кожна з яких відповідає окремій таблиці в PostgreSQL. Під час виконання міграцій ORM автоматично створює або змінює структуру таблиць відповідно до описаних моделей. Це забезпечує узгодженість між логікою застосунку та фізичною структурою бази даних.

На рисунку 3.9 наведено приклад моделі категорій витрат і доходів, яка використовується в системі.

```python
from django.db import models
from colorfield.fields import ColorField

from backend.apps.category.constants import Icons

from backend.apps.space.models import Space

from django.core.validators import MaxValueValidator


class Category(models.Model):
    title = models.CharField(max_length=24)
    spent = models.DecimalField(max_digits=12, decimal_places=2, validators=[MaxValueValidator(99000000000)])
    limit = models.DecimalField(max_digits=20, decimal_places=2, null=True, validators=[MaxValueValidator(99000000000)])
    color = ColorField(format="hex")
    icon = models.CharField(choices=Icons.choices)
    father_space = models.ForeignKey(Space, verbose_name='father_space', on_delete=models.CASCADE)

    def __str__(self):
        return self.title

```

Рисунок 3.9 – Фрагмент коду Django-моделі категорії

Дана модель визначає структуру таблиці category, а також її зв’язок із простором користувача.

Модель Category успадковує базовий клас models.Model, що є стандартним підходом у Django. Поле title використовується для зберігання назви категорії та обмежене максимальною довжиною рядка. Поля spent і limit реалізовані як десяткові значення, що дозволяє коректно працювати з фінансовими даними без втрати точності. Для цих полів також застосовано валідатори, які обмежують максимально допустимі значення, запобігаючи некоректному введенню даних.

Поле color використовується для збереження кольору категорії у шістнадцятковому форматі, що надалі застосовується для візуального відображення категорій у клієнтському інтерфейсі. Поле icon реалізоване з використанням фіксованого набору значень, що забезпечує узгодженість між логікою сервера та інтерфейсом користувача.

Зв’язок між категорією та простором реалізовано за допомогою зовнішнього ключа father_space, який посилається на модель простору. Такий підхід дозволяє ізолювати фінансові дані в межах окремих робочих просторів і забезпечує коректну багатокористувацьку роботу системи. При видаленні простору всі пов’язані з ним категорії автоматично видаляються, що реалізується за рахунок параметра on_delete=models.CASCADE.

Таким чином, використання ORM у поєднанні з Django-моделями дозволяє формувати структуру бази даних PostgreSQL у декларативному вигляді, забезпечувати цілісність фінансових даних та спростити подальше розширення функціоналу вебсервісу.

3.4 Інтеграція API та зовнішніх сервісів

Однією з ключових функціональних можливостей вебсервісу є підтримка мультивалютності, що реалізується шляхом інтеграції із зовнішнім сервісом обміну валют ExchangeRate.host. Використання зовнішнього API дозволяє забезпечити актуальні курси валют та коректні фінансові розрахунки незалежно від обраної користувачем базової валюти.

На рівні клієнтської частини реалізовано інтерфейс вибору основної валюти простору. Користувач може обрати з переліку доступних валют, після чого всі фінансові показники відображаються з урахуванням відповідного курсу обміну (див. рис. 3.10).



Рисунок 3.10 – Інтерфейс вибору базової валюти простору

Інтерфейс дозволяє швидко знайти необхідну валюту, переглянути її поточне значення та підтвердити вибір. Зміна базової валюти не впливає на

збережені первинні дані транзакцій, а лише змінює спосіб їх відображення й агрегування в аналітиці.

Серверна частина інтеграції реалізована через окремий модуль, відповідальний за зберігання та оновлення валютних курсів. Первинне наповнення таблиці валют відбувається під час виконання міграцій бази даних. У цей момент сервер звертається до API ExchangeRate.host, отримує актуальні курси та зберігає їх у базі даних.

Після початкової ініціалізації оновлення валютних курсів виконується в автоматичному режимі за допомогою фонових задач. Для цього використовується планувальник задач, який періодично викликає логіку оновлення курсів з API ExchangeRate.host. Оновлення відбувається щогодини за умови запущеного сервера та сервісу обробки фонових задач, що дозволяє підтримувати актуальність даних без участі користувача.

