Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук
(повна назва)

Кафедра Програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система для управління особистими фінансами
(тема)

Виконав:
Студент 4 курсу, групи ПЗПІ-20-4

Пашкевич К. О.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник ст. викл. Олійник О. В.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри _____ З.В. Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 –Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«_____» _____20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Пашкевичу Кирилу Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи: «Програмна система для управління особистими фінансами» затверджена наказом університету від 20 травня 2024р. № 471Ст

2. Термін подання студентом роботи до екзаменаційної комісії 8 червня 2024 р.

3. Вихідні дані до роботи *В програмній реалізації планується створення системи для управління особистими фінансами. Для розробки серверної частини буде використано мову C# та фреймворк EntityFramework. Для забезпечення взаємодії між клієнтською чстиною та сервером використовуватимуться мова JavaScript та бібліотека React.*

4. Перелік питань, що потрібно опрацювати в роботі: *вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, перелік джерел посилань, додатки.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналіз предметної галузі	25.04.2024	*виконано*
2	Створення специфікації ПЗ	30.04.2024	*виконано*
3	Проєктування ПЗ	15.05.2024	*виконано*
4	Розробка та тестування ПЗ	20.05.2024	*виконано*
5	Написання пояснювальної записки	30.05.2024	*виконано*
6	Підготовка до перевірки на плагіат та нормоконтроль	01.06.2024	*виконано*
7	Оцінка пояснювальної записки керівником та рецензентом	05.06.2024	*виконано*
8	Подання роботи до електронного архіву	05.06.2024	*виконано*
9	Попередній захист	07.06.2024	*виконано*
10	Захист кваліфікаційної роботи	10.06.2024	*виконано*

Дата видачі завдання « 10 » квітня 2024 р.

Студент _____ Пашкевич К. О.
(підпис)

Керівник роботи _____ ст. викл. Олійник О. В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи бакалавра: 64 с., 20 рис., 3 таблиці, 10 джерел, 2 додатки.

БАЗА ДАНИХ, ВЕБ-ДОДАТОК, МОНІТОРИНГ, ВІДСТЕЖУВАННЯ, ОБЛІК, ТРАНЗАКЦІЇ, ФІНАНСИ, C# WEB API, REACT, SWAGGER, MSSQL SERVER

Об’єкт розробки – програмна система для управління особистими фінансами, до якої входить відстежування витрат та доходів, можливість планування бюджету на майбутнє, та аналіз розходів.

Об’єктами дослідження є автоматизація процесів відстежування та додавання нових транзакцій, та аналіз всіх отриманих даних всередині системи.

Методи вирішення завдань – аналіз та моделювання предметної області, концептуальне моделювання, UML-моделювання, використання СКБД MSSQL Server, розробка серверної частини системи з використанням фреймворку Entity Framework.

Результатами роботи є функціонуюча система для управління особистими фінансами.

DATABASE, WEB APPLICATION, MONITORING, TRACKING, ACCOUNTING, TRANSACTIONS, FINANCES, C# WEB API, REACT, SWAGGER, MSSQL SERVER

The object of development is a software system for managing personal finances, which includes tracking of expenses and income, the ability to plan a budget for the future, and analysis of expenses.

The objects of the study are the automation of the processes of tracking and adding new transactions, and the analysis of all received data within the system.

Methods of solving tasks - analysis and modeling of the subject area, conceptual modeling, UML modeling, use of MSSQL Server DBMS, development of the server part of the system using the Entity Framework.

The results of the work are a functioning system for managing personal finances.

Я, Пашкевич Кирило Олександрович, студент гр. ПЗПІ-20-4, здобувач вищої освіти на першому (бакалаврському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Програмна система для управління особистими фінансами», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Усі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови до допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

У сучасному світі, що постійно розвивається, ефективне управління особистими фінансами стає ключовим фактором успішного фінансового планування та забезпечення стабільності. Зростання складності фінансових операцій, динамічні зміни у фінансовому середовищі та постійне розширення ринку фінансових продуктів роблять управління власними фінансами для людей складним завданням.

Ця робота має на меті розробку програмної системи, яка допоможе людям краще управляти своїми особистими фінансами. Система має надати користувачам зручний та ефективний інструмент для відстеження доходів та витрат, планування бюджету, а також аналізу їх фінансового стану.

Для досягнення цієї мети буде проведено аналіз сучасних підходів до управління особистими фінансами та існуючих програмних рішень на ринку. На основі цього аналізу буде розроблена багаторівнева архітектура системи, сплановано весь необхідний функціонал, включаючи інтерфейс користувача, базу даних та алгоритми обробки фінансової інформації.

Розроблена програмна система стане цінним інструментом для людей, які прагнуть покращити свою фінансову грамотність та забезпечити стабільність свого фінансового стану.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної галузі

1.1.1 Дослідження актуальності системи

Сучасне суспільство динамічно змінюється, що ставить нові виклики перед людьми в питаннях управління особистими фінансами. Зростання складності фінансових операцій, стрімкий розвиток фінансового середовища та розширення арсеналу фінансових інструментів роблять процес планування та контролю власних коштів все більш складним.

Для багатьох людей відстеження та аналіз особистих фінансів стає все більш трудомістким завданням. Досягнення фінансової стабільності та ефективне використання наявних ресурсів потребує доступу до сучасних інструментів, які дозволяють зручно відстежувати та аналізувати фінансові потоки, а також розробляти та реалізовувати чіткі стратегії управління власними коштами [1].

Розробка та впровадження такої програмної системи може бути корисною не лише для приватних осіб, але й для підприємців та фінансових консультантів. Завдяки їй, всі ці категорії користувачів зможуть отримати ефективний інструмент для налагодження та оптимізації управління своїми фінансовими ресурсами.

1.1.2 Аналіз ринку

Перед розробкою власної програмної системи для управління особистими фінансами важливо провести ретельний аналіз вже існуючих на ринку рішень. Сучасні продукти пропонують широкий спектр функціоналу та підходів до управління фінансами, від простих інструментів до складних систем аналітики.

Одним з найпопулярніших типів систем для управління фінансами є мобільні додатки. Вони надають користувачам можливість зручно відстежувати свої доходи та витрати, планувати бюджет та аналізувати фінансовий стан. Деякі з цих додатків також мають інтеграцію з банківськими рахунками та іншими фінансовими сервісами, що робить управління фінансами ще простішим.

Першим прикладом такого продукту є Monefy (див.рис.1.1). Цей хмарний додаток дозволяє користувачам легко та швидко реєструвати свої фінансові транзакції, без зайвих складнощів. Додатково, є можливість відстежувати та аналізувати витрати за різними категоріями. Звітність за останній місяць допомагає краще контролювати бюджет [2].

Рисунок 1.1 – Інтерфейс додатку Monefy

До недоліків можна віднести відсутність локалізованих категорій, що може зробити додаток менш зручним для користувачів з різних регіонів. Також немає можливості відстежувати кредитні зобов'язання та депозитні рахунки.

Другий продукт – Money Lover (див.рис.1.2). Цей додаток з інтуїтивно зрозумілим інтерфейсом дозволяє встановлювати регулярні транзакції та поповнення. За допомогою підписки користувачі можуть відстежувати інформацію про свої кредити та депозити.

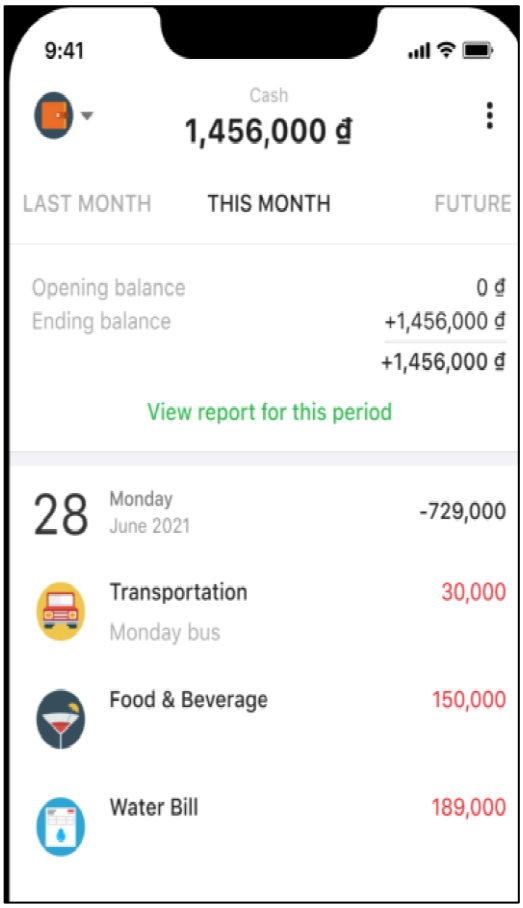


Рисунок 1.2 – Інтерфейс додатку MoneyLover

Його недоліки – це обмежений функціонал для безкоштовних користувачів, і невелика кількість доступних гаманців.

Окрім мобільних додатків, існують також програмні системи для персональних комп'ютерів. Їх функціонал зазвичай включає в себе більш розширені інструменти для аналізу та планування фінансів, а також можливості спільного доступу до даних для родин або команд.

Прикладом такого типу системи є Moneygraph (див.рис.1.3). Цей продукт дозволяє користувачам відстежувати фінанси з декількох рахунків, що корисно для ведення обліку окремо для різних цілей або банківських рахунків. Moneygraph також пропонує візуальний аналіз фінансових даних за допомогою діаграм та графіків, що допомагає краще зрозуміти структуру та динаміку витрат та доходів.

Основним недоліком є незручна навігація в додатку, через що деякі функції буває складно знайти.

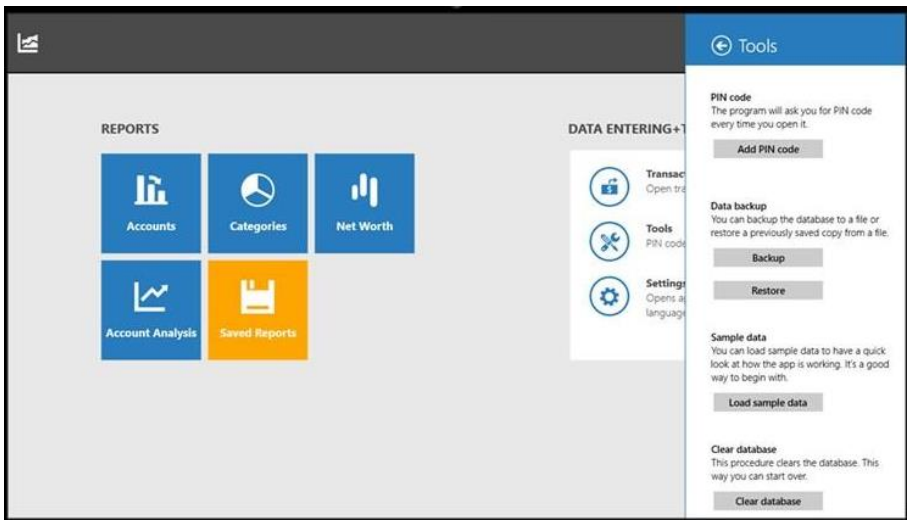


Рисунок 1.3 – Інтерфейс програми Moneygraph

На основі проведеного аналізу можна визначити, які компоненти є найбільш необхідними для власної системи. Огляд та порівняння аналогів представлені в таблиці 1.1.

Таблиця 1.1 – Порівняння нашої системи з аналогами

Аналізований показник	SmartMoney (наш продукт)	Monefy	Money Lover	Moneygraph
Функціонал	8	8	7	7
Інтерфейс	9	7	9	6
Безпека	7	7	7	8
Доступність	5	4	6	6
Інтеграція	8	6	5	6
Аналітика	6	4	7	9
Гнучкість	7	7	8	6
Разом	**50**	**43**	**49**	**48**

Після проведення аналізу, стали зрозумілими потреби ринку та спланована концепція нашого продукту.

1.2 Виявлення та вирішення проблем

На підставі проведеного аналізу існуючих програмних рішень для управління особистими фінансами, було виявлено, що жодна з них не відповідає в

повній мірі нашим вимогам щодо функціональності та зручності використання. З огляду на це, прийнято рішення про розробку власної програмної системи, яка буде враховувати потреби та побажання цільової аудиторії, а також усувати недоліки існуючих аналогів.

Основні завдання, які необхідно вирішити при розробці власного продукту:

- розширення функціоналу: необхідно надавати користувачам можливість здійснювати широкий спектр операцій з управління особистими фінансами, забезпечуючи при цьому точний та детальний аналіз їх фінансового стану;
- вдосконалення інтерфейсу користувача: інтерфейс повинен бути зручним, інтуїтивно зрозумілим та привабливим для користувачів, щоб зробити роботу з системою максимально комфортною;
- персоналізація: користувачі повинні мати можливість налаштовувати систему відповідно до своїх індивідуальних потреб та вподобань;
- забезпечення безпеки: ми маємо гарантувати високий рівень захисту фінансових даних користувачів від несанкціонованого доступу та кіберзагроз;
- мультиплатформність: система має бути доступна для використання з будь-яких пристроїв (комп'ютери, смартфони, планшети) та забезпечувати синхронізацію даних між ними;
- розширені можливості аналітики: користувачам мають бути доступними потужні інструменти для аналізу фінансових даних та генерувати детальні звіти, що допоможуть їм краще розуміти свої витрати, доходи та планувати бюджет;
- ефективність: система повинна забезпечувати швидке та безперебійне виконання операцій, навіть на пристроях з обмеженими ресурсами.

Система повинна регулярно оновлюватися та вдосконалюватися, щоб відповідати актуальним потребам користувачів та ринку.

1.3 Постановка задачі

Ця робота присвячена розробці програмної системи для управління особистими фінансами. Система буде виконувати такі основні завдання:

- додавання та керування транзакціями: користувачі зможуть додавати нові транзакції, редагувати та видаляти існуючі, а також категоризувати їх за типами витрат або доходів;
- планування бюджету: система допоможе користувачам спланувати бюджет на місяць, встановити фінансові цілі та відстежувати їх виконання;
- формування звітів про витрати: програмна система генеруватиме детальні звіти про витрати, які допоможуть користувачам краще розуміти, куди йдуть їхні гроші, та ідентифікувати потенційні сфери економії.

Функціональні можливості програмної системи для управління особистими фінансами:

- збір та систематизація фінансових даних: система буде збирати інформацію про доходи та витрати користувача з різних джерел та автоматично систематизувати її;
- персоналізоване планування бюджету: на основі аналізу доходів та витрат користувача, система допоможе йому створити персоналізований бюджет, який буде відповідати його фінансовим можливостям та цілям;
- аналіз фінансових даних: програма буде проводити аналіз фінансових даних користувача, щоб виявити тенденції, визначити найбільш витратні категорії, оцінити фінансову стабільність та ризики;
- групування витрат за категоріями: користувачі зможуть групувати свої витрати за різними категоріями (харчування, транспорт, розваги, тощо), що допоможе їм краще контролювати свої витрати;

- сповіщення про планові платежі: система надсилатиме користувачам сповіщення про планові платежі (кредити, комунальні послуги), щоб допомогти їм уникнути прострочок;
- звітування: програмна система генеруватиме різноманітні звіти з фінансових даних користувача, включаючи графіки та діаграми для візуалізації та аналізу його фінансової ситуації;
- захист даних: буде гарантована безпека особистої інформації користувача шляхом застосування сучасних методів шифрування та захисту доступу до даних.

