Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Впровадження компонентно-орієнтованої архітектури для створення додатку управління особистими фінансами з використанням React та Electron

Виконав(ла): студент(ка) VI курсу, групи СПм-61
спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

(підпис) Харлан А. А. (прізвище та ініціали)

Керівник (підпис) Петрик М. Р. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М. Р (прізвище та ініціали)

Рецензент (підпис) Сверстюк А. С. (прізвище та ініціали)

Тернопіль
2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет _____
(повна назва факультету)

Кафедра _____
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ _____
(підпис) (прізвище та ініціали)
« » 20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня _____
(назва освітнього ступеня)

за спеціальністю _____
(шифр і назва спеціальності)

студенту _____
(прізвище, ім'я, по батькові)

1. Тема роботи _____

Керівник роботи _____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 20___ року № _____

2. Термін подання студентом завершеної роботи _____

3. Вихідні дані до роботи _____

4. Зміст роботи (перелік питань, які потрібно розробити)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

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

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота на здобуття освітнього ступеню «магістр» за спеціальністю 121 – Інженерія програмного забезпечення. Тернопільський національний технічний університет ім. Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СПм-61, 2024 рік. Пояснювальна записка до кваліфікаційної роботи на здобуття освітнього ступеню «магістр» містить: 74 с., 30 рис., 3 додатки, 22 бібліогр.

Тема: Впровадження компонентно-орієнтованої архітектури для створення додатку управління особистими фінансами з використанням React та Electron.

В кваліфікаційній роботі магістра висвітлено ключові елементи у впровадженні компонентно-орієнтованої архітектури для створення додатку управління особистими фінансами з використанням технологій React та Electron, а також мови програмування TypeScript та платформи Firebase. Проаналізовано та оглянути предметну область розроблюваного додатку. Визначено можливості додатку. Розробено діаграми варіантів використання, класів та послідовностей. Зображено макети сторінок додатку. Створено додаток, що демонструє роботу управління особистими фінансами.

Ключові слова: додаток, React, Electron, TypeScript, додаток управління фінансами, Open Banking.

ANNOTATION

Qualification work for the educational degree ‘Master’ in speciality 121 - Software Engineering. Ternopil National Technical University named after Ivan Puluj, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, group CPm-61, 2024. The explanatory note to the qualification work for the master's degree contains: 74 p., 30 figures, 3 appendix, 22 ref.

Topic: Implementation of component-based architecture for creating a personal finance management application using React and Electron.

The master's thesis highlights the key elements in the implementation of component-oriented architecture for creating a personal finance management application using React and Electron technologies, as well as the TypeScript programming language and the Firebase platform. The subject area of the application is analysed and reviewed. The capabilities of the application are determined. Diagrams of use cases, classes and sequences are developed. The layouts of the application pages are shown. Created an application that demonstrates the work of personal finance management.

Keywords: application, React, Electron, TypeScript, financial management application, Open Banking.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API (Application Programming Interface) – прикладний програмний інтерфейс.

GUI (Graphical User Interface) – графічний інтерфейс користувача.

UML (Unified Modeling Language) – уніфікована мова моделювання.

Open Banking – концепція, що передбачає відкритий доступ до банківських даних третім сторонам.

КОА – компонентно-орієнтована архітектура.

MVC (Model-view-controller) – модель-вид-контролер.

DOM (Document Object Model) – об'єктна модель документа.

IDE (Integrated development environment) – інтегроване середовище розробки

ДНАОП – державні нормативні акти з охорони праці.

ПЗ – програмне забезпечення.

ПК – персональний комп'ютер.

ВСТУП

В теперішній час, де інформаційні технології динамічно розвиваються та охоплюють усі сфери життя, розробка програмного забезпечення для зручного та ефективного управління інформацією є надзвичайно важливою. Додатки для управління особистими фінансами набувають особливої актуальності, оскільки вони допомагають відстежувати доходи й витрати, планувати бюджет, встановлювати фінансові цілі та успішно їх реалізовувати.

Програмні рішення надають користувачам широкий спектр можливостей, які значно спрощують та автоматизують процес контролю над власними коштами. До таких можливостей належить автоматизація обліку доходів та витрат, що позбавляє необхідності ручного введення даних та мінімізує ризик помилок. Програмне забезпечення також надає можливість здійснювати детальний аналіз фінансових даних, виявляти тенденції та закономірності у витратах. Візуалізація інформації у вигляді графіків та діаграм робить дані більш зрозумілими та наочними, допомагаючи користувачам легко оцінити свій фінансовий стан.

В даній магістерській роботі відбудеться впровадження компонентно-орієнтованої архітектури для створення додатку з управління особистими фінансами з використанням сучасних технологій React та Electron. Поєднання компонентно-орієнтованої архітектури, React та Electron надає можливість створити зручний додаток, який відповідатиме потребам сучасних користувачів.

В рамках роботи планується провести аналіз предметної області, визначити основні вимоги до додатку, розробити його архітектуру та інтерфейс, реалізувати функціональність з використанням обраних технологій, провести тестування та оцінку якості створеного продукту. Результати дослідження продемонструють переваги використання даних технологій для створення гнучкого та ефективного програмного забезпечення.

Розроблений додаток буде корисний для всіх людей, що хочуть краще слідкувати за своїми фінансами та постійно мати цю інформацію при собі.

1 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ДОДАТКУ УПРАВЛІННЯ ФІНАНСАМИ

1.1 Огляд конкурентів

У сучасному швидкоплинному світі ефективне керування власними фінансами має вирішальне значення для досягнення фінансової стабільності та досягнення цілей. З розвитком технологій додатки для управління особистими фінансами стають дедалі популярнішими, пропонуючи користувачам зручні інструменти для відстеження витрат, створення бюджетів і планування на майбутнє. У цьому розділі буде розглянуто деякі з провідних додатків на ринку, проаналізовано їх функції, сильні та слабкі сторони [1].

Першим з конкурентів можна виділити один з найпопулярніших додатків такого типу – YNAB (рис. 1.1).

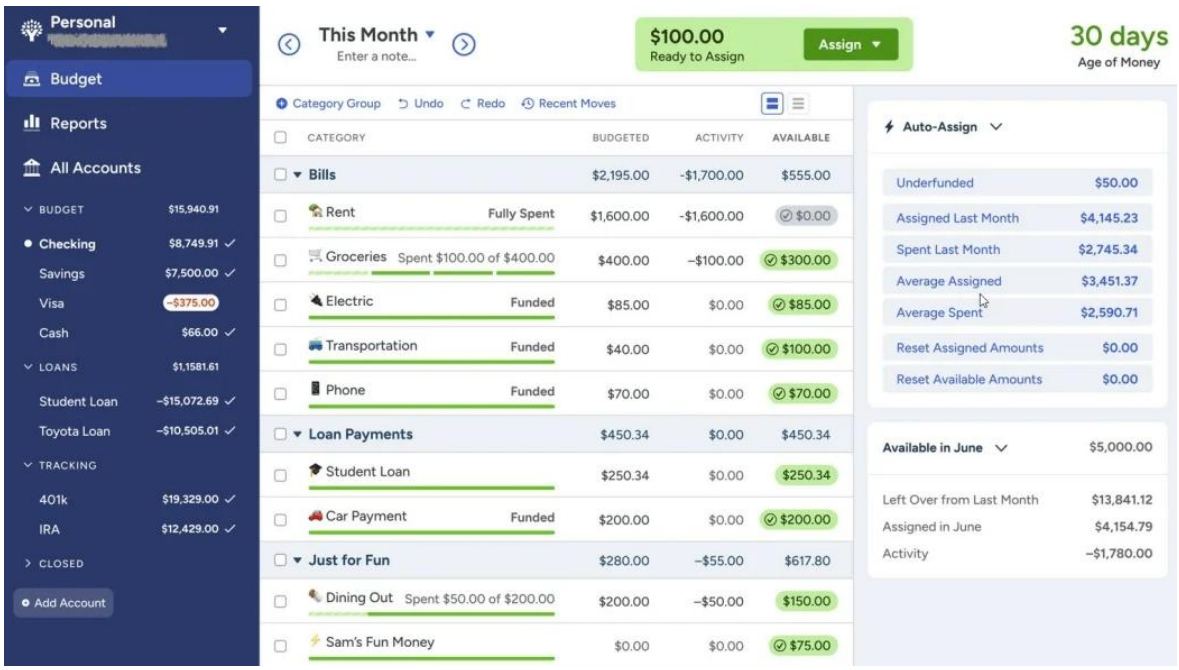

Рисунок 1.1 – Інтерфейс додатку YNAB

YNAB один з найпопулярніших додатків контролю особистими фінансами. Його сильні сторони полягають у комплексному підході до бюджетування, який

допомагає користувачам отримати контроль над своїми фінансами, визначаючи пріоритети витрат, встановлюючи чіткі фінансові цілі та виховуючи звички розумного витрачання коштів. Він має відносно високу абонентську плату порівняно з безкоштовними альтернативами, а його акцент на ручному введенні даних і детальному бюджетуванні може забирати багато часу, що потенційно може стати причиною поганого досвіду для певних користувачів. Крім того, YNAB не має комплексних функцій відстеження інвестицій, а підтримка клієнтів здійснюється переважно електронною поштою.

Наступним у списку конкурентом можна назвати Empower (рис. 1. 2).

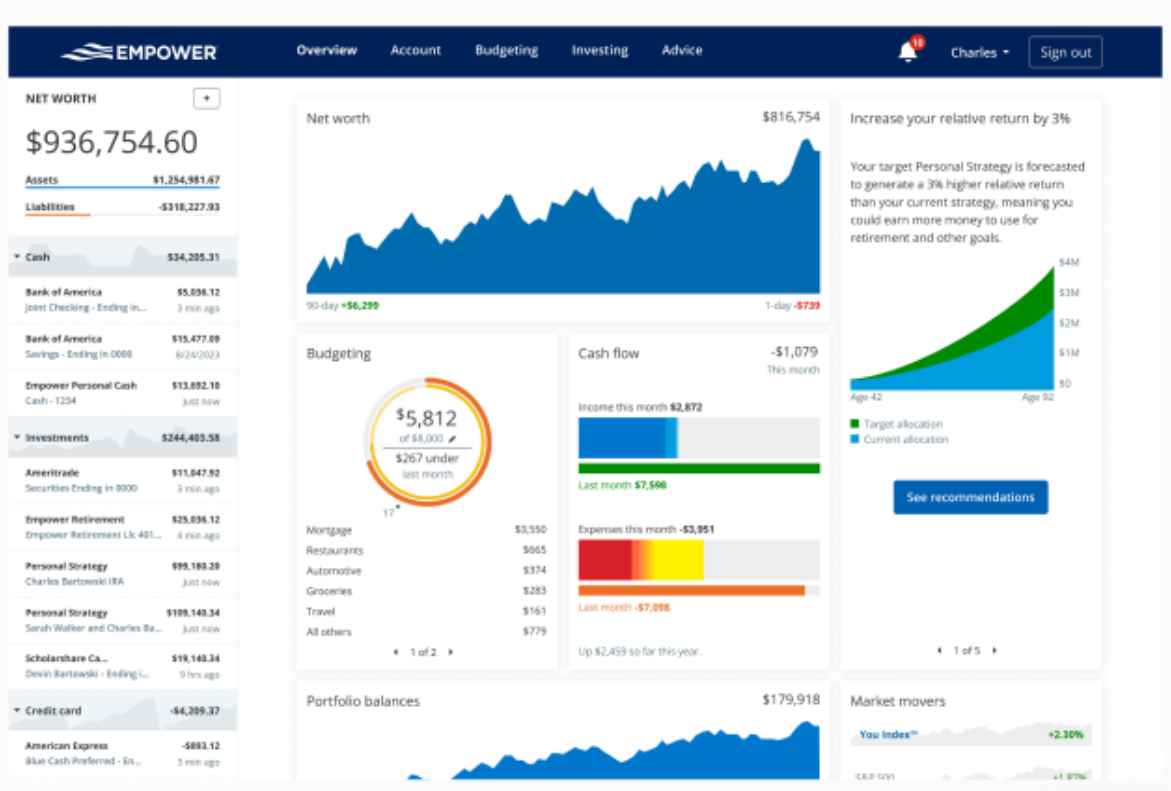

Рисунок 1.2 – Інтерфейс додатку Empower

Empower – це гібридний бот-порадник і додаток для управління особистими фінансами, який призначений для користувачів зі комплекснішими фінансовими ситуаціями, для тих, хто має інвестиції. Його сильні сторони полягають у складних функціях управління капіталом, включаючи відстеження інвестицій та пенсійне планування. Empower надає цілісне уявлення про фінанси, об'єднуючи всі рахунки

в одне, даючи чітку картину чистого капіталу та фінансового стану. Також пропонує доступ до фінансових консультантів для отримання індивідуальних рекомендацій. Основна увага приділяється управлінню інвестиціями, що може не підійти користувачам, які шукають поглиблене бюджетування або функції відстеження витрат. Багато з його розширених функцій, включаючи доступ до фінансових консультантів, вимагають платної підписки.

Simplifi (рис. 1.3) – це додаток для управління особистими фінансами, який прагне зробити процес контролю над грошима простішим та зрозумілішим. Він має зручний інтерфейс та пропонує інтуїтивно зрозумілі функції, такі як планування витрат, відстеження конкретних витрат за допомогою списків спостереження та встановлення фінансових цілей, що допомагає користувачам ефективно керувати своїми коштами. Simplifi також надає персоналізовану інформацію та фінансові звіти. Однак і має деякі обмеження. Його функції відстеження інвестицій не такі надійні, як у спеціалізованих інвестиційних додатках, і йому не вистачає деяких розширених функцій бюджетування.

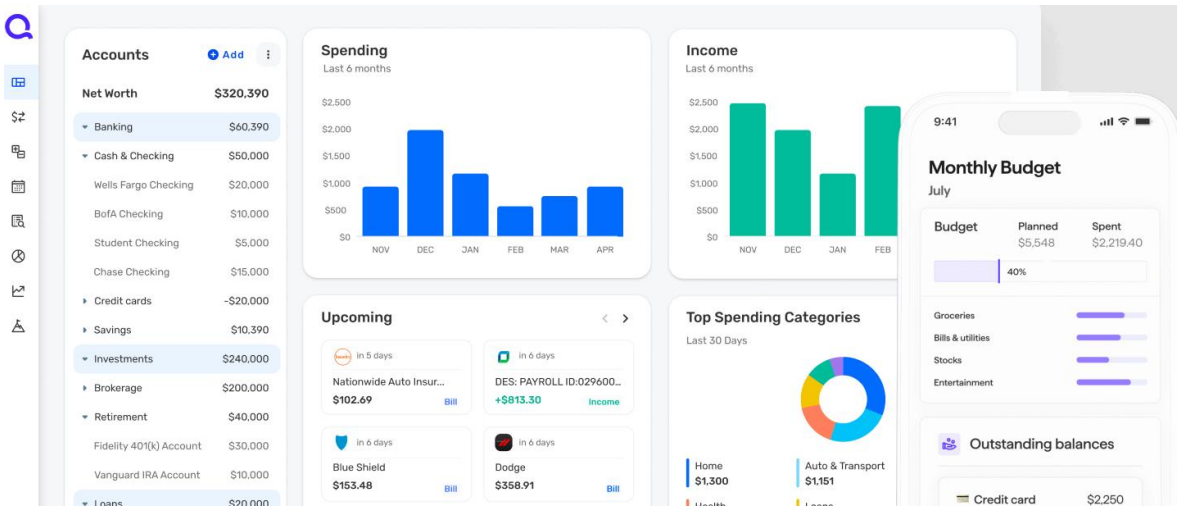

Рисунок 1.3 – Інтерфейс додатку Simplifi

Rocket Money (рис 1.4) допомагає користувачам контролювати свої витрати та заощаджувати гроші. Його сильні сторони полягають у інструментах управління підписками, які дозволяють користувачам легко виявляти та скасовувати небажані підписки. Також надає послуги з узгодження рахунків, де Rocket Money

домовляється про нижчі тарифи з постачальниками послуг від імені користувача. Додаток надає комплексний огляд фінансів, включаючи відстеження витрат, інструменти бюджетування та моніторинг кредитного рейтингу.

