School of Computing and Mathematical Sciences

CO7201 Individual Project

Final Report Template

Finance Tracker

Vivek Shah
vps5@student.le.ac.uk
Student Id - 239062179

Project Supervisor: Dr. Babajide Afeni
Principal Marker: Dr. Craig Bower

Word Count: 17775
Submission Date – 16/05/2025

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Vivek Shah
Date: 16/05/2025

Abstract

This dissertation presents the design, development, and testing of Finance Tracker, an Android mobile app to aid users in managing personal finances. The application aims to make daily financial management easier by allowing users to track income and expenses, manage monthly budgets, and comprehend expenditure patterns through interactive data visualization. It was developed in Kotlin and Jetpack Compose and adheres to Clean Architecture principles to ensure modularity, scalability, and maintainability.

To enable robust data processing and instant sync across devices, the app uses both Room Database as local storage and Firebase Firestore as cloud storage. User authentication via email/password and Google Sign-In, multi-currency with real-time exchange rate conversion, and category-wise income and expense tracking are some of the key features. The app offers graphical summaries in the form of charts and graphs to allow users to monitor their financial trends and make sound decisions. Its development involved iterative manual testing and introducing unit testing of key application modules, verifying functional correctness and increasing the reliability of key business logic. It gathered feedback from users via hands-on testing, which indicated the app's high usability, consistency of performance, and simplicity of interface design. Finance Tracker in its current state effectively delivers its major features such as setting up user profile, adding transactions, and budget management. However, the application realizes that personal finance management is a dynamically evolving field considering user requirements and technological advancements. Therefore, the application is designed with extensibility in mind.

Future development will be directed towards feature along with technical enhancement. Pipeline additions include support for recurring transactions to enable users to automate the entry of regular expenses or income such as rent, subscriptions, and salaries. The visualization capabilities will also be extended with the addition of line charts to track financial trends over time. A more detailed category-based budgeting system will be included to allow users to impose specific budget constraints on individual expense categories, thus improving financial discipline. Technically, ongoing improvements will involve refactoring for even stricter adherence to Clean Architecture principles, improved testability, scalability, and long-term maintainability. More extensive unit testing and UI testing will be prioritized to ensure that future improvements do not break existing functionalities. Lastly, the application roadmap also includes the potential addition of a custom backend service using technologies like Spring Boot or Ktor. This will allow finer control over data processing so that advanced features like personalized insights, predictive analytics using machine learning, and intelligent notification handling can be added.

Table of Contents

List of Figures

List Of Tables

1 Introduction

1.1 Background And Motivation

My interest in application development began when I started working on basic app projects during my earlier studies. Later, in conversation with a friend, I learned that he managed his expenses by manually recording them in notes. This inspired me to think about a more organized and efficient way to manage personal finances through a dedicated application.

During my master's studies, I also took a subject on mobile application development, where I developed a petition management app using XML and Java. This experience further strengthened my interest in the field. After completing that project, I decided to work on an expense management application for myself.

Moving to a new country as a student and managing expenses while balancing part-time work proved to be challenging. This personal need became the primary motivation for building an expense tracking app to help manage my finances more effectively.

While researching the latest trends in app development, I found that Java is gradually being phased out in favor of Kotlin, which is now the preferred and future language for Android development. Additionally, I discovered that Jetpack Compose offers a more streamlined and efficient approach to designing user interfaces compared to the traditional XML method.

One might question why I chose to develop a new app instead of using an already existing expense management application. Although there are many apps available in the market, most either come with limited free features, intrusive advertisements, or require paid subscriptions for full access. I wanted an application that could be tailored exactly to my needs — simple, lightweight, and free of unnecessary complexities. Moreover, building my own app allowed me to apply my learning practically, explore modern development tools, and create a project that would contribute to my professional portfolio.

Driven by the desire to learn modern technologies, strengthen my skills, and lay a strong foundation for my future career, I decided to learn Kotlin and Jetpack Compose and use them to build this project.

This blend of personal necessity, academic learning, and future career aspirations formed the core motivation behind this project.

1.1.1 Problem Statement

In today's fast-paced world, managing personal finances has become increasingly complex, particularly for individuals who struggle to keep track of their income, expenses, and savings. Many people often rely on manual methods, such as note-taking or spreadsheets, which are prone to errors, time-consuming, and lack accessibility. Additionally, numerous finance tracking applications available in the market either come with complex user interfaces, limited free features, require paid subscriptions for full access, or fail to provide essential functionalities like multi-currency support and offline accessibility.

Furthermore, many of the existing apps fail to address privacy concerns, often collecting excessive user data or requiring constant internet access, leaving users vulnerable to data breaches or lack of privacy control. Users need a more efficient, secure, and customizable solution that empowers them to manage their finances seamlessly, without the constraints of ads, subscription costs, or complex setups. There is also a growing demand for applications that provide real-time data visualization, secure cloud synchronization, and offline access, which many existing apps do not adequately support.

Thus, the problem arises from the lack of a comprehensive, secure, and user-friendly mobile solution that allows users to efficiently manage their finances, track their spending habits, set and monitor budgets, and gain insights into their financial situation in a personalized and accessible manner.

This project aims to solve these challenges by developing a Finance Tracker App that is easy to use, provides essential features for effective financial management, and ensures security and privacy while being accessible offline and on multiple devices.

1.2 Aim And Objectives

In today's fast-paced world, managing personal finances effectively has become a critical necessity. Many individuals struggle with tracking their income, expenses, and savings, leading to overspending and inadequate financial planning. While numerous finance tracking applications are available, most are either too complex, require subscriptions, or lack essential features such as multi-currency support and offline accessibility. This project aims to address these shortcomings by developing a user-friendly, secure, and feature-rich Finance Tracker App designed to streamline the management of personal finances.

1.2.1 Relevance of Project

The need for a personalized and efficient finance tracking solution is driven by the following factors:

i. **Growing reliance on digital solutions:** Mobile apps have become the primary medium for individuals to manage various aspects of their daily lives, including financial activities. The increasing reliance on mobile devices for financial management highlights the demand for accessible, intuitive apps.
ii. **Need for better financial awareness:** Many individuals lack the tools to effectively track and analyze their spending patterns, leading to overspending, poor budgeting, and limited savings. Providing an accessible and easy-to-use finance tracker can significantly improve financial literacy and empower users to make better financial decisions.
iii. **Accessibility and privacy concerns:** While many existing finance apps require constant internet access, this limits usability for individuals in offline situations. Additionally, privacy concerns surrounding the collection of personal data by these apps emphasize the need for secure, offline-capable solutions.
iv. **Customization and flexibility:** Every individual has unique financial habits and needs. A finance tracker that offers customizable categories, multi-currency support, advanced filtering options, and data export features can cater to a wider range of user preferences, enhancing their overall financial management experience.

1.2.2 Aim

The primary goal of this project is to develop a secure, intuitive, and feature-rich mobile application that helps users manage their personal finances efficiently. The app will provide expense tracking, budgeting tools, financial data visualization, and multi-device synchronization to enhance user experience and financial awareness.

1.2.3 Objectives

To achieve the aim outlined above, the specific objectives of the project are as follows:

1. **Develop a simple, user-friendly interface** using Jetpack Compose to facilitate easy tracking of expenses and income.
2. **Implement robust budgeting features** that allow users to set monthly budgets and monitor their spending habits.
3. **Provide dynamic financial visualizations** (such as charts and graphs) to enhance users' understanding of their financial data.
4. **Integrate multi-currency support**, allowing users to manage their finances in different currencies with live or cached exchange rates.
5. **Ensure offline functionality** so that users can add and view transactions even when there is no internet connection.
6. **Implement secure authentication mechanisms** (e.g., email/password login and Google Sign-In) to ensure the protection of user data.
7. **Enable data synchronization across multiple devices** via secure cloud storage (Firebase integration) to ensure seamless access to financial data from any device.
8. **Allow the export of financial data** in CSV or PDF formats for offline backup or external analysis.
9. **Maintain high standards of privacy and data security**, minimizing unnecessary data collection and providing users with control over their personal information.
10. **Conduct thorough testing and evaluation** of the application to ensure its reliability, performance, and overall user satisfaction.

1.3 Structure Of Dissertation

Explain briefly what each chapter will contain:

- **Chapter 1: Introduction(this)** - This chapter introduces the project, outlining its background, objectives, and significance. It provides an overview of the problem being addressed and sets the foundation for the report.
- **Chapter 2: Literature Review -** This chapter discusses previous research, existing solutions, and technologies relevant to the project. It highlights gaps and how the current work aims to address them.
- **Chapter 3 Requirement Analysis -** This chapter details the functional and non-functional requirements identified during the planning phase, forming the basis for system development.
- **Chapter 4: System Design and Architecture -** This chapter describes the design approach, including system architecture diagrams, data flow, and design principles used to create an efficient and scalable solution.
- **Chapter 5: Technologies and Tools Used -** This chapter lists and explains the programming languages, frameworks, libraries, and tools utilized during the development of the project.
- **Chapter 6: Implementation -** This chapter provides a detailed explanation of how the system was developed, including key features, coding practices, and integration of various modules.
- **Chapter 7: Testing and Evaluation -** This chapter discusses the testing strategies employed to verify the system's functionality, performance, and reliability, along with evaluation results.
- **Chapter 8: Challenges and Solutions** - This chapter outlines the major challenges encountered during the project and the strategies or solutions adopted to overcome them.
- **Chapter 9: Future Work -** This chapter suggests potential enhancements, optimizations, and additional features that could be explored to further improve the system.
- **Chapter 10: Conclusion -** This chapter summarizes the overall work, reflecting on the achievements, learnings, and the successful fulfillment of the project objectives.
- **Chapter 11: References -** This chapter lists all the academic papers, articles, books, websites, and other resources referenced throughout the project.
- **Chapter 12: Appendices -** This chapter includes supplementary materials like diagrams, additional data, code snippets, or documentation that supports the main content but is too detailed for the main body.

2 Literature Review

Managing personal finances has increasingly become a digital activity, leading to the development of various financial tracking applications. While each application or study offers valuable insights, significant gaps still exist, particularly concerning accessibility, customization, and real-world usability for everyday users.

This paper introduces an Android application designed to support higher-level management, staff, sales personnel, and other stakeholders by providing mobile access to business analytics. The application presents organizational data through various types of visualizations, such as area charts, bar charts, D3 charts, and integrations with Google Maps. By offering these analytical insights directly on mobile phones and tablets, the application aids industries such as insurance, manufacturing, and banking in strategic decision-making and operational planning. Business Intelligence (BI) solutions, when integrated into mobile platforms, can significantly enhance organizational efficiency by enabling the measurement and monitoring of key performance metrics, including sales revenue, product sales, departmental expenditures, and supply chain operations. Traditionally, large volumes of organizational data were maintained in extensive Excel sheets, making analysis time-consuming and complex. This application addresses that challenge by providing intuitive, real-time data visualization tools, thereby simplifying the decision-making process and promoting better strategic outcomes. (Oswal, 223)

Recent studies have highlighted a growing interest in personal finance and budgeting, largely driven by the financial challenges brought on by the COVID-19 pandemic. This project aims to address this demand by developing an easy-to-use mobile application designed to support users in managing their finances effectively. The application features an intuitive user interface that encourages consistent budgeting habits and financial awareness. Additionally, a recommender system within the app offers personalized advice and clear visualizations of spending, helping users better understand and control their expenses. Many individuals, especially newcomers to managing their finances, struggle with basic financial concepts such as credit applications and debt management. This proposed application seeks to fill this gap by providing a practical tool for students and adults to start their budgeting journey, while also contributing to the growing body of literature on mobile application development in the field of personal finance and budgeting. (Wong, 2023)

This study presents the design, development, and evaluation of an Android-based personal finance management application, specifically aimed at young adults in higher education. Recognizing the unique financial challenges faced by this demographic, such as limited financial experience and fluctuating income, the application includes key features such as income tracking, expense monitoring, budgeting, and financial goal setting. Built using the Waterfall model, the app ensures secure logins, offers an intuitive transaction management system, and provides customizable goals, budget projections, and automated reminders to promote better financial habits. Usability testing, conducted with 50 users using a 5-point Likert scale, revealed an overall satisfaction score of 4.6/5, categorized as 'Excellent.' Users appreciated the app's user-friendly interface, precise tracking capabilities, and motivational reminders, though they also suggested incorporating more customization options and advanced financial analysis in future updates. This study highlights the

effectiveness of digital tools in enhancing financial literacy and resilience, demonstrating that personalized technology can positively influence financial behaviors in young adults. Future research will focus on adding more customization features and integrating AI-driven capabilities to enhance the application's impact. (Imawan, 2025)

This study represents the first attempt to evaluate whether smartphone applications can effectively improve financially capable behaviors. In this research, four smartphone apps, collectively branded as ‘Money Matters,’ were distributed to working-age individuals (16–65 years) of the largest credit union in Northern Ireland (Derry Credit Union). The suite of apps included a loan interest comparison app, an expenditure comparison app, a cash calendar app, and a debt management app. The assessment methodology involved a Randomized Control Trial (RCT), using the U.K. Financial Capability Outcome Frameworks to set the context for the evaluation. The treatment group, which received the apps, showed statistically significant improvements in various measures related to 'financial knowledge, understanding, and basic skills,' as well as 'attitudes and motivations.' These improvements led to enhanced financial behaviors, with participants demonstrating a greater tendency to track their income and expenses and increased resilience when faced with financial challenges. (French, 2021)

The Expense Tracker app provides an intuitive solution for managing both income and expenses, allowing users to track their spending on a daily, weekly, monthly, or annual basis. With features such as selecting expense categories, adding location details, and uploading additional data, users can customize their entries easily. The app stores this information in a relational database, enabling users to view and sort their expenditures over different time periods. This functionality reduces the need for manual calculations, offering an efficient way to monitor personal finances. The app also provides users with the ability to enter their income to estimate daily expenses, storing these results for personalized tracking. Ideal for individuals who frequently travel or engage in social activities, the app enables users to track group expenses and split bills effortlessly. The tracker generates graphical representations of spending based on the chosen time frame. Additionally, users can set their monthly income or expenditure limits to maintain better control over their financial habits. The integrated features of this tracker provide a comprehensive tool for managing spending and cash flow. (Girdhar, 2024,September)

Across the reviewed literature, a common limitation persists: while technological innovations are extensively explored, accessibility, user-friendliness, and offline capabilities are often overlooked. Many existing solutions assume technical literacy, consistent internet access, and stable income patterns—assumptions that do not hold true for many real-world users. Therefore, there remains a critical need for a simple, intuitive, secure, and customizable finance tracking application that caters to a broader and more diverse audience.

3 Requirements Analysis

The requirements for the Finance Tracker application were categorized into essential, desirable, and optional requirements...

3.1 Essential Requirements

These are the core features that the Finance Tracker App must have to fulfil its primary purpose.

- **User Authentication:** Users must be able to register and log in using email/password authentication.
- **Google Sign-In:** Support for Google authentication to provide an easier sign-in process.
- **Expense and Income Management:** Users should be able to add, edit, and delete expenses and income with details such as amount, category, and date.
- **Budget Management:** Implement budget tracking, allowing users to set and monitor monthly budgets.
- **Local Data Storage:** Store financial data locally using Room Database to enable offline access.
- **Data Visualization:** Provide charts and graphs to help users analyze their income and expenses.
- **Secure Data Storage:** Ensure proper encryption and authentication to protect sensitive financial data.

3.2 Desirable Requirements

These features will enhance usability and provide additional functionality to improve the user experience.

- **Cloud Synchronization:** Implement Firebase integration to allow users to sync financial data across multiple devices.
- **Notifications & Reminders:** Send alerts to notify users about budget limits and upcoming expenses.
- **Transaction Filtering & Searching:** Allow users to filter and search transactions by category, date, or amount for quick access.
- **Multi-Currency Support:** Integrate a currency conversion feature to support transactions in different currencies.
- **User-Friendly UI:** Design an intuitive and visually appealing interface to ensure smooth navigation and usability.

3.3 Optional Features

These features are not mandatory but would provide additional benefits and improve the overall experience.

- **Data Export**: Enable users to export financial records in CSV and PDF formats for external use.
- **Receipt Scanning (OCR)**: Implement Optical Character Recognition (OCR) to allow users to scan receipts and auto-fill expense details.
- **Fingerprint Authentication:** Allow biometric login for enhanced security and quick access.

- **Multiple Account Support**: Let users manage different financial accounts (e.g., personal and business).
- **Dark Mode**: Introduce a dark mode theme for better accessibility and user preference.

