

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

ПОЯСНЮВАЛЬНА ЗАПИСКА

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

на тему:

«Вебзастосунок для планування сімейного бюджету»

Виконала студентка групи ІПЗ-21-1
ШЕВЕЛЬ Ольга Володимирівна

Керівник роботи:
САВІЦЬКИЙ Роман Святославович

Рецензент:
ЛОКТИКОВА Тамара Миколаївна

Житомир – 2025

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

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«26» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ШЕВЕЛЬ Ольга Володимирівна**

Керівник роботи: **САВІЦЬКИЙ Роман Святославович**

Тема роботи: **«Вебзастосунок для планування сімейного бюджету»**,
затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №61/с**

Вихідні дані для роботи: Об'єктом дослідження є процес управління сімейним бюджетом за допомогою вебтехнологій.

Предметом дослідження є методи та засоби розробки вебзастосунків для фінансового планування.

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

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Савіцький Р.С.	27.02.2025р.	27.03.2025р.
2	Савіцький Р.С.	27.02.2025р.	27.04.2025р.
3	Савіцький Р.С.	27.02.2025р.	27.05.2025р.

РЕФЕРАТ

Пояснювальна записка дипломного проєкту складається з трьох розділів, містить 54 таблиці, 55 рисунків, 39 джерел – загалом 97 сторінок.

Дипломний проєкт присвячений проєктуванню та розробці вебзастосунку для планування сімейного бюджету під назвою «Andarna».

Метою є розробка вебзастосунку, що дозволяє користувачам ефективно контролювати доходи та витрати, аналізувати фінансові тенденції, спільно управляти бюджетами та отримувати поради для покращення фінансового стану.

Об'єктом дослідження є процес управління сімейним бюджетом за допомогою вебтехнологій.

Предметом дослідження є методи та засоби розробки вебзастосунків для фінансового планування.

У першому розділі проведено аналіз предметної області, існуючих рішень, обґрунтовано вибір архітектури та технологічного стеку вебзастосунку.

У другому розділі розроблено вимоги до програмного забезпечення, включаючи варіанти використання, функціональні та нефункціональні вимоги, спроектовано базу даних та алгоритми роботи ключових модулів системи, а також описано реалізацію основних компонентів.

У третьому розділі було розроблено діаграму розгортання, описано структуру користувацького інтерфейсу та порядок взаємодії з ним, а також представлено результати тестування розробленого програмного продукту.

КЛЮЧОВІ СЛОВА: БЮДЖЕТ, ВЕБЗАСТОСУНОК, ВИТРАТИ, ТРАНЗАКЦІЇ, ФІНАНСОВЕ ПЛАНУВАННЯ

					ІПЗ.КР.Б – 121 – 25 – ПЗ				
Змн.	Арк.	№ докум.	Підпис	Дата					
Розроб.		Шевель О.В.			Вебзастосунок для планування сімейного бюджету	Літ.	Арк.	Аркушів	
Керівник		Савіцький Р.С.						168	
						Житомирська політехніка, група ІПЗ-21-1			
Рецензент		Локтікова Т.М.							
Зав. каф.		Вакалюк Т.А							
					Пояснювальна записка				

ABSTRACT

The explanatory note of the diploma project consists of three chapters, contains 54 tables, 55 figures, 39 sources – a total of 97 pages.

The diploma project is dedicated to the design and development of a web application for family budget planning called "Andarna".

Goal is to develop of a web application that allows users to effectively control income and expenses, analyze financial trends, jointly manage budgets, and receive advice to improve their financial condition.

Object of research is the process of managing a family budget using web technologies.

Subject of research includes methods and tools for developing web applications for financial planning.

In the first section, an analysis of the subject area and existing solutions was conducted, and the choice of architecture and technological stack for the future web application was justified.

In the second section, software requirements were developed, including use cases, functional and non-functional requirements, the database and algorithms for key system modules were designed, and the implementation of the main components was described.

In the third section, a deployment diagram was developed, the structure of the user interface and the order of interaction with it were described, and the results of testing the developed software product were presented.

KEYWORDS: BUDGET, WEB APPLICATION, EXPENSES, TRANSACTIONS, FINANCIAL PLANNING

ЗМІСТ

РЕФЕРАТ.....	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП.....	8
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ	11
1.1 Постановка задачі	11
1.2 Аналіз предметної області	13
1.3 Аналіз існуючих рішень	15
1.4 Вибір архітектури.....	21
1.5 Вибір інструментальних засобів та технологій	24
Висновки до першого розділу	31
РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	33
2.1 Варіанти використання	33
2.2 Аналіз системних та нефункціональних вимог, проектування бази даних системи	43
2.3 Проектування та реалізація алгоритмів роботи додатку....	49
2.4 Реалізація додатку планування сімейного бюджету.....	56
Висновок до другого розділу	68
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З СИСТЕМОЮ ДОДАТКУ.....	70
3.1 Встановлення та налаштування системи	70
3.2 Структура інтерфейсу та порядок взаємодії із застосунком.....	72
3.3 Тестування роботи програмного продукту	87

Висновки до третього розділу	91
ВИСНОВКИ	93
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	95
ДОДАТКИ	99

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – база даних

СКБД – система керування базами даних

ШІ – штучний інтелект

API – Application Programming Interface

CRUD – Create Read Update Delete

DOM – Document Object Model

JSON – JavaScript Object Notation

JWT – JSON Web Token

ORM – Object-Relational Mapping

ВСТУП

Опанування фінансової грамотності та ефективне управління особистими фінансами набувають критичного значення для добробуту кожної людини та сім'ї, зважаючи на швидкі економічні зміни та постійну потребу в оптимізації ресурсів. Зростання кількості фінансових операцій, необхідність планування витрат та досягнення довгострокових фінансових цілей вимагають зручних, доступних та ефективних рішень для контролю сімейного бюджету. Традиційні методи ведення бюджету часто є трудомісткими та не завжди дозволяють оперативно реагувати на зміни у фінансовому стані.

Саме тому застосування цифрових інструментів, зокрема вебзастосунків, у сфері фінансового планування відкриває нові можливості для користувачів. Використання спеціалізованих вебзастосунків дозволяє не лише фіксувати доходи та витрати, але й аналізувати фінансові потоки, прогнозувати майбутні потреби, встановлювати та відслідковувати досягнення фінансових цілей, а також приймати обґрунтовані фінансові рішення. Вебзастосунки для планування сімейного бюджету забезпечують інтерактивність, мобільність і доступність, що робить їх потужним інструментом для фінансового управління в руках сучасних користувачів.

Актуальність розробки вебзастосунку для планування сімейного бюджету також підсилюється такими факторами як: потреба у спільному управлінні фінансами, адже багато сімей практикують спільне ведення бюджету, що вимагає інструментів, які підтримують багатокористувацький доступ та спільне планування; інтеграція з банківськими сервісами, так як із розвитком онлайн-банкінгу, користувачі очікують можливості синхронізації даних з банківськими рахунками для автоматизації обліку та уникнення неточностей; попит на персоналізовані поради, адже використання технологій штучного інтелекту для аналізу фінансових даних та надання персоналізованих рекомендацій щодо покращення витрат стають дедалі

частішими; виникнення складнощів у фінансовому плануванні через проблемами нестабільності доходів, непередбачуваними витратами, низьким рівнем фінансової грамотності, імпульсивними покупками, а також впливом інфляції та економічних криз, що ускладнює ефективне управління бюджетом.

Метою кваліфікаційної роботи є проєктування та розробка вебзастосунку для планування сімейного бюджету, що надає можливість ефективно контролювати доходи та витрати, аналізувати фінансові тенденції, ділитися бюджетами з іншими користувачами та отримувати допомогу із покращенням фінансової ситуації, в тому числі за допомогою штучного інтелекту.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

- провести аналіз предметної області, існуючих рішень у сфері фінансового планування та визначити ключові вимоги до майбутнього вебзастосунку;
- обґрунтувати вибір архітектури програмного забезпечення, зокрема клієнт-серверної архітектури та підходу монорепозиторію;
- визначити та обґрунтувати вибір інструментальних засобів та технологій для розробки клієнтської та серверної частин застосунку, а також для інтеграції сторонніх сервісів;
- розробити детальні функціональні та нефункціональні вимоги до системи, визначити варіанти використання;
- спроектувати структуру бази даних для зберігання інформації про користувачів, бюджети, транзакції, категорії, накопичувальні рахунки, підписки та рекомендації;
- розробити та реалізувати основні модулі вебзастосунку, включаючи управління бюджетами та транзакціями, категоризацію витрат, управління накопичувальними акаунтами, систему підписок,

функціонал запрошення користувачів, візуалізацію статистики та генерацію фінансових порад за допомогою ІІІ;

- розробити користувацький інтерфейс, що забезпечує зручну та інтуїтивно зрозумілу взаємодію з системою;
- провести тестування розробленого програмного продукту для перевірки його відповідності заявленим вимогам та коректності роботи.

Об'єктом дослідження є процес управління сімейним бюджетом за допомогою сучасних вебтехнологій.

Предметом дослідження є методи, засоби та технології розробки багатфункціонального вебзастосунку для фінансового планування, що включає можливості спільного ведення бюджету, аналізу даних та інтеграції зі сторонніми сервісами.

Практичне застосування одержаних результатів полягає у створенні вебзастосунку «Andarna», який є готовим до використання інструментом для ефективного управління сімейним бюджетом. Розроблений програмний продукт надає користувачам можливість взяти під контроль свої фінанси, детально відстежувати доходи та витрати, а також аналізувати фінансові потоки для прийняття обґрунтованих рішень.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Постановка задачі

Розробити програмне рішення для обліку та планування сімейного бюджету у вигляді вебзастосунку, що дозволить користувачам ефективно взаємодіяти із фінансовими даними, пришвидшити аналіз та планування економічних цілей та ділитися ними із іншими користувачами.

Програмне рішення повинне містити наступні складові:

- користувачі мають можливість створювати бюджети, додавати та видаляти транзакції, що відображатимуть обіг фінансів, а також можливість додавати інших користувачів до керування бюджетом;
- користувач може створювати та редагувати накопичувальні акаунти, а також вносити дані про кошти, які були відкладені на нього і переглядати прогрес по досягненню цілі;
- користувачеві представлено детальні та різнопланові графіки, що відображають динаміку змін стану бюджету та витрати по різних категоріях та за різні періоди часу;
- передбачення можливості для користувачів отримання поради від штучного інтелекту на основі статистики щодо покращення фінансової ситуації та кращого планування;
- користувачу надано можливість імпортування даних із застосунку онлайн банкінгу.

Процес розробки застосунку охоплює низку послідовних етапів, які тісно пов'язані між собою та потрібні для створення інтуїтивно зрозумілого, функціонального та безпечного програмного забезпечення.

На першому етапі потрібно спроектувати архітектуру користувацької (frontend) та серверної (backend) компонент. Процес передбачає визначення взаємодії між клієнтською частиною, що працює з інтерфейсом користувача, та серверною частиною, яка обробляє запити, зберігає дані й забезпечує бізнес-

логіку. Потрібно розробити модель даних, обрати структуру API та технологічний стек, що забезпечить масштабованість і стабільну роботу в умовах зростаючого навантаження.

Другим етапом є створення дизайну додатку. Він включає вибір загальної концепції зовнішнього вигляду, а також логіки взаємодії з інтерфейсом. Розробка дизайну здійснюється з урахуванням сучасних принципів UX/UI, що забезпечують інтуїтивне сприйняття інтерфейсу кінцевими користувачами. Окрему увагу варто приділити реалізації користувацьких компонентів, таких як кнопки, поля введення, випадаючі списки, тощо. Після цього виконати компонування окремих елементів у цілісні сторінки.

На третьому етапі реалізувати функціональну частину застосунку, що включає налаштування БД, реалізацію бізнес-логіки та підключення сторонніх сервісів. База даних зберігатиме інформацію про користувачів, їхні фінансові портфелі, транзакції, категорії витрат, тощо. Вибір системи управління БД має бути здійснено з урахуванням вимог до продуктивності та безпеки. Реалізація бізнес-логіки охоплює написання коду на серверній частині застосунку. Сервер має забезпечити обробку запитів, валідацію даних, розрахунки, фільтрацію записів, тощо.

Останнім етапом є тестування. Основну увагу слід зосередити на ручному (мануальному) тестуванні, яке дозволяє виявити логічні та функціональні помилки, а також помилки інтерфейсу. У процесі тестування перевіряється коректність виконання основних сценаріїв роботи застосунку. Для системного підходу до тестування розробляються тест-кейси — структуровані документи, що описують вхідні умови, кроки виконання, очікувані результати та фактичну поведінку системи.

Результатом виконання поставленої задачі є вебзастосунок для керування сімейним бюджетом, що складається із мультикористувацького додатку із клієнтською частиною та сервером для обробки запитів і збереження

даних у базі. Ключовими є модулі управління бюджетами, накопиченнями, візуалізація статистики та генерація порад для покращення планування.

1.2 Аналіз предметної області

Фінансове планування та ведення сімейного бюджету є важливими аспектами життя кожної родини, що дозволяють забезпечити стабільність, досягнути довгострокових цілей та уникнення фінансових труднощів. Ефективне управління доходами і витратами допомагає створити резервний фонд, здійснювати великі покупки, інвестувати у майбутнє та забезпечувати комфортне життя.

Фінансове планування включає оцінку доходів, витрат, заощаджень та інвестицій. Доходи можуть надходити з різних джерел, таких як заробітна плата, бонуси, пасивний дохід тощо. Витрати поділяються на обов'язкові (житло, комунальні послуги, продукти харчування) та додаткові (відпочинок, розваги, покупки). Важливим елементом є формування фінансових цілей, що допомагає керувати бюджетом з урахуванням майбутніх потреб.

Однією з головних труднощів у плануванні бюджету є нестабільність доходів, що ускладнює прогнозування витрат. Також проблемою можуть стати непередбачувані витрати, низький рівень фінансової грамотності та імпульсивні покупки. Інфляція та економічні кризи також суттєво впливають на економічне становище і вимагають відповідної адаптації бюджету.

У сучасній фінансовій практиці існують різні методи управління сімейним бюджетом. Наведено декілька основних підходів до розподілу коштів

Метод «50/30/20» передбачає розподіл сімейного доходу на певні категорії витрат із встановленням фіксованих відсоткових часток для кожної з них. Наприклад, 50% доходу спрямовується на основні потреби (житло, харчування, комунальні послуги), 30% — на особисті бажання (розваги, хобі),

а 20% — на заощадження та інвестиції. Такий розподіл допомагає забезпечити баланс між поточними витратами та майбутніми накопиченнями [1].

Метод «конвертів» полягає у фізичному або віртуальному розподілі грошових коштів на окремі категорії витрат, кожна з яких представлена окремим «конвертом». Наприклад, створюються конверти для харчування, транспорту, розваг тощо, і в кожен з них вкладається сума, призначена для відповідної категорії на певний період. Витрачання коштів дозволяється лише в межах суми, наявної в конкретному конверті, що сприяє контролю над витратами та запобігає перевищенню бюджету [2].

Метод нульового бюджету полягає у детальному плануванні, при якому кожна одиниця доходу (наприклад, кожна гривня) призначається на конкретну статтю витрат або заощаджень. Метою є досягнення нульового залишку нерозподілених коштів, що забезпечує повний контроль над фінансами та усвідомлене управління кожною витратою [3].

Застосування цих методів дозволяє родинам ефективно планувати свої фінанси, забезпечувати фінансову стабільність та досягати поставлених економічних цілей.

Для складання бюджету в сім'ї вирізняють такі 3 основні підходи:

- спільний бюджет передбачає об'єднання всіх доходів та спільний розподіл витрат;
- роздільний бюджет означає, що кожен член сім'ї керує своїми доходами окремо. Усі спільні витрати розподіляються порівну;
- змішаний підхід поєднує елементи обох методів, що дає змогу забезпечити гнучкість у розпорядженні коштами.

Існують мобільні додатки та програмні рішення, які дозволяють автоматизувати контроль витрат, створювати фінансові звіти та прогнозувати майбутні фінансові потреби. Використання інструментів допомагає покращити фінансову дисципліну та зробити управління грошима більш

ефективним. Ключовою функцією додатків для сімейного бюджету є можливість створення спільного бюджету та надання доступу до нього іншим користувачам.

Однією із важливих можливостей для додатків, зокрема в Україні, є синхронізація із банківськими даними, так як досить розвиненим є онлайн банкінг. Це дозволяє уникнути неточностей із веденням обліку витрат та економити час при введенні початкових даних.

Також варто врахувати зростаючу популярність та розповсюдженість використання штучного інтелекту у сферах надання цифрових послуг. Це потужний інструмент для пошуку, аналізу та швидкої обробки даних. Маючи доступ до величезних об'ємів інформації, інструменти ШІ, наприклад ChatGPT [4], надають доволі точну та корисну інформацію, в тому числі і у сфері витрат та фінансового планування. Це може значно пришвидшити роботу із статистикою, аналізом та плануванням витрат.

1.3 Аналіз існуючих рішень

Було проведено аналіз представлених рішень для обліку та планування бюджету. Розглянуто деякі із них, а також порівняно наявний функціонал, що допоможе при плануванні реалізації «Andarna».

Варто виділити такі застосунки як «Honeydue» [5] – мобільний додаток для ведення спільного бюджету пар. Надає можливість відстежувати спільні та індивідуальні витрати, встановлювати бюджети, отримувати нагадування про рахунки та спілкуватися в додатку щодо фінансових питань. Ключовою функцією є спільне управління фінансами, що дозволяє партнерам синхронізувати банківські рахунки, кредитні картки, позики та інвестиції в одному місці. Включено функцію контролю доступу, аби кожен із партнерів міг обрати якими фінансовими даними поділитися. Також додаток має вбудований чат, що дозволяє обговорювати транзакції та інші фінансові питання безпосередньо в додатку. HoneyDue є безкоштовним для завантаження

та використання та не передбачає платної версії чи функцій. Користувачі можуть за бажанням надсилати «чайові» розробникам.

На рис. 1.1 наведено вигляд головного екрану мобільного додатку HoneyDue.

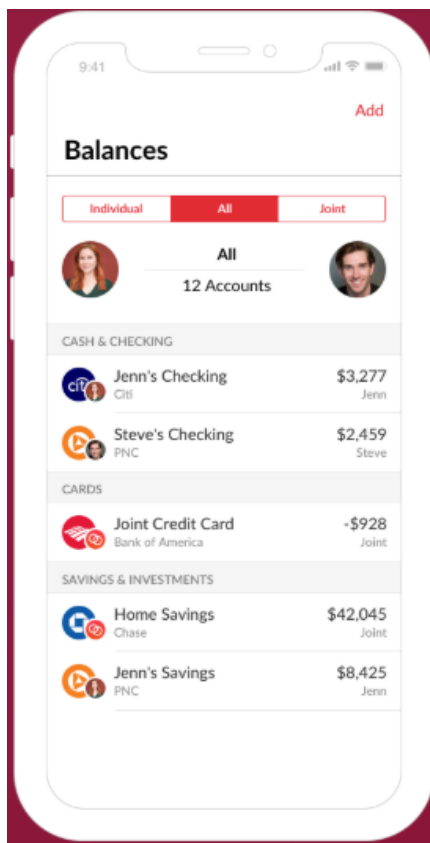


Рисунок 1.1 – Головний екран додатку HoneyDue

«PocketGuard» [6] – доволі масштабний додаток для фінансового планування, що робить акцент на залишку фінансів після витрачення коштів, допомагає користувачам відстежувати витрати, створювати бюджети, планувати погашення боргів та досягати поставлених фінансових цілей. Також надається детальний огляд фінансового стану та грошових потоків. Має функцію автоматичного розбиття витрат на категорії, дозволяє синхронізуватися із банківськими акаунтами та встановлювати та відстежувати цілі для накопичень. Особливою функцією є відображення суми коштів, яку безпечно витратити, зважаючи на наявні рахунки, цілі та створені

бюджети. PocketGuard пропонує платну підписку «PocketGuard Plus», яку можна придбати щомісячно або на рік.

На рис. 1.2 наведено вигляд сторінки додатку PocketGuard, що містить інформацію про витрати в межах бюджету.

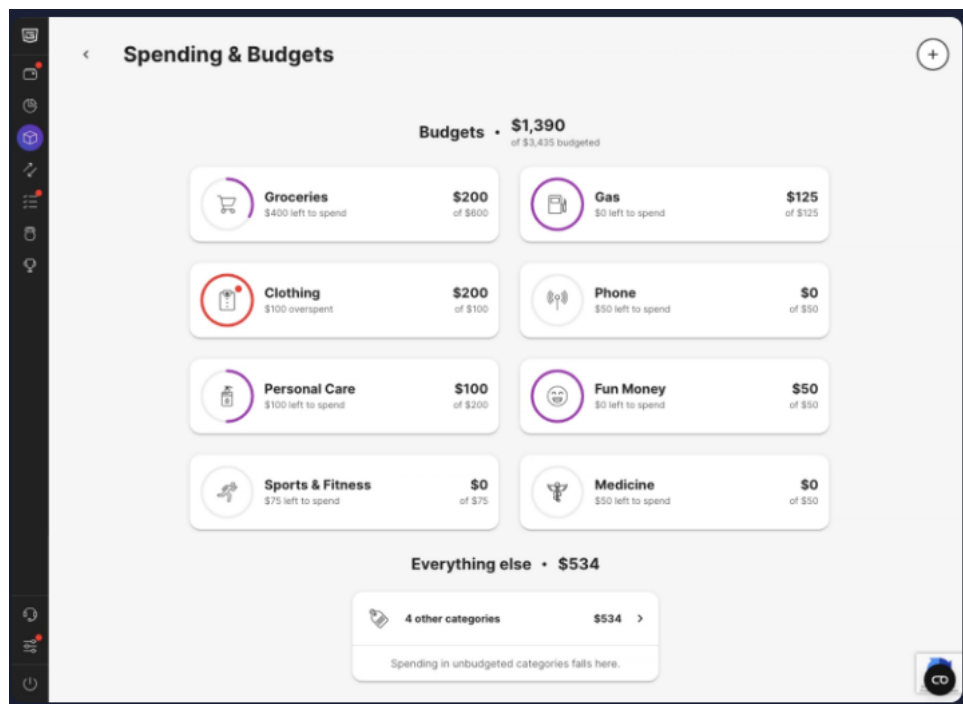


Рисунок 1.2 – Сторінка витрат в межах бюджету додатку PocketGuard

«Spendee» [7] – додаток для відстеження витрат із акцентом на візуальну складову, що використовується для персонального та сімейного бюджетування. Добре інтегрується із банківськими акаунтами, що передбачає підключення банківських рахунків та кредитних карток для синхронізації транзакцій. Функція розбиття на категорії дозволяє спростити управління завдяки автоматичній класифікації транзакцій. Можливості створення декількох «гаманців» для накопичень, створення необмеженої кількості бюджетів та ведення спільних «гаманців» з іншими користувачами забезпечує користувачів необхідними інструментами для ефективного відстеження та планування фінансів. Spendee пропонує безкоштовну версію та платну підписку «Spendee Premium» яку можна придбати щомісячно або на рік. Також є опція «Spendee Lifetime Premium» що забезпечить постійний доступ до преміум функцій.

На рис. 1.3 наведено вигляд головної сторінки додатку Spendee.

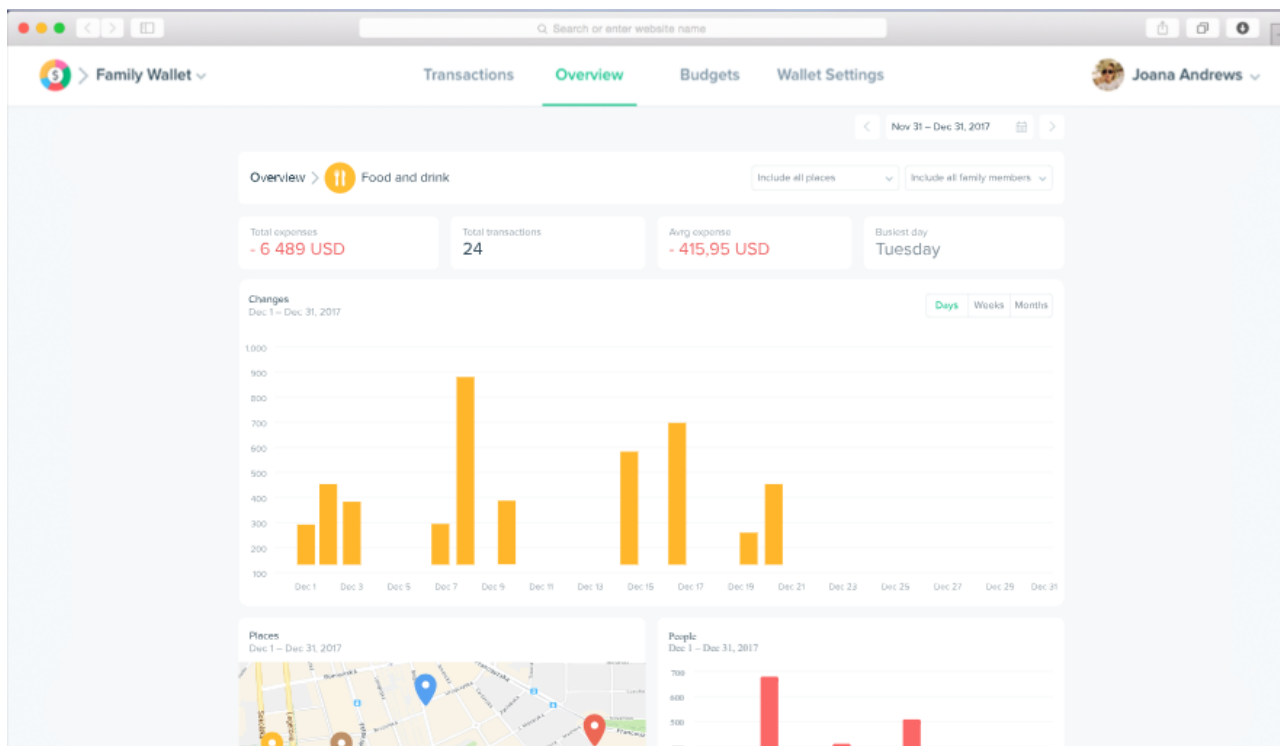


Рисунок 1.3 – Головна сторінка додатку Spendee

Для аналізу було враховано основний функціонал представлених додатків та окреслено переваги та недоліки наявних реалізацій та рішення, яке буде реалізовано в цій роботі.

Результати порівняння із розроблюваним застосунком Andarna, наведено в таблиці 1.1.

Таблиця 1.1

Відомі технічні рішення

Функціонал	HoneyDue	PocketGuard	Spendee	Andarna
Синхронізація банківськими рахунками	+	+	+	+ –
Категоризація витрат	+	+	+	+
Керування бюджетами	+	+	+	+
Підтримка заощаджень	–	+	+	+
Функція чату	+	–		–
Можливість вести спільний бюджет	+	–	+	+
Отримання поради від ІІІ	–	–	–	+

Продовження таблиці 1.1

Можливість запросити декількох користувачів до бюджету	–	–	+	+
Перегляд статистики	+	+	+	+
Підтримка декількох валют	–	–	+	+

Після проведення порівняльного аналізу можна виділити та розширити функціональні вимоги до майбутнього застосунку. При розробці функціональних вимог було вирішено поділити їх на логічні модулі із певним набором функціоналу. Загальну систему вимог наведено в табл. 1.2. Детальний опис вимог наведено в табл. 1.3.

