ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка веб-сервісу "SpendWise" моніторингу особистих фінансів з автоматичним аналізом витрат »

на здобуття освітнього ступеня бакалавра зі спеціальності 121 Інженерія програмного забезпечення освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Віталій БУРАЧИК

(підпис)

Виконав: здобувач вищої освіти групи ПД-41

Віталій БУРАЧИК

Керівник: Наталія ТРІНТІНА

канд.техн.наук, доц.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
_____ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
Бурачику Віталію Олександровичу

1. Тема кваліфікаційної роботи: «Розробка Веб-сервісу “SpendWise” моніторингу особистих фінансів з автоматичним аналізом витрат»
   керівник кваліфікаційної роботи канд.техн.наук, доц. Наталія ТРІНТІНА, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи персонального фінансового обліку та бюджетування, принципи категоризації витрат і побудови фінансової аналітики, документація до фреймворків Express.js та React, специфікація роботи з СУБД PostgreSQL, опис побудови клієнт-серверної архітектури на основі REST API, методи автентифікації користувачів за допомогою JWT, формати обміну даними (CSV, JSON), приклади реалізації веб-застосунків для управління особистими фінансами..

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
    1. Аналіз предметної галузі управління особистими фінансами та визначення вимог до програмного забезпечення.
    2. Огляд та аналіз засобів реалізації веб-застосунку для обліку та аналізу персональних витрат.
    3. Проєктування архітектури веб-застосунку для управління особистими фінансами.
    4. Реалізація та тестування функціональності застосунку.

5. Перелік графічного матеріалу: *презентація*
    1. Аналіз аналогів.
    2. Вимоги до програмного забезпечення.
    3. Програмні засоби реалізації.
    4. Діаграма компонентів.
    5. ER-діаграма структури бази даних системи.
    6. Діаграма діяльності.
    7. Діаграма класів.
    8. Екранні форми.
    9. Демонстрація роботи застосунку.
    10. Апробація результатів дослідження.

6. Дата видачі завдання «23» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	25.02.-04.03.2026	
2	Аналіз та дослідження існуючих аналогів	05.03-13.03.2026	
3	Аналіз предметної галузі навчання професійних оцінювачів кави та визначення вимог до програмного забезпечення	14.03-19.03.2026	
4	Огляд та аналіз засобів реалізації застосунку для підтримки навчання професійних оцінювачів кави	20.03-28.03.2026	
5	Проєктування архітектури застосунку для підготовки оцінювачів кави	29.03-12.04.2026	
6	Реалізація та тестування функціональності застосунку	14.04-28.04.2026	

7	Оформлення роботи: вступ, висновки, реферат	29.04-11.05.2026	
8	Розробка демонстраційних матеріалів	12.05-18.05.2026	
9	Попередній захист роботи	19.05-22.05.2026	
10	Подання кваліфікаційної роботи	23.05-30.05.2026	

Здобувач вищої освіти _____ Віталій БУРАЧИК

(підпис)

Керівник
кваліфікаційної роботи _____ Наталія ТРІНТІНА

(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 54 стор., 1 табл., 6 рис., 20джерел.

Мета роботи – підвищення інтерпритованості особистих фінансівза рахунок розробки вебсервісу для обробки даних, візуалізації та автоматичного аналізу витрат.

Об'єкт дослідження – процеси представлення особистих фінансових даних.

Предмет дослідження – програмні засоби розпрбки вебсервісів для представлення особистих фінансових даних.

Короткий зміст роботи: В роботі проаналізовано предметну галузь персонального фінансового менеджменту та методики бюджетування. Проведено огляд існуючих програмних рішень у сфері обліку витрат та визначено вимоги до функціональності майбутнього застосунку. Реалізовано веб-застосунок, який включає модулі для обліку витрат, бюджетного планування, імпорту банківських транзакцій у форматі CSV та аналітики. Застосунок забезпечує реєстрацію користувачів, особистий кабінет, додавання та редагування витрат за категоріями, встановлення місячних бюджетних лімітів, отримання бюджетних сповіщень та автоматичний аналіз фінансових показників. Для реалізації використано Express.js та Node.js на бекенді, React на фронтенді, PostgreSQL для збереження даних. Автентифікацію реалізовано на основі JWT. Проведено тестування функціоналу.

Сферою використання застосунку є організація самостійного контролю особистих фінансів для фізичних осіб: студентів, фрилансерів та домогосподарств.

КЛЮЧОВІ СЛОВА: BACK-END, FRONT-END, JAVASCRIPT, NODE.JS, EXPRESS.JS, RESTful API, JWT, POSTGRESQL, ОБЛІК ВИТРАТ, БЮДЖЕТУВАННЯ, ВЕБЗАСТОСУНОК

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 10
ВСТУП 11
1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ 13
1.1 Методи персонального фінансового обліку та бюджетування 13
1.2 Аналіз існуючих програмних рішень 14
1.2.1 MoneyManager 14
1.2.2 Toshl Finance 15
1.2.3 Spendee 15
1.3 Визначення вимог до застосунку 18
1.4 Інформаційна модель предметної галузі 19
2 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ 21
2.1 Середовище розробки 21
2.1.1 Visual Studio Code 21
2.2 Мови програмування 22
2.2.1 JavaScript 22
2.3 Веб-фреймворки 23
2.3.1 Node.js та Express.js 24
2.3.2 React 24
2.3.3 Recharts 25
2.4 Система управління базами даних 26
2.4.1 PostgreSQL 27
2.5 Система контролю версій 28
2.5.1 Git 28
2.5.2 GitHub 28
2.6 Інструменти дизайну інтерфейсу 29
2.6.1 Figma 29
2.6.2 Visual Studio Code 30
3 ПРОЄКТУВАННЯ СИСТЕМИ 31
3.1 Проєктування архітектури застосунку 31
3.1.1 Діаграма діяльності користувача 32

3.1.2 Діаграма варіантів використання 35
3.2 Архітектура клієнтської частини 35
3.3 Архітектура серверної частини 37
3.4 Архітектура бази даних 40
4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ 43
4.1 Реалізація серверної частини 43
4.1.1 Загальна архітектура серверного застосунку 43
4.1.2 Структура проєкту та організація модулів 44
4.1.3 Конфігурація середовища 44
4.1.4 Підключення до бази даних 45
4.1.5 Ініціалізація Express-застосунку та middleware 46
4.1.6 Інтернаціоналізація серверних відповідей 47
4.1.7 Архітектура токенів 48
4.1.8 Безпечне зберігання refresh-токенів 48
4.1.9 Middleware авторизації 49
4.1.10 Реєстрація нового користувача 51
4.2 Реалізація клієнтської частини 52
4.3 Інтеграція та взаємодія компонентів 57
4.4 Тестування застосунку 59
ВИСНОВКИ 62
ПЕРЕЛІК ПОСИЛАНЬ 64
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 66
ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ 74

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ПЗ – Програмне забезпечення

IDE – Integrated Development Environment

API – Application Programming Interface

REST – Representational State Transfer

SPA – Single Page Application

HTML – HyperText Markup Language

CSS – Cascading Style Sheets

JS – JavaScript

JWT – JSON Web Token

JSON – JavaScript Object Notation

SQL – Structured Query Language

ORM – Object-Relational Mapping

ВСТУП

Актуальність: управління особистими фінансами є невід'ємною складовою фінансової грамотності сучасних людей, оскільки прийняття грамотних фінансових рішень вимагає відстеження витрат та планування бюджету. У контексті поширення безготівкових розрахунків та складної структури витрат виникає потреба в ефективних інструментах, які б дозволяли відстежувати та аналізувати власні витрати. Використання веб-додатків у цьому випадку дає змогу організувати облік у зручний спосіб, що передбачає візуалізацію зібраних даних, автоматизовані розрахунки та своєчасні повідомлення щодо бюджету.

Розроблений додаток SpendWise створено як універсальний інструмент, що допомагає відстежувати особисті фінанси. Він містить низку функцій, зокрема відстеження витрат за категоріями, інструменти планування бюджету, завантаження транзакцій через CSV-файли та автоматизований аналіз фінансових показників. Використання цифрових інструментів дозволяє відстежувати динаміку витрат, встановлювати особисті ліміти витрат та отримувати своєчасні сповіщення про наближення до ліміту. Представлена програма пропонує персоналізований підхід до управління витратами та може використовуватися широким колом користувачів завдяки гнучкому представленню інформації та підтримці двомовного інтерфейсу.

Об'єктом дослідження є процеси представлення особистих фінансових даних.

Предметом дослідження є програмні засоби розпрбки вебсервісів для представлення особистих фінансових даних.

Метою роботи є підвищення інтерпритованості особистих фінансівза рахунок розробки вебсервісу для обробки даних, візуалізації та автоматичного аналізу витрат.

Методи дослідження: Спочатку потрібно було проаналізувати практики управління особистими фінансами, а також різні рішення, призначені для

відстеження особистих витрат. На основі цієї інформації було сформульовано функціональні та нефункціональні вимоги, яким має відповідати додаток. Потім було розглянуто можливі технологічні варіанти для досягнення цілей і дій висновку, що для реалізації бек-енду можна використовувати фреймворк Express.js та Node.js, а для розробки фронт-енду — React. В якості системи управління базами даних було обрано PostgreSQL. Для аутентифікації я використав технологію JWT, що дозволяє поновлювати токени. Все було об’єднано за допомогою RESTful API. Проектування програмного забезпечення здійснювалося за допомогою UML-діаграм, після чого функціонал було поетапно реалізовано, протестовано, а інтерфейс розроблено відповідно до сучасних стандартів.

Наукова новизна роботи полягає у створенні вебсервісу, який поєднує управління витратами за категоріями з бюджетуванням, враховує коефіцієнт інфляції, має вбудовану систему сповіщень і автоматизовану оцінку фінансової статистики користувача. Однією з особливостей є створення модуля, що дозволяє імпортувати транзакції з файлів CSV з урахуванням шаблонів для найпопулярніших банків України, таких як «Монобанк» і «ПриватБанк», а також для міжнародного сервісу Wise. Вона автоматично оцінює фінанси та надає рекомендації користувачеві на основі аналізу його витрат, що підвищує ефективність самостійного управління фінансами.

Практична значущість результатів полягає у можливості онлайн-управління особистими фінансами за допомогою створеного веб-додатку, що дозволяє контролювати витрати та створювати бюджети, аналізувати особисті фінанси в будь-який час з різних сучасних пристроїв.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Методи персонального фінансового обліку та бюджетування

Особистий фінансовий облік, це звичка вести облік усіх надходжень і витрат грошей, не дивлячись на те, що ведеться у спеціальному зошиті чи банківській програмі. Ця звичка дозволяє людині зрозуміти за що вона платить, які суми грошей йдуть на те, що їй не потрібно і що вона може зробити, щоб розпорядитись своїми грошима більш раціонально. Облік витрат буває ручним (гроші записуються у зошиті чи таблиці), напівавтоматичним (переведення у таблицю інформації зі банківської картки) і автоматичним (автоматичне отримання інформації від банку).

Ведення обліку витрат передбачає розподіл їх на певні категорії. Серед найпопулярніших категорій є: продукти, транспорт, житло, розваги, здоров'я необхідні витрати. Вірний розподіл на категорії дає змогу зрозуміти яку частку в загальному обсязі витрат займає той чи інший напрямок.

Бюджетування, процес планування своїх витрат на певний період (зазвичай місяць) з визначенням верхніх меж по кожній категорії. Серед найпоширеніших виділяють:

- метод 50/30/20, витрати 50% доходу на необхідні потреби, 30% на особисті бажання і 20% на заощадження;
- метод конвертів, виділення певної суми на кожен вид витрат наперед;
- бюджет з нуля (Zero-Based Budgeting), що місяць розподіляєш до нуля наступним чином: по категоріях до статей витрат;
- метод відстеження фактичних витрат, реалізуєш свій бюджет без бюджету.

Фінансова аналітика, логічне продовження обліку і бюджетування. Тут будуються графіки по динаміці витрат за місяцями, кругові діаграми за розподілом по категоріях, а також надаються поради на основі твоїх спостережень. Аналітика дає можливість користувачу побачити на власні очі свою фінансову поведінку і вчасно відреагувати на її відхилення від визначеного бюджету.

Сучасні цифрові засоби автоматизують велику частину описаних процесів, що значно полегшує задачу користувачів, які не мають спеціальних фінансових знань. Саме тому розробка зручного веб-застосунку, який би впроваджував облік, категоризацію, бюджетування та аналітику в єдиному інтерфейсі є актуальною та практично значущою.

1.2 Аналіз існуючих програмних рішень

Сучасні цифрові фінансові технології пропонують досить широкий спектр застосунків, серед яких особливе місце займають сервіси для персонального фінансового менеджменту. Такі сайти дозволяють клієнтам вести облік витрат, планувати горизонти бюджету, візуалізувати і аналізувати фінансові дані, що у підсумку дозволяє краще контролювати витрати, дотримуватися лімітів заощаджень та приймати більш зважені фінансові рішення.

В рамках даної дослідницької роботи ми вирішили звернути увагу на три, мабуть, найбільш популярних сервіси у цій галузі: MoneyManager, Toshl та Spendee. Розглянемо їх трохи детальніше.

1.2.1 MoneyManager

MoneyManager — це мобільний застосунок для обліку особистих фінансів, розроблений компанією Realbyte Inc. Застосунок орієнтований на детальний ручний облік доходів і витрат із широкими можливостями категоризації.

Основні функціональні можливості застосунку:

- ведення обліку доходів і витрат із розподілом за категоріями та підкатегоріями;
- підтримка кількох рахунків (готівка, картки, депозити);
- формування звітів у вигляді кругових і стовпчастих діаграм;
- налаштування повторюваних транзакцій;
- експорт даних у форматі Excel та CSV;
- захист додатку PIN-кодом або біометрією.

Серед переваг MoneyManager варто відзначити детальну систему категорій, зручний журнал транзакцій та підтримку кількох валют. Водночас застосунок має суттєві обмеження: відсутність повноцінної веб-версії, обмежені можливості бюджетування та відсутність автоматичного імпорту банківських виписок. Інтерфейс застосунку орієнтований переважно на досвідчених користувачів і може здатися перевантаженим для новачків.

1.2.2 Toshl Finance

Toshl Finance - це хмарний застосунок для управління особистими фінансами, доступний на платформах iOS, Android та у веб-браузері. Розроблений компанією Toshl Inc., він вирізняється яскравим дизайном та геймифікованим підходом до обліку витрат. Основні функціональні можливості застосунку:

- облік витрат і доходів із підтримкою тегів і нотаток;
- встановлення місячних бюджетів по категоріях із відстеженням виконання;
- автоматична синхронізація з банківськими рахунками (у преміум-версії);
- підтримка понад 160 валют із автоматичною конвертацією;
- розгорнута аналітика у вигляді графіків і звітів;
- експорт даних у форматах CSV, PDF та JSON.

Toshl Finance має розвинену аналітичну складову та зручний інтерфейс, однак більшість ключових функцій (синхронізація з банком, розширені звіти) доступні лише за платною підпискою. Безкоштовна версія суттєво обмежена за функціональністю, що знижує її практичну цінність для широкого кола користувачів.

1.2.3 Spendee

Spendee є одним із найновіших веб-додатків і мобільних застосунків для спільного та індивідуального керування фінансами. Вони розташовані в Чехії. Цей застосунок відрізняється легкістю використання та красивим інтерфейсом.

Основні функції:

- можна додавати транзакції вручну, також є можливість імпорту автоматичних транзакцій із банківських рахунків;
- підтримка спільних гаманців для сімей, груп і так далі;
- маєте можливість встановити бюджет та переглядати виконання;
- консультації ваших витрат у вигляді діаграм і графіків розвинути в часі;
- відмітка витрат на карті;
- підтримка різних мов і валют.

Перш за все, головним стимулом для людей обирати програму Spendee є її зручність у користуванні та наявність можливості спільного використання гаманців. Подібно до Toshl, цей додаток має багато платних послуг (таких як підключення до банку та необмежена кількість гаманців), яких не пропонують інші сервіси. Також у ній відсутня можливість імпортувати дані з банківських виписок через CSV-файли, тому українцям, які працюють з банками, що не підтримують підключення, неможливо користуватися цим програмним забезпеченням.

Отже, порівнявши кілька варіантів, можна помітити, що у них є багато спільного, зокрема обмежена кількість доступних безкоштовних послуг, неможливість імпортувати CSV-дані в українських банках тощо. Це означає, що слід розробити нову веб-закладку для SpendWise з урахуванням цього переліку недоліків, особливо для вітчизняних клієнтів.

Кожна програма має щось своє, починаючи від способів подання інструкцій, методів оцінки та рівня інтерактивності загалом. Є програми, спрямовані на особисте вивчення та теоретичне підґрунтя для цього, а є такі, що пропонують практику, наприклад, дегустацію кави. Особливу увагу слід приділити інструментам оцінки та аналізу їхніх результатів.

Щоб полегшити порівняння між платформами та визначити їх сильні та слабкі сторони, нижче наведено зведену таблицю аналізу ключових характеристик кожного з досліджених інструментів. Таблиця враховує тип матеріалів, рівень

інтерактивності, формат навчання, механізми оцінювання, а також наявність функціоналу аналізу дегустацій та особливості доступу до ресурсів.

Таблиця 1.1

Зведена таблиця аналізу існуючих застосунків для обліку особистих фінансів та їх порівняння

Критерій	**MoneyManager**	**Toshl Finance**	**Spendee**	**SpendWise**
Веб-версія	-	+	+	+
Мобільна версія	+	+	+	+
Безкоштовний доступ	+	+-	+-	+
Облік витрат за категоріями	+	+	+	+
Бюджетування по категоріях	+-	+	+	+
Бюджетні сповіщення	-	+	+	+
Імпорт CSV-виписок	+	+	-	+
Підтримка банків UA (Mono, Privat)	-	-	-	+
Фінансова аналітика / графіки	+	+	+	+
Автоматичний аналіз витрат	-	-	-	+
Двомовний інтерфейс (UA/EN)	-	+	+	+
Відкрита архітектура (REST API)	-	-	-	+

Продовження таблиці 1.1

Зведена таблиця аналізу існуючих застосунків для обліку особистих фінансів та їх порівняння

Реєстрація/ особистий кабінет	+	+	+	+
Відкат імпорту	-	-	-	+

За результатами порівняльного тестування сервіс SpendWise виглядає конкурентніше всіх інших сервісів у частині підтримки пресетів українських банків для імпорту операцій у CSV-форматі, автоматизованого аналізу витрат, також через повністю безкоштовний доступ до всього функціоналу сервісу.