4 System Design and Architecture

4.1 Architectural Overview

The application follows the Clean Architecture pattern, where the codebase is divided into three main layers: Presentation, Domain, and Data. Each layer has a specific role, contributing to enhanced maintainability and scalability. The Domain layer acts as the core of the application, encapsulating all business rules and application-specific logic. This layer contains only the interfaces, making it independent of both the user interface and data sources. The Presentation layer has the MVVM (Model-View-ViewModel) architecture, where ViewModels manage the UI state and handle user inputs by calling the appropriate use cases from the Domain layer. The Data layer handles all data operations, serving as the bridge between the domain logic and data sources like Room (local) and Firebase (remote). It provides the implementations of the repository interfaces defined in the Domain layer.

This structure follows the idea of keeping important parts of the app (like core logic) separate from the details (like how data is stored or shown). This means changes in things like the database or UI won't affect the main logic. Because of this, the app is easier to build in parts, test, and update without breaking other things.

4.2 Clean Architecture Layers

As mentioned earlier, Clean Architecture is divided into three parts: data, domain, and presentation, each responsible for a different aspect of the app. Below is a more detailed explanation of each part.

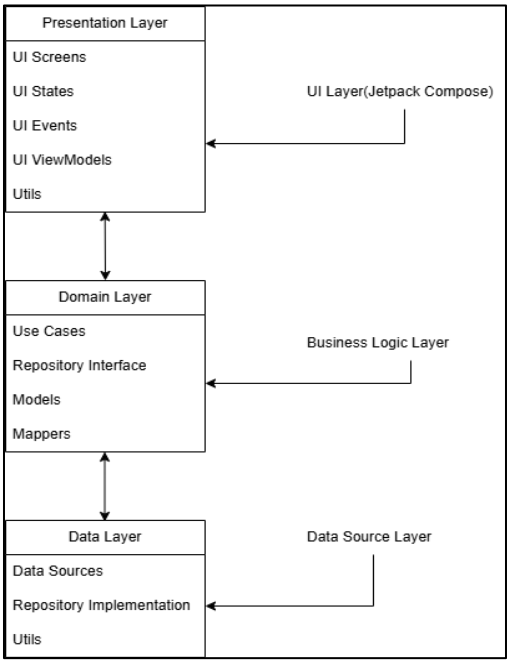


Figure 4-1 Clean Architecture Layers

4.2.1 Presentation Layer

- **Purpose:**
  This is the part of the app that deals with user interaction. It includes all the screens and UI elements built using Jetpack Compose. The main job of this layer is to show the data to the user and collect their input.
- **Components:**
  - UI (Composable): These are the building blocks of the screen like buttons, text fields, and layouts.
  - UI State: Data classes that hold all the information needed to display the screen properly. Think of it as a snapshot of everything that should be shown on the screen.
  - UI Events: These are sealed classes that describe what the user does—like clicking a button or entering text.
  - ViewModel: It acts as a bridge between UI states and UI events. It listens to events, updates the UI state, and calls business logic (use cases).
  - Utils: This module typically contains helper functions and utility classes that perform common tasks across the application. These tasks include operations like displaying data in a user-readable format (e.g., date and time formatting), providing data classes for filtering or search functionality, performing conversions, or executing validation checks. The purpose of the Utils module is to promote code reusability, reduce duplication, and maintain a clean and modular codebase.

4.2.2 Domain Layer

- **Purpose:**
  This layer is complete business logic of the app. It answers the question: “What should the app do?” independent of how it looks or where the data comes from. This layer is pure Kotlin—it doesn’t depend on Android or any external libraries. That makes it easy to test and reuse.
- **Components:**
  - UseCases: These are the tasks that the app can perform, like adding a transaction, fetching a user's profile, or calculating a budget. Each use case focuses only on one thing. They act like a bridge between viewmodel in presentation layer and repository implementation in data layer.
  - Repository Interfaces: These define the rules for how the app gets or saves data, but don’t care whether it's coming from Firebase, Room, or somewhere else.

4.2.3 Data Layer

- **Purpose:**
  This is where managing the data for the app is actually done. It deals with the databases or cloud services and serves as the bridge between the domain and the actual data sources. This layer uses Android and third-party libraries to get real work done.

- **Components:**
    - Repository Implementations: These provide the real code behind the interfaces defined in the domain layer. They know how to fetch data from local database or firebase.
    - Data Source: Generally, it contains the local database logics and source like Room Entity, Room Dao, Room Database file.
    - Utils: It generally contains the local database migration logic i.e. if we update our database or transform our database to new database.

4.3 Data Flow and Communication Between Layers

The data flow in the application follows a clean and modular architecture to ensure scalability, testability, and maintainability. When a user interacts with the UI, it triggers the corresponding function in the ViewModel, which acts as a bridge between the UI and business logic. The ViewModel then triggers the task to a specific UseCase, which contains the business rules and logic of the feature. The UseCase communicates with the Repository, which abstracts the source of data. Depending on the operation and availability, the repository fetches or saves data through either a LocalDataSource (using Room Database for offline support) or a RemoteDataSource (using Firebase Firestore for cloud synchronization). Once the data is retrieved or updated, the response is propagated back up the layers, eventually updating the UI state via the ViewModel. This structured flow ensures a clean separation between UI and data handling, improving maintainability and reliability of the application.

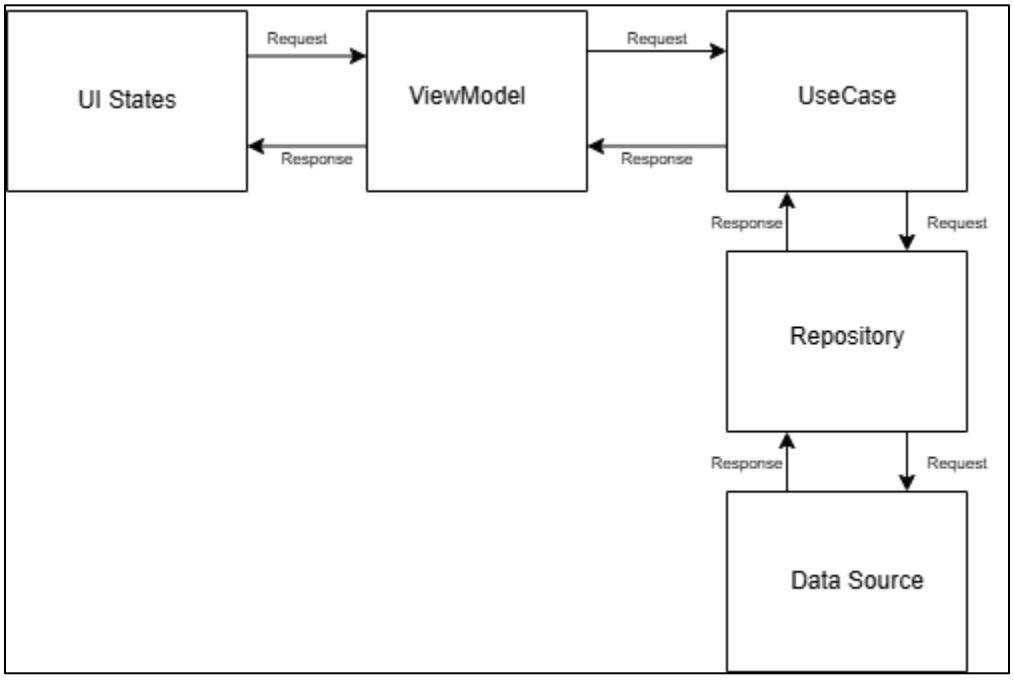


Figure 4-2 Data Flow

4.4 Application Flow

When the user first opens the application, they are directed to the Startup Page, which presents them with two primary options: Login or Register. This is the entry point for both new and returning users. If the user already has an account, they can log in either using their registered email and password or by selecting the Google Sign-In option. Google Sign-In provides a quicker, more seamless login experience by leveraging Firebase Authentication.

For new users, upon successful registration or Google Sign-In, the app automatically redirects them to the Account Setup screen. This screen serves as an initial configuration stage where the user is required to enter basic profile details such as their name and region (country). As soon as the user selects their region, an API call is triggered to fetch the corresponding country name, currency name, currency code, and dialing code. This data is then stored in the local Room Database for offline access and to support functionalities like currency conversion and user profile display.
Once the setup is completed (or skipped if the user has already configured their account), the user is taken to the Home Page. This screen serves as the central hub for accessing all core features of the application. The bottom navigation bar enables users to smoothly switch between different screens of the app. From the Home Screen, users can navigate to various parts of the app via the bottom navigation bar, which ensures smooth and consistent access to core features. These include:

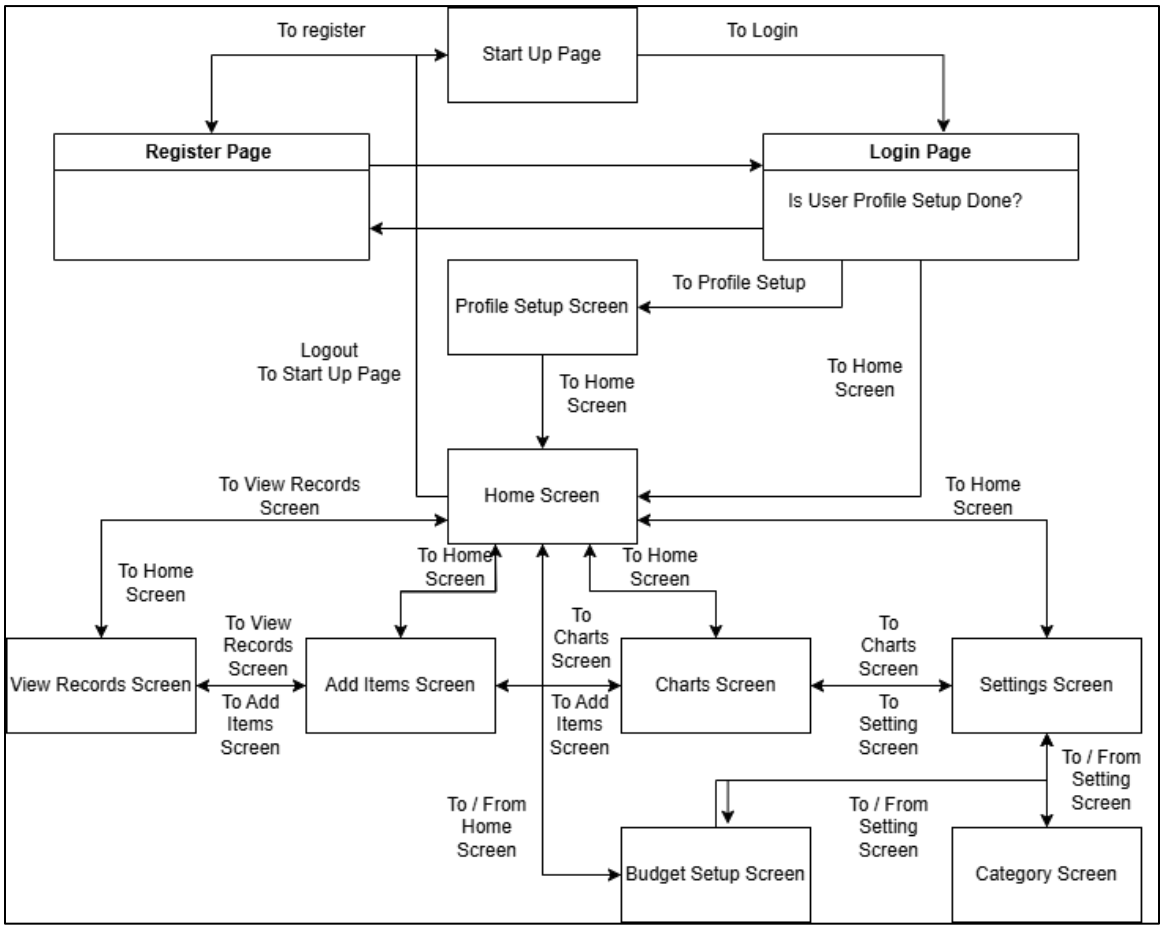


Figure 4-3 Application Flow

1) **Home Screen:** The Home Screen provides a quick overview of the user's financial status, displaying a progress bar representing the user's spending relative to their set monthly budget. It also shows a concise account summary, including total income, total expenses for the month and account summary. This helps users track their finances at a glance.

2) **Add Item Screen:** This screen allows users to log income or expense transactions manually or choose from previously saved items. Each transaction includes fields such as amount, currency, category, date, notes, and optionally a recurring flag. If the transaction currency differs from the user's base currency, the app automatically fetches the real-time exchange rate and calculates the converted amount. This helps users manage multi-currency finances with ease.

3) **View Records Screen:** Users can explore all recorded transactions and saved items in this section. The list is filterable and sortable by transaction type, date, or category, providing flexibility in tracking and reviewing past records. This screen aids users in analyzing their financial behavior and managing their spending effectively.

4) **Charts Screen:** This screen provides graphical insights into user finances through pie charts and bar graphs. Users can view a visual breakdown of income and expenses over time or by category. This helps identify spending patterns and assists in financial planning and decision-making.

5) **Settings Screen:** In the settings section, users have full control over their account and app preferences. They can update profile information (like name, region, base currency), toggle app themes (light/dark), add or manage categories for income and expenses, set a monthly budget, or log out. All user-related changes are stored and synced in Firebase Firestore, ensuring consistency across devices.

Each screen is seamlessly connected through the navigation system, offering an intuitive and efficient user experience. Whether users want to review their budget, analyze trends, or enter a new transaction, they can do so with minimal effort. The thoughtful integration of local and cloud storage ensures reliability, while the clean architecture design keeps each layer of functionality modular and maintainable.

4.5 Scalability and Maintainability

Clean Architecture plays a crucial role in making software scalable and maintainable, especially in long-term projects or apps expected to grow over time. It organizes the code into layers—typically including domain, data, and presentation layers—allowing each part to focus on a single responsibility. This separation makes it easier to modify one part of the application without affecting others.

For example, in finance tracker app, if want to switch from Firebase to another cloud service or your own backend, Clean Architecture allows you to make that change in the data layer only. The

rest of your app—UI, use cases, and business rules—remains untouched. This level of modularity is essential for scalability, as you can expand or swap components as your app evolves without rewriting the whole system.

In terms of maintainability, Clean Architecture encourages writing testable and reusable code. Each layer can be independently tested, making debugging and updating easier. For example want to add a new feature, like OCR receipt scanning. With Clean Architecture, simply introduce a new use case and update the necessary layers, without disturbing existing logic.

Overall, using Clean Architecture improves developer productivity, reduces bugs, and makes onboarding new developers easier. It's especially valuable in team environments or when maintaining the project over time. By enforcing a clear structure, it prepares the app for future growth—whether that's adding new features, supporting more users, or integrating new technologies—making your app robust and future-proof.

5 Technologies and Tools Used

Category	Technology/Tool	Reason for Selection
Programming Language	Kotlin	Official language for Android, modern and concise
UI Framework	Jetpack Compose	Declarative UI, efficient, and recommended by Google
Database	Room Database	Provides an abstraction over SQLite, supports offline access
Authentication	Firebase Authentication	Secure and easy-to-implement authentication (Email/Google)
Cloud Storage	Firebase Firestore	Real-time synchronization and cloud backup
State Management	View Model & Live Data	Manages UI-related data efficiently
Networking	Retrofit	Simplifies API calls (e.g., currency conversion API)
Dependency Injection	Hilt (Dagger) / Koin	Improves modularity and testability
Background Tasks	Coroutines & Flow	Efficient async programming, recommended for Kotlin
Charts & Graphs	MPAndroidChart	Popular library for interactive data visualization
Multi-Currency Support	Exchange Rate API (e.g., OpenExchangeRates, Fixer.io)	Fetches real-time exchange rates for currency conversion
Data Export	Apache POI / OpenCSV	Enables CSV and PDF export for financial reports
User Preferences	SharedPreferences	Saves user setting preferences
Push Notifications	WorkManager	Sends budget reminders and transaction alerts.
Development Environment	Android Studios	Official IDE for Android development, offers a rich set of tools, debugging features, and seamless integration with Android SDK and Kotlin.
Testing	JUnit	Used for writing unit tests for core business logic. JUnit helps in identifying bugs early, ensures components work independently, and improves overall app stability.

Table 1 : Tools and Technology

6 Implementation

In the Implementation chapter of the report, I would detail the process of how your mobile finance tracker app was developed and implemented. This chapter should include technical information, code implementation, design decisions, challenges faced, and how the features were brought to life.