Таблиця 1.2

Загальна модель вимог

Назва	ID вимоги	Пріоритет	Ризики
Авторизація та управління даними користувача			
Реєстрація користувача	FR-1	Високий	Середні
Авторизація користувачів	FR-2	Високий	Середні
Редагування профілю	FR-3	Середній	Низькі
Видалення профілю	FR-4	Середній	Низькі
Управління бюджетами			
Створення бюджету	FR-5	Високий	Середні
Додавання транзакцій	FR-6	Високий	Середні
Видалення транзакцій	FR-7	Середній	Низькі
Створення категорій витрат	FR-8	Середній	Низькі
Додавання категорій до транзакцій	FR-9	Високий	Середні
Перегляд статистики	FR-10	Середній	Низькі
Запрошення інших користувачів	FR-11	Середній	Низькі
Управління накопиченнями			
Створення накопичувального акаунту	FR-12	Середній	Низькі
Поповнення накопичень	FR-13	Середній	Низькі

Продовження таблиці 1.2

III			
Отримання поради від штучного інтелекту	FR-14	Високий	Середні
Інтеграція з онлайн банкінгом			
Імпортування банківських даних	FR-15	Середній	Низькі
Робота з платною підпискою			
Оформлення підписки	FR-16	Середній	Низькі
Скасування підписки	FR-17	Середній	Низькі

Таблиця 1.3

Перелік функціональних вимог

ID вимоги	Назва та опис
FR-1	<i>Реєстрація користувача</i> Можливість нових користувачів зареєструватись в системі
FR-2	<i>Авторизація користувачів</i> Можливість для вже існуючих користувачів здійснити вхід до системи
FR-3	<i>Редагування профілю</i> Можливість для користувачів, що вже мають профіль у додатку, заповнити особистий профіль новими даними
FR-4	<i>Видалення профілю</i> Можливість для користувачів видалити власний обліковий запис з усіма даними
FR-5	<i>Створення бюджету</i> Створення сутності бюджету для користувача
FR-6	<i>Додавання транзакцій</i> Додавання даних про надходження та витрати до бюджету
FR-7	<i>Видалення транзакцій</i> Видалення даних про фінансові операції із бюджету
FR-8	<i>Створення категорій витрат</i> Додавання користувацьких категорій витрат
FR-9	<i>Додавання категорій до транзакцій</i> Категоризація транзакцій за стандартними (вже наявними в системі) та користувацькими категоріями
FR-10	<i>Перегляд статистики</i> Можливість перегляду користувачем графіків змін фінансової ситуації власного бюджету за певний період часу
FR-11	<i>Запрошення інших користувачів</i> Можливість додавання іншого користувача до бюджету, а також надання іншому користувачеві можливості вносити зміни до цього бюджету
FR-12	<i>Створення накопичувального акаунту</i> Можливість створення сутності накопичувального акаунту для відкладання коштів для певних цілей

Продовження таблиці 1.3

FR-13	<i>Поповнення накопичень</i> Можливість додавання транзакції, що відображатиме перенаправлення коштів до накопичувального акаунту
FR-14	<i>Отримання поради від штучного інтелекту</i> Можливість відправки даних до ШІ про стан бюджету та отримання поради, що допоможе оптимізувати витрати та розподіл коштів
FR-15	<i>Імпортування банківських даних</i> Можливість додавання інформації про накопичення із банківського рахунку користувача (Monobank)
FR-16	<i>Оформлення підписки</i> Можливість оформлення і оплати платної підписки у вебзастосунку, що відкриє користувачеві додаткові можливості додатку
FR-17	<i>Скасування підписки</i> Можливість скасування платної підписки в будь-який час

Проведений порівняльний аналіз аналогів та виділення основних функціональних вимог до вебзастосунку дозволяють чітко зрозуміти на які модулі повинні бути присутні та перейти до технічного проєктування.

1.4 Вибір архітектури

При виборі архітектури для вебзастосунку було проаналізовано найбільш актуальні види архітектур, а саме клієнт-серверна [8] та мікросервісна [9]. В процесі було досліджено основні переваги та недоліки кожного варіанту.

Клієнт-серверна архітектура – це модель взаємодії в інформаційних системах, де клієнтські пристрої (клієнти) надсилають запити на сервер, а той в свою чергу оброблює їх і надсилає відповідь. Цей процес постійно повторюється. Клієнтів в свою чергу може бути багато. Приклад клієнт-серверної архітектури наведено на рис. 1.4.

Перевагами даної архітектури є відносна простота розробки і централізоване управління. Основним недоліком можна назвати відмовостійкість, адже якщо сервер вийде з ладу – вебзастосунок повністю перестане працювати.

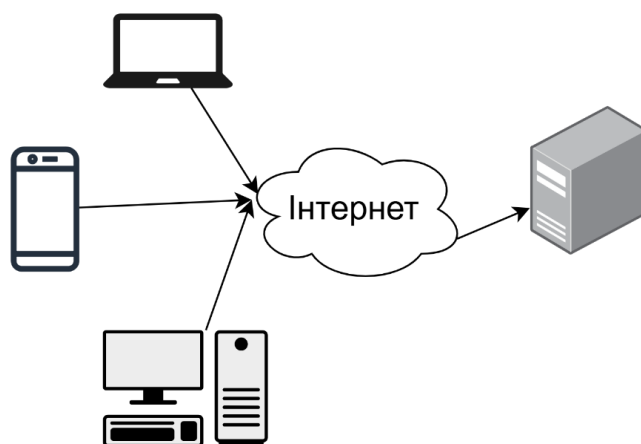


Рисунок 1.4 – Клієнт-серверна архітектура

Мікросервісна архітектура є більш складною за рахунок того, що бізнес-логіка додатку розбита на певну кількість сервісів, що фактично є окремими сервісами, часто з власною базою даних тощо. Ці сервіси мають можливість спілкуватися між собою тим самим виконуючи складні операції.

Перевагами мікросервісної архітектури є відмовостійкість та масштабованість, адже розробнику досить легко додавати сервіси та впроваджувати нову логіку, і при цьому якщо якийсь з сервісів вийде з ладу – це не зачепить роботу інших сервісів. З недоліків можна перерахувати складність у розгортанні та підтримці, а також важливість ефективної мережевої взаємодії. На рис. 1.5 наведено приклад мікросервісної архітектури.

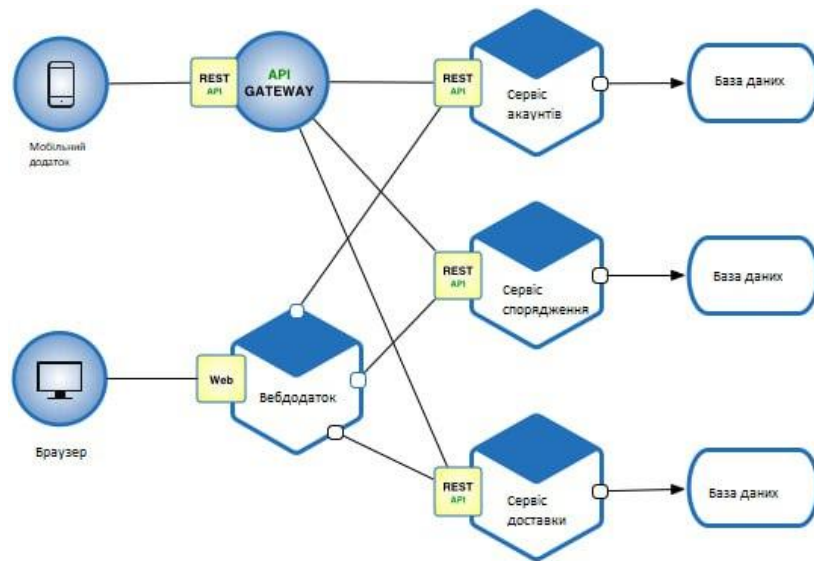


Рисунок 1.5 – Мікросервісна архітектура

У результаті, для розробки майбутнього вебзастосунку було обрано клієнт-серверну архітектуру, де клієнтом буде frontend-додаток, що запускається у браузері і взаємодіє з Node.js сервером. Сервер може звертатися до інших частин системи такі як база даних, кеш, сторонні API тощо.

Крім того, варто одразу визначитися з підходом до розробки програмного забезпечення. Серед варіантів розглядалося два основних, а саме полірепозиторій, або монорепозиторій.

Полірепозиторій – код фронтенду та бекенду розробляється і зберігається у різних репозиторіях і вони технічно на етапі розробки ніяк не пов’язані [10]. Основною проблемою такого підходу є дублювання коду і можливі помилки типізації та роботи програми після оновлень певних частин коду. Наприклад, коли в API бекенду було зроблено певні зміни, а фронтенд про них не знає, і знайти помилку можливо лише запустивши додаток і все протестувавши.

Монорепозиторій – це складніша структура, що фактично являє собою підхід до розробки, коли фронтенд та бекенд знаходяться в одному репозиторії. Зазвичай при цьому підході контракти на API, класи, допоміжні функції тощо виносяться в додаткові проєкти, що можуть використовуватися як фронтендом

так і бекендом [10]. Це вирішує основну проблему попереднього підходу. Основною проблемою монорепозиторію можна назвати велику складність налаштування проекту, а також розгортання.

Після аналізу вимог до функціоналу застосунку, що буде розроблено, було вирішено обрати монорепозиторій як підхід для розробки, адже це дозволить легко відловлювати помилки у коді під час розробки і досягти значно меншого дублювання коду.

При виборі архітектури для бекенд частини практика є використання слоїстої архітектури [11] є досить поширеною та передбачає наступні шари:

- маршрутизації, де визначається куди направити обробку запиту;
- контролерів, де валідуються дані відправлені з клієнта;
- сервісів, де знаходиться бізнес логіка додатку, яка є ізольованою від бази даних;
- репозиторіїв, тобто абстрактних сутностей, що описують таблиці бази даних у вигляді класів і надають методи для роботи з базою даних у форматі звичному програмісту;
- бази даних, що відповідає за збереження даних.

За потреби зміни одного з шарів системи, інші шари не потребуватимуть змін. Наприклад, якщо потрібно змінити базу даних на іншу – шар репозиторіїв залишиться незмінним, тому код додатку буде так само функціонувати, тощо.

1.5 Вибір інструментальних засобів та технологій

При виборі інструментів та технологій для реалізації додатку було розглянуто наступні рішення. Для написання вебдодатків було вирішено здійснити вибір між такими мовами програмування як JavaScript та TypeScript.

JavaScript – стандартна мова для веброзробки, що підтримується всіма браузерами та має велику екосистему [12]. Її перевагами є гнучкість, відносно швидке вивчення і широка підтримка. Проте відсутність статичної типізації може ускладнити масштабування та підтримку великих проєктів.

TypeScript можна назвати надбудовою JavaScript, яка додає статичну типізацію, інтерфейси та покращене автодоповнення в редакторах коду [13]. Основними перевагами TypeScript є вища надійність коду, спрощення коригування коду та уникнення логічних та інтеграційних помилок при масштабуванні проєкту. Серед недоліків можна назвати потребу додаткової конфігурації та компіляції.

Було обрано TypeScript, зважаючи на його широку підтримку, простішу інтеграцію з різними фреймворками та наявність типів для всіх основних бібліотек.

Для створення фронтенд частини було розглянуто наступні популярні рішення: Vue.js, Angular, React.js.

Vue.js — це прогресивна JavaScript-бібліотека, що має доволі простий синтаксис та надає зручний API [14]. Перевагою Vue є легкість вивчення та налаштування, а також ефективна робота у невеликих проєктах. Водночас, масштабування великих застосунків із Vue може вимагати додаткових архітектурних рішень, а екосистема поступається за розвиненістю альтернативам.

Angular — це повноцінний фронтенд-фреймворк від Google, який потребує чіткої структури, забезпечує широкий інструментарій, вбудовану підтримку RxJS та модульність [15]. Перевагою є масштабованість, розширений CLI та чітка організація коду. Проте, через складність синтаксису та велике навантаження на браузер, Angular може виявитися надмірним для немасштабних застосунків.

React.js — бібліотека для побудови інтерфейсів від Meta, яка базується на компонентному підході, використовує віртуальний DOM для побудови сторінок та має широку підтримку [16]. Серед переваг React — продуктивна та швидка робота, величезна спільнота користувачів та розробників, значна кількість бібліотек та гнучкість у побудові архітектури. До недоліків можна

віднести потребу самостійного обирання додаткових інструментів (наприклад для роутингу, керування станом) та відносно високу складність при побудові великих застосунків через відсутність вимог до чіткої структуризації.

Вирішено обрати React.js, зважаючи на його гнучкість, розвинену екосистему та широке використання у вебзастосунках.

Для реалізації елементів інтерфейсу застосунку розглянуто наступні бібліотеки: Material UI, Chakra UI, Ant Design.

Material UI — бібліотека із набором компонентів, для реалізації дизайну від Google [17]. Її перевагою є якісна документація, адаптивність і велика кількість готових компонентів, проте не завжди узгоджується з індивідуальними вимогами дизайну.

Chakra UI — це бібліотека, орієнтована на доступність та зручність у стилізації компонентів [18]. Вона спрощує розробку і швидко інтегрується з React. До недоліків належать обмежена кількість готових компонентів і менш розвинена система таблиць і форм.

Ant Design — це бібліотека компонентів, що активно використовується в бізнес-застосунках, з підтримкою форм, таблиць, модальних вікон та інтеграції іконок [19]. Основними перевагами є великий набір функціональних компонентів, сучасний вигляд і зручна інтеграція з екосистемою React. З недоліків — порівняно великий розмір бібліотеки та незначні труднощі з кастомізацією.

Для реалізації інтерфейсу було обрано Ant Design через його широкий набір компонентів, адаптованість до контексту фінансового застосунку та зручність використання у зв'язці з React та TypeScript.

Для організації проєкту та реалізації монорепо розглянуто наступні рішення: Turborepo, Nx.

Turborepo — інструмент від Vercel, що підтримує кешування, паралельну збірку, та добре інтегрується з Next.js [20]. Його перевага — продуктивність і

простота налаштування. Водночас проєкт відносно новий, і деякі функції мають обмежену документацію чи вимагають додаткових налаштувань для складних сценаріїв.

Nx — це повноцінний фреймворк для організації монорепозіторіїв, що підтримує перевірку залежностей, генератори коду та повну інтеграцію з React, Nest.js, TypeScript [21]. Основними перевагами Nx є масштабованість, продуктивність, велика кількість готових рішень. Недоліки полягають у початковій складності освоєння та деякому перевантаженні проєкту метаданими.

Було обрано Nx, зважаючи на потребу в організації фронтенду й бекенду в межах одного репозіторію, ефективну інтеграцію з Nest.js і React та гнучкий контроль залежностей між модулями.

Для реалізації автентифікації користувачів були розглянуті наступні рішення: Auth0, Firebase Authentication.

Auth0 — це комерційний сервіс для управління ідентифікацією, який забезпечує високу безпеку, інтеграцію з багатьма сторонніми провайдерами і готові інтерфейси для входу [22]. Перевага — простота впровадження й масштабування. Основний недолік — висока вартість при зростанні кількості користувачів.

Firebase Authentication — це готове рішення від Google, яке забезпечує такі способи авторизації як email/password, OAuth, єдиний вхід (SSO), а також зручний функціонал для налаштування фронтенду [23]. Серед переваг — швидке впровадження, масштабованість і безпека. До недоліків належить менша гнучкість у кастомізації механізмів входу, а також залежність від хмарного провайдера Google.

Було обрано Firebase Authentication, зважаючи на його просту інтеграцію з React та підтримку необхідних способів автентифікації.

Для реалізації бекенд-логіки були розглянуті такі фреймворки: Express.js, Nest.js.

Express.js — мінімалістичний фреймворк для Node.js, що надає функції для впровадження HTTP-сервісів [24]. Основними перевагами є простота, гнучкість і загальна підтримка. Із недоліків – Express не надає структурованого підходу до архітектури додатків і вимагає ручної організації залежностей та сервісів.

Nest.js — прогресивний Node.js фреймворк, який реалізує принципи OOP, DI та модульність, використовуючи TypeScript [25]. Його переваги — структурована архітектура, зручна робота з модулями, інтегрованість з TypeORM, OpenAPI, тощо. До недоліків можна віднести вищий рівень підготовки для новачків.

Було обрано Nest.js, оскільки він забезпечує масштабовану, модульну архітектуру, повністю сумісний із TypeScript, а також ефективно працює з PostgreSQL та Redis.

При виборі БД були розглянуті наступні рішення: MongoDB, MySQL, PostgreSQL.

MongoDB — документно-орієнтована NoSQL база даних [26]. Вона забезпечує гнучкість у структурі даних і є легкомасштабованою. Основним недоліком є відсутність транзакцій та складність виконання складних SQL-запитів.

MySQL — популярна реляційна СКБД, що забезпечує стабільну та надійну роботу [27]. Підтримує більшість SQL-функцій, але має обмежені можливості в роботі з JSON, індексацією та паралельними операціями порівняно з PostgreSQL.

PostgreSQL — сучасна реляційна СКБД з відкритим кодом, яка підтримує транзакції, розширення, індекси, роботу з JSON та потужні

можливості аналітики [28]. Її недоліки — складніша конфігурація і трохи нижча швидкість запису у порівнянні з MongoDB.

Було обрано PostgreSQL, зважаючи на потребу у реляційній моделі з чіткою схемою, підтримці транзакцій, гнучких SQL-запитах і інтегрованість з TypeORM.

Для взаємодії з базою даних розглядалися: Sequelize, Prisma, TypeORM.

Sequelize — ORM для Node.js з підтримкою MySQL, PostgreSQL, SQLite [29]. Проста у використанні, але менш зручна у TypeScript-середовищі та обмежена у роботі зі складними зв'язками.

Prisma — ORM з власною мовою розмітки моделей, що має зручну документацію та чудову підтримку TypeScript [30]. Водночас вона не є класичною ORM і має деякі обмеження у складних реляційних сценаріях.

TypeORM — ORM, орієнтована на TypeScript, із підтримкою декораторів та використанням підходів для доступу до БД всередині визначених моделей (Active Record) або всередині репозиторіїв (Data Mapper) [31]. Основні переваги — сумісність з Nest.js, потужна інтеграція з TypeScript і гнучка конфігурація. Недоліки — повільний розвиток проєкту і застаріле API в окремих випадках.

Було обрано TypeORM, зважаючи на його глибоку інтеграцію з Nest.js, можливість використання декораторів та синхронізації з БД у реальному часі.

Для інтеграції платіжної системи були розглянуті: PayPal, Stripe.

PayPal — популярний сервіс з глобальним охопленням [32]. Має просту інтеграцію, але менш зручний у кастомізації інтерфейсу й погано масштабується для складних фінансових сценаріїв.

Stripe — платіжна система, орієнтована на розробників, з потужним API, підтримкою підписок, webhooks та чудовою документацією [33]. Недоліком є вища комісія порівняно з локальними сервісами та є потреба додаткових знань для первинного налаштування.

Було обрано Stripe, зважаючи на просту інтеграцію з Node.js, розширену функціональність для роботи з підписками, а також якісну документацію.

Для реалізації функціоналу, пов'язаного з генерацією порад від ШІ, розглядалися такі моделі OpenAI API [34]: GPT-3.5 turbo, GPT-4 (стандартна) та GPT-4-turbo.

GPT-3.5 Turbo — мовна моделі GPT, що має меншу вартість, характеризується швидким часом відгуку, але має слабку здатність до аналізу та обробки складних даних, проводить поверхневу оцінку даних та має обмеження в пам'яті.

GPT-4 (стандартна) — покращена модель, що надає обґрунтовані відповіді та коректну роботу зі структурованими даними. Проте ця модель є дорожчою, а також недоступна в OpenAI API як окрема модель та є складовою моделі GPT-4-turbo.

GPT-4-turbo – одна з найновіших моделей, що характеризується великим обсягом оброблюваної інформації, добре працює з табличними даними та JSON-форматом та правильно виділяє закономірності.

Було вирішено обрати модель GPT-4-turbo, з огляду на стабільність, високу якість генерації текстів та коректну роботу із різними форматами даних.

Для відправки листів зазвичай використовують готові провайдери. Найпопулярнішими є Sendgrid [35], Brevo [36] та Mailgun [37]. Sendgrid та Mailgun є потужнішими, але для їх використання потрібен куплений домен та пошта прив'язана до домена, наприклад hello@andarna.com. Тому було обрано Brevo, як провідний SMTP провайдер, який дозволяє відправляти листи без купленого домена.

Допоміжні інструменти, що використовувалися в розробці наведено в таблиці 1.4.

Таблиця 1.4

Допоміжні інструменти розробки

Назва	Опис
Visual Studio Code	Редактор коду із можливістю повного налаштування розширень, що полегшує роботу з різними мовами та технологіями.
Postman	Інструмент для тестування API, що дозволяє надсилати різні HTTP-запити та отримувати відповіді від сервера як в момент розробки та тестування.

Висновки до першого розділу

В розділі сформульовано постановку задачі, визначено основні складові програмного забезпечення та етапи розробки. Ключовими модулями майбутнього застосунку визначено модулі управління бюджетами, накопиченнями, візуалізація статистики та генерація порад для покращення планування.

При проведенні аналізу предметної області розглянуто важливість планування бюджету, популярні методи планування та визначено вплив сучасних технологій для пришвидшення та покращення процесу.

Проаналізовано аналоги та виявлено основні функції, потрібні в такого виду додатках. Також розроблено функціональні вимоги, поділені на логічні модулі. Крім того, описано нефункціональні вимоги, такі як кросбраузерність, швидкодія, зручність інтерфейсу та захист від випадкового видалення.

При виборі архітектури застосунку бралися до уваги складність впровадження, доцільність використання архітектури застосунків такого виду, а також ефективність роботи та надійність. Було вирішено застосувати клієнт-серверну архітектуру із таким підходом до реалізації як монорепозіторій. Це допоможе зробити застосунок компактним та перевикористати деякі частини коду як на бекенді, так і на фронтенді.

В якості інструментальних засобів та технологій, було використано при розробці: TypeScript як основна мова програмування; для реалізації фронтенду – React.js, Ant Design; для бекенд-частини – Nest.js, PostgreSQL як основна СКБД та TypeORM для роботи БД із сервером; для авторизації буде використано сервіс Firebase Auth; в якості допоміжних сервісів буде використано Stripe для модуля оплати, Nx для впровадження підходу структури монорепозіторія та модель GPT-4-turbo для модуля генерації порад.

РОЗДІЛ 2. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Варіанти використання

Щоб детально визначити основний функціонал, який повинен надавати застосунок, доцільно виділити користувачів, їх ролі та сценарії користування.

Для візуалізації основних варіантів використання було створено Use Case діаграму, її наведено на рисунку 2.1.



Рисунок 2.1 – Use case діаграма

Було виділено 4 основних актори. Неавторизований користувач – може виконувати базові дії, пов’язані з автентифікацією. Авторизований користувач

– актор, який після входу в систему отримує доступ до різних фінансових функцій, а також зовнішні актори, що надають послуги авторизації та створення рекомендацій штучного інтелекту.

Розроблені варіанти використання повинні забезпечити головну функцію застосунку, а саме – управління та планування сімейного бюджету. Детальний опис наведено в таблицях 2.1 – 2.26.

Таблиця 2.1

Варіант використання UC-01

Назва Use Case	Реєстрація
Use case ID	UC-01
Ціль	Реєстрація користувача у додатку
Актори	Неавторизований користувач, Firebase auth provider
Тригер	Користувач бажає зареєструватися в додатку
Попередні умови	Відсутні
Потік подій	Користувач потрапляє на сторінку реєстрації, де вводить дані про себе, а саме пошту, пароль, а також ім'я та прізвище. Після цього натискає на кнопку реєстрації
Розширення	Якщо введено некоректні дані, наприклад невалідну пошту, або занадто легкий пароль, то користувач отримує повідомлення про помилку
Післяумова	Користувач бачить повідомлення про успіх, і про те, що він отримає лист на пошту

Таблиця 2.2

Варіант використання UC-02

Назва Use Case	Авторизація користувача
Use case ID	UC-02
Ціль	Авторизація користувача у додатку
Актори	Неавторизований користувач, Firebase auth provider
Тригер	Користувач бажає авторизуватися в додатку
Попередні умови	Користувач має акаунт
Потік подій	Користувач потрапляє на сторінку авторизації, де вводить пошту, пароль. Після цього натискає на кнопку входу
Розширення	Якщо введено некоректні дані, наприклад невалідну пошту, або некоректний пароль, то користувач отримує повідомлення про помилку
Післяумова	Користувач бачить повідомлення про успіх і отримує доступ до додатку

Таблиця 2.3

Варіант використання UC-03

Назва Use Case	Відновлення доступу до акаунта шляхом зміни пароля
Use case ID	UC-03
Ціль	Відновлення доступу до акаунта шляхом зміни пароля
Актори	Неавторизований користувач, Firebase auth provider
Тригер	Користувач забув пароль і хоче його змінити
Попередні умови	Користувач має акаунт
Потік подій	Користувач переходить на сторінку відновлення пароля, вводить свою email-адресу, натискає кнопку "Відновити пароль", отримує лист із посиланням для зміни пароля, переходить за посиланням та вводить новий пароль
Розширення	Якщо введено некоректну або незареєстровану email-адресу, користувач отримує повідомлення про помилку
Післяумова	Користувач отримує повідомлення про успішну зміну пароля та може увійти з новими даними

Таблиця 2.4

Варіант використання UC-04

Назва Use Case	Отримання рекомендацій від штучного інтелекту
Use case ID	UC-04
Ціль	Надати користувачу можливість отримувати фінансові поради від штучного інтелекту
Актори	Авторизований користувач
Тригер	Користувач бажає отримати фінансову рекомендацію від штучного інтелекту
Попередні умови	Відсутні
Потік подій	На сторінці бюджету користувач бачить кнопку допомоги штучного інтелекту, на яку натискає.
Розширення	Відсутні
Післяумова	Штучний інтелект аналізує бюджет і всі транзакції і надає користувачеві рекомендації для досягнення фінансових цілей.

Таблиця 2.5

Варіант використання UC-05

Назва Use Case	Створення бюджету
Use case ID	UC-05
Ціль	Створення нового бюджету користувача
Актори	Авторизований користувач
Тригер	Користувач бажає створити новий бюджет
Попередні умови	Відсутні
Потік подій	Користувач натискає на кнопку створення нового бюджету, після чого вводить назву і натискає на кнопку збереження.
Розширення	Якщо введено некоректну назву користувач бачить повідомлення про помилку

Продовження таблиці 2.5

Післяумова	Користувач успішно створив новий бюджет і його одразу перенесено на сторінку новоствореного бюджету
------------	---

Таблиця 2.6

Варіант використання UC-06

Назва Use Case	Встановлення початкового балансу бюджету
Use case ID	UC-06
Ціль	Встановлення початкового балансу бюджету
Актори	Авторизований користувач
Тригер	Користувач бажає встановити певну суму як початковий баланс бюджету
Попередні умови	Користувач має мати існуючий бюджет
Потік подій	На сторінці бюджету користувач натискає на кнопку редагування балансу і вводить бажану суму, після чого натискає кнопку збереження
Розширення	Відсутнє
Післяумова	Користувач успішно встановив початковий баланс бюджету і бачить його на сторінці

Таблиця 2.7

Варіант використання UC-07

Назва Use Case	Редагування назви бюджету
Use case ID	UC-07
Ціль	Можливість редагувати назву бюджету
Актори	Авторизований користувач
Тригер	Користувач бажає змінити назву існуючого бюджету
Попередні умови	Користувач має мати існуючий бюджет
Потік подій	На сторінці бюджету користувач натискає на кнопку редагування назви бюджету, вводить нову назву і натискає кнопку збереження
Розширення	Якщо введено некоректну назву, користувач бачить повідомлення про помилку
Післяумова	Користувач успішно встановив нову назву і бачить її на сторінці

Таблиця 2.8

Варіант використання UC-08

Назва Use Case	Видалення бюджету
Use case ID	UC-08
Ціль	Можливість видалити бюджет
Актори	Авторизований користувач
Тригер	Користувач бажає видалити бюджет
Попередні умови	Користувач має мати існуючий бюджет
Потік подій	На сторінці бюджету користувач натискає на кнопку видалення, після чого натискає на кнопку «Так, я впевнений».

Продовження таблиці 2.8

Розширення	Відсутнє
Післяумова	Користувач видалив бюджет

Таблиця 2.9

Варіант використання UC-09

Назва Use Case	Запрошення іншого користувача в бюджет
Use case ID	UC-09
Ціль	Можливість дати доступ до бюджету іншому користувачеві
Актори	Авторизований користувач
Тригер	Користувач бажає запросити іншого користувача до свого бюджету і дати йому доступ
Попередні умови	Користувач має мати існуючий бюджет
Потік подій	На сторінці бюджету користувач натискає на кнопку запрошення, і вводить пошту іншого користувача, після чого натискає на кнопку відправки запрошення
Розширення	Якщо цей користувач уже має доступ до бюджету, то запрошення не буде відправлено повторно, а буде проігноровано
Післяумова	Користувач успішно запросив іншого користувача

Таблиця 2.10

Варіант використання UC-10

Назва Use Case	Створення користувацьких категорій
Use case ID	UC-10
Ціль	Можливість створювати власні категорії доходів та витрат
Актори	Авторизований користувач
Тригер	Користувач бажає створити власну категорію
Попередні умови	Користувач має існуючий бюджет
Потік подій	На сторінці бюджету користувач переходить у розділ категорій де бачить поточні доступні категорії. Натискає там на кнопку створення і вводить дані, а саме назву, тип і колір, після чого натискає кнопку збереження
Розширення	Відсутні
Післяумова	Користувач успішно створив нову категорію для бюджету

Таблиця 2.11

Варіант використання UC-11

Назва Use Case	Редагування користувацьких категорій
Use case ID	UC-11
Ціль	Можливість редагувати власні категорії доходів та витрат
Актори	Авторизований користувач
Тригер	Користувач бажає редагувати власну категорію
Попередні умови	Користувач має існуючий бюджет, та створену категорію

Продовження таблиці 2.11

Потік подій	На сторінці бюджету користувач переходить у розділ категорій де бачить поточні доступні категорії. Натискає на бажану категорію і натискає на кнопку редагування, вводить всі потрібні зміни у поля і натискає кнопку збереження
Розширення	Відсутні
Післяумова	Користувач успішно оновив категорію для бюджету

Таблиця 2.12

Варіант використання UC-12

Назва Use Case	Видалення користувацьких категорій
Use case ID	UC-12
Ціль	Можливість видалення власних категорій доходів та витрат
Актори	Авторизований користувач
Тригер	Користувач бажає видалити власну категорію
Попередні умови	Користувач має існуючий бюджет та створену категорію
Потік подій	На сторінці бюджету користувач переходить у розділ категорій де бачить поточні доступні категорії. Вибирає бажану категорію і натискає на кнопку видалення. У вікні підтвердження натискає «Видалити» ще раз
Розширення	Відсутні
Післяумова	Користувач успішно видалив категорію бюджету.