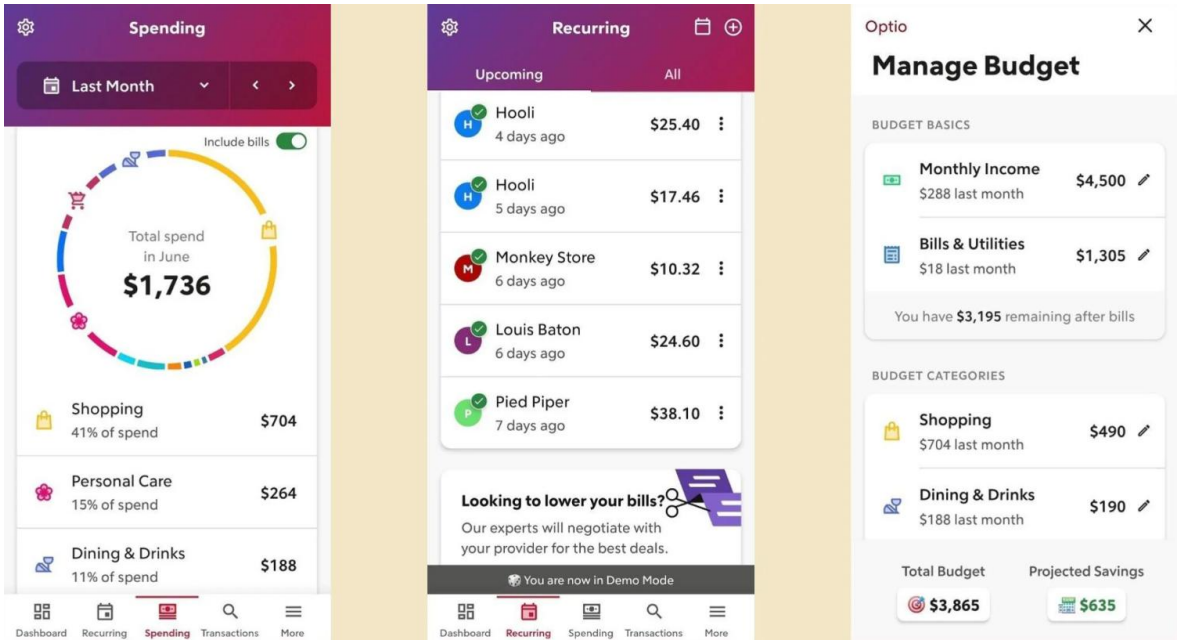

Рисунок 1.4 – Інтерфейс додатку Rocket Money

Monarch (рис 1.5) – це додаток для особистих фінансів, розроблений для можливості мати комплексний погляд на своє фінансове життя, включаючи бюджетування, відстеження чистого капіталу та управління інвестиціями. Його сильні сторони полягають в сучасному інтерфейсі, що робить його візуально привабливим і простим у використанні. Monarch пропонує надійні функції, такі як автоматична категоризація транзакцій, детальне відстеження чистого капіталу з історичними даними та детальні фінансові звіти, які допомагають користувачам зрозуміти структуру своїх витрат і прогрес у досягненні цілей. Він також підтримує кілька валют і пропонує надійні заходи безпеки для захисту даних користувачів. Однак Monarch має відносно високу вартість підписки порівняно з іншими додатками для особистих фінансів, що може стати бар'єром для деяких користувачів.

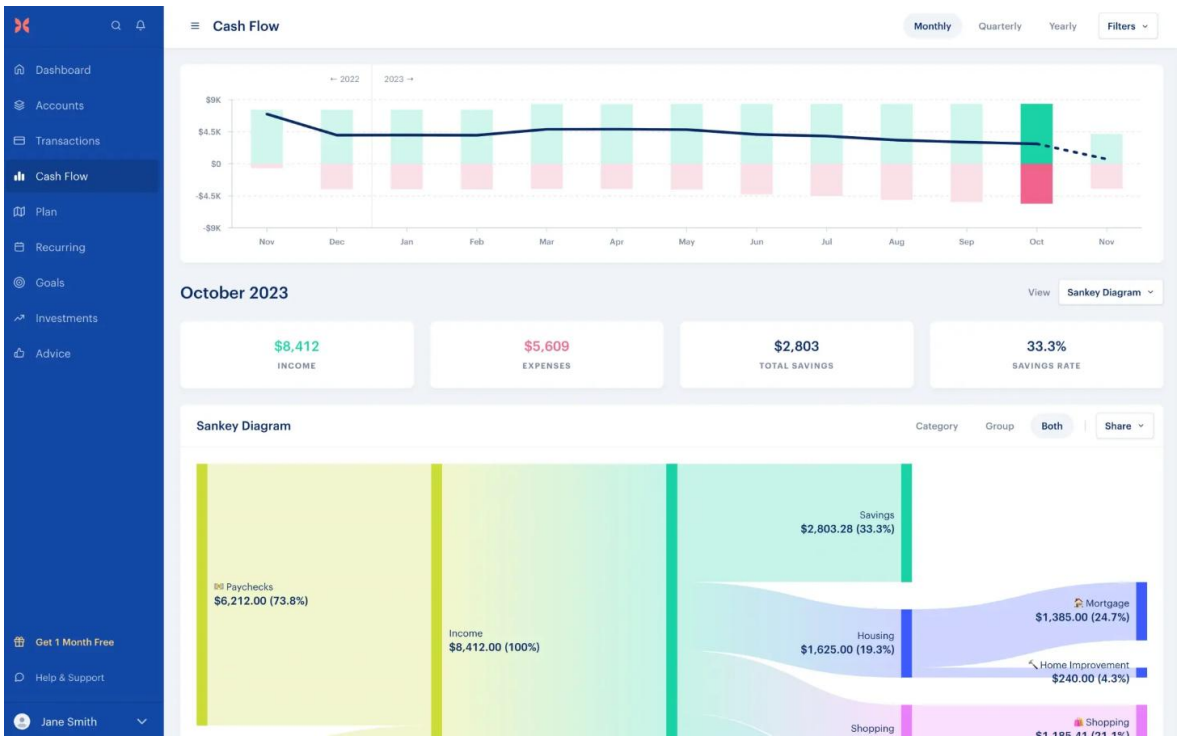

Рисунок 1.5 – Інтерфейс додатку Monarh

Розглянувши різноманітні додатки для управління особистими фінансами, стає зрозуміло, що кожен з них має свої унікальні переваги та недоліки. YNAB пропонує комплексний підхід до бюджетування, Empower зосереджений на управлінні інвестиціями, Simplifi спрощує управління грошима, Rocket Money допомагає контролювати витрати та заощаджувати, а Monarch надає комплексний погляд на фінансове життя.

1.2 Аналіз технології Open Banking

Окремо необхідно згадати технологію, до якої частина конкурентів має доступ – Open Banking. Це революційна концепція, яка змінює систему фінансів. По суті, це надання клієнту, більшого контролю над фінансовими даними і тим, як вони використовуються. Можливо легко об'єднати всі банківські рахунки користувача, кредитні картки та інвестиції в єдину платформу, що дозволить

відстежувати фінанси, здійснювати платежі та отримувати доступ до ширшого спектру фінансових послуг з безпрецедентною легкістю [2].

Традиційно банки ретельно охороняють фінансові дані у власних системах. Open Banking змінює це, вимагаючи від банків безпечно ділитися цими даними з уповноваженими сторонніми постачальниками, але тільки з прямої згоди клієнта. Це відбувається завдяки використанню інтерфейсів прикладного програмування (API), які діють як безпечні канали для передачі даних між банком та обраним стороннім провайдером.

Переваги відкритого банкінгу численні. Для споживачів це означає:

- цілісне уявлення про фінанси, що полегшує складання бюджету, відстеження витрат і визначення сфер для вдосконалення;
- сторонні провайдери можуть використовувати дані, щоб пропонувати індивідуальні фінансові продукти та поради;
- відкритий банкінг сприяє конкуренції та заохочує розробку нових інноваційних фінансових послуг;
- оптимізовані процеси для таких завдань, як подання заявок на кредити або здійснення платежів.

Для бізнесу Open Banking відкриває такі можливості:

- більш персоналізовані та зручні послуги;
- доступ до більш багатих наборів даних дозволяє краще зрозуміти потреби та вподобання клієнтів, що є основою для розробки продуктів та маркетингових стратегій;
- інноваційні фінансові продукти та послуги, які орієнтовані на нішеві ринки або пропонують унікальні ціннісні пропозиції.

Однак відкритий банкінг не позбавлений викликів. Безпека даних і конфіденційність є першочерговими питаннями, і для захисту інформації від несанкціонованого доступу та зловживань повинні бути створені надійні засоби захисту. Крім того, регуляторне середовище, що оточує Open Banking, все ще розвивається.

Національний банк України в 2023 році зробив важливий крок на шляху до модернізації фінансового ландшафту країни, затвердивши концепцію відкритого банкінгу. Цей комплексний документ, розроблений у співпраці з ключовими стейкхолдерами платіжного ринку, окреслює дорожню карту та основні вимоги для впровадження відкритого банкінгу в Україні. Цей крок свідчить про прагнення України привести свої фінансові послуги у відповідність до європейських стандартів, сприяти інноваціям та покращувати якість обслуговування клієнтів.

У вересні 2024 року UPC запустила першу в Україні платформу Open Banking. UPC тісно співпрацює з Національним банком та банківською спільнотою з метою розробки єдиного API, який має значно покращити ефективність роботи фінансових установ та якість обслуговування клієнтів.

Негативна сторона цього, що Open Banking в даному додатку можливо буде використати лише в майбутньому, коли необхідне API від сторонніх сервісів буде доступне широкому загалу.

1.3 Обґрунтування вибору напрямку дослідження

Для створення додатку для управління особистими фінансами було вирішено обрати формат десктоп програми з використанням React та Electron з ОС Windows, а також мову програмування TypeScript

Вибір дослідження реалізації компонентно-орієнтованої архітектури додатку для управління особистими фінансами з використанням React та Electron обґрунтований кількома переконливими факторами. Управління особистими фінансами – це сфера, яка за своєю суттю виграє від модульності та багаторазового використання. Фінансові додатки часто включають окремі функції, такі як бюджетування, відстеження транзакцій, управління інвестиціями та звітність. Компонентно-орієнтований підхід дозволяє інкапсулювати ці функціональні можливості в незалежні компоненти, що підвищує зручність обслуговування та

гнучкість. Це особливо важливо для домену, який часто оновлюється та змінюється відповідно до потреб користувачів [3].

Крім того, цей напрямок досліджень відповідає практичним потребам ринку. Незважаючи на те, що існує безліч додатків для персональних фінансів, багато з них страждають від проблем, пов'язаних зі складністю, ремонтопридатністю та користувацьким досвідом.

Далі необхідно детальніше розглянути інструменти, що будуть використовуватись при створенні додатку.

ReactJS – це відома бібліотека JavaScript для створення користувацьких інтерфейсів, зокрема односторінкових додатків. Вона була створена Facebook, а зараз підтримується Meta та спільнотою індивідуальних розробників і компаній.

React-додатки будуються з використанням компонентів, які є частинами інтерфейсу, що можна використовувати повторно. Кожен компонент керує власним станом та логікою рендерингу, і їх можна компонувати разом для створення складних інтерфейсів. React використовує синтаксичне розширення до JavaScript – JSX, яке дозволяє писати HTML-подібні структури у JavaScript-коді, що робить його більш читабельним. React створює віртуальне представлення реального DOM. Дані в React рухаються в одному напрямку, від батьківських компонентів до дочірніх, що полегшує розуміння того, як відбувається передача даних. Компоненти можна використовувати повторно у всьому додатку або навіть у різних проектах. React має відносно хороший поріг навчання та велику і активну спільноту, яка надає багато ресурсів [4].

Його сильними сторонами є:

- віртуальний DOM та компонентна архітектура роблять React-додатки швидкими та ефективними;
- створив один раз – використовуй будь-де. Компоненти сприяють багаторазовому використанню коду та його підтримці;
- React можна рендерити на сервері, що покращує видимість додатку в пошукових системах;
- за допомогою React можна створювати додатки на різних платформах.

Слабкі сторони:

- деякі розробники вважають, що поєднання HTML та JavaScript у JSX спочатку збиває з пантелику;
- екосистема React дуже швидко розвивається, що вимагає від розробників постійно слідкувати за новими оновленнями;
- React в першу чергу фокусується на рівні інтерфейсу користувача. Що означає потребу у додаткових бібліотеках для таких речей, як управління станами та маршрутизація;
- для дуже маленьких проектів React може бути надмірним через його налаштування.

Наступним інструментом для розробки додатку з використанням React на пк було взято Electron.

Electron є фреймворком з відкритим кодом від GitHub, який надає можливість розробникам створювати багатоплатформні рішення для настільних комп'ютерів, використовуючи знайомі веб-технології. Він поєднує в собі Chromium, рушій рендерингу, який стоїть за Google Chrome, і Node.js, середовище виконання JavaScript. Це означає, що можливо використовувати свої навички веб-розробки для створення програм, що працюють на Windows, macOS і Linux без необхідності писати окремий код для кожної платформи. Electron досягає цього, включаючи Chromium і Node.js у свій дистрибутив, що дозволяє підтримувати єдину кодову базу JavaScript. Він також надає доступ до API операційної системи через Node.js, дозволяючи використовувати такі функції, як інтеграція з системним треєм, налаштування меню і доступ до файлової системи. Хоча додатки Electron можуть бути дещо ресурсомісткими, його простота використання, кросплатформна сумісність і доступ до величезної екосистеми веб-технологій роблять його чудовим вибором для розробки настільних додатків [5].

Позитивними сторони Electron є:

- використання та розробка на всіх платформах (Windows, Linux, macOS);

- можливість використання наявних навичок веб-розробки для розробки додатків для ПК;
- цикли створення прототипів та розробки є швидшими в порівнянні з нативними додатками;
- активна спільнота та широкі ресурси для підтримки та усунення несправностей;
- автоматичні оновлення;
- інтеграція з функціями ОС через Node.js.

Негативні сторони:

- може бути ресурсоємним, особливо для складних додатків;
- Chromium у комплекті збільшує загальний розмір додатку;
- потенційні вразливості, якщо не захищений належним чином (особливо з інтеграцією Node.js);
- не зовсім нативний вигляд і відчуття, може не повністю імітувати нативний інтерфейс користувача на кожній платформі;
- може споживати значний обсяг пам'яті через вбудований екземпляр браузера.

TypeScript – це потужна мова програмування, що розширює можливості JavaScript, додаючи підтримку статичної типізації. Завдяки цьому розробники можуть заздалегідь визначати типи змінних, параметрів функцій та властивостей об’єктів. Це сприяє виявленню помилок ще на етапі компіляції та спрощує подальшу підтримку й масштабування коду [6].

TypeScript, хоча і є хорошою мовою, має як сильні, так і слабкі сторони. Ось деякі з них:

Сильні сторони:

- статична типізація допомагає виявляти помилки на ранніх стадіях, що полегшує рефакторинг та підтримку коду, особливо у великих проектах;
- TypeScript пропонує кращі можливості завершення коду, навігації та рефакторингу в IDE, що призводить до підвищення продуктивності;

- перевірка типів допомагає запобігти помилкам при виконання, що часто зустрічаються в JavaScript, що призводить до підвищення надійності додатків;
- анотації типів полегшують розуміння призначення та використання змінних і функцій;
- можливість поступово впроваджувати TypeScript в існуючі JavaScript-проекти, що полегшує перехід на нову мову.

Слабкі сторони:

- анотації типів можуть додавати додатковий код, збільшуючи розмір файлів і потенційно сповільнюючи розробку на початковому етапі;
- система типів TypeScript має обмеження, і деякі типові помилки все ще можуть прослизнути при виконанні;
- потребує етапу компіляції для перетворення коду TypeScript в JavaScript, що може ускладнити процес збірки;
- у невеликих проектах переваги TypeScript можуть не переважити додаткову складність.

TypeScript добре сумісний як з React, так і з Electron. Для React TypeScript надає чудову підтримку JSX, синтаксичного розширення, яке дозволяє писати HTML-подібні структури в JavaScript-коді. TypeScript може виводити типи в JSX, забезпечувати завершення коду та перевірку помилок, роблячи React-розробку більш надійною.

У контексті Electron, TypeScript може допомогти писати більш надійний код як для основного процесу, так і для процесу рендерингу. Визначаючи типи для API Electron, допомагає запобігти поширеним помилкам і забезпечити коректну взаємодію коду з фреймворком Electron.

Використання TypeScript у проекті принесе значні переваги. Це покращить якість коду, зменшить кількість помилок та покращить досвід розробки. Комбінація TypeScript, React і Electron утворює сильний і сучасний технологічний стек, що дозволяє створювати якісні та функціональні настільні додатки.

Для реалізації проєкту буде обрано FireBase як платформу для створення бекенд-частини. Це рішення обумовлене рядом переваг, які надає FireBase, зокрема його модульність, легкість інтеграції з фронтенд-технологіями та готовий набір інструментів для побудови сучасних веб- і десктопних додатків [7].

FireBase забезпечить функціональність для автентифікації користувачів, що дозволить реалізувати безпечний доступ до персональних даних, використовуючи сучасні методи, такі як авторизація через електронну пошту, соціальні мережі або двофакторну аутентифікацію. Крім того, FireStore, як одна з ключових складових FireBase, стане основою для організації бази даних. Її структура, побудована на основі NoSQL, дозволить ефективно зберігати та обробляти складну ієрархічну інформацію, пов’язану з категоріями витрат, бюджетами та звітами.

Також використання FireBase забезпечить можливість реального часу синхронізації даних між користувачами. Це дозволить створювати сучасні інтерактивні сценарії використання, наприклад, ведення спільного бюджету або динамічне відображення змін у фінансових показниках.

Для редагування та написання коду проєкту було обрано Visual Studio Code, безкоштовне середовище розробки ПЗ.

Має відкритий вихідний код, створений Microsoft. Завдяки своїй багатофункціональності, зручності та універсальності, він здобув велику популярність серед розробників. Незважаючи на свою легкість, це середовище є потужним інструментом, що підтримує безліч мов програмування та фреймворків.