Для успішної розробки програмної системи необхідно реалізувати функціонал, який автоматизує основні фінансові процеси користувача. Також варто створити зручний та інтуїтивно зрозумілий інтерфейс користувача і забезпечити високий рівень безпеки даних.

2 ПЕРЕЛІК ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Постановка мети

В рамках цієї роботи буде розроблена програмна система для управління особистими фінансами, яка буде доступна для широкого кола користувачів. Для розробки системи будуть використовуватись такі технології, як мова програмування C#, WEB API, база даних MSSQL Server, Entity Framework Core, JavaScript та React.

Використання цих технологій дозволить створити масштабовану та гнучку систему зі зручним та інтуїтивно зрозумілим інтерфейсом. Програмний продукт буде використовувати сучасні методи захисту даних для забезпечення безпеки фінансової інформації користувачів.

2.2 Загальний опис системи

Розроблена система являє собою клієнт-серверний веб-додаток, який охоплює всі необхідні функціональні можливості для ефективного управління особистими фінансами.

Система підтримує інтеграцію з різними платіжними системами, що робить процес здійснення фінансових операцій максимально зручним для користувачів.

Серверна частина системи побудована на основі WEB API та Entity Framework Core. Цей тандем технологій забезпечує надійну та гнучку основу для роботи з даними, що зберігаються в реляційній базі даних MS SQL Server.

Клієнтська частина системи розроблена з використанням мови JavaScript та фреймворку React.js. Ці технології пропонують ряд переваг, які роблять їх оптимальним вибором для даного проекту. Наприклад, React.js дозволяє легко розширювати та модифікувати додаток з мінімальними зусиллями, що робить його стійким до змін та адаптивним до нових вимог. Також завдяки використанню віртуального DOM, система оптимізує швидкість рендерингу сторінок та зменшує навантаження на браузер, роблячи роботу з додатком плавною та комфортною [3].

2.3 Основний функціонал системи

Доступ до функціоналу системи диференціюється залежно від ролі користувача.

Система передбачає три основні ролі:

- Admin (Адміністратор) – володіє найширшим набором повноважень, включаючи доступ до всіх функцій системи, можливість керування користувачами, налаштування параметрів системи та адміністрування даних;
- User (Користувач) – має доступ до базового функціоналу системи, що дозволяє йому здійснювати основні операції з управління особистими фінансами;
- Premium User (Преміум-користувач) – володіє розширеними можливостями порівняно з базовим користувачем, що може включати доступ до додаткових звітів, аналітичних інструментів, персональних консультацій тощо.

Функціонал користувачів представлений у таблиці 2.1.

Таблиця 2.1 – Функціонал користувачів

Дія	**Опис**
Unauthorized User	
Зареєструватись	Створення нового профілю в системі
User	
Увійти	Вхід до вже існуючого акаунту за допомогою логіну та паролю
Відстеження доходів і витрат	Можливість вносити дані про доходи та витрати, класифікувати їх за категоріями
Створення бюджету	Можливість створення та відстеження виконання бюджету на певний період часу
Створення фінансових цілей	Можливість встановлення фінансових цілей та відстеження їхнього виконання
Premium User	
Генерація звітів	Виведення звітів про стан фінансів за певний період часу

Кінець таблиці 2.1

Розширений аналіз фінансів	Доступ до розширених інструментів для аналізу фінансових даних та виведення статистики
Персоналізовані поради	Отримання персоналізованих порад щодо оптимізації фінансового стану
Admin	
Керування користувачами	Можливість створення, блокування, видалення акаунтів користувачів
Моніторинг безпеки	Відслідковування та управління безпекою системи

Для всіх ролей, окрім незареєстрованих користувачів, доступний базовий функціонал системи.

2.4 Загальні обмеження

Програмна система має наступні обмеження:

- користувач повинен мати активний обліковий запис для доступу до системи;
- для коректної роботи системи необхідне наявність доступу до мережі Інтернет;
- система може підтримувати лише певні типи та версії веб-браузерів для забезпечення оптимальної сумісності та безпеки;
- користувачі повинні створювати паролі, які відповідають певним критеріям безпеки, наприклад, містять комбінацію букв, цифр та символів;
- різні типи користувачів можуть мати різний рівень доступу до функцій системи в залежності від їхніх прав.

2.5 Припущення та залежності

Відповідно до нашої системи, було враховано наступні припущення та залежності:

- робота системи передбачає наявність безпечного мережевого з'єднання для забезпечення безперебійної роботи та захисту фінансових даних користувачів;
- деякі функції або можливості системи можуть залежати від інтеграції з іншими сервісами або технологіями третіх сторін;
- робота системи може бути залежною від відповідності законодавству щодо обробки фінансових даних та конфіденційності інформації;
- система передбачає відповідність технічним вимогам до апаратного та програмного забезпечення для забезпечення її ефективної роботи.

3 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 UML-проєктування програмної системи

Під час розробки архітектури системи була створена діаграма прецедентів, яка відображає основний функціонал системи (див. рис. 3.1).

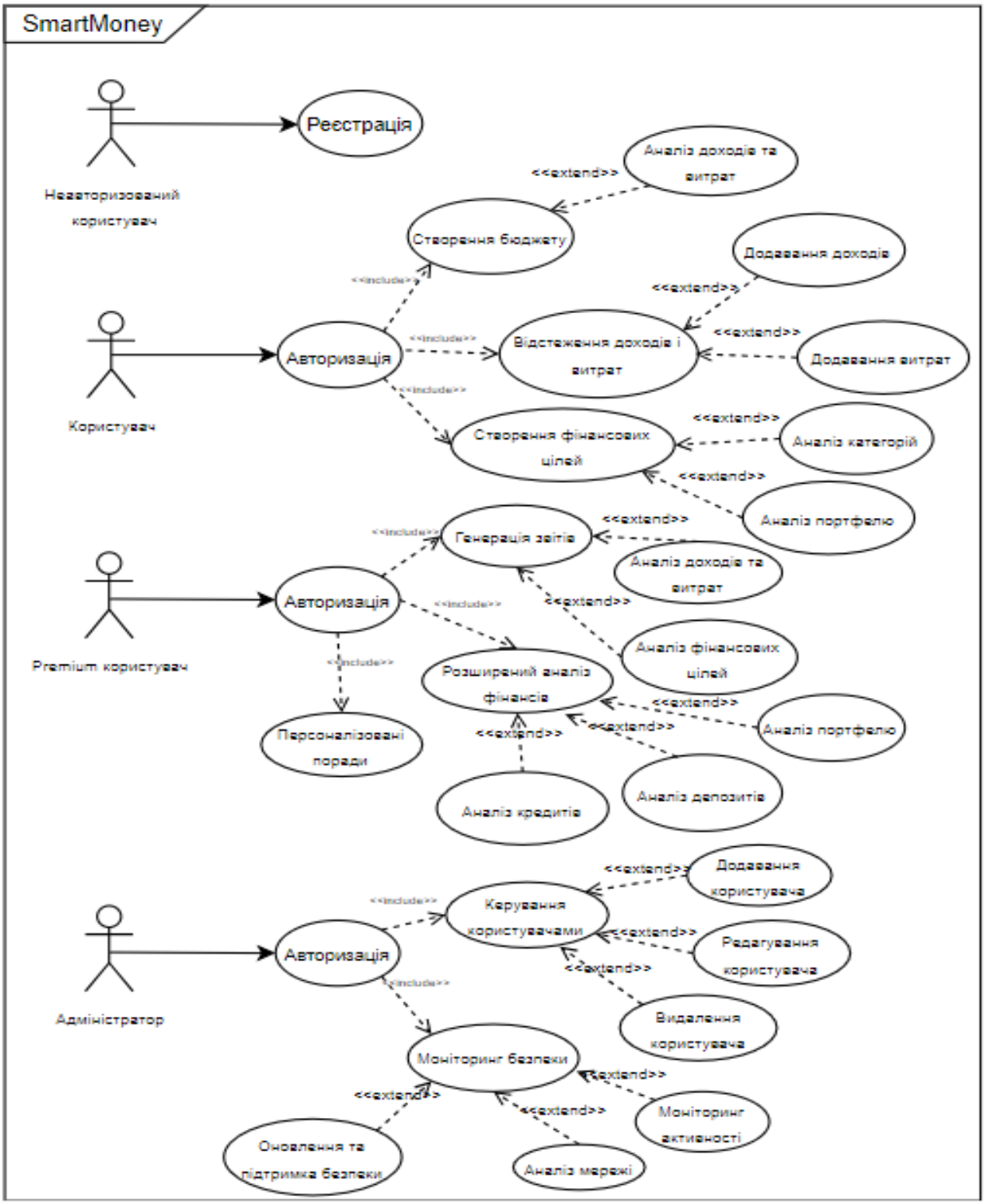


Рисунок 3.1 – Діаграма прецедентів для системи «Програмна система для управління особистими фінансами»

Система передбачає три основні типи користувачів.

Звичайні користувачі мають доступ до базового функціоналу, який дозволяє їм авторизуватися в системі, створити бюджет на певний період,
відстежувати свої доходи та витрати, створювати фінансові цілі, отримати аналіз своїх транзакцій для кращого розуміння фінансової ситуації.

Premium-користувачі володіють всіма можливостями звичайних користувачів, а також мають доступ до розширених функцій, таких як генерування детальних звітів про свої фінанси; отримання персоналізованих порад щодо управління особистими фінансами, ґрунтуючись на глибокому аналізі їхніх даних.

Адміністратори мають повний контроль над системою та можуть додавати, редагувати та видаляти акаунти користувачів; налаштовувати параметри системи; моніторити безпеку системи та вживати заходів у разі виявлення загроз.

При розробці веб-системи важливо ретельно проаналізувати всі можливі сценарії взаємодії користувачів з системою. Це включає в себе визначення всіх функціональних елементів, які можуть бути використані користувачами; розробку інтерфейсу користувача, який буде зручним та інтуїтивно зрозумілим для всіх типів користувачів; тестування системи, щоб переконатися, що вона працює коректно у всіх можливих ситуаціях [4].

На рисунку 3.2 показано діаграму стану користувача. Ця діаграма вказує на те, як користувач починає використовувати програму.

На початку користувачі переходять на сторінку авторизації, де вони мають можливість зареєструвати новий обліковий запис або увійти за допомогою вже наявного. У разі успішного введення необхідної інформації користувач буде перенаправлений на свій особистий профіль, де матиме доступ до різноманітних функцій та завдань:

- внести зміни до інформації в профілі;
- додати джерело доходів;
- додати витрати;
- створення бюджету;

– створення фінансових цілей.

Користувач може покинути програму на будь-якому стані, щоб завершити роботу.

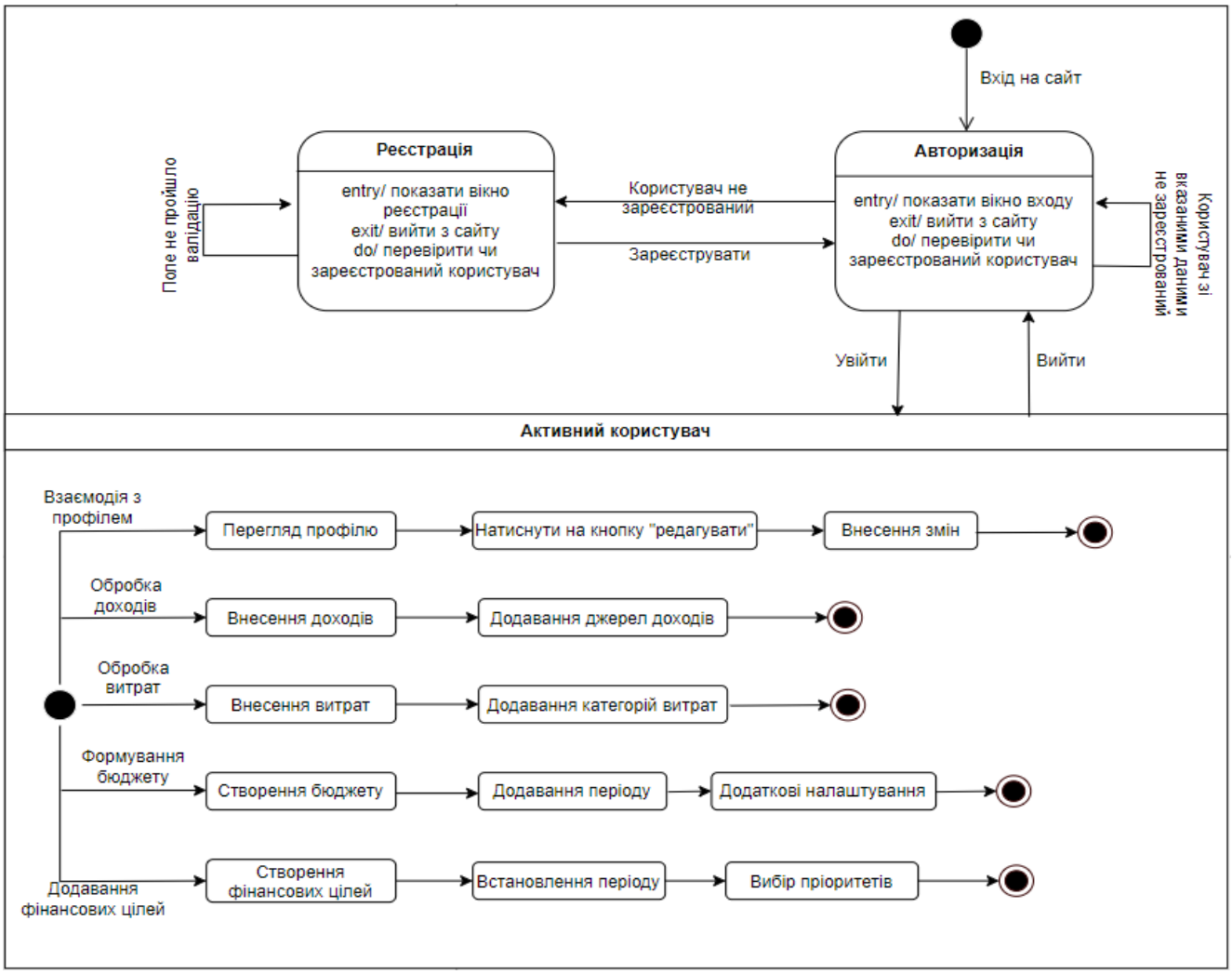


Рисунок 3.2 – Діаграма станів для системи «Програмна система для управління особистими фінансами»

3.2 Проєктування архітектури програмного забезпечення

Дуже важливим елементом для розробки архітектури програмної системи є створення діаграми бази даних (див. рис. 3.3).

Діаграму бази даних можна використовувати для створення та підтримки бази даних. Вона може допомогти розробнику зрозуміти структуру бази даних та зв'язки між різними сутностями.