Таблиця 2.13

Варіант використання UC-13

Назва Use Case	Створення транзакцій у бюджеті
Use case ID	UC-13
Ціль	Можливість створення транзакцій
Актори	Авторизований користувач
Тригер	Користувач бажає додати транзакцію до бюджету
Попередні умови	Користувач має існуючий бюджет
Потік подій	На сторінці бюджету користувач натискає на кнопку додавання транзакції. Вибирає тип (доходи/витрати/заощадження), вводить суму, валюту, назву, вибирає категорію. Якщо валюта не гривня – вводить курс валюти. Натискає зберегти.
Розширення	Якщо певні поля форми заповнені не вірно користувач бачить помилку на екрані
Післяумова	Користувач успішно додав транзакцію до поточного бюджету і бачить її на інтерфейсі.

Таблиця 2.14

Варіант використання UC-14

Назва Use Case	Відміна платної підписки
Use case ID	UC-14
Ціль	Надати користувачу можливість відмінити підписку в будь-який момент

Продовження таблиці 2.17

Ціль	Можливість видалення накопичувальних акаунтів
Актори	Авторизований користувач
Тригер	Користувач бажає видалити накопичувальний акаунт
Попередні умови	Існуючий накопичувальний акаунт
Потік подій	На сторінці накопичувальних акаунтів користувач бачить список існуючих накопичувальних акаунтів. На картці акаунта, який потрібно видалити, натискає на кнопку на кнопку видалення. У вікні підтвердження користувач натискає «Видалити»
Розширення	Відсутні
Післяумова	Користувач успішно видалив накопичувальний акаунт

Таблиця 2.18

Варіант використання UC-18

Назва Use Case	Встановлення початкового балансу накопичувального акаунта
Use case ID	UC-18
Ціль	Можливість встановлення початкового балансу накопичувальних акаунтів
Актори	Авторизований користувач
Тригер	Користувач бажає встановити початковий баланс для накопичувального акаунта
Попередні умови	Існуючий накопичувальний акаунт
Потік подій	На сторінці накопичувальних акаунтів користувач бачить список існуючих накопичувальних акаунтів. На картці акаунта, який потрібно змінити, натискає на кнопку редагування балансу. У формі редагує вводить потрібну суму поточного балансу і зберігає
Розширення	Якщо певні дані у формі введені невірно, користувач отримає помилку
Післяумова	Користувач успішно встановив початковий баланс накопичувальний акаунт

Таблиця 2.19

Варіант використання UC-19

Назва Use Case	Відстеження прогресу накопичувального акаунта
Use case ID	UC-19
Ціль	Можливість відстеження прогресу накопичувальних акаунтів
Актори	Авторизований користувач
Тригер	Користувач бажає відстежити прогрес досягнення цілі для накопичувального акаунта
Попередні умови	Існуючий накопичувальний акаунт
Потік подій	На сторінці накопичувальних акаунтів користувач бачить список існуючих накопичувальних акаунтів. Користувач натискає на накопичувальний акаунт і переходить на його сторінку. Користувач бачить індикатор прогресу у відсотках, що показує прогрес досягнення цілі.
Розширення	Відсутні

Продовження таблиці 2.19

Післяумова	Користувач бачить прогрес досягнення цілі накопичувального акаунта
------------	--

Таблиця 2.20

Варіант використання UC-20

Назва Use Case	Імпорт банок з Monobank у вигляді накопичувальних акаунтів
Use case ID	UC-20
Ціль	Можливість імпорту банок з Monobank у вигляді накопичувальних акаунтів
Актори	Авторизований користувач
Тригер	Користувач бажає імпортувати банки з Monobank
Попередні умови	Користувач має токен Monobank збережений у профілі
Потік подій	На сторінці накопичувальних акаунтів користувач натискає на кнопку імпорту і підтверджує дію
Розширення	Якщо токен Monobank відсутній – дія буде заблокована. Якщо токен невалідний – користувач побачить повідомлення з помилкою
Післяумова	Користувач успішно імпортував усі банки з Monobank до додатку у вигляді накопичувальних акаунтів

Таблиця 2.21

Варіант використання UC-21

Назва Use Case	Видалення акаунта користувача з усіма даними
Use case ID	UC-21
Ціль	Можливість повного видалення акаунта з усіма даними
Актори	Авторизований користувач
Тригер	Користувач бажає видалити акаунт з платформи
Попередні умови	Відсутні
Потік подій	На сторінці профілю користувач натискає на кнопку видалення акаунта, і підтверджує це
Розширення	Відсутні
Післяумова	Користувач успішно видалив акаунт з усіма даними.

Таблиця 2.22

Варіант використання UC-22

Назва Use Case	Перегляд статистики доходів за вказаний період
Use case ID	UC-22
Ціль	Можливість перегляду статистики про доходи
Актори	Авторизований користувач
Тригер	Користувач бажає перегляду статистику доходів
Попередні умови	Відсутні
Потік подій	На сторінці профілю користувач натискає на кнопку перегляду статистики і бачить графіки серед яких графік, що відображає зміни в доходах за вказаний період.
Розширення	Якщо транзакцій ще не було додано, графік буде пустим

Продовження таблиці 2.22

Післяумова	Користувач бачить статистику про доходи за вказаний період
------------	--

Таблиця 2.23

Варіант використання UC-23

Назва Use Case	Перегляд статистики доходів за вказаний період
Use case ID	UC-23
Ціль	Можливість перегляду статистики про доходи
Актори	Авторизований користувач
Тригер	Користувач бажає перегляду статистику доходів
Попередні умови	Відсутні
Потік подій	На сторінці профілю користувач натискає на кнопку перегляду статистики і бачить графіки серед яких графік, що відображає зміни в доходах за вказаний період
Розширення	Якщо транзакцій ще не було додано, графік буде пустим
Післяумова	Користувач бачить статистику про доходи за вказаний період

Таблиця 2.24

Варіант використання UC-24

Назва Use Case	Перегляд статистики витрат за категоріями
Use case ID	UC-24
Ціль	Можливість перегляду статистики про витрати по категоріям
Актори	Авторизований користувач
Тригер	Користувач бажає перегляду статистику витрат по категоріям
Попередні умови	Відсутні
Потік подій	На сторінці профілю користувач натискає на кнопку перегляду статистики і бачить графіки серед яких графік, що відображає витрати по категоріях
Розширення	Якщо транзакцій ще не було додано, графік буде пустим
Післяумова	Користувач бачить статистику про витрати по категоріях

Таблиця 2.25

Варіант використання UC-25

Назва Use Case	Додавання токена Monobank у профілі
Use case ID	UC-25
Ціль	Можливість додавання токена монобанку у профілі користувача
Актори	Авторизований користувач
Тригер	Користувач бажає додати токен Monobank у профілі
Попередні умови	Відсутні
Потік подій	На сторінці профілю користувач натискає на кнопку додавання токена і вводить значення, після чого зберігає.
Розширення	Відсутні
Післяумова	Користувач успішно додав токен до профілю

Таблиця 2.26

Варіант використання UC-26

Назва Use Case	Видалення транзакцій у бюджеті
Use case ID	UC-26
Ціль	Можливість видалення існуючих транзакцій
Актори	Авторизований користувач
Тригер	Користувач бажає видалити транзакцію у бюджеті
Попередні умови	Користувач має існуючий бюджет і транзакцію у ньому
Потік подій	На сторінці бюджету користувач бачить список останніх транзакцій. Біля бажаної натискає на кнопку видалення. У вікні підтвердження натискає «Видалити»
Розширення	Відсутні
Післяумова	Користувач успішно видав транзакцію

2.2 Аналіз системних та нефункціональних вимог, проєктування бази даних системи

При розробці вебзастосунку важливо визначити рекомендовані та мінімальні системні вимоги, щоб забезпечити коректну роботу на різних пристроях і платформах. Вимоги вказані в табл. 2.27.

Таблиця 2.27

Системні вимоги

Назва вимоги	Характеристика
Рекомендовані системні вимоги:	
Браузер	Chrome, Mozilla Firefox, Opera останніх версій
Швидкість інтернет-з'єднання	10 Mbps і вище
Вимоги до оперативної пам'яті	8 ГБ
Мінімальні системні вимоги:	
Браузер	Chrome 90+, Mozilla Firefox 88+, Opera 75+
Швидкість інтернет-з'єднання	3 Mbps і вище
Вимоги до оперативної пам'яті	4 ГБ

Вказані вимоги мають забезпечити користувачу коректну роботу системи при використанні застосунку.

Важливою складовою розробки вебзастосунку є розробка нефункціональних вимог. Вони допоможуть більше детально описати важливі деталі продукту, що розробляється. В ході аналізу було описано наступні вимоги, що представлені в таблиці 2.28.

Таблиця 2.28

Перелік нефункціональних вимог

Вимога	Опис
Кросбраузерність	Вебзастосунок має працювати коректно у визначених браузерах
Швидкодія	Час відклику сторінок не має перевищувати 600 мс
Зручність інтерфейсу	Графічний інтерфейс застосунку має бути інтуїтивно зрозумілим та зручним в користуванні
Захист від випадкового видалення	Якщо користувач натискає на кнопку видалення, система завжди має попросити додаткове підтвердження

База даних розроблялася за допомогою технології PostgreSQL, що є реляційною базою даних. Структуру бази даних відображено за допомогою ER-діаграми, що відображена на рис. 2.2.

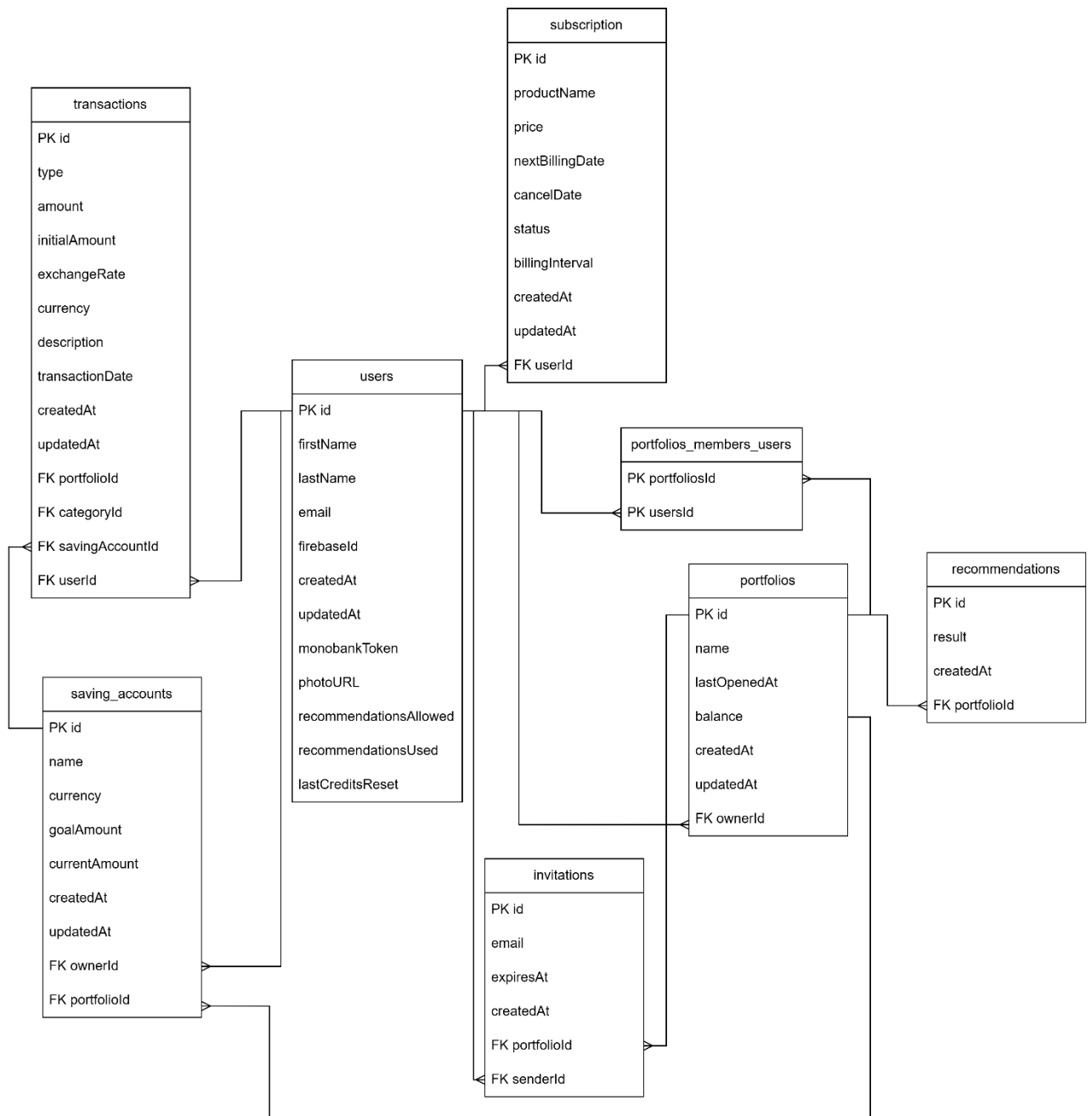


Рисунок 2.2 – ER-діаграма бази даних

База даних складається з 8 основних таблиць. Опис таблиць БД, що містять основні сутності наведено в таблицях 2.29 - 2.36.

Таблиця 2.29

Таблиця Users

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер користувача
firstName	varchar	Ім'я користувача
lastName	varchar	Прізвище користувача
email	varchar	Електронна адреса
firebaseId	varchar	Ідентифікатор клієнта в системі авторизації
photoUrl	text	Деталі про клієнта
monobankToken	varchar	Зашифрований персональний токен користувача в Monobank
createdAt	date	Дата створення запису про користувача
updatedAt	date	Дата оновлення запису про користувача

Таблиця 2.30

Таблиця Transactions

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер транзакції
transactionType	enum	Тип транзакції
amount	number	Сума транзакції
initialAmount	number	Сума транзакції у валюті
exchangeRate	number	Курс валют на момент здійснення транзакції
currency	enum	Валюта транзакції
description	varchar	Опис транзакції
transactionDate	date	Дата здійснення транзакції
createdAt	date	Дата створення запису про транзакцію
updatedAt	date	Дата оновлення транзакції

Продовження таблиці 2.30

portfolioId	uuid (FK)	Ідентифікатор бюджету
categoryId	uuid(FK)	Ідентифікатор категорії
savingAccountId	uuid(FK)	Ідентифікатор накопичувального акаунту
userId	uuid(FK)	Ідентифікатор користувача

Таблиця 2.31

Таблиця Categories

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер категорії
name	varchar	Назва категорії
color	varchar	Колір категорії
portfolioId	uuid (FK)	Ідентифікатор бюджету
createdAt	date	Дата створення категорії
updatedAt	date	Дата оновлення категорії

Таблиця 2.32

Таблиця Portfolios

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер бюджету
name	varchar	Назва бюджету
lastOpenedAt	date	Дата останнього перегляду бюджету
balance	number	Баланс бюджету
owner	uuid(FK)	Ідентифікатор власника бюджету
createdAt	date	Дата створення бюджету
updatedAt	date	Дата оновлення бюджету

Таблиця 2.33

Таблиця Savings Accounts

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер накопичувального акаунта
name	varchar	Назва накопичувального акаунта
currency	varchar	Валюта накопичувального акаунта
goalAmount	number	Бажана ціль фінансових накопичень
currentAmount	number	Поточна кількість фінансів на акаунті
owner	uuid(FK)	Ідентифікатор власника накопичувального акаунта
portfolioId	uuid(FK)	Ідентифікатор бюджету
createdAt	date	Дата створення накопичувального акаунта
updatedAt	date	Дата оновлення запису накопичувального акаунта

Таблиця 2.34

Таблиця Subscriptions

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер підписки
productName	varchar	Назва плану підписки
price	number	Ціна підписки
nextBillingDate	date	Дата наступного знаття коштів за підпискою
cancelDate	date	Дата скасування підписки
status	varchar	Статус підписки
billingInterval	varchar	Інтервал з яким знімаються кошти за підпискою
userId	uuid(FK)	Ідентифікатор користувача що придбав підписку
createdAt	date	Дата оформлення підписки

Продовження таблиці 2.34

updatedAt	date	Дата оновлення запису про підписку
-----------	------	------------------------------------

Таблиця 2.35

Таблиця Invitations

Назва поля	Тип даних	Опис
id	uuid	Ідентифікаційний номер запрошення
email	varchar	Електронна пошта отримувача запрошення
expiresAt	date	Дата закінчення строку дії запрошення
senderId	uuid(FK)	Ідентифікатор користувача що надіслав запрошення
createdAt	date	Дата створення запрошення
portfolioId	uuid(FK)	Ідентифікатор бюджету

Таблиця 2.36

Таблиця Recommendations

Назва поля	Тип даних	Опис
id	uuid	ідентифікаційний номер рекомендації
result	text	Текст рекомендації
createdAt	date	Дата створення рекомендації
portfolioId	uuid(FK)	Ідентифікатор бюджету

2.3 Проектування та реалізація алгоритмів роботи додатку

Для реалізації вище представленого функціоналу було виділено 7 основних сценаріїв роботи додатку та наведено алгоритм їх виконання за допомогою діаграм активностей на рис. 2.3 – 2.9.

На рис. 2.3 наведено діаграму активності для роботи авторизації у додатку. При складанні алгоритму враховано вимоги різних способів входу та автентифікацію користувачів.

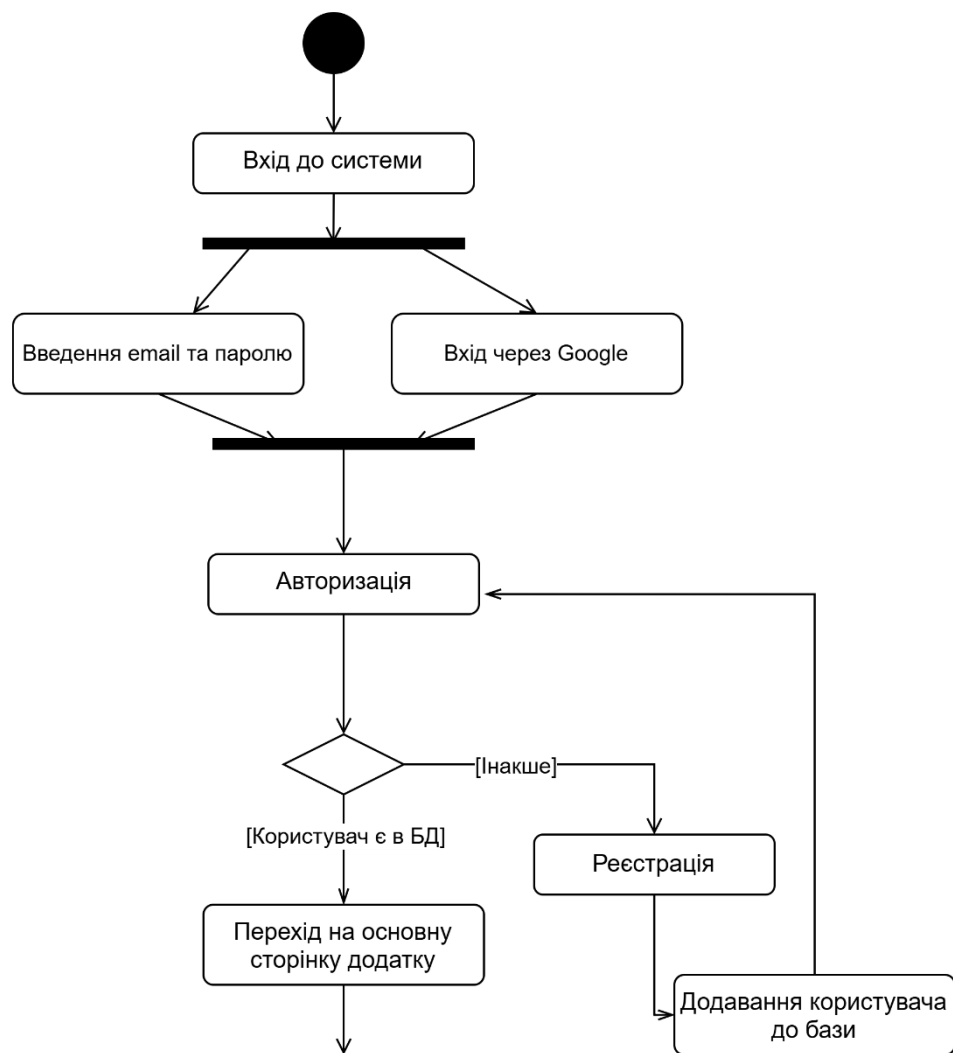


Рисунок 2.3 – Діаграма активностей для авторизації

На рис. 2.4 відображено діаграму активностей для створення користувацьких категорій витрат.



Рисунок 2.4 – Діаграма активностей для створення категорій

На рис. 2.5 відображено діаграму активностей для процесу створення транзакцій. В проектуванні алгоритму важливим було передбачити розгалуження в процесі створення для забезпечення лаконічності та зрозумілості інтерфейсу та надати користувачу достатньо можливостей для налаштування параметрів транзакції.

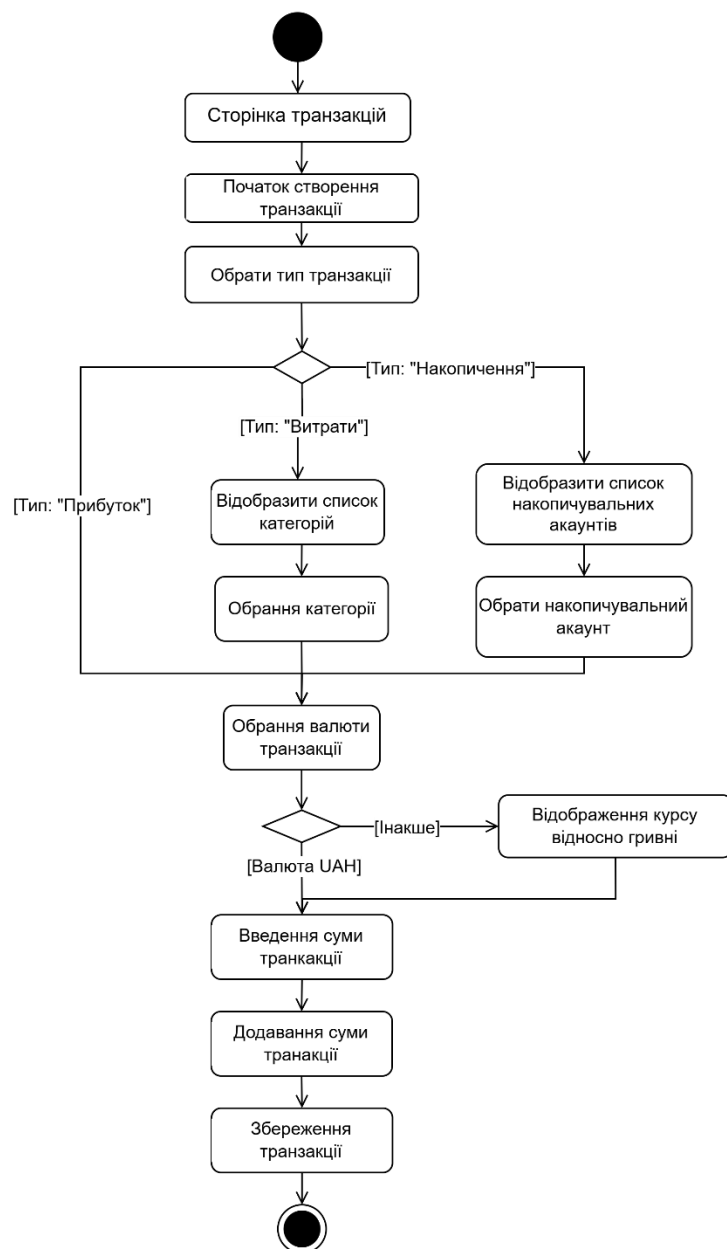


Рисунок 2.5 – Діаграма активностей для створення транзакцій

На рис. 2.6 відображено діаграму активностей для створення та імпортування накопичувальних акаунтів. Варто зауважити що можливість імпортування накопичувальних рахунків з Monobank можлива лише якщо користувач згенерував та додав Monobank токен до свого профілю.

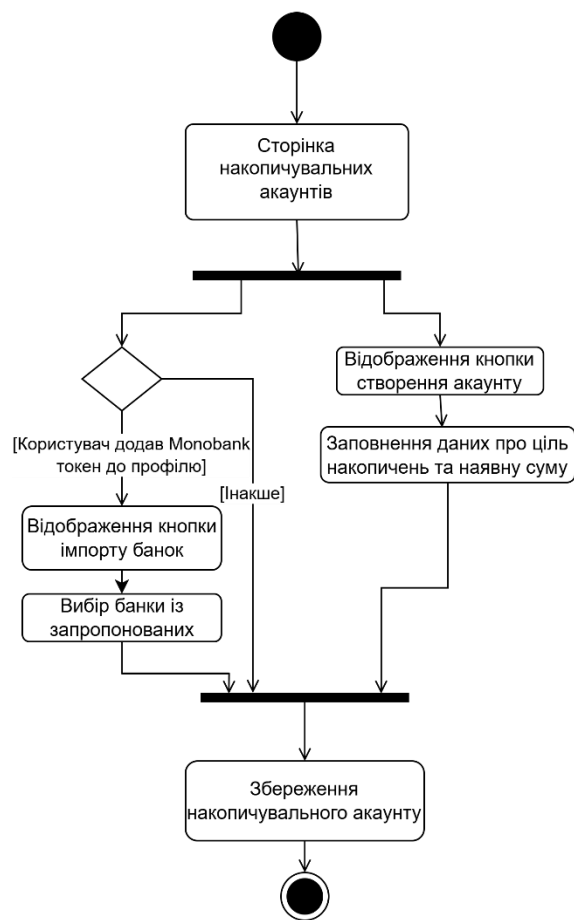


Рисунок 2.6 – Діаграма активностей для накопичувальних акаунтів

На рис. 2.7 зображено діаграму активностей для процесу оформлення платної підписки на сервіс. Для забезпечення роботи грошових переказів буде використано сторонній сервіс Stripe.

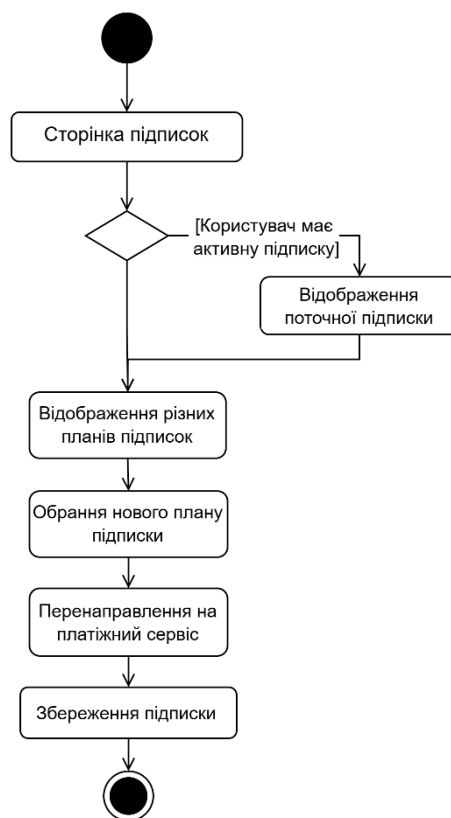


Рисунок 2.7 – Діаграма активностей для оформлення підписки

На рис. 2.8 відображено діаграму активностей для процесу генерації рекомендацій від ШІ. При реалізації також буде використано сторонній сервіс для створення порад, а також варто відмітити що кількість порад на місяць залежить від того, який план у користувача та яка оформлена підписка.