Однією з ключових переваг цього середовища є його кастомізація. Присутня можливість персоналізувати майже кожен аспект редактора, від тем і комбінацій клавіш до розширень, які додають підтримку мов, інструменти налагодження та інтеграцію з іншими сервісами. Така гнучкість дозволяє розробникам пристосовувати VS Code до своїх конкретних потреб та вподобань.

Це хороший вибір для веб-розробки, так як Visual Studio Code має вбудовану підтримку JavaScript, TypeScript та Node.js. Він також включає такі функції, як IntelliSense, що забезпечує розумне закінчення коду та відповідні пропозиції, а

також інструменти налагодження, які сприяють виявленню та виправленню помилок у коді.

1.4 Технічний аспект проблеми

Враховуючи, що створюється додаток для управління особистими фінансами з акцентом на компонентно-орієнтовану архітектуру з використанням React та Electron, для проекту найкраще підійде гнучка методологія, зокрема Scrum [8]. Ось чому:

- Scrum наголошує на розбитті розробки на короткі ітерації, які називаються спринтами. Це дозволяє зосередитися на створенні та тестуванні окремих компонентів або функцій у кожному спринті, гарантуючи, що вони функціонують належним чином та добре інтегруються із загальною системою. Це має вирішальне значення для компонентно-орієнтованого підходу, коли окремі компоненти повинні безперебійно працювати разом;
- додатки для персональних фінансів часто потребують коригування на основі відгуків користувачів та потреб, що змінюються. Ітеративний характер Scrum дозволяє швидко адаптуватися до змін і включати нові вимоги або функції в міру їх появи;
- сприяє ефективній взаємодії між розробниками, дизайнерами та зацікавленими сторонами. Це має особливе значення для проєкту, оскільки важливо гарантувати, що компоненти спроєктовані та реалізовані відповідно до потреб користувачів і гармонійно інтегруються в загальну архітектуру;
- наголошує на регулярних оглядах та ретроспективах, надаючи можливість збирати відгуки про розроблені компоненти та визначати сфери для покращення. Цей ітеративний цикл зворотного зв'язку

гарантує, що додаток відповідає очікуванням користувачів, а архітектура компонентів залишається надійною та ефективною.

Реалізація програми для управління особистими фінансами може передбачати модульну структуру, в якій основні функціональні можливості інкапсульовані в незалежні компоненти. Наприклад, окремі компоненти для управління рахунками, відстеження транзакцій, бюджетування, звітності та відстеження інвестицій. Кожен компонент відповідатиме за управління власними даними та логікою, взаємодіючи з іншими компонентами через чітко визначені інтерфейси [9].

Наприклад, компонент відстеження транзакцій може надавати функції для додавання, редагування та категоризації транзакцій, тоді як компонент звітності може отримати доступ до цих даних для створення візуалізацій та зведень про звички витрачання коштів. Така модульність дозволяє незалежну розробку, тестування та підтримку кожного з компонентів.

Компонентна модель React природно узгоджується з цією архітектурою. Можливість створювати багаторазові UI-компоненти для таких завдань, як відображення списків транзакцій, створення бюджетних форм або рендеринг діаграм. Ці компоненти можна компонувати для побудови складних користувацьких інтерфейсів, сприяючи узгодженості та зручності обслуговування. Electron забезпечує основу для пакування цих компонентів у десктопний додаток з доступом до можливостей операційної системи.

Electron дозволяє взаємодіяти з операційною системою на рівні її базових функцій, що уможливлює такі функції, як системні сповіщення про нагадування про рахунки або інтеграцію з файловою системою для імпорту та експорту даних.

Було визначено основні принципи КОА, такі як модульність, чітко визначені інтерфейси та взаємодія компонентів. Також було проаналізовано переваги використання React та Electron для створення сучасних та функціональних користувацьких інтерфейсів.

2 РОЗРОБКА МОДЕЛІ ТА ПРОГРАМНОГО КОМПЛЕКСУ

2.1 Проєктування додатку для управління особистими фінансами

Створення програми з використанням компонентно-орієнтованої архітектури передбачає особливий підхід до проектування та розробки програмного забезпечення. КОА наголошує на створенні самодостатніх, багаторазових компонентів, які інкапсулюють певну функціональність і дані. Ці компоненти взаємодіють один з одним через чітко визначені інтерфейси, що сприяє модульності та гнучкості.

З технічної точки зору, розробка додатку КОА починається з визначення основних функціональних одиниць системи і проектування їх як незалежних компонентів. Потім кожен компонент будується як єдине ціле, що потенційно може охоплювати різні технології та мови програмування, якщо вони дотримуються визначеного інтерфейсу. Цей інтерфейс слугує контрактом, що визначає, як компонент взаємодіє з рештою системи [10].

Взаємодія компонентів є ключовим аспектом КОА. Цього можна досягти за допомогою різних механізмів, таких як виклик методів, передача повідомлень або архітектура, керована подіями. Вибір механізму залежить від конкретних потреб програми та складності взаємодії.

Іншим ключовим моментом є управління компонентами. Це передбачає вирішення таких питань, як життєвий цикл компонента (створення, ініціалізація, знищення), керування версіями та розгортання. Фреймворки та інструменти, спеціально розроблені для компонентно-орієнтованої розробки, можуть допомогти в управлінні цими аспектами, надаючи такі функції, як виявлення, реєстрація та вирішення залежностей компонентів.

Крім того, КОА часто використовує такі методи, як пізнє зв'язування та рефлексія для підвищення гнучкості та адаптивності. Пізнє зв'язування дозволяє виявляти та інтегрувати компоненти під час виконання, тоді як рефлексія дає змогу аналізувати можливості компонентів.

По суті, розробка додатків з КОА вимагає переходу до модульного мислення, зосередження на чітко визначених інтерфейсах і незалежних компонентах. Такий підхід сприяє багаторазовому використанню, підтримці та еволюції програмних систем.

Далі було обрано архітектурний шаблон, що буде використовуватись під час розробки, ним став модель-вид-контролер.

Модель-вид-контролер – це широко прийнятий архітектурний патерн програмного забезпечення, який сприяє розділенню завдань у додатку. Він розділяє додаток на три взаємопов'язані частини: модель, представлення і контролер. Модель представляє дані та бізнес-логіку додатку, представлення відповідає за представлення даних користувачеві, а контролер обробляє взаємодію з користувачем і відповідно оновлює модель. Таке розділення завдань покращує організацію коду, зручність супроводу та тестування [11].

У контексті дипломної роботи з реалізації компонентно-орієнтованої архітектури для додатку для управління особистими фінансами з використанням React та Electron, MVC може доповнити та покращити дизайн. У той час як компонентно-орієнтована архітектура фокусується на модульності та багаторазовому використанні, MVC забезпечує структуру для управління потоком даних та взаємодією з користувачами в межах кожного компонента.

Наприклад, у компоненті, що відповідає за управління транзакціями, модель буде представлена структурами даних та логікою для зберігання та маніпулювання деталями транзакцій. Вигляд – це React-компоненти, що відповідають за відображення списку транзакцій та форм вводу. Контролер обробляв би дії користувача, такі як додавання, редагування або видалення транзакцій, оновлення моделі, а згодом і представлення.

Сам React природно узгоджується з частиною View в MVC, забезпечуючи декларативний спосіб визначення користувацького інтерфейсу на основі стану додатку. Використання методів життєвого циклу компонентів React для управління оновленнями та взаємодією з моделлю і контролером.

Інтегруючи принципи MVC у компонентно-орієнтовану архітектуру, є можливість покращити організацію та супроводжуваність додатку. Кожен компонент може бути структурований з чітким розподілом завдань, що полегшує його розуміння, тестування та модифікацію.

MVC також сприяє покращенню тестованості додатку. Завдяки розділенню на модель, представлення та контролер, можна створювати окремі тести для кожного компонента, забезпечуючи їхню коректну роботу як самостійно, так і в поєднанні з іншими.

В той час як компонентно-орієнтована архітектура забезпечує основу для створення модульних і багаторазових компонентів, включення принципів MVC може ще більше покращити додаток для управління особистими фінансами.

Ось як архітектура MVC буде реалізована у додатку для управління особистими фінансами:

Модель:

- майбутні класи TypeScript представляють модель даних. Вони відповідають за зберігання та обробку даних додатку;
- методи всередині кожного класу реалізують бізнес-логіку додатку.
- моделі будуть взаємодіяти з Firebase для збереження, отримання та оновлення даних. TypeScript може використовувати бібліотеку Firebase для JavaScript для доступу до API Firebase.

Вигляд:

- компоненти інтерфейсу, реалізовані за допомогою React будуть використовуватися для створення інтерфейсу користувача додатку. Компоненти React будуть відображати дані з моделі та реагувати на дії користувача;
- Electron дозволить створити настільний додаток з використанням веб-технологій.

Контролер:

- контролери будуть реалізовані як функції TypeScript. Вони будуть відповідати за обробку подій від користувача в інтерфейсі, оновлення

моделі на основі дій користувача, виклик відповідних методів моделі для виконання бізнес-логіки та інші події.

Після визначення та опису необхідної архітектури та патерну потрібно визначити функціональні та нефункціональні вимоги додатку.

Функціональні вимоги додатку:

- реєстрація та вхід користувачів, тобто додаток повинен надавати змогу створювати облікові записи та безпечної авторизації користувачів у системі;
- відстеження та додавання транзакцій, користувачі повинні мати змогу вводити дані про свої доходи та витрати, розподіляючи їх за відповідними категоріями;
- створення та управління бюджетом: додаток має дозволяти користувачам планувати свій бюджет, встановлюючи ліміти витрат для різних категорій;
- перегляд фінансової історії: користувачам слід забезпечити можливість переглядати свою фінансову історію за різні періоди часу;
- додаток має генерувати звіти про доходи, витрати, баланс та інші фінансові показники;
- додаток має використовувати графіки та діаграми для візуалізації фінансових даних;
- додаток має враховувати та відстежувати комунальні послуги та підписки на різні сервіси;
- можливість створення та відстеження боргів, кредитів та позик.
- налаштування профілю користувача (валюта, мова, адреса, зміна паролю тощо);
- додаток може дозволяти користувачам ставити фінансові цілі та відстежувати прогрес;
- експорт всіх наявних даних.

Нефункціональні вимоги додатку:

- додаток повинен запускатися та працювати швидко, без помітних затримок;
- операції, такі як додавання транзакції, генерація звіту або завантаження даних, повинні виконуватися швидко;
- додаток повинен мати надійну систему аутентифікації для захисту даних користувачів;
- конфіденційні дані, такі як паролі та фінансова інформація, повинні бути зашифровані;
- додаток повинен мати простий та зрозумілий інтерфейс, який легко використовувати;
- користувачі повинні легко знаходити потрібні функції та інформацію;
- додаток повинен працювати стабільно, без збоїв та помилок.

Коли буде така необхідність, функціональні та нефункціональні вимоги додатку будуть змінюватись та доповнюватись. Необхідно постійно працювати над моделю предметної області для забезпечення актуальності в швидкоплинному світі.

Гнучкість додатку для управління особистими фінансами, побудованого на React, Electron, TypeScript та Firebase, буде проявлятися в кількох аспектах:

Адаптивний інтерфейс:

- Electron дає змогу розробляти додатки, які працюють на різних операційних системах, усуваючи потребу створювати окремий код для кожної платформи;
- компоненти React можна легко адаптувати до різних розмірів екрану, роблячи перегляд зручнішим на різних типах пристроїв.

Розширюваність:

- архітектура MVC сприяє модульності коду. Це дозволяє легко додавати нові функції та можливості до додатку в майбутньому, не впливаючи на існуючий код;
- завдяки використанню TypeScript та відкритого API Firebase, додаток може бути легко інтегрований з іншими сервісами та API, наприклад,

банківськими API для автоматичного імпорту транзакцій, сервісами платіжних систем, інвестиційними платформами.

Офлайн доступ:

- Electron дозволяє кешувати дані локально, що забезпечує доступ до основних функцій додатку навіть без підключення до Інтернету;
- при відновленні підключення до Інтернету, додаток може синхронізувати локальні дані з Firebase, забезпечуючи актуальність інформації.

Було проведено аналіз вимог до додатку управління особистими фінансами та визначено ключові аспекти її проєктування. Було обрано використання шаблону модель-вид-контролер та детально описано принципи КОА та MVC, їх переваги та взаємодію в контексті розробки платформи.

2.2 Розробка бізнес моделі додатку

Діаграма варіантів використання – це поведінкова діаграма на UML, яка візуально представляє взаємодію між користувачами, так званими акторами, і системою для досягнення певних цілей або функціональних можливостей. Вона надає високорівневе уявлення про поведінку системи з точки зору користувача, фокусуючись що робить система, а не на тому, як вона це робить [12].

По суті, діаграма варіантів використання зображує різні способи взаємодії акторів з системою, фіксуючи функціональні можливості, які пропонує система, та акторів, які залучені до кожної функціональності. Кожен варіант використання представляє конкретну дію або мету, яку актор може виконати за допомогою системи, наприклад, увійти в систему, здійснити покупку або створити звіт.

Діаграми варіантів використання корисні на різних етапах розробки ПЗ. На час збору вимог вони допомагають зафіксувати і задокументувати потреби користувачів і функціональні можливості системи. Під час проектування та аналізу

вони слугують основою для подальшого моделювання та визначення поведінки системи.

Створена діаграма варіантів використання майбутнього додатку зображена на рисунку 2.1.

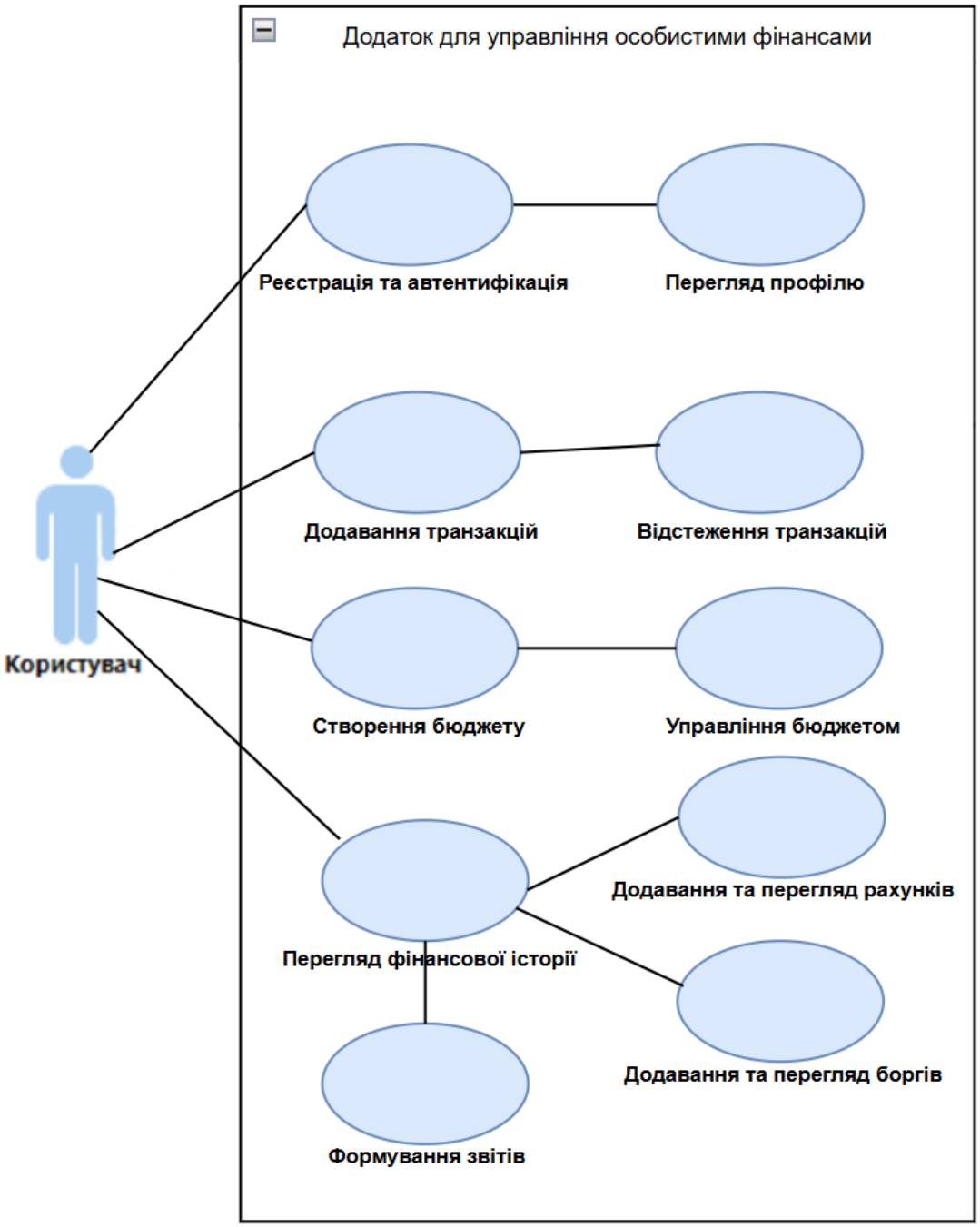

Рисунок 2.1 – Діаграма ВВ додатку

Ключові елементи діаграми варіантів використання:

- користувачі є зовнішніми сутностями, які взаємодіють із системою;

- варіанти використання описують функції або можливості, що система надає акторам;
- взаємозв'язки або стрілки, з'єднують актора з варіантами використання, вказуючи функціональні можливості.