6.1 Front End Development (UI) Implementation

6.1.1 Architecture

In the development of the mobile finance tracker application, the MVVM (Model-View-ViewModel) pattern was implemented in combination with Clean Architecture to ensure a scalable, maintainable, and testable codebase. This architectural approach was selected due to its clear separation of concerns, which simplifies the development process and supports modular design, allowing individual layers to evolve independently.

The application is structured into three primary layers: Presentation, Domain, and Data. The Presentation Layer is responsible for the user interface and interaction logic. It is built using Jetpack Compose, a modern declarative UI toolkit that allows for dynamic and efficient UI rendering. Within this layer, ViewModels are used to manage UI-related data and business logic. They expose state to the UI using State, MutableState, and StateFlow, enabling reactive UI updates. User interactions trigger events that the ViewModel processes asynchronously using Kotlin Coroutines, ensuring a smooth and responsive experience. The Domain Layer contains the core business logic and is structured around use cases, which encapsulate specific operations such as adding transactions, validating inputs, or retrieving user preferences. This layer acts as a mediator between the UI and the data, providing a clean interface for interacting with the underlying logic. The Data Layer handles all data-related operations and consists of repositories that interact with both Firebase for cloud data storage and Room for local database management. The use of abstraction through interfaces ensures loose coupling and flexibility in switching or combining data sources.

Hilt, a dependency injection library, is integrated to manage and provide dependencies across layers efficiently, reducing boilerplate code and enhancing testability. This overall architecture enhances the clarity, reliability, and adaptability of the application, making it easier to maintain and extend in future development cycles.

6.1.2 Development Process

The application is designed using the Material Design 3 (Material You) for a better, consistent and modern user experience. Jetpack Compose is used to build the user interface, allowing for a clean, responsive layout that adapts well across different screen sizes and orientations. By using a Stateflow UI approach, it becomes easier to manage UI state and update components dynamically as the app data changes. A well-defined MaterialTheme is applied across the app, which includes a carefully chosen color scheme and typography to maintain visual consistency. The color theme supports both light and dark modes and provides a soft, accessible contrast that enhances readability. The app's typography is structured using predefined text styles such as headings, body text, and labels, giving the UI a clear hierarchy and helping users focus on important content.

Standard Material components like Button, Card, Surface, TopAppBar, BottomNavigationBar, and OutlinedTextField, Text, TextButton, AlertBox, BottomSheet are used throughout the app to create a familiar and intuitive experience. These components are styled using the app's theme, including consistent padding, elevation, and rounded corners, aligning with Material Design 3's guidelines. Overall, the design aims to be simple, clean, and user-friendly, while ensuring consistency and scalability as more features are added over time. The development started with creating the navigation system, a component responsible for managing the app's redirections and transitions between screens.

6.1.3 Startup Screen

This screen is the application's entry point. User gets the two options to navigate which are Login, Registration. From here if user have account he can go to login screen and login their account authenticating login details. If user have no account and if he is new, he can navigate to Registration Screen and register their account and then login to their account.

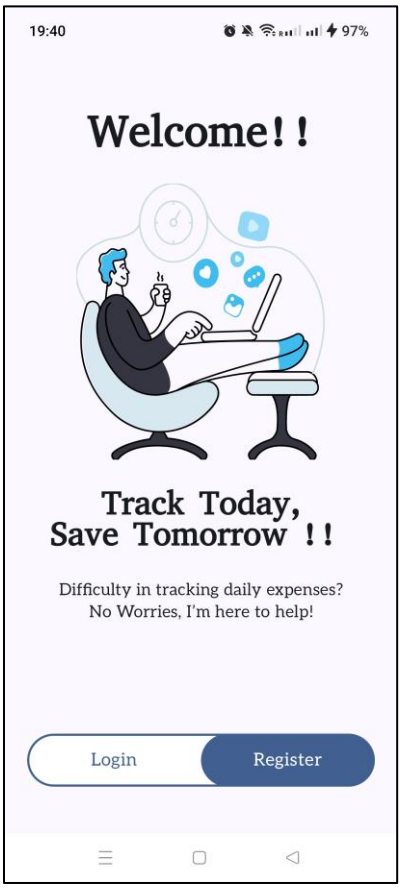


Figure 6-1 Start Up Screen

6.1.4 Login Screen

This is the screen where users authenticate themselves. It features two primary input fields for the user's email and password. The email field is optimized for email entry, ensuring ease of use, while the password field keeps the input hidden, safeguarding the user's sensitive information.

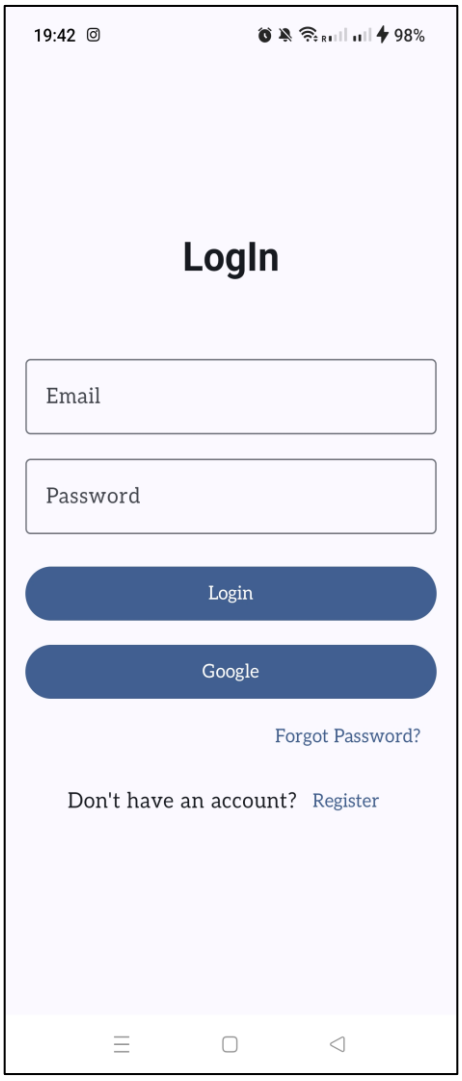


Figure 6-2 Login Screen

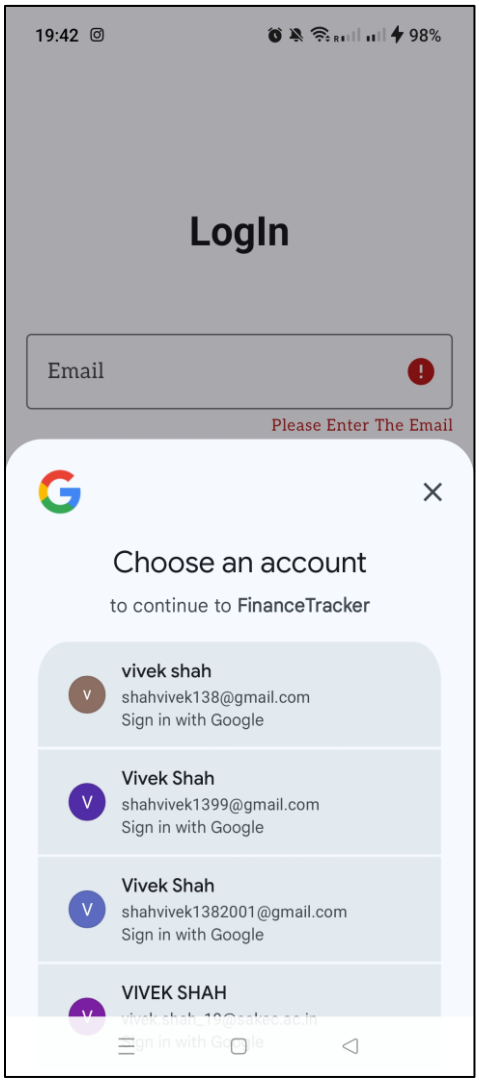


Figure 6-3 Google Sign In

Also, there is a Sign In with Google button where the user can sign in with google Account. Also, the user can use the saved credential which he saved at the time of registration for Login. There is another TextButton called Forgot Password, which lets the user update the password if he forgets. Although, user is saving credential at the time of registration, forgot password features add an extra help in recovering the user account. After filling the login details the details is properly authenticated and user is navigated to Home Screen of the App.

6.1.5 Registration Screen

If the user is new and want to make new account for this application, user can register her by filling the three field Email, Password and Confirm Password. All field are validated and shows the proper error message. Also, it has a register button whenever clicked after filling details, the new account is created and user is navigated to login screen**.**

More Figures of Login/Register Screens are further mentioned in Appendices 12.3.112.3.2

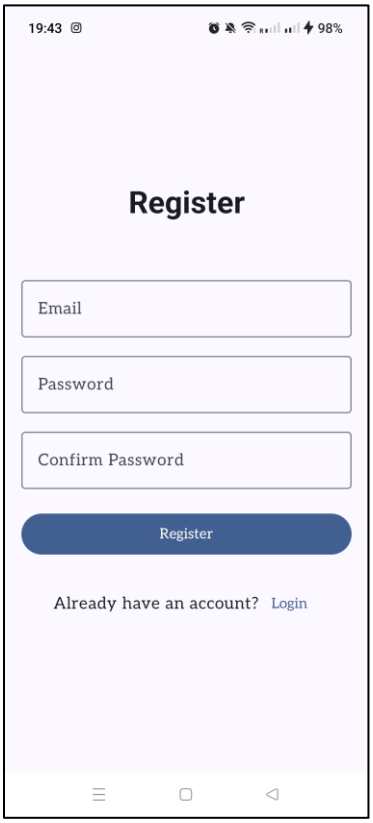


Figure 6-4 Register Screen

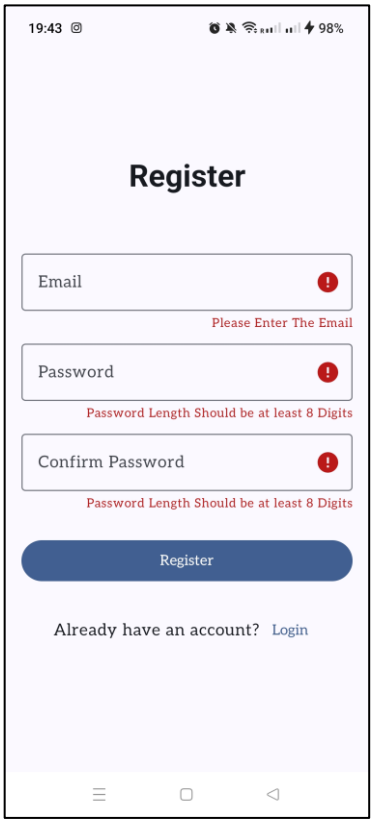


Figure 6-5 Register Screen Validation

6.1.6 Profile Setup Screens

If the user is new, a flag is stored to know that user profile is updated or not. If the user profile is found in firebase is not null then it is retrieved from firebase and state for user profile is updated.

```
val userProfile = coreUseCasesWrapper.getUserProfileUseCase(userId)
val userName = userProfile?.firstName + " " + userProfile?.lastName
Log.d( tag: "LoginViewModel", msg: "userProfile: ${userProfile}")
Log.d( tag: "LoginViewModel", msg: "userName: ${userName}")

coreUseCasesWrapper.setUserNameLocally(userName)
Log.d( tag: "LoginViewModel", msg: "userName: ${userName}")


if(userProfile == null){
    val email = coreUseCasesWrapper.getUserEmailUserCase() ?: "Unknown"
    val newUserProfile = UserProfile(email = email, profileSetUpCompleted = false)
    coreUseCasesWrapper.saveUserProfileUseCase(userId, newUserProfile)
    _loginState.value = loginState.value.copy(
        userProfile = newUserProfile
    )
}
else{
    _loginState.value = loginState.value.copy(
        userProfile = userProfile
    )
    setupAccountUseCasesWrapper.keepUserLoggedIn(keepLoggedIn = true)
}
```

Figure 6-6 Profile Setup Check

Then it is checked that flag of user profile setup is true or false and if flag is true it is navigated to Home Screen and if flag is false then it is navigated to Profile Setup Screen.

```
if(!state.userProfile.profileSetUpCompleted){
    navController.navigate(Screens.NewUserProfileOnBoardingScreen.routes)
}
else{
    navController.navigate(Screens.HomePageScreen.routes)
}
```

Figure 6-7 After Login Navigation

And if the user profile is not updated the user lands on this profile setup screens. First Screen is about Username, then comes the Region/Country where user selects the region using dropdown, the user enters the Phone Number and the user selects the Base Currency which is used for saving all transactions. And when the user saves these details, user is landed to Home Screen and user profile is updated**.**

More Figures for setup accounts are further mentioned in Appendices 12.3.2

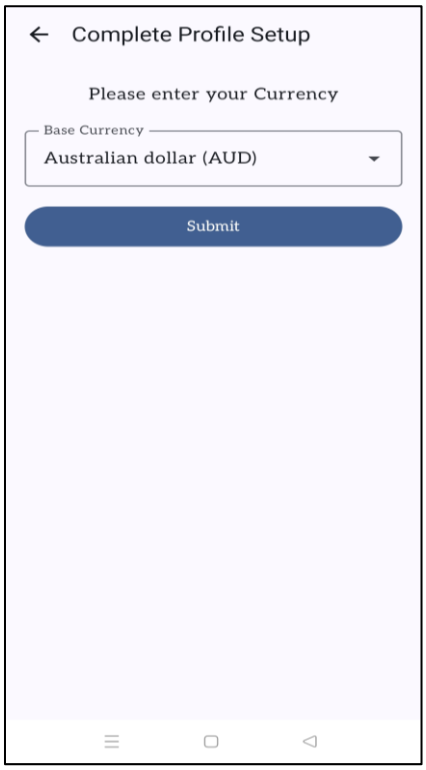


Figure 6-8 Profile Setup Screen

6.1.7 Home Screen

Home Screen gives the general summary of the user account. Here user have the summary of Expense, Income of the current month along with the total account balance. User is able to see the overall as well as category wise progress bar which defines which category is covering how much amount of budget and how much is remaining. If the overall user expenses are heading towards budget amount, once the receive alert amount is reached the user receives the notification about it.

```
IconButton(onClick = {
    val route = if (screen.routes == Screens.ViewRecordsScreen.routes) {
        "${screen.routes}/0" // Add tabIndex = 0 as default
    } else {
        screen.routes
    }

    if (navController.currentDestination?.route != screen.routes) {
        navController.navigate(route)
    }
```

Figure 6-9 Navigation Bar

Home Screen also contains the bottom navigation bar where user can navigate to View Records Screen, Add Transactions Screen, Charts Screen and Settings Screen**.** Also, Home Screen have a menu button where user can navigate to Budget Screen for setting budget for month and user can

also logout of current account. Additionally, whenever user visualize the category wise progress bar, when he wants the details of such all transactions, Details tab will land user to View Records Screen.