Рисунок 2.8 – Діаграма активностей для III рекомендацій

На рис. 2.9 відображено діаграму активностей для алгоритму роботи запрошення користувачів до бюджету. Важливо відмітити, що користувач має бути вже зареєстрований в системі для успішної авторизації та імпорту бюджету до профілю. Відправка електронних листів здійснюватиметься через сторонній сервіс.



Рисунок 2.9 – Діаграма активностей для запрошення нових учасників

2.4 Реалізація додатку планування сімейного бюджету

Розглянемо детальну реалізацію найважливіших та найскладніших частин вебзастосунку.

Авторизація

Одним з ключових компонентів будь-якого додатку є авторизація. Для спрощення розробки і при цьому отримання кращої безпеки застосунку було вирішено використати провайдер Firebase Authentication, що дозволяє досить легко інтегрувати авторизацію в застосунок. Основний код авторизації на фронтенді реалізовано у файлі AuthProvider.tsx. Даний файл надає React Context, в який передає всі потрібні методи для роботи такі як signIn (авторизація), signInWithGoogle (авторизація через Google), signUp (реєстрація), logout (вихід з акаунта). На рис. 2.10 винесено частину коду, що

відповідає за оновлення стану застосунку при зміні статусу авторизації користувача, такі як: після авторизації або закінчення активної сесії.



```
const AuthProvider = ({ children }: AuthProviderProps) => {
  // ...
  useEffect(() => {
    const unsubscribe = onAuthStateChanged(auth, async (currentUser) => {
      if (currentUser) {
        setUser(currentUser);
        await fetchToken(currentUser);
      }
      setLoading(false);
    });
    return () => {
      unsubscribe();
    };
  }, [contractQueryClient]);
  // ...
};

export default AuthProvider;
```

Рисунок 2.10 – Керування станом застосунку згідно стану авторизації користувача

Коли користувача було успішно авторизовано Firebase дає можливість отримати accessToken – JWT токен, що відправляється на бекенд у заголовку HTTP запита у такому форматі: “Authorization”: “Bearer <JWT>”. Бекенд у свою чергу використовує цей токен для автентифікації та авторизації.

Перед тим як бекенд починає обробляти запит і виконувати бізнес логіку у контролерах та сервісах, спрацьовує так званий Guard (охоронець) – сутність у Nest.js яка призначена для попередньої валідації запиту і за потреби – його миттєвого закінчення і надсилання результату про помилку 401 Not Authorized. Крім того було розроблено спеціальні декоратори у файлах currentUser.decorator.ts та userFromToken.decorator.ts, що дозволяють дістати інформацію про розкодовані дані JWT токена, або дані користувача з бази даних для подальшого їх використання в коді. На рис. 2.11 показано частину

коду реалізації механізму захисту бекенду від несанкціонованого входу з файлу auth.middleware.ts.

```

@Injectable()
export class AuthMiddleware implements NestMiddleware {
  private auth: firebase.auth.Auth;

  constructor(
    private firebaseApp: FirebaseApp,
    @InjectRepository(User)
    private userRepository: Repository<User>
  ) {
    this.auth = firebaseApp.getAuth();
  }

  use(req: ExtendedRequest, res: Response, next: () => void) {
    const token = req.headers.authorization;
    if (token != null && token != '') {
      this.auth
        .verifyIdToken(token.split(' ')[1])
        .then(async (decodedToken) => {
          const userFromDb = await this.userRepository.findOne({
            where: {
              firebaseId: decodedToken.uid,
            },
            select: {
              // fields from DB
            },
          });

          req.user = userFromDb;
          req.firebasePayload = decodedToken;
          next();
        })
        .catch((e) => {
          console.log(e);

          next();
        });
    } else {
      next();
    }
  }
}
```

Рисунок 2.11 – Реалізація middleware для верифікації JWT токена і отримання даних користувача.

Механізм взаємодії клієнтської та серверної частини

Авторизований користувач має доступ на керування визначеними системою ресурсами, тобто отримує можливість робити CRUD запити до бюджету, транзакції, категорії тощо. Робота з усіма сутностями є дуже схожою і фактично дублюється. Це було досягнуто саме завдяки використанню

правильних архітектурних підходів. Розглянемо приклад реалізації на сутності категорій витрат.

Вся логіка реалізації завжди починається з опису контракту, за яким будуть працювати фронтенд та бекенд. Контракт включає в себе метод HTTP запиту, шлях, параметри, query-параметри, схему тіла запиту та перелік всіх можливих відповідей включно з статус кодами. Приклад реалізації контракту для отримання всіх категорій, створених у бюджеті та для створення нової категорії наведено на рис. 2.12 і взято з файлу `category.contract.ts`.

```
export const CategorySchema = z.object({
  id: z.string().uuid(),
  name: z.string(),
  color: z.string(), // Assuming color is a hex code
  createdAt: z.date(),
  updatedAt: z.date(),
});

export const categoriesContract = c.router({
  getCategories: {
    method: 'GET',
    path: '/portfolios/:portfolioId/categories',
    pathParams: z.object({
      portfolioId: z.string().uuid(),
    }),
    responses: {
      200: createResponseSchema(z.array(CategorySchema)),
      403: createErrorSchema(),
      404: createErrorSchema(),
    },
  },
  createCategory: {
    method: 'POST',
    path: '/portfolios/:portfolioId/categories',
    pathParams: z.object({
      portfolioId: z.string().uuid(),
    }),
    body: CategorySchema.omit({
      id: true,
      createdAt: true,
      updatedAt: true,
    }),
    responses: {
      201: createResponseSchema(CategorySchema),
      400: createErrorSchema(),
      403: createErrorSchema(),
      404: createErrorSchema(),
    },
  },
  // ... update and delete
});
```

Рисунок 2.12 – Контракт для категорій витрат

Для написання коду контрактів використано бібліотеку ts-rest [38], що дозволяє описати строгий контракт, після чого ми отримаємо повністю типізований інтерфейс для роботи з ним на бекенді та фронтенді, з готовою валідацією даних, тощо.

Наступним кроком на бекенді реалізується бізнес логіка шляхом написання контролеру та сервісів. У контролерах зазвичай отримуються і валідуються дані запиту, перевіряються наявні доступи, після чого викликаються потрібні методи сервісів, де знаходиться вся бізнес логіка і робота з БД. В результаті, дані з сервісів відправляються в правильному форматі, відповідно до контракту. На рис. 2.13 показано приклад реалізації з файлу outcomeCategory.controller.ts.

```

@controller()
@useGuards(AuthGuard)
export class OutcomeCategoryController {
  constructor(
    private readonly outcomeCategoryService: OutcomeCategoryService,
    private readonly portfolioService: PortfolioService
  ) {}

  @tsRestHandler(contract.categories.getCategories)
  async getCategories(@currentUser() user: User) {
    return tsRestHandler(
      contract.categories.getCategories,
      async ({ params: { portfolioId } }) => {
        const portfolio = await this.portfolioService.verifyPortfolioAccess(
          user.id,
          portfolioId
        );
        if (!portfolio) {
          return {
            status: 403,
            body: createErrorObject({
              message: 'You do not have access to this portfolio',
              code: 'FORBIDDEN',
            }),
          };
        }
        const categories = await this.outcomeCategoryService.findAllInPortfolio(
          portfolioId
        );

        return {
          status: 200,
          body: createResponseObject(categories),
        };
      }
    );
  }
}

```

Рисунок 2.13 – Приклад реалізації типового контролера

У наведеному контролері, після закінчення валідації даних викликається метод сервісу OutcomeCategoryService з назвою findAllInPortfolio. На рис 2.14 наведено реалізацію даного методу, як приклад типового методу сервісу.

```

@Injectable()
export class OutcomeCategoryService {
  constructor(
    @InjectRepository(Category)
    private readonly outcomeCategoryRepository: Repository<Category>
  ) {}

  async findAllInPortfolio(portfolioId: string): Promise<Category[]> {
    const customCategories = await this.outcomeCategoryRepository.find({
      where: {
        portfolio: {
          id: portfolioId,
        },
      },
      order: {
        createdAt: 'ASC',
      },
    });
    return customCategories;
  }
  // ... other methods
}

```

Рисунок 2.14 – Приклад реалізації коду сервісу

Можна помітити, що сервіс не взаємодіє з базою даних напряму, виконуючи SQL запити. Замість цього використовується ORM, в даному випадку TypeORM. Це бібліотека, яка дозволяє розробнику за допомогою класів та декораторів описати сутності бази даних, і потім за допомогою вбудованих методів автоматично конструювати SQL запити будь-якої складності. Приклад створення сутностей у вигляді класів наведено на рис. 2.15 і показано файл outcomeCategory.entity.ts.

```

@Entity('categories')
export class Category implements IOutcomeCategory {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'varchar', length: 255 })
  name: string;

  @Column({ type: 'varchar', length: 7 }) // Assuming color is a hex code
  color: string;

  @ManyToOne(() => Portfolio, (portfolio) => portfolio.categories, {
    onDelete: 'CASCADE',
  })
  portfolio: Portfolio;

  @OneToMany(() => Transaction, (transaction) => transaction.category)
  transactions: Transaction[];

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}

```

Рисунок 2.15 – Приклад опису сутності бази даних

Імпорт банок користувача з Monobank

Для реалізації функціоналу імпорту банок з Monobank було використано публічне API Monobank [39]. Було розроблено 2 ендпоінти на бекенді – один для отримання банок користувача, другий – для їх імпорту. Запит для імпорту не є цікавим, адже нічим не відрізняється від звичайного створення накопичувального акаунта, крім того що їх можна імпортувати одразу багато. Код реалізації отримання списку банок користувача, маючи його API токен, можна знайти у файлі monobank.service.ts і його наведено на рис. 2.16.

```

@injectable()
export class MonobankService {
  private readonly monobankApiUrl = 'https://api.monobank.ua';
  constructor(private readonly httpService: HttpService) {}

  async getUserJars(monobankToken: string): Promise<SavingAccount[]> {
    const url = `${this.monobankApiUrl}/personal/client-info`;
    const headers = {
      'X-Token': monobankToken,
    };

    try {
      const response = await firstValueFrom(
        this.httpService.get<MonobankClientInfoResponse>(url, { headers })
      );
      const clientInfo = response.data;

      if (!clientInfo || !clientInfo.jars) {
        return [];
      }

      const mappedSavingAccounts: SavingAccount[] = clientInfo.jars
        .map<SavingAccount | null>((jar) => {
          const currencyEnum = mapCurrencyCodeToEnum(jar.currencyCode);
          if (!currencyEnum) {
            console.warn(
              `Unsupported currency code received: ${jar.currencyCode} for jar ${jar.id}`
            );
            return null;
          }

          const divisor = 100;

          return {
            id: jar.id,
            name: jar.title,
            currency: currencyEnum,
            goalAmount: jar.goal ? jar.goal / divisor : null,
            currentAmount: jar.balance / divisor,
          } as SavingAccount;
        })
        .filter((account): account is SavingAccount => account !== null);

      return mappedSavingAccounts;
    } catch (error) {
      // ...
    }
  }
}

```

Рисунок 2.16 – Реалізація отримування списку доступний банок користувача у Monobank

Інтеграція Stripe для роботи підписок

Для інтеграції Stripe спочатку було створено продукт «Pro plan», що фактично являється підпискою, і має 2 ціни – 15 доларів на місяць та 144 долари на рік. Описані налаштування показані на рис. 2.17.

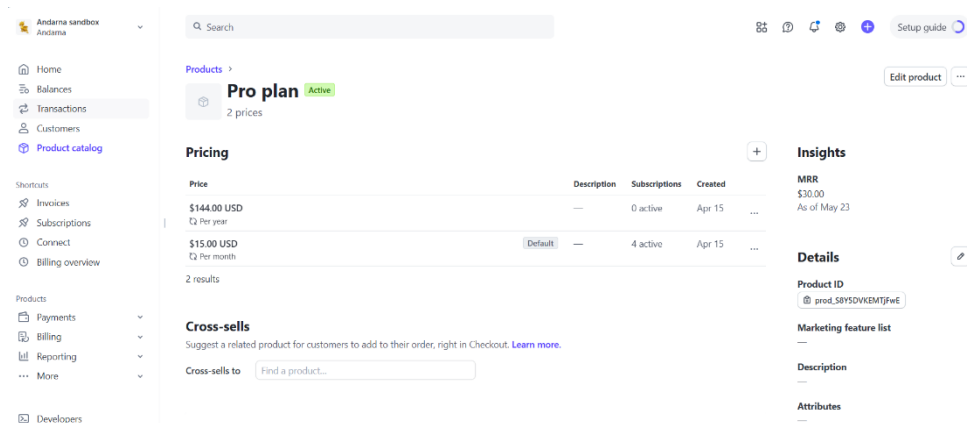


Рисунок 2.17 – Налаштування продукту та цін на платформі Stripe

Дані про підписки користувача синхронізуються з базою даних PostgreSQL для того, щоб можна було швидко і ефективно отримувати дані про наявність підписки у користувача. Якщо підписка досі не оформлена, то на сторінці користувач бачить всі можливі плани, а саме місячний та річний. Якщо ж підписка уже є, користувач бачить всі потрібні йому дані про те, коли буде наступний платіж, яка сума платежу, які переваги він отримав тощо. Крім того є кнопка «Керувати підпискою» що дозволяє користувачу переходити з одного плану на інший, або відмінити підписку.

Оформлення підписки було розроблено шляхом інтеграції Stripe Hosted Checkout сторінки. Коли користувач натискає на кнопку «Підписатися» бекенд викликає метод `createCheckoutSession`, і відкривається сторінка розроблена компанією Stripe, де користувач проводить оплату будь-яким доступним методом. Після оплати – підписка відображається у панелі управління Stripe, а на бекенді буде отримано вебхук від Stripe з інформацією про користувача та про підписку, яку було оформлено. Вебхук спрацюватиме на 3 події – оформлення підписки, оновлення підписки (це стається, коли користувач змінює план, або переходить на новий платіжний інтервал) та відміну підписки. Це потрібно для того, щоб синхронізувати дані зі Stripe з даними в БД проєкту. Код реалізації вебхука можна знайти у файлі `webhook.controller.ts`, що наведено в додатку А.

Генерація фінансових рекомендацій за допомогою ШІ

Початково було зареєстровано акаунт на платформі OpenAI, де було створено новий проект «Andarna». Було згенеровано API ключ, підключено платіжний метод та сплачено певну суму доступних кредитів. На платформі було створено і налаштовано асистента. Налаштування включають в себе назву асистента, інструкції і вибір моделі.

Асистент отримав назву Financial Helper, модель – gpt-4-turbo. Цю модель було обрано, адже вона є не надто говіркою, а її відповіді є відносно швидкими у порівнянні з іншими. Інструкції, що отримав асистент продемонстровано на рис. 2.18. Крім інформації про своє завдання, асистент отримав дані про приклад вхідних даних користувача, а також дані про середню заробітну плату та ціни в Україні.

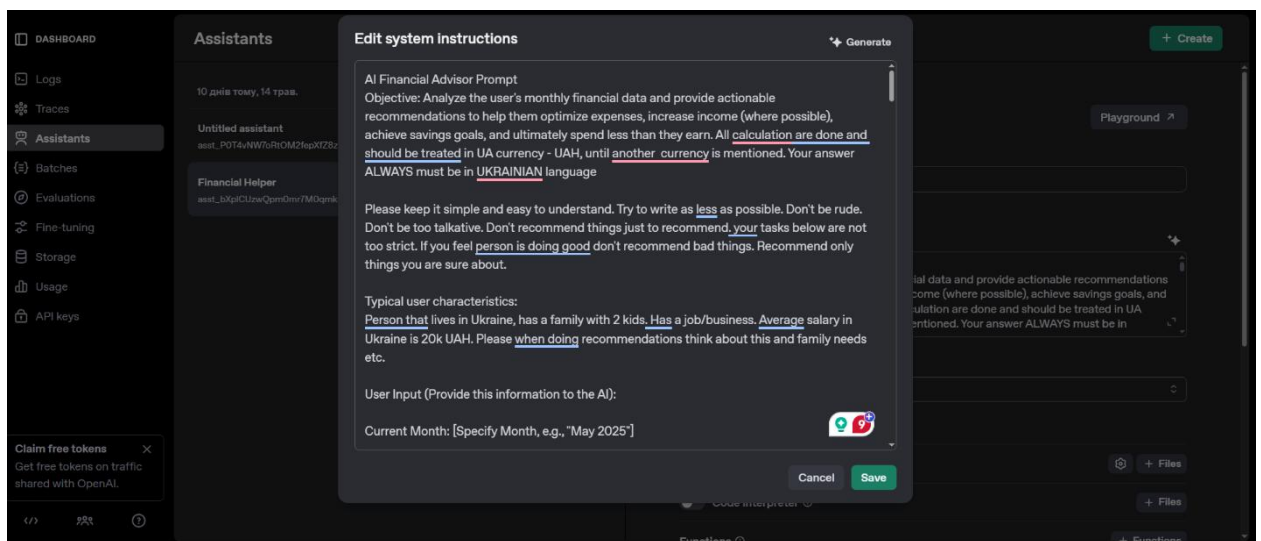


Рисунок 2.18 – Інструкції отримані асистентом для генерації відповідей

Код реалізації підключення OpenAI до бекенду, а саме частини відправки запиту до LLM і отримання результату продемонстровано на рис. 2.19. Код взято з файлу openai.service.ts.

```

async askFinanceAssistant(prompt: string): Promise<string> {
  try {
    const thread = await this.openAi.beta.threads.create();
    await this.openAi.beta.threads.messages.create(thread.id, {
      role: 'user',
      content: prompt,
    });

    const run = await this.openAi.beta.threads.runs.createAndPoll(thread.id, {
      assistant_id: 'asst_bXpLCUzwQpm0mr7M0qmKKBzv',
    });
    if (run.status === 'completed') {
      const messages = await this.openAi.beta.threads.messages.list(
        run.thread_id
      );
      for (const message of messages.data.reverse()) {
        if (message.role === 'assistant')
          return (message.content[0] as any).text.value as string;
      }
    }
    return 'No response from assistant';
  } catch (error: any) {
    console.error('Error communicating with OpenAI:', error.message);
    throw new Error('Failed to communicate with OpenAI API');
  }
}

```

Рисунок 2.19 – Реалізація відправки запиту до асистента і отримання результату

Результати отримані штучним інтелектом зберігаються у БД для можливого майбутнього перегляду користувачем Andarna.

Запрошення користувачів до бюджету та налаштування прав доступу

Вебзастосунок призначений для планування сімейного фінансового бюджету, тому запрошення інших користувачів до бюджету є однією з ключових функціональних вимог. Для реалізації запрошень використано стандартний підхід – запрошення мають зберігатися як сутність в базі даних. Проте, є важлива деталь, що запрошення має бути унікальним, а також мати термін придатності. Якщо запрошення не прийнято протягом 24 годин – воно стає недоступним, і більше не може бути прийнятим. Щоб запросити цю людину в бюджет потрібно перевідправити запрошення і тоді старе запрошення буде видалено, а нове відправлено.

Щоб забезпечити зручну доставку запрошення користувачу було вирішено надсилати їх через електронну пошту. Спочатку потрібно отримати доступ до певного SMTP сервера, який і буде відправляти листи. Було

zareestrowano akaunt na servisi Brevo, який дає можливість безкоштовно відправляти листи. Після реєстрації було отримано API ключ, який на стороні бекенду передається у бібліотеку @getbrevo/brevo, яка в свою чергу викликає API Brevo і надсилає листи. Для відправки листів було створено окремий сервіс mailer.service.ts., код реалізації якого наведено в додатках.

Можна помітити що метод sendMail має логіку для надсилання листів з заготовок. Це реалізовано наступним чином: в тій же папці, де знаходиться mailer.service.ts є папка templates/ де розміщуються HTML файли. Далі при виклику методу потрібно вказати назву заготовки, тобто назву HTML файлу, який буде використано як базу для листа. Крім того, всередині HTML файлу є можливість додавати змінні, які потрібно також передати ззовні, з методу sendMail.

Отримання актуального курсу валют і його кешування

Важливою деталлю отримання актуального курсу валют є вибір надійного ресурсу, звідки цей курс можна завжди отримати. Найкращим варіантом є отримання курсу валют з офіційного API Національного банку України. Фактично отримання курсу валют з API є досить тривіальною задачею, але при цьому постає інше питання – кешування результатів. Коли на вебзастосунок заїде велика кількість людей, під час користування ним їм усім потрібен актуальний курс валют. Кожне завантаження сторінки змушує робити новий запит на отримання курсу валют до API НБУ, проте курс валют фактично не змінюється так часто, зазвичай це відбувається один чи кілька разів на день, тож є сенс закешувати результати у якомусь тимчасовому сховищі на певний час. Таке рішення зменшить кількість запитів до НБУ в рази, а також значно прискорить виконання більшості запитів на курс валют, адже дістати дані з кешу набагато швидше ніж зі стороннього API.

Як сховище для кешування було обрано найбільш оптимальний сервіс – Redis. Кожного разу коли робиться запит на курс валют, спочатку перевіряється

чи є ці дані в кеші. Якщо вони там знаходяться – запит одразу закінчується. Якщо ж даних нема, то робиться запит до АРІ НБУ для отримання актуального курсу, після чого курс кешується в Redis на 5 хвилин, що означає, що всі запити на отримання курсу валют наступні 5 хвилин будуть братися з кешу. Код реалізації цього кешування можна знайти в файлі `exchangeNbu.controller.ts` і наведено на рис. 2.20.

```

@Controller()
export class ExchangeNbuController {
  constructor(
    private exchangeNbuService: ExchangeNbuService,
    @Inject(CACHE_MANAGER) private cacheManager: Cache
  ) {}

  @TsRestHandler(contract.nbuExchange.getExchangeRate)
  async getExchangeRate() {
    return tsRestHandler(
      contract.nbuExchange.getExchangeRate,
      async ({ params: { currency } }) => {
        const existingCurrencyRate = await this.cacheManager.get<number>(
          `exchange-rate/${currency}`
        );
        if (existingCurrencyRate) {
          return {
            status: 200,
            body: {
              rate: existingCurrencyRate,
            },
          };
        }
        const exhcangeRate = await this.exchangeNbuService.getExchangeRate(
          currency as Currency
        );
        if (!exhcangeRate) {
          throw new NotFoundException();
        }
        await this.cacheManager.set(
          `exchange-rate/${currency}`,
          exhcangeRate.rate,
          { ttl: 300 } as any
        );
        return {
          status: 200,
          body: {
            rate: exhcangeRate.rate,
          },
        };
      }
    );
  }
}

```

Рисунок 2.20 – Запит на отримання курсу валют з кешуванням у Redis

Висновок до другого розділу

У другому розділі було детально розроблено вимоги до програмного забезпечення. На початку визначено основні варіанти використання системи, що охоплюють взаємодію неавторизованих та авторизованих користувачів із

системою, а також взаємодію з зовнішніми сервісами. Було представлено 26 деталізованих варіантів використання, що описують ключові сценарії роботи користувачів із застосунком.

Сформульовано системні вимоги, що включають рекомендовані та мінімальні конфігурації для забезпечення коректної роботи вебзастосунку на пристроях користувачів.

Важливою частиною розділу стала розробка бази даних системи. Було зазначено, що використовується реляційна база даних PostgreSQL. Представлено ER-діаграму, що відображає структуру бази даних, яка складається з 8 основних таблиць. Наведено детальний опис ключових таблиць: User, Transaction, Category, Portfolios, Saving accounts, Subscriptions, Invitation та Recommendations, із зазначенням полів, типів даних та їх описів.

На завершення розділу було спроектовано та реалізовано алгоритми роботи основних функцій додатку. Розглянуто алгоритми для авторизації, створення категорій, створення транзакцій, створення та імпортування накопичувальних акаунтів, оформлення платної підписки, генерації рекомендацій від ШІ та запрошення користувачів до бюджету. Також було описано реалізацію ключових компонентів застосунку, таких як механізм авторизації через Firebase Authentication, взаємодія клієнтської та серверної частини за допомогою контрактів ts-rest та шарової архітектури Nest.js, імпорт даних з Monobank, інтеграція з платіжною системою Stripe для управління підписками, генерація фінансових рекомендацій за допомогою OpenAI API (модель gpt-4-turbo), механізм запрошення користувачів через електронну пошту за допомогою сервісу Brevo та отримання актуального курсу валют з API НБУ з кешуванням результатів у Redis.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З СИСТЕМОЮ ДОДАТКУ

3.1 Встановлення та налаштування системи

Розглянемо детальніше взаємодію основних модулів застосунку. На рис. 3.1 наведено діаграму компонентів додатку.

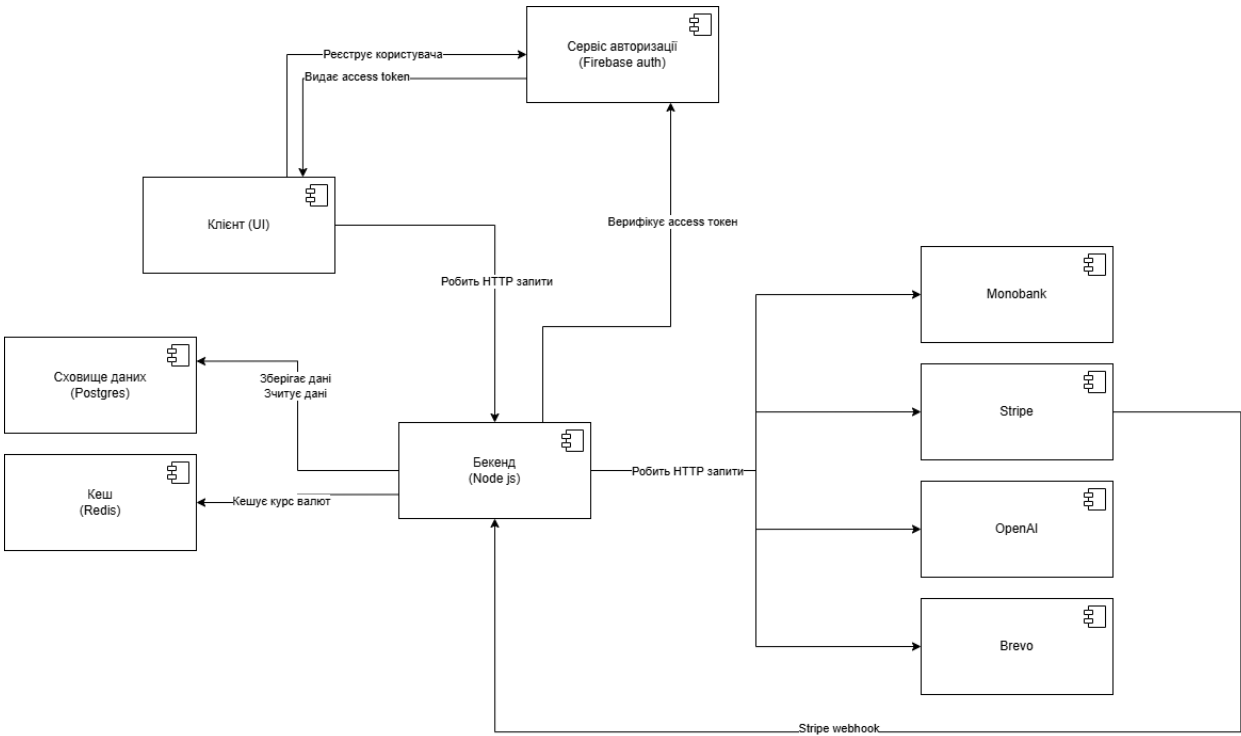


Рисунок 3.1 – Діаграма компонентів веб застосунку

Клієнт (UI) – основний компонент з яким взаємодіє користувач шляхом використання веб-браузера на своєму пристрої. Клієнт взаємодіє напряму з модулем авторизації і, якщо користувач авторизований, отримує access token для авторизації. Цей токен клієнт включає в усі запити на бекенд.

Фронтенд спілкується з бекендом по HTTP/HTTPS протоколам. Бекенд перед обробкою запитів перевіряє валідність токenu у того ж самого сервісу авторизації, що і видав токен. Бекенд регулярно займається збереженням та зчитуванням даних з сховища даних Postgres, а також кешує певні дані у Redis. Крім того система активно взаємодіє з сторонніми API такими як Monobank,

НБУ, Stripe, OpenAI. Взаємодія з сторонніми API відбувається згідно з їх документацією, зазвичай це HTTP запит за принципом REST.

І останньою деталлю є той факт, що Stripe повинен надсилати вебхук назад на бекенд, коли стаються певні події в системі. Він це також робить по HTTP.

Розглянемо також діаграму розгортання додатку, що зображено на рис. 3.2.