У результаті реалізована інтеграція із зовнішнім валютним сервісом забезпечує підтримку мультивалютності, підвищує точність фінансових розрахунків і розширює практичні можливості вебсервісу для користувачів з різних країн.

3.5 Тестування функціоналу вебсервісу

Тестування функціоналу вебсервісу управління фінансами виконувалося з метою перевірки коректності серверної логіки, розмежування доступів, стабільності бізнес-операцій та рівня захищеності системи. Для цього використано автоматизовані API-тести, які імітують типові сценарії взаємодії користувача з вебсервісом у межах різних просторів.

Першим кроком виконується підготовка тестового середовища. У межах цього етапу створюються тестові користувачі, простори, ролі доступу, рахунки та категорії. Такий підхід дозволяє виконувати всі наступні тести в ізольованих і відтворюваних умовах, незалежно від стану робочої бази даних.

```python
def setUp(self):
    self.client = APIClient()
    self.user1 = CustomUser.objects.create_user(username='user1', email='gmail1@gmail.com', password='password1')
    self.user2 = CustomUser.objects.create_user(username='user2', email='gmail2@gmail.com', password='password2')
    self.user3 = CustomUser.objects.create_user(username='user3', email='gmail3@gmail.com', password='password3')
    self.dm_chat = DmChat.objects.create(owner_1=self.user1, owner_2=self.user2, )
    self.space1 = Space.objects.create(title='space1')
    self.space1.members.set([self.user1])
    self.space2 = Space.objects.create(title='space2')
    self.perms = MemberPermissions(member=self.user2, space=self.space2)
    self.perms.owner = True
    self.perms.save()
    self.account1 = Account.objects.create(father_space=self.space1, title='account1',  balance=0, currency="USD")
    self.account2 = Account.objects.create(father_space=self.space2, title='account2',  balance=0, currency="USD")
    self.category1 = Category.objects.create(father_space=self.space1, title='category1', spent=0)
    self.category2 = Category.objects.create(father_space=self.space2, title='category2', spent=0)
```

Рисунок 3.11 – Підготовка тестового середовища

У результаті формується стабільне середовище для перевірки поведінки системи у різних сценаріях доступу та взаємодії.

Після підготовки середовища перевіряється створення фінансових категорій у межах простору (див. рис. 3.12). Тест фіксує кількість категорій до і після виконання запиту, а також коректність HTTP-відповіді сервера. Окремо перевіряється, що категорії створюються в межах відповідного простору і не впливають на дані інших просторів.

```python
def test_create_category(self):
    counter = Category.objects.count()
    self.client.login(username='user1', password='password1')
    self.client.force_authenticate(user=self.user1)
    response = self.client.post(
        f'/api/v1/my_spaces/{self.space1.pk}/create_category/',
        {
            'title': 'category1',
            'spent': 0
         }, format='json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertEqual(Category.objects.count(), counter + 1)
    self.client.login(username='user2', password='password2')
    self.client.force_authenticate(user=self.user2)
    response = self.client.post(
        f'/api/v1/my_spaces/{self.space2.pk}/create_category/',
        {
            'title': 'category2',
            'spent': 0
        }, format='json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)
    self.assertEqual(Category.objects.count(), counter + 2)

    self.client.logout()
```

Рисунок 3.12 – Створення категорії витратити

Цей етап підтверджує правильну роботу механізмів створення ресурсів та ізоляцію фінансових даних.