1.3 Визначення вимог до застосунку

Вимоги були розроблені на основі аналізу галузі та наявних програмних рішень.

Функціональні вимоги визначають функції, якими повинна володіти система. По-перше, реєстрація та автентифікація користувачів повинні здійснюватися за допомогою JSON Web Tokens або технології федеративної автентифікації (FR-01), що забезпечить безпечну ізоляцію доступу до особистих музейних експозицій або розділів веб-сайту. Повинна бути передбачена функціональність, що дозволяє поповнювати витрати за категоріями та автоматично обмежувати поточні витрати (FR-02, FR-03).

До додатку також включена функціональність, що дозволяє обробляти дані, а саме фільтрувати транзакції та здійснювати пошук за певний період часу (FR-04), а також отримувати певні дані за обраний період (FR-05). Аналітичний аспект забезпечується подвоєнням зростання діаграми витрат, реалізованим за допомогою Recharts (FR-06). Ще однією важливою функцією, яку слід включити в додаток, є редагування та видалення існуючих даних про особисті фінанси (FR-07).

Нефункціональні вимоги описують властивості системи. Вимоги щодо продуктивності передбачають час відгуку менше 300 мс та час завантаження сторінки не більше 1 секунди. Аспекти безпеки включають впровадження JSON Web Tokens, шифрування паролів із використанням алгоритму хешування bcrypt та передачу даних через HTTPS, а також захист від SQL-ін’єкцій.

Кросплатформність досягається адаптивним інтерфейсом на основі responsive CSS, що гарантує звичайне використання застосунку на ПК, планшеті чи смартфоні. Надійність системи забезпечується валідацією даних як на фронтенді, так і на бекенді, автоматичним резервним копіюванням бази даних, а також коректною роботою застосунку у випадку обриву з'єднання (graceful degradation). З позиції масштабованості застосунок розроблений за модульною архітектурою, що дає змогу додавати нові модулі без потреби у переписуванні існуючого коду.

Підтримуваність забезпечується задокументованими описом REST API та коментарями у коді. До доступності‹ застосунок може виконуватися на пристроях з ОС Android v10 і вище, iOS 14 і вище, а також коректно працює у браузерах Google Chrome, Mozilla Firefox, Microsoft Edge.

1.4 Інформаційна модель предметної галузі

Інформаційна модель предметної галузі, це уявлення про ключові сутності та їх взаємозв’язки у системі управління особистими фінансами. Модель формалізує логіку взаємодії користувачів, фінансових операцій, категорій, бюджетів та інших компонентів. Це дозволяє ефективно спроектувати як структуру бази даних, так і бізнес-логіку програмного забезпечення.

У центрі інформаційної моделі стоїть користувач (User), який взаємодіє із системою через створення фінансових записів, налаштування бюджетів і перегляд аналітичних звітів. Користувач має базову інформацію (ім'я, email, пароль) і пов'язаний з усіма об'єктами системи через унікальний ідентифікатор. Для автентифікації використовуються токени (Token), які зберігають хешоване значення refresh-токена та дозволяють безпечно підтримувати активну сесію.

Кожна витрата (Expense) є основною одиницею обліку і містить назву, суму, дату, примітку і належить до певної категорії (Category). Категорії визначають тематичний розподіл витрат (продукти, транспорт, житло, розваги, здоров'я, інше) і дають змогу будувати аналітику у різних розрізах. Тобто, фінансові дані користувача отримують змістовну структуру.

Бюджет (Budget) дозволяє користувачу встановлювати місячний ліміт витрат для кожної категорії. Порівнюючи фактичні витрати з лімітом, система формує бюджетні сповіщення (BudgetAlert), які мають три стани: в нормі (ok), наближаємось до ліміту (near) та перевищено ліміт (over). Тобто, користувача оперативно інформують про стан його фінансів.

Система імпорту побудована на сутності імпортної сесії (ImportSession), яка фіксує дату імпорту, вибраний пресет банку, кількість імпортованих та пропущених записів. Кожна імпортована витрата зберігає посилання на відповідну сесію, що дозволяє реалізувати відкат (rollback) останнього імпорту у випадку помилки.

Аналіз базується на агрегованих даних: місячні витрати (MonthlyStat) та розподіл коштів по категоріям (CategoryBreakdown). З урахуванням цих даних генерується автоматичний аналіз (Analysis), який містить текстовий інсайт і список корисних рекомендацій для економії коштів.

Кожна частина моделі тісно пов'язана з іншою. Наприклад, запис про витрати не може існувати без користувача і категорії, бюджет, без категорії, а імпортований запис про витрати, без відповідної сесії імпорту. Таким чином, забезпечується логічна цілісність моделі і виключається можливість «сирітських» записів у базі даних.

2 ЗАСОБИ РЕАЛІЗАЦІЇ ЗАСТОСУНКУ

2.1 Середовище розробки

Інтегроване середовище розробки (IDE) - це комплексне програмне забезпечення, яке надає розробникам повний набір інструментів для написання, тестування та налагодження коду. IDE зазвичай включає текстовий редактор, інструменти для автоматичної збірки, дебагер та засоби інтеграції з системами контролю версій. У межах розробки веб-застосунку “SpendWise” було використано одне середовище – Visual Studio Code.

2.1.1 Visual Studio Code

Visual Studio Code (VS Code) - це багатоплатформовий редактор коду з відкритим кодом, розроблений корпорацією Microsoft. Незважаючи на те, що VS Code не є повноцінною IDE (інтегрованим середовищем розробки), він має безліч розширень, які дозволяють йому працювати як справжня IDE з можливістю створення веб-додатків з використанням таких технологій, як JavaScript, HTML та CSS.

Під час роботи над веб-додатком для наших клієнтів з компанії SpendWise фронтенд-розробники використовували VS Code для створення React-компонентів, застосування стилів до користувацького інтерфейсу, налагодження викликів API та управління станом додатка. Серед розширень, що використовувалися для цієї мети, - ESLint для перевірки коду та дотримання високих стандартів, Prettier для форматування коду, GitLens для відстеження версій та історії комітів і, нарешті, вбудований термінал для ще більшої ефективності роботи.

Що стосується інтеграції, вбудована функція git дозволила відстежувати коміти, зміни та керувати гілками безпосередньо в IDE. Такі функції, як автозавершення та підказки щодо коду, а також вбудований відладчик,

пришвидшили робочий процес. У результаті Visual Studio Code став основним редактором для фронтенд-розробки в нашій компанії, і ми змогли дуже швидко створити привабливий та повнофункціональний користувацький інтерфейс.

2.2 Мови програмування

Мови програмування – це формальні системи, призначені для створення інструкцій, які можуть інтерпретуватися та виконуватися обчислювальними машинами. Вони дають змогу фахівцям розробляти алгоритми, керувати апаратним забезпеченням, обробляти дані та створювати різноманітні типи інтерфейсів і програмних додатків. Сучасні мови програмування підтримують різні моделі програмування, зокрема об’єктно-орієнтовану, функціональну та процедурну, що дозволяє розробляти як прості скрипти, так і складні розподілені системи.

Для створення цього додатка було використано дві основні мови програмування. Перша - це JavaScript, яку було обрано як єдину мову, необхідну для створення повнофункціонального додатка: на стороні сервера в контексті Node.js разом із фреймворком Express.js, а на стороні клієнта - за допомогою React. Цей вибір відіграв вирішальну роль у уніфікації технологічного стеку, оптимізації передачі даних між рівнями додатка та спрощенні переходу від фронтенду до бекенду. Друга мова - це SQL, яка використовується для створення схеми реляційної бази даних PostgreSQL, написання скриптів міграції та запиту даних із серверної частини додатка.

2.2.1 JavaScript

JavaScript – це мова програмування високого рівня, динамічна та інтерпретована. Як мова, орієнтована на веб-розробку, JavaScript набула популярності завдяки своїй здатності підтримувати об’єктно-орієнтоване та функціональне програмування. Поява Node.js дозволила використовувати JavaScript не лише в браузері, а й у серверних додатках, як програми командного рядка або навіть як мікросервіси.

У цьому проєкті JavaScript став мовою програмування, яку я використовував для реалізації всіх рівнів повнофункціонального додатка SpendWise. На серверній стороні я застосував JavaScript для створення REST API, обробки HTTP-запитів, застосування логіки, взаємодії з базою даних PostgreSQL за допомогою драйвера pg та аутентифікації за допомогою JSON Web Tokens у рамках фреймворків Node.js та Express.js.

На стороні клієнта JavaScript разом із бібліотекою React допоміг створити адаптивний інтерфейс користувача завдяки управлінню станами компонентів, обробці подій, надсиланню HTTP-запитів через API та візуалізації даних. Я також візуалізував різні діаграми та графіки за допомогою бібліотеки Recharts.

Використання виключно JavaScript як на стороні клієнта, так і на стороні сервера допомогло мені створити більш оптимізовану архітектуру проєкту, уніфікувати обмін даними у форматі JSON між цими двома рівнями та пришвидшити кодування всіх частин системи.

2.3 Веб-фреймворки

Веб-фреймворки - це програми, які допомагають розробникам створювати веб-застосунки. Вони надають набір готових компонентів, інструментів і архітектурних шаблонів, що дозволяє розробникам швидко реалізовувати функціонал для взаємодії з базами даних, обробки запитів, маршрутування, автентифікації користувачів тощо.

У цьому проєкті використовувалися два основні інструменти веб-розробки: Express.js для реалізації серверної частини та React для побудови клієнтського інтерфейсу. Також застосовувалася бібліотека Recharts для візуалізації фінансових даних у вигляді графіків і діаграм. Такий розподіл дозволив реалізувати чітку архітектуру «клієнт-сервер», що базується на принципах REST, та створити гнучкий, масштабований і сучасний веб-застосунок для управління особистими фінансами.

2.3.1 Node.js та Express.js

Node.js - це JavaScript-середовище виконання, яке базується на рушії V8 від Google і дозволяє запускати JavaScript-код поза межами браузера. Це середовище асинхронне та орієнтоване на події, що забезпечує високу продуктивність при роботі з великою кількістю одночасних з'єднань через неблокуючі операції введення-виведення. Завдяки цьому Node.js є чудовим вибором для створення серверів API та застосунків, що працюють у реальному часі.

Express.js - це мінімалістичний і гнучкий веб-фреймворк для Node.js, який пропонує набір інструментів для створення серверних застосунків та REST API. Він не задає строгих правил чи структуру для проєкту, тому розробники можуть на свій розсуд організовувати архітектуру власних застосунків.

У проєкті SpendWise Node.js використовувався як середовище виконання серверної частини, а Express.js є основним фреймворком для реалізації серверної частини. На базі Express.js було створено маршрути REST API, реалізовано обробку HTTP-запитів (GET, POST, PUT, DELETE), підключено проміжні шари для обробки JSON-даних, налаштовано CORS для взаємодії з фронтендом, а також реалізовано автентифікацію за допомогою JWT із механізмом оновлення токенів.

Завдяки мінімалістичному дизайну Express.js та продуктивності Node.js серверна частина застосунку швидко обробляє запити і легко масштабується просто шляхом додавання нових маршрутів і модулів без необхідності змінювати існуючу кодову базу.

2.3.2 React

React - це відкрита JavaScript-бібліотека для створення користувацьких інтерфейсів, розроблена компанією Meta (Facebook). Вона базується на концепції компонентів, згідно з якою інтерфейс складається з незалежних, повторно використовуваних компонентів, кожен з яких керує власним станом і відповідає за відображення певної частини сторінки. React використовує віртуальний DOM для ефективного оновлення інтерфейсу, замість повного перерендерингу сторінки

оновлюються тільки ті елементи, які фактично змінилися, що забезпечує високу продуктивність застосунку.

В проєкті SpendWise React використовувався для створення всієї клієнтської частини застосунку. За допомогою хуків useState та useEffect був реалізований механізм управління станом компонентів та завантаження даних з API при зміні фільтрів або оновленні сторінки. Хук useMemo застосовувався для оптимізації обчислень похідних значень, зокрема для форматування загальної суми витрат.

Інтерфейс застосунку складається з компонентів форми додавання та редагування витрат, панелі фільтрації, модуля імпорту CSV, розділів бюджетування та аналітики. Усі компоненти динамічно реагують на зміну стану та відображають актуальні дані без необхідності перезавантаження сторінки. Підтримка двомовного інтерфейсу (українська та англійська) реалізована за допомогою системи перекладів безпосередньо на рівні React-компонентів.

Завдяки декларативному підходу React та зручній моделі управління станом розробка інтерактивного та адаптивного інтерфейсу SpendWise була швидкою та структурованою.

2.3.3 Recharts

Recharts - це відкрита бібліотека для побудови графіків і діаграм у React-застосунках, розроблена на основі бібліотеки D3.js та SVG-рендерингу. Вона надає готовий набір декларативних компонентів для візуалізації даних, які органічно інтегруються в компонентну архітектуру React. Recharts вирізняється простотою використання, гнучкістю налаштування зовнішнього вигляду та адаптивністю, графіки автоматично масштабуються під розмір контейнера завдяки компоненту ResponsiveContainer.

У межах проєкту SpendWise бібліотека Recharts використовувалася для реалізації аналітичної складової застосунку. Зокрема, було побудовано два основні типи діаграм:

- стовпчаста діаграма (BarChart), відображає динаміку витрат користувача по місяцях. Для покращення сприйняття даних застосано округлені верхні кути стовпців, відключено вертикальні лінії сітки та налаштовано стилізований tooltip з інформацією про суму витрат за обраний місяць;
- кругова діаграма (PieChart), відображає розподіл витрат за категоріями у вигляді кільцевої діаграми з внутрішнім радіусом (donut-стиль). Кожна категорія виділена окремим кольором із палітри, а підписи та легенда забезпечують зручне читання даних.

Обидві діаграми динамічно оновлюються при зміні фільтрів (місяць, категорія) та відображають актуальні дані, отримані з REST API. Завдяки Recharts аналітична частина застосунку наочна та інформативна, що суттєво підвищує цінність SpendWise як інструменту фінансового планування.

2.4 Система управління базами даних

Система управління базами даних (СУБД) — це програмне забезпечення, що забезпечує створення, організацію, збереження, модифікацію та отримання даних у структурованому вигляді. СУБД є невід'ємною складовою будь-якого сучасного застосунку, оскільки дозволяє ефективно працювати з великими обсягами інформації, підтримувати зв'язки між сутностями та гарантувати цілісність і безпеку даних.

У рамках розробки застосунку SpendWise було обрано об'єктно-реляційну СУБД PostgreSQL, яка є однією з найпотужніших і найнадійніших систем з відкритим вихідним кодом. PostgreSQL підтримує повний стандарт SQL, забезпечує транзакційну цілісність даних відповідно до принципів ACID, а також надає розвинені механізми індексування та виконання складних запитів.

У проєкті PostgreSQL використовується для збереження всіх ключових сутностей системи: облікових записів користувачів, витрат, бюджетів, сесій імпорту та refresh-токенів. Взаємодія серверної частини з базою даних реалізована через

драйвер pg для Node.js із використанням параметризованих SQL-запитів, що унеможливлює SQL-ін'єкції та забезпечує безпечну обробку користувацьких даних. Структура бази даних визначена у вигляді SQL-схеми з явним описом таблиць, типів даних, зовнішніх ключів та обмежень, що гарантує логічну цілісність збережених даних та коректність зв'язків між сутностями.

2.4.1 PostgreSQL

PostgreSQL — це потужна об'єктно-реляційна система управління базами даних з відкритим вихідним кодом, розробка якої ведеться з 1986 року в рамках проєкту POSTGRES Каліфорнійського університету в Берклі. На сьогодні PostgreSQL є одним із найпопулярніших рішень у сфері реляційних баз даних завдяки своїй надійності, розширюваності та повній відповідності стандарту SQL.

Серед ключових переваг PostgreSQL варто виділити підтримку транзакцій відповідно до принципів ACID (атомарність, узгодженість, ізольованість, довговічність), що гарантує цілісність даних навіть у разі збоїв. Система підтримує складні типи даних, зокрема JSON, масиви та користувацькі типи, а також забезпечує розвинені механізми індексування, що суттєво підвищує продуктивність запитів при роботі з великими обсягами інформації.

У проєкті SpendWise PostgreSQL виконує роль основного сховища даних. У базі визначено такі таблиці: users для зберігання облікових записів користувачів, expenses для фінансових записів із прив'язкою до категорії та дати, budgets для місячних лімітів по категоріях, refresh_tokens для управління сесіями автентифікації та import_sessions для журналювання операцій імпорту CSV-виписок.

Взаємодія з PostgreSQL реалізована на стороні сервера через драйвер pg для Node.js із використанням параметризованих запитів, що забезпечує захист від SQL-ін'єкцій. Структура бази даних описана у вигляді SQL-схеми (schema.sql), що спрощує розгортання застосунку в нових середовищах та підтримку актуального стану бази даних протягом усього циклу розробки.

2.5 Система контролю версій

Система керування версіями (СКВ) - програмний засіб, що дає можливість відслідковувати зміни у вихідному коді проєкту з плином часу, зберігати історію модифікацій і, у разі потреби, повертатися до минулих станів. СКВ є невід'ємною частиною сучасного процесу розробки програмного забезпечення, оскільки гарантує безпечну роботу з кодовою базою, полегшує співпрацю у команді та зменшує ризик втрати або пошкодження коду.

У процесі розробки застосунку SpendWise використовувалися два взаємодоповнюючі інструменти: Git як розподілена система керування версіями та GitHub як хмарна платформа для розміщення репозиторію.

2.5.1 Git

Git - розподілена система керування версіями, створена Лінусом Торвальдсом у 2005 році. На відміну від централізованих систем, Git зберігає повну копію репозиторію локально на кожному пристрої розробника, що дозволяє працювати автономно без постійного підключення до мережі. Ключовими перевагами Git є висока швидкість виконання операцій, гнучка система гілок (branching) та надійний механізм злиття змін (merging).

У проєкті SpendWise Git застосовувався для фіксації змін у коді на всіх етапах розробки, від початкового налаштування проєкту до реалізації окремих функціональних модулів. Використання гілок дало змогу ізолювати розробку нових функцій від стабільної версії застосунку та інтегрувати зміни контрольованим способом після їх перевірки.