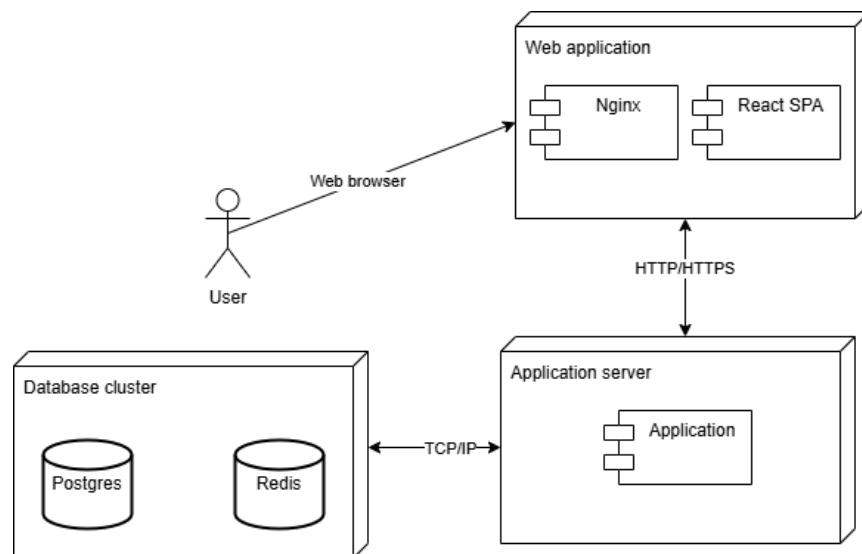


Рисунок 3.2 – Діаграма розгортання веб застосунку

Як було згадано в пункті 2.4, користувач взаємодіє з вебзастосунком за допомогою веббраузера. Розроблений вебзастосунок написаний на React.js, який в результаті збірки дає один HTML файл, CSS і багато JavaScript файлів, що імпортуються. Тому для розгортання веб застосунку найдоцільніше використати Nginx – який виконує роль вебсервера і дозволяє легко додати кешування, стиснення, безпеку тощо.

Серверну частину додатку буде задеплоєно окремо, тому вони спілкуватимуться по захищеному протоколу HTTPS, що надає безпеку трафіку.

Бекенд найбільше взаємодіє з БД і вони взаємодіють по TCP/IP.

3.2 Структура інтерфейсу та порядок взаємодії із застосунком

Для будь-якої взаємодії із додатком користувач має спочатку зареєструватися або увійти до системи.

Для реєстрації потрібно перейти на сторінку реєстрації та заповнити форму правильними даними. Приклад заповненої форми реєстрації наведено на рис. 3.3.

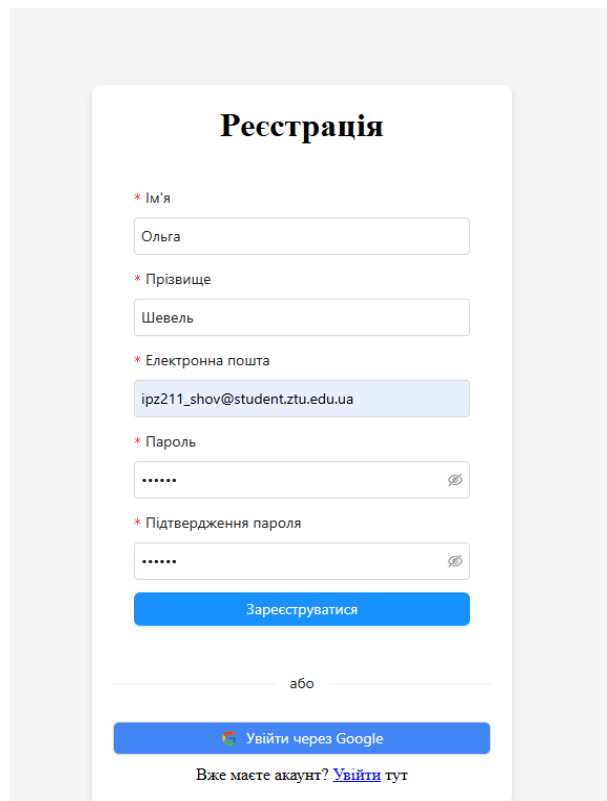


Рисунок 3.3 – Реєстрація нового користувача

Також реєстрація можлива через Google-акаунт користувача, для цього потрібно обрати опцію «Увійти через Google». Приклад реєстрації через Google наведено на рис. 3.4.

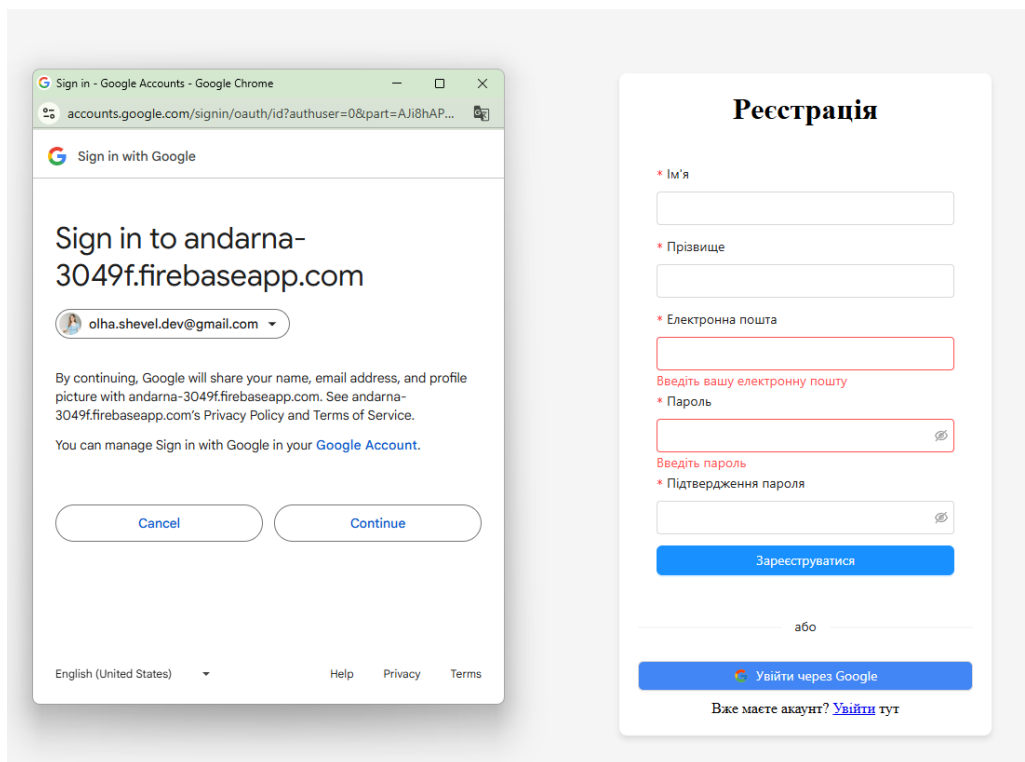


Рисунок 3.4 – Реєстрація через Google

Якщо користувач вже має акаунт у застосунку, тоді варто перейти на сторінку авторизації та ввести правильні дані для входу. Приклад заповнення форми авторизації наведено на рис. 3.5.

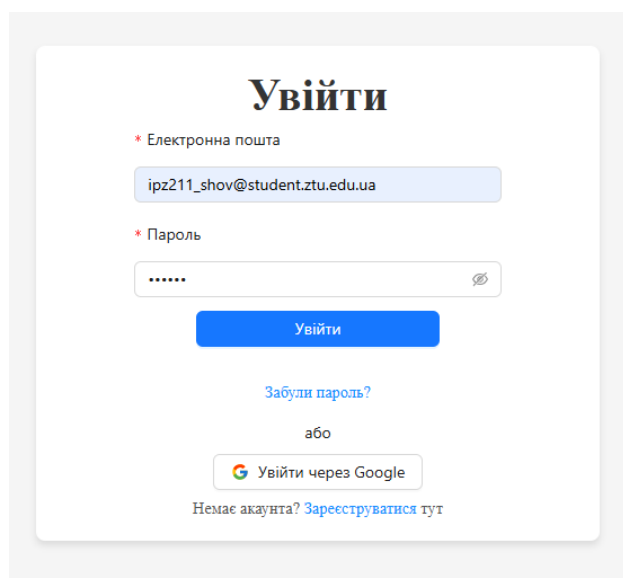


Рисунок 3.5 – Авторизація користувача

Додатково можлива авторизація через Google, це працює аналогічно реєстрації.

Після успішної авторизації та реєстрації користувача перенаправлено на головну сторінку, де відображені усі бюджети. На початку у користувача лише один бюджет, що створюється автоматично при реєстрації. На рис. 3.6 можна побачити як виглядає головна сторінка додатку.

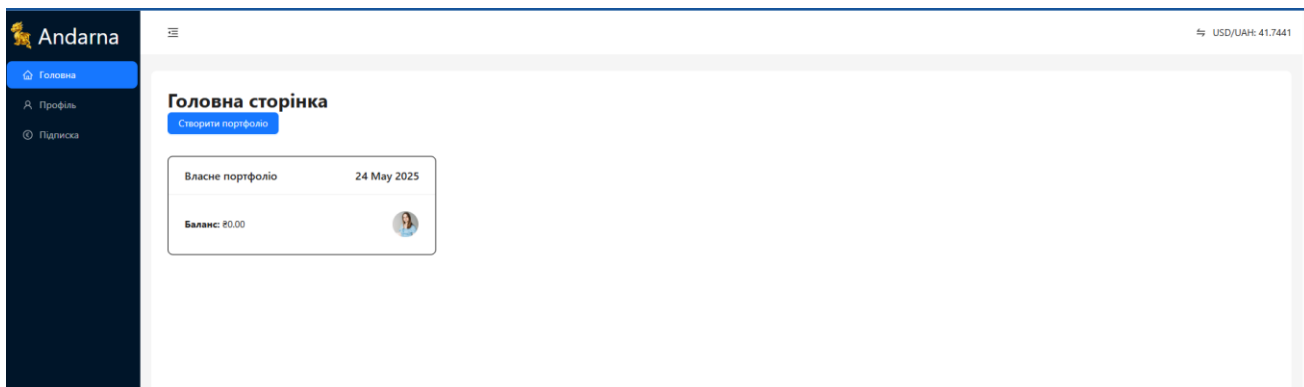


Рисунок 3.6 – Головна сторінка додатку

Користувач має можливість створити новий бюджет шляхом натиснення кнопки «Створити бюджет» та заповнення форми у модальному вікні (рис. 3.7) , де потрібно вказати назву нового бюджету.

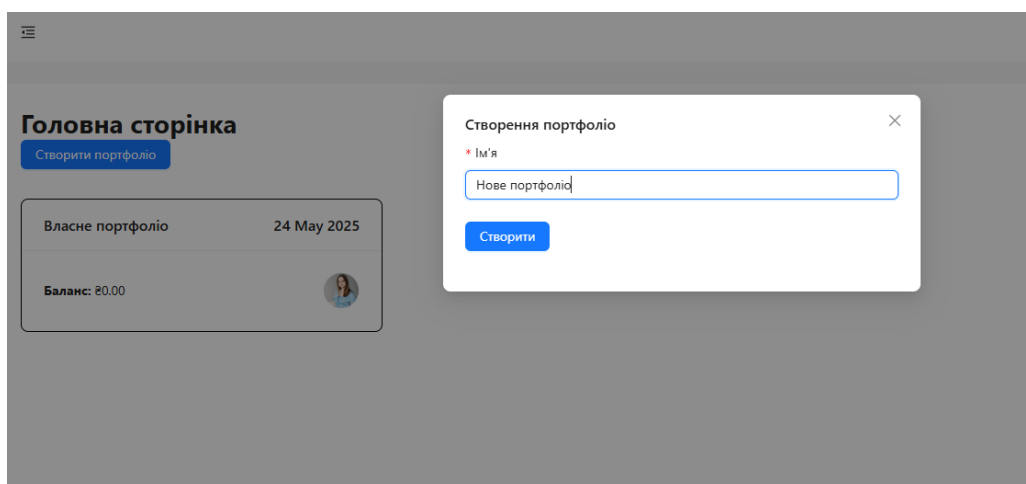


Рисунок 3.7 – Створення нового бюджету

Після створення новий бюджет відображається на головній сторінці (рис. 3.8).

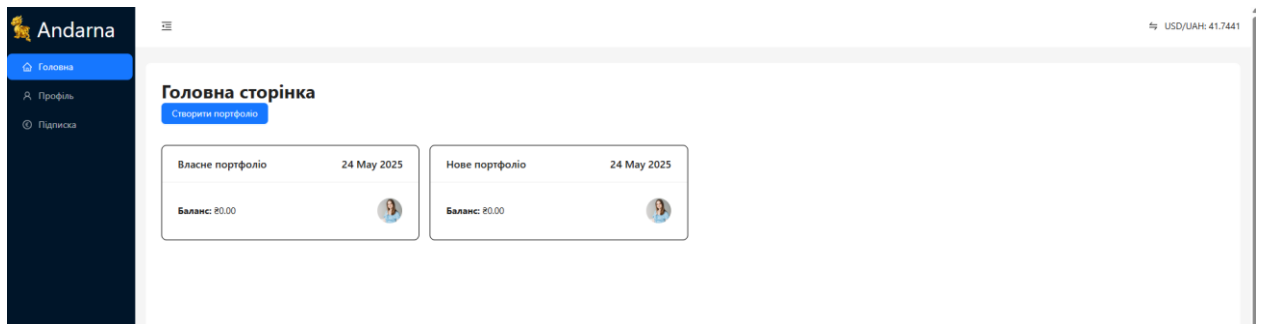


Рисунок 3.8 – Відображення декількох бюджетів

Для керування бюджетом потрібно перейти на його сторінку шляхом натиснення на картку бюджету. Сторінка бюджету має декілька вкладок, а саме «Транзакції», «Статистика», «Рекомендація ШІ», «Учасники», «Збереження» та «Категорії». Інтерфейс вкладки «Транзакції», яка є головною на сторінці бюджету можна побачити на рис. 3.9.

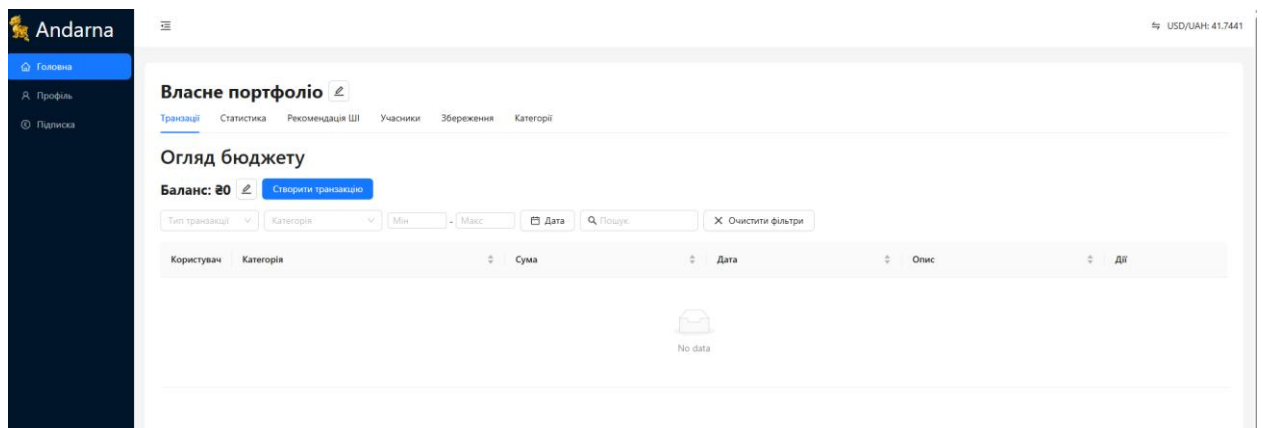


Рисунок 3.9 – Головна сторінка бюджету

Для початку роботи з бюджетом доцільним є для початку створити користувацькі категорії витрат. Для цього потрібно перейти на вкладку «Категорії» (рис. 3.10) та натиснути «Створити категорію». При створенні потрібно дати назву категорії та задати колір. Приклад створення категорії наведено на рис. 3.11.

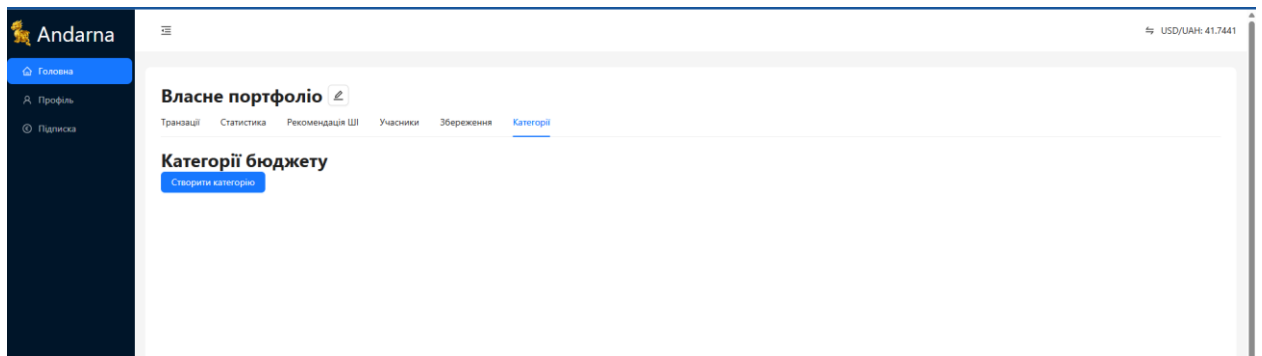


Рисунок 3.10 – Сторінка категорій

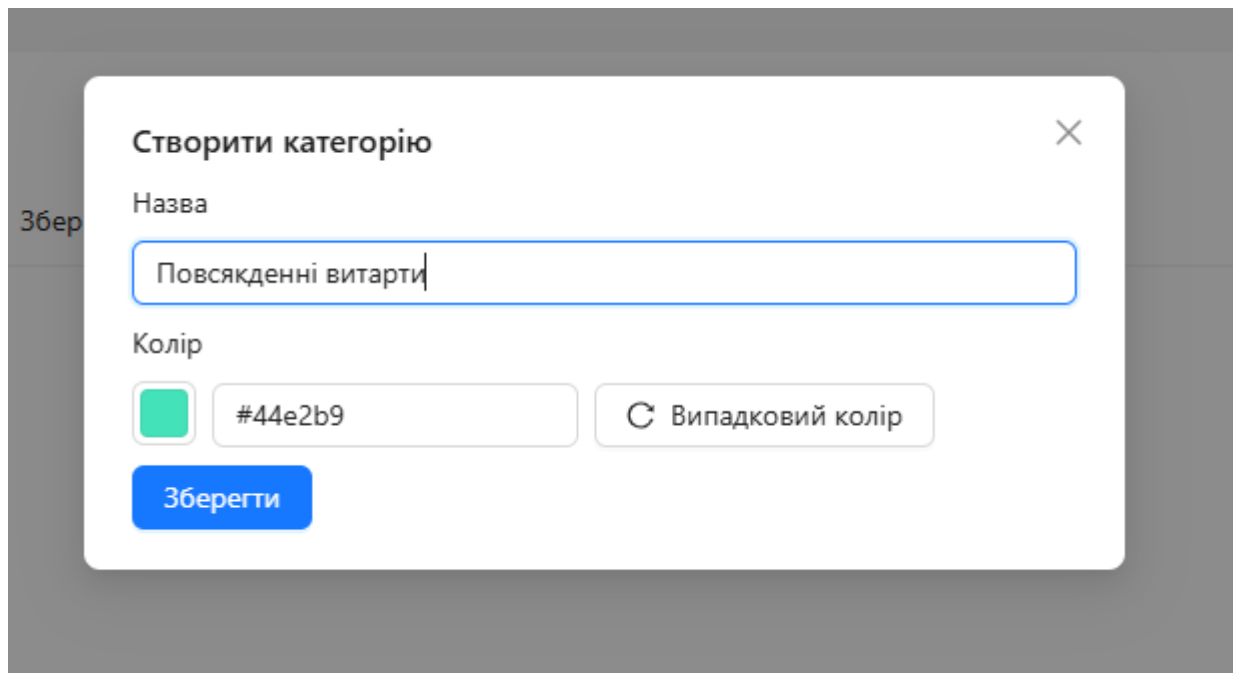


Рисунок 3.11 – Створення категорії

Після створення категорії відображаються у вигляді карток на сторінці (рис. 3.12).

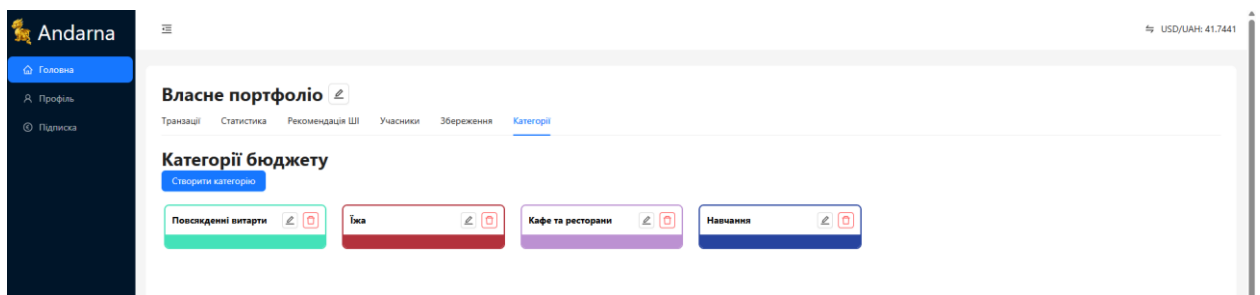


Рисунок 3.12 – Відображення категорій на сторінці

Тепер, коли категорії створено, можна перейти до створення транзакцій. Транзакції можна створити транзакції 3 типів: дохід, витрати та заощадження.

Для створення транзакції надходження коштів до бюджету потрібно натиснути кнопку «Створити транзакцію» та при заповненні форми створення транзакції обрати тип «Дохід». Далі потрібно заповнити дані про суму надходжень, валюту, дату транзакції та, опціонально, додати опис. Приклад створення наведено на рис. 3.13.

The screenshot shows a modal window titled "Створити транзакцію" (Create Transaction) with a close button (X) in the top right corner. The form contains the following fields:

- * Тип транзакції** (Transaction Type): A dropdown menu with "Дохід" (Income) selected.
- * Сума** (Amount): A text input field containing "20000".
- * Валюта** (Currency): A dropdown menu with "UAH" selected.
- * Дата транзакції** (Transaction Date): A date picker field showing "2025-05-01".
- Опис** (Description): A text area containing "Зарплата за квітень" (Salary for May).

At the bottom of the form, there is a large blue button labeled "Створити транзакцію" (Create Transaction), and two smaller buttons labeled "Cancel" and "OK".

Рисунок 3.13 – Створення транзакції надходження коштів до бюджету

Створення транзакції витрат має схожі етапи як із доходами, але також потребує вибору категорії витрат. Хоча категорія є опціональною, але для ефективнішої організації бюджету краще додавати категорії до витрат. Категорію можна обрати серед тих, що були створені у секції «Категорії». Приклад створення транзакції з витратами та під'єднанням до неї категорії наведено на рис. 3.14.

Рисунок 3.14 – Створення транзакції з витратами із користувацькою категорією

Для створених транзакцій розроблено спеціальний функціонал для їх пошуку, фільтрування за категоріями, типами транзакцій, сумою транзакції та датою здійснення. Роботу фільтрів можна побачити на рис. 3.15.

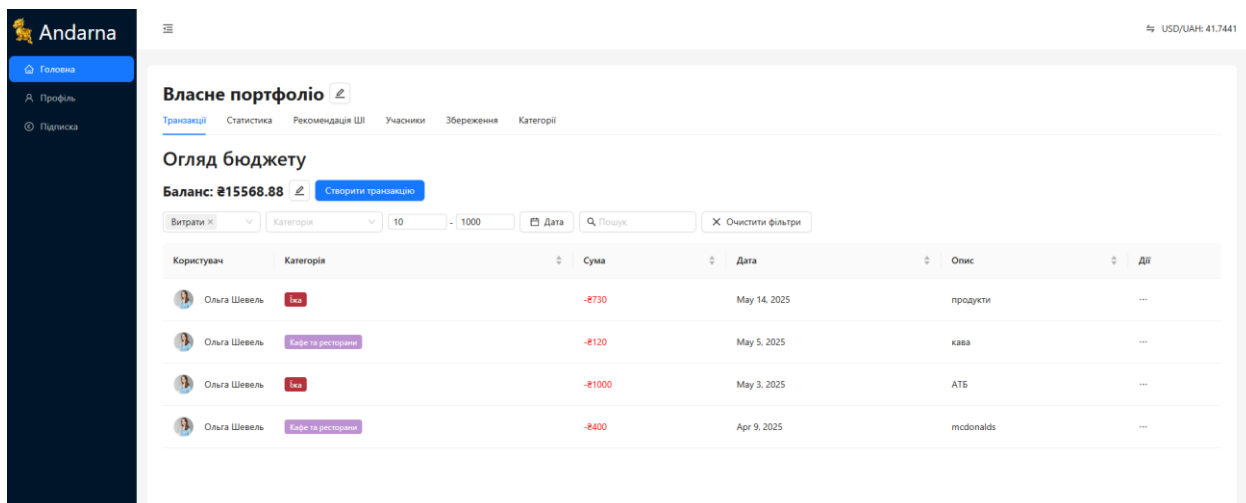


Рисунок 3.15 – Фільтрування транзакцій за типом та сумою

Для роботи із рахунками заощаджень потрібно спочатку створити такий рахунок у бюджеті. Для цього потрібно перейти на вкладку «Збереження» та натиснути кнопку «Створити рахунок заощаджень». Після потрібно заповнити дані для нового рахунку вказавши назву, валюту та ціль (тобто бажану цільову суму накопичень). Приклад створення рахунку заощаджень можна побачити на рис. 3.16.

Створити рахунок заощаджень

Назва

На машину

Валюта

USD

Ціль

5000

Зберегти

Рисунок 3.16 – Створення рахунку накопичень

Також є розроблений функціонал для імпорту накопичувальних рахунків з Monobank. Проте, щоб ця функція була доступна, користувачеві потрібно спочатку додати свій токен Monobank до профілю. Щоб це зробити, спочатку потрібно перейти на сторінку профілю (рис. 3.17).

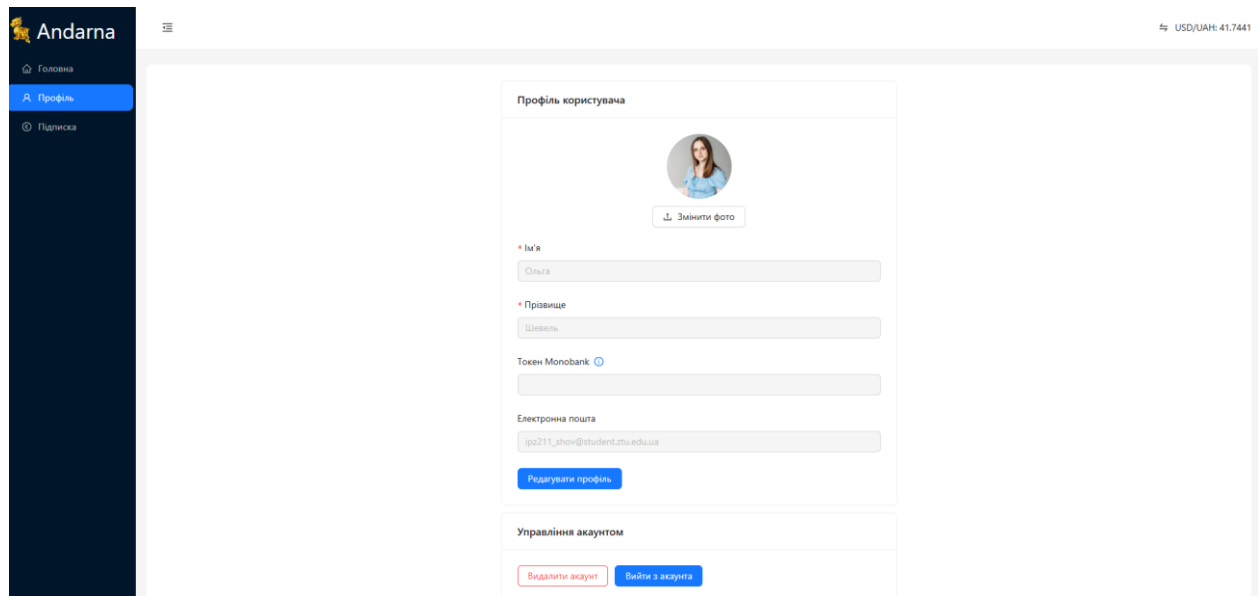


Рисунок 3.17 – Сторінка профілю користувача

Токен можна отримати перейшовши на сайт **<https://api.monobank.ua/index.html>** та авторизувавшись у ньому через застосунок Monobank (рис. 3.18). Після цього користувачеві буде відображено його персональний токен (рис. 3.19), який потрібно скопіювати та вставити у поле «Token Monobank» у застосунку планування бюджету на сторінці профілю (рис. 3.20).