Далі виконується тестування перегляду категорій з урахуванням прав доступу (див. рис. 3.13). Перевіряється поведінка системи для неавторизованих запитів, користувачів без доступу до простору та користувачів з коректними правами. Очікуваними результатами є відповідні статус-коди 401, 403 та 200.

```python
def test_view_category(self):
    counter = Category.objects.count()
    response = self.client.get(f'/api/v1/my_spaces/{self.space1.pk}/my_categories/')
    self.assertEqual(response.status_code, status.HTTP_401_UNAUTHORIZED)

    self.client.login(username='user1', password='password1')
    self.client.force_authenticate(user=self.user1)
    response = self.client.get(f'/api/v1/my_spaces/{self.space2.pk}/my_categories/')
    self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)

    self.client.login(username='user1', password='password1')
    self.client.force_authenticate(user=self.user1)
    response = self.client.get(f'/api/v1/my_spaces/{self.space1.pk}/my_categories/')
    categories = Category.objects.filter(father_space=self.space1.pk)
    self.assertEqual(response.status_code, status.HTTP_200_OK)

    self.assertEqual(Category.objects.count(), counter)

    self.client.logout()
```

Рисунок 3.13 – Перевірка доступу до категорій

Результати тесту підтверджують коректну реалізацію авторизації та контроль доступу до ресурсів.

Наступним етапом перевіряється бізнес-логіка списання коштів по категорії (див. рис. 3.14). Тест моделює спробу виконання фінансової операції з використанням рахунку іншого простору, а також коректну операцію в межах дозволеного простору. Це дозволяє перевірити не лише правильність обробки фінансових даних, а й захист від несанкціонованих дій.

```python
def test_spend_to_category(self):
    self.client.login(username='user1', password='password1')
    self.client.force_authenticate(user=self.user1)
    response = self.client.put(f'/api/v1/my_spaces/{self.space1.pk}/my_categories/{self.category1.pk}/spend/',
                               {
                                   "amount": 244,
                                   "comment": "milk",
                                   "account_pk": self.account2.pk
                               }, format='json')
    self.assertEqual(response.status_code, status.HTTP_403_FORBIDDEN)
    response = self.client.put(f'/api/v1/my_spaces/{self.space1.pk}/my_categories/{self.category1.pk}/spend/',
                               {
                                   "amount": 244,
                                   "comment": "milk",
                                   "account_pk": self.account1.pk
                               }, format='json')
    self.assertEqual(response.status_code, status.HTTP_200_OK)

    self.client.logout()
```

Рисунок 3.14 – Тест витрати по категорії

Цей блок демонструє, що фінансові операції виконуються лише за умови дотримання правил доступу та зв’язків між сутностями.

Після цього тестується створення рахунків у межах просторів (див. рис. 3.15). Перевіряється обробка JSON-даних, створення нового об’єкта в базі та коректність відповіді сервера. Окремо підтверджується, що рахунки різних просторів не впливають один на одного.

```python
def test_create_account(self):
    counter = Account.objects.count()

    self.client.force_authenticate(user=self.user1)
    response = self.client.post(f'/api/v1/my_spaces/{self.space1.pk}/create_account/',
                                {'title': 'account1',  "balance": 0, "currency": "USD"}, format='json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)

    self.assertEqual(Account.objects.count(), counter + 1)

    self.client.force_authenticate(user=self.user2)
    response = self.client.post(f'/api/v1/my_spaces/{self.space2.pk}/create_account/',
                                {'title': 'account2', "balance": 0, "currency": "USD"}, format='json')
    self.assertEqual(response.status_code, status.HTTP_201_CREATED)

    self.assertEqual(Account.objects.count(), counter + 2)
```

Рисунок 3.15 – Тест створення рахунку

Цей етап підтверджує стабільність базових фінансових операцій і коректність роботи з базою даних.

Завершальним етапом тестування є перевірка безпеки вебсервісу (див. рис. 3.16). Авторизація реалізована на основі JWT-токенів, де access-токен має обмежений термін життя, а його оновлення здійснюється через refresh-токен, що зберігається в HTTPOnly cookies.

```python
MIDDLEWARE = [
    'backend.config.middleware.UpdateSpaceLastModifiedMiddleware',
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "corsheaders.middleware.CorsMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]
```

Рисунок 3.16 – Middleware та політики безпеки

Захист від XSS, CSRF, CORS, clickjacking, session attacks та auth bypass забезпечується відповідними middleware на сервері. Додатково реалізовано обмеження кількості запитів для протидії DDoS-атакам, а політика CORS дозволяє запити лише з домену фронтенду та визначених зовнішніх API.