2.5.2 GitHub

GitHub - хмарна платформа для розміщення Git-репозиторіїв, що надає веб-інтерфейс для перегляду історії змін, управління гілками та організації командної роботи над проєктом. У межах розробки SpendWise GitHub слугував віддаленим

сховищем вихідного коду, забезпечуючи резервне копіювання репозиторію та зручний доступ до актуальної версії проєкту з будь-якого пристрою.застосунком.

2.6 Інструменти дизайну інтерфейсу

При розробці сучасного веб-застосунку одним з ключових процесів є не лише вибір технологій для створення, а й засобів, для ефективного написання коду та проєктування користувацького інтерфейсу. Сучасні нструменти розробки дозволяють скоротити час на рутинні операції, підвищити якість коду та забезпечити узгодженість візуального стилю засупунку на всіх етапах створення застосунку.

У процесі створення застосунку SpendWise двома основними інструментами були Visual Studio Code та Figma відповідно для розробки та проєктування застосунку.

2.6.1 Figma

Figma - це хмарний інструмент для проєктування та прототипування користувацьких інтерфейсів, який працює безпосередньо у браузері та не потребує встановлення додаткового програмного забезпечення. Figma дозволяє створювати макети сторінок, визначати кольорову палітру, типографіку та налаштовувати комопненти інтерфейсу ще до початку його створення у програмі. У проєкті SpendWise, Figma стала в нагоді для розробки wireframe-макетів основних сторінок застосунку, лендингу, панелі авторизації та головного дашборду, що дозволило заздалегідь врегулювати структуру інтерфейсу та узгодити його візуальний стиль перед початком розробки коду.послідовність.

Використання Figma дозволило суттєво скоротити час на узгодження дизайну та забезпечити ефективну комунікацію між розробниками, дизайнерами та іншими учасниками проєкту.

2.6.2 Visual Studio Code

Visual Studio Code (VS Code) - це редактор коду з відкритим вихідним кодом, розроблений Microsoft, який відомий своєю розгалуженою екосистемою розширень, що дозволяють налаштувати редактор під повноцінне середовище розробки для JavaScript, HTML, CSS та інших веб-технологій. Деякі розширення, такі як ESLint для контролю якості коду, Prettier для автоматичного форматування, та GitLens для роботи з історією змін, використовувалися при розробці SpendWise. Вбудована інтеграція з Git, інтелектуальне автодоповнення, та вбудований термінал також уникнули необхідності у використанні зовнішніх інструментів, дозволивши працювати ефективно в одному інтерфейсі.

Однією з важливих переваг VS Code є можливість налаштування під конкретний проєкт. Завдяки файлу .vscode/settings.json нам вдалося встановити єдині правила форматування та редагування для всіх членів команди, що гарантувало узгодженість коду незалежно від налаштувань кожного учасника. У нашій конфігурації, що поєднувала Prettier та ESLint, будь-які стилістичні помилки виправлялися одразу після збереження файлу.

Незважаючи на наявність понад 50 000 розширень, під час розробки SpendWise ми підійшли до ринку розширень VS Code вибірково: було обрано лише мінімум необхідних інструментів. Такий підхід допоміг нам зосередитися на функціях, що сприяли прискоренню розробки.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Проєктування архітектури застосунку

Архітектура веб-застосунку SpendWise базується на трьохшаровій моделі «клієнт, сервер, база даних», це типова схема для сучасних веб-застосунків. Розподіл відповідальності по рівнях дозволяє підвищити тестованість та підтримуваність коду і створює передумови для масштабування окремих шарів незалежно один від одного.

Рівень клієнта (Browser) написано як односторінковий веб-застосунок (SPA) за допомогою React та Vite. Клієнт відповідає за відображення інтерфейсу, керування станом компонентів, відправлення HTTP-запитів до API та динамічний рендеринг сторінок. Для побудови графіків використовується бібліотека Recharts, що дозволяє створювати стовпчасті і кругові діаграми. Застосунок підтримує двомовність, українською та англійською мовами, храни переклади вбудовані у застосунок. Запити до сервера відбуваються через захищені HTTPS-запити з передаванням JWT-токена у заголовку Authorization: Bearer.

Рівень сервера (Node.js) написано на Express.js і він реалізує всю бізнес-логіку системи. Сервер обробляє HTTP-запити, виконує їх валідацію, проводить автентифікацію через middleware authRequired і формує відповіді у форматі JSON. В якості механізму аутентифікації використовується JWT з коротким життям access-токена (15 хвилин) та довготривалим refresh-токеном (30 днів), що дозволяє досягти балансу між безпекою і зручністю. Паролі зберігаються у хешованому виді за допомогою bcrypt. Розроблено окремий модуль для парсингу CSV із пресетами для Monobank, PrivatBank, Wise та generic.

Рівень бази даних (PostgreSQL) відповідається за збереження всіх даних системи. В базі даних є таблиці users, expenses, budgets, refresh_tokens, import_batches і imported_expenses. Сервер взаємодіє з базою даних через драйвер

pg з використанням параметризованих SQL-запитів. Щоб забезпечити цілісність транзакцій при засвоєнні та відкаті імпортів, використовується код BEGIN / COMMIT / ROLLBACK. Для прискорення виконання типових запитів фільтрації та агрегації даних, індекси задані на полях user_id і date.

Взаємодія між рівнями відбувається за принципами REST: клієнт надсилає запити до серверного API, сервер опрацьовує їх і звертається до бази даних, після чого повертає структуровану відповідь у форматі JSON. Така архітектура унеможливлює пряму взаємодію клієнта з базою даних, що суттєво підвищує рівень безпеки системи.

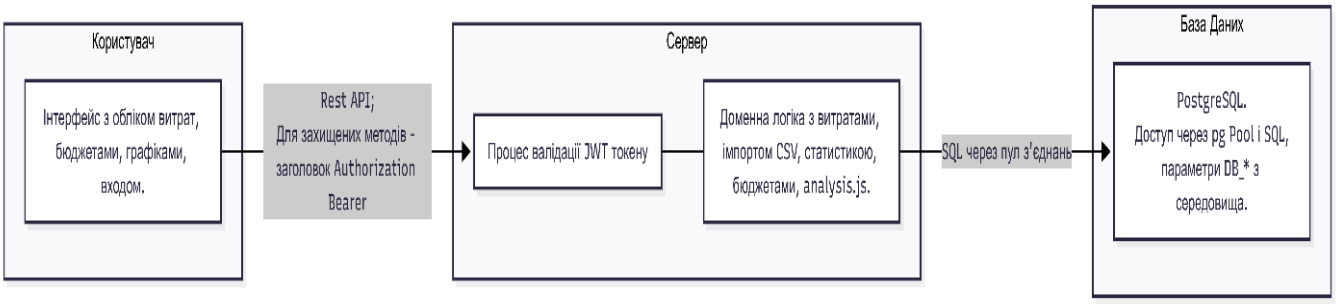


Рис. 3.1 Схема трирівневої архітектури застосунку SpendWise із зазначенням технологій та протоколів взаємодії між компонентами.

3.1.1 Діаграма діяльності користувача

Дизайн архітектури системи не обходить стороною і проектування взаємодії користувача із додатком. Для цього була створена відповідна діаграма діяльності, що ілюструє послідовні кроки користувача в перебігу роботи веб-застосунку SpendWise. Серед інших сценаріїв, дана діаграма демонструє типовий випадок використання системи:

Першим кроком є перевірка інформації про сесію користувача. Якщо користувач не авторизований, йому пропонується ввійти в обліковий запис або зареєструватися. Після успішної авторизації користувач потрапляє на головну панель керування (дашборд) де може спостерігати актуальний стан витрат та бюджетів. Користувач має можливість додати нову витрату вручну або імпортувати

транзакції через CSV-файл. У випадку імпорту система генерує прев’ю даних, і після підтвердження збереження витрати фіксуються у базі даних. Після збереження даних система автоматично оновлює статистику та перевіряє стан бюджетів, генеруючи відповідні сповіщення. Користувач може переглядати аналітику у вигляді графіків і діаграм або отримати автоматичний аналіз витрат із корисними порадами.

Завдяки даній діаграмі, користувач може легше уявити процес взаємодії із системою, а також логіку переходів між різними функціональними блоками застосунку (див. рис. 3.2).

Важливо зазначити, що алгоритм представлено як для основного сценарію робочого процесу, так і для альтернативних сценаріїв, пов’язаних із помилками при введенні даних, неправильним форматом CSV або перевищенням встановленого бюджетного ліміту. Це дає змогу перевірити, як система працює у випадках таких винятків, і гарантує ефективну роботу додатка з точки зору користувацького досвіду.

Процедуру управління бюджетом слід особливо виділити, оскільки вона включає порівняння витраченої суми грошей та ліміту бюджету для певних категорій після того, як кожна транзакція була зареєстрована та додана до таблиці статистики. У разі ризику перевищення встановленого ліміту бюджету система надсилає користувачеві відповідні повідомлення.

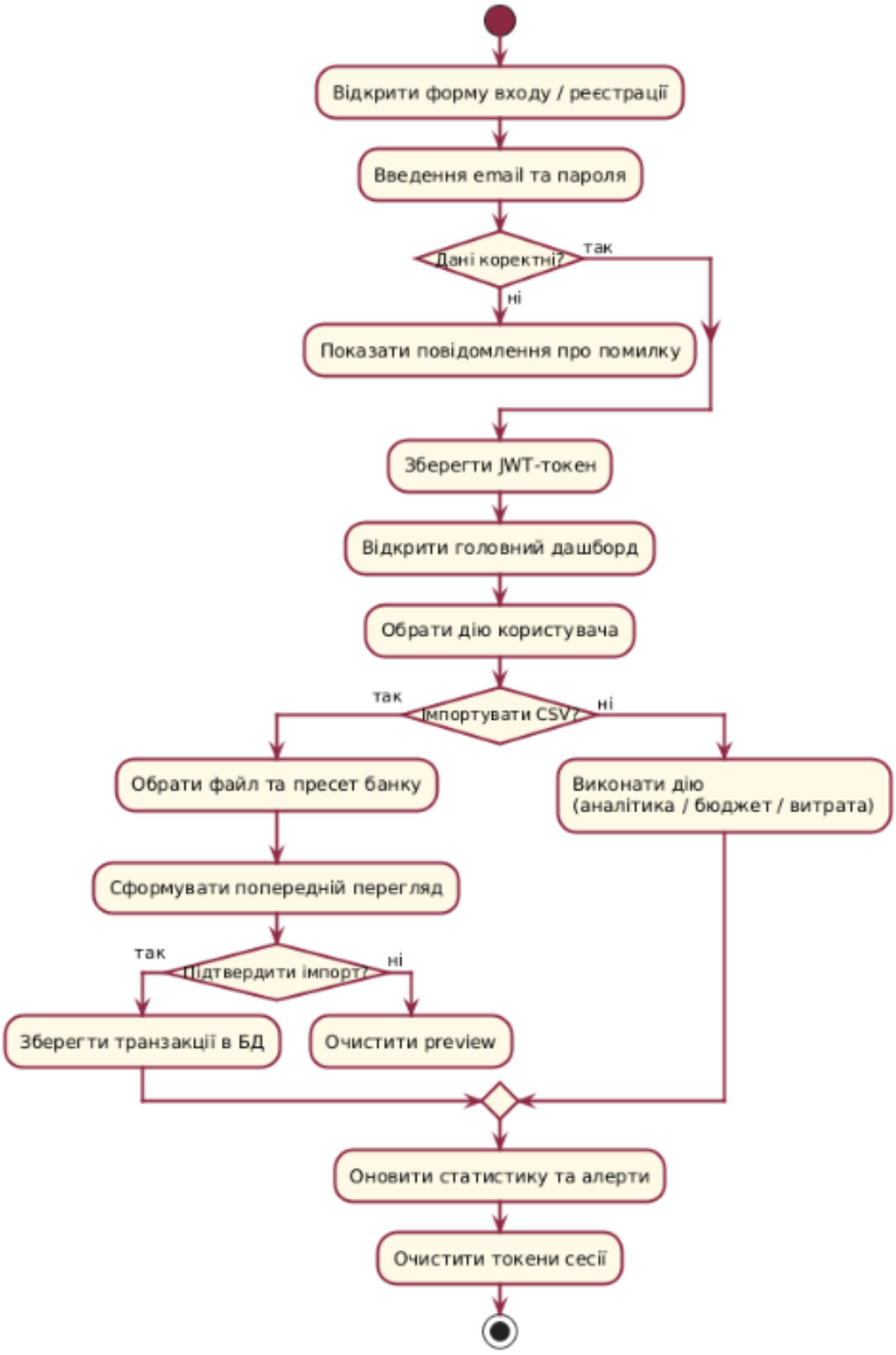


Рис. 3.2 Діаграма діяльності користувача під час проходження курсу

3.1.2 Діаграма варіантів використання

Use Case Diagram є важливою діаграмою в моделюванні вимог до програмного забезпечення в нотації UML. Вона показує, які функції системи доступні користувачам, визначає межі системи, ілюструючи, які дії можуть виконувати актори системи та як вони взаємодіють із цими функціями.

У системі SpendWise визначено двох основних акторів: Гість (неавторизований користувач) та Користувач (авторизований обліковий запис). Гість має обмежений доступ до системи, він може переглянути лендинг сторінку, ознайомитися з інформацією про застосунок, зареєструватися або увійти в існуючий обліковий запис. Після входу користувач отримує повний доступ до функціоналу застосунку.

Авторизований користувач може виконувати наступні дії: керувати витратами (додавати, редагувати, видаляти та фільтрувати записи), імпортувати банківські транзакції через CSV-файл із вибором пресету, переглядати фінансову статистику у вигляді графіків і діаграм, отримувати автоматичний аналіз витрат із корисними рекомендаціями, керувати місячними бюджетними лімітами по категоріях, переглядати бюджетні сповіщення, а також змінювати мову інтерфейсу та виходити з облікового запису (див. рис.3.3).

3.2 Архітектура клієнтської частини

Клієнтська частина застосунку SpendWise реалізована на базі бібліотеки React із використанням збирача Vite, що забезпечує швидку розробку та оптимізовану збірку для продакшн. Архітектура клієнтської частини організована за компонентним принципом: інтерфейс розділено на логічно відокремлені компоненти, кожен з яких відповідає за конкретну функціональну область і керує своїм власним станом.

Основним компонентом клієнтської частини є App.jsx, який ініціалізує застосунок, визначає поточний стан сесії користувача та керує відображенням

відповідного розділу інтерфейсу. Навігація між розділами здійснюється за допомогою window.location.hash без використання зовнішніх бібліотек для маршрутизації, що спрощує архітектуру та зменшує кількість залежностей.

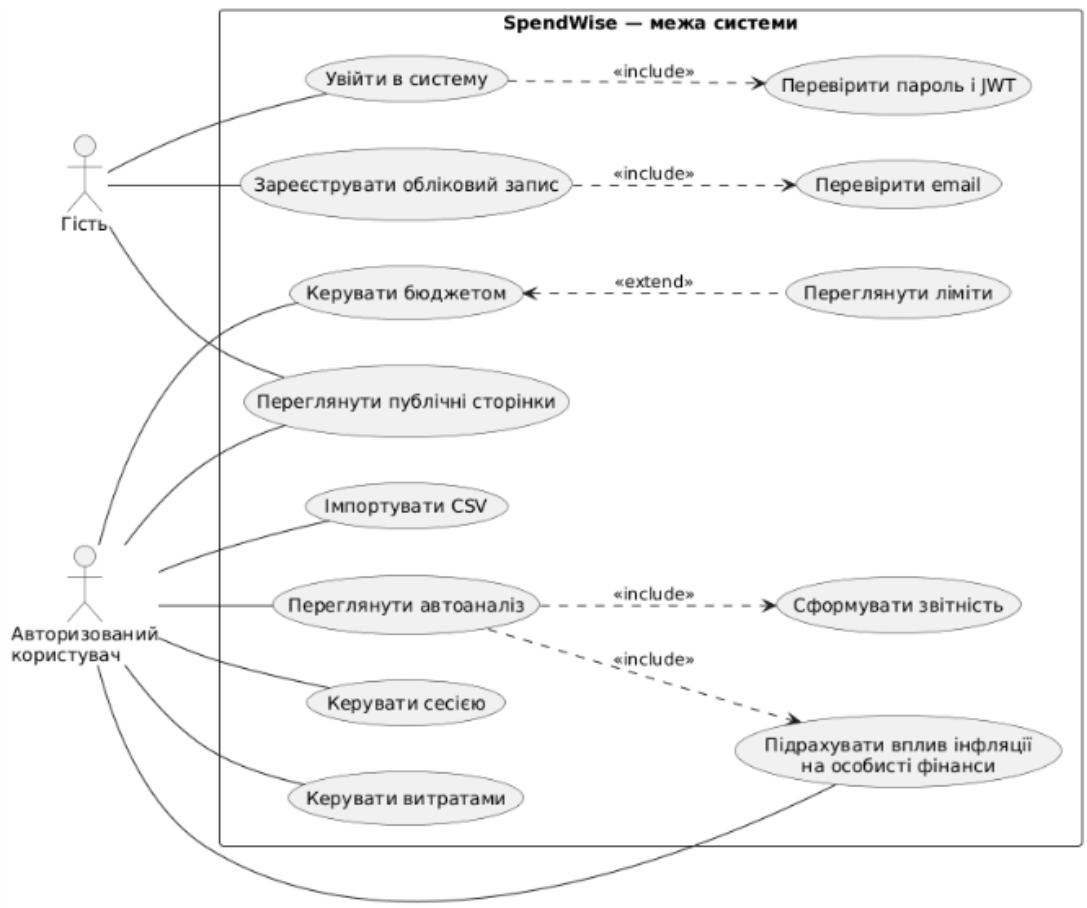


Рис. 3.3 Діаграма варіантів використання системи

Для неавторизованих користувачів застосунок показує публічну частину, яка складається з трьох розділів: лендингова сторінка з описом функціоналу (home), сторінка з інформацією про застосунок (about) та форма авторизації (auth). Перемикатись між ними можна за допомогою компонента landing-nav з відповідними хеш-посиланнями.