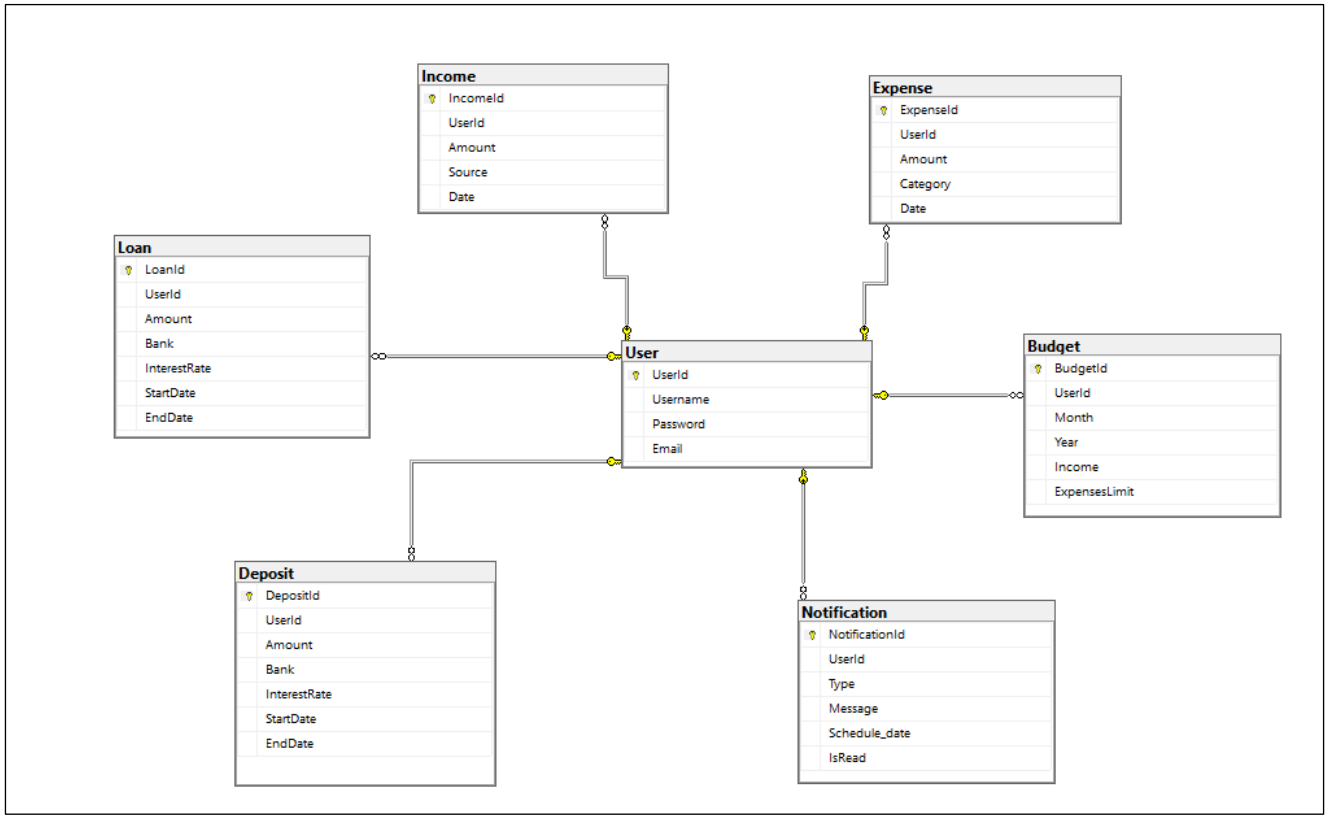


Рисунок 3.3 – Діаграма бази даних

Діаграма показує зв'язки між таблицями бази даних. Головною є таблиця «User», яка зберігає інформацію про користувачів системи, включаючи їхні імена, паролі, адреси електронної пошти та інші дані. Таблиця «Budget» зберігає інформацію про бюджети користувачів, включаючи суму, період та категорії. За інформацію про транзакції відповідає «Expense» та «Income».

Розроблена база даних включає наступні зв’язки:

– користувач може мати один або кілька банківських рахунків;

– на кожному банківському рахунку може бути багато транзакцій;

– кожна транзакція повинна бути категоризована за однією з категорій;

– користувач може мати один або кілька бюджетів;

– кожен бюджет може включати одну або кілька категорій.

Діаграма пакетів використовується для наглядного представлення структури програмної системи, що дозволяє зручно групувати компоненти за логічними критеріями. У веб-частині діаграми пакетів присутні два основних пакети - src та node_modules (див. рис. 3.4).

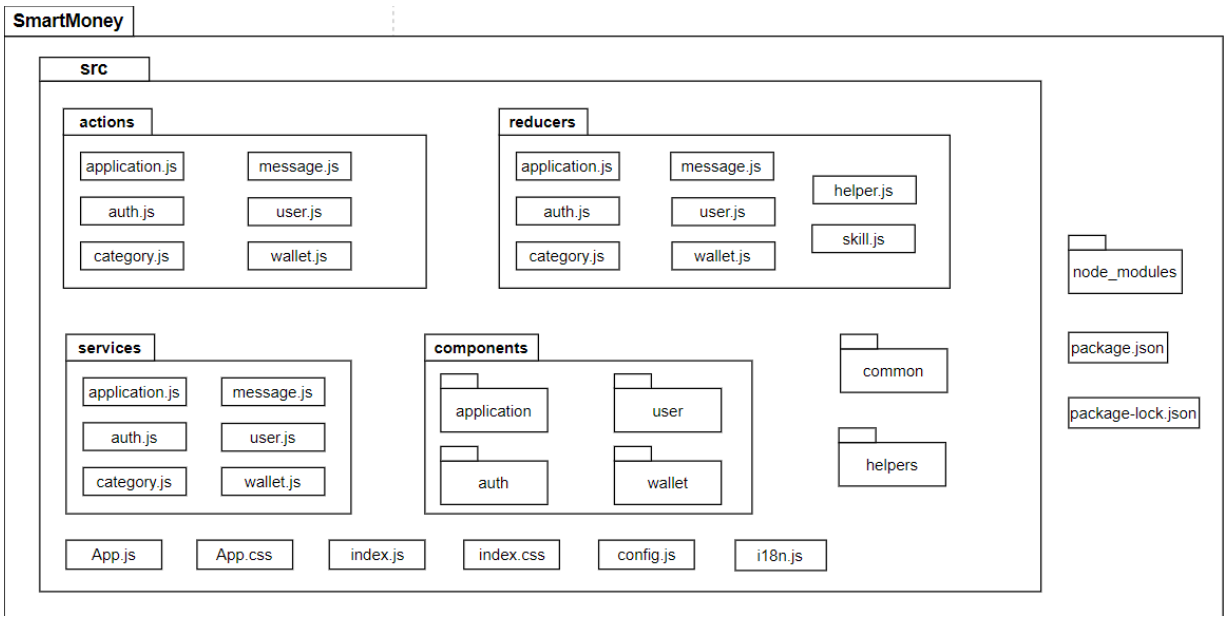


Рисунок 3.4 – Діаграма пакетів для системи «Програмна система для управління особистими фінансами»

Папка «***src***» є серцем проекту і містить всі основні файли з кодом.

Вона поділена на підпапки, кожна з яких відповідає за певну функціональність:

- «***actions***»: містить файли з набором функцій для різних аспектів додатку, таких як робота з даними, аутентифікація, категорії, повідомлення, користувачі, гаманець. Ці функції використовуються для оновлення стану програми;
- «***reducers***»: складається з файлів, що описують логіку обробки стану для різних аспектів додатку. Наприклад, файл auth.js відповідає за стан автентифікації, category.js - за стан категорій, і так далі;
- «***services***»: містяться файли з API-запитами до сервера. Ці файли використовуються для отримання та оновлення даних на сервері;
- «***components***»: містить компоненти React, які будують інтерфейс користувача. Ці компоненти можуть бути reutilizable (повторно використовуваними) та модульними;
- «***common***»: містить файли, які є загальними для багатьох компонентів, такі як стилі, константи, утиліти;

– «***helpers***»: містить файли з допоміжними функціями та інструментами, які використовуються в інших частинах проекту.

Файли інтерфейсу користувача:

– App.css, App.js – файли, що відповідають за візуальну частину головного контейнера програми. App.css містить стилі, а App.js використовується для динамічного створення та рендерингу HTML-елементів;
– index.css, index.js – файли, що є точкою входу для програми. Index.css містить стилі для всієї сторінки, а index.js використовується для завантаження та ініціалізації компонентів React.

Файл конфігурації – config.js. Цей файл використовується для зберігання налаштувань програми, таких як URL-адреси API, ключі авторизації та інші параметри, необхідні для роботи програми.

Файл для розширених можливостей – i18n.js. Цей файл містить налаштування для локалізації програми, дозволяючи їй відображати текст та інтерфейс користувача різними мовами.

Папка «node_modules» містить всі залежності, необхідні для роботи проекту. Завдяки їм розробникам не потрібно самостійно завантажувати та налаштовувати сторонні бібліотеки, що економить час та спрощує процес розробки.

Інші важливі файли:

– package.json – цей файл є ключовим для проекту, адже він містить опис проекту, список залежностей, необхідних для його роботи, та команди для їх встановлення та запуску;
– package-lock.json – у цьому файлі зберігається інформацію про точні версії залежностей, які були встановлені для проекту.

На рисунку 3.5 представлено діаграму розгортання. До основних елементів можна віднести веб-браузер, веб-сервер, сервер аутентифікації та сервер бази даних.

Алгоритм взаємодії компонентів:

– користувач вводить своє ім'я користувача та пароль у веб-браузері;

– веб-браузер надсилає HTTP-запит на веб-сервер;
– веб-сервер перенаправляє HTTP-запит на сервер аутентифікації;
– сервер аутентифікації перевіряє ім'я користувача та пароль користувача;
– якщо ім'я користувача та пароль правильні, сервер аутентифікації надсилає веб-серверу маркер автентифікації;
– веб-сервер перенаправляє HTTP-запит на сервер бази даних;
– сервер бази даних виконує запит і надсилає результат веб-серверу;
– веб-сервер надсилає відповідь користувачеві.

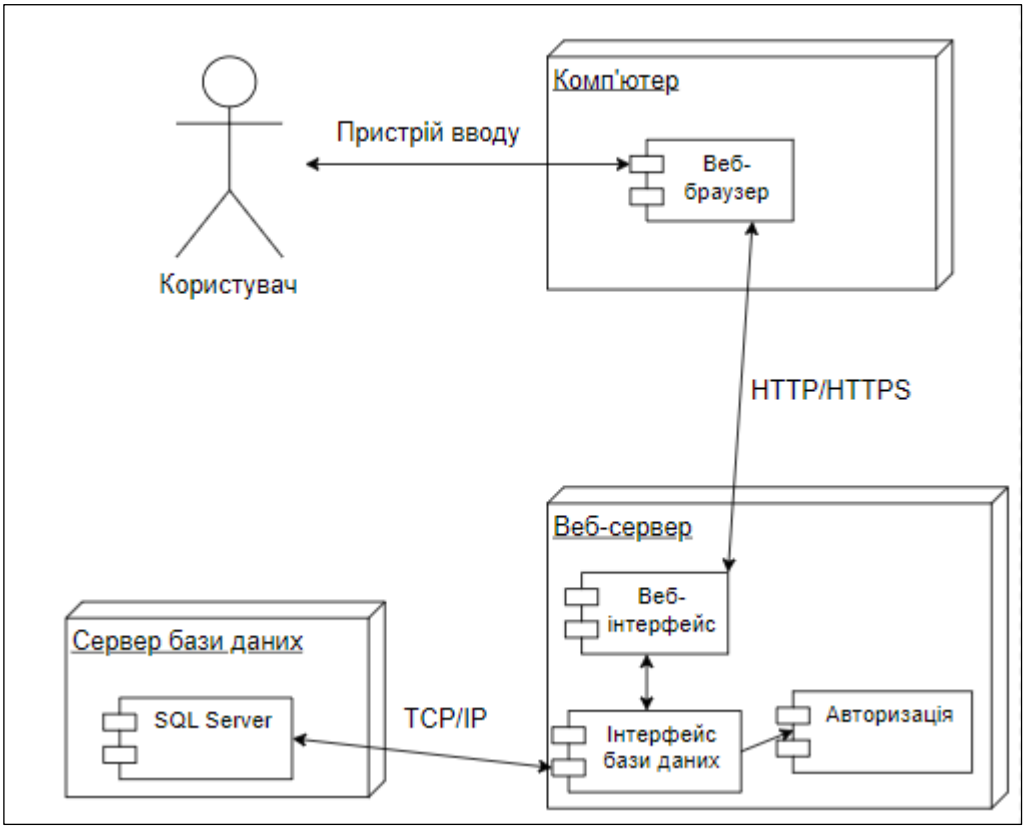


Рисунок 3.5 – Діаграма розгортання

Ця архітектура розгортання є гнучкою та масштабованою, що робить її хорошою основою для програмної системи управління особистими фінансами.

3.3 Проєктування компонентів системи

У процесі проєктування компонентів для системи для управління особистими фінансами, важливо врахувати деталі, що включають в себе структуру, функціональність та інтерфейси кожного компонента. Також

необхідно дізнатись потреби користувачів і забезпечити зручний інтерфейс для їхнього використання. Важливо дотримуватись принципів дизайну, такі як простота, доступність та ефективність, щоб забезпечити зручне використання системи для всіх користувачів. Кожен компонент буде проектуватися з урахуванням його ролі в системі та потреб користувачів, з метою забезпечення високої функціональності та зручного інтерфейсу.

Діаграма компонентів – це тип структурної діаграми, що використовується для візуалізації програмного забезпечення або систем, які складаються з взаємозалежних компонентів. Вона чітко показує, як різні частини системи взаємодіють між собою та як загалом організована система. Мета діаграми компонентів - продемонструвати архітектуру системи та підкреслити зв'язки між її компонентами. На ній кожен компонент зображується як блок, а зв'язки між ними позначаються стрілками або лініями. Рисунок 3.6 наводить приклад діаграми компонентів для нашої системи [5].

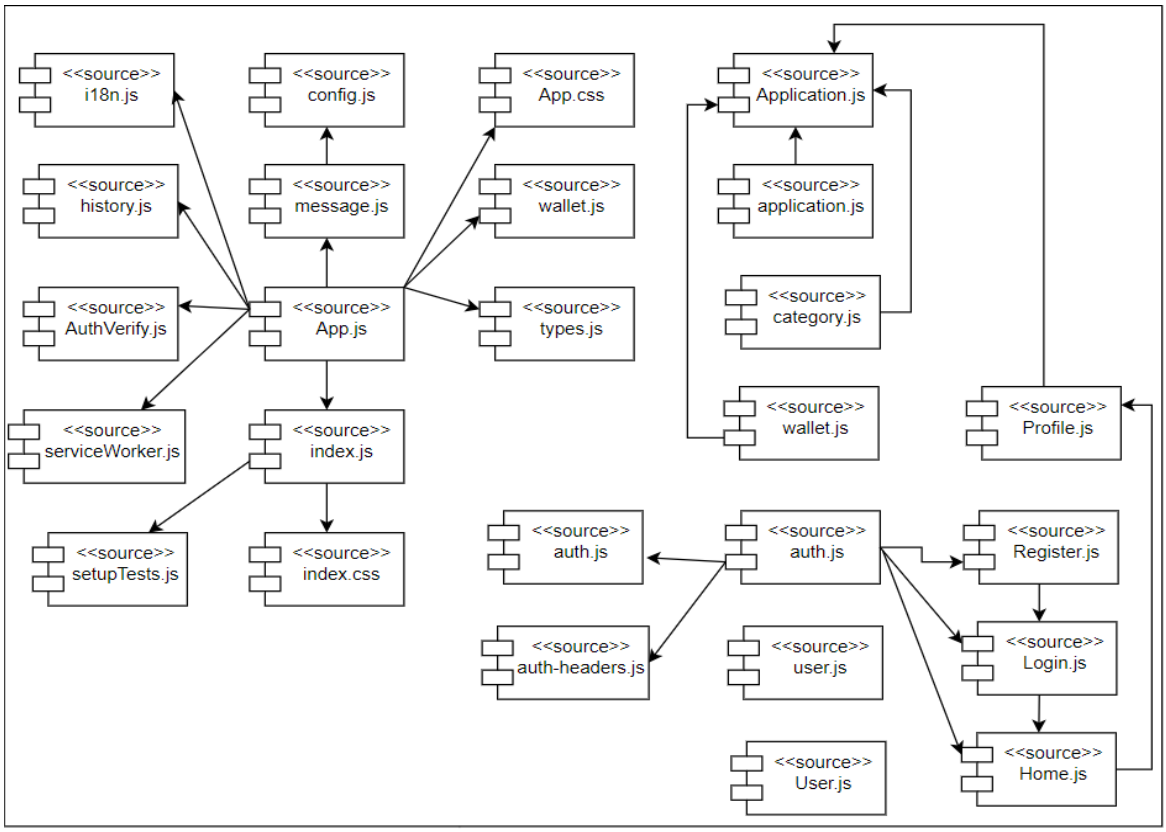


Рисунок 3.6 – Діаграма компонентів для системи «Програмна система для управління особистими фінансами»

3.4 Приклади найцікавіших алгоритмів та методів

Діаграма послідовності – це візуальний інструмент, який використовується для представлення взаємодії між різними елементами або компонентами системи. Вона чітко показує, як різні частини системи обмінюються повідомленнями та взаємодіють між собою. Діаграма послідовності відображає послідовність викликів методів та їх обробку, даючи чітке уявлення про те, як відбувається передача інформації між об'єктами [6].