Цей етап підтверджує наявність комплексного підходу до захисту даних і стабільності роботи вебсервісу.

Окремим етапом тестування перевірялася інтеграція вебсервісу з зовнішніми банківськими API (див. рис. 3.17).

Для підключення банківських рахунків використовується сервіс FinApi, який надає доступ до банківської інфраструктури більшості європейських банків і працює відповідно до концепції Open Banking. Це дозволяє користувачам безпечно прив’язувати свої банківські рахунки та автоматично отримувати фінансові дані.

На серверній стороні інтеграція реалізована через окремий клієнтський клас, який інкапсулює логіку взаємодії з FinApi API. Під час тестування перевірялася коректність отримання та оновлення токенів доступу, формування HTTP-запитів і обробка відповідей зовнішнього сервісу. Особливу увагу приділено сценаріям повторної автентифікації, коли токен доступу втрачає актуальність і потребує оновлення.

Представлений фрагмент коду демонструє реалізацію клієнта для роботи з FinApi, зокрема механізм отримання access-токена, його автоматичного оновлення та централізовану обробку HTTP-запитів. Авторизація здійснюється з використанням OAuth-механізмів, що відповідає вимогам Open Banking і забезпечує безпечний доступ до банківських даних користувачів.

У процесі тестування підтверджено, що серверна частина коректно взаємодіє з FinApi API, обробляє помилки зовнішнього сервісу та не порушує роботу основного фінансового функціоналу вебсервісу. Такий підхід дозволяє автоматизувати імпорт транзакцій і зменшити кількість ручних операцій з боку користувача.



```python
class FinAPIClient:
    BASE_URL_API = "https://live.finapi.io"
    BASE_URL_WEB = "https://webform-live.finapi.io"

    def __init__(self):
        self.token = self.get_finapi_token()

    @staticmethod
    def get_finapi_token():
        """Получает последний FinAPI токен КЛИЕНТА из базы данных"""
        token_obj = ClientToken.objects.first()
        return token_obj.access_token if token_obj else None

    def refresh_or_authenticate_user(self, user_space):
        """Обновляет или получает новый токен доступа для пользователя."""
        user_response = self.refresh_token(user_space.refresh_token, grant_type="refresh_token")
        if 'error' in user_response:
            user_response = self.auth_token(user_space.password, user_space.username, grant_type="password")
        access_token = user_response.get("access_token")
        user_space.access_token = access_token
        user_space.refresh_token = user_response.get("refresh_token")
        user_space.updated_at = timezone.now()
        user_space.save()
        return access_token

    def _request(self, method, endpoint, content_type, BASE_URL, data=None, params=None, access_token=None):
        """Базовый метод для выполнения HTTP-запросов к FinAPI"""
        headers = {
            'Content-Type': content_type,
        }

        token = access_token or self.token
        # Добавляем токен авторизации, если он есть
        if token and endpoint != "/api/v2/oauth/token":
            headers['Authorization'] = f'Bearer {token}'

        url = f"{BASE_URL}{endpoint}"
        try:
            print(method, url, data, params, headers)
            if content_type == 'application/x-www-form-urlencoded':
                response = requests.request(method, url, data=data, params=params, headers=headers)
            else:
                response = requests.request(method, url, json=data, params=params, headers=headers)

            # Проверяем статус код ответа
            if response.status_code == 200 and not response.text:
                # Пустой ответ с кодом 200 - это нормально
                return {}

            response.raise_for_status()
            print("Ответ", response.json())
            return response.json()
        except requests.exceptions.RequestException as e:
            print(f"Request error: {str(e)}")
            print(f"Details: {e.response.text if e.response else 'No response body available'}")
            return {'error': str(e)}

    def make_finapi_token(self):
        """Обновляет токен клиента FinAPI."""
        payload = {
            'grant_type': 'client_credentials',
            'client_id': settings.FINAPI_CLIENT_ID,
            'client_secret': settings.FINAPI_CLIENT_SECRET,
        }
        response = self._request("POST", "/api/v2/oauth/token", data=payload, content_type='application/x-www-form-urlencoded', BASE_URL=self.BASE_URL_API)

        print(response)
```

Рисунок 3.17 – Інтеграція з банківським API FinApi

Після перевірки серверної логіки виконано тестування клієнтської частини вебсервісу.