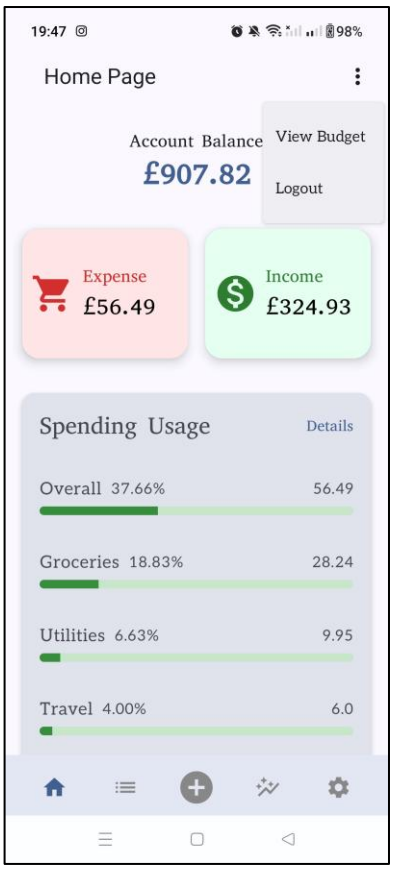


Figure 6-10 Home Screen Menu

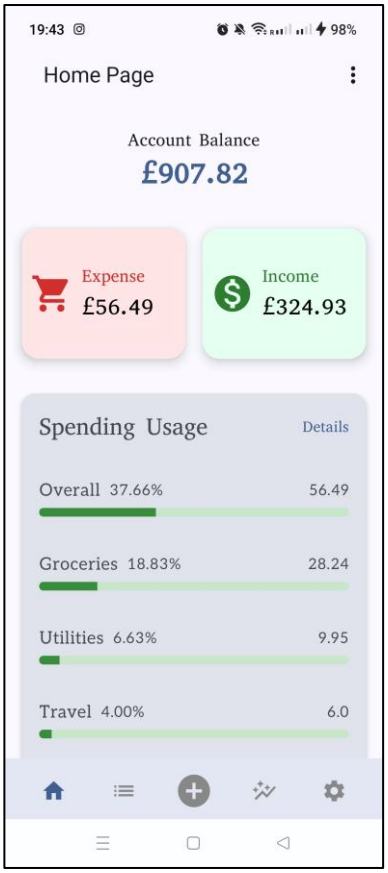


Figure 6-11 Home Screen

6.1.8 Add Transaction Screen

User can add the transactions details manually or he can use the previously saved items**.** If he toggles the Saved Item Switch, user is able to search the particular needed item and set the quantity for that item and after selecting the quantity the other details like category, type, description, currency and price will be added automatically and then user can add the transaction.

```
Box(
    modifier = Modifier.fillMaxSize()
){
    if(states.searchBarFocusedState){

        WithSearchableMode(
            viewModel = viewModel,
            states = states,
            focusRequester = focusRequester,
            selectedItem = selectedItem
        )
    }
    else{
        WithoutSearchableMode(
            scrollState = scrollState,
            viewModel = viewModel,
            states = states,
            focusRequester = focusRequester
        )
    }
}
```

Figure 6-12 Data Entry Selection Mode

Here if the user switches the saved item toggle, then the state for searchBarFocusedState is set to true and then user can add transaction using searchable mode where he can select item and autofill its remaining fields. More Figures for Add Transactions are further mentioned in Appendices 12.3.312.3.2

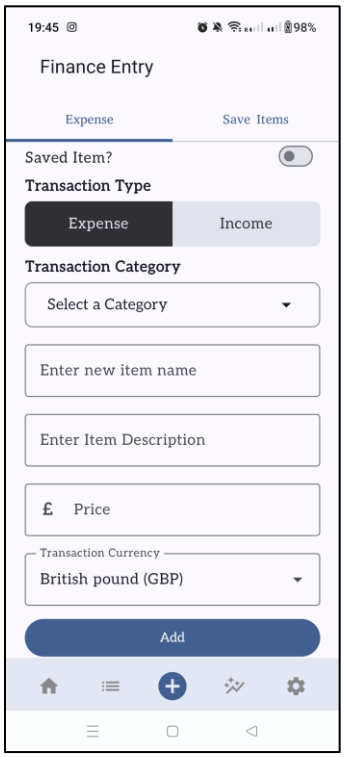


Figure 6-13 Add Transaction Manual

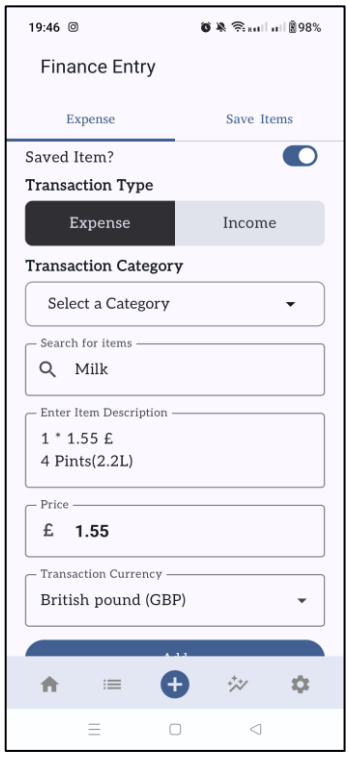


Figure 6-14 Add Transaction Autofill

User can also add the transaction manually by adding all the fields like type, category, description, price, and currency. If the transaction currency is different from base currency, then there is button to convert the amount and the exchange rate is shown and price is converted to base currency and then user can add the transaction**.**

6.1.9 Add Saved Item Screen

User can save the items where user can enter item name, price, description, shop name and currency. Here again the currency can be selected using dropdown where the default currency is set to base currency selected by the user at the time of profile setup. Here required fields are price and name and user cannot save the item keeping them blank**.** These saved items are used to add the transactions automatically**.** More Figures for Add Transactions are further mentioned in Appendices 12.3.412.3.2

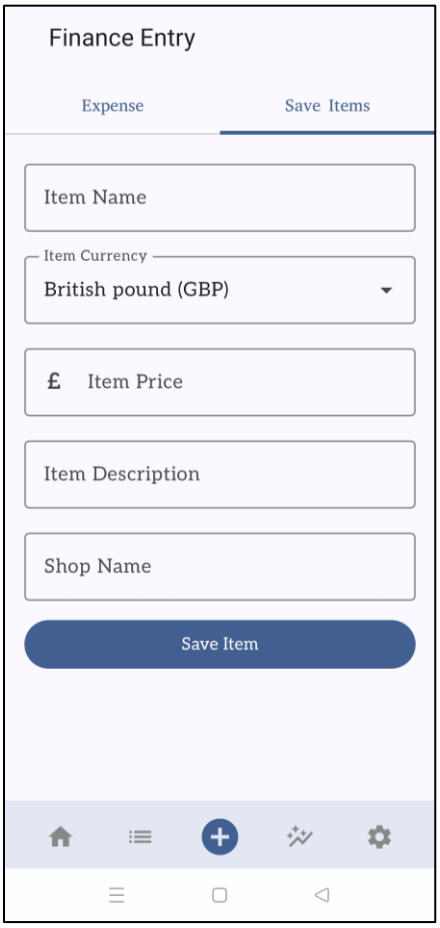


Figure 6-15 Add Saved Item Screen

6.1.10 View Transactions Screen

View Transactions Screen is responsible for displaying the list of transactions added earlier. User can perform actions like select multiple transactions, delete single or multiple transactions and view the info of single transaction.

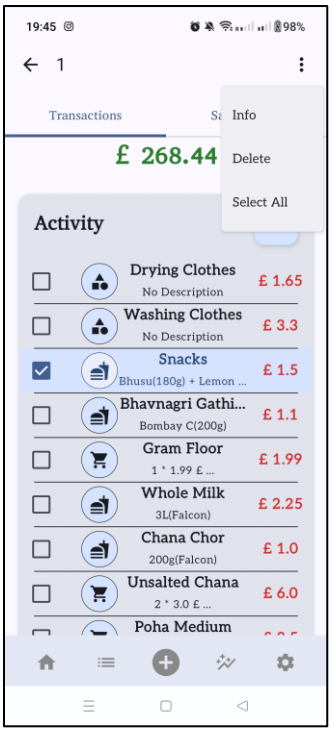


Figure 6-16 Selected Transactions Record

Figure 6-17 Transactions Record

Clicking in a single transaction user can view the details of that transaction. Where single transaction can be deleted. Also user can select option from menu item like if single transaction is selected then menu item options will be “Info”, “Delete” and “Selected All”. Where if multiple transactions are selected the options are “Delete All” and “Selected All”.

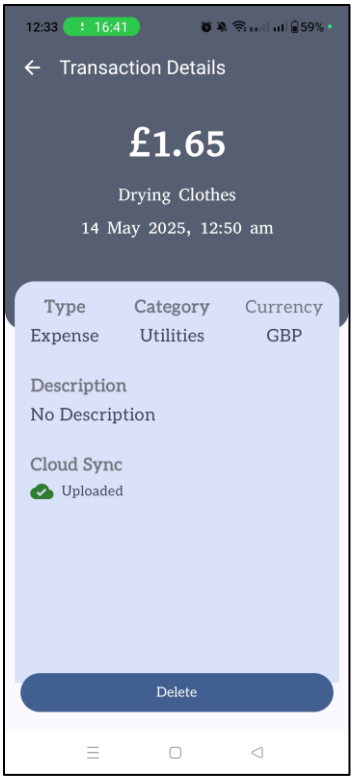


Figure 6-18 Single Transaction Record

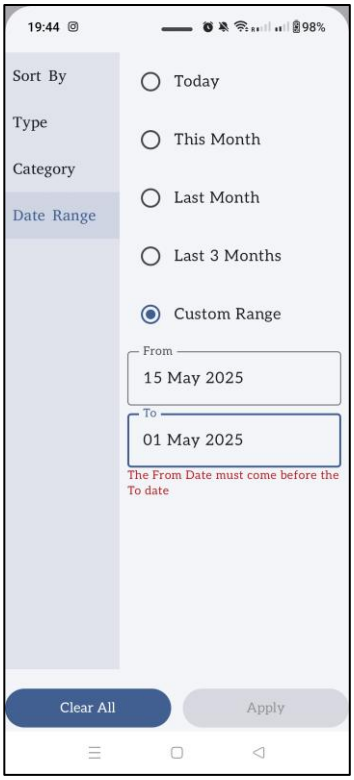


Figure 6-19 Transaction Filter Date Range

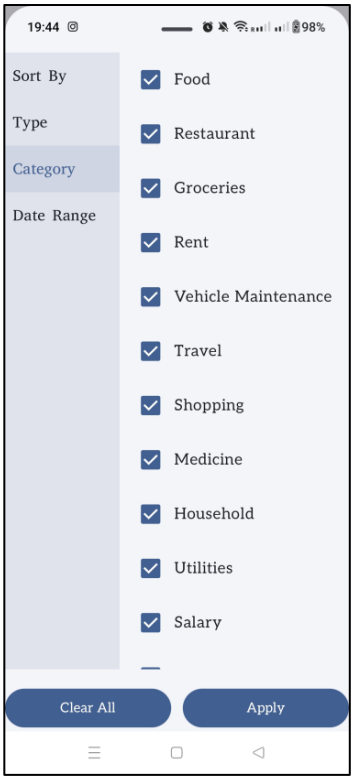


Figure 6-20 Transaction Filter Category

Also, user can view the transaction based on filters. Once click on filters a bottom sheet appears where user selects the order of transactions “Ascending”, “Descending”. In the bottom sheet state, user can select multiple filters like sort by where transactions could be in ascending or descending order, by default descending order is selected. Also, transaction type can be filtered like Expense Income or Both, by default Both types are selected. Transactions could also be filtered based on categories, by default all categories are selected. The duration for all transactions which include “Today”,” This Month”,” Last Month”, “Last 3 Months” and “Custom Date” where user decides the To and From Date using date picker. Date Range filter could also be applied on transactions where user can enter either custom date or view transactions from last 3 months, last month, current month or the present day, by default current month is selected. Additionally, this field has feature validation that To date should not be lower than From date.

More Figures for transaction records are further mentioned in Appendices 12.3.512.3.2

6.1.11 View Saved Items Screen

This screen displays list of all the saved items**.** User can search the particular item to either view the item details or delete a particular item**.** This Screen has a menu item where user can select a particular item to view its details or delete it. Also, just by clicking the single item user can view the details of that item where he is also able to delete or edit that item. User can also select multiple items and delete multiple items. More Figures for saved items records are further mentioned in Appendices 12.3.6

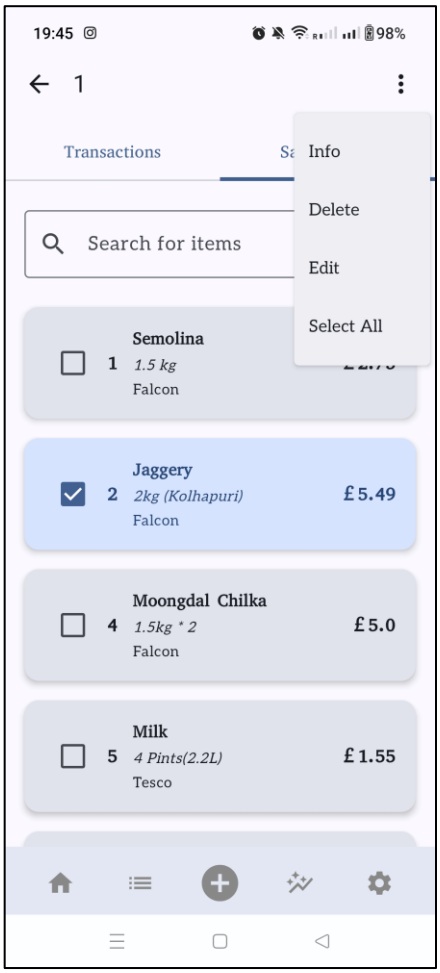


Figure 6-21 Saved Items Menu

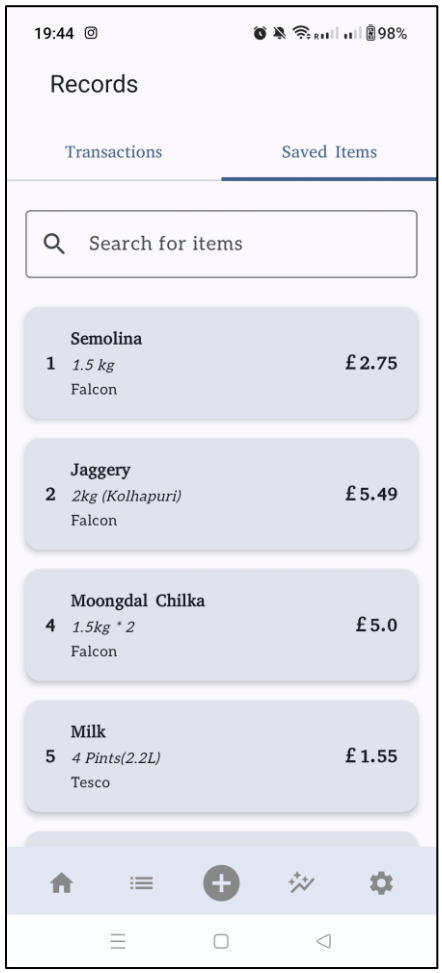


Figure 6-22 Saved Items Menu

12.3.5

6.1.12 Charts Screen

This screen provides a visualization of user transaction data. Users can select either 'Expense' or 'Income' type and view both monthly and yearly visualizations based on categories. Pie charts offer a clear representation of how much each category contributes as a percentage of the total. Here user can select the month via date picker or year picker by clicking the month/year. Also, the user can switch to preview and next month. If you are at current latest month/ year, there is validation that user cannot move to next coming month/year**.** More Figures for charts screen are further mentioned in Appendices 12.3.7

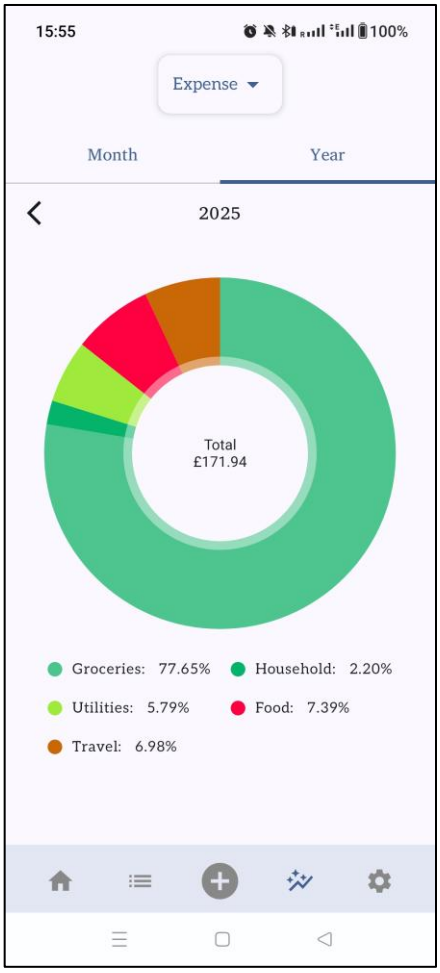


Figure 6-23 Yearly Chart

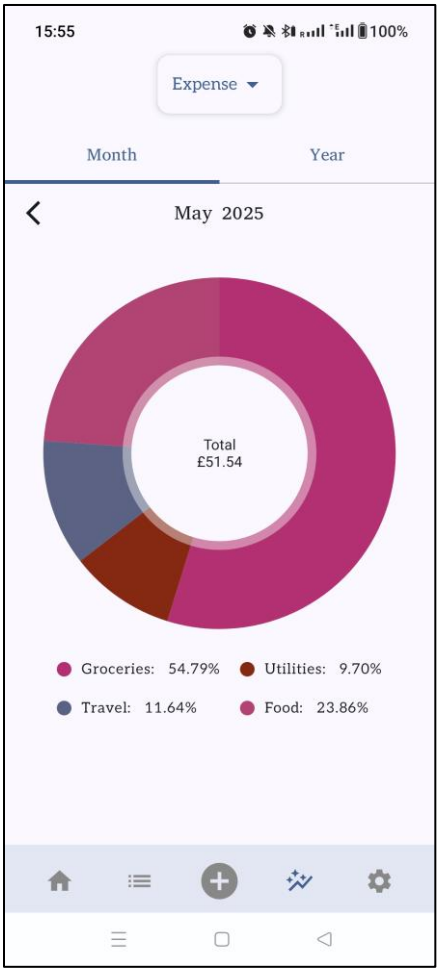


Figure 6-24 Monthly Chart

6.1.13 Settings Screen

Setting Screen starts by displaying the username of user. Below Setting Screen contains a Profile Component which displays the user details**,** Cloud Sync Toggle to sync all the saved items, transactions and budget to cloud, Dark Theme Toggle to turn on the dark theme of the application, Category Component which display and edit all the categories expense and income**,** Budget Screen which is used for setting the budget for certain month and a Logout button where user can logout from the current account.