Після успішної авторизації користувач відкривається для нього захищена частина застосунку, головний дашборд, який об'єднує кілька функціональних модулів. Модуль фільтрування дає можливість обмежити відображення витрат за

місяцем і категорією. Модуль обліку витрат забезпечує можливість додавати нові записи через форму та редагувати існуючі. Модуль імпорту CSV реалізує завантаження банківських виписок із попереднім переглядом та підтвердженням. Модуль бюджетування надає можливість встановлювати місячні ліміти по категоріях і бачити поточний стан бюджетних алертів. Аналітичний модуль показує зведену статистику, графіки динаміки витрат і розподілу за категоріями, а також автоматичний аналіз із рекомендаціями.

Для організації взаємодії з серверною частиною використовується службовий модуль api.js, який централізує всі HTTP-запити до REST API, керує збереженням токенів у localStorage та реалізує автоматичне оновлення access-токена у разі отримання відповіді з кодом 401. Глобальні стилі інтерфейсу визначено у файлі styles.css з використанням CSS-змінних для підтримки єдиної дизайн-системи.

Архітектура забезпечує модульність, зручне розширення і високу підтримуваність, що є важливим для подальшого розвитку платформи.

3.3 Архітектура серверної частини

Серверна архітектура SpendWise реалізована з використанням фреймворку Express.js у середовищі Node.js. Підхід, застосований при проектуванні серверної архітектури, передбачає використання модульності для розділення функціональних блоків між різними компонентами архітектури. Відповідно, це спростить процес розробки, тестування та масштабування.

Загальна структура сервера починається з модуля server.js, який виконує ініціалізацію додатка Express, інтеграцію проміжного програмного забезпечення, реєстрацію маршрутів та запускає HTTP-сервер на обраному порту. Усі налаштування, включаючи порт, підключення до бази даних та секретний ключ JWT, зберігаються у файлі .env і зчитуються за допомогою бібліотеки dotenv.

Серверна архітектура використовує трирівневу модель для організації функціональності компонентів:

1. Презентаційний рівень складається з компонентів, відповідальних за прийом та обробку HTTP-запитів і повернення відповіді. Server.js містить визначення маршрутів REST API, організованих за їхніми доменами: аутентифікація /api/auth, витрати /api/expenses, статистика /api/stats, бюджети /api/budgets та імпорт /api/import. Кожен маршрут використовує один із методів HTTP GET, POST, PUT, DELETE та відповідний обробник, який повертає відповідь у форматі JSON.

2. Бізнес-логіки шар (Business Logic Layer) реалізує основну логіку роботи застосунку. Модуль auth.js містить функції генерації та перевірки JWT-токенів, функції хешування паролів і посередник (middleware) authRequired для захисту приватних маршрутів. Модуль analysis.js реалізує логіку автоматичного аналізу витрат користувача і генерації персональних рекомендацій на основі зведених даних. Вбудований CSV-парсер у server.js обробляє банківські виписки відповідно до вибраного пресету (Monobank, PrivatBank, Wise, generic).

3. Доступу до даних шар (Data Access Layer) відповідає за взаємодію з базою даних PostgreSQL. Модуль db.js створює пул з'єднань за допомогою драйвера pg та експортує його для використання у маршрутах. Усі запити до бази даних виконуються за допомогою параметризованих SQL-виразів, що запобігає SQL-ін'єкціям. Транзакційна цілісність при виконанні імпорту та відкату операцій забезпечується командами BEGIN / COMMIT / ROLLBACK.

Завдяки такій структурі можна ізолювати логіку різних частин застосунку, мінімізувати залежності між модулями та підтримувати чисту архітектуру, яка дозволяє розвивати кожен компонент окремо.

З архітектурної точки зору вибір пулу з'єднань замість окремих з'єднань має вирішальне значення. Пул з'єднань постійно підтримує певну кількість відкритих з'єднань, які можна повторно використовувати для кожного запиту, що позбавляє необхідності встановлювати нові з'єднання для кожного запиту. Це мінімізує час встановлення з'єднання та дозволяє обробляти одночасні запити від декількох клієнтів.

На діаграмі класів нижче представлено структуру основних сутностей серверної частини та зв'язки між ними (див. рис. 3.4).

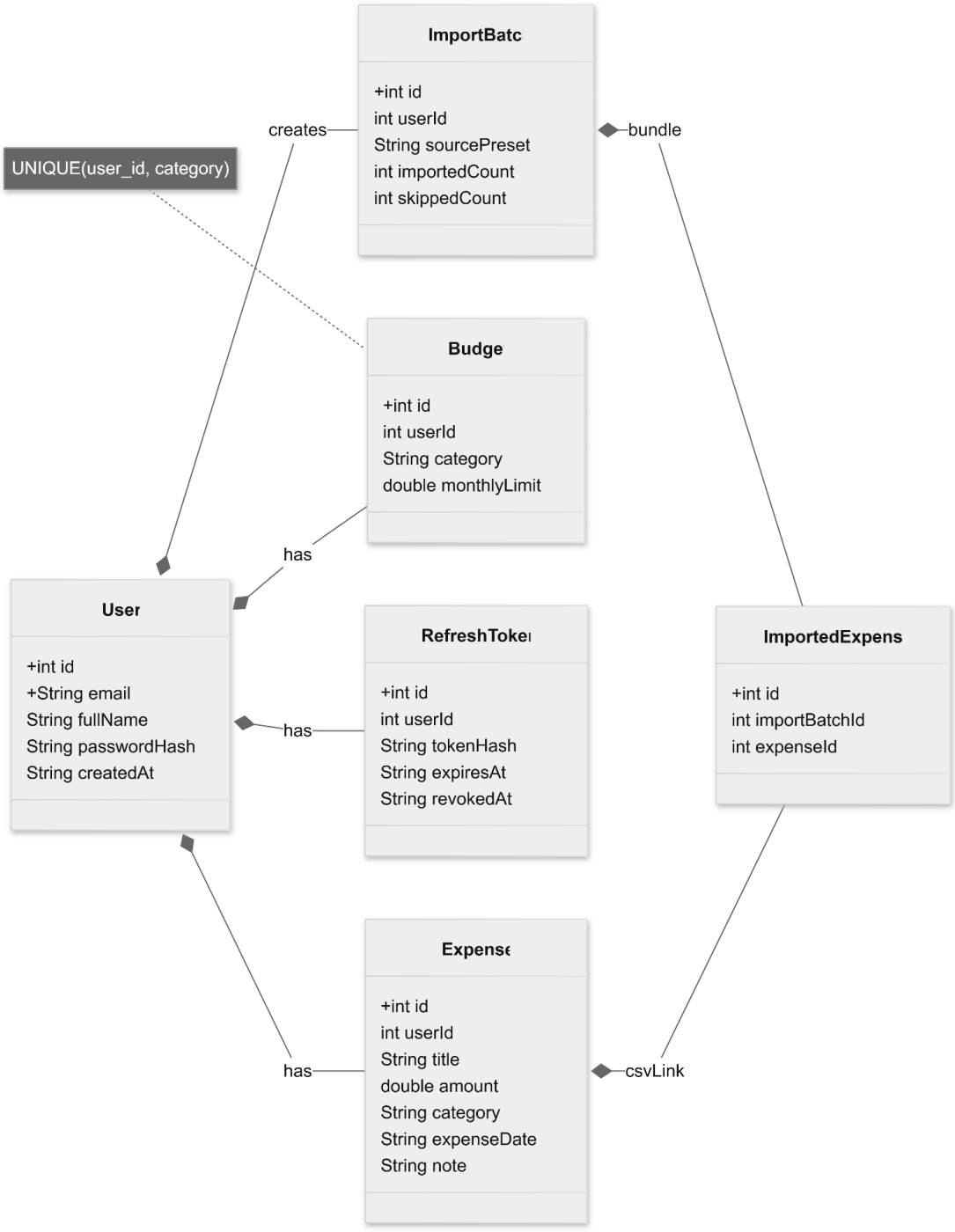


Рис. 3.4 Діаграма класів серверної частини застосунку

3.4 Архітектура бази даних

Базу даних проєкту реалізували за допомогою реляційної моделі і вона працює на СУБД PostgreSQL. Такий підхід дозволяє надійно, структуровано і логічно розміщувати дані, що актуально для пов’язаних сутностей, наприклад, користувачі, витрати, бюджети, токени сесій та імпортований журнал.

У базі даних додатка SpendWise є п'ять таблиць, кожна представляє окрему сутність у системі. Таблиця users зберігає інформацію про користувачів, expenses, записи про фінансові операції, які прив'язані до категорії та дати, budgets, ліміти витрат за категоріями на місяць, refresh_tokens, хешовані токени для активних сесій, а import_sessions, журнал операцій з імпорту банківських звітів.

Кожна таблиця:

Має колонки (поля), вони визначають тип даних, що зберігаються (текст, число, дата, часова мітка тощо). Про структуру таблиць у вигляді SQL-схеми описано у файлі schema.sql, що дозволяє у відтворюваний спосіб розгортати базу даних у будь-якому середовищі.

Має рядки (записи), це окремі екземпляри сутностей. Наприклад, у таблиці expenses один запис це одна витрата конкретного користувача з певною датою, сумою та категорією.

Для забезпечення відношень між сутностями та гарантії цілісності даних зв’язки між таблицями реалізовані за допомогою зовнішніх ключів. Це дає можливість встановлювати відношення між сутностями та гарантію цілісності даних. Запити до бази даних виконуються з серверної частини через драйвер pg із використанням параметризованих SQL-запитів. Такий підхід дозволяє захиститися від SQL-ін'єкцій та передбачувано працювати з даними.

У системі реалізовано такі зв’язки:

Один до багатьох (One-to-Many): кожен користувач (users) може мати багато витрат (expenses), бюджетів (budgets), токенів (refresh_tokens) та сесій імпорту (import_sessions), але кожен із цих записів належить лише одному користувачу.

Необов'язковий один до багатьох (Optional One-to-Many): таблиця import_sessions пов'язана з expenses через необов'язковий зовнішній ключ import_session_id, який може мати значення NULL. Витрати, додані вручну, не належать жодній сесії імпорту, тоді як імпортовані записи групуються в межах однієї сесії. Це дозволяє відстежувати походження кожної транзакції та виконувати відкат цілого пакету імпорту одним запитом без впливу на записи, внесені вручну.

Щоб підвищити продуктивність виконання запитів на полях user_id та expense_date таблиці expenses визначено індекси. Вони суттєво прискорюють виконання типових операцій фільтрації та агрегації даних при побудові статистики.

Для наочного відображення структури бази даних побудовано ER-діаграму, яка демонструє таблиці, типи даних, первинні та зовнішні ключі, а також зв'язки між основними сутностями системи.

Усі зв’язки зовнішніх ключів у системі налаштовані на каскадне видалення за допомогою опції ON DELETE CASCADE. Це означає, що під час видалення основної сутності всі дочірні записи, пов’язані з цією сутністю, видаляються автоматично. Наприклад, під час видалення певного користувача його витрати, бюджети, токени оновлення та сеанси імпорту видаляються автоматично. У базі даних не залишається жодних «покинутих» записів.

Окрім зв’язків між таблицями, цілісність даних можна забезпечити за допомогою типів даних та обмежень, застосованих безпосередньо до стовпців. Використання типу даних NUMERIC без вказівки масштабу дозволяє уникнути проблем з округленням, властивих числам з плаваючою комою при обробці грошових значень. Деякі стовпці вимагають введення значень (обмеження NOT NULL), тоді як для інших стовпців встановлюються значення за замовчуванням, що дозволяє уникнути появи некоректних записів під час введення даних.

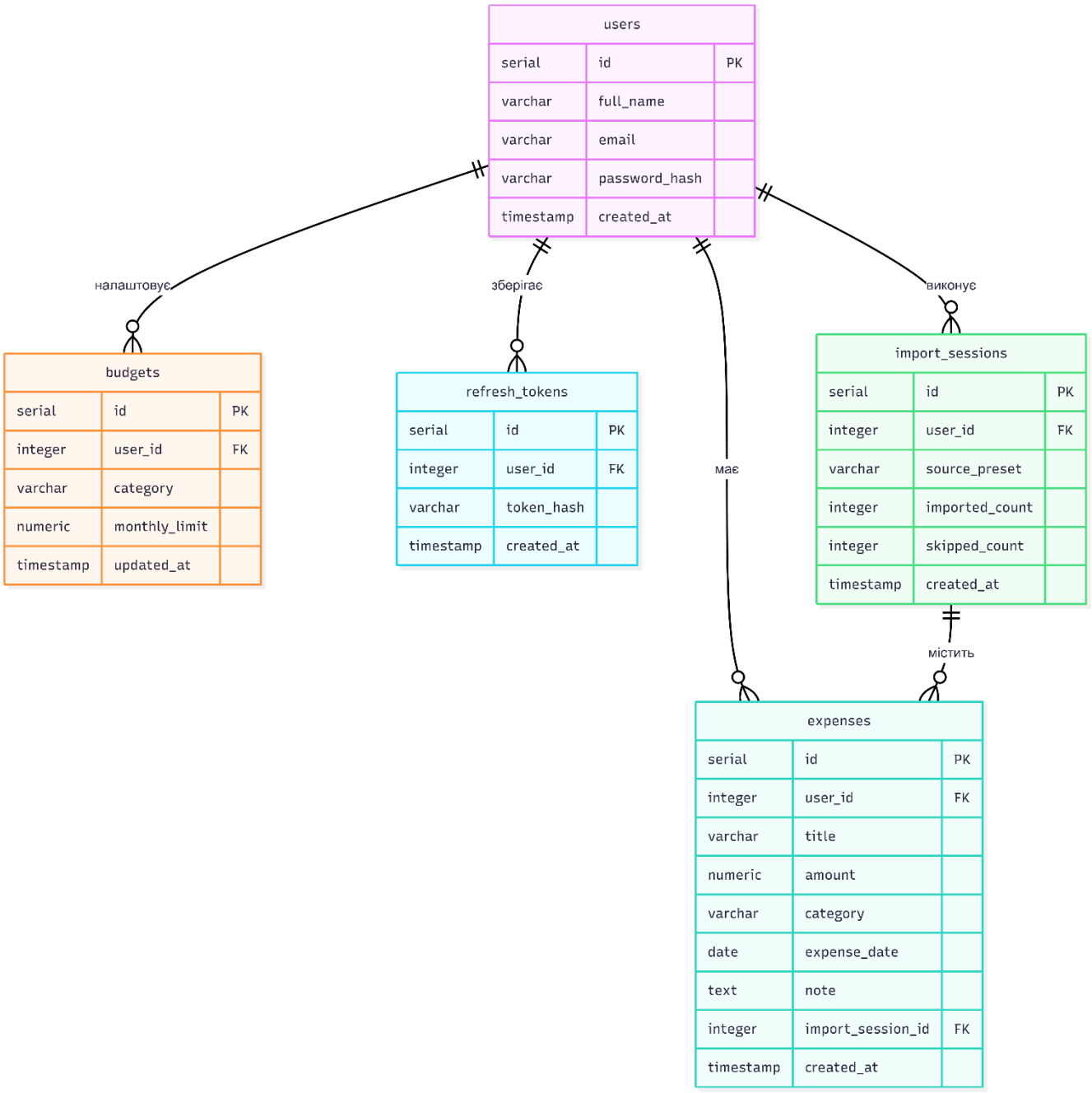


Рис. 3.5 ER-діаграма структури бази даних системи

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Реалізація серверної частини

Серверна частина застосунку SpendWise побудована на базі фреймворка Express.js у середовищі виконання Node.js. Основними перевагами обраного стеку є його легковаговість, висока швидкодія при обробці асинхронних запитів, використання єдиної мови програмування JavaScript для клієнтської та серверної частин, а також широкий вибір пакетів екосистеми npm, що суттєво прискорює розробку.

4.1.1 Загальна архітектура серверного застосунку

Ініціалізація серверного проєкту виконана через пакетний менеджер npm з використанням модульної системи ECMAScript (ESM), що дозволяє застосовувати сучасний синтаксис import/export замість застарілої системи CommonJS. Конфігурацію типу модулів визначено у файлі package.json через налаштування "type": "module".

Залежності серверного застосунку включають:

- express (v4.19.2) — мінімалістичний веб-фреймворк для побудови REST API;
- pg (v8.12.0) — драйвер PostgreSQL для Node.js із підтримкою пулу з'єднань;
- bcryptjs (v2.4.3) — бібліотека для безпечного хешування паролів із використанням алгоритму bcrypt;
- jsonwebtoken (v9.0.2) — реалізація JSON Web Token (RFC 7519) для автентифікації;
- cors (v2.8.5) — middleware для налаштування Cross-Origin Resource Sharing;
- dotenv (v16.4.5) — завантаження змінних середовища з файлу .env.

Для розробки додатково використовується nodemon (v3.1.4), що забезпечує автоматичний перезапуск серверу при зміні файлів вихідного коду.

4.1.2 Структура проєкту та організація модулів

Архітектура серверної частини побудована за монолітним патерном із логічним розділенням відповідальностей між модулями:

```
backend/
├── .env                - конфігурація середовища
├── .env.example        - шаблон конфігурації
├── package.json        - маніфест проєкту та залежності
├── db/
│   └── schema.sql      - DDL-скрипт бази даних
└── src/
    ├── server.js       - головний файл (маршрути, бізнес-логіка, CSV)
    ├── db.js           - підключення до PostgreSQL
    ├── auth.js         - автентифікація та авторизація
    └── analysis.js     - автоматичний аналіз та інфляційна модель
```

Головний файл server.js об'єднує реєстрацію middleware, визначення всіх REST-маршрутів, логіку парсингу CSV-файлів, обробку транзакцій бази даних та допоміжні утиліти інтернаціоналізації. Такий підхід забезпечує простоту розгортання та зменшує накладні витрати на координацію модулів, залишаючись цілком прийнятним для застосунку поточного масштабу.

4.1.3 Конфігурація середовища

Налаштування середовища виконання визначено у файлі .env, який завантажується за допомогою бібліотеки dotenv при старті серверу. Такий підхід відповідає методології Twelve-Factor App та дозволяє розмежувати конфігурацію між середовищами розробки, тестування та продуктивним розгортанням:

```
PORT=4000
DB_HOST=localhost
```

```
DB_PORT=5432
DB_USER=postgres
DB_PASSWORD=postgres
DB_NAME=spendwise
JWT_SECRET=your_secret_key
```

Змінна PORT визначає порт, на якому Express-сервер очікує вхідні HTTP-з'єднання. Блок DB_* налаштовує параметри підключення до PostgreSQL. Змінна JWT_SECRET містить криптографічний секрет для підпису та верифікації токенів автентифікації. У продуктивному середовищі рекомендується використовувати випадково згенерований секрет довжиною не менше 256 біт.