Далі відбулось формування необхідних для діаграми сценаріїв опису варіантів використання.

Сценарії основних потоків подій для додатку для управління особистими фінансами:

1. Реєстрація та автентифікація

Основний сценарій подій:

1) Користувач відкриває додаток.
2) Користувач обирає "Реєстрація".
3) Користувач вводить дані (ім'я, email, пароль).
4) Користувача авторизовано

Альтернативний сценарій подій:

1) Користувач обирає "Автентифікація".
2) Користувач вводить email та пароль.
3) Користувача авторизовано

2. Додавання транзакцій

Основний сценарій подій:

1) Користувач обирає "Додати транзакцію".
2) Користувач обирає тип (дохід/витрата).
3) Користувач вводить суму, дату та категорію.
4) Користувач додає опис транзакції.

Альтернативний сценарій подій:

1) Користувач відміняє додавання транзакції.

3. Відстеження транзакцій

Основний сценарій подій:

1) Користувач обирає "Переглянути транзакції".
2) Система показує перелік транзакцій.

3) Користувач може фільтрувати транзакції за датою, категорією, типом.

4. Перегляд профілю

Основний сценарій подій:

1) Користувач обирає "Профіль".
2) Система відображає інформацію профілю (ім'я, email).
3) Користувач може редагувати інформацію профілю.
4) Користувач може змінити пароль.

5. Створення бюджету

Основний сценарій подій:

1) Користувач обирає "Створити бюджет".
2) Користувач обирає період (місяць, рік).
3) Користувач встановлює ліміти для кожної категорії.
4) Користувач натискає "Зберегти".
5) Система зберігає бюджет.

6. Управління бюджетом

Основний сценарій подій:

1) Користувач обирає "Переглянути бюджет".
2) Система відображає поточний бюджет.
3) Користувач може редагувати ліміти бюджету.
4) Користувач може переглядати прогрес виконання бюджету.

7. Перегляд фінансової історії

Основний сценарій подій:

1) Користувач обирає "Головна сторінка".
2) Користувач обирає табличку з транзакціями.

8. Додавання та перегляд рахунків

Основний сценарій подій:

1) Користувач обирає "Рахунки" в головному меню.
2) Система показує перелік наявних рахунків.
3) Користувач обирає опцію "Додати рахунок".
4) Система показує форму для введення даних рахунку.

5) Користувач вводить дані рахунку та обирає кнопку "Зберегти".
6) Система здійснює збереження рахунку та оновлює список рахунків.
7) Користувач може переглядати деталі рахунку, натиснувши на нього в списку.

9. Додавання та перегляд боргів

Основний сценарій подій:

1) Користувач обирає "Борги" в головному меню.
2) Система показує перелік наявних боргів.
3) Користувач обирає кнопку "Додати борг".
4) Система показує форму для введення даних боргу.
5) Користувач вводить дані боргу та обирає кнопку "Зберегти".
6) Система зберігає інформацію про борг та оновлює перелік боргів.
7) Користувач може переглядати деталі боргу, натиснувши на нього в списку.
8) Користувач може відмічати борг як сплачений.

10. Формування звітів

Основний сценарій подій:

1) Користувач обирає "Звіти" в головному меню.
2) Система показує доступні типи звітів.
3) Користувач обирає тип звіту.
4) Система показує форму з параметрами звіту.

Таким чином, було розроблено бізнес-модель додатку для управління особистими фінансами. Основним інструментом для моделювання стала діаграма варіантів використання, яка дозволила наочно відобразити взаємодію користувачів із системою та визначити основні функціональні можливості додатку. Розроблені сценарії дозволяють не лише структурувати функціональні вимоги до додатку, але й забезпечити зручність користування для кінцевих користувачів.

2.3 Проєктування архітектури

Діаграма класів – це візуальне представлення різних класів у системі та їх взаємозв'язків. Вона як план об'єктно-орієнтованої програми. Кожен клас схожий на шаблон, що визначає характеристики (атрибути) та поведінку (методи) своїх об'єктів.

На діаграмі класів є блоки, що представляють ці класи. Кожна рамка зазвичай складається з трьох частин: зверху – назва класу, посередині – його атрибути, а знизу – методи. Лінії між цими блоками ілюструють взаємозв'язки між класами. Ці зв'язки можуть бути такими, як успадкування (один клас є спеціалізованим типом іншого) або асоціація (один клас використовує або взаємодіє з іншим) [13].

Проаналізувавши компоненти та архітектуру для даного додатку, що розробляється з допомогою TypeScript, було створено діаграму класів, що вичерпно показує та описує взаємодію частин програми (рис. 2.2).

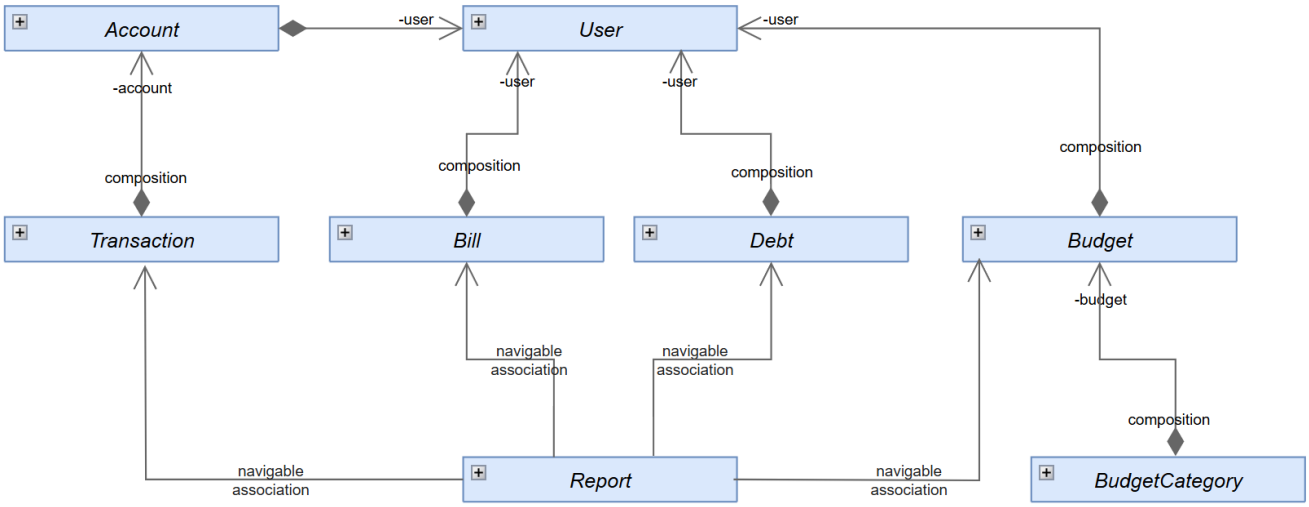

Рисунок 2.2 – Діаграма класів додатку

Клас User забезпечує зберігання інформації про користувача, включаючи його ім'я, прізвище, email, пароль, а також інші налаштування. Має атрибути, такі

як id, firstName, lastName, email, password, address, birthDate та інші. Містить наступні методи:

- login(email, password) аутентифікує користувача за email та паролем.
- logout() завершує сеанс користувача.
- updateProfile(profileData) оновлює профіль користувача з новими даними.
- changePassword() змінює пароль користувача.

Клас Account забезпечує зберігання інформації про фінансовий рахунок користувача, включаючи назву рахунку, тип рахунку, баланс та валюту. Має атрибути accountName, accountType, balance, currency. Містить методи:

- getAccountDetails() повертає детальну інформацію про рахунок (баланс, валюта, транзакції тощо).
- getAccountTransactions повертає список транзакцій, пов'язаних з рахунком.
- getBalance() повертає поточний баланс рахунку.

Клас Transaction забезпечує зберігання інформації про фінансові транзакції, включаючи тип транзакції , суму, дату, категорію, опис та метод оплати. Атрибути transactionType, amount, date, category, description. Містить методи:

- createTransaction(transactionData) створює нову транзакцію з вказаними даними.
- editTransaction(transactionId, transactionData) редагує існуючу транзакцію.
- deleteTransaction(transactionId) видаляє транзакцію.

Клас Bill забезпечує зберігання інформації про рахунки, які потрібно оплатити, включаючи назву рахунку, постачальника, суму, дату оплати, частоту, категорію, нотатки, а також налаштування сповіщень та автоматичної оплати. Присутні атрибути billName, vendor, amount, dueDate, frequency та інші. Містить методи:

- payBill(billId, paymentMethod) оплачує рахунок вказаним методом.

- schedulePayment(billId, paymentDate) планує оплату рахунку на вказану дату.
- setNotification(billId, enabled) вмикає/вимикає сповіщення про рахунок.
- setAutoPay(billId, enabled) вмикає/вимикає автоматичну оплату рахунку.

Клас Budget забезпечує зберігання інформації про бюджет користувача на певний період, включаючи період бюджету, дату початку та закінчення, загальний дохід, загальні витрати. Є атрибути budgetPeriod, startDate, endDate, totalIncome, totalExpenses. Містить методи:

- allocateBudget(budgetData) розподіляє бюджет по категоріях.
- trackExpenses(expensesData) відстежує фактичні витрати по категоріях.
- generateReport(reportType) генерує звіт про бюджет за вказаним типом.

Клас BudgetCategory забезпечує зберігання інформації про категорію бюджету, включаючи назву категорії, заплановану суму та фактичну суму витрат. Атрибути categoryName, allocatedAmount, actualAmount. Містить методи:

- setAllocatedAmount(amount) встановлює заплановану суму для категорії.
- trackActualAmount(amount) додає фактичну суму витрат до категорії.

Клас Debt забезпечує зберігання інформації про борг користувача, включаючи кредитора, тип боргу, баланс, процентну ставку, мінімальний платіж та дату оплати. Атрибути як creditor, debtType, balance, interestRate, minimumPayment. Містить методи:

- makePayment(debtId, amount) здійснює платіж по боргу.
- calculateInterest(debtId) розраховує нараховані проценти по боргу.
- trackProgress(debtId) відстежує прогрес погашення боргу.

Клас Report забезпечує зберігання інформації про звіти, включаючи тип звіту, дату генерації та дані звіту. Атрибути reportType, dateGenerated, data. Містить методи:

- generateReport(reportType, data) генерує звіт за вказаним типом та даними.

- viewReport(reportId) відображає звіт.
- exportReport(reportId, format) експортує звіт у вказаному форматі.

Ця діаграма класів надає візуальне представлення всіх компонентів програми, їх взаємозв'язків та основних функцій.

Далі було прийнято рішення створити діаграму послідовності — тип діаграми взаємодії в UML, що візуально демонструє послідовність обміну повідомленнями між об'єктами або компонентами системи під час виконання конкретної функціональності. Цей тип діаграми дозволяє наочно відобразити процес взаємодії різних елементів системи, акцентуючи увагу на хронологічному порядку та взаємозалежності дій.

Використовується для моделювання логіки виконання сценаріїв, описаних на діаграмі варіантів використання, та деталізації взаємодії між користувачами й внутрішніми компонентами системи.

Цей підхід є корисним під час проєктування системи, оскільки допомагає ідентифікувати ключові взаємозв'язки між її складовими та виявити можливі помилки у логіці роботи системи, а також оптимізувати її функціонування.

Для початку створено діаграму послідовності для сценарію ВВ "Додавання транзакцій". Діаграма зображена на рисунку 2.3.

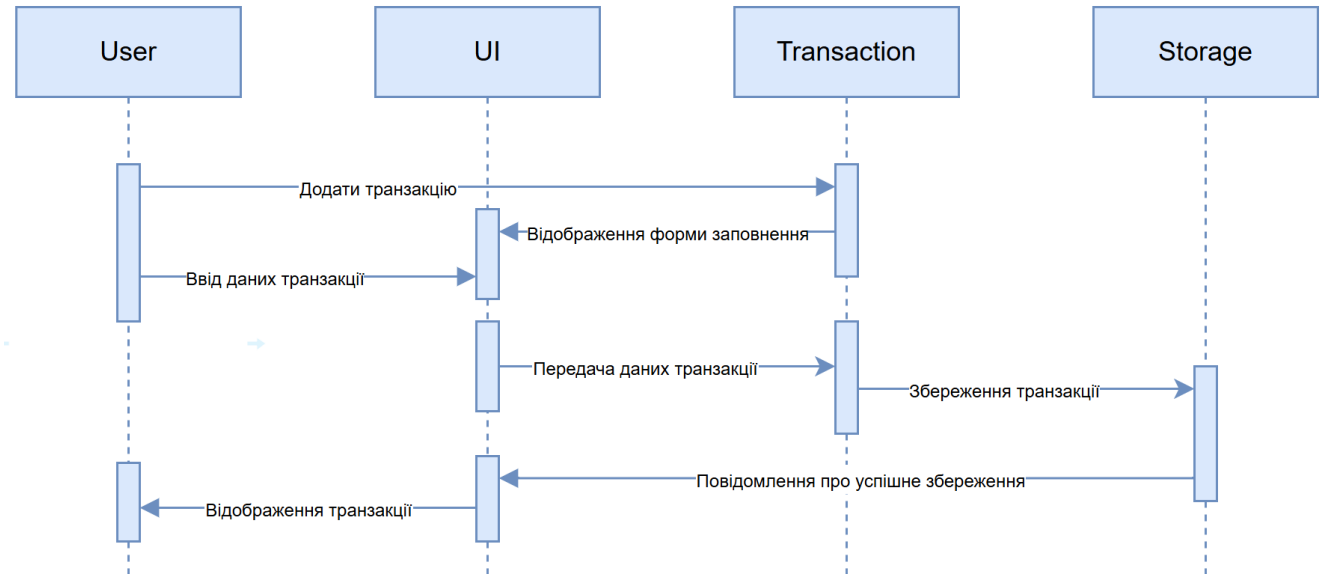

Рисунок 2.3 – Діаграма послідовності для сценарію ВВ "Додавання транзакцій"

Наступною створено діаграму послідовності для сценарію ВВ "Формування звітів". Вона показана на рисунку 2.4.

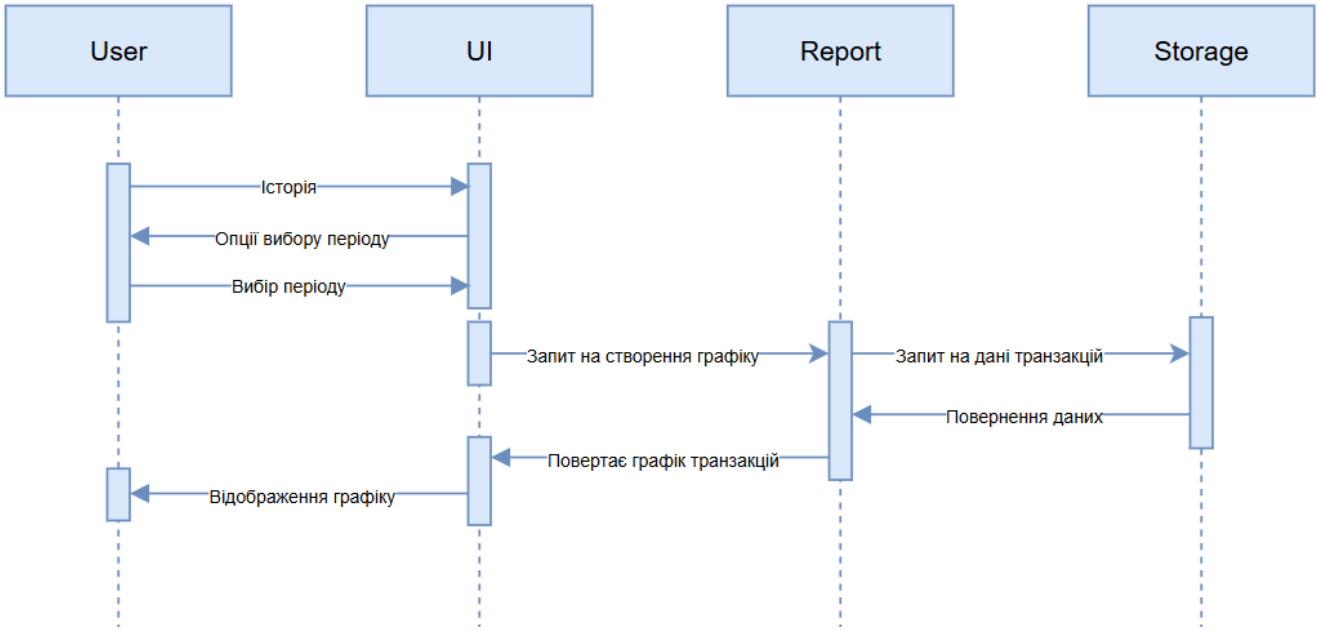

Рисунок 2.4 – Діаграма послідовності для сценарію ВВ "Формування звітів"

Ці діаграми послідовності дозволяють простежити послідовність дій та обмін даними між об'єктами в реальному часі, що дає глибше розуміння принципів роботи програми.