На рисунку 3.7 наведено приклад діаграми послідовності, яка описує процес отримання даних про транзакції з банку. На цій діаграмі чітко показано, які кроки виконують користувач, сервер та банк для отримання необхідної інформації.

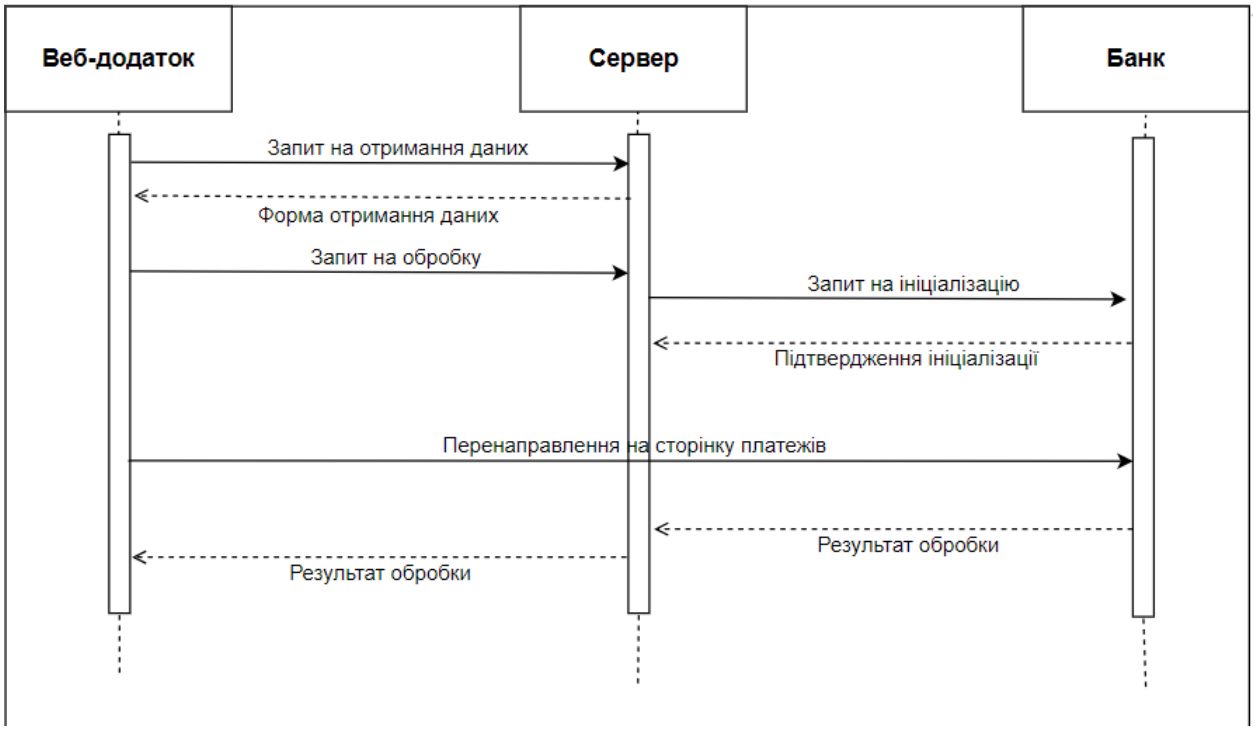


Рисунок 3.7 – Діаграма послідовності для обробки платежів з картки для системи «Програмна система для управління особистими фінансами»

Опис діаграми послідовності для отримання транзакції з банку.

В процесі беруть участь декілька учасників: користувач, як ініціатор процесу, який бажає отримати інформацію про свої транзакції; система – програмне забезпечення, що використовується користувачем для доступу до банківських даних; банк – місце, де зберігаються дані про транзакції користувача.

Спочатку користувач вводить свої дані та натискає кнопку "Отримати дані про транзакції". Далі система формує запит, який містить інформацію про користувача та тип транзакцій, що цікавлять його. Цей запит надсилається до банку через захищений канал. Банк отримує запит від системи та перевіряє, чи дійсно це запит від авторизованого користувача. Якщо автентифікація та авторизація успішні, банк переходить до наступного кроку. Банк використовує отриману інформацію про користувача та тип транзакцій, щоб знайти відповідні записи у своїй базі даних. Ці записи можуть включати дату, опис, суму та інші деталі транзакцій. Банк формує файл з даними, який містить знайдені записи про транзакції, та надсилає його системі. Система отримує файл від банку та розшифровує його. Потім вона обробляє дані, щоб сформувати зрозумілий для користувача звіт. Система презентує звіт про транзакції користувачеві на екрані. Звіт може містити таблицю з даними, графіки або інші візуалізації.

Протягом всієї взаємодії використовується захищений канал зв'язку для забезпечення конфіденційності та безпеки даних. Банк ретельно перевіряє автентифікацію та авторизацію користувача, перш ніж надавати доступ до даних. Система обробляє дані про транзакції згідно з вимогами та стандартами безпеки.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Опис використаних інструментів програмної розробки

Для створення сучасної та багатофункціональної серверної частини кваліфікаційного проекту було обрано мову програмування C#. Цей вибір обумовлений її високою функціональністю, динамічністю та широкою популярністю у веб-розробці.

C# надає доступ до великої та активної спільноти розробників, що забезпечує доступ до численних бібліотек та інструментів з відкритим кодом. Це полегшує пошук готових рішень для типових задач. Додатково, C# є об'єктно-орієнтованою мовою, що робить її зручною для розробки модульного та повторно використовуваного коду, що значно спрощує процес розробки та обслуговування складних програмних систем [7].

Було проведено аналіз двох популярних фреймворків веб-розробки на C#: ASP.NET Core та ASP.NET MVC. Обидва фреймворки мають свої переваги та недоліки:

ASP.NET Core:

- переваги: кросплатформність, модульність, гнучкість, підтримка сучасних веб-технологій, вбудована підтримка мікросервісів, простий синтаксис та добре задокументований API;
- недоліки: не виявлено.

ASP.NET MVC:

- переваги: зрілість, доступність бібліотек та ресурсів, велике та активне співтовариство розробників;
- недоліки: менша підтримка сучасних веб-технологій, порівняно з ASP.NET Core.

На основі проведеного аналізу було обрано фреймворк ASP.NET Core для розробки серверної частини проекту. Переваги роблять його оптимальним вибором для створення сучасної та масштабованої системи.

Entity Framework Core – це об'єктно-реляційний маппер для .NET, який використовується для спрощення взаємодії з базами даних з програм на C#. Він полегшує розробникам доступ до даних, їх маніпулювання та зберігання, без необхідності писати складні SQL-запити [8].

EF Core дозволяє створювати моделі, які представляють структуру бази даних в об'єктно-орієнтованому вигляді. Ці моделі відображають таблиці, стовпчики та зв'язки в базі даних, використовуючи класи C# замість SQL-коду.

Також відстежуються зміни, внесені до об'єктів моделі, і автоматично генерує відповідні SQL-запити для оновлення бази даних. Це звільняє розробників від необхідності вручну писати SQL-запити для оновлення даних.

Додатково, EF Core підтримує LINQ, що дозволяє використовувати знайомий синтаксис для запиту до даних з об'єктів моделі, значно спрощуючи процес отримання та обробки даних з бази даних.

Ця технологія використовується в проекті для спрощення взаємодії з базою даних, покращення продуктивності та зменшення складності коду. Це дозволяє зосередитися на розробці основної функціональності системи, а не на написанні рутинних SQL-запитів.

Далі наведено код контролеру бюджету. Спочатку розглянемо додавання нового бюджету.

```
public async Task<ActionResult> PostBudget(BudgetModel budgetModel)
   {
        try
        {

_context.Budgets.Add(GetBudgetFromModel(budgetModel));
            await _context.SaveChangesAsync();
                  return CreatedAtAction("GetBudget", new { id =
budgetModel.BudgetId }, budgetModel);
        }
        catch (Exception ex)
        {
            return BadRequest(ex.Message);
        }
          }
```

Вхідний параметр – *BudgetModel*, модель даних бюджету, що потрібно додати. Викликається метод *Add* для додавання об'єкта бюджету до контексту бази даних. Асинхронно зберігаються зміни в базі даних за допомогою *SaveChangesAsync*. Якщо запис успішно додано, повертається *CreatedAtAction* з деталями новоствореного запису. У разі помилки повертається BadRequest з повідомленням про помилку.

```
     public async Task<IActionResult> PutBudget(int id, BudgetModel
budgetModel)
        {
            if (id != budgetModel.BudgetId)
            {
                return BadRequest();
            }

            _context.Entry(GetBudgetFromModel(budgetModel)).State =
EntityState.Modified;

            try
            {
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!BudgetExists(id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }

            return NoContent();
             }
```

Наведений вище метод призначений для обробки HTTP PUT-запитів, спрямованих на оновлення існуючого запису бюджету в базі даних. Вхідні параметри ідентифікатор бюджету і модель даних бюджету. Якщо ідентифікатори не співпадають, повертається *BadRequest*. Використовується контекст бази даних для позначення об'єкта як зміненого і асинхронного збереження змін. У разі виникнення *DbUpdateConcurrencyException* перевіряється, чи існує бюджет з

вказаним ідентифікатором. Якщо ні, повертається *NotFound*. Якщо операція успішна, повертається *NoContent*.

```
public async Task<ActionResult<Budget>> GetBudget(int id)
   {
       var budget = await _context.Budgets.FindAsync(id);

       if (budget == null)
       {
           return NotFound();
       }

       return Ok(GetModelFromBudget(budget));
         }
```

Цей код представляє асинхронний метод *GetBudget*. Бюджет шукається у базі даних за заданим ідентифікатором. Якщо бюджет не знайдено, повертається *NotFound*. Якщо бюджет знайдено, повертається *Ok* з даними бюджету, перетвореними за допомогою *GetModelFromBudget*.

```
public async Task<IActionResult> DeleteBudget(int id)
   {
       var budget = await _context.Budgets.FindAsync(id);
       if (budget == null)
       {
           return NotFound();
       }

       _context.Budgets.Remove(budget);
       await _context.SaveChangesAsync();

       return NoContent();
         }
```

Метод призначений для обробки HTTP DELETE-запитів, спрямованих на видалення запису бюджету з бази даних за його ідентифікатором. Асинхронно шукається бюджет у базі даних за заданим ідентифікатором. Використовується контекст бази даних для видалення знайденого об'єкта. Асинхронно зберігаються зміни в базі даних для підтвердження видалення.

Для розробки клієнтської частини кваліфікаційної роботи було обрано мову програмування JavaScript, а саме бібліотеку React. Цей вибір обумовлений

численними перевагами JavaScript та React, які роблять їх оптимальними інструментами для створення сучасних та інтерактивних веб-застосунків.

JavaScript та React використані в проекті для створення динамічного та інтерактивного інтерфейсу користувача, який буде зручним та інтуїтивно зрозумілим для користувачів.

Компонентний підхід React буде використовуватися для розробки модульних та повторно використовуваних компонентів інтерфейсу користувача, що зробить код більш організованим та легким у підтримці.

Віртуальний DOM React використовувався для покращення продуктивності та ефективності інтерфейсу користувача.

4.2 Створення бюджету на місяць

Однією з основних задач є створення бюджету на місяць для відстеження витрат. Код, який використовується для створення бюджету, складається з декількох компонентів, серед яких:

- App.jsx – цей файл є головною точкою входу для React-додатку. Він описує загальну структуру інтерфейсу користувача, включаючи маршрутизацію та компонування сторінок. App.jsx містить компонент Router, який використовується для навігації між різними сторінками додатку;
- BudgetPage.jsx – цей файл описує компонент, який відповідає за відображення інтерфейсу сторінки бюджету. Він містить такі елементи, як таблиця для відображення доходів та витрат, діаграми для візуалізації фінансових даних, а також кнопки для додавання нових записів та редагування існуючих;
- AddBudgetForm.jsx – описує компонент, який використовується для додавання нових записів до бюджету. Він містить поля вводу для введення суми, категорії та опису нового запису, а також кнопку для збереження запису;

– helpers.js – містить допоміжні функції, які використовуються в інших файлах. Допоміжні функції можуть використовуватися для різних цілей, таких як форматування даних, розрахунок суми доходів та витрат, або перевірка введених даних.

Далі розглянемо фрагменти коду, які відповідають за певний функціонал у процесі створення бюджету на місяць.

```
export async function budgetLoader({ params }) {
  const budget = await getAllMatchingItems({
    category: "budgets",
    key: "id",
    value: params.id,
  })[0];

  const expenses = await getAllMatchingItems({
    category: "expenses",
    key: "budgetId",
    value: params.id,
  });

  if (!budget) {
    throw new Error("The budget you’re trying to find doesn’t
exist");
  }

  return { budget, expenses };
}
```

Цей фрагмент коду відповідає за отримання конкретного бюджету та його відповідних витрат на основі наданого ID. Він отримує дані асинхронно та обробляє потенційні помилки, якщо бюджет не знайдено.

```
if (_action === "createBudget") {
    try {
      createBudget({
        name: values.newBudget,
        amount: values.newBudgetAmount,
      });
      return toast.success("Budget created!");
    } catch (e) {
      throw new Error("There was a problem creating your budget.");
    }
  }
```

Цей код створює новий бюджет на основі вводу користувача (*newBudget* та *newBudgetAmount*) та надає зворотний зв'язок користувачеві за допомогою повідомлень про успіх або обробки помилок.

```
      return (
          <div className="form-wrapper">
            <h2 className="h3">
              Create budget
            </h2>
            <fetcher.Form
              method="post"
              className="grid-sm"
              ref={formRef}>
              <div className="grid-xs">
                <label htmlFor="newBudget">Budget Name</label>
                <input
                  type="text"
                  name="newBudget"
                  id="newBudget"
                  placeholder="e.g., Groceries"
                  required
                  ref={focusRef}/>
              </div>
              <div className="grid-xs">
                <label htmlFor="newBudgetAmount">Amount</label>
                <input
                  type="number"
                  step="0.01"
                  name="newBudgetAmount"
                  id="newBudgetAmount"
                  placeholder="e.g., $350"
                  required
                  inputMode="decimal"/>
              </div>
              <input type="hidden" name="_action" value="createBudget" />
              <button type="submit" className="btn btn--dark"
disabled={isSubmitting}>
                {
                  isSubmitting ? <span>Submitting…</span> : (<>
                      <span>Create budget</span>
                      <CurrencyDollarIcon width={20} /></>)
                }
              </button>
            </fetcher.Form>
          </div>)
```

Наведений вище код створює форму для введення назви та суми нового бюджету. Користувач може заповнити поля форми та натиснути кнопку "Створити бюджет" для надсилання даних на сервер.