4.1.4 Підключення до бази даних

Підключення до PostgreSQL реалізовано через пул з'єднань (connection pool) драйвера pg, що забезпечує ефективне повторне використання TCP-з'єднань при обробці паралельних HTTP-запитів. Пул автоматично керує життєвим циклом з'єднань: створює нові при потребі, повертає звільнені до пулу після завершення запиту та закриває неактивні з'єднання після таймауту:

```
import pg from "pg";
import dotenv from "dotenv";

dotenv.config();

const { Pool } = pg;

export const pool = new Pool({
  host: process.env.DB_HOST,
  port: Number(process.env.DB_PORT || 5432),
```

```
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME
});
```

За замовчуванням пул pg утримує до 10 одночасних з'єднань, що є оптимальним для одноекземплярного розгортання. Параметр порту зчитується зі змінної середовища із резервним значенням 5432 (стандартний порт PostgreSQL). Модуль експортує єдиний екземпляр пулу, який імпортується в усі інші модулі, що потребують доступу до бази даних.

4.1.5 Ініціалізація Express-застосунку та middleware

Точкою входу серверного застосунку є файл server.js, який створює екземпляр Express, підключає глобальні middleware та визначає маршрути:

```
import express from "express";
import cors from "cors";
import dotenv from "dotenv";
import bcrypt from "bcryptjs";
import { pool } from "./db.js";
import { buildAutoAnalysis } from "./analysis.js";
import {
  authRequired, hashToken, signAccessToken,
  signRefreshToken, verifyRefreshToken
} from "./auth.js";

dotenv.config();

const app = express();
```

```
const port = Number(process.env.PORT || 4000);

app.use(cors());
app.use(express.json());
```

Middleware cors() дозволяє міждоменні запити з клієнтської частини, що працює на іншому порту під час розробки (Vite dev-сервер на порту 5173). Middleware express.json() автоматично розбирає тіло запитів у форматі JSON та додає розпарсений об'єкт до req.body.

4.1.6 Інтернаціоналізація серверних відповідей

Сервер підтримує двомовність повідомлень про помилки та статуси (українська та англійська мови). Мова визначається на основі HTTP-заголовка Accept-Language, який передається клієнтом:

```
function lang(req) {
  const header = String(req.headers["accept-language"] || "en").toLowerCase();
  return header.startsWith("uk") ? "uk" : "en";
}

function tr(req, uk, en) {
  return lang(req) === "en" ? en : uk;
}
```

Функція lang() аналізує заголовок запиту та повертає код мови. Функція tr() приймає обидва варіанти тексту та повертає відповідний до визначеної мови. Такий підхід дозволяє зберігати серверні повідомлення зрозумілими для кінцевого користувача незалежно від його мовних налаштувань.

4.1.7 Архітектура токенів

Автентифікацію реалізовано на основі стандарту JSON Web Token (JWT, RFC 7519) із двома типами токенів, що реалізує патерн «short-lived access + long-lived refresh»:

- access-токен - має час життя 15 хвилин (expiresIn: "15m"), передається у заголовку Authorization: Bearer <token> кожного захищеного запиту. Короткий термін дії мінімізує вікно уразливості при компрометації токена.
- refresh-токен - має час життя 30 днів (expiresIn: "30d"), зберігається у базі даних у хешованому вигляді (SHA-256) та використовується виключно для отримання нової пари токенів без повторного введення облікових даних.

```
export function signAccessToken(payload) {
  return jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: "15m" });
}

export function signRefreshToken(payload) {
  return jwt.sign(payload, process.env.JWT_SECRET, { expiresIn: "30d" });
}
```

Payload обох токенів містить userId та email автентифікованого користувача, що дозволяє серверу ідентифікувати відправника запиту без додаткового звернення до бази даних.

4.1.8 Безпечне зберігання refresh-токенів

Refresh-токени не зберігаються у відкритому вигляді. Перед збереженням до бази даних токен хешується за допомогою алгоритму SHA-256:

```
export function hashToken(token) {
  return
crypto.createHash("sha256").update(token).digest("hex");
}
```

При спробі оновлення сесії сервер хешує отриманий від клієнта refresh-токен і порівнює результат із збереженим хешем у таблиці refresh_tokens. Такий підхід гарантує, що навіть у разі несанкціонованого доступу до бази даних зловмисник не зможе використати збережені хеші для відновлення дійсних токенів.

4.1.9 Middleware авторизації

Middleware authRequired захищає всі приватні маршрути API, виконуючи перехоплення вхідних HTTP-запитів та послідовну перевірку облікових даних:

```
export function authRequired(req, res, next) {
  const authHeader = req.headers.authorization;
  const token = authHeader?.startsWith("Bearer ") ?
authHeader.slice(7) : null;
  const isEn = String(req.headers["accept-language"] || "uk")
    .toLowerCase().startsWith("en");

  if (!token) {
    return res.status(401).json({
      message: isEn ? "Authorization is required" : "Потрібна
авторизація"
    });
  }
```

```
      try {
        const decoded = jwt.verify(token,
process.env.JWT_SECRET);
        req.user = decoded;
        return next();
      } catch (_e) {
        return res.status(401).json({
          message: isEn ? "Invalid or expired token" : "Недійсний
або прострочений токен"
        });
      }
    }
```

Алгоритм роботи middleware складається з кількох послідовних кроків:

- Отримання токена - middleware витягує значення заголовка Authorization з вхідного запиту, перевіряє наявність префіксу Bearer та ізолює сам токен.
- Перевірка наявності - якщо заголовок відсутній або не відповідає очікуваному формату, запит негайно відхиляється з HTTP-статусом 401 (Unauthorized) та локалізованим повідомленням про необхідність авторизації.
- Верифікація - токен перевіряється за допомогою бібліотеки jsonwebtoken з використанням секретного ключа зі змінних середовища. Під час верифікації перевіряються; цілісність підпису (signature) — гарантує, що payload не був - модифікований після підпису; термін дії (поле exp) - токен з вичерпаним терміном дії автоматично відхиляється, структурна коректність - malformed JWT призводить до виключення; передача контексту - у разі успішної верифікації декодований payload (що містить userId та email) прикріплюється до об'єкта запиту як req.user, забезпечуючи подальшим обробникам доступ до інформації про автентифікованого

користувача; делегування - виклик next() передає управління наступному обробнику в ланцюжку middleware Express.

4.1.10 Реєстрація нового користувача

Маршрут POST /api/auth/register реалізує повний цикл створення облікового запису:

```
app.post("/api/auth/register", async (req, res) => {
  const { fullName, email, password } = req.body;
  if (!fullName || !email || !password) {
    return res.status(400).json({ message: "Заповніть ім'я, email і пароль" });
  }
  if (String(password).length < 6) {
    return res.status(400).json({ message: "Пароль має містити щонайменше 6 символів" });
  }

  const exists = await pool.query("SELECT id FROM users WHERE email = $1", [email.toLowerCase()]);
  if (exists.rows[0]) {
    return res.status(409).json({ message: "Користувач з таким email вже існує" });
  }

  const passwordHash = await bcrypt.hash(password, 10);
  const result = await pool.query(
    `INSERT INTO users (full_name, email, password_hash)
     VALUES ($1, $2, $3) RETURNING id, full_name, email`,
    [fullName, email.toLowerCase(), passwordHash]
  );
  const user = result.rows[0];
  const tokens = await createTokenPair(user);
  return res.status(201).json({ ...tokens, user });
});
```

Процес реєстрації включає:

- валідацію вхідних даних (наявність обов'язкових полів, мінімальна довжина пароля 6 символів);

- перевірку унікальності email-адреси з нормалізацією до нижнього регістру;
- хешування пароля алгоритмом bcrypt з cost-фактором 10 (2^10 = 1024 раунди), що забезпечує стійкість до атак brute-force навіть при витоку хешів;
- створення запису користувача в базі даних;
- генерацію пари access/refresh токенів та збереження хешу refresh-токена;
- повернення токенів та профілю користувача у відповіді

4.2 Реалізація клієнтської частини

Архітектура на стороні клієнта розроблена за принципом компонентної архітектури, що лежить в основі бібліотеки React. Кожен елемент інтерфейсу - від кнопки до цілої сторінки - відповідає окремому компоненту, який має власну логіку та код шаблону, що забезпечує максимальну можливість повторного використання кодової бази та полегшує обслуговування, а також окрему розробку й тестування кожного компонента інтерфейсу.

Маршрутизація на стороні клієнта з однієї сторінки додатка на іншу здійснюється за допомогою бібліотеки React Router, що забезпечує можливість навігації всередині клієнтського додатка без необхідності звернень до сервера. Крім того, приватні маршрути захищені перевіркою наявності токена JWT у локальному сховищі браузера. За відсутності цієї інформації користувачі будуть перенаправлені на сторінку авторизації.

Взаємодія з сервером бекенд-додатку здійснюється бібліотекою Axios через протокол HTTP. Запити до захищених URL-адрес автоматично доповнюються HTTP-заголовком Authorization за допомогою налаштування відповідних інтерцепторів Axios, які містять поточний токен.

Клієнтський застосунок має просту та лаконічну структуру, де весь інтерфейс зосереджено в кількох ключових файлах:

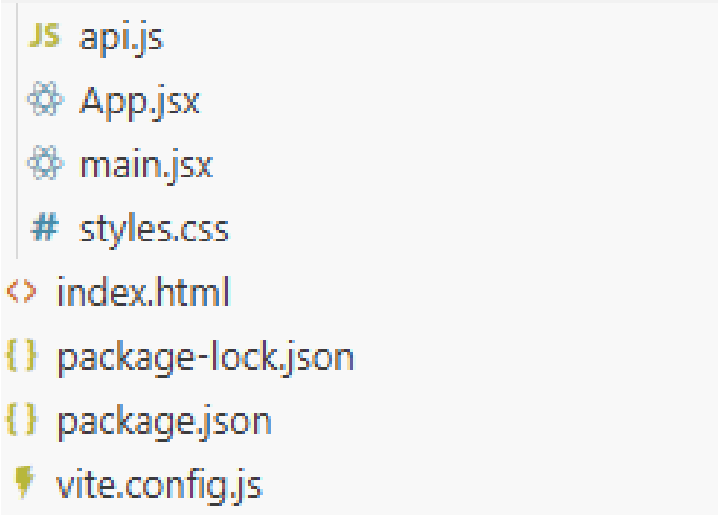


Рис. 4.1 Структура Клієнтського застосунку

- main.jsx - точка входу, монтування React-застосунку
- App.jsx - головний компонент, стан сесії, вся логіка UI
- api.js - HTTP-клієнт, управління токенами
- styles.css - глобальні стилі, CSS-змінні

Для взаємодії з серверною частиною було реалізовано окремий модуль api.js, у якому зосереджені всі HTTP-запити до REST API. Для авторизованих запитів JWT-токен передається у заголовку Authorization:

```
const baseHeaders = {
  "Content-Type": "application/json",
  "Accept-Language": getLanguage()
};
if (token) baseHeaders.Authorization = `Bearer ${token}`;
```

Зокрема, зроблено функції для:

- реєстрації та входу користувача з автоматичним збереженням токенів;
- автоматичного оновлення access-токена при отриманні відповіді 401;
- отримання, створення, редагування та видалення витрат;

- імпорту CSV-виписок із попереднім переглядом та підтвердженням;
- отримання статистики, аналітики та бюджетних алертів.

Перемикання між розділами публічної частини застосунку зроблено через window.location.hash без використання бібліотек для маршрутизації:

```
useEffect(() => {
const applyHashPage = () => {
const page = window.location.hash.replace("#", "");
if (["home", "about", "auth"].includes(page)) {
          setPublicPage(page);
        }
};
window.addEventListener("hashchange", applyHashPage);
      return () => window.removeEventListener("hashchange",
applyHashPage);
    }, [isLoggedIn]);
```

Перевірку автентифікації при завантаженні застосунку реалізовано через зчитування токена з localStorage. У разі наявності активної сесії користувач одразу потрапляє на дашборд, в іншому випадку відображається публічна частина:

```
const [session, setSession] = useState(api.getSession());
const isLoggedIn = Boolean(session.accessToken);
```

Логіку стану застосунку покладено винятково на React-хуки. Так, useState служить сховищем для списку витрат, фільтрів, форм та станів завантажень, useEffect відповідає за підвантажування даних при зміні фільтрів чи обліковому записі користувача, а useMemo оптимізує виконання функції форматування числових значень:

```
```

```
    const totalAmountFormatted = useMemo(
      () => Number(summary.total_amount ||
0).toLocaleString(locale, {
        style: "currency", currency: "UAH"
      }),
      [summary.total_amount, locale]
    );
```

Для аналітичного розділу була використана бібліотека Recharts. Стовпчаста діаграма показує зміну витрат у розрізі місяців, а кільцева, розподіл витрат за категоріями. Обидві діаграми бувають адаптивними завдяки обгортці у компонент ResponsiveContainer та оновлюються автоматично при зміні фільтрів.

Двомовність (українська та англійська) було реалізовано шляхом вбудованої системи перекладів у вигляді об’єктів translations.uk і translations.en. Поточна мова зберігається у localStorage та існує для мовних стрічок інтерфейсу, а також для форматування числових значень і дат відповідно до локалі (uk-UA чи en-US).

Модуль HTTP-клієнта (api.js) реалізує механізм автоматичного оновлення сесії (silent refresh) при отриманні від серверу відповіді зі статусом 401 (Unauthorized). Коли access-токен втрачає дійсність, функція-обгортка request() перехоплює помилку, ініціює запит на оновлення через збережений refresh-токен, і після отримання нової пари токенів автоматично повторює оригінальний запит вже з валідним access-токеном.

Для уникнення паралельних запитів на оновлення використовується патерн «shared promise» — змінна refreshPromise зберігає єдиний проміс оновлення, до якого приєднуються всі одночасні запити, що очікують нову сесію:

```
    if (!refreshPromise) {
```

```
      refreshPromise = refreshSession().finally(() => { refreshPromise = null; });
    }
    await refreshPromise;
    return request(path, options, false);
```

Такий підхід гарантує, що навіть при одночасних 401-відповідях на кілька запитів до серверу відбудеться лише один запит на оновлення токена, після чого всі очікувані запити будуть повторені з новим access-токеном.

Завантаження даних у компоненті дашборду виконується паралельно за допомогою Promise.all(), що об'єднує сім незалежних запитів (витрати, підсумки, помісячна статистика, розподіл за категоріями, аналітика, бюджети, алерти) в одну асинхронну операцію. Це суттєво зменшує загальний час завантаження порівняно з послідовним виконанням запитів і забезпечує атомарне оновлення стану — всі дані відображаються одночасно, а не частинами:

```
    const [expenseData, summaryData, monthlyData, categoryData, analysisData, budgetData, alertsData] =
      await Promise.all([
        api.getExpenses(filters),
        api.getSummary(filters),
        api.getMonthly(),
        api.getCategoryBreakdown({ month: filters.month }),
        api.getAnalysis({ month: filters.month, inflationAnnualPct }),
        api.getBudgets(),
        api.getBudgetAlerts({ month: filters.month })
      ]);
```

Ефект useEffect відстежує зміни ключових залежностей - фільтра місяця, категорії, поточної мови та параметра інфляції - і автоматично перезавантажує дані при будь-якій їхній зміні, що забезпечує реактивне оновлення інтерфейсу без

ручного перезавантаження сторінки. Управління станом завантаження через прапорець loading дозволяє відображати індикатор прогресу під час очікування відповідей серверу, покращуючи користувацький досвід при повільному з'єднанні.

4.3 Інтеграція та взаємодія компонентів

Після реалізації окремих функціональних частин застосунку SpendWise їх було інтегровано у повноцінну систему. Компоненти клієнтської частини (React), серверної частини (Express.js) та бази даних (PostgreSQL) взаємодіють через REST API, що і забезпечує безперервну та стабільну роботу системи.

Вся взаємодія фронт- і бекендом ґрунтується на HTTP-запитах до REST API. React-застосунок надсилає запити до певних ендпоінтів, отримує у відповідь JSON-дані і відображає їх у вигляді інтерактивного інтерфейсу. Наприклад, дашборд при завантаженні одночасно робить сім паралельних запитів для отримання витрат, зведеної статистики, помісячних трендів, розподілу витрат за категоріями, результатів аналізу, бюджетів та алертів:

```
    const [expenseData, summaryData, monthlyData, categoryData,
           analysisData,  budgetData,  alertsData] = await
Promise.all([
      api.getExpenses(filters),
      api.getSummary(filters),
      api.getMonthly(),
      api.getCategoryBreakdown({ month: filters.month }),
      api.getAnalysis({ month: filters.month }),
      api.getBudgets(),
      api.getBudgetAlerts({ month: filters.month })
    ]);
```

Після успішної автентифікації користувач отримує два типи токенів — токен доступу та токен оновлення. Обидва вони зберігаються у локальному сховищі та додаються до заголовків у кожному наступному запиті за допомогою спеціального

інтерцептора Axios. Термін дії токена доступу швидко закінчується, і він використовується для автентифікації кожного запиту до захищених маршрутів, тоді як токен оновлення діє довше і використовується виключно для отримання токена доступу. Серверна сторона перевіряє легітимність кожного запиту за допомогою проміжного програмного забезпечення authRequired, яке бере токен із заголовка Authorization і перевіряє його автентичність.

Якщо термін дії токена доступу закінчується, сервер відповідає кодом статусу 401, і клієнт надсилає ще один запит для отримання нового токена доступу на кінцеву точку токена оновлення. Після успішного отримання токена токен доступу знову зберігається в локальному сховищі, і початковий запит повторюється. Усі ці дії відбуваються автоматично, без втручання користувача. Іншими словами, це рішення поєднує безпеку та зручність, оскільки термін дії токена доступу швидко закінчується, що зменшує можливі ризики, а процес запиту токена доступу відбувається автоматично, що дозволяє уникнути повторної автентифікації.