Перевіряється коректність відображення аналітичних даних, робота інтерактивних елементів і узгодженість інтерфейсу з серверними даними (див. рис. 3.18).

Далі тестується відображення карток рахунків із балансами, доходами та витратами (див. рис. 3.19).

Особливу увагу приділено коректності валют і оновленню даних після виконання фінансових операцій.

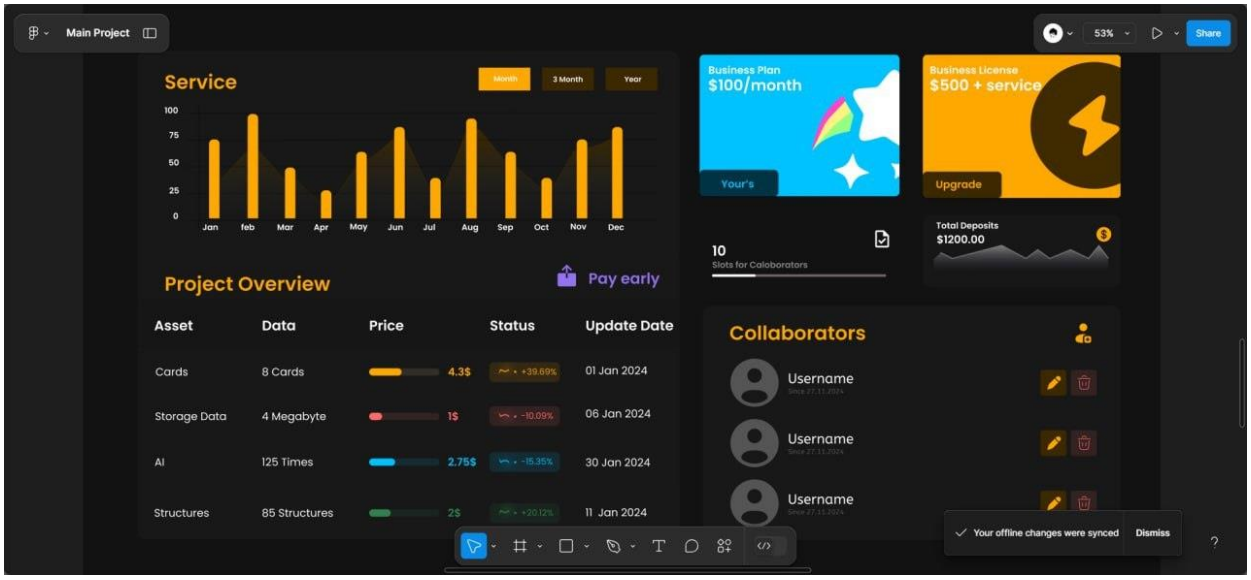


Рисунок 3.18 – Головна панель з аналітикою та зведеними показниками

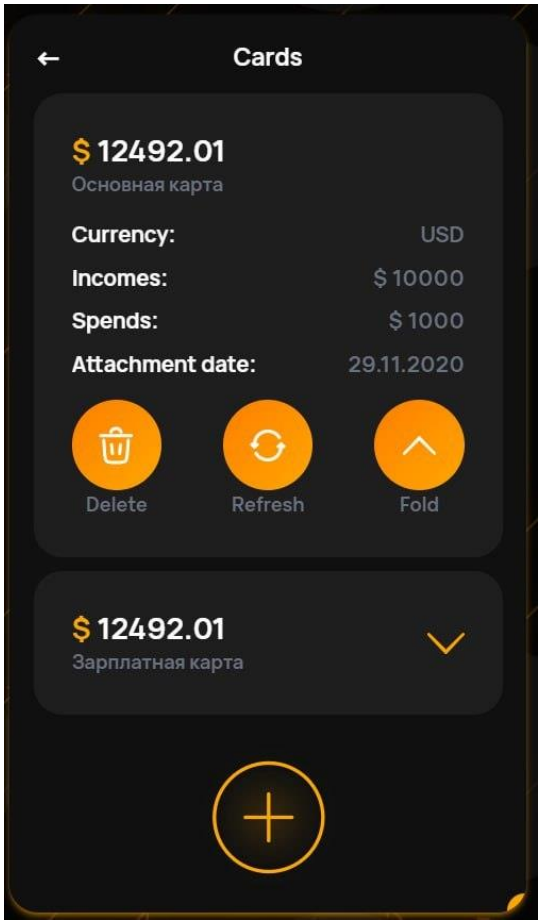


Рисунок 3.19 – Картка рахунку з фінансовою інформацією

Наступним кроком перевіряється функціонал фінансових цілей: створення, редагування, відображення прогресу та періодичності накопичень (див. рис. 3.20 та 3.21).