4.3 Додавання витрат до бюджету

Ще однією важливою функцією є додавання витрат до бюджету. Для цього процесу залучено наступні елементи:

- App.jsx – цей файл є головною точкою входу для React-додатку. Він описує загальну структуру інтерфейсу користувача, включаючи маршрутизацію та компонування сторінок;
- ExpensesPage.jsx – файл описує компонент, який відповідає за відображення інтерфейсу сторінки витрат. Він містить такі елементи, як список для відображення витрат, фільтри для сортування та категоризації витрат, а також кнопки для додавання нових витрат та редагування існуючих;
- helpers.js – файл, що містить допоміжні функції, які використовуються в інших файлах. Допоміжні функції можуть використовуватися для різних цілей, таких як форматування даних, розрахунок суми витрат, або перевірка введених даних;
- ExpenseItem.jsx – описує компонент, який використовується для відображення окремої витрати. Він містить інформацію про категорію витрати, суму, опис, дату та інші релевантні деталі;
- AddExpenseForm.jsx – файл описує компонент для додавання нових витрат до бюджету. Він містить поля вводу для введення категорії, суми, опису, дати та інших деталей витрати, а також кнопку для збереження нової витрати.

Далі звернемо увагу на основні функції, які допомагають у реалізації функціоналу додавання витрати до бюджету.

```
export async function expensesAction({ request }) {
  const data = await request.formData();
  const { _action, ...values } = Object.fromEntries(data);

  if (_action === "deleteExpense") {
    try {
      deleteItem({
        key: "expenses",
```

```
        id: values.expenseId,
      });
      return toast.success("Expense deleted!");
    } catch (e) {
      throw new Error("There was a problem deleting your expense.");
    }
  }
}
```

Цей код визначає асинхронну функцію під назвою *expensesAction*, яка обробляє запит користувача, пов'язаний з витратами в React-додатку. Всередині обробляється запит користувачів, пов'язаний з витратами, зосереджуючись саме на видаленні витрати на основі її ID. Він отримує дані з форми, витягує бажану дію та деталі витрат, а також виконує видалення з обробкою помилок та повідомленнями про успіх.

```
const budget = getAllMatchingItems({
    category: "budgets",
    key: "id",
    value: expense.budgetId,
  })[0];
```

Наведений вище код використовується для пошуку бюджету, пов'язаного з певною витратою. Викликається функція *getAllMatchingItems* для пошуку бюджету, пов'язаного з певною витратою, на основі ID витрати. Шукається перший бюджет, який відповідає ID витрати, і зберігає його в змінній *budget*.

```
export const updateBudgetAfterPayment = ({ budgetId, amount }) => {
  const budgets = JSON.parse(localStorage.getItem("budgets")) || [];
  const updatedBudgets = budgets.map((budget) => {
    if (budget.id === budgetId) {
      return {
        ...budget,
        amount: budget.amount - amount
      };
    }
    return budget;
  });

  localStorage.setItem("budgets", JSON.stringify(updatedBudgets));
};
```

Тут ми визначаємо функцію updateBudgetAfterPayment, яка оновлює суму конкретного бюджету після здійснення платежу. Для кожного бюджету перевіряє, чи збігається *id* бюджету з наданим *budgetId*. Якщо знайдено збіг, повертається новий об'єкт бюджету із зменшеною наданою сумою *amount*. Решта властивостей бюджету залишаються незмінними. Оновлені дані зберігаються в базі даних.

```
export const createRegularPayment = ({ name, amount, budgetId, date
}) => {
  const newPayment = {
    id: crypto.randomUUID(),
    name: name,
    createdAt: Date.now(),
    amount: +amount,
    budgetId: budgetId,
    date: new Date(date).toISOString(),
  };

  const                    existingRegularPayments                    =
JSON.parse(localStorage.getItem("regularPayments")) || [];
  localStorage.setItem(
    "regularPayments",
    JSON.stringify([...existingRegularPayments, newPayment])
  );

  updateBudgetAfterPayment({ budgetId, amount: amount });
};
```

Цей код визначає функцію createRegularPayment, яка створює новий регулярний платіж, та оновлює відповідний бюджет після здійснення платежу. Для цього використовуються надані параметри (ім'я, сума, ідентифікатор бюджету, дата).

```
const validateInputs = () => {
      const errors = {};
      if (!/^\d{16}$/.test(state.number)) {
        errors.number = 'Invalid card number';
      }
      if (!/^\d{4}$/.test(state.expiry)) {
        errors.expiry = 'Invalid expiry date';
      }
      if (!/^\d{3,4}$/.test(state.cvc)) {
        errors.cvc = 'Invalid CVC';
      }
      if (!state.name) {
        errors.name = 'Name is required';
      }
      return errors;
```

```
};
```

Визначається функція *validateInputs*, яка перевіряє валідність даних форми введення платіжної картки. Вона перевіряє номер картки, дату закінчення дії, код CVC та ім'я власника картки. Для цього використовуються регулярні вирази для перевірки валідності формату даних і повертається об'єкт *errors*, який містить повідомлення про помилки для кожного поля, яке не пройшло валідацію.

4.4 Основні сторінки програмної системи

Для деяких ключових сторінок проекту, щоб краще зрозуміти інтерфейс і функціональність системи, наведені скріншоти.

Сторінка авторизації: тут користувачі можуть увійти до системи за допомогою своїх облікових записів (див. рис. 4.1).

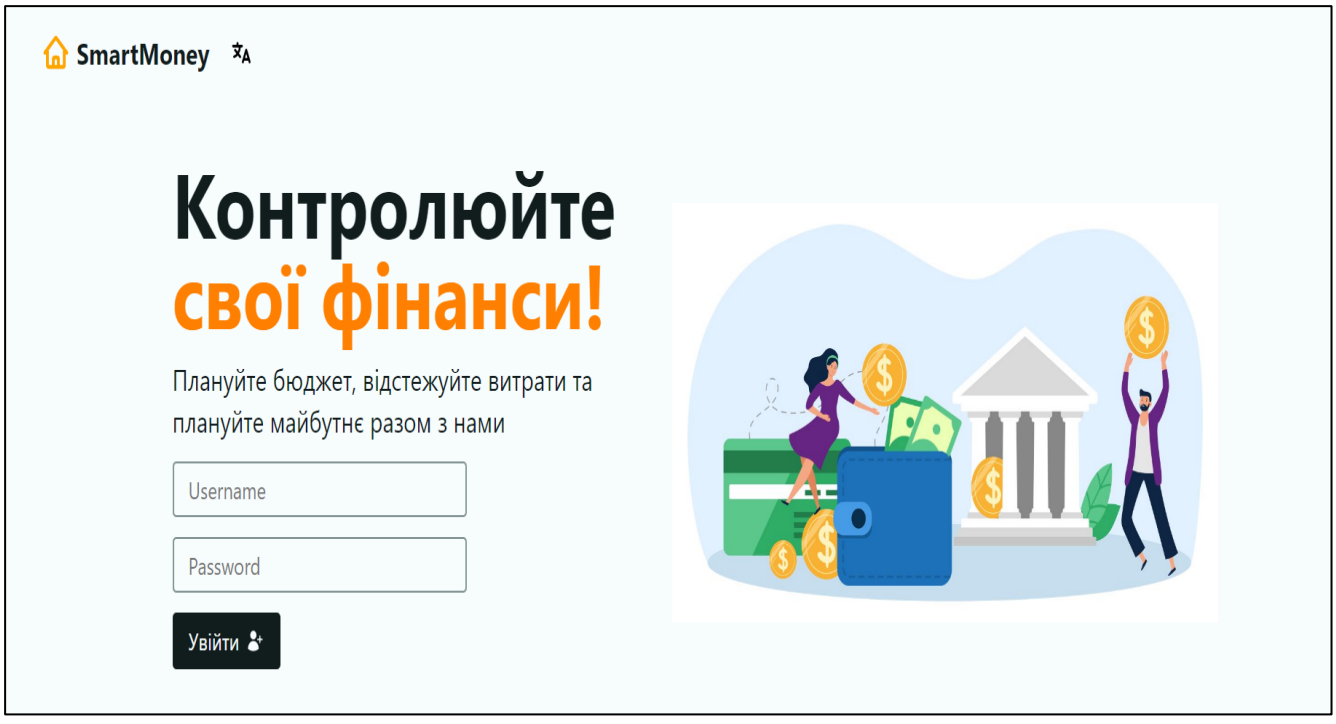


Рисунок 4.1 – Сторінка авторизації

Головний екран: тут можна переглянути основну інформацію, що надає програма (див. рис. 4.2).

Додавання бюджету: форма для створення нового бюджету (див. рис. 4.3).

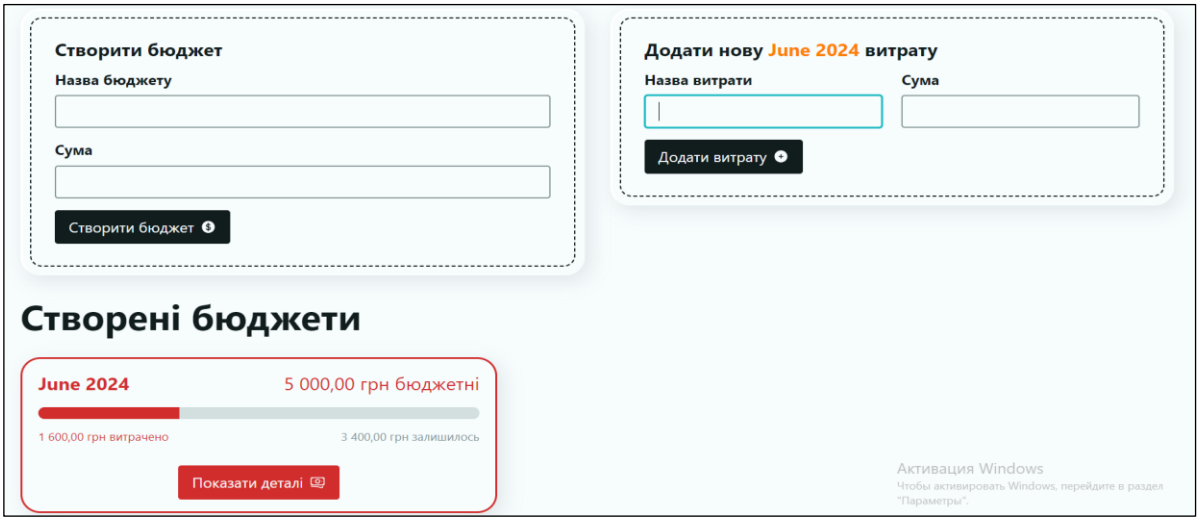


Рисунок 4.2 – Головний екран програми

Рисунок 4.3 – Форма для додавання бюджету

Додавання витрат: тут можна додати нові витрати до бюджетів (див. рис. 4.4).

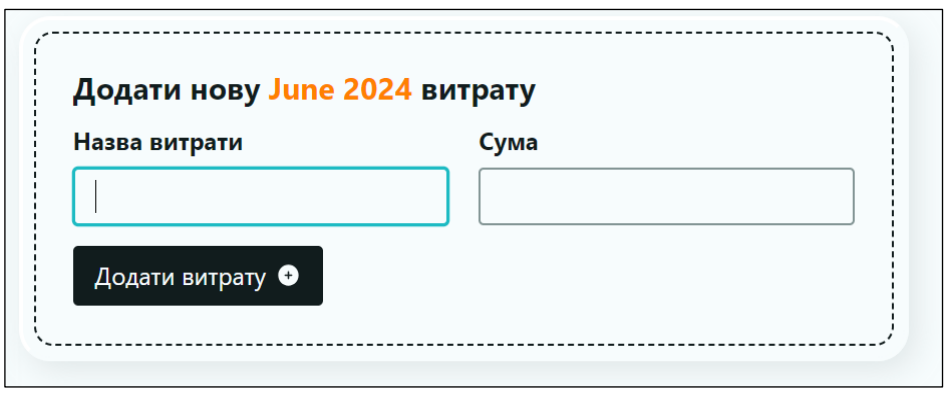


Рисунок 4.4 – Форма для додавання витрат

Додавання регулярних платежів: якщо користувач має платежі, які проводяться в певний день кожного місяця, можна тут додати інформацію про них (див. рис. 4.5).

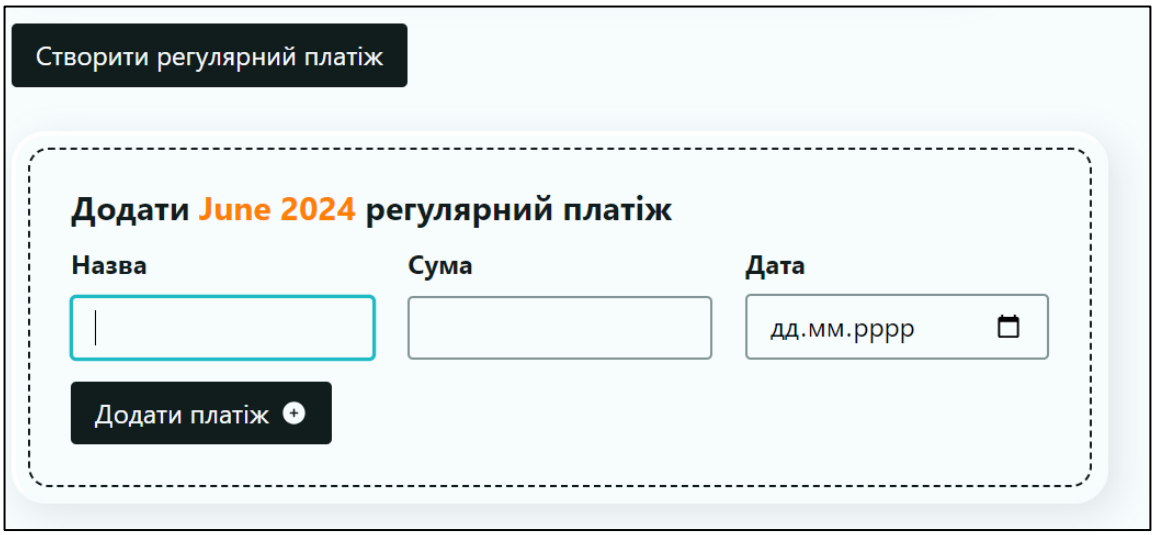


Рисунок 4.5 – Форма для створення регулярного платежу

Додавання карток: тут можна прив’язати картки до свого профілю, що дозволить в подальшому використовувати інформацію про транзакції для кращого аналізу бюджету (див. рис. 4.6).