Якщо термін дії access-токена минув, клієнтська частина автоматично надсилає запит на отримання нового access-токена за допомогою refresh-токена, після чого повторює оригінальний запит без участі користувача:

```
    if (response.status === 401 && retry) {
      await refreshSession();
      return request(path, options, false);
    }
```

Клієнтська частина взаємодіє з усіма основними частинами системи:

- розділ витрат, дозволяє додавати, редагувати, видаляти та фільтрувати фінансові записи за місяцем і категорією;
- імпорт CSV, реалізує завантаження банківських виписок з можливістю попереднього перегляду, підтвердження і скасування останньої дії;

- бюджетування, дозволяє встановлювати місячні обмеження за категоріями і отримувати алерти у трьох станах: все в нормі, наближаєтесь до ліміту, ліміт вичерпано;
- аналітика, показує зведену статистику витрат, графіки трендів і розподілу за категоріями, а також автоматичний аналіз з рекомендаціями.

Кожна дія користувача спричиняє відповідну логіку на стороні сервера, яка змінює стан у базі даних і потім оновлені дані надсилаються назад і відображаються на екрані. Наприклад, після підтвердження імпорту CSV сервер у межах транзакції виконує збереження сесії імпорту та пов'язаних з нею витрат, а потім повертає результат із кількістю імпортованих та пропущених записів. Клієнт одразу оновлює всі дані дашборду, відображаючи актуальний стан фінансів.

Інтерфейс застосунку завжди відображає актуальні дані. Так, якщо ви оберете інший місяць фільтра або іншу категорію, система поступає автоматично: перерахує статистику, оновить графіки та згенерує відповідний аналіз. Сповіщення бюджету відображають актуальний стан використання ліміту: після кожного додавання або видалення витрати їхній стан автоматично оновлюється. Із REST API, що поєднує усю логіку, застосунок легко розширювати. Для нового модулю або нової функції достатньо здійснити лише дві операції: визначити новий маршрут на сервері та відповідну функцію в api.js клієнтського модуля. Вам не потрібно чіпати код будь-яких інших модулів.

4.4 Тестування застосунку

Тестування є необхідним етапом життєвого циклу програмного забезпечення, який дозволяє переконатися у правильності реалізації функціоналу, виявити помилки та упевнитися у стійкості системи. У рамках цього проєкту щодо застосунку SpendWise було виконано функціональне тестування основних модулів серверної частини шляхом відправлення HTTP-запитів до REST API за допомогою

інструменту Postman. Тестування охоплювало всі основні функціональні модулі системи: аутентифікацію керування витратами, імпорт CSV-файлів, бюджетування та аналітику. Тестування модуля аутентифікації охоплювало такі сценарії:

- реєстрація нового користувача з валідними даними;
- спроба зареєструватися з уже використаною електронною адресою;
- вхід до системи з правильними обліковими даними;
- вхід з неправильним паролем;
- оновлення access-токена за допомогою refresh-токена;
- вихід із системи та анулляція токена.

Приклад тестового запиту для перевірки реєстрації користувача:

```
POST /api/auth/register
Content-Type: application/json
{
 "fullName": "Тестовий Користувач",
  "email": "test@example.com",
  "password": "password123"
    }
    Очікувана відповідь: 201 Created
    {
      "accessToken": "eyJ...",
      "refreshToken": "eyJ..."
    }
```

Тестування модуля витрат охоплювало такі сценарії:

- Створення нової витрати з коректними даними;
- отримання списку витрат із фільтрацією за місяцем та категорією;
- редагування існуючої витрати;
- видалення витрати;
- спроба доступу до витрат без авторизації.

Тестування модуля імпорту CSV охоплювало такі сценарії:

- формування попереднього перегляду CSV-файлу з пресетом Monobank;

- підтвердження імпорту та перевірка збереження витрат у базі даних;
- відкат останнього імпорту та перевірка видалення записів;
- імпорт файлу з некоректним форматом даних.

Тестування модуля бюджетування охоплювало такі сценарії:

- створення та оновлення місячного ліміту по категорії;
- отримання списку бюджетів користувача;
- перевірка формування алертів зі статусами ok, near та over залежно від співвідношення фактичних витрат до встановленого ліміту.

Тестування аналітичного модуля охоплювало такі сценарії:

- отримання зведеної статистики із загальною кількістю записів та сумою витрат;
- отримання помісячної динаміки витрат;
- отримання розподілу витрат за категоріями;
- отримання автоматичного аналізу з інсайтом та переліком рекомендацій.

У результаті виконання тестів було підтверджено коректність бізнес-логіки, стабільність роботи REST API, дотримання прав доступу для неавторизованих користувачів і правильність збереження та агрегації даних в базі. Усі перевірені ендпоїнти повертали очікувані HTTP-статуси та коректні JSON-відповіді згідно з вимогами.

Окрім перевірки базових функцій, значна увага приділялася також тестуванню програми в граничних випадках, зокрема тих, що передбачають введення некоректних даних, спроби несанкціонованого доступу та використання токенів JWT із вичерпаним терміном дії або неіснуючих токенів. У всіх цих випадках система видавала належні відповіді у вигляді кодів помилок, що свідчить про належну обробку винятків на стороні сервера.

ВИСНОВКИ

Під час виконання кваліфікаційної роботи було створено повноцінний веб-застосунок для керування особистими фінансами SpendWise. У ході реалізації проєкту проведено аналіз предметної області, обрано відповідні інструменти розробки, спроєктовано архітектуру системи, реалізовано функціональні компоненти та проведено їх тестування.

Досліджено предметну область персонального фінансового менеджменту та бюджетування, що дало змогу розібратися в особливостях ведення обліку витрат, у принципах категоризації фінансових операцій та у вимогах, які ставляться до інструментів фінансової аналітики. На основі цього сформульовано вимоги до майбутнього застосунку як у плані функціональних можливостей, так і у частині нефункціональних.

Проведено огляд технологій, фреймворків і мов програмування, які можуть бути використані для реалізації проєкту. Аргументовано вибір Express.js та Node.js для серверної частини, React, для клієнтської, PostgreSQL, для зберігання даних, а Recharts, для побудови аналітичних графіків і діаграм.

Спроєктовано архітектуру застосунку за принципом трирівневої організації з чітким поділом на клієнтську, серверну та бази даних. Окремі модулі було спроєктовано для обліку витрат, бюджетування, імпорту CSV-виписок та фінансової аналітики, що відповідають ключовим потребам користувачів у сфері особистих фінансів.

Реалізовано основну функціональність: реєстрацію та автентифікацію користувачів на основі JWT, додавання, редагування та видалення витрат із категоризацією, імпорт Банківських транзакцій із CSV-файлів із підтримкою пресетів Monobank, PrivatBank та Wise, встановлення місячних бюджетних лімітів із триступеневою системою сповіщень, відображення аналітики у вигляді графіків

та автоматичний аналіз витрат із персональними рекомендаціями. Усе компоненти взаємодіють між собою через REST API.

Проведено функціональне тестування всіх основних модулів застосунку (автентифікація, витрати, імпорт CSV, бюджетування, аналітика) за допомогою інструменту Insomnia, що підтвердило правильність реалізації логіки. Усі перевірені ендпоінти повернули очікувані HTTP-статуси та коректні JSON-відповіді, що свідчить про стабільність і готовність системи до використання.

Загалом, усі поставлені задачі було вирішено. Результатом роботи став сучасний веб-застосунок SpendWise, яким може бути ефективно використаний для власного контролю за особистими фінансами широкий спектр користувачів: студенти, фрілансери та домогосподарства.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Бурачик В.О., Трінтіна Н.А. РОЗРОБКА WEB-СЕРВІСУ МОНІТОРИНГУ ОСОБИСТИХ ФІНАНСІВ МОВОЮ ПРОГРАМУВАННЯ JAVASCRIPT. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунцікаційних технологій. Збірник тез. К.:ДУІКТ, 2025. С. 109-111.

2. Бурачик В.О., Трінтіна Н.А ВИЗНАЧЕННЯ ВИМОГ ДО ВЕБ-СЕРВІСУ МОНІТОРИНГУ ОСОБИСТИХ ФІНАНСІВ З АВТОМАТИЧНИМ АНАЛІЗОМ ВИТРАТ «SPENDWISE». Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Державний університет інформаційно-комунцікаційних технологій. Збірник тез. К.:ДУІКТ, 2026. С. 450-453.

ПЕРЕЛІК ПОСИЛАНЬ

1. Express.js documentation. Express. URL: https://expressjs.com/en/4x/api.html.
2. Node.js documentation. Node.js. URL: https://nodejs.org/en/docs.
3. React documentation. React. URL: https://react.dev/learn.
4. PostgreSQL: Documentation. PostgreSQL. URL: https://www.postgresql.org/docs/.
5. JavaScript Guide - JavaScript | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide.
6. JSON Web Token Introduction. JWT.io. URL: https://jwt.io/introduction.
7. Recharts documentation. Recharts. URL: https://recharts.org/en-US/api.
8. Vite documentation. Vite. URL: https://vitejs.dev/guide/.
9. OpenAPI Specification v3.1.0. OpenAPI Initiative Publications. URL: https://spec.openapis.org/oas/v3.1.0.
10. GeeksforGeeks. Class Diagram | Unified Modeling Language. GeeksforGeeks. URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/.
11. Loshin P. Personal budgeting and financial planning in the digital age. Journal of Personal Finance. 2022. Vol. 21, no. 1. P. 44–58. URL: https://doi.org/10.1016/j.jfineco.2022.01.004.
12. bcrypt documentation. npm. URL: https://www.npmjs.com/package/bcryptjs.
13. Flanagan D. JavaScript: The Definitive Guide. 7th ed. Sebastopol: O'Reilly Media, 2020. 706 p.
14. Nixon R. Learning PHP, MySQL & JavaScript: A Step-by-Step Guide to Creating Dynamic Websites. 6th ed. Sebastopol: O'Reilly Media, 2021. 832 p.
15. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol: O'Reilly Media, 2020. 310 p.

16. Wexler S. Node.js in Practice. Shelter Island: Manning Publications, 2015. 408 p.

17. Converse T., Park J., Morgan C. PHP5 and MySQL Bible. Indianapolis: Wiley Publishing, 2004. 963 p.

18. Obe R., Hsu L. PostgreSQL: Up and Running. 3rd ed. Sebastopol: O'Reilly Media, 2017. 346 p.

19. Frisbie M. Professional JavaScript for Web Developers. 4th ed. Indianapolis: John Wiley & Sons, 2019. 1200 p.

20. Rambaugh J., Jacobson I., Booch G. The Unified Modeling Language Reference Manual. 2nd ed. Boston: Addison-Wesley, 2004. 721 p.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка веб-сервісу "SpendWise" моніторингу особистих фінансів з автоматичним аналізом витрат

Виконав студент 4 курсу
групи ПД-41
Бурачик Віталій Олександрович
Керівник роботи
канд.техн.наук, доц. Трінтіна Наталія Альбертівна

Київ – 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: підвищення інтерпритованості особистих фінансів за рахунок розробки вебсервісу для обробки даних, візуалізації та автоматичного аналізу витрат.

Об'єкт дослідження: процеси представлення особистих фінансових даних користувачів.

Предмет дослідження: програмні засоби розробки вебсервісів для представлення особистих фінансових даних.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.Проаналізувати предметну галузь управління особистими фінансами.
2.Провести огляд та аналіз існуючих програмних рішень для обліку персональних витрат.
3.Визначити функціональні й нефункціональні вимоги до програмного забезпечення.
4.Визначити засоби реалізації застосунку.
5.Спроєктувати архітектуру веб-застосунку для управління особистими фінансами.
6.Реалізувати Базу даних.
7.Реалізувати Frontend та Backend.
8. Протестувати застосунок.

3

АКТУАЛЬНІСТЬ ТЕМИ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

Критерій	Money Manager	Spendee	Toshl	SpendWise
Місячні бюджетні ліміти з автоматичними сповіщеннями	-	+	-	+
Україномовний з можливістю змінити мову на англійську	-	-	-	+
Автоматичний аналіз витрат	-	-	-	+
Підтримка CSV-імпорту з пресетами українських банків	-	-	-	+
Редагування та видалення місячних витрат	+	-	+	+
Підрахунок впливу інфляції на особисті фінанси	-	-	-	+
Ведення журналу персональних витрат із реквізитами операції	+	+	+	+
Фінансова аналітика / графіки	+	+	+	+

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ФУНКЦІОНАЛЬНІ ВИМОГИ
Реєстрація та автентифікація користувача.
Додавання доходів і витрат із класифікацією за категоріями.
Автоматичний розрахунок поточного балансу з урахуванням інфляції.
Фільтрація та пошук транзакцій за датою, категорією, сумою.
Перегляд статистики за вибрані часові періоди.
Побудова графіків доходів і витрат.
Редагування та видалення фінансових записів.
Редагування коефіцієнту інфляції.
Система повинна підтримувати локалізацію інтерфейсу українською та англійською мовами.

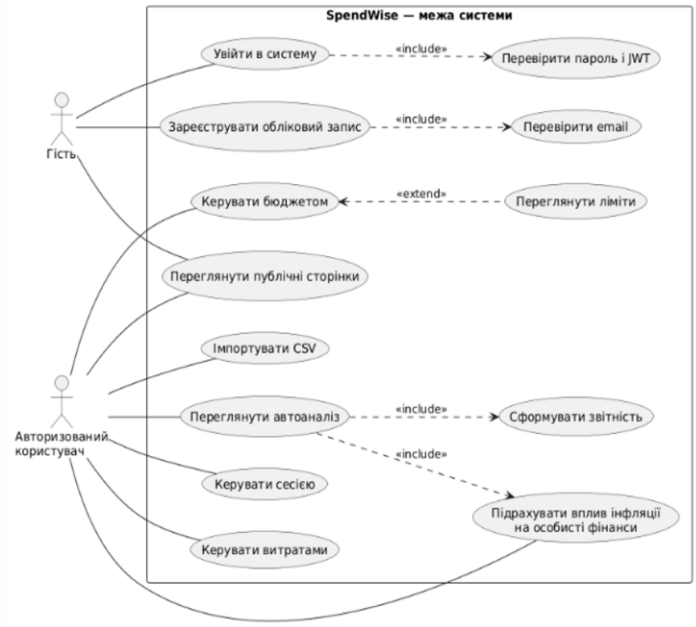
НЕФУНКЦІОНАЛЬНІ ВИМОГИ
Час відповіді API ≤ 300 мс; рендер сторінки ≤ 1 сек.
JWT-токени, хешування bcrypt, HTTPS, захист від SQL-ін'єкцій.
Кросплатформеність - ПК, планшет, смартфон. Мінімальна підтримувана роздільна здатність від 320px до 2560px.
Валідація даних на frontend та backend. Автоматичне резервне копіювання БД (щодобово), коректна робота при втраті з'єднання (graceful degradation).
Модульна архітектура - нові модулі без рефакторингу.
REST API задокументований; код покритий коментарями.
Робота в браузерах Google Chrome, Mozilla Firefox, Microsoft Edge, Safari.