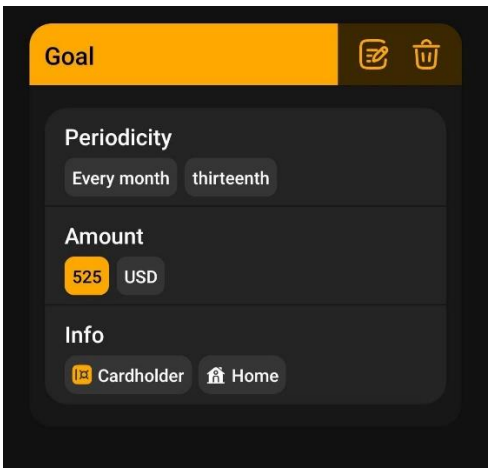


Рисунок 3.20 – Створення фінансової цілі

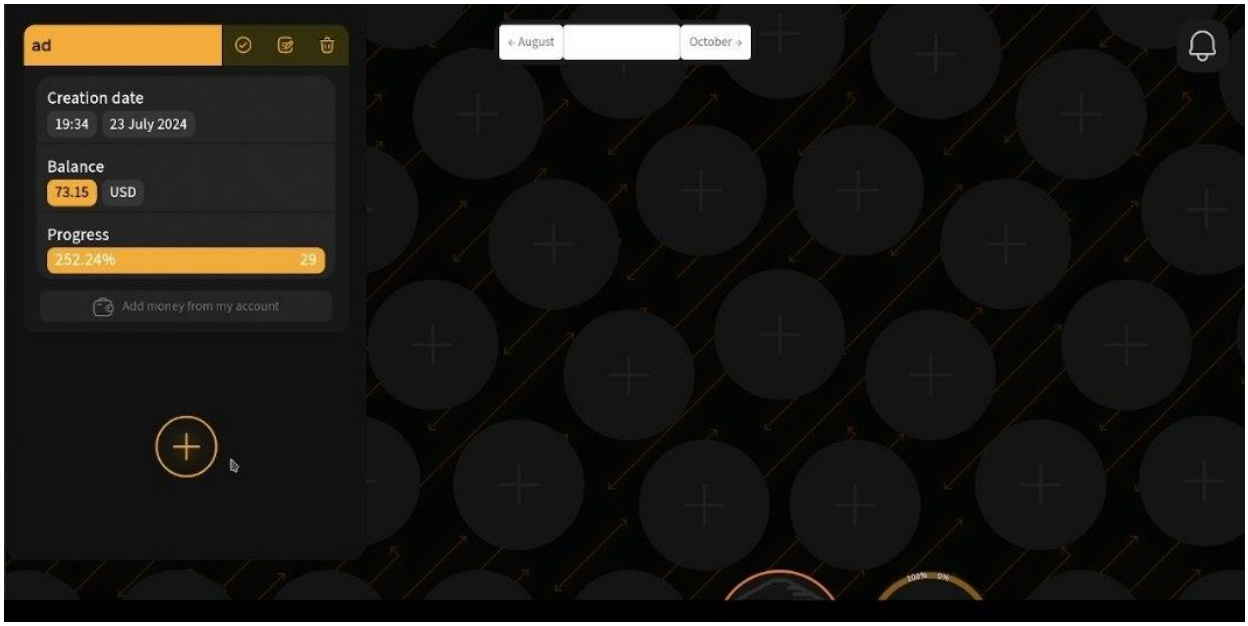


Рисунок 3.21 – Інтерфейс фінансової цілі з прогресом виконання

Окремо перевіряється створення категорій через інтерфейс користувача, зокрема вибір іконки, кольору та встановлення ліміту (див. рис. 3.22).

Завершальним етапом UI-тестування є аналіз аналітичних діаграм витрат за категоріями, що дозволяє оцінити наочність і коректність візуалізації даних (див. рис. 3.23).

Проведене тестування охопило всі ключові рівні вебсервісу – від ініціалізації серверного середовища до перевірки