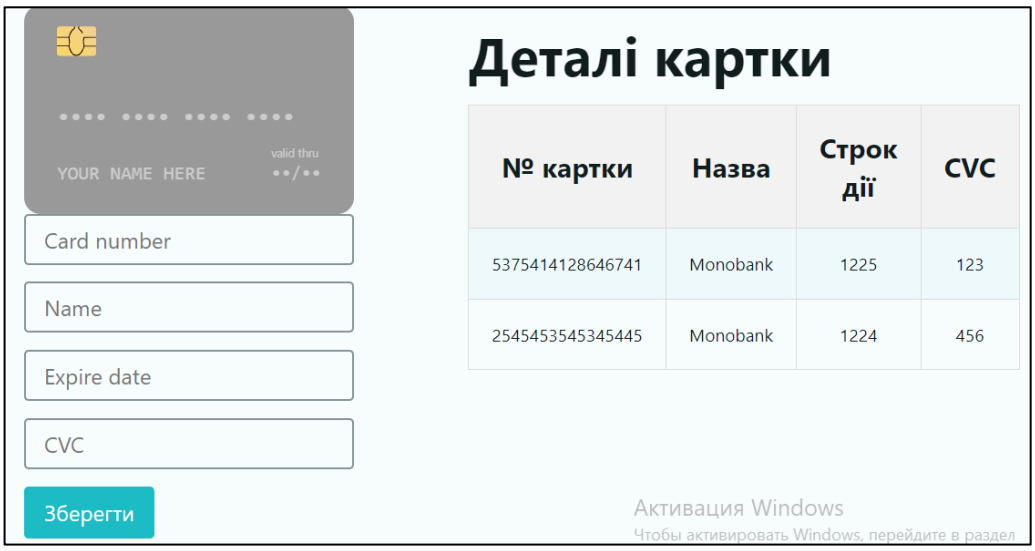


Рисунок 4.6 – Додавання карток

Огляд транзакцій: можна переглянути усі транзакції з певного бюджету (див. рис. 4.7).

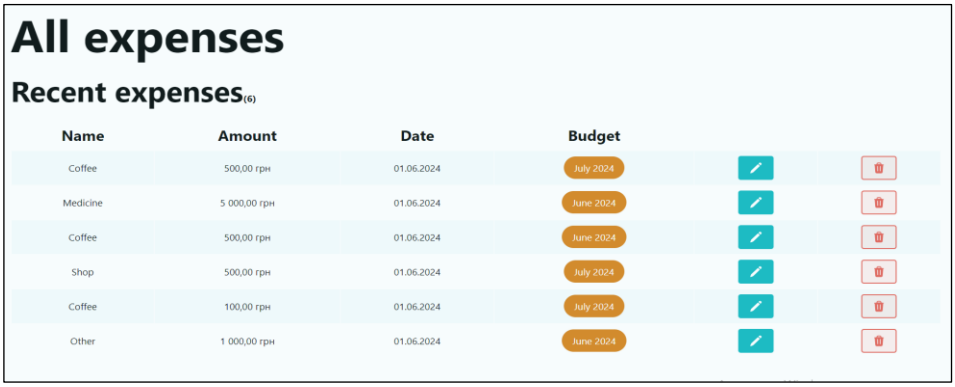


Рисунок 4.7 – Огляд всіх транзакцій

Сторінка профілю користувача: тут користувач може відредагувати свої дані (див. рис. 4.8).

Рисунок 4.8 – Сторінка профілю користувача

Редагування бюджету: в цій формі можна внести зміни до бюджету (див. рис. 4.9).

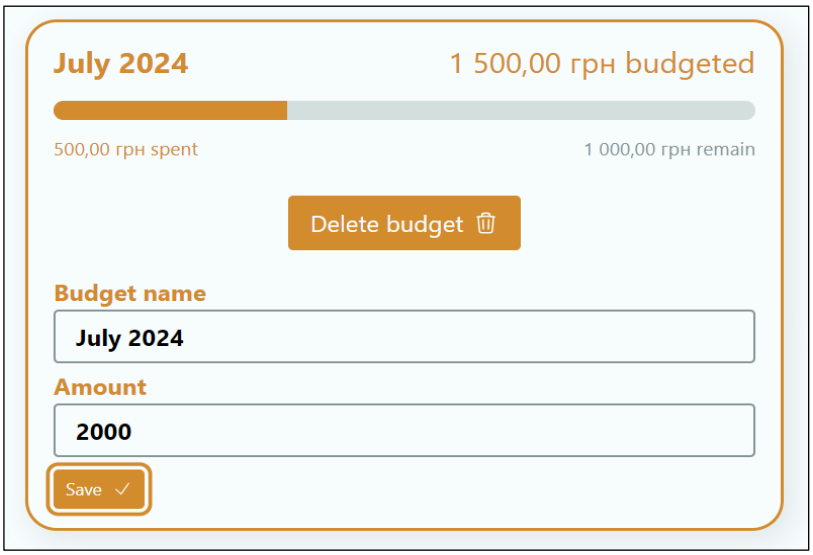


Рисунок 4.9 – Форма для редагування бюджету

Сторінка статистики: тут відображається статистика витрат за останні місяці, а також діаграми з витратами по категоріях (див. рис. 4.10).

Рисунок 4.10 – Сторінка статистики

Дизайн зроблений максимально інтуїтивно зрозумілим, оскільки він орієнтований на аудиторію різного віку.

5 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Обґрунтування вибору типів тестування

Тестування є невід'ємною частиною життєвого циклу розробки програмної системи управління особистими фінансами. Воно гарантує, що кожен компонент системи функціонує безперебійно, відповідає очікуванням користувачів та забезпечує високий рівень надійності [9].

На клієнтській стороні тестування зосереджується на інтерфейсі користувача, перевіряючи його зручність, відповідність потребам та загальну привабливість для користувачів. Це потребує ретельного розуміння вимог до користувацького досвіду.

Ключові аспекти тестування клієнтської частини:

- функціональність: перевірка того, чи всі функції системи працюють коректно та відповідають очікуванням користувачів;
- графічний інтерфейс користувача (GUI): оцінка естетики, ергономічності та зручності використання інтерфейсу;
- простота використання: перевірка того, чи інтерфейс зрозумілий та простий у навігації для користувачів з різним рівнем досвіду.

Для забезпечення високої якості та надійності програмної системи рекомендується використовувати комбінацію різних типів тестування, включаючи:

Тестування допомагає виявити та виправити помилки до того, як вони вплинуть на користувачів. Це оптимізація системи та усунення вузьких місць для покращення швидкості та надійності. Тестуваня дозволяє перевірити, чи відповідає система очікуванням користувачів та забезпечує позитивний досвід використання. Таким чином, гарантовано, що система відповідає всім вимогам та стандартам якості.

5.2 Приклад створення тест-кейсів

У якості прикладу наведено створення тест-кейсу №1 для опису функції створення бюджету та додавання до нього витрат наведено в таблиці 5.1.

Таблиця 5.1 – Тест-кейс №1

Інформація про тест-кейс			
Ідентифікатор тесту:	Тест-кейс №1		
Опис функції:	Створення бюджету та додавання до нього витрат		
Власник тесту:	Пашкевич Кирило Олександрович		
Дата створення:	25.05.2024		
Мета тесту:	Перевірити коректність реалізації створення бюджету та додавання до нього витрат		
Передумова			
№	**Опис випадку**	**Очікуваний результат**	**Висновок**
1	Відкрити веб-застосунок	Користувач має доступ до відкритого сайту	Пройдено
2	Користувач хоче авторизуватись у системі	Перехід на головну сторінку	Пройдено
Створення бюджету			
№	**Опис випадку**	**Очікуваний результат**	**Висновок**
1	Відкриття форми створення бюджету	З’являється форма для додавання нового бюджету	Пройдено
2	Заповнити поле «Назва»	Введені дані коректні	Пройдено
3	Заповнити поле «Розмір»	Введено коректне число	Пройдено
4	Натиснути кнопку «Створити бюджет»	Виведення повідомлення про успішне створення бюджету та перехід на головну сторінку	Пройдено
Додавання витрати до бюджету			
1	Відкриття форми для додавання витрати	З’являється форма для додавання нової витрати	Пройдено
2	Заповнити поле «Категорія»	Введено коректну категорію	Пройдено
3	Заповнити поле «Сума»	Введено коректне число	Пройдено
4	Натиснути на кнопку «Додати витрату»	Витрату додано до бюджету, перехід до сторінки бюджету	Пройдено
Результати тестування			
Тестувальник: Пашкевич К.О.	**Дата виконання тесту: 25.05.2024**	**Результат тесту (P/F/B): ПРОЙДЕНО (P)**	

У таблиці 5.1 ми можемо побачити успішне проходження тест-кейсу та вдало протестовані функціональні аспекти. Усі аспекти тестування були виконанні , а фактичні результати збіглися . Цей факт говорить нам про те, що функціональність програми та системи відповідає всім поставленим вимогам та готова до використання.

5.3 Тестування взаємодії користувача

Під час тестування користувацького досвіду ретельно досліджуються всі аспекти, що впливають на взаємодію користувачів з інтерфейсом та загальний досвід від використання продукту.

Під час тестування користувацького досвіду були протестовані такі елементи:

- авторизація в системі: перевірка коректності роботи полів для авторизації;
- додавання витрат: перевірка коректності роботи полів додавання витрат;
- перевірка можливості додавати витрати з різних категорій;
- створення бюджету: перевірка коректності роботи полів створення бюджету;
- перевірка доступності інтерфейсу для людей з різними можливостями;
- перевірка зручності використання інтерфейсу;
- перевірка інтуїтивності інтерфейсу;
- перевірка ефективності роботи з інтерфейсом;
- перевірка адаптивності інтерфейсу до різних екранів мобільних пристроїв.

Важливо зазначити, що тестування UI/UX - це не просто перевірка окремих елементів інтерфейсу. Це комплексне дослідження, яке враховує всі аспекти взаємодії користувачів з продуктом, щоб зробити його максимально комфортним, зручним та приємним.

5.4 Навантажувальне тестування

Навантажувальне тестування - це тип тестування програмного забезпечення, яке використовується для оцінки того, як система поводиться під навантаженням. Цей тип тестування допомагає визначити, чи може система витримувати очікуване навантаження користувачів та транзакцій, не знижуючи при цьому свою продуктивність та стійкість.

Мета проведення навантажувального тестування – оцінити, чи може система витримувати очікуване навантаження користувачів, не знижуючи при цьому свою продуктивність та стійкість.

Система повинна бути в змозі витримувати пікове навантаження користувачів без зниження продуктивності. Час відгуку системи повинен залишатися в межах прийнятного діапазону. Система не повинна зазнавати збоїв або помилок. Програмний продукт повинен бути масштабованим, щоб можна було додавати нових користувачів без негативного впливу на продуктивність [10].

Успішне проходження навантажувальних тестів дає ряд переваг, таких як: покращення досвіду користувачів, зниження ризику простоїв, підвищення конкурентоспроможності, підвищення впевненості в системі. Це означає, що система може витримувати очікуване навантаження користувачів без зниження продуктивності, збоїв або помилок.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи було створено та розроблено програмну систему, яка автоматизує управління особистими фінансами.

Перед розробкою системи було проведено ґрунтовне дослідження проблемної області та аналіз існуючих рішень. Аналіз показав, що жодна з наявних систем не в змозі повністю задовольнити потреби користувачів. Враховуючи це, було прийнято рішення розробити власну веб-систему, яка б відповідала всім вимогам та надавала чітку та зрозумілу інформацію.

Під час розробки системи було визначено основні потреби користувачів, а також проаналізовано переваги та недоліки існуючих продуктів на ринку. В результаті було сформульовано ключові функції системи, які дозволяють планувати бюджет, аналізувати витрати, отримувати персоналізовані рекомендації щодо покращення фінансового становища.

Клієнтська частина веб-додатку розроблена з використанням фреймворку React. Цей фреймворк дозволяє створювати добре структуровані, динамічні та масштабовані інтерфейси користувача. Завдяки React, система забезпечує швидке оновлення компонентів, передбачуваність стану програми, стабільність роботи.

Розроблена система стане незамінним інструментом для людей, які прагнуть покращити управління своїми фінансами, оптимізувати витрати та досягти фінансових цілей. Ця система відповідає сучасним потребам користувачів та пропонує інноваційні рішення для ефективного управління особистими фінансами.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What Is Personal Finance, and Why Is It Important? URL: https://www.investopedia.com/terms/p/personalfinance.asp (дата звернення: 04.05.2024)

2. Monefy. URL: https://monefy.me/ (дата звернення: 04.05.2024)

3. Ramsey D. The Total Money Makeover: A Proven System for Financial Freedom, 2003. – 272 с.

4. The Financial Diet. URL: https://thefinancialdiet.com/ (дата звернення: 04.05.2024)

5. Hamm T. The Simple Dollar: A Guide to Getting Your Money Right, 2010. – 272 с.

6. Minfin. URL: https://minfin.com.ua/ (дата звернення: 04.05.2024)

7. Making Sense Of Cents. URL: https://www.makingsenseofcents.com/ (дата звернення: 04.05.2024)

8. "The Complete Guide to Personal Finance Software". URL: https://www.amazon.com/Best-Sellers-Personal-Money-Management/zgbs/software/283015 (дата звернення: 28.05.2024)

9. "Zero to One: Notes on Startups, or How to Build the Future". URL: https://www.amazon.com/Zero-One-Notes-Startups-Future/dp/0804165254 (дата звернення: 28.05.2024)