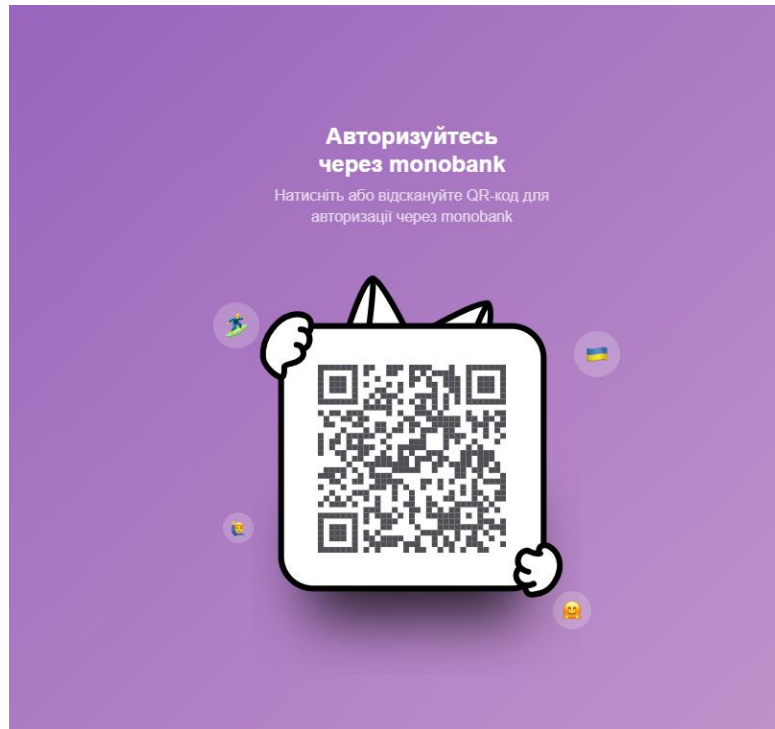


Рисунок 3.18 – Сторінка авторизації в сервісі Monobank

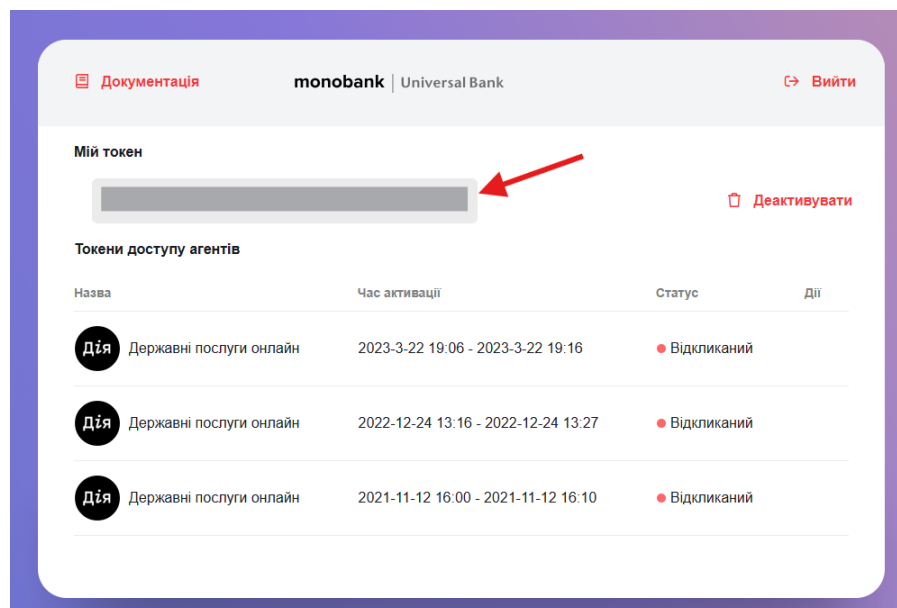


Рисунок 3.19 – Сторінка з отриманим токеном Monobank

Профіль користувача



Змінити фото

* Ім'я
Ольга

* Прізвище
Шевель

Токен Monobank ⓘ
.....

Електронна пошта
ipz211_shov@student.ztu.edu.ua

Скасувати Зберегти

Рисунок 3.20 – Додавання токenu Monobank до профілю користувача

Після збереження токenu та повернення на сторінку «Збереження» можна побачити, що з'явилася кнопка «Імпортувати банки з Monobank». При її натисненні з'являється список отриманих банок, які можна обрати для імпорту (рис. 3.21).

Банки Monobank

☒ Вибрати/зняти всі

☒ Подушка

Валюта: UAH
Ціль: Н/Д
Поточна сума: 12000

Скасувати Підтвердити

Рисунок 3.21 – Імпортування банок Monobank

Для поповнення заощаджень потрібно перейти на сторінку транзакцій та створити транзакцію із типом «Заощадження». У відповідному полі потрібно вибрати рахунок заощаджень, на який будуть нараховані кошти (рис. 3.22).

Створити транзакцію

* Тип транзакції
Заощадження

Рахунок заощаджень
На машину

* Сума
50

* Валюта
USD

* Курс обміну
41,7441

* Дата транзакції
2025-05-22

Опис

Створити транзакцію

Cancel OK

Рисунок 3.22 – Створення транзакції поповнення заощаджень

Щоб запросити іншого учасника до застосунку потрібно перейти до секції «Учасники» сторінки бюджету. Натиснувши кнопку «Запросити учасника» потрібно ввести адресу електронної пошти зареєстрованого користувача (рис. 3.23).

Власне портфоліо

Транзакції Статистика Рекомендації ЦІ Учасники 36

Запросити учасника

Запросити нового учасника

olha.shevel.dev@gmail.com

Cancel OK

Власник:
Ольга Шевель (p211_shov@student.ztu.edu.ua)

Очікувані запрошення

Електронна пошта	Термін дії	Надіслано
No data		

Рисунок 3.23 – Запрошення нового учасника

На пошту запрошеного учасника надійде лист із запрошенням (рис. 3.24). Його необхідно прийняти і тоді учасника буде додано до бюджету.

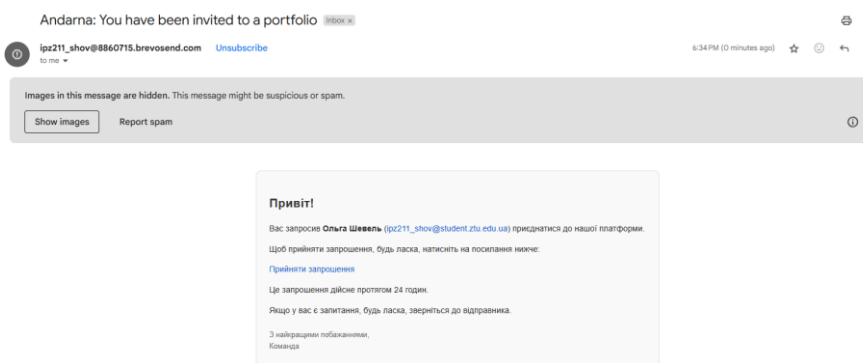


Рисунок 3.24 – Лист запрошення нового учасника

Після успішного прийняття запрошення нового учасника додано до бюджету (рис. 3.25).

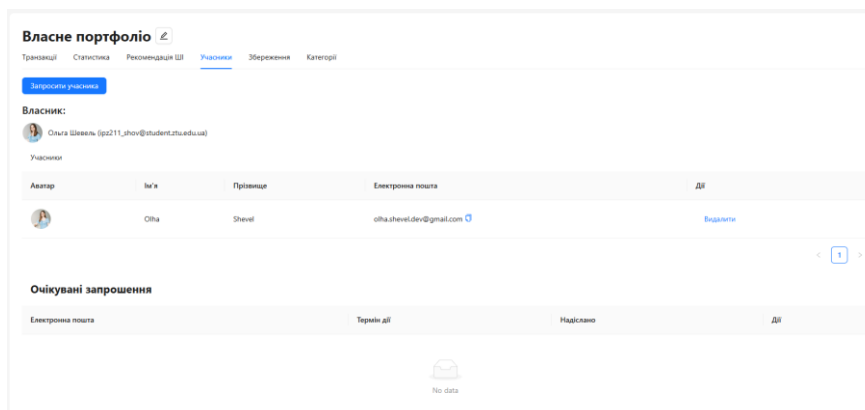


Рисунок 3.25 – Список учасників бюджету

На сторінці «Статистика» (рис. 3.26) учасники можуть побачити статистику витрат по категоріях та учасниках та також графік зміни доходів.

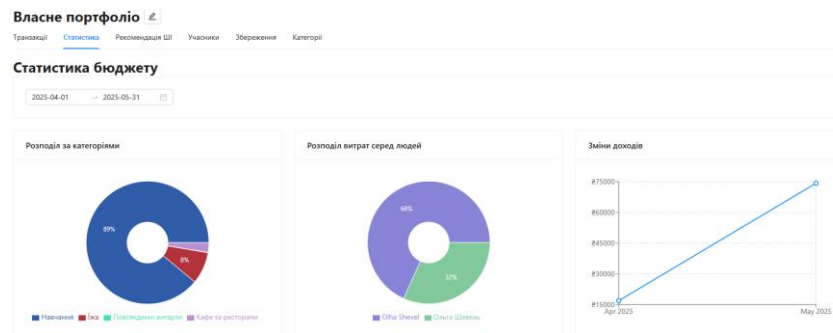


Рисунок 3.26 – Сторінка статистики бюджету

Для отримання рекомендації від ШІ потрібно перейти до секції «Рекомендація ШІ» та натиснути на кнопку «Згенерувати AI-рекомендацію». Цей процес може зайняти до декількох хвилин, тож варто зачекати. Сторінку з рекомендаціями можна побачити на рис. 3.27.

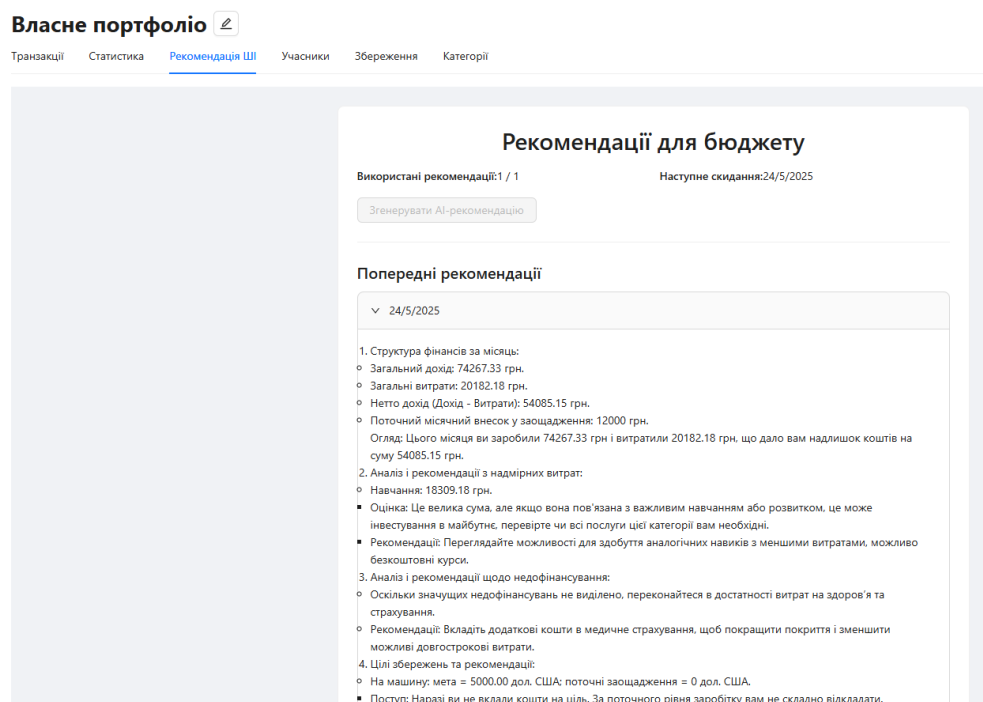


Рисунок 3.27 – Сторінка з рекомендаціями від ШІ

Останньою важливою частиною розробленого функціоналу є оформлення платної підписки. Для цього потрібно перейти на сторінку «Підписка», її оформлення можна побачити на рис. 3.28.

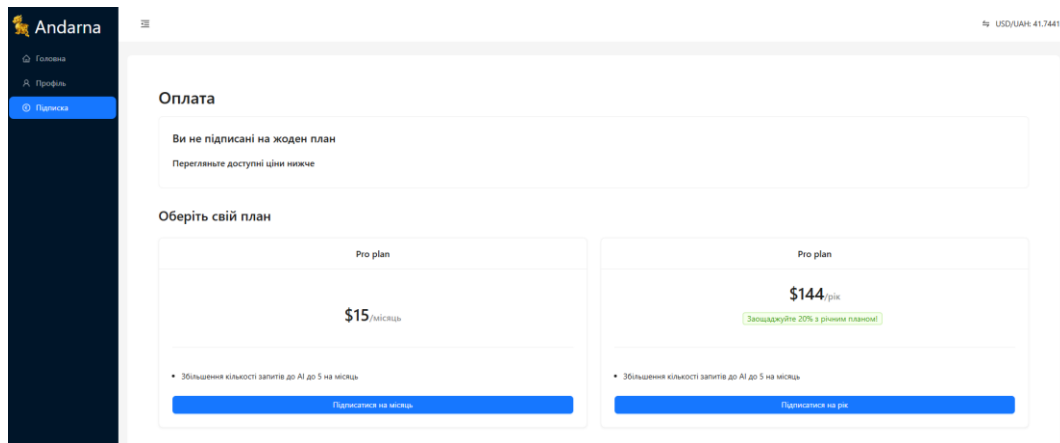


Рисунок 3.28 – Сторінка оформлення підписки

Користувач обирає потрібний йому план та натискає кнопку «Підписатися». Його перенаправлено на сторінку Stripe, де потрібно ввести дані карти та здійснити транзакцію (рис. 3.29).

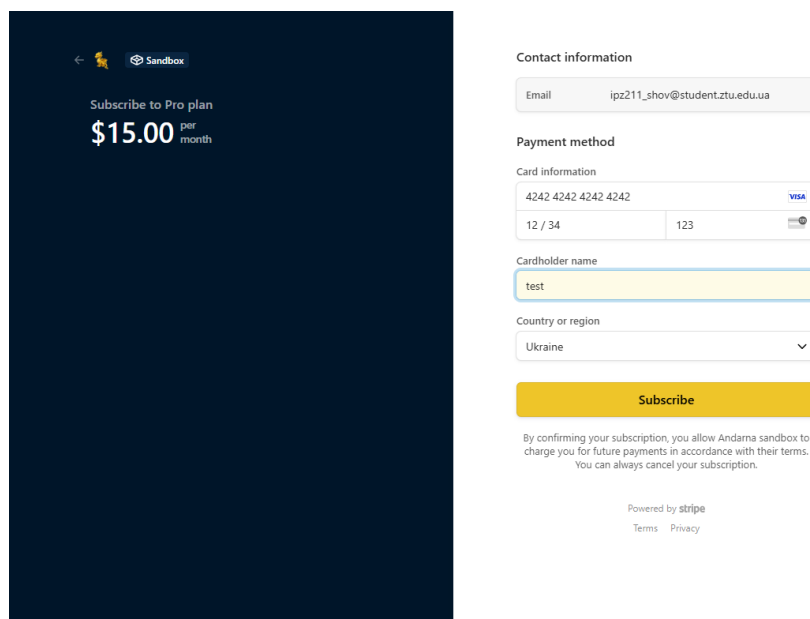


Рисунок 3.29 – Сторінка здійснення оплати

Після успішної оплати користувача перенаправлено назад на сторінку підписок, де відображено статус щойно придбаної підписки. Приклад наведено на рис. 3.30.

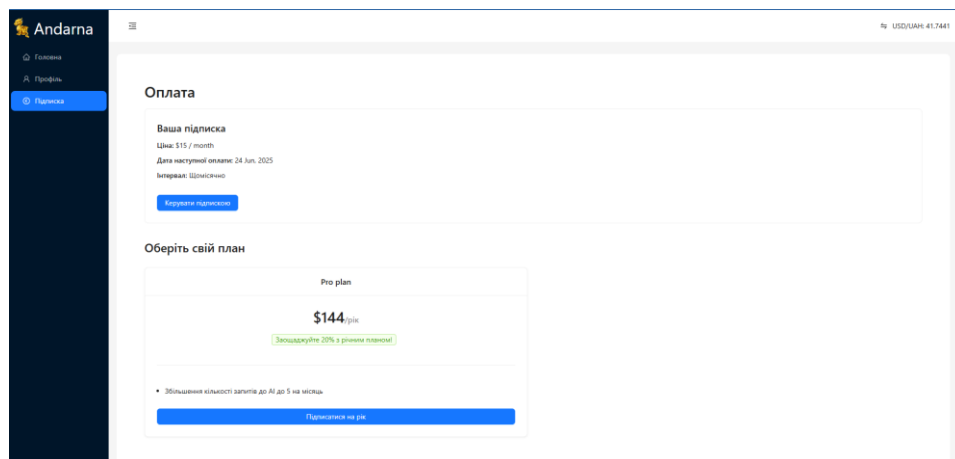


Рисунок 3.30 – Оформлена підписка

3.3 Тестування роботи програмного продукту

Для перевірки коректності роботи вебзастосунку було проведено мануальне тестування на відповідність функціональним вимогам. Було створено тест-кейси та перевірено роботу додатку відповідно наведеним у них інструкціям. Тест-кейси наведено в таблицях 3.1 – 3.14.

Таблиця 3.1

Тест-кейс 1.1

Тест	Реєстрація користувача з валідними даними
Модуль	Авторизація
Номер тесту	1.1
Початковий стан системи	Відкрито сторінку реєстрації. Користувач неавторизований.
Опис проведення тесту	Заповнити форму реєстрації валідними ім'ям, email та паролем. Натиснути «Зареєструватися».
Очікуваний результат	Користувач успішно зареєстрований та перенаправлений до головної сторінки додатку.
Фактичний результат	Як очікувалося

Таблиця 3.2

Тест-кейс 1.2

Тест	Авторизація через email і пароль
Модуль	Авторизація
Номер тесту	1.2
Початковий стан системи	Користувач вже зареєстрований в системі. Відкрито сторінку входу.
Опис проведення тесту	Ввести email і пароль. Натиснути «Увійти».

Продовження таблиці 3.2

Очікуваний результат	Користувач успішно авторизований та перенаправлений до панелі керування бюджетом.
Фактичний результат	Як очікувалося

Таблиця 3.3

Тест-кейс 1.3

Тест	Авторизація через Google
Модуль	Авторизація
Номер тесту	1.3
Початковий стан системи	Користувач неавторизований. Відкрито сторінку входу.
Опис проведення тесту	Натиснути кнопку «Увійти через Google». Авторизуватися через Google.
Очікуваний результат	Користувач успішно авторизований та перенаправлений до панелі керування бюджетом.
Фактичний результат	Як очікувалося

Таблиця 3.4

Тест-кейс 1.4

Тест	Відновлення паролю
Модуль	Авторизація
Номер тесту	1.4
Початковий стан системи	Користувач неавторизований. Відкрито сторінку входу
Опис проведення тесту	Натиснути «Забули пароль?», ввести email, підтвердити запит на відновлення
Очікуваний результат	На email надіслано лист для зміни пароля
Фактичний результат	Як очікувалося

Таблиця 3.5

Тест-кейс 2.1

Тест	Створення транзакції
Модуль	Транзакції
Номер тесту	2.1
Початковий стан системи	Користувач авторизований та має активний бюджет.
Опис проведення тесту	Відкрити сторінку бюджету, натиснути «Додати транзакцію», ввести дані, зберегти.
Очікуваний результат	Транзакція збережена і з'явилася у списку транзакцій. Баланс бюджету змінився.
Фактичний результат	Як очікувалося

Таблиця 3.6

Тест-кейс 2.3

Тест	Видалення транзакції
Модуль	Транзакції
Номер тесту	2.3
Початковий стан системи	У бюджеті є хоча б одна транзакція.
Опис проведення тесту	Натиснути кнопку видалення біля транзакції, підтвердити видалення.
Очікуваний результат	Транзакцію видалено зі списку. Баланс бюджету змінився.
Фактичний результат	Як очікувалося

Таблиця 3.7

Тест-кейс 3.1

Тест	Додавання нової категорії
Модуль	Категорії
Номер тесту	3.1
Початковий стан системи	Користувач авторизований, бюджет активний
Опис проведення тесту	Користувач натискає «Створити категорію», вводить назву, натискає «Зберегти»
Очікуваний результат	Категорію збережено та відображено в доступних при додаванні транзакцій
Фактичний результат	Як очікувалося

Таблиця 3.8

Тест-кейс 3.2

Тест	Редагування категорії
Модуль	Категорії
Номер тесту	3.2
Початковий стан системи	У користувача є хоча б одна створена категорія
Опис проведення тесту	Користувач натискає «Редагувати», змінює назву категорії та колір
Очікуваний результат	Назву та колір категорії оновлено
Фактичний результат	Як очікувалося

Таблиця 3.9

Тест-кейс 3.3

Тест	Видалення існуючої категорії
Модуль	Категорії
Номер тесту	3.3
Початковий стан системи	У користувача є хоча б одна створена категорія
Опис проведення тесту	Користувач натискає «Видалити» біля категорії

Продовження таблиці 3.9

Очікуваний результат	Категорію видалено, вона більше не відображається у списку доступних категорій. Усі транзакції, що були віднесені до цієї категорії переходять до категорії «Невідомо»
Фактичний результат	Як очікувалося

Таблиця 3.10

Тест-кейс 4.1

Тест	Отримання фінансової поради
Модуль	Рекомендації від ІІІ
Номер тесту	4.1
Початковий стан системи	Користувач авторизований, має транзакції в бюджеті
Опис проведення тесту	Користувач натискає «Отримати пораду», система аналізує дані
Очікуваний результат	Користувач отримує текстову пораду
Фактичний результат	Як очікувалося

Таблиця 3.11

Тест-кейс 4.1

Тест	Отримання фінансової поради
Модуль	Рекомендації від ІІІ
Номер тесту	4.1
Початковий стан системи	Користувач авторизований, має транзакції в бюджеті
Опис проведення тесту	Користувач натискає «Отримати пораду», система аналізує дані
Очікуваний результат	Користувач отримує текстову пораду
Фактичний результат	Як очікувалося

Таблиця 3.12

Тест-кейс 5.1

Тест	Оформлення платної підписки
Модуль	Підписка користувача
Номер тесту	5.1
Початковий стан системи	Користувач не має активної підписки
Опис проведення тесту	Користувач обирає тариф, вводить дані карти, натискає «Оплатити»
Очікуваний результат	Користувач отримує підтвердження оплати, статус підписки оновлюється
Фактичний результат	Як очікувалося

Таблиця 3.13

Тест-кейс 6.1

Тест	Створення накопичувального акаунту
Модуль	Накопичення
Номер тесту	6.1
Початковий стан системи	Користувач має активний бюджет та перебуває на вкладці «Збереження»
Опис проведення тесту	Користувач натискає «Додати», вводить назву, цільову суму та валюту
Очікуваний результат	Створено накопичувальну ціль, відображено у списку
Фактичний результат	Як очікувалося

Таблиця 3.14

Тест-кейс 6.2

Тест	Видалення накопичувальної цілі
Модуль	Накопичення
Номер тесту	6.2
Початковий стан системи	Користувач має створений накопичувальний акаунт
Опис проведення тесту	Користувач натискає «Видалити» поруч із акаунтом
Очікуваний результат	Акаунт зникає зі списку
Фактичний результат	Як очікувалося

Висновки до третього розділу

У цьому розділі було детально розглянуто проектування системи вебзастосунку «Andarna». Наведено діаграму компонентів, яка ілюструє взаємодію клієнтської частини (UI), модуля авторизації, серверної частини (бекенду), бази даних PostgreSQL, кешу Redis та сторонніх API (Monobank, НБУ, Stripe, OpenAI). Також представлена діаграма розгортання, що показує розміщення React-додатку на вебсервері Nginx, окреме розгортання серверної частини та їх взаємодію через HTTPS, а також зв'язок бекенду з базою даних по TCP/IP.

У третьому розділі було детально розглянуто проектування системи вебзастосунку «Andarna». Наведено діаграму компонентів, яка ілюструє взаємодію клієнтської частини (UI), модуля авторизації, серверної частини (бекенду), бази даних PostgreSQL, кешу Redis та сторонніх API (Monobank, НБУ, Stripe, OpenAI). Також представлена діаграма розгортання, що показує

розміщення React-додатку на вебсервері Nginx, окреме розгортання серверної частини та їх взаємодію через HTTPS, а також зв'язок бекенду з базою даних по TCP/IP.

Значну увагу приділено структурі інтерфейсу та порядку взаємодії користувача із застосунком. Детально, з наведенням прикладів у вигляді зображень екранів, було описано ключові сценарії послідовно.

На завершення розділу було проведено мануальне тестування розробленого функціоналу на відповідність функціональним вимогам. Було створено та виконано тест-кейси для основних модулів системи, таких як. Результати тестування підтвердили коректність роботи реалізованих функцій.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було успішно спроектовано та розроблено вебзастосунок «Andarna», призначений для комплексного планування та управління сімейним бюджетом. Метою роботи було створення програмного продукту, що дозволяє користувачам ефективно контролювати доходи та витрати, аналізувати фінансові потоки, спільно працювати над бюджетами та отримувати інтелектуальні поради для покращення свого фінансового стану. Мету було досягнуто шляхом послідовного вирішення поставлених завдань.

На першому етапі було проведено всебічний аналіз предметної області фінансового планування, розглянуто актуальні методики ведення бюджету та досліджено існуючі на ринку програмні рішення, такі як «Honeydue», «PocketGuard» та «Spendee». Це дозволило визначити ключові функціональні потреби користувачів та сформулювати унікальні переваги майбутнього застосунку, зокрема розширені можливості для спільного бюджетування, інтеграцію з українськими банківськими сервісами та використання штучного інтелекту для надання рекомендацій. На основі аналізу було визначено детальну постановку задачі та основні вимоги до системи.

У процесі дослідження та проєктування було обґрунтовано вибір клієнт-серверної архітектури з використанням підходу монорепозиторію, що забезпечує гнучкість розробки, можливість перевикористання коду та масштабованість системи. Було визначено стек технологій: для клієнтської частини обрано TypeScript та React.js з бібліотекою компонентів Ant Design; для серверної частини – Nest.js на платформі Node.js з використанням TypeScript, реляційної бази даних PostgreSQL та системи управління базами даних TypeORM, а також Redis для кешування. Для реалізації специфічного функціоналу було інтегровано сторонні сервіси: Firebase Authentication для автентифікації, Stripe для обробки платежів за підписку, OpenAI API (модель

GPT-4-turbo) для генерації фінансових порад, Brevo для надсилання електронних листів та публічне API Monobank для імпорту даних.

Проектування системи включало розробку варіантів використання, що охоплюють увесь спектр взаємодії користувачів із застосунком, від реєстрації до управління складними фінансовими операціями та отримання аналітики. Було сформульовано функціональні та нефункціональні вимоги до системи. Особливу увагу було приділено проектуванню бази даних: розроблено ER-діаграму та описано структуру основні таблиці для зберігання даних про користувачів, бюджети, транзакції, категорії, накопичувальні рахунки, підписки, запрошення та рекомендації.

Розроблено інтуїтивно зрозумілий користувацький інтерфейс, детально описано порядок взаємодії користувача з усіма модулями системи та проведено мануальне тестування, яке підтвердило відповідність реалізованого функціоналу поставленим завданням та вимогам.

Практична цінність отриманих результатів полягає у створенні повноцінного вебзастосунку, що надає користувачам потужний та зручний інструмент для управління сімейними фінансами. «Andarna» допомагає підвищити фінансову обізнаність, оптимізувати витрати, ефективно накопичувати кошти та досягати фінансових цілей, що є особливо актуальним в умовах сучасної економічної нестабільності.

Напрямами подальшого наукового дослідження та вдосконалення програмного комплексу можуть бути: розширення функціоналу шляхом додавання нових інструментів аналітики та прогнозування та поглиблена інтеграція з більшою кількістю українських банків та платіжних систем.