5

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

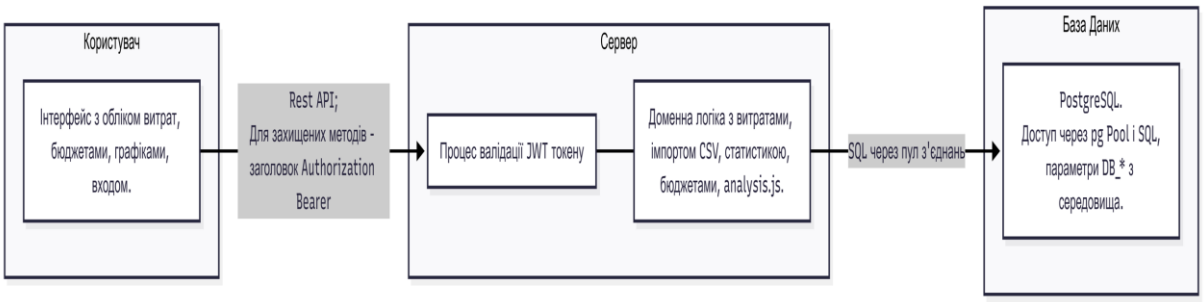
6

ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ



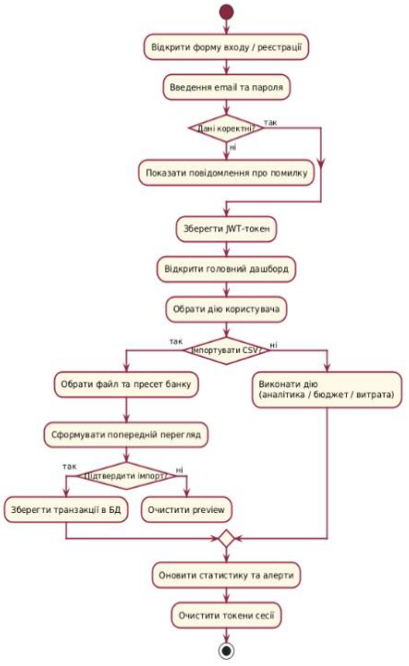
7

АРХІТЕКТУРА ЗАСТОСУНКУ



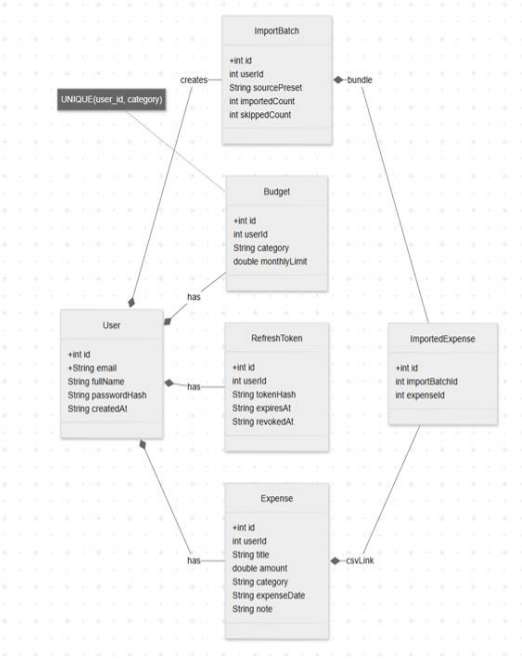
8

ДІАГРАМА ДІЯЛЬНОСТІ



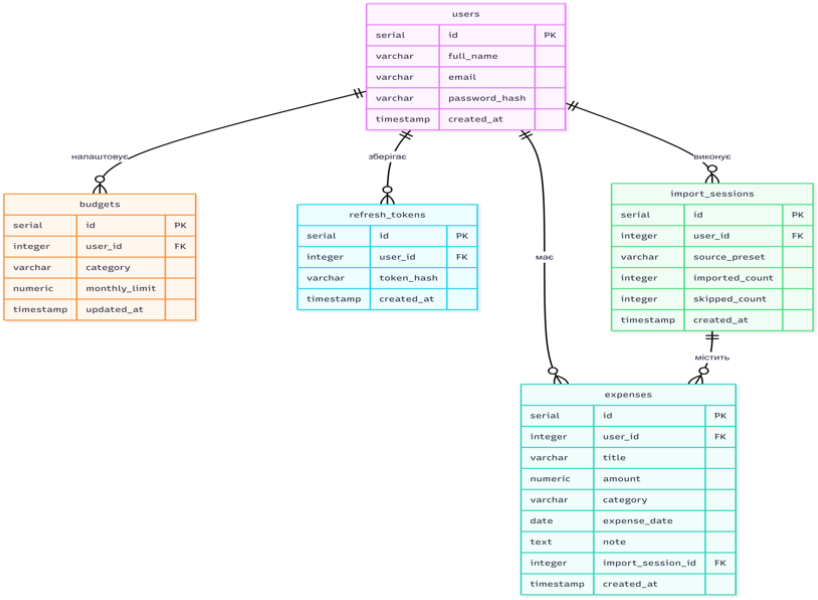
9

ДІАГРАМА КЛАСІВ СЕРВЕРНОЇ ЧАСТИНИ ЗАСТОСУНКУ



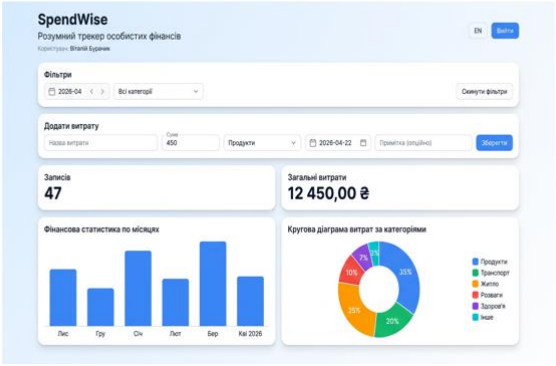
10

ER-ДІАГРАМА СТРУКТУРИ БАЗИ ДАНИХ СИСТЕМИ

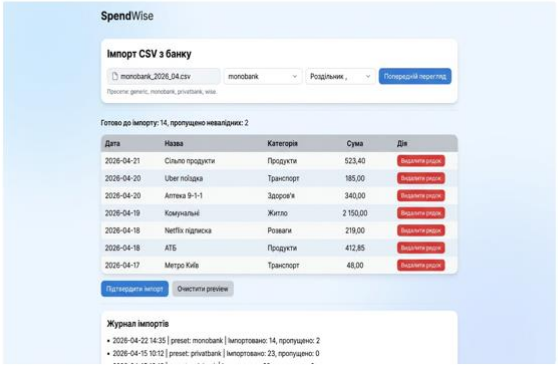


11

ЕКРАННІ ФОРМИ ЗАСТОСУНКУ



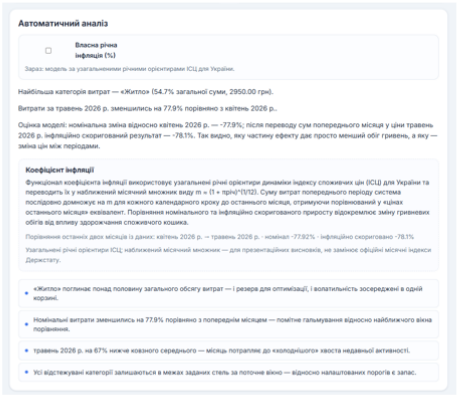
Дашборд: фільтри, підсумки, діаграми Bar і Pie



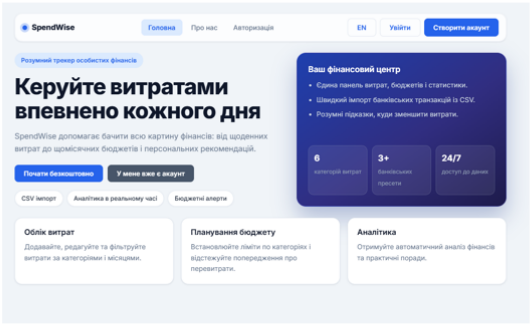
CSV-імпорт: preview з пресетами банків

12

ЕКРАННІ ФОРМИ ЗАСТОСУНКУ

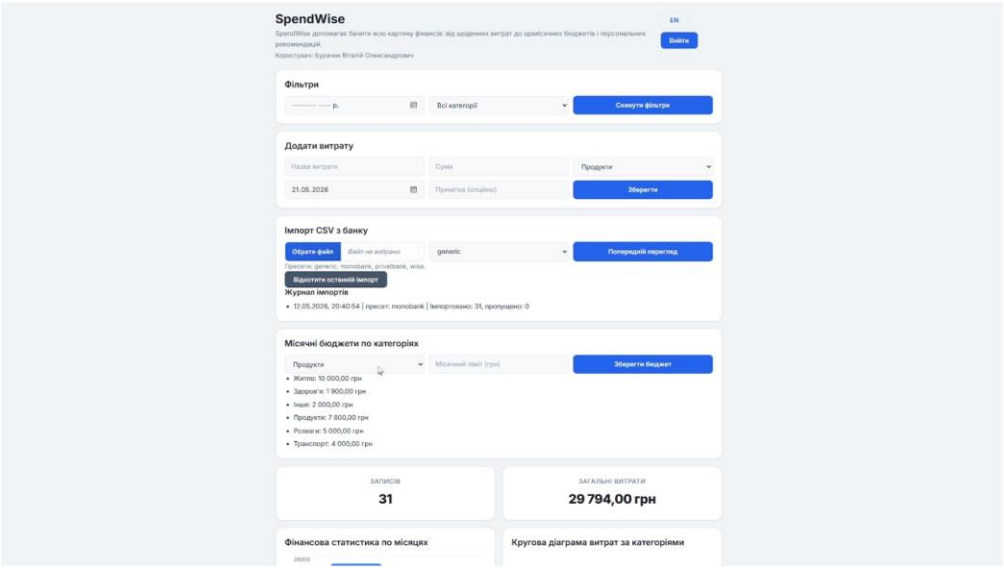


Автоматичний аналіз



Домашня сторінка

ЕКРАННІ ФОРМИ ЗАСТОСУНКУ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Бурачик В.О., Трінтіна Н.А. Розробка web-сервісу моніторингу особистих фінансів мовою програмування javascript. Матеріали II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії» 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.:ДУІКТ, 2025. С. 109-111.

2. Бурачик В.О., Трінтіна Н.А Визначення вимог до веб-сервісу моніторингу особистих фінансів з автоматичним аналізом витрат "spendwise" , VII Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно комунікаційних технологіях» 23 квітня 2026 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.:ДУІКТ, 2026. С.450-453.

ВИСНОВКИ

1. Проаналізовано предметну галузь управління особистими фінансами, та виявлено такі ключові проблеми як складність ручного обліку в умовах безготівкових розрахунків, відсутність прозорої категоризації витрат, недостатність інструментів бюджетування та своєчасної аналітики, підтримка українських банків та автоматичного аналізу з рекомендаціями.
2. Проведено огляд і порівняльний аналіз існуючих програмних рішень для обліку персональних витрат (типовий функціонал: категорії, ліміти, звіти, імпорт) на основі якого було визначено функціональні та нефункціональні вимоги до застосунку, що роблять розробку вебсервісу доцільною.
3. На основі аналізу існуючих рішень визначено засоби реалізації: клієнт — React 18, збірка Vite, графіки Recharts; сервер — Node.js, фреймворк Express, JWT та bcrypt, доступ до даних через node-postgres; СУБД — PostgreSQL (pool, змінні DB_*); обмін даними з клієнтом — REST API, формат JSON, базовий шлях /api.
4. Спроєктовано архітектуру веб-застосунку у вигляді трьохрівневої моделі «клієнт — сервер застосунків — база даних» і логічну модель основних модулів (клієнтський API-шар, маршрути Express, модуль аналітики та інфляційних допущень для порівняння періодів).
5. Розроблено реляційну базу даних. Сутності: користувач, витрата, бюджет, refresh-токени, партії імпорту, прив'язка імпорту до витрат; реалізовано обмеження цілісності та унікальність пари (користувач, категорія) для бюджету; категорія витрат моделюється як атрибут (рядок), окремої таблиці категорій немає.
6. Розроблено застосунок: Frontend — інтерфейс обліку, звітів, бюджетів, імпорту, блоку автоаналізу, налаштування річного % інфляції, hash-маршрутизація публічних сторінок; Backend — ендпоїнти автентифікації, витрат, статистики, бюджетів, імпорту CSV, формування текстового аналізу та підказок.
7. Виконано ручне функціональне тестування через браузер за тест-кейсами, також за допомогою Insomnia — тестування REST API ендпоінтів напряму, яке дозволило підвердити коректність роботи застосунку і відповідність функціональним та нефункціональним вимогами.

ДОДАТОК Б. ЛІСТИНГИ ПРОГРАМНИХ МОДУЛІВ

Реєстрація та вхід користувача (server.js)

```
app.post("/api/auth/register", async (req, res) => {
  const { fullName, email, password } = req.body;
  const lang = getLang(req);
  if (!email || !password || !fullName) {
    return res.status(400).json({ message: tr(req,
"Заповніть усі поля", "Fill in all fields") });
  }
  try {
    const exists = await pool.query(
      "SELECT id FROM users WHERE email = $1",
[email.toLowerCase()]
    );
    if (exists.rows.length) {
      return res.status(409).json({
        message: tr(req, "Email вже використовується",
"Email already in use")
      });
    }
    const passwordHash = await bcrypt.hash(password,
10);
    const result = await pool.query(
      "INSERT INTO users (full_name, email,
password_hash) VALUES ($1,$2,$3) RETURNING id,
email",
      [fullName.trim(), email.toLowerCase(),
passwordHash]
    );
    const user = result.rows[0];
    const accessToken = signAccessToken({ userId:
user.id });
    const refreshToken = signRefreshToken({ userId:
user.id });
    await pool.query(
      "INSERT INTO refresh_tokens (user_id, token_hash)
VALUES ($1, $2)",
      [user.id, hashToken(refreshToken)]
    );
    res.status(201).json({ accessToken, refreshToken });
  } catch (e) {
    res.status(500).json({ message: e.message });
  }
});

app.post("/api/auth/login", async (req, res) => {
  const { email, password } = req.body;
  try {
    const result = await pool.query(
      "SELECT * FROM users WHERE email = $1",
[email.toLowerCase()]
    );
    const user = result.rows[0];
    if (!user || !(await bcrypt.compare(password,
user.password_hash))) {
      return res.status(401).json({
        message: tr(req, "Невірний email або пароль",
"Invalid email or password")
      });
    }
    const accessToken = signAccessToken({ userId:
user.id });
    const refreshToken = signRefreshToken({ userId:
user.id });
    await pool.query(
      "INSERT INTO refresh_tokens (user_id, token_hash)
VALUES ($1, $2)",
      [user.id, hashToken(refreshToken)]
    );
    res.json({ accessToken, refreshToken });
  } catch (e) {
    res.status(500).json({ message: e.message });
  }
});
```

Модуль автентифікації (auth.js)

```
import jwt from "jsonwebtoken";
import { createHash } from "crypto";

export function signAccessToken(payload) {
  return jwt.sign(payload, process.env.JWT_SECRET, {
expiresIn: "15m" });
}

export function signRefreshToken(payload) {
  return jwt.sign(payload, process.env.JWT_SECRET, {
expiresIn: "30d" });
}

export function hashToken(token) {
  return
createHash("sha256").update(token).digest("hex");
}

export function verifyRefreshToken(token) {
  return jwt.verify(token, process.env.JWT_SECRET);
}

export function authRequired(req, res, next) {
  const authHeader = req.headers.authorization;
  if (!authHeader || !authHeader.startsWith("Bearer ")) {
    return res.status(401).json({ message: "Unauthorized"
});
  }
  const token = authHeader.split(" ")[1];
  try {
    req.user = jwt.verify(token,
process.env.JWT_SECRET);
    next();
```

```
  } catch {
    res.status(401).json({ message: "Invalid or expired
token" });
  }
}
```

CRUD операції над витратами (server.js)

```
app.get("/api/expenses", authRequired, async (req, res)
=> {
  const { month, category } = req.query;
  let query = "SELECT * FROM expenses WHERE
user_id = $1";
  const params = [req.user.userId];
  if (month) {
    params.push(month);
    query += ` AND TO_CHAR(expense_date, 'YYYY-
MM') = $${params.length}`;
  }
  if (category) {
    params.push(category);
    query += ` AND category = $${params.length}`;
  }
  query += " ORDER BY expense_date DESC,
created_at DESC";
  const result = await pool.query(query, params);
  res.json(result.rows);
});

app.post("/api/expenses", authRequired, async (req, res)
=> {
  const { title, amount, category, expenseDate, note } =
req.body;
  const result = await pool.query(
    `INSERT INTO expenses (user_id, title, amount,
category, expense_date, note)
     VALUES ($1,$2,$3,$4,$5,$6) RETURNING *`,
    [req.user.userId, title, amount, category, expenseDate,
note || null]
  );
  res.status(201).json(result.rows[0]);
});

app.put("/api/expenses/:id", authRequired, async (req,
res) => {
  const { title, amount, category, expenseDate, note } =
req.body;
  const result = await pool.query(
    `UPDATE expenses SET title=$1, amount=$2,
category=$3,
     expense_date=$4, note=$5
     WHERE id=$6 AND user_id=$7 RETURNING *`,
    [title, amount, category, expenseDate, note || null,
     req.params.id, req.user.userId]
  );
  if (!result.rows.length) return res.status(404).json({
message: "Not found" });
  res.json(result.rows[0]);
});

app.delete("/api/expenses/:id", authRequired, async (req,
res) => {
  await pool.query(
    "DELETE FROM expenses WHERE id=$1 AND
user_id=$2",
    [req.params.id, req.user.userId]
  );
  res.status(204).end();
});
```

Статистика та аналітика (server.js)

```
app.get("/api/stats/summary", authRequired, async (req,
res) => {
  const { month, category } = req.query;
  let query = `SELECT COUNT(*) AS total_records,
            COALESCE(SUM(amount), 0) AS
total_amount
            FROM expenses WHERE user_id = $1`;
  const params = [req.user.userId];
  if (month) {
    params.push(month);
    query += ` AND TO_CHAR(expense_date, 'YYYY-
MM') = $${params.length}`;
  }
  if (category) {
    params.push(category);
    query += ` AND category = $${params.length}`;
  }
  const result = await pool.query(query, params);
  res.json(result.rows[0]);
});

app.get("/api/stats/monthly", authRequired, async (req,
res) => {
  const result = await pool.query(
    `SELECT TO_CHAR(expense_date, 'YYYY-MM')
AS month,
          COALESCE(SUM(amount), 0) AS total
     FROM expenses WHERE user_id = $1
     GROUP BY month ORDER BY month`,
    [req.user.userId]
  );
  res.json(result.rows);
});

app.get("/api/stats/category-breakdown", authRequired,
async (req, res) => {
  const { month } = req.query;
  let query = `SELECT category,
            COALESCE(SUM(amount), 0) AS total
            FROM expenses WHERE user_id = $1`;
  const params = [req.user.userId];
  if (month) {
    params.push(month);
    query += ` AND TO_CHAR(expense_date, 'YYYY-
MM') = $${params.length}`;
  }
  query += " GROUP BY category ORDER BY total
DESC";
```

```
  const result = await pool.query(query, params);
  res.json(result.rows);
});
```

Бюджети та алерти (server.js)

```
app.post("/api/budgets", authRequired, async (req, res)
=> {
  const { category, monthlyLimit } = req.body;
  const result = await pool.query(
    `INSERT INTO budgets (user_id, category,
monthly_limit)
     VALUES ($1, $2, $3)
     ON CONFLICT (user_id, category)
     DO UPDATE SET monthly_limit =
EXCLUDED.monthly_limit,
     updated_at = NOW() RETURNING *`,
    [req.user.userId, category, monthlyLimit]
  );
  res.json(result.rows[0]);
});

app.get("/api/budgets/alerts", authRequired, async (req,
res) => {
  const { month } = req.query;
  const currentMonth = month || new
Date().toISOString().slice(0, 7);
  const result = await pool.query(
    `SELECT b.category, b.monthly_limit,
          COALESCE(SUM(e.amount), 0) AS spent,
          ROUND(COALESCE(SUM(e.amount), 0) /
b.monthly_limit * 100, 2) AS percent
     FROM budgets b
     LEFT JOIN expenses e
       ON e.user_id = b.user_id
       AND e.category = b.category
       AND TO_CHAR(e.expense_date, 'YYYY-MM') =
$2
     WHERE b.user_id = $1
     GROUP BY b.category, b.monthly_limit`,
    [req.user.userId, currentMonth]
  );
  const alerts = result.rows.map((row) => ({
    category: row.category,
    monthly_limit: row.monthly_limit,
    spent: row.spent,
    percent: row.percent,
    status: row.percent >= 100 ? "over" : row.percent >=
80 ? "near" : "ok"
  }));
  res.json(alerts);
});
```