10. You need a budget. URL: https://www.ynab.com/ (дата звернення: 30.05.2024)

ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

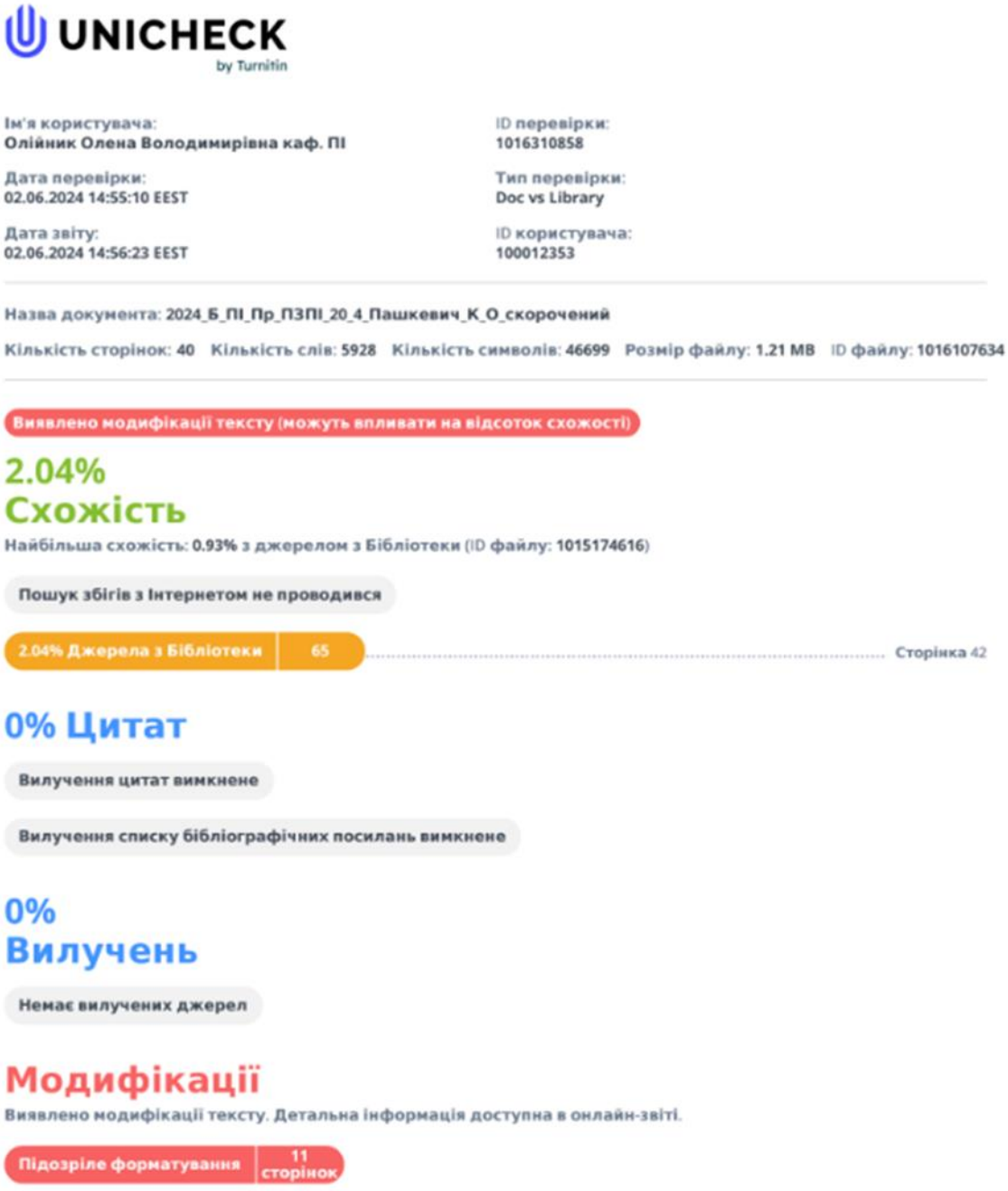


Рисунок А.1 – Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

ДОДАТОК Б

Слайди презентації

Рисунок Б.1 – Слайд презентації 1

Рисунок Б.2 – Слайд презентації 2

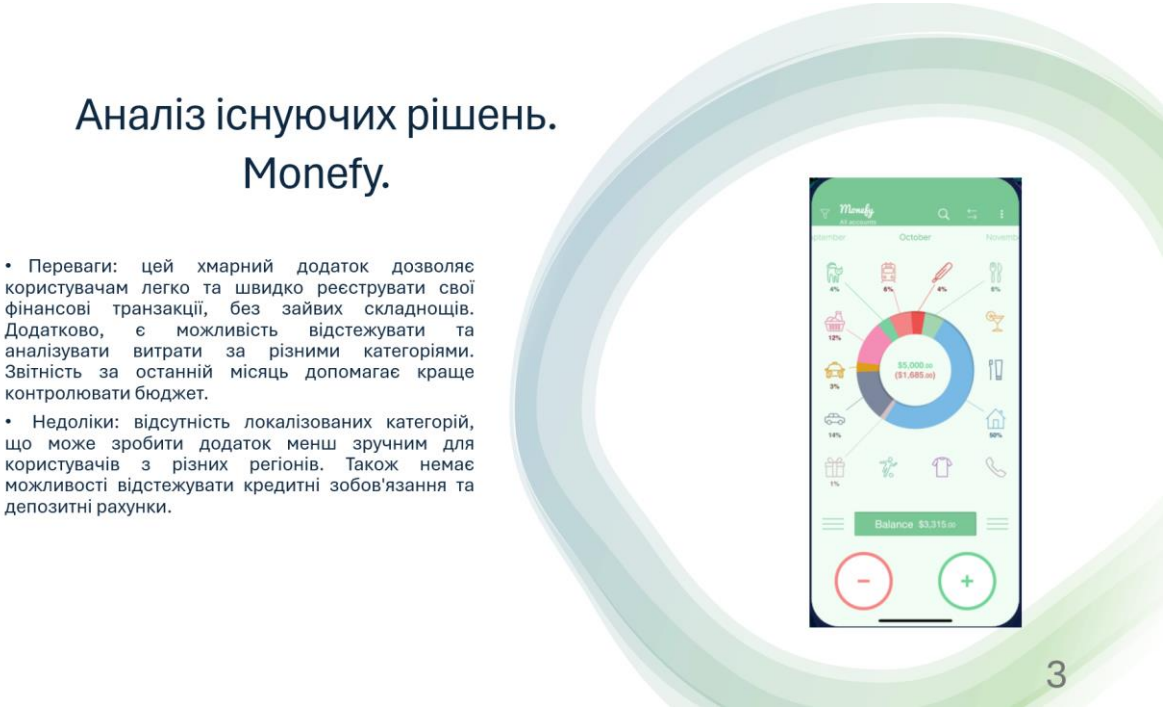


Рисунок Б.3 – Слайд презентації 3

Рисунок Б.4 – Слайд презентації 4

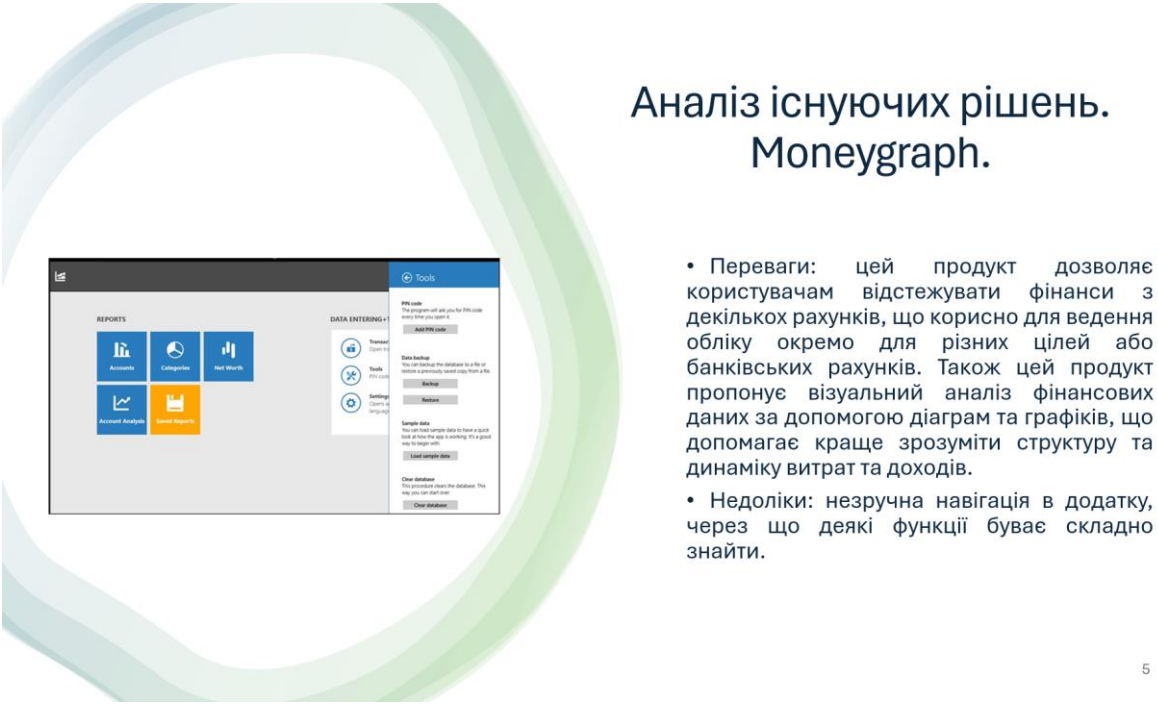


Рисунок Б.5 – Слайд презентації 5

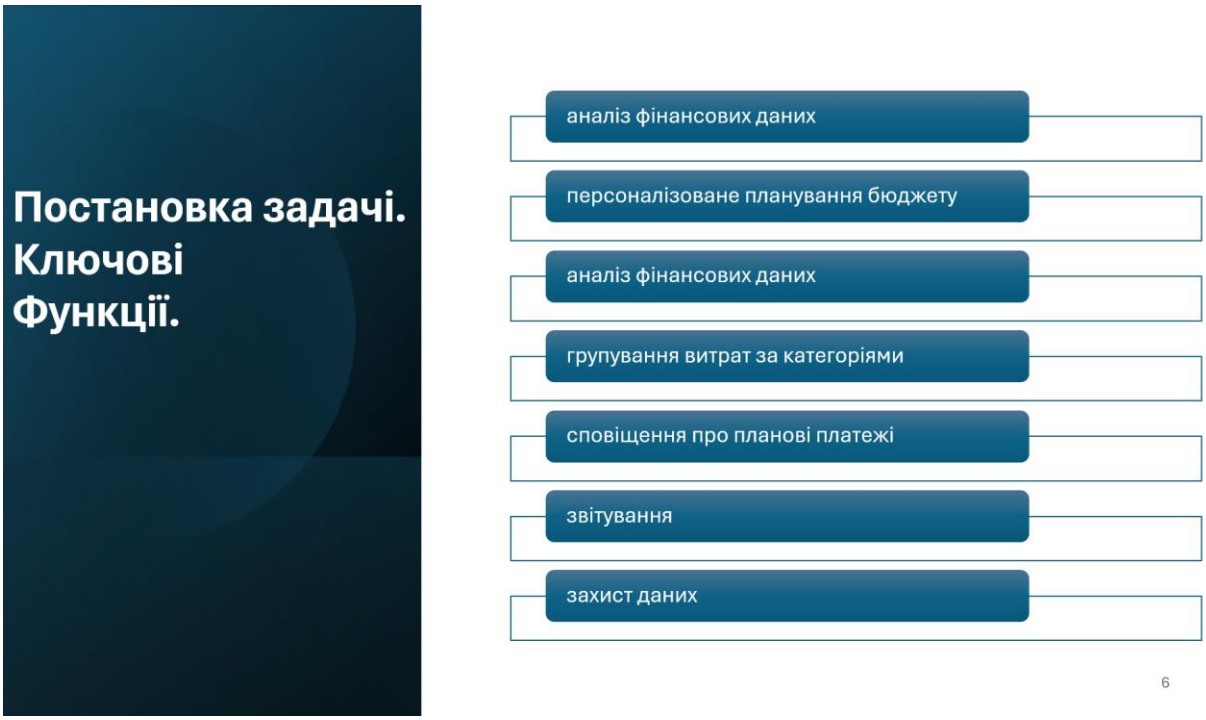
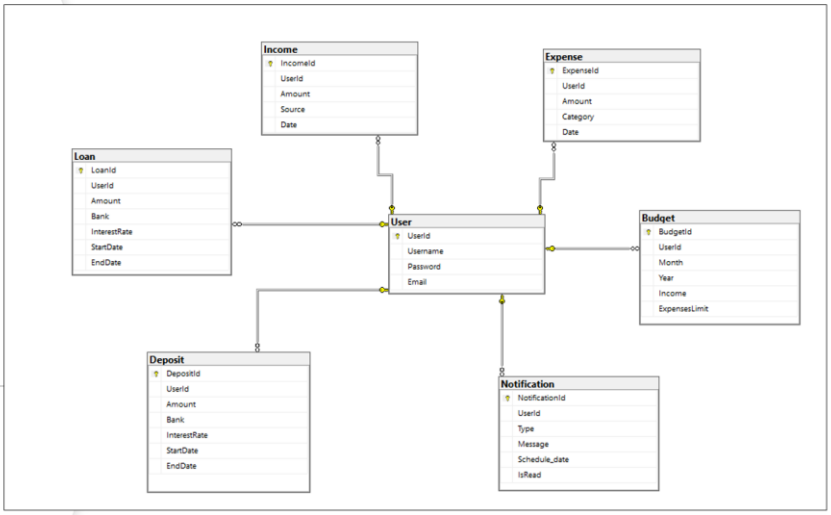


Рисунок Б.6 – Слайд презентації 6

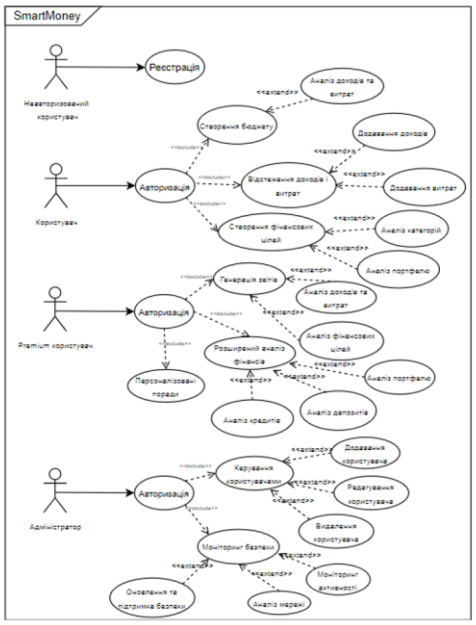
Діаграма бази даних



7

Рисунок Б.7 – Слайд презентації 7

Діаграма прецедентів



8

Рисунок Б.8 – Слайд презентації 8

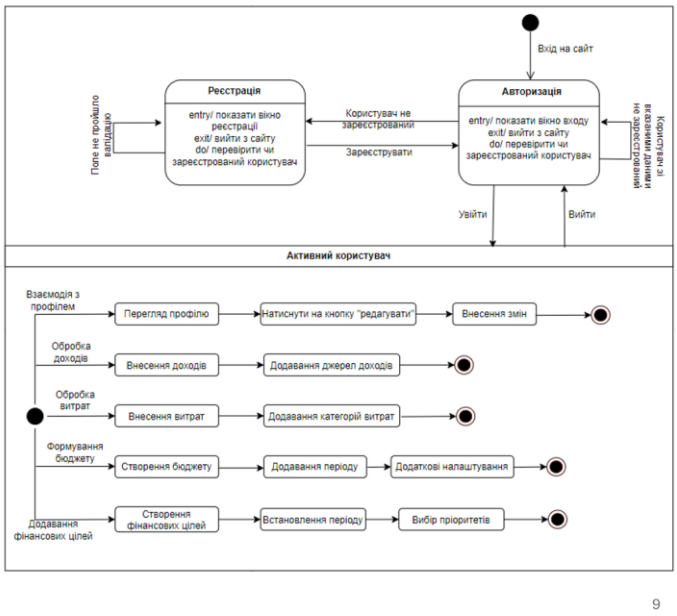


Рисунок Б.9 – Слайд презентації 9

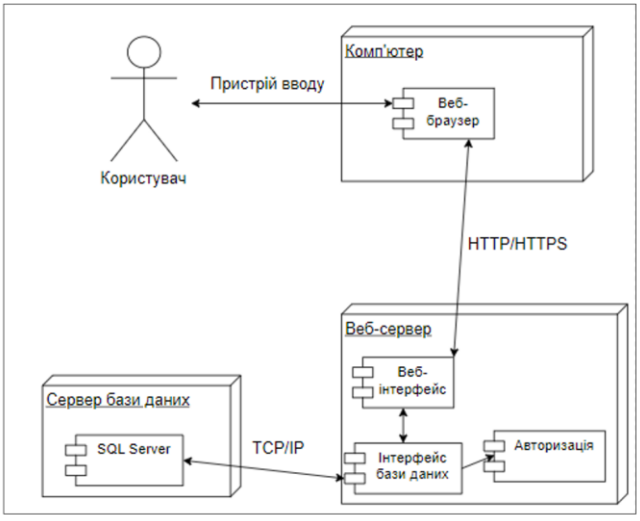
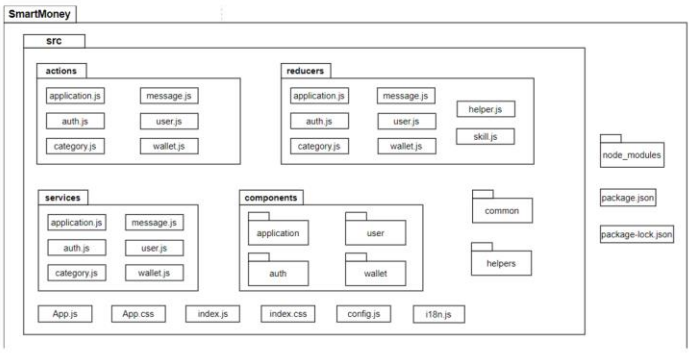