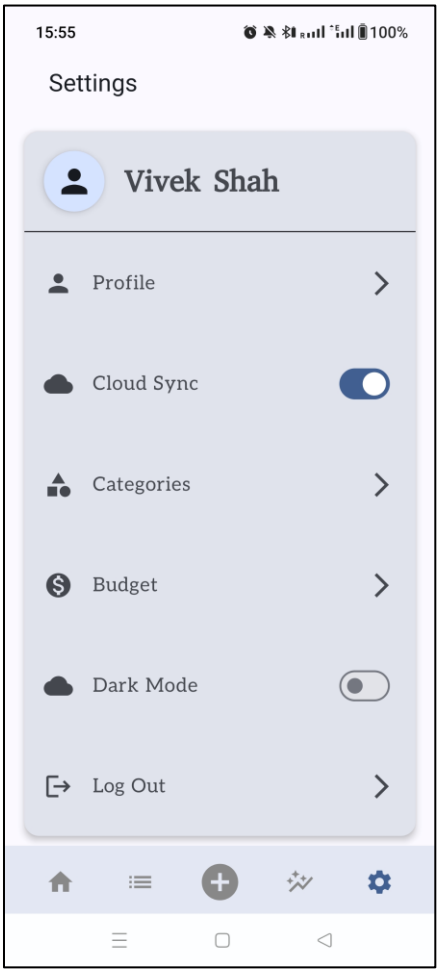


Figure 6-25 Settings Screen

6.1.14 Categories Screen

The categories Screen has expense and income categories listed in this screen. It has both pre-defined and custom categories. Here user can edit and delete the custom categories for both expense and income type**.** However, user cannot edit or deleted the predefined categories which are injected when the app is installed for the first time. More Figures for categories screen are further mentioned in Appendices 12.3.9

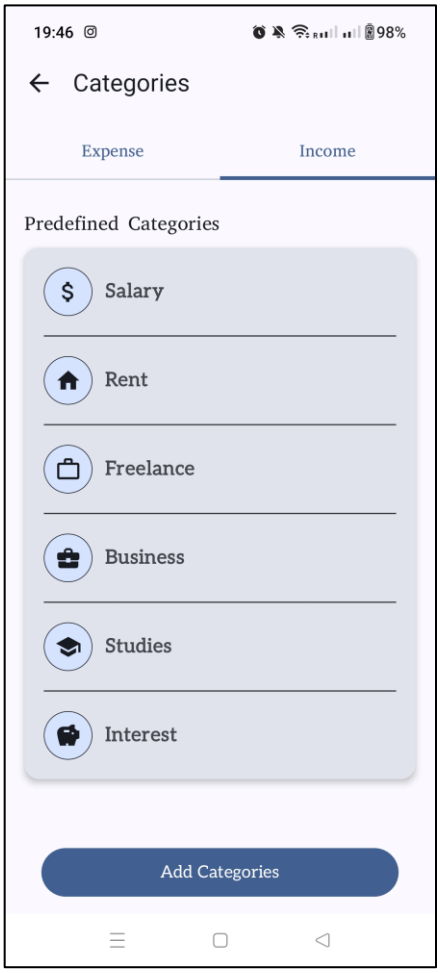


Figure 6-26 Income Categories

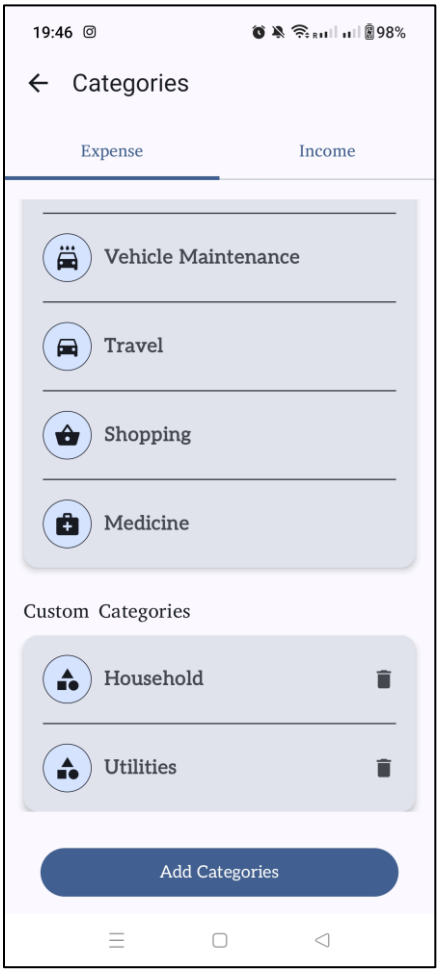


Figure 6-27 Expense Categories

6.1.15 Budget Screen

Budget Screen plays very vital role in user savings for month**.** User can set the budget amount for the month, also decide whether he should be receiving the alerts of not for that month and if yes at which percentage he should be receiving alerts via notifications.

```
// Update visibility logic after NextMonthClicked
_budgetStates.value = _budgetStates.value.copy(
    nextMonthVisibility = (current.get(Calendar.YEAR) < Calendar.getInstance().get(Calendar.YEAR)) ||
            (current.get(Calendar.YEAR) == Calendar.getInstance().get(Calendar.YEAR) &&
                    current.get(Calendar.MONTH) < Calendar.getInstance().get(Calendar.MONTH))
)
```

Figure 6-28 Next Month Click Validation

Here user can select the month via date picker or year picker by clicking the month/year. Also, the user can switch to preview and next month. If you are at current latest month/ year, there is validation that user cannot move to next coming month/year. More Figures for budget screen are further mentioned in Appendices 12.3.8

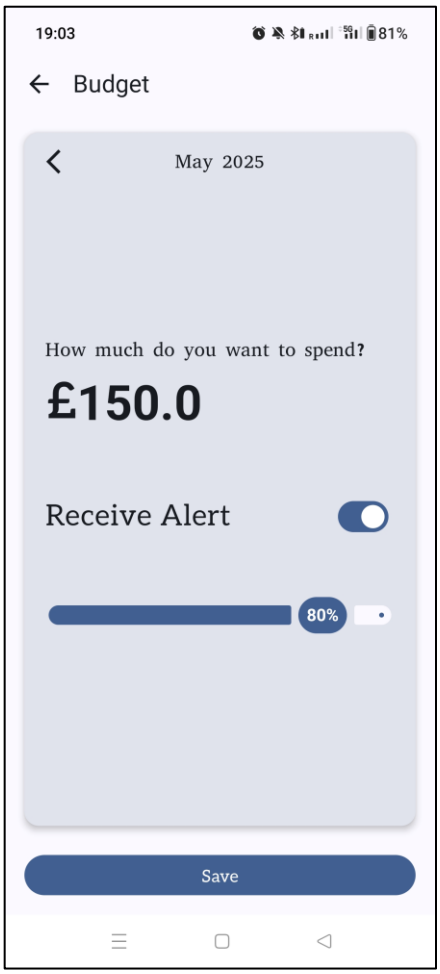


Figure 6-29 View Budget

Figure 6-30 Create Budget

6.1.16 Profile Screen

Whenever user navigates to this screen, user is able to see his user account details like email, first name, second name, phone number, region/country and base currency. User is able to edit the first name, last name, phone number, region/country and base currency based on region. If new region is selected, the base currency is changed for that particular region, later user can set different base currency.

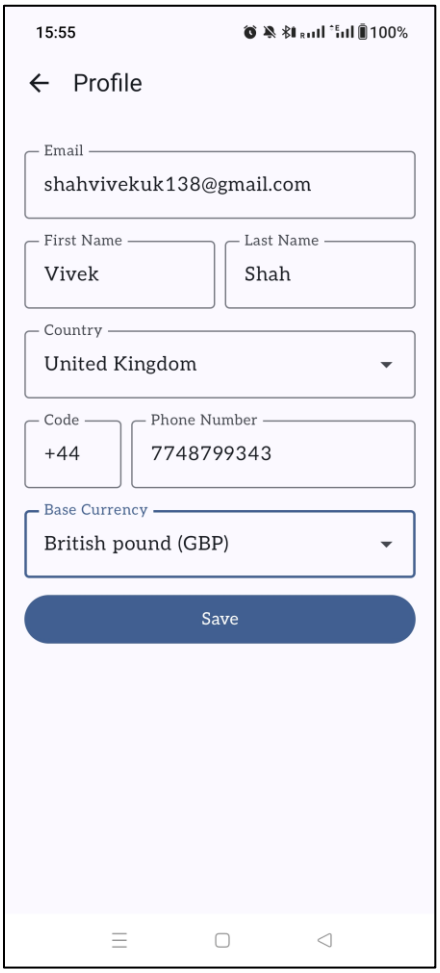


Figure 6-31 User Profile Screen

6.2 Back End Development

6.2.1 Architecture

In the development of the mobile finance tracker application, the MVVM (Model-View-ViewModel) pattern was implemented in combination with Clean Architecture to ensure a scalable, maintainable, and testable codebase. This architectural approach was selected due to its clear separation of concerns, which simplifies the development process and supports modular design, allowing individual layers to evolve independently.

The application is structured into three primary layers: Presentation, Domain, and Data. The Presentation Layer is responsible for the user interface and interaction logic which is already described in the front-end part. The Domain Layer contains the core business logic and is structured around use cases, which encapsulate specific operations such as adding transactions, validating inputs, or retrieving user preferences. This layer acts as a mediator between the UI and the data, providing a clean interface for interacting with the underlying logic. When you want to change the data source for your project you can make it happen without disturbing your whole project flow, this is possible due to this layer. The Data Layer handles all data-related operations and consists of repositories that interact with both Firebase for cloud data storage and Room for local database management. The use of abstraction through interfaces ensures loose coupling and flexibility in switching or combining data sources. All Room Entities, Room Daos, Room Database Scheme and Migration Code is present in this layer.

Hilt, a dependency injection library, is integrated to manage and provide dependencies across layers efficiently, reducing boilerplate code and enhancing testability. This overall architecture enhances the clarity, reliability, and adaptability of the application, making it easier to maintain and extend in future development cycles.

6.2.2 Development Process

Development began with creating database files written in Kotlin, which defined each component's expected behavior. Every Entity contains fields with their proper datatype in which the data is saved locally. After defining the Entities their respective DAO (Direct Access Objects) is defined where the functions to retrieve the data and save the data is defined. After this room database instance is created which inherits the properties from the Super Class name Room Database where the version of that database, database name and tables are defined.

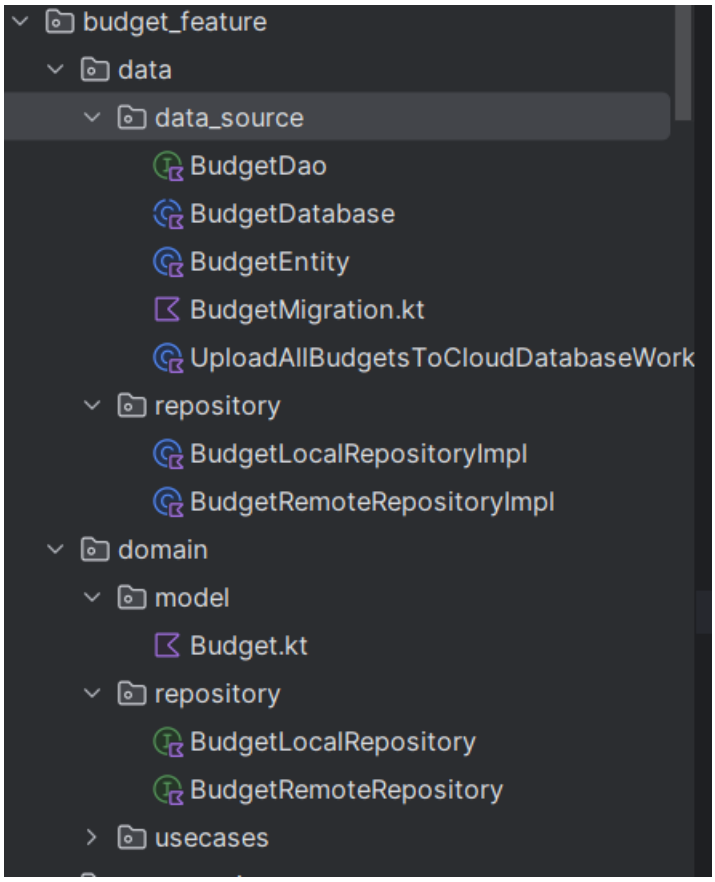


Figure 6-32 Backend Structure

6.2.3 Local Database Design

The local database in the finance tracker application is implemented using Room, Android's persistence library, to provide an offline-first experience and ensure data persistence even when the device is not connected to the internet. The Room database consists of multiple entities, each representing a table that stores structured data relevant to the application's features. More Figures for database setups are further mentioned in Appendices 12.4.1

1. **Transactions Table**

- **Purpose**: Stores all income and expense transactions added by the user.
- **Fields**:
    a. transactionId (Primary Key): Unique ID for the transaction.
    b. transactionName: Name of the transaction
    c. amount: Amount of the transaction.
    d. currency: Currency used.
    e. convertedAmount: Amount converted to the base currency.
    f. exchangeRate: Exchange rate used for conversion.
    g. transactionType: Income or Expense.
    h. category: Name of Category
    i. dateTime: Timestamp of the transaction.
    j. description: Optional notes or details.

k. userUid: User logged In ID.
l. cloudSync: Boolean flag for cloud uploaded transactions.

2. Deleted Transaction Table

- **Purpose:** Stores soft-deleted transactions for deleting it from cloud.
- **Fields:**
    a. transactionId (Primary Key): Unique ID for the transaction.
    b. userUid: User logged In ID.

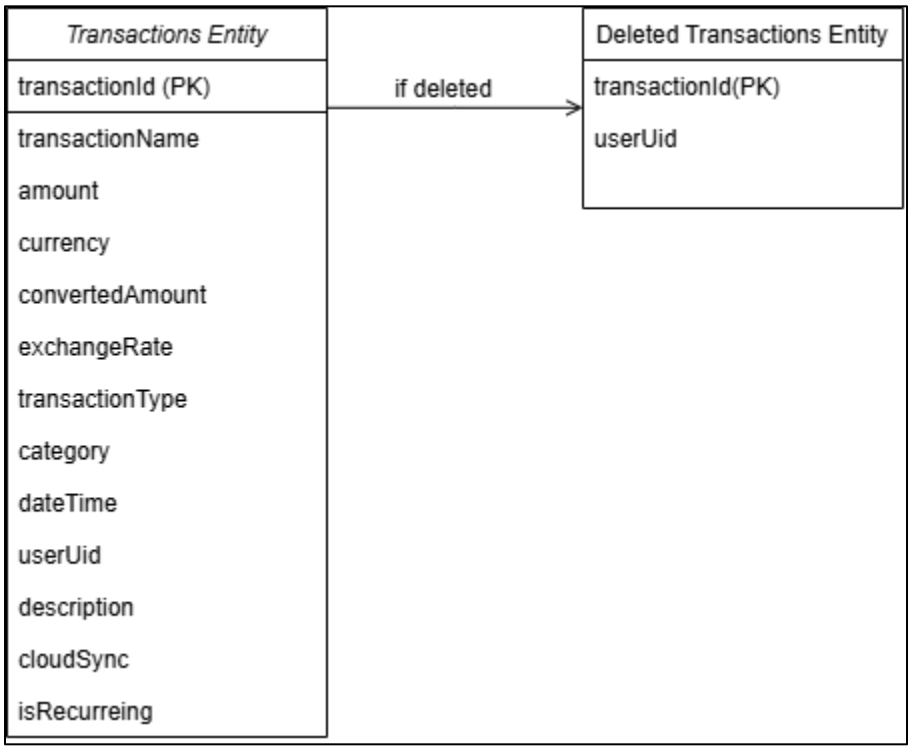


Figure 6-33 Transaction Entity

3. Saved Item Table

- **Purpose**: Stores all saved items added by the user.
- **Fields:**
    a. itemId (Primary Key): Unique ID for the saved item.
    b. itemName: Name of the item
    c. itemPrice: Amount of the item.
    d. itemCurrency: Currency used.
    e. itemShopName: Income or Expense.
    f. itemDescription: Optional notes or details.
    g. userUid: User logged In ID.
    h. cloudSync: Boolean flag for cloud uploaded saved items.