3 КОНСТРУЮВАННЯ ДОДАТКУ

3.1 Реалізація ключових компонентів

Ефективна реалізація ключових компонентів має вирішальне значення для хорошого програмного проекту. Ці компоненти, що представляють основні концепції в системі, слугують фундаментальними будівельними блоками, на яких будується вся програма. Їх правильна реалізація гарантує, що система буде надійною, підтримуваною та масштабованою. Вони сприяють повторному використанню коду, підтримці, масштабованості та читабельності, що сприяє створенню високоякісних, надійних та адаптивних програмних систем.

Добре реалізовані ключові компоненти сприяють повторному використанню коду та зменшують надмірність. Також роблять значний внесок у загальну ремонтопридатність системи. Коли потрібні зміни, можна внести модифікації до цих ключових класів, і вплив поширюється на всю програму без необхідності значного переписування коду.

Реалізовані компоненти та їх методи були створені з використанням VS Code (рис. 3.1).

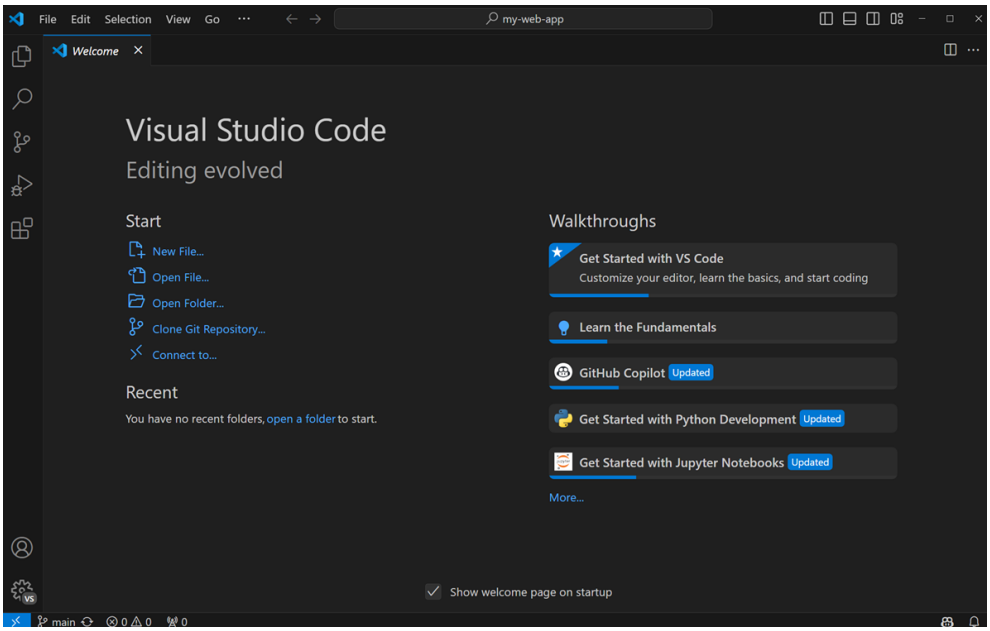

Рисунок 3.1 – Редактор вихідного коду VS Code

В цьому розділі буде показано ключові методи деяких компонентів додатку.

User відповідає за представлення користувача програми та зберігання його даних. Забезпечує аутентифікацію та авторизацію.

Визначає такі методи, як login, logout, updateProfile, changePassword.

Основні функції класу:

- зберігання персональної інформації;
- аутентифікація;
- зміна паролю.

Ключову частину коду методів компоненту User наведено на рисунку 3.2.

```
async login(): Promise<boolean> {
  try {
    // Example logic: Validate credentials against a database or API
    if (!this.email || !this.password) throw new Error("Missing credentials");
    console.log("User logged in successfully");
    return true;
  } catch (error) {
    console.error("Login failed:", error.message);
    return false;
  }
}

async changePassword(oldPassword: string, newPassword: string): Promise<boolean> {
  try {
    if (oldPassword !== this.password) throw new Error("Old password does not match");
    this.password = newPassword;
    console.log("Password changed successfully");
    return true;
  } catch (error) {
    console.error("Failed to change password:", error.message);
    return false;
  }
}
```

Рисунок 3.2 – Основні методи User

Transaction представляє фінансову транзакцію (дохід або витрата).

Визначає такі методи, як createTransaction, editTransaction, deleteTransaction.

Основні функції класу:

- зберігання деталей транзакцій, як сума, дата, категорія, опис, метод оплати;
- відображення транзакцій в історії рахунку;
- класифікація транзакцій за категоріями.

Ключову частину коду методів компонента Transaction наведено на рисунку 3.3.

```
createTransaction(): void {
  try {
    if (this.amount <= 0) throw new Error("Transaction amount must be positive");
    console.log("Transaction created successfully");
  } catch (error) {
    console.error("Failed to create transaction:", error.message);
  }
}

editTransaction(newTransactionData: Partial<Transaction>): void {
  try {
    Object.assign(this, newTransactionData);
    console.log("Transaction updated successfully");
  } catch (error) {
    console.error("Failed to update transaction:", error.message);
  }
}

deleteTransaction(): void {
  console.log("Transaction deleted successfully");
}
```

Рисунок 3.3 – Основні методи Transaction

Bill представляє рахунок, який потрібно оплатити (комунальні послуги, підписки та інші сервіси).

Визначає такі методи, як schedulePayment, payBill та інші.

Основні функції класу:

- зберігання інформації про рахунок;
- нагадування про оплату рахунків.

Ключову частину коду методів компонента Bill наведено на рисунку 3.4.

```
payBill(): void {
  try {
    console.log(`Bill ${this.billName} paid successfully`);
  } catch (error) {
    console.error("Failed to pay bill:", error.message);
  }
}

schedulePayment(date: Date): void {
  try {
    if (date <= new Date()) throw new Error("Scheduled date must be in the future");
    console.log(`Bill ${this.billName} scheduled for payment on ${date}`);
  } catch (error) {
    console.error("Failed to schedule payment:", error.message);
  }
}
```

Рисунок 3.4 – Основні методи Bill

Budget представляє бюджет користувача.

Визначає такі методи, як allocateBudget, trackExpenses, generateReport.

Основні функції класу:

- планування доходів та витрат;
- розподіл бюджету по категоріях;
- відстеження фактичних витрат;
- аналіз відповідності фактичних витрат запланованим;
- ефективне планування фінансів.

Ключову частину коду методів компонента Budget наведено на рисунку 3.5.

```
allocateBudget(): void {
  try {
    this.categories.forEach((category) => {
      if (category.allocatedAmount > this.totalIncome) {
        throw new Error(
          `Allocated amount for category ${category.categoryName} exceeds total income.`
        );
      }
    });
    console.log("Budget allocated successfully.");
  } catch (error) {
    console.error(`Failed to allocate budget: ${error.message}`);
  }
}

trackExpenses(): void {
  try {
    const totalTracked = this.categories.reduce(
      (sum, category) => sum + category.actualAmount,
      0
    );
    if (totalTracked > this.totalIncome) {
      console.warn("Expenses exceed total income.");
    }
    console.log("Expenses tracked successfully.");
  } catch (error) {
    console.error(`Failed to track expenses: ${error.message}`);
  }
}
```

Рисунок 3.5 – Основні методи Budget

Debt представляє борги користувача (позика, кредит та інші).

Визначає такі методи, як makePayment, calculateInterest, trackProgress.

Основні функції класу:

- зберігання інформації про борг, як кредитор, тип боргу, баланс, процентна ставка, термін погашення;

- відстеження платежів по боргам;
- розрахунок відсотків та загальної суми до сплати.

Ключову частину коду методів компонента Debt наведено на рисунку 3.6.

```
calculateInterest(): number {
  try {
    const interest = (this.balance * this.interestRate) / 100;
    console.log(
      `Interest calculated for debt ${this.creditor}: ${interest.toFixed(2)}`
    );
    return interest;
  } catch (error) {
    console.error(`Failed to calculate interest: ${error.message}`);
    return 0;
  }
}

trackProgress(): number {
  try {
    const progress = ((this.minimumPayment / this.balance) * 100).toFixed(2);
    console.log(
      `Progress tracked for debt ${this.creditor}: ${progress}% paid`
    );
    return parseFloat(progress);
  } catch (error) {
    console.error(`Failed to track progress: ${error.message}`);
    return 0;
  }
}
```

Рисунок 3.6 – Основні методи Debt

Report представляє звіти про фінансовий стан користувача.

Визначає такі методи, як generateReport, viewReport, exportReport.

Основні функції класу:

- генерація звітів за різними параметрами (доходи, витрати, бюджет, борги тощо);
- візуалізація даних у вигляді графіків та таблиць;
- експорт звітів у різні формати.

Ключову частину коду методів компонента Report наведено на рисунку 3.7.

```
generateReport(): void {
  try {
    if (!this.data) {
      throw new Error("No data available to generate the report.");
    }
    console.log(`${this.reportType} report generated successfully.`);
  } catch (error) {
    console.error(`Failed to generate report: ${error.message}`);
  }
}

viewReport(): void {
  try {
    console.log(`${this.reportType} report viewed:`, this.data);
  } catch (error) {
    console.error(`Failed to view report: ${error.message}`);
  }
}
```

Рисунок 3.7 – Основні методи Report

Таким чином було розглянуто реалізацію ключових компонентів додатку, які є основою його функціональності. Було описано призначення та основні методи класів User, Transaction, Bill, Budget, Debt та Report. Кожен компонент відповідає за певний аспект управління фінансами, такий як обробка даних користувача, транзакцій, рахунків, бюджету, боргів та генерація звітів.

3.2 Розробка GUI

Графічний інтерфейс користувача розроблений так, щоб бути зручним для користувача, полегшуючи людям навігацію, розуміння та роботу з програмним забезпеченням. Графічні інтерфейси використовують візуальні елементи для представлення функцій та інформації в програмі [14].

Графічні інтерфейси розробляються з думкою про зручність використання. Вони мають бути інтуїтивно зрозумілими та ефективними, щоб користувачі могли легко виконувати свої завдання.

Для розуміння розробки інтерфейсу було створено макети компонентів додатку.

Першим було вирішено розробити макет основної сторінки додатку, яку користувачі будуть бачити одразу при відкритті. На даному макеті зображено такі елементи:

- кнопка сповіщень та профілем користувача;
- модуль з балансом, витратами та збереженнями;
- відповідні графіки.

Створений макет показаний на рисунку 3.8.

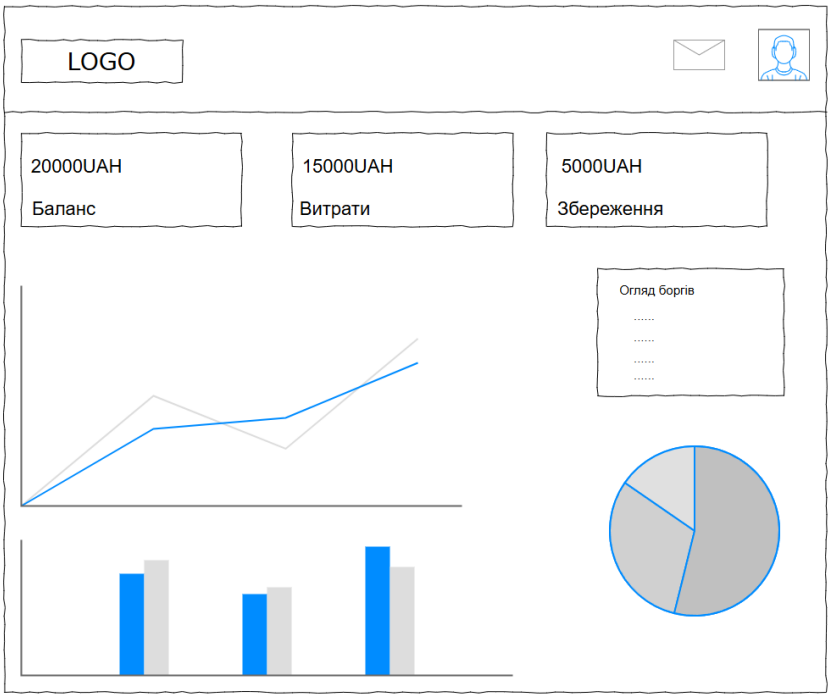

Рисунок 3.8 – Мокап основної сторінки

Далі створено макет для сторінки з бюджетом користувача. На даному макеті зображено такі елементи:

- модуль з джерелами доходу та ліміт на витрати;
- кругова діаграма з розподіленням бюджету;
- модуль зі збереженнями.

Створений макет показано на рисунку 3.9.

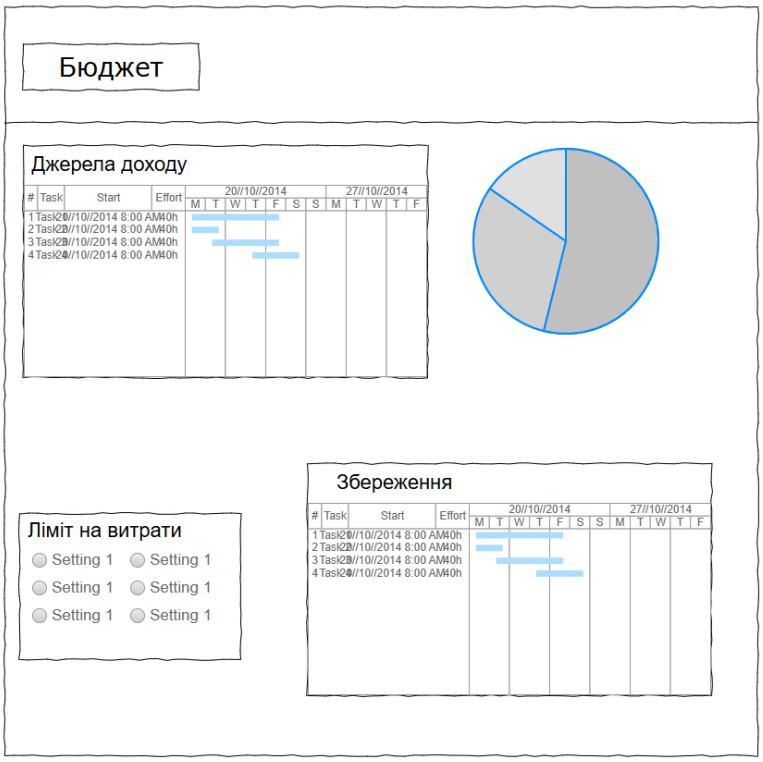

Рисунок 3.9 – Компонент з бюджетом

Наступним створено сторінку рахунками, на даному макеті модуль з рахунками та графік з тенденціями в оплаті рахунків (рис. 3.10).

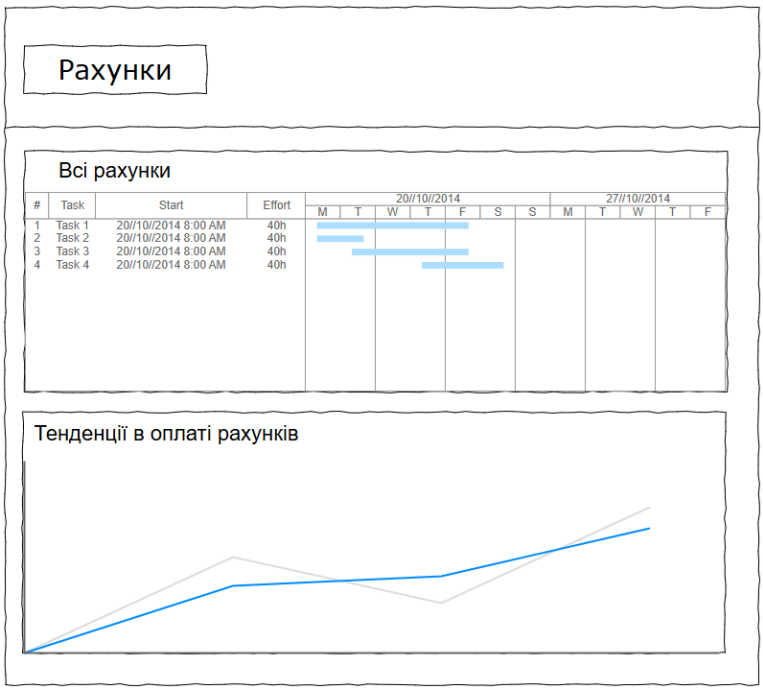

Рисунок 3.10 – Компонент з рахунками

Далі відбулось створення макету, де буде присутня аналітика та звіти, тобто графіки або діаграми, витрат або доходів. Макет показано на рисунку 3.11.