Таким чином, проведена робота заклала міцну основу для подальшого розвитку та впровадження вебзастосунку «Andarna» як ефективного помічника в управлінні особистими та сімейними фінансами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Метод 50/30/20. Nerdwallet [Електронний ресурс] – Режим доступу до ресурсу: <https://www.nerdwallet.com/article/finance/how-to-choose-the-right-budget-system>
2. Метод «конвертів». Ucap IO [Електронний ресурс] – Режим доступу до ресурсу: <https://ucap.io/envelope-budgeting/>
3. Метод «нульового бюджету». Nerdwallet [Електронний ресурс] – Режим доступу до ресурсу: <https://www.nerdwallet.com/article/finance/zero-based-budgeting-explained>
4. Crabtree M. 10 Ways to Use ChatGPT for Finance. Datacamp. Artificial Intelligence. 20.08.2024 [Електронний ресурс] – Режим доступу до ресурсу: <https://www.datacamp.com/blog/10-ways-to-use-chatgpt-for-finance>
5. Honeydue [Електронний ресурс] – Режим доступу до ресурсу: <https://www.honeydue.com/>
6. PocketGuard [Електронний ресурс] – Режим доступу до ресурсу: <https://pocketguard.com/>
7. Spendee [Електронний ресурс] – Режим доступу до ресурсу: <https://www.spendee.com/>
8. Client Server Architecture. Cheriton School Of Computer Science – University Of Waterloo. 22.07.2019 [Електронний ресурс] – Режим доступу до ресурсу: https://cs.uwaterloo.ca/~m2nagapp/courses/CS446/1195/Arch_Design_Activity/ClientServer.pdf
9. Microservice Architecture [Електронний ресурс] – Режим доступу до ресурсу: <https://microservices.io/patterns/microservices.html>

10. Полірепозиторій та монорепозиторій. Kinsta [Електронний ресурс] – Режим доступу до ресурсу: <https://kinsta.com/blog/monorepo-vs-multi-repo/>
11. Clean code with Node.js [Електронний ресурс] – Режим доступу до ресурсу: <https://javascript.plainenglish.io/clean-code-with-node-js-994e9b6b7e56>
12. Офіційна документація JavaScript [Електронний ресурс] – Режим доступу до ресурсу: <https://uk.javascript.info/>
13. Офіційна документація TypeScript [Електронний ресурс] – Режим доступу до ресурсу: <https://www.typescriptlang.org/>
14. Офіційна документація Vue.js [Електронний ресурс] – Режим доступу до ресурсу: <https://vuejs.org/>
15. Офіційна документація Angular [Електронний ресурс] – Режим доступу до ресурсу: <https://angular.dev/>
16. Офіційна документація React [Електронний ресурс] – Режим доступу до ресурсу: <https://react.dev/>
17. Офіційна документація Material UI [Електронний ресурс] – Режим доступу до ресурсу: <https://mui.com/material-ui/>
18. Офіційна документація Chakra UI [Електронний ресурс] – Режим доступу до ресурсу: <https://chakra-ui.com/>
19. Офіційна документація Ant Design [Електронний ресурс] – Режим доступу до ресурсу: <https://ant.design/>
20. Офіційна документація Turborepo [Електронний ресурс] – Режим доступу до ресурсу: <https://turborepo.com/>
21. Офіційна документація Nx [Електронний ресурс] – Режим доступу до ресурсу: <https://nx.dev/>
22. Офіційна документація Auth0 [Електронний ресурс] – Режим доступу до ресурсу: <https://auth0.com/>

- 23.Документація Firebase Authentication [Електронний ресурс] – Режим доступу до ресурсу: <https://firebase.google.com/docs/auth>
- 24.Офіційна документація Express.js [Електронний ресурс] – Режим доступу до ресурсу: <https://expressjs.com/>
- 25.Офіційна документація NestJS [Електронний ресурс] – Режим доступу до ресурсу: <https://nestjs.com/>
- 26.Офіційна документація MongoDB [Електронний ресурс] – Режим доступу до ресурсу: <https://www.mongodb.com/>
- 27.Офіційна документація MySQL [Електронний ресурс] – Режим доступу до ресурсу: <https://www.mysql.com/>
- 28.Документація PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: <https://www.postgresql.org/>
- 29.Документація Sequelize [Електронний ресурс] – Режим доступу до ресурсу: <https://sequelize.org/>
- 30.Офіційна документація Prisma [Електронний ресурс] – Режим доступу до ресурсу: <https://www.prisma.io/>
- 31.Офіційна документація TypeORM [Електронний ресурс] – Режим доступу до ресурсу: <https://typeorm.io/>
- 32.PayPal [Електронний ресурс] – Режим доступу до ресурсу: <https://www.paypal.com/>
- 33.Stripe [Електронний ресурс] – Режим доступу до ресурсу: <https://stripe.com/>
- 34.Офіційна документація OpenAI API [Електронний ресурс] – Режим доступу до ресурсу: <https://openai.com/docs/api>
- 35.Офіційна документація SendGrid [Електронний ресурс] – Режим доступу до ресурсу: <https://sendgrid.com/>
- 36.Офіційна документація Brevo [Електронний ресурс] – Режим доступу до ресурсу: <https://www.brevo.com/>

- 37.Офіційна документація Mailgun [Електронний ресурс] – Режим доступу до ресурсу: <https://www.mailgun.com/>
- 38.Офіційна документація Ts-rest [Електронний ресурс] – Режим доступу до ресурсу: <https://ts-rest.com/>
- 39.Офіційна документація API Monobank [Електронний ресурс] – Режим доступу до ресурсу: <https://api.monobank.ua/index.html>

ДОДАТКИ

					ІІЗ.КР.Б – 121 – 25 – ІІЗ	
						99

Додаток А – Програмна реалізація окремих модулів

Модуль авторизації

apps/frontend/src/providers/AuthProvider.tsx

```
import {
  ReactNode,
  createContext,
  useContext,
  useEffect,
  useState,
} from 'react';
import { auth, googleAuth } from '../firebase';
import {
  User,
  createUserWithEmailAndPassword,
  onAuthStateChanged,
  signInWithEmailAndPassword,
  signInWithPopup,
  signOut,
  updateProfile,
} from 'firebase/auth';
import { ApiClientContext } from '../ApiClientProvider';

interface AuthProviderProps {
  children: ReactNode;
}

interface AuthContextType {
  user: User | null;
}
```

```

loading: boolean;
idToken: string | null;
signIn: (email: string, password: string) => Promise<any>;
signUp: (
  email: string,
  password: string,
  firstName: string,
  lastName: string
) => Promise<any>;
logOut: () => Promise<void>;
signInWithGoogle: () => Promise<any>;
}

export const AuthContext = createContext<AuthContextType>({} as any);

const AuthProvider = ({ children }: AuthProviderProps) => {
  const { contractQueryClient } = useContext(ApiClientContext);
  const [user, setUser] = useState<User | null>(null);
  const [loading, setLoading] = useState(true);
  const [idToken, setIdToken] = useState<string | null>(null);

  const fetchToken = async (currentUser: User | null) => {
    if (currentUser) {
      const token = await currentUser.getIdToken();
      setIdToken(token);
    } else {
      setIdToken(null);
    }
  }
}

```

```

};

const signUp = async (
  email: string,
  password: string,
  firstName: string,
  lastName: string
) => {
  try {
    const userCredentials = await createUserWithEmailAndPassword(
      auth,
      email,
      password
    );
    await updateProfile(userCredentials.user, {
      displayName: `${firstName} ${lastName}`,
    });
  } catch (e: any) {
    // TODO error handling on the screen
    console.log(e.message);
  }
};

const signIn = async (email: string, password: string) => {
  try {
    await signInWithEmailAndPassword(auth, email, password);
  } catch (e: any) {
    // TODO error handling on the screen

```

```

    console.log(e.message);
  }
};

const signInWithGoogle = async () => {
  try {
    await signInWithPopup(auth, googleAuth);
  } catch (e: any) {
    // TODO error handling on the screen
    console.log(e.message);
  }
};

const logOut = async () => {
  return signOut(auth);
};

useEffect(() => {
  const unsubscribe = onAuthStateChanged(auth, async (currentUser) => {
    if (currentUser) {
      setUser(currentUser);
      await fetchToken(currentUser);
    }
    setLoading(false);
  });
  return () => {
    unsubscribe();
  };
});

```

```

    }, [contractQueryClient]);

const authValue = {
  user,
  loading,
  idToken,
  logOut,
  signUp,
  signIn,
  signInWithGoogle,
};
console.log(authValue);

return (
  <AuthContext.Provider value={authValue}>{children}</AuthContext.Provider>
);
};

export default AuthProvider;

apps/backend/src/app/auth/middlewares/auth.middleware.ts
import { Injectable, NestMiddleware } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Response } from 'express';
import * as firebase from 'firebase-admin';
import { Repository } from 'typeorm';
import { FirebaseApp } from '../../../firebase/firebase-app';
import { ExtendedRequest } from '../../../interface/user.interface';

```

```

import { User } from '../user/user.entity';

@Injectable()
export class AuthMiddleware implements NestMiddleware {
  private auth: firebase.auth.Auth;

  constructor(
    private firebaseApp: FirebaseApp,
    @InjectRepository(User)
    private userRepository: Repository<User>
  ) {
    this.auth = firebaseApp.getAuth();
  }

  use(req: ExtendedRequest, res: Response, next: () => void) {
    const token = req.headers.authorization;
    if (token != null && token != "") {
      this.auth
        .verifyIdToken(token.split(' ')[1])
        .then(async (decodedToken) => {
          const userFromDb = await this.userRepository.findOne({
            where: {
              firebaseId: decodedToken.uid,
            },
            select: {
              id: true,
              firstName: true,
              lastName: true,

```

```

        email: true,
        firebaseId: true,
        photoURL: true,
        monobankToken: true,
        recommendationsAllowed: true,
        recommendationsUsed: true,
        lastCreditsReset: true,
        createdAt: true,
        updatedAt: true,
    },
});

req.user = userFromDb;
req.firebasePayload = decodedToken;
next();
})
.catch((e) => {
    console.log(e);

    next();
});
} else {
    next();
}
}
}

```

apps/backend/src/app/auth/guards/auth.guard.ts

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';
import { ExtendedRequest } from '../interface/user.interface';
```

```
@Injectable()
```

```
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request: ExtendedRequest = context.switchToHttp().getRequest();
    if (!request.user) {
      return false;
    }
    return true;
  }
}
```

```
apps/backend/src/app/decorators/userFromToken.decorator.ts
```

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';
import { ExtendedRequest } from '../interface/user.interface';
```

```
export const UserFromToken = createParamDecorator(
  (data: void, ctx: ExecutionContext) => {
    const req: ExtendedRequest = ctx.switchToHttp().getRequest();
    const decodedUser = req.firebasePayload;
    return decodedUser;
  }
);
```

apps/backend/src/app/decorators/userFromDb.decorator.ts

```
import { createParamDecorator, ExecutionContext } from '@nestjsjs/common';
```

```
import { ExtendedRequest } from '../interface/user.interface';
```

```
export const CurrentUser = createParamDecorator(
```

```
  (data: string, ctx: ExecutionContext) => {
```

```
    const req: ExtendedRequest = ctx.switchToHttp().getRequest();
```

```
    const user = req.user;
```

```
    return data ? (user as any)[data] : user;
```

```
  }
```

```
);
```

apps/frontend/src/providers/AuthenticatedUserProvider.tsx

```
import { IUser } from '@andarna/common';
```

```
import {
```

```
  createContext,
```

```
  useCallback,
```

```
  useContext,
```

```
  useEffect,
```

```
  useState,
```

```
} from 'react';
```

```
import { ApiClientContext } from './ApiClientProvider';
```

```
export type AuthenticatedUserContextType = {
```

```
  user: IUser | null;
```

```

    isAuthenticated: boolean;
    refetchUser: () => Promise<void>;
  };

```

```

const                                     AuthenticatedUserContext                                     =
createContext<AuthenticatedUserContextType>({
  isAuthenticated: true,
  user: null,
  // eslint-disable-next-line @typescript-eslint/no-empty-function
  refetchUser: async () => {},
});

```

```

const AuthenticatedUserProvider = ({
  children,
}: {
  children: React.ReactNode;
}) => {
  const [user, setUser] = useState<IUser | null>(null);
  const [isAuthenticated, setIsAuthenticated] = useState(true);
  const { contractQueryClient, isInitializing } = useContext(ApiClientContext);

  const fetchUser = useCallback(async () => {
    try {
      if (isInitializing) return; // Wait until API client is initialized
      if (!contractQueryClient) {
        setIsAuthenticated(false);
        return;
      }
    }
  });

```

```

const { status, body } =
  await contractQueryClient.users.findOrCreate.query();

if (status === 200) {
  setUser(body.data);
}
} catch (e) {
  console.log(e);
} finally {
  setIsAuthenticating(false);
}
}, [contractQueryClient, isInitializing]);

useEffect(() => {
  fetchUser();
}, [fetchUser]);

const refetchUser = useCallback(async () => {
  setIsAuthenticating(true);
  await fetchUser();
  setIsAuthenticating(false);
}, [fetchUser]);

return (
  <AuthenticatedUserContext.Provider
    value={{ user, isAuthenticated, refetchUser }}
  >

```

```

    {children}
  </AuthenticatedUserContext.Provider>
);
};

export { AuthenticatedUserContext, AuthenticatedUserProvider };

apps/frontend/src/firebase/index.ts
// Import the functions you need from the SDKs you need
import { initializeApp } from 'firebase/app';
import { getAuth, GoogleAuthProvider } from 'firebase/auth';
import { getStorage } from 'firebase/storage';

// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: process.env.NX_PUBLIC_FIREBASE_API_KEY,
  authDomain: process.env.NX_PUBLIC_FIREBASE_AUTH_DOMAIN,
  projectId: process.env.NX_PUBLIC_FIREBASE_PROJECT_ID,
  storageBucket: process.env.NX_PUBLIC_FIREBASE_STORAGE_BUCKET,
  messagingSenderId:
process.env.NX_PUBLIC_FIREBASE_MESSAGING_SENDER_ID,
  appId: process.env.NX_PUBLIC_FIREBASE_APP_ID,
};

// Initialize Firebase

```

```

export const app = initializeApp(firebaseConfig);
export const auth = getAuth(app);
export const storage = getStorage(app);
export const googleAuth = new GoogleAuthProvider();

```

Модуль CRUD операцій на прикладі категорій

libs/common/src/lib/contracts/category.contract.ts

```

import { initContract } from '@ts-rest/core';
import { z } from 'zod';
import { createErrorSchema, createResponseSchema } from '../utils';

```

```

const c = initContract();

```

```

export const CategorySchema = z.object({
  id: z.string().uuid(),
  name: z.string(),
  color: z.string(), // Assuming color is a hex code
  createdAt: z.date(),
  updatedAt: z.date(),
});

```

```

export type CategoryContract = z.infer<typeof CategorySchema>;

```

```

export const categoriesContract = c.router({
  getCategories: {
    method: 'GET',
    path: '/portfolios/:portfolioId/categories',

```

```

pathParams: z.object({
  portfolioId: z.string().uuid(),
}),
responses: {
  200: createResponseSchema(z.array(CategorySchema)),
  403: createErrorSchema(),
  404: createErrorSchema(),
},
},
createCategory: {
  method: 'POST',
  path: '/portfolios/:portfolioId/categories',
  pathParams: z.object({
    portfolioId: z.string().uuid(),
  }),
  body: CategorySchema.omit({
    id: true,
    createdAt: true,
    updatedAt: true,
  }),
  responses: {
    201: createResponseSchema(CategorySchema),
    400: createErrorSchema(),
    403: createErrorSchema(),
    404: createErrorSchema(),
  },
},
updateCategory: {

```

```

method: 'PUT',
path: '/portfolios/:portfolioId/categories/:categoryId',
pathParams: z.object({
  portfolioId: z.string().uuid(),
  categoryId: z.string().uuid(),
}),
body: CategorySchema.omit({
  id: true,
  createdAt: true,
  updatedAt: true,
}),
responses: {
  200: createResponseSchema(CategorySchema),
  400: createErrorSchema(),
  403: createErrorSchema(),
  404: createErrorSchema(),
},
},
deleteCategory: {
  method: 'DELETE',
  path: '/portfolios/:portfolioId/categories/:categoryId',
  pathParams: z.object({
    portfolioId: z.string().uuid(),
    categoryId: z.string().uuid(),
  }),
  responses: {
    204: createResponseSchema(z.void()),
    403: createErrorSchema(),
  }
}

```

```

        404: createErrorSchema(),
      },
    },
  });

```

```

apps/backend/src/app/outcome_categories/outcomeCategory.controller.ts
import { contract } from '@andarna/common';
import { Controller, UseGuards } from '@nestjs/common';
import { tsRestHandler, TsRestHandler } from '@ts-rest/nest';
import { AuthGuard } from '../auth/guards/auth.guard';
import { CurrentUser } from '../decorators/currentUser.decorator';
import { PortfolioService } from '../portfolios/portfolio.service';
import { User } from '../user/user.entity';
import { createErrorObject, createResponseObject } from '../utils/response';
import { OutcomeCategoryService } from '../outcomeCategory.service';

```

```
@Controller()
```

```
@UseGuards(AuthGuard)
```

```
export class OutcomeCategoryController {
```

```
  constructor(
```

```
    private readonly outcomeCategoryService: OutcomeCategoryService,
```

```
    private readonly portfolioService: PortfolioService
```

```
  ) {}
```

```
@TsRestHandler(contract.categories.getCategories)
```

```
async getCategories(@CurrentUser() user: User) {
```

```
  return tsRestHandler(
```

```
    contract.categories.getCategories,
```

```

async ({ params: { portfolioId } }) => {
  const portfolio = await this.portfolioService.verifyPortfolioAccess(
    user.id,
    portfolioId
  );
  if (!portfolio) {
    return {
      status: 403,
      body: createErrorObject({
        message: 'You do not have access to this portfolio',
        code: 'FORBIDDEN',
      }),
    };
  }
  const categories = await this.outcomeCategoryService.findAllInPortfolio(
    portfolioId
  );

  return {
    status: 200,
    body: createResponseObject(categories),
  };
};
}

@TsRestHandler(contract.categories.createCategory)
async createCategory(@CurrentUser() user: User) {

```

```

return tsRestHandler(
  contract.categories.createCategory,
  async ({ params: { portfolioId }, body }) => {
    const portfolio = await this.portfolioService.verifyPortfolioAccess(
      user.id,
      portfolioId
    );
    if (!portfolio) {
      return {
        status: 403,
        body: createErrorObject({
          message: 'You do not have access to this portfolio',
          code: 'FORBIDDEN',
        }),
      };
    }
    const category = await this.outcomeCategoryService.create(
      portfolioId,
      body
    );

    return {
      status: 201,
      body: createResponseObject(category),
    };
  }
);
}

```

```

@TsRestHandler(contract.categories.updateCategory)
async updateCategory(@CurrentUser() user: User) {
  return tsRestHandler(
    contract.categories.updateCategory,
    async ({ params: { portfolioId, categoryId }, body }) => {
      const portfolio = await this.portfolioService.verifyPortfolioAccess(
        user.id,
        portfolioId
      );
      if (!portfolio) {
        return {
          status: 403,
          body: createErrorObject({
            message: 'You do not have access to this portfolio',
            code: 'FORBIDDEN',
          }),
        };
      }
      const category = await this.outcomeCategoryService.verifyCategory(
        portfolioId,
        categoryId
      );
      if (!category) {
        return {
          status: 404,
          body: createErrorObject({
            message: 'Category not found',

```

```

        code: 'NOT_FOUND',
      }),
    };
  }
  const updatedCategory = await this.outcomeCategoryService.update(
    category,
    body
  );
  return {
    status: 200,
    body: createResponseObject(updatedCategory),
  };
}
);
}

```

```

@TsRestHandler(contract.categories.deleteCategory)
async deleteCategory(@CurrentUser() user: User) {
  return tsRestHandler(
    contract.categories.deleteCategory,
    async ({ params: { portfolioId, categoryId } }) => {
      const portfolio = await this.portfolioService.verifyPortfolioAccess(
        user.id,
        portfolioId
      );
      if (!portfolio) {
        return {
          status: 403,

```

```

    body: createErrorObject({
      message: 'You do not have access to this portfolio',
      code: 'FORBIDDEN',
    }),
  };
}

const category = await this.outcomeCategoryService.verifyCategory(
  portfolioId,
  categoryId
);
if (!category) {
  return {
    status: 404,
    body: createErrorObject({
      message: 'Category not found',
      code: 'NOT_FOUND',
    }),
  };
}

await this.outcomeCategoryService.remove(categoryId);

return {
  status: 204,
  body: createResponseObject(undefined),
};
}
);
}

```

```
}
```

```
apps/backend/src/app/outcome_categories/outcomeCategory.service.ts
```

```
import { Injectable } from '@nestjs/common';
```

```
import { InjectRepository } from '@nestjs/typeorm';
```

```
import { Repository } from 'typeorm';
```

```
import {
```

```
  ICreateOutcomeCategory,
```

```
  IUpdateOutcomeCategory,
```

```
} from '../interface/outcomeCategory.interface';
```

```
import { Category } from '../outcomeCategory.entity';
```

```
@Injectable()
```

```
export class OutcomeCategoryService {
```

```
  constructor(
```

```
    @InjectRepository(Category)
```

```
    private readonly outcomeCategoryRepository: Repository<Category>
```

```
) {}
```

```
  async findStandardCategories(): Promise<Category[]> {
```

```
    return await this.outcomeCategoryRepository.find({
```

```
      // where: { user: IsNull() },
```

```
    });
```

```
}
```

```
  async findAllInPortfolio(portfolioId: string): Promise<Category[]> {
```

```
    const customCategories = await this.outcomeCategoryRepository.find({
```

```
      where: {
```

```

    portfolio: {
      id: portfolioId,
    },
  },
  order: {
    createdAt: 'ASC',
  },
});
return customCategories;
}

```

```

async verifyCategory(
  portfolioId: string,
  categoryId: string
): Promise<Category | null> {
  const verifiedCategory = await this.outcomeCategoryRepository.findOne({
    where: { portfolio: { id: portfolioId }, id: categoryId },
  });
  return verifiedCategory;
}

```

```

async findOne(id: string, portfolioId: string): Promise<Category | null> {
  const outcomeCategory = await this.outcomeCategoryRepository.findOne({
    where: { id, portfolio: { id: portfolioId } },
  });
  return outcomeCategory;
}

```

```

async create(
  portoflioId: string,
  outcomeCategoryData: ICreateOutcomeCategory
): Promise<Category> {
  return await this.outcomeCategoryRepository.save({
    ...outcomeCategoryData,
    portfolio: {
      id: portoflioId,
    },
  });
}

async update(
  category: Category,
  updateOutcomeCategoryData: IUpdateOutcomeCategory
): Promise<Category> {
  Object.assign(category, updateOutcomeCategoryData);
  return await this.outcomeCategoryRepository.save(category);
}

async remove(id: string): Promise<boolean> {
  const result = await this.outcomeCategoryRepository.delete(id);

  return (result?.affected || 0) > 0;
}
}

```

apps/backend/src/app/outcome_categories/outcomeCategory.module.ts

```

import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { PortfolioService } from '../portfolios/portfolio.service';
import { OutcomeCategoryController } from './outcomeCategory.controller';
import { Category } from './outcomeCategory.entity';
import { OutcomeCategoryService } from './outcomeCategory.service';
import { Portfolio } from '../portfolios/portfolio.entity';
import { TransactionService } from '../transactions/transaction.service';
import { Transaction } from '../transactions/transactions.entity';

@Module({
  imports: [TypeOrmModule.forFeature([Category, Portfolio, Transaction])],
  controllers: [OutcomeCategoryController],
  providers: [OutcomeCategoryService, PortfolioService, TransactionService],
  exports: [OutcomeCategoryService],
})
export class OutcomeCategoryModule {}

```

apps/backend/src/app/outcome_categories/outcomeCategory.entity.ts

```

import { IOutcomeCategory } from '@andarna/common';
import {
  Column,
  CreateDateColumn,
  Entity,
  ManyToOne,
  OneToMany,
  PrimaryGeneratedColumn,

```

```

    UpdateDateColumn,
  } from 'typeorm';
import { Portfolio } from '../portfolios/portfolio.entity';
import { Transaction } from '../transactions/transactions.entity';

@Entity('categories')
export class Category implements IOutcomeCategory {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'varchar', length: 255 })
  name: string;

  @Column({ type: 'varchar', length: 7 }) // Assuming color is a hex code
  color: string;

  @ManyToOne(() => Portfolio, (portfolio) => portfolio.categories, {
    onDelete: 'CASCADE',
  })
  portfolio: Portfolio;

  @OneToMany(() => Transaction, (transaction) => transaction.category)
  transactions: Transaction[];

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()

```

```

    updatedAt: Date;
}

```

Модуль інтеграції з Monobank

apps/backend/src/app/saving_accounts/monobank.service.ts

```

import { mapCurrencyCodeToEnum } from '@andarna/common';
import { HttpService } from '@nestjs/axios';
import { HttpException, HttpStatus, Injectable } from '@nestjs/common';
import { AxiosError } from 'axios';
import { firstValueFrom } from 'rxjs';
import { SavingAccount } from '../savingAccount.entity';

```

```

interface MonobankJar {
  id: string;
  sendId: string;
  title: string;
  description?: string;
  currencyCode: number;
  balance: number; // In minimal units (e.g., kopecks)
  goal?: number; // In minimal units
}

```

```

interface MonobankClientInfoResponse {
  clientId: string;
  name: string;
  webHookUrl?: string;
  permissions: string;
}

```

```

accounts: any[]; // Define more strictly if needed
jars?: MonobankJar[];
}

@Injectable()
export class MonobankService {
  private readonly monobankApiUrl = 'https://api.monobank.ua';
  constructor(private readonly httpService: HttpService) {}

  async getUserJars(monobankToken: string): Promise<SavingAccount[]> {
    const url = `${this.monobankApiUrl}/personal/client-info`;
    const headers = {
      'X-Token': monobankToken,
    };

    try {
      const response = await firstValueFrom(
        this.httpService.get<MonobankClientInfoResponse>(url, { headers })
      );
      const clientInfo = response.data;

      if (!clientInfo || !clientInfo.jars) {
        return [];
      }

      const mappedSavingAccounts: SavingAccount[] = clientInfo.jars
        .map<SavingAccount | null>((jar) => {
          const currencyEnum = mapCurrencyCodeToEnum(jar.currencyCode);

```

```

if (!currencyEnum) {
    console.warn(
        `Unsupported currency code received: ${jar.currencyCode} for jar ${jar.id}`
    );
    return null;
}

```

```

const divisor = 100;

```

```

return {
    id: jar.id,
    name: jar.title,
    currency: currencyEnum,
    goalAmount: jar.goal ? jar.goal / divisor : null,
    currentAmount: jar.balance / divisor,
} as SavingAccount;
}))
.filter((account): account is SavingAccount => account !== null);

```

```

return mappedSavingAccounts;
} catch (error) {
    const axiosError = error as AxiosError;
    if (axiosError.response) {
        console.error(
            'Monobank API Error:',
            axiosError.response.status,
            axiosError.response.data
        );
    }
}

```

```

    throw new HttpException(
      `Failed to fetch data from Monobank: ${axiosError.response.status}`,
      axiosError.response.status || HttpStatus.INTERNAL_SERVER_ERROR
    );
  } else {
    console.error('Network or other error calling Monobank API:', error);
    throw new HttpException(
      'Could not connect to Monobank API',
      HttpStatus.SERVICE_UNAVAILABLE
    );
  }
}
}
}
}
}

```

Модуль інтеграції Stripe

apps/backend/src/app/stripe/stripe.module.ts

```

import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { UserModule } from '../user/user.module';
import { StripeController } from './stripe.controller';
import { StripeService } from './stripe.service';
import { Subscription } from './subscription.entity';
import { WebhookController } from './webhook.controller';

```

```

@Module({

```

```

        imports: [ConfigModule, UserModule,
TypeOrmModule.forFeature([Subscription])],
        controllers: [StripeController, WebhookController],
        providers: [StripeService],
        exports: [StripeService, TypeOrmModule],
    })
export class StripeModule {}

```

```

apps/backend/src/app/stripe/stripe.controller.ts
import { contract } from '@andarna/common';
import { Controller, UseGuards } from '@nestjs/common';
import { tsRestHandler, TsRestHandler } from '@ts-rest/nest';
import { AuthGuard } from '../auth/guards/auth.guard';
import { CurrentUser } from '../decorators/currentUser.decorator';
import { User } from '../user/user.entity';
import { createErrorObject, createResponseObject } from '../utils/response';
import { StripeService } from '../stripe.service';