4. **Deleted Saved Item Table**

- **Purpose:** Stores soft-deleted saved items for deleting it from cloud.
- **Fields:**
    a. itemId (Primary Key): Unique ID for the item.
    b. userUid: User logged In ID.

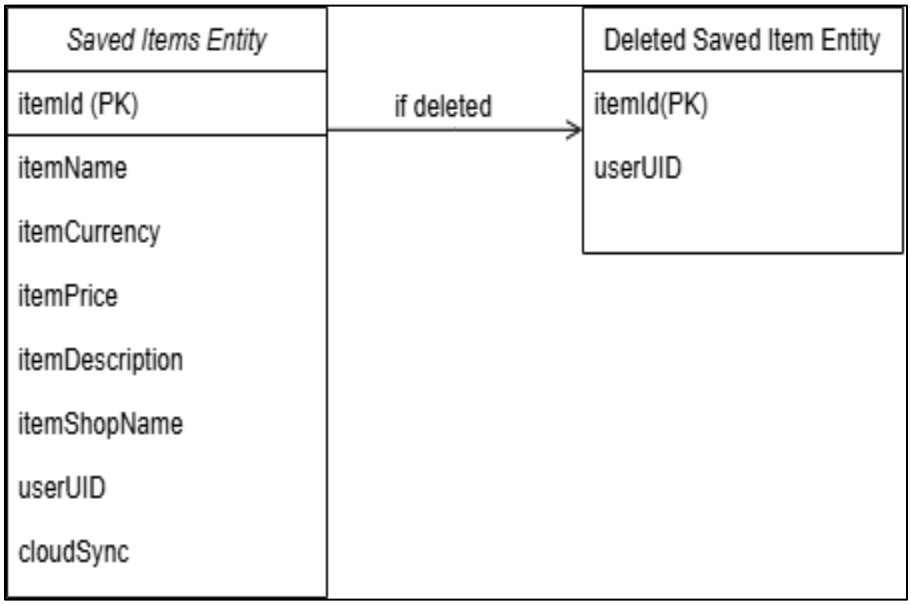


Figure 6-34 Saved Item Entity

5. **Budget Table**

- **Purpose:** Stores monthly budget allocations.
- **Fields:**
    a. id (Primary Key): Unique ID for the budget.
    b. month: month for setting budget.
    c. year: year for setting budget.
    d. amount: Amount of the monthly budget.
    e. receiveAlerts: Boolean flag for sending alerts.
    f. thresholdAmount: Optional notes or details.
    g. userId: User logged In ID.
    h. cloudSync: Boolean flag for cloud uploaded budget.

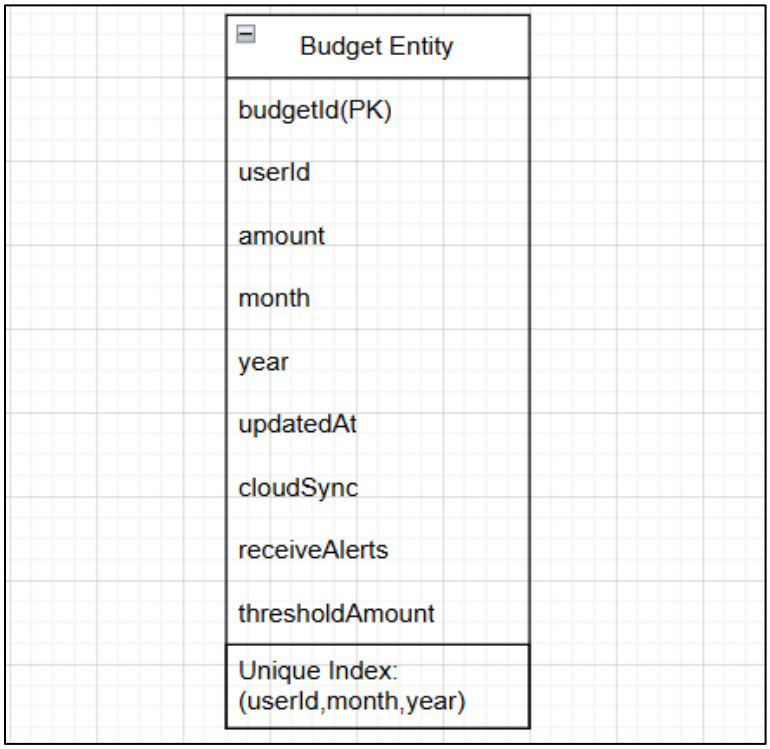


Figure 6-35 Budget Entity

Here the Index of user Id, month, year is set to true, it means that when any of them is different new item in the table is entered. If all are same, it is just updated.

6. **Categories Table**

- **Purpose:** Stores user-defined or predefined categories for transactions.
- **Fields:**
    a. categoryId (Primary Key): Unique ID for the Category.
    b. name: name of the category
    c. type: type for the category.
    d. cloudSync: Boolean flag for identifying category is predefined or custom.
    e. uid: User logged In ID.

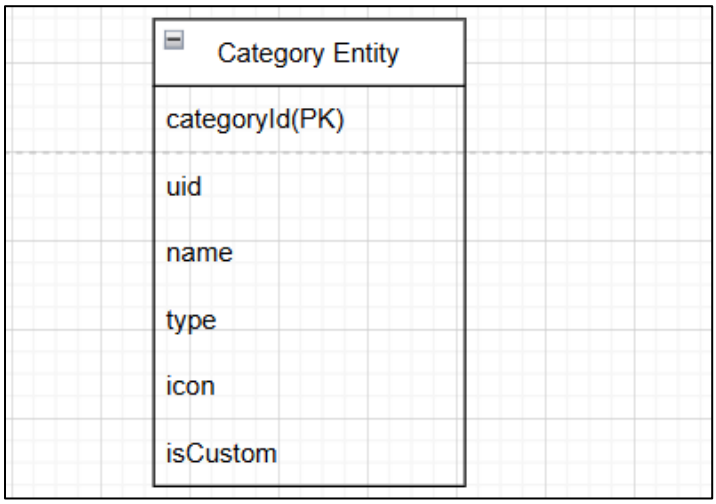


Figure 6-36 Category Entity

7. **User Profile Table**

- **Purpose:** Stores personal and account-related information about the user.
- **Fields:**
    a. uid: User logged In ID.
    b. firstName: First Name of the user entered.
    c. lastName: Last Name of the user entered.
    d. email: Email of the user logged In.
    e. baseCurrency: Base currency selected by user.
    f. country: country/region selected by user.
    g. callingCode: calling code according to the region selected.
    h. phoneNumber: phone number filled by user
    i. profileSetUpCompleted: Boolean flag for identifying has user profile setup completed or not.

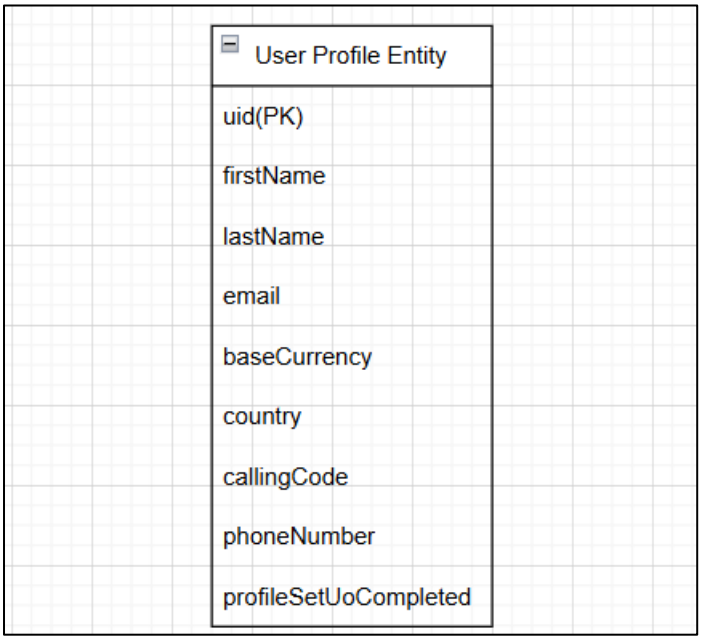


Figure 6-37 User Profile Entity

6.2.4 Local Core Features

The finance tracker application possesses several essential local features to deliver offline functionality seamlessly and improve the overall user experience. These features make extensive use of Room Database for storing user and application data locally so that the application can be accessed even without an active internet connection.

6.2.4.1 Local Storage of Country Details:

When a user selects their country within the application and when the user launches the application, it is checked that is country data already inserted into local database or not and if not then an API call is initiated to fetch country-specific detailed information. More Figures for local storage of country details are further mentioned in Appendices 12.4.2.1

```
val existingCount = countryDao.getCountryCount()
if (existingCount > 0) {
    Log.d( tag: "WorkManagerCountries",  msg: "Countries already inserted: $existingCount")
    return Result.success()
}

return try {
    Log.d( tag: "WorkManagerCountries",  msg: "Fetching countries from API...")

    val countries = try {
        api.getCountries()
    } catch (e: Exception) {
        Log.e( tag: "WorkManagerCountries",  msg: "Error fetching countries from API", e)

        return when (e) {
            is SocketTimeoutException,
            is ProtocolException,
            is HttpException,
            is IOException -> {
                Log.d( tag: "WorkManagerCountries",  msg: "Recoverable error: ${e::class.java.simpleName}, retry:
                Result.retry()
            }
            else -> {
                Log.d( tag: "WorkManagerCountries",  msg: "Unrecoverable: ${e::class.java.simpleName}")
                Result.failure()
            }
        }
    }
```

Figure 6-38 Insert Country WorkManager

As soon as this data is received, it's cached in the local Room database immediately. This habit saves the app from continuously downloading the same data from the network, which makes things more reliable and faster, especially when the user has no connection or a slow connection. By caching country data locally, the application offers a more responsive and faster user experience with reduced dependence on continuous internet connectivity.

```
val countryEntities = countries.map { country ->
    try {
        val entity = country.toEntity()
        Log.d( tag: "WorkManagerCountries",  msg: "Mapped to Entity: $entity")
        entity
    } catch (e: Exception) {
        Log.e( tag: "WorkManagerCountries",  msg: "Error mapping country: $country", e)
        throw e
    }
}

Log.d( tag: "WorkManagerCountries",  msg: "Inserting countries into Room...")
countryDao.insertAll(countryEntities)

val insertedData = countryDao.getAllCountries()
Log.d( tag: "WorkManagerCountries",  msg: "Countries inserted successfully: ${insertedData.size}")
Result.success()
```

Figure 6-39 Insert Country WorkManager

6.2.4.2 Exchange Rate Management:

The application manages currency exchange rates dynamically to facilitate accurate conversions. Upon saving the user's profile with their base currency or changing the base currency, the application invokes the API to fetch the current exchange rates. These rates are cached locally in the database for quick conversions without additional network requests. More Figures for exchange rate management are further mentioned in Appendices 12.4.2.2

```
val baseCurrencyCode = CountryMapper.toCurrencies(baseCurrency)?.keys?.firstOrNull()

if (baseCurrencyCode.isNullOrEmpty()) {
    Log.e(TAG,  msg: "Base currency is null or empty for user $userUID. Cannot proceed.")
    return Result.failure()
}

return try {
    Log.d(TAG,  msg: "Fetching exchange rates for base currency: $baseCurrencyCode")
    val currencyRates = try {
        api.getExchangeRates(baseCurrency = baseCurrencyCode)
    } catch (e: Exception) {
        Log.e(TAG,  msg: "Error fetching exchange rates from API: ${e.message}")

        return if (e is java.net.UnknownHostException || e is java.net.ConnectException) {
            Log.d(TAG,  msg: "No Internet. Retrying in 30 seconds...")
            Result.retry()  // Retry after backoff delay (30s)
        } else {
            Result.failure()
        }
    }

    val currencyRatesEntity = CurrencyRatesMapper.fromCurrencyResponseToEntity(currencyRates)
    currencyRatesDao.insertCurrencyRates(currencyRatesEntity)
    Log.d(TAG,  msg: "Currency rates inserted successfully.")
    userPreferences.setCurrencyRatesUpdated(true)
    Log.d(TAG,  msg: "Set Currency rates updated to TRUE successfully.")

    Result.success()
} catch (e: Exception) {
```

Figure 6-40 Insert Exchange Rates WorkManager

To ensure these rates are updated, the app uses WorkManager to schedule a daily background task that runs every morning at 6:00 AM. This task updates the local exchange rates database so users will have the latest currency information even if they use the app offline later during the day.

```
override suspend fun insertCurrencyRatesLocalPeriodically() {
    val workRequest = PeriodicWorkRequestBuilder<PrepopulateCurrencyRatesDatabaseWorker>(
        repeatInterval: 24, TimeUnit.HOURS
    )
        .setInitialDelay(calculateInitialDelay(), TimeUnit.MILLISECONDS) // Start at 6:00 AM
        .setConstraints(
            Constraints.Builder()
                .setRequiredNetworkType(NetworkType.CONNECTED) // Only run if internet is available
                .build()
        )
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL, // Ensures exponential retry delay
            backoffDelay: 10, TimeUnit.MINUTES       // Starts retrying after 10 minutes, doubles each
        )
        .addTag( tag: "CurrencyUpdateWorker")
        .build()

    workManager.enqueueUniquePeriodicWork(
        uniqueWorkName: "CurrencyUpdateWorker",
        ExistingPeriodicWorkPolicy.CANCEL_AND_REENQUEUE, // Ensures it re-enqueues correctly if resch
        workRequest
    )

    Log.d( tag: "WorkManagerCurrencyRates",  msg: "Scheduled daily currency update at 6:00 AM")
}
```

Figure 6-41 Periodic Exchange Rate Worker

```
private fun calculateInitialDelay(): Long {
    val now = Calendar.getInstance()
    val targetTime = Calendar.getInstance().apply {
        set(Calendar.HOUR_OF_DAY, 6)
        set(Calendar.MINUTE, 0)
        set(Calendar.SECOND, 0)
    }

    if (now.after(targetTime)) {
        targetTime.add(Calendar.DAY_OF_YEAR,  amount: 1) // Schedule for next day if time has passed
    }

    return targetTime.timeInMillis - now.timeInMillis
}
```

Figure 6-42 Exchage Rate Worker Time Period

6.2.4.3 Predefined and Custom Categories:

The app comes with a set of predefined financial categories, e.g., Food, Transport, and Rent, which are present in a JSON file in the assets folder of the app. More Figures for **Predefined and Custom Categories** are further mentioned in Appendices 12.4.2.3

Figure 6-43 Predefined Categories JSON

When the app is run for the first time, this JSON file is read and the categories are added to the local database with the isCustom flag set to false. Every time the user launches the app it is checked that are predefined categories inserted or not and if not then they are inserted.

```
override suspend fun doWork(): Result {
    if(categoryDao.getCategoryCount() > 0) {
        Log.d( tag: "WorkManager",  msg: "Categories already exist. Skipping prepopulation.")
        return Result.success()
    }

    return try {
        Log.d( tag: "WorkManager",  msg: "Inserting predefined categories...")

        Log.d( tag: "WorkManager", msg: "Reached CategoryImplementation")
        val jsonString = JsonUtils.loadJsonFromAssets(context,  fileName: "categories.json")
        if (jsonString != null) {
            Log.d( tag: "WorkManager", msg: "Reached CategoryImplementation inside if jsonString not null")
            Log.d( tag: "WorkManager", msg: "jsonString: $jsonString")
        }
        jsonString?.let { it ->
            val predefinedCategories = JsonUtils.parseJsonToCategories(jsonString = it,uid = "predefined")
            Log.d( tag: "WorkManager", msg: "predefinedCategories: $predefinedCategories")
            categoryDao.insertCategories(
                categories = predefinedCategories.map {
                    it.toEntity()
                }
            )
        }
        Log.d( tag: "WorkManager",  msg: "Predefined categories inserted successfully.")
        Result.success()
    } catch (e: Exception) {
        Log.e( tag: "WorkManager",  msg: "Error inserting categories: ${e.message}")
        e.printStackTrace()
        Result.failure()
    }
```

Figure 6-44 Predefined Categories Worker

Users can also create their own custom categories, which are saved with isCustom as true. Having this distinction allows the app to know the difference between default and custom categories, making it easier to manage categories while still maintaining a defined schema for syncing and display purposes.