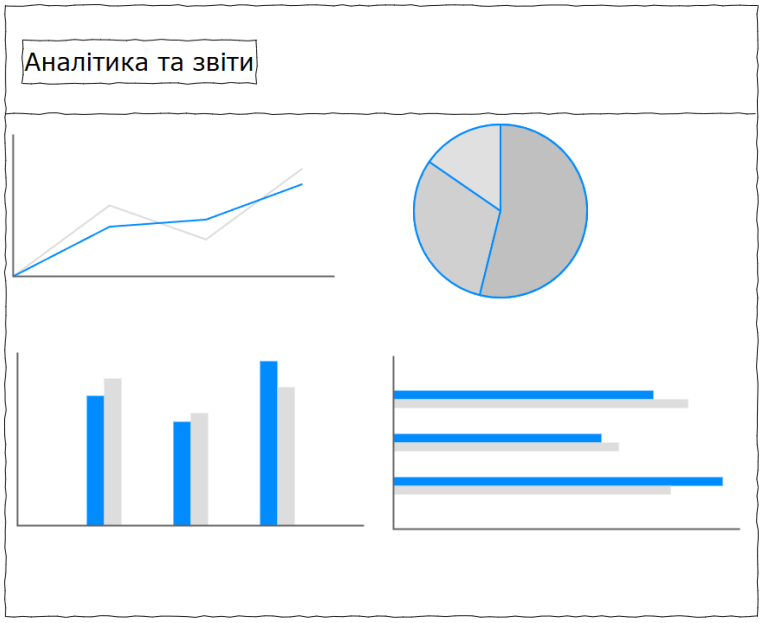

Рисунок 3.11 – Компонент аналітики та звітів

Остіннім розробленим макетом стала сторінка з огляд боргів (кредити, позики тощо) . На макеті зображено історію боргів та їх виплат, а також графіки порівняння боргів до виплат. Макет показано на рисунку 3.12.

Рисунок 3.12 – Мокап сторінки “Огляд боргів”

3.3 Тестування, оцінка якості та результат розробки ПЗ

Розробка програмного забезпечення – це складний процес, що вимагає ретельного планування, проектування та реалізації. Але не менш важливим етапом є тестування, яке дозволяє переконатися в якості продукту та його відповідності вимогам користувачів. У цьому розділі буде розглянуто процес тестування розробленого додатку, проведено оцінку його якості та демонстранція досягнутих результатів.

Додаток успішно запрацював на операційних системах Windows 10 та Windows 11.

Додаток для управління особистими фінансами, розроблений на базі React та Electron, продемонстрував високу ефективність та зручність використання.

Далі розпочався огляд та тестування додатку. Одразу на основній сторінці можна побачити такі елементи:

- зліва розташоване вертикальне меню з пунктами бюджет, рахунки, звіти та аналітика, управління заборгованістю та налаштування;
- у правому верхньому куті міститься розділ профілю користувача з ім'ям («Андрій Харлан») та аватаркою, а також дзвіночок з сповіщеннями;
- у верхній частині розташовані три картки, що відображають ключові фінансові показники, такі як витрати, баланс та заощадження;
- лінійна діаграма, що показує тенденції доходів і витрат за місяцями;
- таблиця з деталями боргу, включаючи кредитора, тип боргу та залишок, з можливістю експорту або пошуку даних.

На рисунку 3.13 зображено основну сторінку додатка.

Рисунок 3.13 – Початкова сторінка додатку

Далі необхідно розглянути нижню половину основної сторінки, де розмістились такі елементи. Ця частина демонструє додаткові розділи інтерфейсу програми для управління фінансами, зосереджуючи увагу на детальних фінансових даних та візуалізаціях, таких як:

- історія транзакцій, де є таблиця зі стовпчиками для деталей транзакцій (ідентифікатор, дата транзакції, її тип та сама транзакція);
- функції вгорі включають опції експорту даних і пошуку конкретних транзакцій;
- стовпчаста гістограма, що відображає щоденні витрати протягом тижня;
- кожен стовпчик розділений на сегменти, що показують різні категорії витрат або суми (наприклад, 120 гривень, 150 гривень). Діаграма дає чітке уявлення про розподіл витрат у різні дні;
- кругова діаграма, що показує, як бюджет розподіляється за різними категоріями (електроенергія, телефон, газ, воду/каналізацію та інші). Ця візуалізація підкреслює частку загальних витрат на кожну категорію.

Ці елементи зображено на рисунку 3.14.

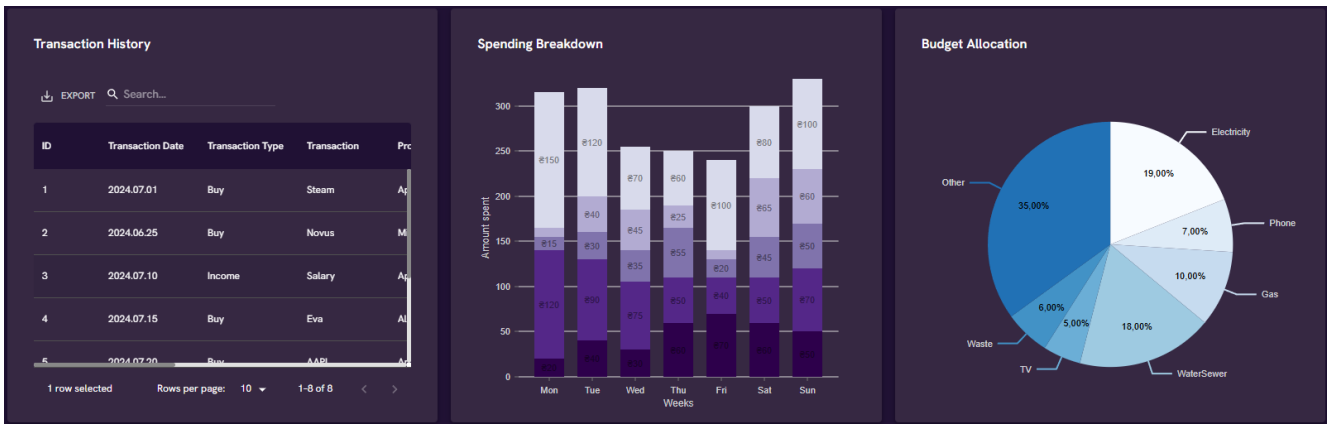

Рисунок 3.14 – Друга половина основної сторінки

Далі необхідно перейти до сторінки, що представляє розділ «Бюджет» додатку для управління особистими фінансами, що надає огляд джерел надходжень і розподілу бюджету. Сторінка зображена на рисунку 3.15. Тут присутні такі елементи:

- таблиця, яка детально описує різні потоки надходжень з стовпчиками, як ідентифікатор, назва джерела (зарплата на роботі, фріланс, гроші в подарунок), сума, метод оплати, частота та дата отримання з категорією доходу. Також є опція експорту даних або пошуку конкретних джерел доходу;
- кругова діаграма, що візуалізує розподіл бюджету за різними категоріями.

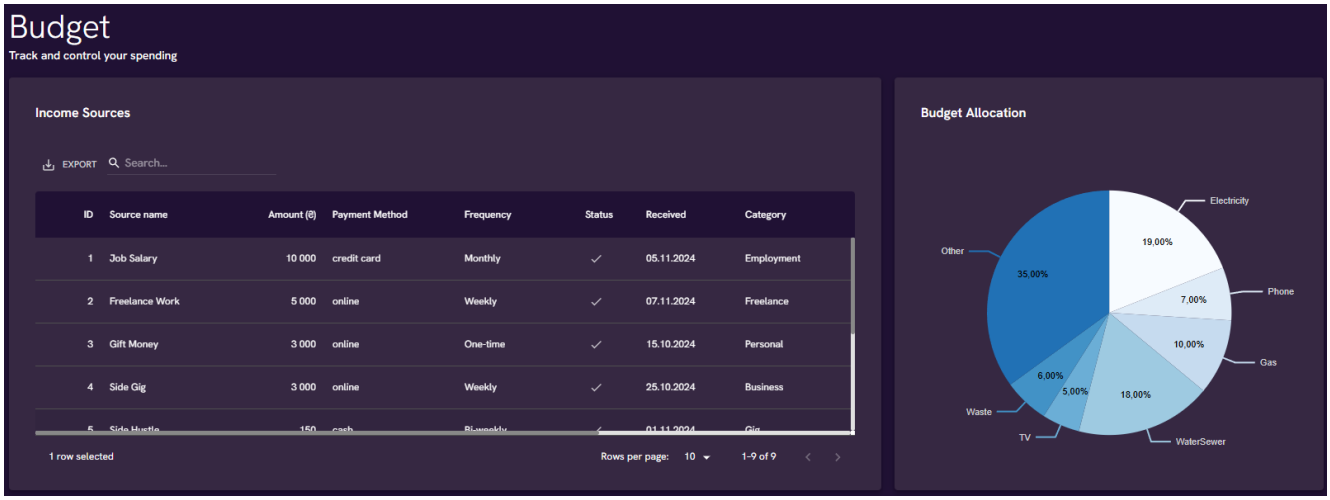

Рисунок 3.15 – Розділ «Бюджет»

Далі відображається наступна частина розділу «Бюджет», де є інтерфейс управління фінансами, розділений на два розділи:

- ліміти витрат зліва, де перелічуються різні категорії з їхніми поточними витратами та лімітами (комунальні послуги, їжа, ігри, книги та інші). Існує також опція для додавання нових категорій або зміни існуючих лімітів;
- заощадження з правої сторони, що відображає таблицю з детальними цілями заощаджень, включаючи мету заощаджень (непередбачувані витрати, на відпустку), поточну сума заощаджень для цілі, коротка інформація про мету, цільову суму та кінцевий термін досягнення цілі.

Ця частина орієнтована на ефективне відстеження витрат і заощаджень. Все це зображено на рисунку 3.16.

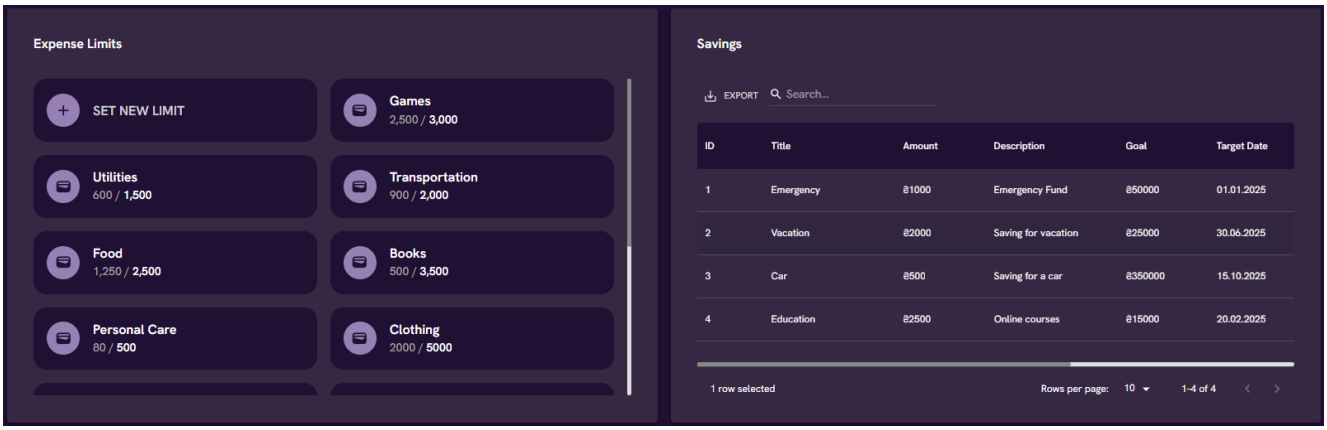

Рисунок 3.16 – Друга частина розділу «Бюджет»

Наступним зображено сторінку з рахунками, призначену для відстеження фінансових зобов'язань. Нижче наведено розбивку того, що включає в себе таблиця з детальною інформацією про різні рахунки:

- опції для експорту даних і пошуку рахунків;
- назви рахунків (Електроенергія, телефон», орендна плата);
- постачальник послуг (Тернопільобленерго, Lifecell);
- статус, що показує, чи є рахунок сплаченим або неоплаченим;

- дата, коли було здійснено платіж та кінцевий термін оплати, вартість рахунку;
- періодичність виставлення рахунку;
- спосіб оплати (кредитна картка, прямий дебет, банківський переказ);
- класифікація рахунку за категоріями .

Ця сторінка призначена для зручного керування та відстеження оплат рахунків. Її зображено на рисунку 3.17.

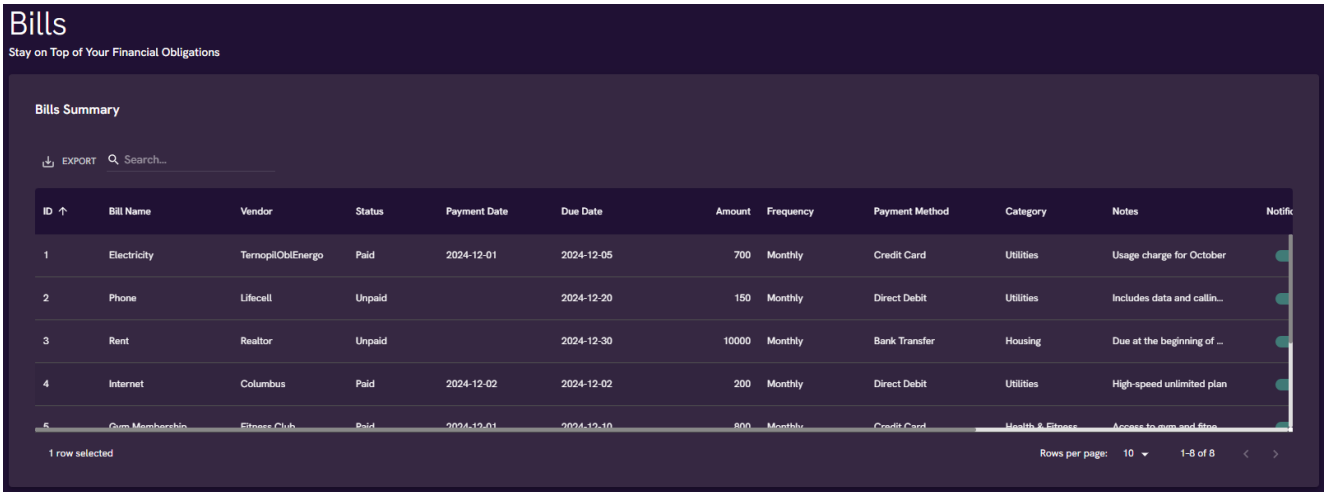

Рисунок 3.17 – Рахунки на створеній сторінці

Далі представлено сторінку звітів та аналізу з візуальною інформацією про фінансовий стан, розділений на два розділи:

- порівняння бюджету, де є гістограма, що порівнює бюджетні та фактичні витрати за різними категоріями (наприклад, продукти ₴500 (фактично) проти ₴400 (бюджет)). Вона допомагає оцінити перевитрати або недовитрати в межах кожної категорії;
- фінансовий огляд, що є лінійним графіком, який відслідковує доходи (зелений) та витрати (червоний) за місяцями року (січень-грудень). Цей графік візуалізує загальні фінансові тенденції, допомагаючи виявити закономірності та потенційні дисбаланси.

Цю сторінку показано на рисунку 3.18.

Рисунок 3.18 – Звіти та аналіз фінансової ситуації

На цій сторінці показано два основні розділи, пов'язані з управлінням боргами в фінансовому додатку:

- лівий розділ – це огляд боргу, що містить зведену інформацію про різні борги в таблиці, кредитор, тип боргу (кредитна картка, позика), відсоткова ставка, мінімальний платіж, тата погашення, статус та частота платежів;
- правий розділ – це історія виплат боргу з відповідною таблицею, де є дата, опис (за що було здійснено платіж) і тип.

Сторінка надає чітке та організоване уявлення про борги користувача та його історію платежів. Створену сторінку зображено на рисунку 3.19

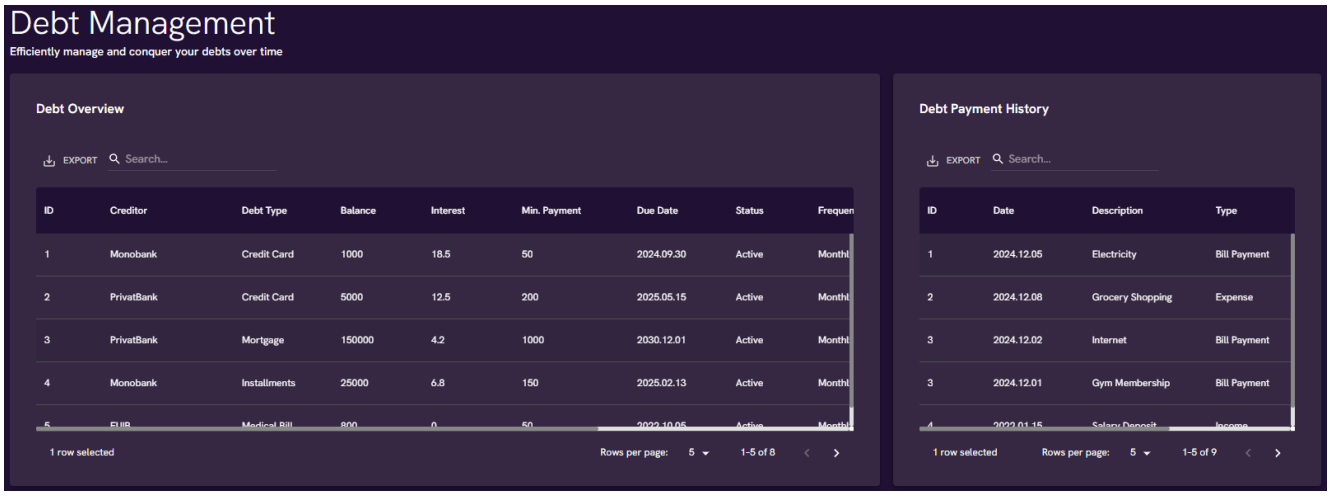

Рисунок 3.19 – Сторінка управління боргами

Ця сторінка є продовженням попередньої, де відображаються дві стовпчикові діаграми, пов'язані з аналізом боргу в додатку для особистих фінансів.