користувацького інтерфейсу. Отримані результати підтверджують коректність бізнес-логіки, ефективне розмежування доступів, стабільність фінансових операцій і відповідність

інтерфейсу очікуваним сценаріям використання. Це створює логічну основу для підсумкових висновків розділу та загального завершення роботи.

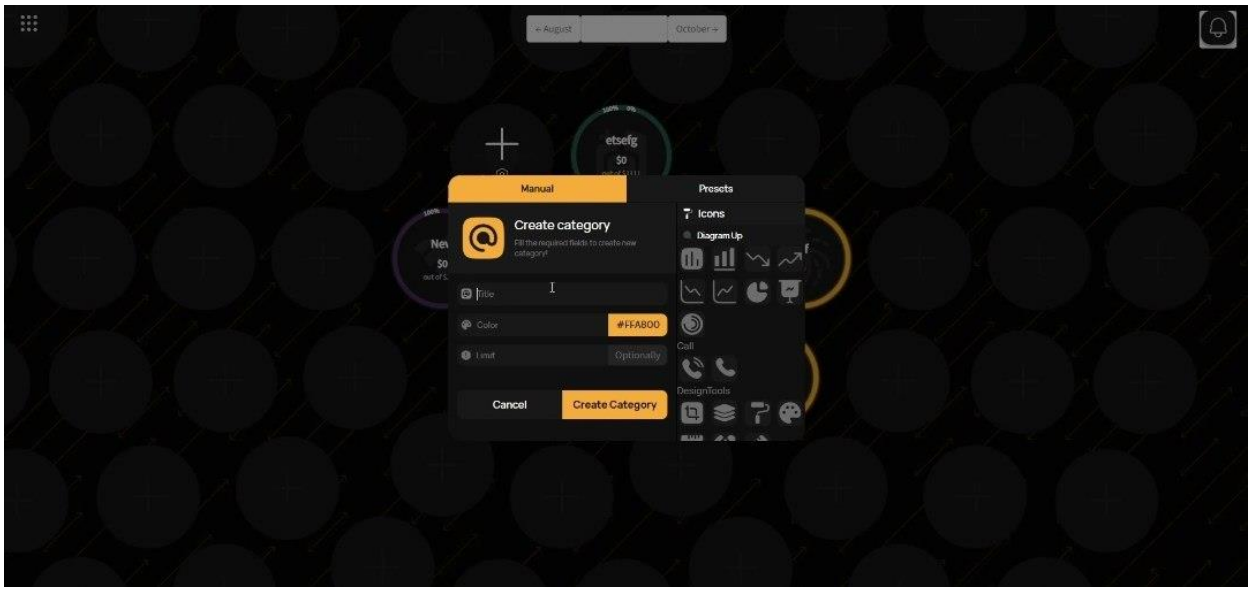


Рисунок 3.22 – Вікно створення категорії витрат



Рисунок 3.23 – Візуалізація витрат за категоріями

3.6 Висновки до розділу 3

У третьому розділі наведено основні деталі програмної реалізації вебсервісу управління фінансами відповідно до спроєктованої архітектури.