```

```

@Controller()
export class StripeController {
    constructor(private stripeService: StripeService) {}

    @UseGuards(AuthGuard)
    @TsRestHandler(contract.billing.getPrices)
    async getPrices() {
        return tsRestHandler(contract.billing.getPrices, async () => {
            const prices = await this.stripeService.getPrices();

```

```

return {
  status: 200,
  body: createResponseObject(prices),
};
});
}

```

```

@UseGuards(AuthGuard)
@TsRestHandler(contract.billing.createCheckoutSession)
async createCheckoutSession(@CurrentUser() user: User) {
  return tsRestHandler(
    contract.billing.createCheckoutSession,
    async (req) => {
      const { priceId } = req.body;
      const subscription = await this.stripeService.getUserSubscription(
        user.id
      );
      if (subscription) {
        return {
          status: 400,
          body: createErrorObject({
            message: 'User already has an active subscription',
          }),
        };
      }
      const session = await this.stripeService.createCheckoutSession(
        user.email,

```

```

        priceId
    );

    return {
        status: 200,
        body: createResponseObject({ url: session.url as string }),
    };
}
);
}

```

```

@UseGuards(AuthGuard)
@TsRestHandler(contract.billing.getUserSubscription)
async getUserSubscription(@CurrentUser() user: User) {
    return tsRestHandler(contract.billing.getUserSubscription, async () => {
        const subscription = await this.stripeService.getUserSubscription(
            user.id
        );

        return {
            status: 200,
            body: createResponseObject(subscription),
        };
    });
}

```

```

@UseGuards(AuthGuard)
@TsRestHandler(contract.billing.createBillingPortalSession)

```

```

async createBillingPortalSession(@CurrentUser() user: User) {
  return tsRestHandler(
    contract.billing.createBillingPortalSession,
    async () => {
      const session = await this.stripeService.createBillingPortalSession(
        user.email
      );

      return {
        status: 200,
        body: createResponseObject({ url: session.url as string }),
      };
    }
  );
}
}

```

```

apps/backend/src/app/stripe/stripe.service.ts
import { PriceSchema } from '@andarna/common';
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { InjectRepository } from '@nestjs/typeorm';
import Stripe from 'stripe';
import { Repository } from 'typeorm';
import { Subscription } from './subscription.entity';

@Injectable()
export class StripeService {

```

```

private stripe: Stripe;
constructor(
    private configService: ConfigService,
    @InjectRepository(Subscription)
    private subscriptionRepository: Repository<Subscription>
) {
    this.stripe = new Stripe(
        this.configService.getOrThrow<string>('STRIPE_SECRET_KEY')
    );
}

async getCustomerByEmail(email: string): Promise<Stripe.Customer> {
    const customers = await this.stripe.customers.list({
        email,
        limit: 1,
    });

    if (customers.data.length > 0) {
        return customers.data[0];
    }

    const customer = await this.stripe.customers.create({ email });
    return customer;
}

async retrieveCustomer(customerId: string): Promise<Stripe.Customer> {
    const customer = await this.stripe.customers.retrieve(customerId);
    return customer as Stripe.Customer;
}

```

```

async retrieveProduct(productId: string): Promise<Stripe.Product> {
    const product = await this.stripe.products.retrieve(productId);
    return product as Stripe.Product;
}

async createCheckoutSession(
    userEmail: string,
    priceId: string
): Promise<Stripe.Checkout.Session> {
    const customer = await this.getCustomerByEmail(userEmail);
    const appUrl = this.configService.getOrThrow<string>('APP_URL');
    const session = await this.stripe.checkout.sessions.create({
        mode: 'subscription',
        payment_method_types: ['card'],
        customer: customer.id,
        line_items: [
            {
                price: priceId,
                quantity: 1,
            },
        ],
        success_url: `${appUrl}/subscription?paid=true`,
        cancel_url: `${appUrl}/subscription?canceled=true`,
    });

    return session;
}

```

```

private async retrievePrice(priceId: string): Promise<PriceSchema> {
    const price = await this.stripe.prices.retrieve(priceId, {
        expand: ['product'],
    });
    return {
        id: price.id,
        unit_amount: price.unit_amount,
        currency: price.currency,
        recurring: price.recurring
            ? { interval: price.recurring.interval }
            : { interval: 'unknown' },
        product: {
            id: (price.product as Stripe.Product).id,
            name: (price.product as Stripe.Product).name,
        },
    };
}

async getPrices(): Promise<{ monthly: PriceSchema; yearly: PriceSchema }> {
    const monthlyPriceId = this.configService.getOrThrow<string>(
        'STRIPE_MONTHLY_PRICE'
    );
    const yearlyPriceId = this.configService.getOrThrow<string>(
        'STRIPE_YEARLY_PRICE'
    );

    const [monthly, yearly] = await Promise.all([

```

```

        this.retrievePrice(monthlyPriceId),
        this.retrievePrice(yearlyPriceId),
    ]);

    return { monthly, yearly };
}

async createBillingPortalSession(
    email: string
): Promise<Stripe.BillingPortal.Session> {
    const appUrl = this.configService.getOrThrow<string>('APP_URL');
    const customer = await this.getCustomerByEmail(email);
    const session = await this.stripe.billingPortal.sessions.create({
        customer: customer.id,
        return_url: `${appUrl}/subscription`,
    });

    return session;
}

constructEvent(payload: any, signature: string): Stripe.Event {
    return this.stripe.webhooks.constructEvent(
        payload,
        signature,
        this.configService.getOrThrow('STRIPE_WEBHOOK_SECRET')
    );
}

```

```

async getUserSubscription(userId: string): Promise<Subscription | null> {
  const subscription = await this.subscriptionRepository.findOne({
    where: {
      user: {
        id: userId,
      },
      status: 'active',
    },
  });

  return subscription;
}

```

```

async cancelActiveSubscription(userId: string): Promise<boolean> {
  const subscription = await this.subscriptionRepository.findOne({
    where: {
      user: {
        id: userId,
      },
      status: 'active',
    },
  });

  if (!subscription) {
    return false; // No active subscription found
  }

  // Cancel the subscription in Stripe

```

```

    await this.stripe.subscriptions.cancel(subscription.id);

    // Update the subscription status in the database
    subscription.status = 'canceled';
    await this.subscriptionRepository.save(subscription);

    return true; // Subscription was canceled
  }
}

```

apps/backend/src/app/stripe/subscription.entity.ts

```

import {
  Column,
  CreateDateColumn,
  Entity,
  ManyToOne,
  PrimaryColumn,
  UpdateDateColumn,
} from 'typeorm';
import { User } from '../user/user.entity';

```

```

@Entity()
export class Subscription {
  @PrimaryColumn()
  id: string;

  @Column()
  productName: string;

```

@Column()

price: number;

@Column()

nextBillingDate: Date;

@Column({ nullable: true, type: 'timestamp' })

cancelDate: Date | null;

@Column()

status: string;

@Column()

billingInterval: string;

@CreateDateColumn()

createdAt: Date;

@UpdateDateColumn()

updatedAt: Date;

@ManyToOne(() => User, (user) => user.subscriptions, {

 nullable: false,

 onDelete: 'CASCADE',

})

user: User;

}

```
apps/backend/src/app/stripe/webhook.controller.ts
import { Controller, Post, RawBodyRequest, Req, Res } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Request, Response } from 'express';
import Stripe from 'stripe';
import { Repository } from 'typeorm';
import { UserService } from '../user/user.service';
import { StripeService } from '../stripe.service';
import { Subscription } from '../subscription.entity';
```

```
@Controller('webhook')
export class WebhookController {
  constructor(
    private stripeService: StripeService,
    private userService: UserService,
    @InjectRepository(Subscription)
    private subscriptionRepository: Repository<Subscription>
  ) {}
```

```
@Post('/stripe')
async handleStripeWebhook(
  @Req() req: RawBodyRequest<Request>,
  @Res() res: Response
) {
  const sig = req.headers['stripe-signature'] as string;
```

```

try {
  const event = this.stripeService.constructEvent(req.rawBody!, sig);
  await this.handleEvent(event);
  res.sendStatus(200);
} catch (err: any) {
  console.error(err);
  res.status(400).send(`Webhook Error: ${err.message}`);
}
}

```

```

private async handleEvent(event: Stripe.Event) {
  switch (event.type) {
    case 'customer.subscription.created':
    case 'customer.subscription.updated':
    case 'customer.subscription.deleted':
      await this.handleSubscriptionEvent(event);
      break;
    // Handle other event types as needed
  }
}

```

```

private async handleSubscriptionEvent(
  event:
    | Stripe.CustomerSubscriptionCreatedEvent
    | Stripe.CustomerSubscriptionDeletedEvent
    | Stripe.CustomerSubscriptionUpdatedEvent
) {
  const subscription = event.data.object;

```

```

const customerId = subscription.customer as string;
const customer = await this.stripeService.retrieveCustomer(customerId);
console.log(customer);
const user = await this.userService.findByEmail(customer.email as string);
console.log(user);

const product = await this.stripeService.retrieveProduct(
    subscription.items.data[0].price.product as string
);

if (!user) {
    console.error(`User not found for Stripe customer ID: ${customerId}`);
    return;
}

const subscriptionData = {
    id: subscription.id,
    user: {
        id: user.id,
    },
    productName: product.name,
    price: subscription.items.data[0].price.unit_amount as number,
    nextBillingDate: new Date(
        subscription.items.data[0].current_period_end * 1000
    ),
    cancelDate: subscription.cancel_at
        ? new Date(subscription.cancel_at * 1000)
        : null,

```

```

    status: subscription.status,
    billingInterval: subscription.items.data[0].price.recurring
      ?.interval as string,
  };

  if (event.type === 'customer.subscription.created') {
    const sub = this.subscriptionRepository.create(subscriptionData);
    await this.subscriptionRepository.save(sub);
    await this.userService.updateUser(user.id, {
      recommendationsAllowed: 5,
      lastCreditsReset: new Date(),
      recommendationsUsed: 0,
    });
  } else if (event.type === 'customer.subscription.updated') {
    await this.subscriptionRepository.update(
      { id: subscription.id },
      subscriptionData
    );
    await this.userService.updateUser(user.id, {
      recommendationsAllowed: 5,
      lastCreditsReset: new Date(),
      recommendationsUsed: 0,
    });
  } else if (event.type === 'customer.subscription.deleted') {
    await this.subscriptionRepository.update(
      { id: subscription.id },
      subscriptionData
    );
  }

```

```

    await this.userService.updateUser(user.id, {
      recommendationsAllowed: 1,
      lastCreditsReset: new Date(),
      recommendationsUsed: 0,
    });
  }
}
}

```

apps/backend/src/app/openai/openai.module.ts

```

import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { TypeOrmModule } from '@nestjs/typeorm';
import { PortfolioModule } from '../portfolios/portfolio.module';
import { SavingAccountModule } from '../saving_accounts/savingAccount.module';
import { UserService } from '../user/user.service';
import { OpenAiService } from './openai.service';
import { RecommendationsController } from './recommendations.controller';
import { Recommendation } from './recommendations.entity';
import { User } from '../user/user.entity';
import { StripeModule } from '../stripe/stripe.module';

```

```

@Module({
  imports: [
    ConfigModule,
    SavingAccountModule,
    PortfolioModule,
    StripeModule,

```

```

    TypeOrmModule.forFeature([Recommendation, User]),
  ],
  providers: [OpenAiService, UserService],
  controllers: [RecommendationsController],
})
export class OpenAiModule {}

```

apps/backend/src/app/openai/recommendations.entity.ts

```

import {
  Column,
  CreateDateColumn,
  Entity,
  ManyToOne,
  PrimaryGeneratedColumn,
} from 'typeorm';
import { Portfolio } from '../portfolios/portfolio.entity';

```

```

@Entity('recommendations')
export class Recommendation {
  @PrimaryGeneratedColumn('uuid')
  id: string;

```

```

  @Column({ type: 'text' })
  result: string;

```

```

  @CreateDateColumn()
  createdAt: Date;

```

```

    @ManyToOne(() => Portfolio, (portfolio) => portfolio.invitations, {
      onDelete: 'CASCADE',
    })
    portfolio: Portfolio;
  }

```

apps/backend/src/app/openai/recommendations.controller.ts

```

import { contract } from '@andarna/common';
import { Controller } from '@nestjs/common';
import { tsRestHandler, TsRestHandler } from '@ts-rest/nest';
import dayjs from 'dayjs';
import { CurrentUser } from '../decorators/currentUser.decorator';
import { PortfolioService } from '../portfolios/portfolio.service';
import { SavingsService } from '../saving_accounts/savings.service';
import { TransactionService } from '../transactions/transaction.service';
import { User } from '../user/user.entity';
import { UserService } from '../user/user.service';
import { createErrorObject, createResponseObject } from '../utils/response';
import { OpenAiService } from './openai.service';

```

```

@Controller()

```

```

export class RecommendationsController {
  constructor(
    private readonly openAiService: OpenAiService,
    private readonly transactionService: TransactionService,
    private readonly portfolioService: PortfolioService,
    private readonly savingAccountService: SavingsService,
    private readonly userService: UserService
  ) {}

```

```
) {}
```

```
@TsRestHandler(contract.recommendations.generateFinancialRecommendations)
async generateFinancialRecommendations(@CurrentUser() user: User) {
  return tsRestHandler(
    contract.recommendations.generateFinancialRecommendations,
    async ({ params: { portfolioId } }) => {
      try {
        await this.portfolioService.verifyPortfolioAccess(
          user.id,
          portfolioId
        );
      } catch (e) {
        return {
          status: 403,
          body: createErrorObject({
            message: 'You do not have access to this portfolio',
          }),
        };
      }
      if (user.recommendationsUsed >= user.recommendationsAllowed) {
        return {
          status: 403,
          body: createErrorObject({
            message: 'You have reached the maximum number of recommendations',
          }),
        };
      }
    }
  );
}
```

```

const startOfMonth = dayjs().startOf('month').toDate();
const endOfMonth = dayjs().endOf('month').toDate();

const transactions = await this.transactionService.findAllByPortfolioId(
  portfolioId,
  { take: 1000, skip: 0 },
  'transactionDate',
  'desc',
  {
    dateFrom: startOfMonth.toISOString(),
    dateTo: endOfMonth.toISOString(),
  }
);
const savings = await this.savingAccountService.getSavingsByPortfolioId(
  portfolioId
);

const prompt = this.transactionService.constructAIPrompt(
  transactions,
  savings
);

const message = await this.openAiService.askFinanceAssistant(prompt);

const rec = await this.openAiService.saveRecommendation(
  portfolioId,
  message
);

```

```

    await this.userService.updateUser(user.id, {
      recommendationsUsed: user.recommendationsUsed + 1,
    });

    return {
      status: 200,
      body: createResponseObject(rec),
    };
  }
);
}

@TsRestHandler(contract.recommendations.getAllRecommendations)
async getAllRecommendations(@CurrentUser() user: User) {
  return tsRestHandler(
    contract.recommendations.getAllRecommendations,
    async ({ params: { portfolioId }, query }) => {
      try {
        await this.portfolioService.verifyPortfolioAccess(
          user.id,
          portfolioId
        );
      } catch (e) {
        return {
          status: 403,
          body: createErrorObject({
            message: 'You do not have access to this portfolio',
          }),
        };
      }
    }
  );
}

```

```

    };
  }
  const recommendations =
    await this.openAiService.getAllRecommendationsByPortfolioId(
      portfolioId,
      query
    );
  return {
    status: 200,
    body: createResponseObject(recommendations),
  };
}
);
}
}

```

```

apps/backend/src/app/openai/openai.service.ts
import { HttpService } from '@nestjs/axios';
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { InjectRepository } from '@nestjs/typeorm';
import OpenAI from 'openai';
import { Repository } from 'typeorm';
import { Recommendation } from './recommendations.entity';

@Injectable()
export class OpenAiService {

```

```

private readonly apiKey: string;
private readonly openAi: OpenAI;
constructor(
  private readonly httpService: HttpService,
  private readonly configService: ConfigService,
  @InjectRepository(Recommendation)
  private readonly recommendationRepository: Repository<Recommendation>
) {
  this.apiKey = this.configService.getOrThrow<string>('OPENAI_API_KEY');
  this.openAi = new OpenAI({
    apiKey: this.apiKey,
  });
}

```

```

async askFinanceAssistant(prompt: string): Promise<string> {
  try {
    const thread = await this.openAi.beta.threads.create();
    await this.openAi.beta.threads.messages.create(thread.id, {
      role: 'user',
      content: prompt,
    });

    const run = await this.openAi.beta.threads.runs.createAndPoll(thread.id, {
      assistant_id: 'asst_bXplCUzwQpm0mr7M0qmkKBzv',
    });

    if (run.status === 'completed') {
      const messages = await this.openAi.beta.threads.messages.list(
        run.thread_id

```

```

    );
    for (const message of messages.data.reverse()) {
        if (message.role === 'assistant')
            return (message.content[0] as any).text.value as string;
        }
    }
    return 'No response from assistant';
} catch (error: any) {
    console.error('Error communicating with OpenAI:', error.message);
    throw new Error('Failed to communicate with OpenAI API');
}
}

```

```

async saveRecommendation(
    portfolioId: string,
    result: string
): Promise<Recommendation> {
    const recommendation = this.recommendationRepository.create({
        portfolio: { id: portfolioId },
        result,
    });
    return this.recommendationRepository.save(recommendation);
}

```

```

async getAllRecommendationsByPortfolioId(
    portfolioId: string,
    options: { take?: number; skip?: number } = {}
): Promise<Recommendation[]> {

```

```

return this.recommendationRepository.find({
  where: { portfolio: { id: portfolioId } },
  order: { createdAt: 'DESC' },
  take: options.take,
  skip: options.skip,
});
}
}

```

constructAIPrompt

```

constructAIPrompt(transactions: Transaction[], savings: SavingAccount[]) {
  let prompt = "";

  const outcomeTransactions = transactions.filter(
    (transaction) => transaction.type === TransactionType.OUTCOME
  );
  const incomeTransactions = transactions.filter(
    (transaction) => transaction.type === TransactionType.INCOME
  );
  const savingsTransactions = transactions.filter(
    (transaction) => transaction.type === TransactionType.SAVINGS
  );
  const totalIncome = incomeTransactions.reduce(
    (acc, transaction) => acc + transaction.amount,
    0
  );
}

```

```

);
const totalOutcome = outcomeTransactions.reduce(
  (acc, transaction) => acc + transaction.amount,
  0
);
const totalSavings = savingsTransactions.reduce(
  (acc, transaction) => acc + transaction.amount,
  0
);
const categorizedOutcome = outcomeTransactions.reduce(
  (acc, transaction) => {
    const categoryName = transaction.category?.name || 'Uncategorized';
    if (!acc[categoryName]) {
      acc[categoryName] = 0;
    }
    acc[categoryName] += transaction.amount;
    return acc;
  },
  {} as Record<string, number>
);

prompt += 'Outcome: \n';
Object.entries(categorizedOutcome).forEach(([category, amount]) => {
  prompt += ` - ${category}: ${amount.toFixed(2)}\n`;
});
prompt += `Total Outcome: ${totalOutcome.toFixed(2)}\n`;

prompt += `Total Income: ${totalIncome.toFixed(2)}\n`;

```

```

prompt += `Total Savings: ${totalSavings.toFixed(2)}\n`;
prompt += 'Savings: ';
savings.forEach((saving) => {
  prompt += ` - ${saving.name}: Goal = ${
    saving.goalAmount
    ? `${saving.goalAmount.toFixed(2)}${saving.currency}`
    : 'N/A'
  }; Current=${saving.currentAmount}${saving.currency}`;
});

return prompt;
}

```

Модуль запрошення користувачів

```

apps/backend/src/app/mailer/mailer.module.ts

import { Global, Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { MailerService } from './mailer.service';

@Global()
@Module({
  imports: [ConfigModule],
  providers: [MailerService],
  exports: [MailerService],
})
export class MailerModule {}

```

```

apps/backend/src/app/mailer/mailer.service.ts

```

```
import * as Brevo from '@getbrevo/brevo'; // this way you get all types, but it will
work as with const SibApiV3Sdk = require('@getbrevo/brevo'); too.
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import * as fs from 'fs'; // to read custom email templates
```

```
@Injectable()
```

```
export class MailerService {
```

```
  constructor(private config: ConfigService) {}
```

```
  // TODO: rewrite this method to prevent plagiat ?
```

```
  async sendMail(data: {
```

```
    mailData: {
```

```
      receivers: Brevo.SendSmtEmail['to'];
```

```
      subject: Brevo.SendSmtEmail['subject'];
```

```
    };
```

```
    variables?: Record<string, string>;
```

```
    template: string;
```

```
    attachments?: Brevo.SendSmtEmailAttachmentInner[];
```

```
  }): Promise<boolean> {
```

```
    try {
```

```
      const apiKey = this.config.get('BREVO_API_KEY');
```

```
      const apiInstance = new Brevo.TransactionalEmailsApi();
```

```
      apiInstance.setApiKey(0, apiKey);
```

```
      const { mailData, template, attachments, variables = {} } = data;
```

```
      const { receivers, subject } = mailData;
```

```

const emailTemplateSource = fs.readFileSync(
  `apps/backend/src/app/mailer/templates/${template}.html`,
  'utf8'
);

const htmlContent = emailTemplateSource.replace(
  /\{\{(\w+)\}\}\}/g,
  (_, key) => {
    return variables[key] || "";
  }
);

// Add this to ENV
await apiInstance.sendTransacEmail({
  sender: { email: 'ipz211_shov@student.ztu.edu.ua' },
  to: receivers,
  subject: subject,
  htmlContent,
  attachment:
    attachments && attachments.length > 0 ? attachments : undefined,
});

return true;
} catch (error) {
  console.log('ERROR SENDING EMAIL: ', error);
  return false;
}
}

```

}

TEMPLATES

INVITATION.html

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<title>Вас запрошено!</title>
```

```
<style>
```

```
body {
```

```
    font-family: Arial, sans-serif;
```

```
    line-height: 1.6;
```

```
}
```

```
.container {
```

```
    max-width: 600px;
```

```
    margin: auto;
```

```
    padding: 20px;
```

```
    border: 1px solid #ddd;
```

```
    border-radius: 8px;
```

```
    background-color: #f9f9f9;
```

```
}
```

```
h2 {
```

```
    color: #333;
```

```
}
```

```
a {
```

```
    color: #3498db;
```

```
    text-decoration: none;
```

```

    }
    .footer {
        margin-top: 20px;
        font-size: 0.9em;
        color: #666;
    }
</style>
</head>
<body>
    <div class="container">
        <h2>Привіт!</h2>
        <p>
            Вас запросив <strong>{{name}}</strong> (<a href="mailto:{{email}}"
                >{{email}}</a>
            >) приєднатися до нашої платформи.
        </p>
        <p>Щоб прийняти запрошення, будь ласка, натисніть на посилання
нижче:</p>
        <p>
            <a href="{{invitationLink}}" target="_blank">Прийняти запрошення</a>
        </p>
        <p>Це запрошення дійсне протягом 24 годин.</p>
        <p>Якщо у вас є запитання, будь ласка, зверніться до відправника.</p>
        <div class="footer">
            <p>З найкращими побажаннями,<br />Команда</p>
        </div>
    </div>
</body>

```

</html>

```
apps/backend/src/app/invitations/invitations.service.ts
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { InjectRepository } from '@nestjs/typeorm';
import dayjs from 'dayjs';
import { Repository } from 'typeorm';
import { MailerService } from '../mailer/mailer.service';
import { PortfolioService } from '../portfolios/portfolio.service';
import { User } from '../user/user.entity';
import { Invitation } from './invitation.entity';
```

@Injectable()

```
export class InvitationsService {
  constructor(
    @InjectRepository(Invitation)
    private invitationsRepository: Repository<Invitation>,
    private mailerService: MailerService,
    private configService: ConfigService,
    private portfolioService: PortfolioService
  ) {}
```

```
  async createInvitation(email: string, portfolioId: string, sender: User) {
    return await this.invitationsRepository.save({
      email,
      expiresAt: dayjs().add(1, 'day').toDate(),
```

```

    portfolio: { id: portfolioId },
    sender: { id: sender.id },
  });
}

```

```

async deleteInvitation(id: string) {
  const invitation = await this.invitationsRepository.findOne({
    where: { id },
    relations: {
      portfolio: true,
      sender: true,
    },
  });
  if (!invitation) {
    throw new Error('Invitation not found');
  }
  return this.invitationsRepository.remove(invitation);
}

```

```

async inviteUserByEmail(
  email: string,
  portfolioId: string,
  sender: User,
  invitationId: string
) {
  const appUrl = this.configService.get('APP_URL');

  await this.mailerService.sendMail({

```

```

mailData: {
  receivers: [{ email }],
  subject: 'Andarna: You have been invited to a portfolio',
},
variables: {
  name: `${sender.firstName} ${sender.lastName}`,
  email: sender.email,
  invitationLink: `${appUrl}/invitation?portfolioKey=${invitationId}`,
},
template: 'INVITATION',
});

return (await this.getInvitationByEmailAndPortfolio(
  email,
  portfolioId
)) as Invitation;
}

async getInvitationByEmailAndPortfolio(email: string, portfolioId: string) {
  return this.invitationsRepository.findOne({
    where: { email, portfolio: { id: portfolioId } },
    relations: {
      sender: true,
    },
  });
}

async getInvitationsByPortfolioId(portfolioId: string) {

```

```

return this.invitationsRepository.find( {
  where: { portfolio: { id: portfolioId } },
  relations: {
    sender: true,
  },
});
}

```

```

async getInvitationById(id: string) {
  return this.invitationsRepository.findOne( {
    where: { id },
    relations: {
      sender: true,
      portfolio: {
        members: true,
      },
    },
  });
}

```

```

async acceptInvitation(invitation: Invitation, user: User) {
  const portfolio = invitation.portfolio;

  await this.portfolioService.addMemberToPortfolio(portfolio, user);

  await this.invitationsRepository.delete(invitation.id);

  return true;
}

```

```
}
}
```

Модуль курсу валют

apps/backend/src/app/exchangeNbuApi/exchangeNbu.module.ts

```
import { Module } from '@nestjs/common';
import { ExchangeNbuService } from './exchangeNbu.service';
import { ExchangeNbuController } from './exchangeNbu.controller';
```

```
@Module({
  imports: [],
  controllers: [ExchangeNbuController],
  providers: [ExchangeNbuService],
})
export class ExchangeNbuModule {}
```

apps/backend/src/app/exchangeNbuApi/exchangeNbu.controller.ts

```
import { contract, Currency } from '@andarna/common';
import { Cache, CACHE_MANAGER } from '@nestjs/cache-manager';
import { Controller, Inject, NotFoundException } from '@nestjs/common';
import { tsRestHandler, TsRestHandler } from '@ts-rest/nest';
import { ExchangeNbuService } from './exchangeNbu.service';
```

```
@Controller()
export class ExchangeNbuController {
  constructor(
    private exchangeNbuService: ExchangeNbuService,
```

```
@Inject(CACHE_MANAGER) private cacheManager: Cache
) {}
```

```
@TsRestHandler(contract.nbuExchange.getExchangeRate)
async getExchangeRate() {
  return tsRestHandler(
    contract.nbuExchange.getExchangeRate,
    async ({ params: { currency } }) => {
      const existingCurrencyRate = await this.cacheManager.get<number>(
        `exchange-rate/${currency}`
      );
      if (existingCurrencyRate) {
        return {
          status: 200,
          body: {
            rate: existingCurrencyRate,
          },
        };
      }
      const exhchangeRate = await this.exchangeNbuService.getExchangeRate(
        currency as Currency
      );
      if (!exhchangeRate) {
        throw new NotFoundException();
      }
      await this.cacheManager.set(
        `exchange-rate/${currency}`,
        exhchangeRate.rate,
      );
    }
  );
}
```

```

        { ttl: 300 } as any
    );
    return {
        status: 200,
        body: {
            rate: exhchangeRate.rate,
        },
    };
}
);
}
}

```

```

apps/backend/src/app/exchangeNbuApi/exchangeNbu.service.ts
import { Currency } from '@andarna/common';
import { HttpService } from '@nestjs/axios';
import { Injectable } from '@nestjs/common';
import { IExchangeRate } from '../interface/exchangeRate.interface';
import { firstValueFrom } from 'rxjs';

```

```

@Injectable()
export class ExchangeNbuService {
    constructor(private readonly httpService: HttpService) {}
    async getExchangeRate(currency: Currency) {
        const { data } = await firstValueFrom(
            this.httpService.get<IExchangeRate[]>(
                'https://bank.gov.ua/NBUStatService/v1/statdirectory/exchange?json'
            )
        );
    }
}

```

```

    )
  );
  const finalRate = data.find((item) => item.cc === currency);
  return finalRate;
}
}

```