6.2.4.4 Transaction Sync Workflow:

The workflow of transaction handling—be the transactions income or expense postings—has been designed methodically to ensure data consistency, accommodate offline use, and synchronize devices. The robust mechanism allows the users to natively record financial activity without internet connectivity concerns or manual synchronizing.

The moment a user enters a new transaction, the application stores it in the local database right away. Besides the transaction data (amount, currency, category, date, etc.), the application sets the cloudSync flag to false. The flag is especially indicating that the transaction has not been synchronized yet to the cloud. More Figures for inserting new transactions are further mentioned in Appendices 12.4.2.412.4.2.2

```
    private val transactionDao: TransactionDao
): TransactionLocalRepository {
    override suspend fun insertTransaction(transactions: Transactions) {
        transactionDao.insertTransaction(
            transactionsEntity = transactions.toEntity()
        )
    }

    override suspend fun insertTransactionReturningId(transactions: Transactions): Long {
        return transactionDao.insertTransactionReturningId(
            transactionsEntity = transactions.toEntity()
        )
    }
```

Figure 6-45 Inserting Transaction

In case the device has an internet connection and cloud sync is enabled in the app settings, the app proceeds to upload the transaction to Firebase (or the designated cloud storage backend). When the upload is complete, the app modifies the entry of the transaction in the local database by setting the cloudSync flag to true, indicating that the transaction is synced with the cloud.

But if network connectivity is lost or cloud syncing is disabled, the transaction is in the local database with sync status set to false. This ensures that user input is never lost even if the user is offline or chooses to defer syncing.

WorkManager is responsible for defered transactions. WorkManager has been configured to run background periodic sync jobs. WorkManager periodically checks on its own any such transactions in local database where cloudSync is still false. When the internet connection is restored if the sync was enabled by the user, WorkManager syncs all these unsynced transactions automatically into the cloud background.

```
override suspend fun cloudSyncMultipleTransaction() {

    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED) // Ensures work runs only when connected
        .build()

    val workRequest = OneTimeWorkRequestBuilder<UploadAllTransactionsToCloudDatabaseWorker>()
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL,
            backoffDelay = 30, TimeUnit.SECONDS
        )
        .setConstraints(constraints)
        .build()

    WorkManager.getInstance(context).enqueueUniqueWork(
        uniqueWorkName: "upload_local_transactions_to_cloud",
        ExistingWorkPolicy.KEEP,
        workRequest
    )
}
```

Figure 6-46 Cloud Sync Transaction Worker

Once every transaction is successfully uploaded, the corresponding entry in the local database is updated to reflect the current sync status to maintain correct tracking and prevent duplicate uploads.

This smart, background-based synchronisation mechanism is designed to deliver users a complete and uninterrupted experience, even between online and offline states. It also provides constant availability of transaction data across many devices, subject to the requirement that the user is signed in to their account and sync is enabled.

In general terms, this workflow offers a reliable and secure transaction management process by combining local-first storage, cloud-based synchronizing, and background syncing automation as all of them play significant roles in ensuring data consistency and user confidence in this application.

6.2.4.5 Saved Items Synchronization:

The workflow employed to handle saved items entries that are used routinely—has been designed to provide an uninterrupted, seamless user experience on different devices under different network availability. The process is extremely similar to the financial transaction management method in the app, with duplicated logic as well as design.

When something (e.g., a frequently purchased grocery expense or recurring income source) is saved, the thing is saved locally in the device's database immediately. Aside from its data, the thing is also flagged with a sync status flag, which indicates to us whether the thing has been synced with the cloud (Firebase) or not. The flag initially is in a state indicating that the thing has not yet been

synced. More Figures for inserting new saved items are further mentioned in Appendices 12.4.2.512.4.2.2

```
SavedItemsLocalRepository {
    override suspend fun insertSavedItems(savedItems: SavedItems) {
        return savedItemsDao.insertSavedItems(savedItems.toEntity())
    }

    override suspend fun insertNewSavedItemReturnId(savedItems: SavedItems): Long {
        return savedItemsDao.insertSavedItemReturningId(savedItemsEntity = savedItems.toEntity())
    }
```

Figure 6-47 Inserting Saved Item

Such local-first approach enables users to save items and be capable of accessing them at a later time—whether they are offline or have an unreliable internet connection. Users can then continue to work on their saved items without loss of progress, with the assurance that their data would eventually be uploaded once internet is available.

Syncing itself is also cleverly taken care of in the background by the WorkManager API of Android. WorkManager checks the local database periodically for anything stored and yet marked as not synced. Whenever there's available internet connection and the synchronization of data has been turned on by the user (sync enabled within settings), WorkManager initiates a background task to sync such unsynced items into the cloud.

```
override suspend fun cloudSyncMultipleSavedItems() {
    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED) // Ensures work runs only when connected
        .build()

    val workRequest = OneTimeWorkRequestBuilder<UploadAllSavedItemsToCloudDatabaseWorker>()
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL,
            backoffDelay = 30, TimeUnit.SECONDS
        )
        .setConstraints(constraints)
        .build()

    WorkManager.getInstance(context).enqueueUniqueWork(
        uniqueWorkName: "upload_local_saved_items_to_cloud",
        ExistingWorkPolicy.KEEP,
        workRequest
    )
}
```

Figure 6-48 Cloud Sync Save Item Worker

Once the items are successfully uploaded to Firebase, their sync status flag in the local database is set indicating they are synced at the moment. This prevents redundant uploading and shows an accurate image of the sync state.

This background sync operation ensures all the items that are saved eventually get synced to all the devices where the user is logged in with his account. For example, if an item is saved by a user on his phone when he's offline and the app is launched on his tablet later, the item will appear on both the devices after syncing is complete.

By executing a strong and efficient sync strategy, the app ensures users have persistent, consistent access to their most-used items, both online and offline. This makes user experience better, especially in cases where network bandwidth is limited, and serves the mission of the app to deliver a seamless, cross-device financial management experience.

6.2.4.6 Transaction Deletion Handling:

Removing the transactions from the app is given extra care to ensure data integrity and synchronization consistency between the local store and the cloud. Rather than simply deleting a transaction record entirely, the app uses a conservative and traceable deleting algorithm designed to support offline access as well as cloud consistency.

When a transaction is deleted by a user, the app immediately removes the transaction from the master transactions table of the local database to effectuate the intention of the user and provide a fast UI experience. But in order to ensure that this deletion will also be replicated in the cloud, the app duplicates an entry for a deleted transaction into another table called the deletedTransactions table simultaneously. More Figures for deleting the transactions are further mentioned in Appendices 12.4.2.612.4.2.2

```
override suspend fun deleteSelectedTransactionsByIds(transactionId: Int) {
    return transactionDao.deleteSelectedTransactionsByIds(transactionId)
}
```

Figure 6-49 Delete from Transaction Entity

```
override suspend fun insertDeletedTransaction(deletedTransactions: DeletedTransactions) {
    return deletedTransactionDao.insertDeletedTransaction(deletedTransactions.toEntity())
}
```

Figure 6-50 Insert into Deleted Transaction Entity

This deletedTransactions table is a deletion log that is in-memory, saving important metadata such as the transaction ID, user ID, and deletion timestamp. This enables the app to recall transactions that need to be deleted from cloud storage—even if the device is offline at the time of deletion.

When app launch or background sync periods are established, the WorkManager feature checks whether the device is online and cloud sync is enabled. If so, it processes the records in the

deletedTransactions table by posting deletion requests to the cloud database (e.g., Firebase). Every transaction item in the table is located in the cloud using its ID and deleted from the user's cloud data set.

```
override suspend fun deleteMultipleTransactionsFromCloud() {
    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED) // Ensures work runs only when connected
        .build()

    val workRequest = OneTimeWorkRequestBuilder<DeletedAllTransactionsFromCloudDatabaseWorker>()
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL,
            backoffDelay = 30, TimeUnit.SECONDS
        )
        .setConstraints(constraints)
        .build()

    WorkManager.getInstance(context).enqueueUniqueWork(
        uniqueWorkName: "delete_transactions_from_cloud",
        ExistingWorkPolicy.KEEP,
        workRequest
    )
}
```

Figure 6-51 Cloud Sync Transaction Deletion Worker

Once a transaction is successfully deleted from the cloud, the app removes the corresponding record from the local deletedTransactions table to finish the deletion process successfully. This also eliminates duplicate or repeated attempts for deletion, keeping the syncing process clean and efficient.

6.2.4.7 Saved Item Deletion Sync:

The deletion of saved item—is based on a symmetrical and uniform deletion pattern, similar to the case of transactions. Such an architecture ensured that local deletions are correctly replicated to the cloud so that data consistency is maintained across devices and no form of orphaned or stale data occurs.When the user chooses to remove a saved object (e.g., a common food, rent posting, or source of income), the app immediately removes it from the local saved items table so the change can be made in the user interface in an instant. Instead of just throwing away the information, the app adds a record of this removal to a special local deletion queue or deletedSavedItems table. More Figures for deleting saved items are further mentioned in Appendices 12.4.2.712.4.2.2

```
override suspend fun deleteSelectedSavedItemsByIds(savedItemsId: Int) {
    return savedItemsDao.deleteSelectedSavedItemsByIds(savedItemsId)
}
```

Figure 6-52 Delete from Saved Item Entity

```
SavedItemsRemoteRepository {
 override suspend fun insertDeletedSavedItems(deletedSavedItems: DeletedSavedItems) {
     deletedSavedItemsDao.insertDeletedSavedItems(deletedSavedItems.toEntity())
 }
```

Figure 6-53 Insert into Deleted Saved Item Entity

This delete table keeps track of significant information such as the item ID, user ID, and deletion time. These being stored makes the app aware of which stored objects must be deleted from the cloud even if the deletion was performed while offline or at a time when cloud syncing had been disabled.

When the app is launched, or when it runs in normal background sync operations managed by WorkManager, the app checks whether the device has an active internet connection and whether cloud sync is active. If both conditions are met, WorkManager calls the deletedSavedItems table and sends deletion requests to the cloud database (e.g., Firebase) against the saved items listed there.

```
override suspend fun deleteMultipleSavedItemsFromCloud() {
    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED) // Ensures work runs only when connected
        .build()

    val workRequest = OneTimeWorkRequestBuilder<DeletedAllSavedItemsFromCloudDatabaseWorker>()
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL,
            backoffDelay = 30, TimeUnit.SECONDS
        )
        .setConstraints(constraints)
        .build()

    WorkManager.getInstance(context).enqueueUniqueWork(
        uniqueWorkName: "delete_saved_items_from_cloud",
        ExistingWorkPolicy.KEEP,
        workRequest
    )
}
```

Figure 6-54 Cloud Sync Item Deletion Worker

For each item deleted that was removed successfully from the cloud, the app then removes the corresponding entry from the local deletion queue. This cleaning process maintains the deletion record for only as long as necessary and keeps the sync status on local and cloud storage up to date.

6.2.4.8 Budget Management:

The budget management workflow in the app is just as consistent, dependable, and offline-first in nature as transaction and saved item workflow. This ensures users can view, create, and modify budget data unrestricted—online or offline—while keeping data synchronized between devices when cloud sync is enabled.All of a user's monthly budgets is stored in its own independent table in the local Room database. This table has all the fields of interest for budget tracking plus a cloudSync flag to indicate whether the budget has been synced with the Firebase.When a user is entering or updating a budget, the app first saves the budget entry locally with cloudSync = false, indicating that the update has not yet been pushed to the cloud. That way, users can work with their budgets offline without the requirement of an internet connection, including full offline access and functionality. More Figures for inserting and updating budgets are further mentioned in Appendices 12.4.2.8

```
override suspend fun insertBudget(budget: Budget) {
    return budgetDao.insertBudget(budget = budget.toEntity())
}

override suspend fun sendBudgetNotifications(title: String, message: String) {
    val builder = NotificationCompat.Builder(context, channelId: "budget_alert_channel")
        .setSmallIcon(R.drawable.warning) // ✅ use your icon resource
        .setContentTitle(title)
        .setContentText(message)
        .setPriority(NotificationCompat.PRIORITY_HIGH)

    val notificationManager = NotificationManagerCompat.from(context)
    notificationManager.notify( id: 1001, builder.build())
}
```

Figure 6-55 Insert Budget

If the device is connected to the internet and cloud syncing is enabled by the user, the app initiates a sync process, attempting to upload the unsynced budget data into the cloud. After the upload is done, the app updates the corresponding local record and sets the cloudSync flag to true to show that the data is synchronized.If the user is offline or temporarily disabled cloud syncing, the budget change is safely saved in the local database, marked as unsynced. The change is not lost.

```
override suspend fun uploadSingleBudgetToCloud(userId: String, budget: Budget) {

    firestore.collection( collectionPath: "Users")
        .document(userId)
        .collection( collectionPath: "Budgets")
        .document(budgetId)
        .set(budget.copy(cloudSync = true)) // Save with cloudSync = true
        .await()
}

override suspend fun uploadMultipleBudgetsToCloud() {
    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED) // Ensures work runs only when connected
        .build()

    val workRequest = OneTimeWorkRequestBuilder<UploadAllBudgetsToCloudDatabaseWorker>()
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL,
            backoffDelay = 30, TimeUnit.SECONDS
        )
        .setConstraints(constraints)
        .build()

    WorkManager.getInstance(context).enqueueUniqueWork(
        uniqueWorkName: "upload_local_budgets_to_cloud",
        ExistingWorkPolicy.KEEP,
        workRequest
    )
}
```

Figure 6-56 Budget Cloud Sync Worker

In order to handle automatic synchronizing, the application utilizes WorkManager, which is configured to execute periodical background jobs. WorkManager, at periodic intervals, checks the local budget entity for all of the records where cloudSync = false. Once the device is back online and syncing is re-enabled, WorkManager synchronizes all pending budget records to the cloud so that there is latest data on every user device.

6.2.5 Cloud Integration Core Features

The application employs robust cloud synchronization capabilities through Firebase services, primarily Firebase Authentication and Firebase Firestore (or alternatively, Realtime Database). The services enable secure management of user identity and effortless syncing of information between devices. Firebase provides scalability, real-time, and solid cloud infrastructure without the need for manual management of backend servers.

6.2.5.1 Firebase Authentication Google / Register and Login

The app supports both Google Sign-In and email/password sign-up for authentication. There are alternative means of convenient sign in using the Google account to facilitate easy and comfortable use by the user. As an option, they may register using a legitimate email and password. On registering or login, the application utilizes Firebase Authentication for uniquely identifying the user via a UID (User ID) which will hence become the master key to accessing and managing all of their private information in the cloud. More Figures for firebase authentication are further mentioned in Appendices 12.4.3.1

```
suspend fun registerUser(username: String, password: String) : RegisterResult {
    return try{
        firebaseAuth.createUserWithEmailAndPassword(username, password).await()

        try{
            credentialManager.createCredential(
                context = activity,
                request = CreatePasswordRequest(
                    id = username,
                    password = password
                )
            )
            RegisterResult.Success
        }
        catch (e: CreateCredentialCancellationException){
            e.printStackTrace()
            RegisterResult.CredentialCancelled
        }catch (e: CreateCredentialException){
            e.printStackTrace()
            RegisterResult.CredentialFailure
        }
    }catch (e: FirebaseAuthUserCollisionException) {
        e.printStackTrace()
        RegisterResult.FirebaseAuthUserCollisionException
    }catch (e: FirebaseAuthException){
        e.printStackTrace()
        RegisterResult.RegistrationFailure
    }catch (e: Exception){
        e.printStackTrace()
        RegisterResult.UnknownFailure
    }
```

Figure 6-57 Firebase Registration

```
suspend fun loginInUser(username: String, password: String): LogInResult {
                context = activity,
                request = GetCredentialRequest(
                    credentialOptions = listOf(GetPasswordOption())
                )
            )
        } catch (e: Exception) {
            // Handle when the user cancels the credentials dialog
            null
        }

        val credential = credentialResponse?.credential as? PasswordCredential

        val authResult = if (credential != null) {
            // Use saved credentials to sign in
            firebaseAuth.signInWithEmailAndPassword(credential.id, credential.password).await()
        } else {
            // Fallback to manual login with provided username and password
            firebaseAuth.signInWithEmailAndPassword(username, password).await()
        }

        LogInResult.Success( userName: authResult.user?.email ?: "Unknown")
    } catch (e: FirebaseAuthException) {
        e.printStackTrace()
        LogInResult.Failure // Firebase-specific failure
    } catch (e: Exception) {
        e.printStackTrace()
        LogInResult.Failure // General failure (network issues, etc.)
    }
}
```

Figure 6-58 Firebase Login

```
suspend fun signInWithGoogle() : GoogleSignInResult {
    return try{
        val response = credentialManager.getCredential(
            context = activity,
            request = getSignInRequest()
        )

        val googleCredential = GoogleIdTokenCredential.createFrom(response.credential.data)

        val firebaseCredential = GoogleAuthProvider.getCredential(googleCredential.idToken,null)

        val authResult = firebaseAuth.signInWithCredential(firebaseCredential).await()
        Log.d( tag: "AccountManager", msg: "sucesss")
        GoogleSignInResult.Success( username: authResult.user?.email ?: "Unknown")
    }catch (e: GetCredentialCancellationException){
        e.printStackTrace()
        Log.d( tag: "AccountManager", msg: "error ${e.localizedMessage}")
        GoogleSignInResult.Cancelled
    }catch (e: GetCredentialException){
        e.printStackTrace()
        Log.d( tag: "AccountManager", msg: "error ${e.localizedMessage}")
        GoogleSignInResult.Failure
    }catch (e: Exception){
        e.printStackTrace()
        Log.d( tag: "AccountManager", msg: "error ${e.localizedMessage}")
        GoogleSignInResult.Failure
    }
}
```

Figure 6-59 Firebase Login With Google

To facilitate login and registration, the app also utilizes stored credentials, allowing repeat users to log in automatically without entering credentials each time. This improves the overall user experience while ensuring security through Firebase's native authentication management.