Імпорт CSV з підтримкою пресетів (server.js)

```
const PRESETS = {
  monobank: { date: "Дата", amount: "Сума",
description: "Опис" },
  privatbank: { date: "Дата операції", amount: "Сума",
description: "Опис операції" },
  wise: { date: "TransferWise ID", amount: "Amount",
description: "Description" },
  generic: { date: "date", amount: "amount", description:
"title" }
};

function parseCsv(csvText, delimiter, preset) {
  const config = PRESETS[preset] || PRESETS.generic;
  const lines = csvText.split("\n").filter(Boolean);
  const headers = lines[0].split(delimiter).map((h) =>
h.trim().replace(/"/g, ""));
  const rows = [];
  let skipped = 0;
  lines.slice(1).forEach((line, idx) => {
    const cols = line.split(delimiter).map((c) =>
c.trim().replace(/"/g, ""));
    const obj = {};
    headers.forEach((h, i) => (obj[h] = cols[i] || ""));
    const rawAmount =
parseFloat(obj[config.amount]?.replace(",", ".") || "0");
    const amount = Math.abs(rawAmount);
    const date = obj[config.date];
    const title = obj[config.description] || "Імпорт";
    if (!date || isNaN(amount) || amount <= 0) {
skipped++; return; }
    rows.push({ rowNo: idx + 2, expenseDate:
date.slice(0, 10),
                title, amount, category: "Інше", note: "" });
  });
  return { rows, skipped };
}

app.post("/api/expenses/import-csv/commit",
authRequired, async (req, res) => {
  const { rows, preset, skipped } = req.body;
  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    const session = await client.query(
      `INSERT INTO import_sessions (user_id,
source_preset, imported_count, skipped_count)
       VALUES ($1, $2, $3, $4) RETURNING id`,
      [req.user.userId, preset, rows.length, skipped || 0]
    );
    const sessionId = session.rows[0].id;
    for (const row of rows) {
      await client.query(
        `INSERT INTO expenses
         (user_id, title, amount, category, expense_date,
note, import_session_id)
         VALUES ($1,$2,$3,$4,$5,$6,$7)`,
        [req.user.userId, row.title, row.amount,
row.category,
         row.expenseDate, row.note || null, sessionId]
      );
    }
    await client.query("COMMIT");
    res.json({ imported: rows.length, skipped: skipped || 0,
sessionId });
  } catch (e) {
```

```
    await client.query("ROLLBACK");
    res.status(500).json({ message: e.message });
  } finally {
    client.release();
  }
});
```

HTTP-клієнт із автооновленням токена (api.js)

```
const API_BASE =
(import.meta.env.VITE_API_BASE_URL ||
  "http://localhost:4000/api").replace(/\/$/, "");

let refreshPromise = null;

async function refreshSession() {
  const refreshToken =
localStorage.getItem("spendwise_refresh_token");
  if (!refreshToken) throw new Error("Session expired");
  const response = await
fetch(`${API_BASE}/auth/refresh`, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ refreshToken })
  });
  if (!response.ok) {
    localStorage.removeItem("spendwise_access_token");
    localStorage.removeItem("spendwise_refresh_token");
    throw new Error("Session expired");
  }
  const data = await response.json();
  localStorage.setItem("spendwise_access_token",
data.accessToken);
  localStorage.setItem("spendwise_refresh_token",
data.refreshToken);
  return data;
}

async function request(path, options = {}, retry = true) {
  const token =
localStorage.getItem("spendwise_access_token") || "";
  const headers = {
    "Content-Type": "application/json",
    "Accept-Language":
localStorage.getItem("spendwise_lang") || "en",
    ...(token ? { Authorization: `Bearer ${token}` } : {}),
    ...(options.headers || {})
  };
  const response = await fetch(`${API_BASE}${path}`,
{ ...options, headers });
  if (!response.ok) {
    if (response.status === 401 && retry) {
      if (!refreshPromise) {
        refreshPromise = refreshSession().finally(() => {
          refreshPromise = null;
        });
      }
      await refreshPromise;
      return request(path, options, false);
    }
    const data = await response.json().catch(() => ({}));
    throw new Error(data.message || "Request failed");
  }
  return response.json();
}
```

Завантаження даних дашборду (App.jsx)

```
async function loadData() {
  if (!isLoggedIn) return;
  setLoading(true);
  setError("");
  try {
    const [
      expenseData, summaryData, monthlyData,
      categoryData, analysisData, budgetData, alertsData
    ] = await Promise.all([
      api.getExpenses(filters),
      api.getSummary(filters),
      api.getMonthly(),
      api.getCategoryBreakdown({ month: filters.month }),
      api.getAnalysis({ month: filters.month }),
      api.getBudgets(),
      api.getBudgetAlerts({ month: filters.month })
    ]);
    setExpenses(expenseData);
    setSummary(summaryData);
    setMonthly(monthlyData);
    setCategoryBreakdown(categoryData);
    setAnalysis(analysisData);
    setBudgets(budgetData);
    setBudgetAlerts(alertsData);
    const history = await api.getImportHistory();
    setImportHistory(history);
  } catch (e) {
    setError(e.message);
    if (!api.getSession().accessToken) setUser(null);
  } finally {
    setLoading(false);
  }
}

useEffect(() => {
  loadData();
}, [isLoggedIn, filters.month, filters.category]);
```

Компоненти графіків (App.jsx)

```
<section className="charts-grid">
  <section className="card">
    <h2>{t.monthlyStats}</h2>
    <div className="chart-wrap">
      <ResponsiveContainer width="100%"
height={320}>
        <BarChart data={monthly}
barCategoryGap="20%">
          <CartesianGrid strokeDasharray="3 3"
            stroke="#e2e8f0" vertical={false} />
          <XAxis dataKey="month"
            tick={{ fontSize: 12, fill: "#64748b" }}
```

```
          axisLine={{ stroke: "#e2e8f0" }}
tickLine={false} />
        <YAxis tick={{ fontSize: 12, fill: "#64748b" }}
          axisLine={false} tickLine={false} />
        <Tooltip contentStyle={{
          borderRadius: 10,
          border: "1px solid #e2e8f0",
          boxShadow: "0 4px 12px rgba(0,0,0,.08)"
        }} />
        <Bar dataKey="total" fill="#3b82f6" radius={[6,
6, 0, 0]} />
      </BarChart>
    </ResponsiveContainer>
  </div>
 </section>

 <section className="card">
  <h2>{t.pieStats}</h2>
  <div className="chart-wrap">
   <ResponsiveContainer width="100%"
height={320}>
     <PieChart>
      <Pie
       data={categoryBreakdown}
       dataKey="total"
       nameKey="category"
       outerRadius={110}
       innerRadius={50}
       paddingAngle={3}
       label={({ category }) =>
categoryLabel(category)}
      >
       {categoryBreakdown.map((entry, index) => (
        <Cell
         key={entry.category}
         fill={pieColors[index % pieColors.length]}
        />
       ))}
      </Pie>
      <Tooltip contentStyle={{
       borderRadius: 10,
       border: "1px solid #e2e8f0",
       boxShadow: "0 4px 12px rgba(0,0,0,.08)"
      }} />
      <Legend />
     </PieChart>
    </ResponsiveContainer>
  </div>
 </section>
</section>
```

Оновлення токена та вихід із системи (server.js)

```
app.post("/api/auth/refresh", async (req, res) => {
 const { refreshToken } = req.body;
 if (!refreshToken) {
  return res.status(401).json({
   message: tr(req, "Refresh-токен відсутній", "Refresh
token missing")
  });
 }
 try {
  const payload = verifyRefreshToken(refreshToken);
  const hashed = hashToken(refreshToken);
  const stored = await pool.query(
   "SELECT id FROM refresh_tokens WHERE
user_id=$1 AND token_hash=$2",
   [payload.userId, hashed]
  );
  if (!stored.rows.length) {
   return res.status(401).json({
    message: tr(req, "Сесія завершена", "Session
expired")
   });
  }
  await pool.query(
   "DELETE FROM refresh_tokens WHERE
user_id=$1 AND token_hash=$2",
   [payload.userId, hashed]
  );
  const newAccess = signAccessToken({ userId:
payload.userId });
  const newRefresh = signRefreshToken({ userId:
payload.userId });
  await pool.query(
   "INSERT INTO refresh_tokens (user_id, token_hash)
VALUES ($1,$2)",
   [payload.userId, hashToken(newRefresh)]
  );
  res.json({ accessToken: newAccess, refreshToken:
newRefresh });
 } catch {
  res.status(401).json({
   message: tr(req, "Недійсний токен", "Invalid token")
  });
 }
});

app.post("/api/auth/logout", authRequired, async (req,
res) => {
 const { refreshToken } = req.body;
 if (refreshToken) {
  await pool.query(
   "DELETE FROM refresh_tokens WHERE
user_id=$1 AND token_hash=$2",
   [req.user.userId, hashToken(refreshToken)]
  );
 }
 res.json({ ok: true });
});

app.get("/api/auth/me", authRequired, async (req, res) =>
{
 const result = await pool.query(
  "SELECT id, full_name, email, created_at FROM
users WHERE id=$1",
  [req.user.userId]
 );
 if (!result.rows.length) {
```

```
    return res.status(404).json({ message: "User not
found" });
  }
  res.json(result.rows[0]);
});
```

Відкат останнього імпорту (server.js)

```
app.post("/api/import/rollback-last", authRequired, async
(req, res) => {
  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    const sessionResult = await client.query(
      `SELECT id FROM import_sessions
       WHERE user_id = $1
       ORDER BY created_at DESC LIMIT 1`,
      [req.user.userId]
    );
    if (!sessionResult.rows.length) {
      await client.query("ROLLBACK");
      return res.status(404).json({
        message: tr(req, "Імпортів не знайдено", "No
imports found")
      });
    }
    const sessionId = sessionResult.rows[0].id;
    const deleted = await client.query(
      "DELETE FROM expenses WHERE
import_session_id=$1 RETURNING id",
      [sessionId]
    );
    await client.query(
      "DELETE FROM import_sessions WHERE id=$1",
      [sessionId]
    );
    await client.query("COMMIT");
    res.json({ removed: deleted.rows.length, sessionId });
  } catch (e) {
    await client.query("ROLLBACK");
    res.status(500).json({ message: e.message });
  } finally {
    client.release();
  }
});

app.get("/api/import/history", authRequired, async (req,
res) => {
  const result = await pool.query(
    `SELECT id, source_preset, imported_count,
            skipped_count, created_at
     FROM import_sessions
     WHERE user_id = $1
     ORDER BY created_at DESC
     LIMIT 20`,
    [req.user.userId]
  );
  res.json(result.rows);
});

app.get("/api/import/presets", authRequired, async (req,
res) => {
  const presets = [
    { key: "generic",    label: "Generic CSV" },
    { key: "monobank",   label: "Monobank" },
    { key: "privatbank", label: "PrivatBank" },
    { key: "wise",       label: "Wise" }
  ];
  res.json(presets);
});
```

Модуль автоматичного аналізу витрат (analysis.js)

```
export function buildAutoAnalysis(categoryRows,
monthlyRows, lang = "en") {
  const uk = lang === "uk";

  if (!categoryRows.length) {
    return {
      insight: uk
        ? "Витрат за обраний період не знайдено."
        : "No expenses found for the selected period.",
      tips: []
    };
  }

  const total = categoryRows.reduce((s, r) => s +
Number(r.total), 0);
  const top = [...categoryRows].sort((a, b) => b.total -
a.total)[0];
  const topPercent = ((Number(top.total) / total) *
100).toFixed(1);

  const insight = uk
    ? `Найбільша частка витрат припадає на категорію
«${top.category}» — ${topPercent}% від загальної
суми.`
    : `The largest share of expenses is in the
«${top.category}» category — ${topPercent}% of the
total.`;

  const tips = [];

  if (Number(topPercent) > 40) {
    tips.push(
      uk
        ? `Витрати на «${top.category}» перевищують
40% бюджету. Розгляньте можливість їх скорочення.`
        : `Expenses on «${top.category}» exceed 40% of
budget. Consider reducing them.`
    );
  }

  if (monthlyRows.length >= 2) {
    const last =
Number(monthlyRows[monthlyRows.length - 1].total);
    const prev =
Number(monthlyRows[monthlyRows.length - 2].total);
    const diff = (((last - prev) / prev) * 100).toFixed(1);
    if (last > prev) {
```

```
      tips.push(
        uk
          ? `Витрати зросли на ${diff}% порівняно з попереднім місяцем.`
          : `Expenses increased by ${diff}% compared to the previous month.`
      );
    } else {
      tips.push(
        uk
          ? `Витрати знизились на ${Math.abs(diff)}% порівняно з попереднім місяцем. Гарна робота!`
          : `Expenses decreased by ${Math.abs(diff)}% compared to the previous month. Great job!`
      );
    }
  }

  if (categoryRows.length >= 3) {
    tips.push(
      uk
        ? "Розподіліть бюджет по категоріях і встановіть місячні ліміти для кращого контролю."
        : "Distribute your budget across categories and set monthly limits for better control."
    );
  }

  return { insight, tips };
}
```

Попередній перегляд CSV (server.js)

```
app.post("/api/expenses/import-csv/preview", authRequired, async (req, res) => {
  const { csvText, delimiter = ";", preset = "generic" } = req.body;
  if (!csvText) {
    return res.status(400).json({
      message: tr(req, "CSV-текст відсутній", "CSV text is missing")
    });
  }
  try {
    const { rows, skipped } = parseCsv(csvText, delimiter, preset);
    res.json({ rows, skipped });
  } catch (e) {
    res.status(400).json({ message: e.message });
  }
});

function parseCsv(csvText, delimiter, preset) {
  const config = PRESETS[preset] || PRESETS.generic;
  const lines = csvText
    .split("\n")
    .map((l) => l.trim())
    .filter(Boolean);
  if (lines.length < 2) return { rows: [], skipped: 0 };

  const headers = lines[0]
    .split(delimiter)
    .map((h) => h.replace(/"/g, "").trim());

  const rows = [];
  let skipped = 0;

  lines.slice(1).forEach((line, idx) => {
    const cols = line.split(delimiter).map((c) => c.replace(/"/g, "").trim());
    const obj = {};
    headers.forEach((h, i) => { obj[h] = cols[i] || ""; });

    const rawAmount = parseFloat(
      (obj[config.amount] || "0").replace(",", ".")
    );
    const amount = Math.abs(rawAmount);
    const rawDate = (obj[config.date] || "").slice(0, 10);
    const title = obj[config.description] || "Import";

    const isValidDate = /^\d{4}-\d{2}-\d{2}$/.test(rawDate);
    if (!isValidDate || isNaN(amount) || amount <= 0) {
      skipped++;
      return;
    }
    rows.push({
      rowNo: idx + 2,
      expenseDate: rawDate,
      title: title.slice(0, 255),
      amount,
      category: "Інше",
      note: ""
    });
  });
  return { rows, skipped };
}
```

Форма авторизації (App.jsx)

```
{publicPage === "auth" && showAuthForm ? (
  <section id="auth-section" className="card auth-card">
    <div className="auth-card-head">
      <h2>
        {authMode === "login" ? t.authLoginTitle : t.authRegisterTitle}
      </h2>
      <button
        type="button"
        className="link-btn"
        onClick={() => setShowAuthForm(false)}
      >
        {t.close}
      </button>
    </div>

    {error ? <div className="error">{error}</div> : null}
```

```
   <form className="auth-form"
onSubmit={handleAuthSubmit}>
     {authMode === "register" ? (
      <input
       required
       placeholder={t.fullName}
       value={authForm.fullName}
       onChange={(e) =>
        setAuthForm((prev) => ({ ...prev, fullName:
e.target.value }))
       }
      />
     ) : null}
     <input
      required
      type="email"
      placeholder="Email"
      value={authForm.email}
      onChange={(e) =>
       setAuthForm((prev) => ({ ...prev, email:
e.target.value }))
      }
     />
     <input
      required
      type="password"
      placeholder={t.password}
      value={authForm.password}
      onChange={(e) =>
       setAuthForm((prev) => ({ ...prev, password:
e.target.value }))
      }
     />
     <button type="submit">
      {authMode === "login" ? t.login : t.register}
     </button>
   </form>

   <button
    type="button"
    className="link-btn"
    onClick={() => {
     setError("");
     setAuthMode((prev) => (prev === "login" ?
"register" : "login"));
    }}
   >
    {authMode === "login" ? t.noAccount :
t.hasAccount}
   </button>
  </section>
) : null}
```

Форма додавання та редагування витрати (App.jsx)

```
<section className="card">
  <h2>{t.addExpense}</h2>
  <form className="form-grid"
onSubmit={handleSubmit}>
   <input
    required
    placeholder={t.expenseTitle}
    value={form.title}
    onChange={(e) =>
     setForm((prev) => ({ ...prev, title: e.target.value }))
    }
   />
   <input
    required
    type="number"
    min="0.01"
    step="0.01"
    placeholder={t.amount}
    value={form.amount}
    onChange={(e) =>
     setForm((prev) => ({ ...prev, amount: e.target.value
}))
    }
   />
   <select
    value={form.category}
    onChange={(e) =>
     setForm((prev) => ({ ...prev, category: e.target.value
}))
    }
   >
    {categories.map((item) => (
     <option key={item.value} value={item.value}>
      {item[lang]}
     </option>
    ))}
   </select>
   <input
    required
    type="date"
    value={form.expenseDate}
    onChange={(e) =>
     setForm((prev) => ({ ...prev, expenseDate:
e.target.value }))
    }
   />
   <input
    placeholder={t.noteOptional}
    value={form.note}
    onChange={(e) =>
     setForm((prev) => ({ ...prev, note: e.target.value }))
    }
   />
   <button type="submit">{t.save}</button>
  </form>
</section>

{editExpense ? (
  <section className="card">
   <h2>{t.editExpense}</h2>
   <form className="form-grid"
onSubmit={handleUpdateExpense}>
    <input
     required
     value={editExpense.title}
```

```
        onChange={(e) =>
          setEditExpense((prev) => ({ ...prev, title:
e.target.value }))
        }
      />
      <input
        required
        type="number"
        min="0.01"
        step="0.01"
        value={editExpense.amount}
        onChange={(e) =>
          setEditExpense((prev) => ({ ...prev, amount:
e.target.value }))
        }
      />
      <select
        value={editExpense.category}
        onChange={(e) =>
          setEditExpense((prev) => ({ ...prev, category:
e.target.value }))
        }
      >
        {categories.map((item) => (
          <option key={item.value} value={item.value}>
            {item[lang]}
          </option>
        ))}
      </select>
      <input
        required
        type="date"
        value={editExpense.expenseDate}
        onChange={(e) =>
          setEditExpense((prev) => ({
            ...prev,
            expenseDate: e.target.value
          }))
        }
      />
      <input
        value={editExpense.note || ""}
        onChange={(e) =>
          setEditExpense((prev) => ({ ...prev, note:
e.target.value }))
        }
      />
      <div className="inline-actions">
        <button type="submit">{t.update}</button>
        <button
          type="button"
          className="secondary-btn"
          onClick={() => setEditExpense(null)}
        >
          {t.cancel}
        </button>
      </div>
    </form>
  </section>
) : null}
```