- перша діаграма візуалізує, як співвідношення боргу до доходу користувача змінювалося з часом. Вище співвідношення боргу до доходу, як правило, вказує на більший фінансовий тягар;
- друга діаграма ілюструє частку кожної категорії боргу відносно загального боргу користувача. Наприклад, «Житлові» мають найбільший внесок у борговий тягар.

Ці діаграми надають швидкий та інформативний огляд боргової ситуації користувача. Графік співвідношення боргу до доходу показує динаміку боргу порівняно з доходом за роками, а графік коефіцієнта боргового навантаження – структуру поточного боргу. Сторінка зображена на рисунку 3.20.

Рисунок 3.20 – Графіки щодо статистики боргів

Останньою сторінкою будуть налаштування додатку та профілю користувача. Тут користувачі можуть керувати своєю особистою інформацією та налаштуваннями програми. Тут містяться такі елементи:

- інформація про користувача, що відображає ім'я , фотографію профілю, коротку біографію та його місцезнаходження;
- розділ особистих даних включає поля для введення та редагування інформації, такої як прізвище, ім'я, email, дата народження та номер телефону;

- є такі налаштування, як мова, формат дати і часу, валюта та часовий пояс;
- поля для країни, міста, поштового індексу та РНОКПП;
- поле, де користувачі можуть змінити пароль власного облікового запису.

Сторінка надає повний огляд профілю користувача і дозволяє йому керувати своєю особистою інформацією, налаштуваннями облікового запису і параметрами безпеки. Зображена на рисунку 3.21.

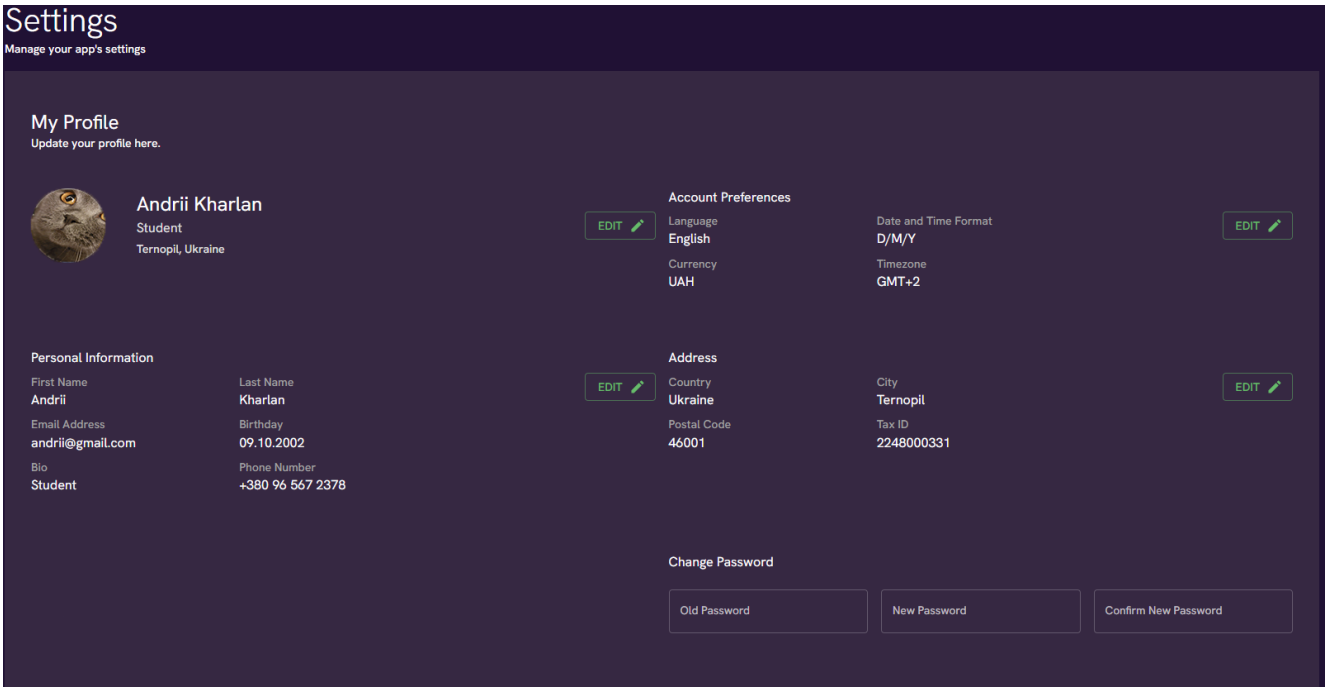

Рисунок 3.21 – Налаштування додатку

У цьому розділі було проведено тестування розробленого додатку для управління особистими фінансами. Було детально розглянуто інтерфейс додатку. Кожна сторінка містить інтуїтивно зрозумілі елементи управління, інформативні візуалізації та зручні функції для ефективного управління особистими фінансами.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Охорона праці

Розробка додатку для управління особистими фінансами проводилась з дотриманням сучасних стандартів у сфері інструкцій з охорони праці.

Інструкція з охорони праці для ІТ-фахівця визначає обов'язкові правила безпеки, яких він має дотримуватися під час виконання своїх робочих обов'язків. Інструкція розроблена відповідно до:

- Закону України «Про охорону праці», що був введений в дію Постановою ВР № 2695-XII від 14.10.92 [15];
- Положення про розробку інструкцій з охорони праці, затвердженого наказом Держанаглядохоропраці від 29.01.1998 № 9, із змінами, внесеними згідно з Наказом Міністерства соціальної політики № 526 від 30.03.2017 [16];
- Типового положення про порядок навчання та перевірки знань з охорони праці, затверджене наказом Держнаглядохоронпраці України № 15 від 26 січня 2005 року, що визначає види та порядок проведення інструктажів з охорони праці [17];
- Про затвердження вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями, затверджених наказом Міністерства соціальної політики України від 14.02.2018 № 207 [18].

Протягом всієї робочої зміни програміст працює на спеціально обладнаному для цього місці.

З метою дотримання правил охорони праці на робочому місці, ІТ-фахівець зобов'язаний:

- дотримуватися правил внутрішнього трудового розпорядку, інструкцій з охорони праці, пожежної безпеки та електробезпеки;
- підтримувати чистоту і порядок на робочому місці, дотримуватися правил особистої гігієни;

- не допускати сторонніх осіб до свого робочого місця;
- не працювати у стані алкогольного, наркотичного чи медикаментозного сп'яніння, а також у хворобливому або втомленому стані.

Під час виконання своїх професійних обов'язків ІТ-фахівець стикається з низкою шкідливих та небезпечних факторів, які можуть негативно вплинути на його здоров'я та працездатність. Саме тому дотримання вимог інструкції з охорони праці є важливим елементом забезпечення безпеки та комфорту на робочому місці. Під час роботи на ІТ-фахівця можуть негативно впливати такі фактори:

- недостатнє або неправильно організоване освітлення;
- ризик ураження електричним струмом;
- випромінювання від монітора та іншої оргтехніки;
- психологічне навантаження та емоційне виснаження;
- навантаження на зір;
- малорухливий спосіб життя;
- інші шкідливі фактори, пов'язані з роботою за комп'ютером.

Монітор на робочому місці має бути налаштований так, щоб працівник міг вільно змінювати його положення (поворот і нахил) для зручності роботи. Зображення на екрані повинно бути чітким та стабільним, а яскравість і контрастність символів - легко регулюватися для комфортного сприйняття.

Перед початком роботи ІТ-фахівець має:

- отримати інструктаж від керівника;
- перевірити, чи немає пошкоджень на техніці, що необхідна для роботи;
- впевнитися, що освітлення достатнє, а контраст між екраном і навколишнім середовищем – оптимальний;
- перевірити, чи справні шнури живлення, вимикачі, розетки, штепсельні з'єднання, а також вентиляція;
- якщо виявлені несправності в обладнанні, повідомити про це керівника і не починати роботу, доки їх не усунуть.

Під час виконання роботи фахівець повинен:

- забезпечувати чистоту і порядок на робочому місці;
- стежити за справністю обладнання та правильно його використовувати;
- вимкнути оргтехніку та інше обладнання з електромережі, якщо воно не використовується цілодобово.

Після закінчення роботи потрібно:

- прибрати робоче місце та провітрити приміщення;
- вимкнути будь-яке робоче обладнання з електромережі, крім того, яке працює цілодобово;
- привести себе до ладу, помити руки та обличчя, переодягнутися;
- повідомити безпосереднього керівника про всі виявлені недоліки та особливості роботи обладнання.

Якщо під час роботи сталася надзвичайна ситуація, внаслідок якої працівник отримав травму, керівник структурного підрозділу та сам працівник повинні негайно повідомити про це безпосереднього керівника. Він, в свою чергу, має терміново проінформувати керівництво організації та відповідального за охорону праці для вжиття необхідних заходів [5].

Інструкція з охорони праці для ІТ-фахівця є важливим документом, що забезпечує безпеку та здоров'я працівника під час виконання робочих обов'язків. Дотримання встановлених вимог дозволяє мінімізувати ризики негативного впливу виробничих факторів, таких як навантаження на зір, малорухливий спосіб життя, психологічне виснаження та технічні небезпеки.

Виконання рекомендацій щодо організації робочого місця, правильного використання обладнання, підтримання порядку та особистої гігієни сприяє підвищенню продуктивності праці та створенню комфортних умов для роботи. Регулярне проведення інструктажів, перевірка стану техніки та дотримання правил безпеки є ключовими елементами профілактики нещасних випадків і збереження здоров'я ІТ-фахівців.

4.1 Безпека в надзвичайних ситуаціях

Під час розробки додатку для управління особистими фінансами можуть впливати такі небезпечні та шкідливі фактори, як фізичні та психофізіологічні. Тому необхідно дотримуватись наступних правил для безпечної роботи за ПК.

Згідно наказу «Про затвердження Правил безпечної експлуатації електроустановок споживачів», при роботі з комп'ютерами, щоб уникнути ураження електричним струмом, важливо правильно розмістити обладнання та кабелі. Додаткові заходи електробезпеки збігаються із загальними правилами пожежної та електробезпеки [19].

Для попередження пожеж:

- використовувати приховану електропроводку та надійні розетки з вогнетривких матеріалів;
- обирати кабелі живлення, розраховані на навантаження в 3-5 разів більше, ніж споживає обладнання;
- вмикати та вимикати обладнання тільки штатними вимикачами;
- регулярно очищувати комп'ютери та інше обладнання від пилу;
- розміщувати комп'ютери на окремих столах з негорючих матеріалів;
- щоб запобігти іскрінню, намагатись якомога рідше вставляти та виймати вилки з розеток.

Відповідно до Наказу «Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями», система освітлення на робочому місці ІТ-фахівця має відповідати наступним вимогам:

- освітленість повинна бути достатньою та відповідати характеру зорової роботи, враховуючи розмір об'єктів на екрані, фон, контрастність;
- яскравість на робочій поверхні та в навколишньому просторі має бути рівномірно розподілена;
- на робочій поверхні не повинно бути різких тіней;

- необхідно уникати відблисків, які можуть засліплювати та викликати дискомфорт;
- освітленість має бути стабільною протягом робочого дня;
- слід обрати правильний напрямок світлового потоку та склад світла (колірну температуру).

Робочі місця повинні бути розташовані таким чином:

- на відстані не менше 1,5 м від стіни з вікнами, не менше 1 м від інших стін, а відстань між робочими місцями має становити щонайменше 1,5 м;
- їх слід організовувати так, щоб уникати потрапляння в очі прямого світла. Джерела освітлення краще розміщувати по обидва боки від екрана паралельно до напрямку погляду;
- для зменшення світлових відблисків від екрана чи клавіатури, що можуть утворюватися від світильників загального освітлення або сонячних променів, слід використовувати на вікнах жалюзі.

Екран монітора потрібно розташовувати перпендикулярно до напрямку погляду користувача. Нахил екрана може призвести до сутулості. Відстань від монітора до очей повинна бути трохи більшою, ніж звична відстань до книги під час читання. Для збереження зору важливо створювати неоднорідне поле зору. Це можна досягти, розмістивши на стінах плакати або картини у спокійних тонах, наприклад, із зображенням пейзажів. Під час роботи з текстовою інформацією (введення даних, редагування тексту, читання з екрана) найбільш зручним є відображення чорних символів на світлому тлі. Монітор повинен розташовуватися в центрі поля зору користувача на відстані 400–700 мм від очей. Елементи робочого місця слід розміщувати так, щоб забезпечити однакову відстань очей до екрана, клавіатури та текстових матеріалів [20].

Зручна робоча поза під час роботи за комп’ютером теж важлива для забезпечення безпеки життєдіяльності, досягається завдяки регулюванню висоти робочого столу, крісла та підставки для ніг. Оптимальним є положення, коли ступні розташовані горизонтально на підлозі або підставці, стегна знаходяться у

горизонтальній площині, а верхні частини рук — у вертикальному положенні. Кут у ліктьовому суглобі має бути в межах 70–90°, зап'ястя слід тримати зігнутими під кутом не більше 20°, а нахил голови має становити 15–20°.

Особливо важлива форма спинки крісла, яка повинна повторювати природні вигини спини. Висота крісла має забезпечувати відсутність тиску на стегна. Бажано, щоб крісло було обладнане підлокітниками. Крісло слід встановлювати так, щоб не доводилося тягнутися до клавіатури. Для підтримання працездатності необхідно періодично змінювати положення тіла, рухатися і робити короткі перерви. При інтенсивній роботі за комп’ютером рекомендується робити 15-хвилинну перерву після кожної години роботи.

Забороняється:

- розміщувати будь-які предмети на комп’ютерному обладнанні, оскільки це може призвести до його пошкодження;
- закривати вентиляційні отвори апаратури, оскільки це перешкоджає нормальній циркуляції повітря, що може викликати перегрівання та вихід обладнання з ладу;
- використовувати комп’ютерну техніку у вологих приміщеннях або розташовувати її поблизу рідин, щоб уникнути короткого замикання та інших технічних несправностей.

Дотримання правил безпеки під час роботи за комп’ютером є необхідною умовою для збереження здоров'я ІТ-фахівців і забезпечення безперебійної роботи обладнання. Ефективна організація робочого місця, включаючи правильне розташування комп’ютерної техніки, регулювання освітлення, оптимальну робочу позу та дотримання правил електробезпеки, дозволяє мінімізувати негативний вплив на здоров'я [21].

Належний догляд за комп'ютерним обладнанням забезпечує безпечну експлуатацію техніки та запобігає її передчасному виходу з ладу. Таким чином, суворе дотримання встановлених норм і рекомендацій з охорони праці є запорукою продуктивної роботи, підтримки здоров'я працівника та тривалого функціонування комп'ютерного обладнання [22].

ВИСНОВКИ

У ході виконання дипломної роботи магістра на тему “Впровадження компонентно-орієнтованої архітектури для створення додатку управління особистими фінансами з використанням React та Electron” відбулось дослідження теоретичних аспектів компонентно-орієнтованої архітектури та особливостей її застосування у розробці сучасних програмних продуктів. Було проаналізовано переваги та недоліки використання технологій React та Electron, мови програмування TypeScript, для створення кросплатформних додатків, а також обґрунтовано їх вибір для реалізації поставленої мети.

Також було розроблено детальну бізнес-модель додатку, включаючи діаграму варіантів використання, сценарії використання, діаграму класів та діаграми послідовності. Це дозволило чітко визначити функціональні вимоги та структуру додатку, а також забезпечити його відповідність потребам користувачів.

Крім того, було проведено реалізацію ключових компонентів додатку та створення макетів інтерфейсу. А також тестування розробленого додатку, що підтвердило його високу ефективність, стабільність та зручність використання. Додаток успішно реалізує необхідні функції, включаючи реєстрацію та авторизацію користувачів, додавання та відстеження доходів і витрат, створення та управління бюджетом, перегляд фінансової історії та інші.

Додаток може бути доповнений за допомогою додавання двохфакторної автентифікації, що допоможе підвищити безпеку користувача та, коли це буде доступно, додавання системи Open Banking, що дозволить набагато краще та всеосяжніше відслідковувати фінансову історію.