Реалізовано клієнтську частину на основі React, яка забезпечує зручний інтерфейс, адаптивність і наочне відображення фінансових даних. Компонентний підхід та управління станом дозволили створити стабільний і масштабований користувацький інтерфейс.

Серверна частина, реалізована з використанням TypeScript і бекенд-фреймворку, забезпечує обробку бізнес-логіки, контроль доступу та взаємодію з базою даних і зовнішніми сервісами. Налаштовані механізми автентифікації, перевірки прав доступу й агрегації фінансових даних гарантують коректність роботи системи та безпеку інформації.

У межах розділу виконано налаштування баз даних PostgreSQL і MongoDB, що дозволило поєднати надійне зберігання структурованих фінансових даних із гнучким збереженням допоміжної інформації. Інтеграція зовнішніх API, зокрема сервісу ExchangeRate.host, забезпечила підтримку мультивалютності та підвищила точність фінансових розрахунків.

Проведене тестування підтвердило коректну роботу основних функціональних модулів вебсервісу, стабільність інтерфейсу та відповідність реалізованого функціоналу поставленим вимогам.

ВИСНОВКИ

У кваліфікаційній роботі розроблено вебзастосунок для автоматизації обліку доходів і витрат, централізованого управління фінансовими даними та підвищення прозорості фінансових рішень. Реалізований сервіс поєднує зручний користувацький інтерфейс із серверною логікою, що забезпечує коректну обробку операцій, збереження даних і контроль доступу в межах окремих робочих просторів.

У першому розділі сформовано теоретичну основу проєкту. Розглянуто особливості предметної області фінансового менеджменту, визначено ключові вимоги до сучасних вебсервісів та обґрунтовано вибір технологій TypeScript і React. Аналіз сучасних підходів до веброзробки дозволив сформувати функціональні й нефункціональні вимоги до системи.

Другий розділ присвячено проєктуванню вебсервісу. Сформовано архітектуру системи, визначено структуру клієнтської та серверної частин, спроєктовано модель бази даних і логіку взаємодії між основними сутностями. Запропоновані рішення орієнтовані на модульність, розділення відповідальностей і безпеку даних.

У третьому розділі продемонстровано ключові модулі програмної реалізації. Створено клієнтський інтерфейс на React із використанням компонентного підходу та механізмів управління станом. Серверна частина забезпечує виконання бізнес-логіки, автентифікацію користувачів і роботу з базами даних PostgreSQL та MongoDB. Інтеграція сервісу ExchangeRate.host дозволила реалізувати підтримку мультивалютності.

У результаті виконання роботи отримано повноцінний вебсервіс управління фінансами, придатний для практичного використання та подальшого розвитку.

ПЕРЕЛІК ПОСИЛАНЬ

1. Sommerville I. Software Engineering : 10th ed. Boston : Pearson Education, 2016. 816 p.

2. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner’s Approach : 8th ed. New York : McGraw-Hill Education, 2019. 976 p.

3. Martin R. C. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. Boston : Pearson, 2017. 432 p.

4. Banks E., Porcello E. Learning React: Modern Patterns for Developing React Apps. Sebastopol : O’Reilly Media, 2020. 350 p.

5. Silberschatz A., Korth H. F., Sudarshan S. Database System Concepts : 7th ed. New York : McGraw-Hill Education, 2019. 1376 p.

6. React Documentation. URL : https://react.dev/ (дата звернення: 12.09.2025).

7. TypeScript Documentation. URL : https://www.typescriptlang.org/docs/ (дата звернення: 12.09.2025).

8. Node.js Documentation. URL : https://nodejs.org/docs/latest/api/ (дата звернення: 12.09.2025).

9. PostgreSQL Documentation. URL : https://www.postgresql.org/docs/ (дата звернення: 18.09.2025).

10. API Documentation. *ExchangeRate.host*. URL : https://exchangerate.host/ (дата звернення: 18.09.2025).

11. OWASP Top Ten Web Application Security Risks. *OWASP Foundation*. URL : https://owasp.org/www-project-top-ten/ (дата звернення: 20.09.2025).