Рисунок Б.10 – Слайд презентації 10

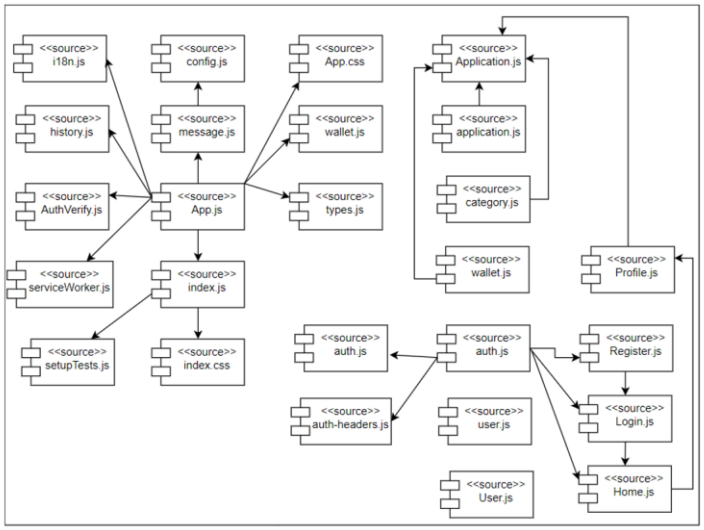
Діаграма пакетів



11

Рисунок Б.11 – Слайд презентації 11

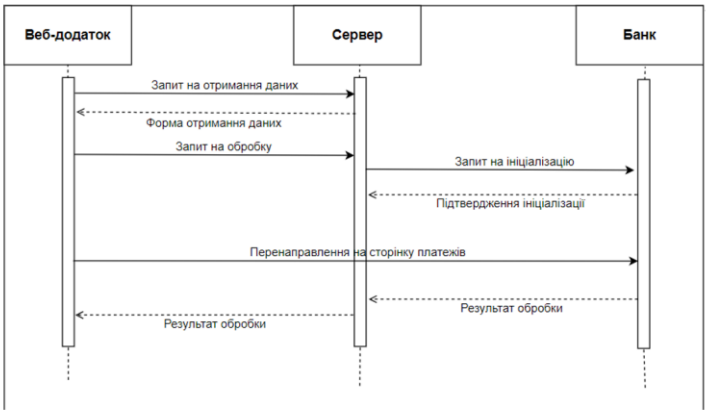
Діаграма компонентів



12

Рисунок Б.12 – Слайд презентації 12

Діаграма послідовності



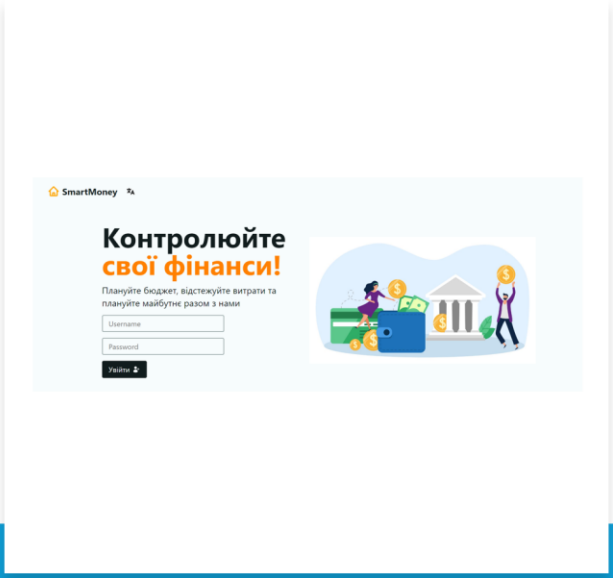
13

Рисунок Б.13 – Слайд презентації 13

Рисунок Б.14 – Слайд презентації 14

Сторінка авторизації

```
{!showEmailInput && (
      <button
       type="button"
       className="btn btn--dark"
       onClick={toggleEmailInput}
      >
       <Trans i18nKey={isRegistering ?
"login" : "register"} />
      </button>
     )}
```

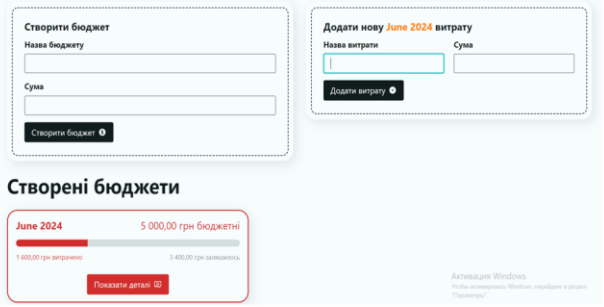


15

Рисунок Б.15 – Слайд презентації 15

Головний екран

```
<button
        type="button"
        className="btn btn--dark mt-3"
        onClick={handleToggleRegularPaymentForm}
       >
        <Trans i18nKey="create_regular_payment" />
       </button>
       {showRegularPaymentForm &&
<AddRegularPaymentForm budgets={budgets} />}
```

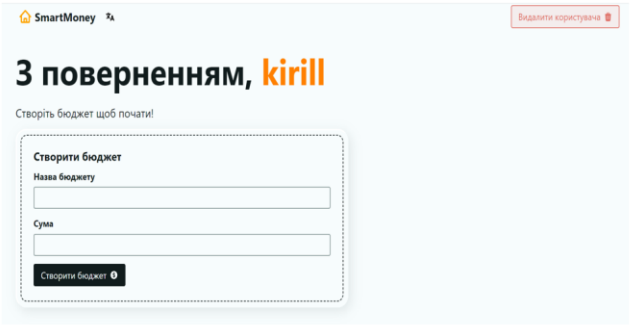


16

Рисунок Б.16 – Слайд презентації 16

Форма для додавання бюджету

```
<label
htmlFor="newBudget"><Trans
i18nKey="budget_name" /></label>
      <input
       type="text"
       name="newBudget"
       id="newBudget"
       placeholder=""
       required
       ref={focusRef}
      />
```

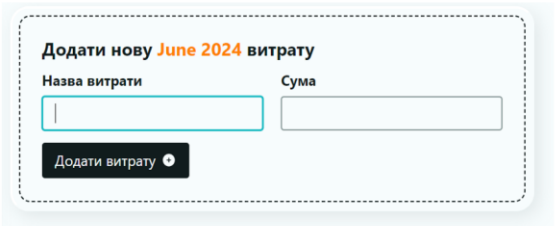


17

Рисунок Б.17 – Слайд презентації 17

Форма для додавання витрати

```
<label htmlFor="newExpenseBudget"><Trans
i18nKey="category" /></label>
      <select name="newExpenseBudget"
id="newExpenseBudget" required>
       {
        budgets
         .sort((a, b) => a.createdAt - b.createdAt)
         .map((budget) => {
          return (
           <option key={budget.id} value={budget.id}>
            {budget.name}
           </option>
          )
         })
       }
      </select>
```

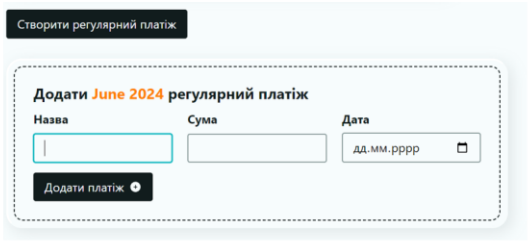


18

Рисунок Б.18 – Слайд презентації 18

Додавання регулярних платежів

```
<label htmlFor="newPaymentDate"><Trans
i18nKey="date" /></label>
        <input
          type="date"
          name="newPaymentDate"
          id="newPaymentDate"
          value={date}
          onChange={(e) =>
setDate(e.target.value)}
          required
        />
```

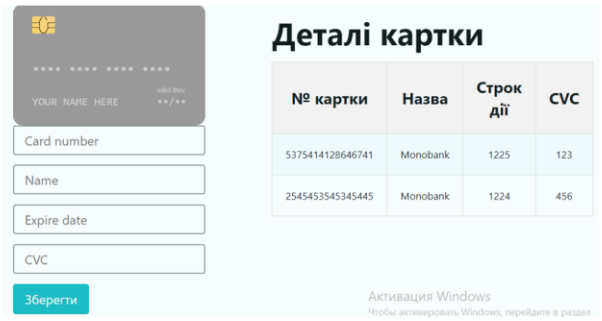


19

Рисунок Б.19 – Слайд презентації 19

Додавання карток

```
const validateInputs = () => {
  const errors = {};
  if (!/^\d{16}$/.test(state.number)) {
   errors.number = 'Invalid card number';
  }
  if (!/^\d{4}$/.test(state.expiry)) {
   errors.expiry = 'Invalid expiry date';
  }
  if (!/^\d{3,4}$/.test(state.cvc)) {
   errors.cvc = 'Invalid CVC';
  }
  if (!state.name) {
   errors.name = 'Name is required';
  }
  return errors;
 };
```

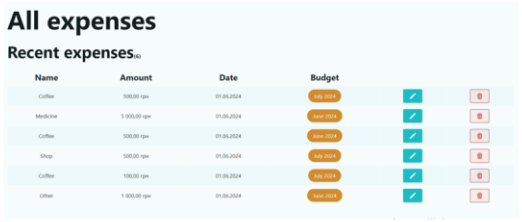


20

Рисунок Б.20 – Слайд презентації 20

Сторінка транзакцій

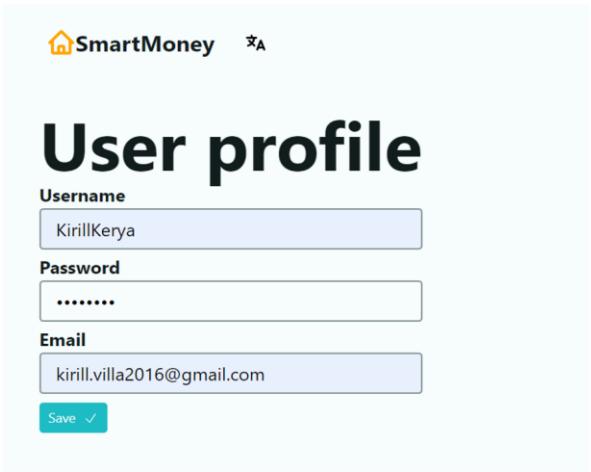
```
{expenses && expenses.length > 0 ? (
    <div className="grid-md">
     <h2>
     <Trans i18nKey="recent"
/><small>({expenses.length})</small
>
     </h2>
     <Table expenses={expenses} />
    </div>
```



21

Рисунок Б.21 – Слайд презентації 21

Сторінка профілю користувача




```
<thead>
        <tr>
          <th><Trans i18nKey="card_number" /></th>
          <th><Trans i18nKey="card_name" /></th>
          <th><Trans i18nKey="expiry_date" /></th>
          <th><Trans i18nKey="cvc" /></th>
        </tr>
      </thead>
      <tbody>
        {cards.map((card, index) => (
          <tr key={index}>
            <td>{card.number}</td>
            <td>{card.name}</td>
            <td>{card.expiry}</td>
            <td>{card.cvc}</td>
          </tr>
        ))}
      </tbody>
```

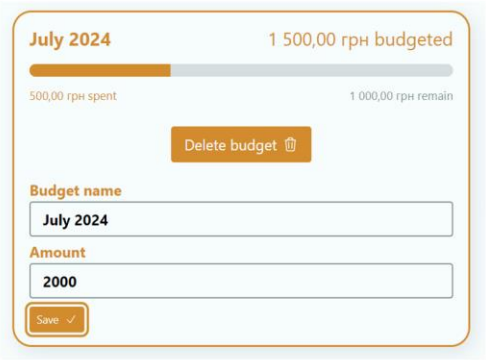
Рисунок Б.22 – Слайд презентації 22

Форма для редагування бюджету

```
export const updateBudget = (id, name, amount) => {
 const existingBudgets = JSON.parse(localStorage.getItem("budgets")) ||
[];

 const budgetIndex = existingBudgets.findIndex(budget => budget.id ===
id);


 if (budgetIndex !== -1) {
  existingBudgets[budgetIndex] = {
   ...existingBudgets[budgetIndex],
   name: name,
   amount: +amount,
  };
  localStorage.setItem("budgets", JSON.stringify(existingBudgets));
 } else {
  console.error("Budget not found");
 }
};
```



23

Рисунок Б.23 – Слайд презентації 23

Сторінка статистики

```
<Line
      data={{
       labels: revenueData.map((data) =>
data.label),
       datasets: [
        {
         label: "Incomes",
         data: revenueData.map((data) =>
data.income),
         backgroundColor: "#064FF0",
         borderColor: "#064FF0",
        },
        {
         label: "Expenses",
         data: revenueData.map((data) =>
data.expense),
         backgroundColor: "#FF3030",
         borderColor: "#FF3030",
        },
       ],
      }}
```

24

Рисунок Б.24 – Слайд презентації 24

Тестування

Передумова			
№	**Опис випадку**	**Очікуваний результат**	**Висновок**
1	Відкрити веб-застосунок	Користувач має доступ до відкритого сайту	Пройдено
2	Користувач хоче авторизуватись у системі	Перехід на головну сторінку	Пройдено
Створення бюджету			
№	**Опис випадку**	**Очікуваний результат**	**Висновок**
1	Відкриття форми створення бюджету	З'являється форма для додавання нового бюджету	Пройдено
2	Заповнити поле «Назва»	Введені дані коректні	Пройдено
3	Заповнити поле «Розмір»	Введено коректне число	Пройдено
4	Натиснути кнопку «Створити бюджет»	Виведення повідомлення про успішне створення бюджету та перехід на головну сторінку	Пройдено
Додавання витрати до бюджету			
1	Відкриття форми для додавання витрати	З'являється форма для додавання нової витрати	Пройдено
2	Заповнити поле «Категорія»	Введено коректну категорію	Пройдено
3	Заповнити поле «Сума»	Введено коректне число	Пройдено

25

Рисунок Б.25 – Слайд презентації 25

Висновки

1. Здійснено аналіз предметної галузі.
2. Проведено досліджееея існуючих рішень та визначено цільову аудиторію.
3. Сформульовано вимоги до системи.
4. Розроблено клієнтську частину програмної системи, що відповідає поставленим вимогам.
5. Протестовано програму за допомогою декількох видів тестувань.

24

Рисунок Б.26 – Слайд презентації 26