Результати дипломної роботи демонструють ефективність впровадження компонентно-орієнтованої архітектури, технологій React та Electron для створення складних програмних продуктів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Fintech Analysis of Personal Finance App Usage among Millennials / Tika Handayani et al. Journal of Economic Education and Entrepreneurship Studies. 2024. Vol. 5, no. 2. P. 150–162.
2. Everything about Open Banking [Електронний ресурс] – Режим доступу до ресурсу: https://www.openbanking.org.uk/
3. Stefanyshyn, I., Pastukh, O., Stefanyshyn, V., Baran, I., Boyko, I. Robustness of AI algorithms for neurocomputer interfaces based on software and hardware technologies CEUR Workshop Proceedings, 2024, 3742, pp. 137–149.
4. Документація по React [Електронний ресурс] – Режим доступу до ресурсу: https://react.dev/
5. Документація по Electron [Електронний ресурс] – Режим доступу до ресурсу: https://www.electronjs.org/docs
6. Документація по TypeScript [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/docs/
7. Документація Firebase [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs
8. Що таке Scrum? [Електронний ресурс] – Режим доступу до ресурсу: https://www.scrum.org/resources/what-scrum-module
9. Петрик М. Р., Мудрик І. Я. Проєктування програмного забезпечення на основі об’єктно-орієнтованого аналізу вимог та інструментальних засобів розробки IBM Rational Software Architect / М. Р. Петрик, І. Я. Мудрик – Тернопіль : ТНТУ ім. І. Пулюя, 2022. – 56 с.
10. Все про компонентно-орієнтовану архітектуру [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/component-based-architecture-system-design/
11. Словник MVC [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Glossary/MVC

12. Основи та практичне використання UML [Електронний ресурс] – Режим доступу до ресурсу: https://www.uml.org/

13. The Unified Modeling Language [Електронний ресурс] – Режим доступу до ресурсу: https://www.uml-diagrams.org/

14. Hiatt J. C. React JS Foundations Building User Interfaces with ReactJS: An Approachable Guide. Wiley & Sons, Incorporated, John, 2022.

15. Закон України «Про охорону праці». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/2694-12

16. ДНАОП 0.00-4.15-98 «Про затвердження Положення про розробку інструкцій з охорони праці». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0226-98

17. НПАОП 0.00-4.12-05 «Типове положення про порядок проведення навчання і перевірки знань з питань охорони праці». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0231-05

18. НПАОП 0.00-7.15-18 «Про затвердження Вимог щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0508-18

19. ДНАОП 0.00-1.21-98 «Про затвердження Правил безпечної експлуатації електроустановок споживачів» [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0093-98

20. ДСанПіН 3.3.2.007-98 «Норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин» [Електронний ресурс] URL: https://zakon.rada.gov.ua/rada/show/v0007282-98.

21. Навчальний посібник «ТЕХНОЕКОЛОГІЯ ТА ЦИВІЛЬНА БЕЗПЕКА. ЧАСТИНА «ЦИВІЛЬНА БЕЗПЕКА»» / автор-укладач В.С. Стручок– Тернопіль: ФОП Паляниця В. А., – 156 с.

22. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання «БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ» / В.С. Стручок –Тернопіль: ФОП Паляниця В. А., - 156 с.

ДОДАТКИ

ДОДАТОК А

Публікація в науковому виданні

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
імені ІВАНА ПУЛЮЯ
НАУКОВЕ ТОВАРИСТВО ім. ШЕВЧЕНКА

М А Т Е Р І А Л И

XII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

18–19 грудня 2024 року

ТЕРНОПІЛЬ
2024

УДК 004.41
А. Харлан; М. Петрик, д.ф.-м.н., професор
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

РОЗРОБКА МОДУЛЬНИХ СИСТЕМ ЗВІТНОСТІ У ФІНАНСОВИХ ДОДАТКАХ З КОМПОНЕНТНОЮ АРХІТЕКТУРОЮ

UDC 004.41
A. Kharlan; M. Petryk, Dr., Prof.

DEVELOPMENT OF MODULAR REPORTING SYSTEMS IN FINANCIAL APPLICATIONS WITH COMPONENT ARCHITECTURE

Системи звітності є ключовим елементом фінансових додатків, оскільки вони забезпечують користувачів аналітичними даними для прийняття рішень, прогнозування та моніторингу. Впровадження компонентно-орієнтованої архітектури відкриває нові можливості для створення гнучких, масштабованих і налаштовуваних систем звітності, які відповідають індивідуальним потребам користувачів. Модульність стає основою для адаптації систем до нових вимог без необхідності перепроєктування базової інфраструктури.

Система звітності в рамках компонентної архітектури складається з окремих незалежних елементів, кожен з яких відповідає за певний функціонал: побудову діаграм, формування таблиць, реалізацію фільтрів даних, візуалізацію ключових показників тощо [1]. Всі ці елементи можуть бути організовані у вигляді бібліотеки готових модулів, що забезпечує їх багаторазове використання у різних частинах додатку. Такий підхід спрощує оновлення функціональності, оскільки заміна або вдосконалення одного модуля не потребує значних змін у роботі інших компонентів.

Динамічна побудова звітів є одним із ключових завдань. Користувачі повинні мати можливість самостійно налаштовувати структуру звітів, вибираючи потрібні компоненти з доступної бібліотеки. Наприклад, фінансовий аналітик може створити власний звіт із діаграмою грошового потоку, таблицею витрат за категоріями та графіком прогнозування доходів. Реалізація таких функцій можлива завдяки використанню React, що забезпечує високу швидкість візуалізації компонентів.

Захист даних у фінансових системах є одним із найважливіших аспектів, адже робота з конфіденційною інформацією вимагає високого рівня надійності. Використання компонентно-орієнтованої архітектури [2] створює умови для впровадження передових механізмів безпеки, таких як шифрування, контроль доступу та багаторівнева аутентифікація, безпосередньо на рівні окремих модулів. Цей підхід дозволяє ефективно ізолювати потенційні вразливості та забезпечувати баланс між гнучкістю системи й захистом даних.

Розробка модульних систем звітності відкриває нові горизонти для фінансових додатків, що поєднують високу продуктивність, інтуїтивно зрозумілий інтерфейс і адаптивність до зростаючих потреб користувачів. Такі рішення сприяють оптимізації операційних процесів для бізнесу та приватних осіб, стаючи інструментом для аналізу й прогнозування фінансової діяльності. Використання сучасних технологічних підходів у розробці подібних систем формує основу для фінансових додатків майбутнього, здатних відповідати викликам цифрової трансформації та запитам нової ери економіки.

Література

1. Healy Kieran. Data Visualization: A Practical Introduction. Princeton University Press, 2018. 296 p.
2. George T. Heineman, Ivica Crnkovic, Heinz G. Schmidt. Component-Based Software Engineering. Springer, 2008. 304 p.

ДОДАТОК Б

Лістинг коду розробленого додатку

Лістинг 1

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import {
  createHashRouter as createRouter,
  createRoutesFromElements as defineRoutes,
  Route,
  RouterProvider as AppRouter,
} from "react-router-dom";

// Importing app components
import Dashboard from "./pages/Dashboard";
import Budget from "./features/budget";
import Bills from "./features/bills";
import Reports from "./features/reports";
import Debt from "./features/debt";
import Settings from "./features/settings";
import Overview from "./features/overview";
import ErrorPage from "./pages/Error";

// Define routes
const appRoutes = defineRoutes(
  <Route path="/" element={<Dashboard />} errorElement={<ErrorPage />}>
    <Route index element={<Overview />} />
    <Route path="budget" element={<Budget />} />
    <Route path="bills" element={<Bills />} />
    <Route path="reports" element={<Reports />} />
    <Route path="debt" element={<Debt />} />
    <Route path="investments" element={<Investments />} />
    <Route path="networth" element={<NetWorth />} />
    <Route path="settings" element={<Settings />} />
  </Route>
);

// Create the router
const routerConfig = createRouter(appRoutes);

// Render the application
ReactDOM.createRoot(document.getElementById("root")!).render(
  <React.StrictMode>
    <AppRouter router={routerConfig} />
  </React.StrictMode>
);
```

Лістинг 2

```
export class Budget {
  private incomeSources: { id: number; name: string; amount: number }[] =
[];
```

```
  private expenseLimits: { id: number; title: string; currentValue: number;
limit: number }[] = [];
  private savings: { id: number; description: string; amount: number }[] =
[];
  private allocation: { id: string; label: string; value: number; color:
string }[] = [];

  // Income Sources
  getIncomeSources() {
    return [...this.incomeSources]; // Return a copy for immutability
  }
  addIncomeSource(name: string, amount: number) {
    if (amount <= 0) throw new Error("Income amount must be greater than
0");
    if (this.incomeSources.some((source) => source.name === name)) {
      throw new Error(`Income source "${name}" already exists`);
    }
    this.incomeSources.push({    id:    this.generateId(this.incomeSources),
name, amount });
    this.updateAllocation();
  }

  // Expense Limits
  getExpenseLimits() {
    return [...this.expenseLimits];
  }
  addExpenseLimit(title: string, limit: number) {
    if (limit <= 0) throw new Error("Expense limit must be greater than 0");
    this.expenseLimits.push({    id:    this.generateId(this.expenseLimits),
title, currentValue: 0, limit });
    this.updateAllocation();
  }
  updateExpenseLimit(id: number, value: number) {
    const limit = this.expenseLimits.find((limit) => limit.id === id);
    if (!limit) throw new Error(`Expense with ID ${id} not found`);
    if (value < 0) throw new Error("Expense value cannot be negative");
    limit.currentValue = value;
    this.updateAllocation();
  }

  // Savings
  getSavings() {
    return [...this.savings];
  }
  addSavings(description: string, amount: number) {
    if (amount <= 0) throw new Error("Savings amount must be greater than
0");
    this.savings.push({  id:  this.generateId(this.savings),  description,
amount });
    this.updateAllocation();
  }

  // Budget Allocation
  getAllocation() {
    return [...this.allocation];
  }
  private updateAllocation() {
    const totalIncome = this.calculateTotal(this.incomeSources, "amount");
```

```
    const totalSavings = this.calculateTotal(this.savings, "amount");
    const    totalExpenses    =    this.calculateTotal(this.expenseLimits,
"currentValue");

    this.allocation = [
      { id: "income", label: "Income", value: totalIncome, color: "#4caf50"
},
      { id: "savings", label: "Savings", value: totalSavings, color:
"#2196f3" },
      { id: "expenses", label: "Expenses", value: totalExpenses, color:
"#f44336" },
    ];
  }
```

Лістинг 3

```
  updateBill(id: number, updatedFields: Partial<Omit<Bill["bills"][number],
"id" | "userId">>) {
    const bill = this.bills.find((bill) => bill.id === id);
    if (!bill) throw new Error(`Bill with ID ${id} not found`);
    Object.assign(bill, updatedFields);
    this.updateAllocation();
  }
  // Payment History
  getPaymentHistory() {
    return [...this.paymentHistory];
  }
  recordPayment(billId: number, amount: number) {
    const bill = this.bills.find((bill) => bill.id === billId);
    if (!bill) throw new Error(`Bill with ID ${billId} not found`);
    if (amount <= 0) throw new Error("Payment amount must be greater than
0");
    this.paymentHistory.push({
      id: this.generateId(this.paymentHistory),
      billId,
      date: new Date(),
      amount,
    });
    console.log(`Payment     of     ${amount}     recorded     for     bill
"${bill.billName}"`);
  }
  // Scheduled Payments
  getScheduledPayments() {
    return [...this.scheduledPayments];
  }
  schedulePayment(billId: number, scheduledDate: Date) {
    const bill = this.bills.find((bill) => bill.id === billId);
    if (!bill) throw new Error(`Bill with ID ${billId} not found`);
    if (scheduledDate <= new Date()) throw new Error("Scheduled date must be
in the future");
    this.scheduledPayments.push({
      id: this.generateId(this.scheduledPayments),
      billId,
      scheduledDate,
    });
    console.log(`Payment   for   bill   "${bill.billName}"   scheduled   on
${scheduledDate}`);
  }
  // Notifications & AutoPay
```

```
  toggleNotification(billId: number) {
    const bill = this.bills.find((bill) => bill.id === billId);
    if (!bill) throw new Error(`Bill with ID ${billId} not found`);
    bill.notificationEnabled = !bill.notificationEnabled;
    console.log(
      `Notifications for bill "${bill.billName}" ${
        bill.notificationEnabled ? "enabled" : "disabled"
      }`
    );
  }
  toggleAutoPay(billId: number) {
    const bill = this.bills.find((bill) => bill.id === billId);
    if (!bill) throw new Error(`Bill with ID ${billId} not found`);

    bill.autoPayEnabled = !bill.autoPayEnabled;
    console.log(
      `Autopay for bill "${bill.billName}" ${
        bill.autoPayEnabled ? "enabled" : "disabled"
      }`
    );
  }
  // Allocation
  getAllocation() {
    return [...this.allocation];
  }
  private updateAllocation() {
    const totalAmount = this.calculateTotal(this.bills, "amount");
    const totalPayments = this.calculateTotal(this.paymentHistory,
"amount");
    this.allocation = [
      { id: "bills", label: "Total Bills", value: totalAmount, color:
"#ff9800" },
      { id: "payments", label: "Total Payments", value: totalPayments, color:
"#4caf50" },
    ];
  }
```

Лістинг 4

```
makePayment(debtId: number, amount: number) {
    const debt = this.debts.find((d) => d.id === debtId);
    if (!debt) throw new Error(`Debt with ID ${debtId} not found`);
    if (amount <= 0) throw new Error("Payment amount must be greater than
0");
    if (amount > debt.balance) throw new Error("Payment exceeds debt
balance");

    debt.balance -= amount;

    this.paymentHistory.push({
      id: this.generateId(this.paymentHistory),
      debtId,
      paymentAmount: amount,
      paymentDate: new Date(),
    });

    this.updateAllocation();
    console.log(`Payment of ${amount} made to creditor "${debt.creditor}"`);
  }
```

```
  calculateInterest(debtId: number): number {
    const debt = this.debts.find((d) => d.id === debtId);
    if (!debt) throw new Error(`Debt with ID ${debtId} not found`);

    return (debt.balance * debt.interestRate) / 100;
  }

  trackProgress(debtId: number): string {
    const debt = this.debts.find((d) => d.id === debtId);
    if (!debt) throw new Error(`Debt with ID ${debtId} not found`);

    return ((debt.minimumPayment / debt.balance) * 100).toFixed(2);
  }

  // Debt Overview
  generateDebtOverviewData() {
    return this.debts.map((debt) => ({
      id: debt.id,
      creditor: debt.creditor,
      debtType: debt.debtType,
      balance: debt.balance.toFixed(2),
      interestRate: `${debt.interestRate}%`,
      minimumPayment: debt.minimumPayment.toFixed(2),
      dueDate: debt.dueDate.toDateString(),
    }));
  }

  generatePaymentHistoryData() {
    return this.paymentHistory.map((payment) => {
      const debt = this.debts.find((d) => d.id === payment.debtId);
      return {
        debtId: payment.debtId,
        creditor: debt?.creditor || "Unknown",
        paymentAmount: payment.paymentAmount.toFixed(2),
        paymentDate: payment.paymentDate.toDateString(),
      };
    });
  }

  // Debt Ratios
  calculateDebtBurdenRatio(): string {
    const totalDebt = this.calculateTotal(this.debts, "balance");
    const     totalMinimumPayment     =     this.calculateTotal(this.debts,
"minimumPayment");

    return ((totalMinimumPayment / totalDebt) * 100).toFixed(2);
  }

  calculateDebtToIncomeRatio(income: number): string {
    if (income <= 0) throw new Error("Income must be greater than 0");

    const totalDebt = this.calculateTotal(this.debts, "balance");
    return ((totalDebt / income) * 100).toFixed(2);
  }
```

Лістинг 5

```
makePayment(debtId: number, amount: number) {
```

```
generateReport(dataSources: { budgets?: Budget[]; debts?: Debt[] }): void {
    try {
      if (!dataSources || (!dataSources.budgets && !dataSources.debts)) {
        throw new Error("No data sources provided to generate the report.");
      }

      if (this.reportType === "Budget Comparison" && dataSources.budgets) {
        this.data = dataSources.budgets.map((budget) => ({
          category: budget.category,
          planned: budget.amount,
          actual: budget.calculateRemaining(),
        }));
      } else if (
        this.reportType === "Debt Overview" &&
        dataSources.debts
      ) {
        this.data = dataSources.debts.map((debt) => ({
          creditor: debt.creditor,
          balance: debt.balance,
          minimumPayment: debt.minimumPayment,
        }));
      } else {
        throw new Error("Unsupported report type or missing data source.");
      }

      console.log(`${this.reportType} report generated successfully.`);
    } catch (error) {
      console.error(`Failed to generate report: ${error.message}`);
    }
  }

  viewReport(): void {
    try {
      console.log(`${this.reportType} report viewed:`, this.data);
    } catch (error) {
      console.error(`Failed to view report: ${error.message}`);
    }
  }

  exportReport(format: string): void {
    try {
      if (!["PDF", "CSV"].includes(format)) {
        throw new Error("Unsupported export format.");
      }
      console.log(`${this.reportType}   report   exported   in   ${format}
format.`);
    } catch (error) {
      console.error(`Failed to export report: ${error.message}`);
    }
  }
}
```

ДОДАТОК В

Диск з розробленою програмою