6.2.5.2 Firestore Database for sync

Data synchronization is handled effortlessly by WorkManager, which sends all local, unsynced data—transactions, saved items, or budgets—when the device is connected to the internet and cloud sync is on. The cloudSync flag in local storage keeps track of whether or not each record synced successfully. When a transaction or saved item is deleted by the user, the deletion is first logged locally. And then the transactions or saved items which are deleted by user is stored in the another table with same primary key and when the device is connected to the internet those all transactions/saved items are deleted using WorkManager. More Figures for firestore database sync are further mentioned in Appendices 12.4.3.212.4.3.1

```
override suspend fun cloudSyncSingleSavedItem(
    userId: String,
    savedItems: SavedItems,
    updateCloudSync: suspend (Int, Boolean) -> Unit
) {
    try {

        val itemId = savedItems.itemId?.toString() ?: firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "SavedItems")
            .document().id

        firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "SavedItems")
            .document(itemId)
            .set(savedItems.copy(cloudSync = true)) // Save with cloudSync = true
            .await()

        // Update local Room DB
        savedItems.itemId?.let { updateCloudSync(it, true) }

    }catch (e: Exception){

        // If failed, make sure cloudSync remains false
        savedItems.itemId?.let { updateCloudSync(it, false) }
    }
}
```

Figure 6-60 Inserting Saved Item to Firestore

```kotlin
override suspend fun cloudSyncSingleTransaction(
    userId: String,
    transactions: Transactions,
    updateCloudSync: suspend (Int, Boolean) -> Unit
) {
    try {

        val transactionId = transactions.transactionId?.toString() ?: firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "Transactions")
            .document().id

        firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "Transactions")
            .document(transactionId)
            .set(transactions.copy(cloudSync = true)) // Save with cloudSync = true
            .await()

        // Update local Room DB
        transactions.transactionId?.let { updateCloudSync(it, true) }

    }catch (e: Exception){

        // If failed, make sure cloudSync remains false
        transactions.transactionId?.let { updateCloudSync(it, false) }
    }
}
```

Figure 6-61 Inserting Transaction to Firestore

6.2.5.3 Explain structure of cloud data.

After authentication, the user's personal and financial data is held in the cloud in a hierarchical format under his own unique UID. The UID is the top node for all the data of that specific user. The cloud data structure is designed to be modular and scalable so that updating and retrieval is handled efficiently. More Figures for firebase structure are further mentioned in Appendices 12.4.3.312.4.3.1

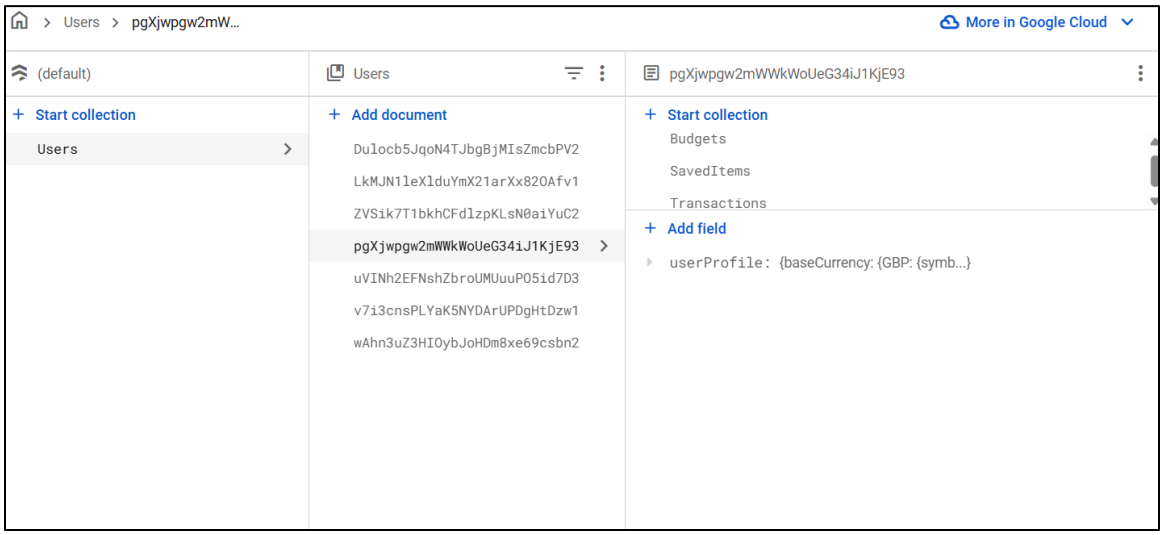


Figure 6-62 Firestore Database Structure

Below is the structure breakdown:

- **UID (Head Node):** Each user's UID is the root node where all the related data is stored.
- **User Profile:** This section stores the user's own data such as first name, last name, email, phone number, selected base currency, country, and profile setup flag. These are fetched upon profile creation and set as and when required.
- **Transactions:** All income and expense transactions are stored here. All records of transactions have fields such as transaction ID, amount, currency, converted amount, exchange rate, type (expense or income), category, date and time, item name, notes, and a flag indicating if it's an automatic repeating transaction. This is a simple manner in which financial data can be queried and filtered in the cloud.
- **Saved Items:** This division stores the often-used items for quick entry while entering transactions. Every stored item has an ID, name, type (income or expense), and associated category. The items are synchronized with the local database as well and can be used offline.
- **Budget:** The budget node holds data for monthly budgets. Similar to transactions, it holds the amount of the budget, corresponding categories, date range, and any other accompanying settings. This helps users track and regulate their financial goals.

6.2.6 API Integration

The app uses third-party APIs for enhancing functionality and user experience, primarily for retrieving country data and exchange rates. It uses the same with Retrofit, an Android HTTP client which is type-safe, to ease sending network requests and handling API responses efficiently through the use of Kotlin coroutines and data classes. More Figures for API integration are further mentioned in Appendices 12.4.4

6.2.6.1 Country Information API Integration:

```
import ...

interface CountryApi {

    @GET("v3.1/all")
    suspend fun getCountries(): List<Country>
}
```

Figure 6-63 Country API Interface

One of the core API integrations fetches in-depth information for various countries, e.g., their name, flag, IDD, and currencies they support. This information is fetched when the user initially sets up his/her profile or in suitable operations that require country-related information.The fetched information from the API is structured into a data model through Kotlin data classes as shown below:

```
data class Country(
    val name: Name,
    val flags: Flags,
    val idd: Idd,
    val currencies: Map<String, Currency>
)

data class Name(val common: String)

data class Flags(val svg: String)

data class Idd(val root: String, val suffixes: List<String>)

data class Currency(val name: String = "", val symbol: String = "")
```

Figure 6-64 Country Data Class

This model cleanly represents the nested structure received from the JSON response. To persist the country details locally, a mapping is done between the API model and the database entity model using a mapper function:

```
fun Country.toEntity(): CountryEntity{
    return CountryMapper.fromCountryResponseToEntity(this)
}

fun CountryEntity.toDomain(): Country{
    return CountryMapper.fromEntityToCountryResponse(this)
}
```

Figure 6-65 Country Model to Entity Mapper

This abstraction layer ensures that the app maintains a clean architecture, where the API layer, domain layer, and data layer remain decoupled. The received country data is stored in a local Room database for offline access and faster loading.

6.2.6.2 Exchange Rate API Integration:

```
interface CurrencyRatesApi {

    @GET("{apiKey}/latest/{baseCurrency}")
    suspend fun getExchangeRates(
        @Path("apiKey") apiKey: String = ApiClient.API_KEY,  // Pass the API key dynamically
        @Path("baseCurrency") baseCurrency: String
    ): CurrencyResponse
}
```

Figure 6-66 Exchange Rate API Interface

Another key API integration involves fetching real-time currency exchange rates, which are essential for supporting multi-currency transactions and budget tracking. When a user saves their profile or chooses a base currency, the app makes an API call to retrieve exchange rates for that base currency.The API returns a simple JSON object with a base_code and a map of conversion_rates. The response is mapped into the following Kotlin data class:

```
data class CurrencyResponse(
    val base_code: String,
    val conversion_rates: Map<String, Double>
)
```

Figure 6-67 Exchange Rates Data Class

This structure offers instant access to conversion rates between the base currency and all other currencies supported. They are stored locally and used in real-time to show converted amounts when users input or view transactions in other currencies.In order to ensure that the app has up-to-date currency data at all times, a WorkManager task is scheduled to fetch the latest exchange rates daily at 6:00 AM. The new rates are saved in the local database so that the app can provide accurate conversions even without a constant internet connection.

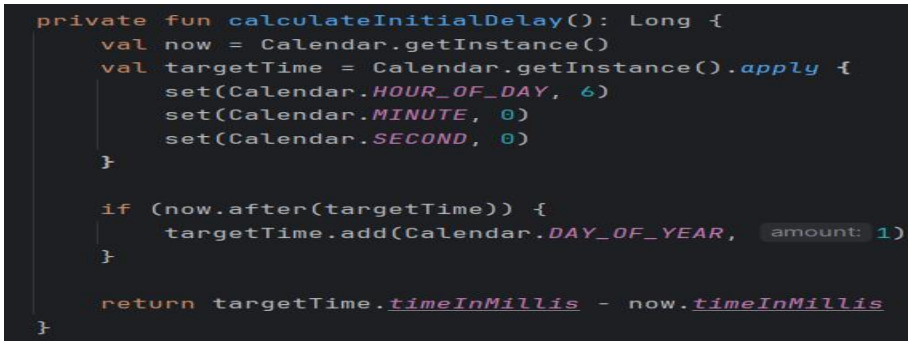


Figure 6-68 API Calling Time

7 Testing and Evaluation

Testing and evaluation are also crucial to the success of the finance tracker app since they ensure that the app works properly, securely, and efficiently. With thorough testing, bugs, errors, or security issues might be found and corrected before it's live. Evaluation guarantees that all the features such as transaction tracking, budgeting, and syncing of data work in perfect harmony on different devices and scenarios. It also ensures the app delivers a smooth user experience, maintains financial calculations precise and maintains data security. Last but not least, appropriate testing and evaluation instill confidence in the users and enhance the app's reliability and overall quality.

7.1 System Testing

7.1.1 Purpose

System testing is an important phase of the software development lifecycle that guarantees the finance tracker application performs as intended in a complete, fully integrated environment. Its only intention is to verify the overall functionality, consistency, and overall performance of the system before it is handed over to customers. System testing detects faults or inconsistencies which would have otherwise gone unnoticed during earlier test phases by testing the application with real-world scenarios.System testing involves various types of testing, but of them, unit testing and integration testing are most important.

Unit Testing focuses on testing individual elements or modules of the application separately. Each function or procedure, such as calculating costs or saving transaction history, is tested to ensure it behaves as described. This facilitates detection of bugs in small pieces of the system early on, thus simpler and faster debugging.Integration Testing verifies the interaction among the different modules after integration. It verifies data is passing between different components correctly, e.g., between user interface and database, or between application and cloud services like Firebase. Integration testing gives confidence that the integrated components work together smoothly, verifying the overall process.

In this finance tracker app project, primary focus was on unit testing to check individual features like adding transaction, adding saved items, validating input fields and many more. Due to time and project constraints, extensive integration testing has not been performed yet. While unit testing provides component reliability, future development will include integration testing and remaining unit testing components to check if all modules interact well together for a seamless user experience.

7.1.2 Unit Testing

Unit testing is an important role of verifying the correctness of every component of the finance tracker application. Unit testing is to test every module individually to ensure it functions as it is supposed to in various conditions. In this project, unit testing was applied to several key components dealing with very critical operations.The first set of functions tested were Email Validation, Password Validation, and Confirm Password Validation. These tests ensure that user input during registration and login is in the proper format and meets security specifications. Validations check for good email syntax, strong password, and matching confirmation passwords to prevent user mistakes and enhance security.

Second, critical financial functionality tests were conducted, including Adding Budget, where it was ensured that users could properly set and update monthly budgets without any problems. The Add Saved Item feature was tested to verify that users could save and remember commonly encountered expense or income items accurately, increasing transaction entry speed and efficiency.

Add Transaction was tested to ensure that the app properly tracks expenses and income, handles currency exchange, and stores transaction information correctly. Finally, unit tests briefly touched on Add User Profile, which is responsible for saving and updating user profile information such as name, preferred currency, and notifications.Unit tests guaranteed these features were stable and functional early on and allowed bugs to be caught early on and fixed.

7.1.3 Test Cases

7.1.3.1 Validate the Register/Login Page Text Fields:

The unit tests focused on validating the correctness and integrity of user input fields on login and registration forms, paying extra attention to fields for email, password, and confirm password.

1. **Confirm Password Validation Tests**
   These tests ensure that the confirmation password provided by the user meets security and match criteria:

a. If confirm password has fewer than 8 characters, an error message alerts the user about the minimum number of characters.

b. If the confirm password is not exactly the same as the original password, an error message notifies the user to correct the mismatch.

c. If the confirm password is the same as the original password and meets the length criteria, the validation succeeds.

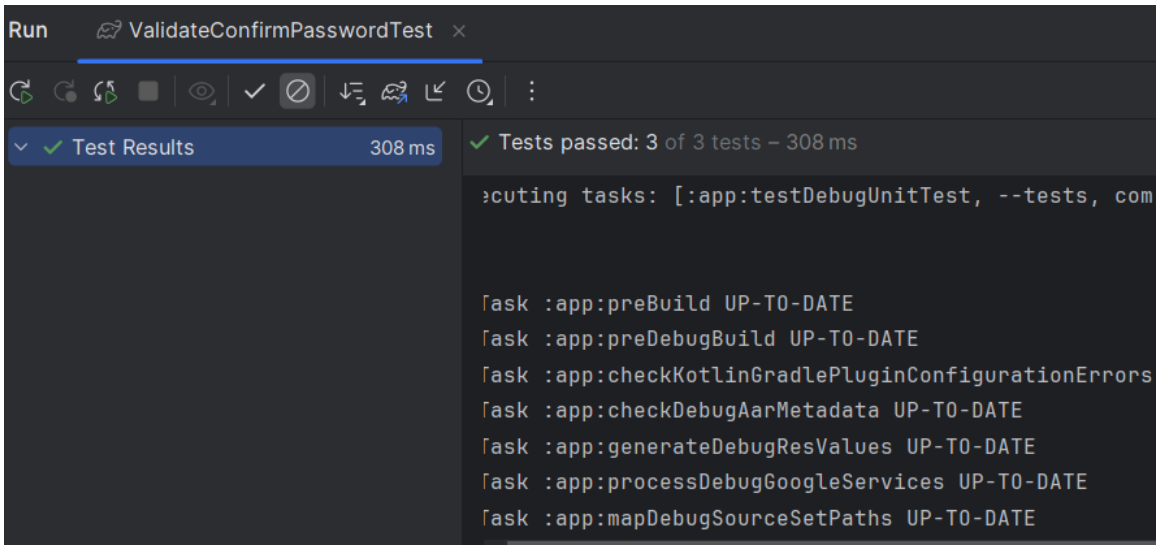


Figure 7-1 Validate Password Testcases

2. **Email Validation Tests**

These tests validate the email input field to have the correct format and presence:

a. An empty email field triggers an error message requesting the user to enter an email address.

b. An ill-formatted email (e.g., missing '@' or domain parts) prompts an error message to input a valid email.

c. A correctly formatted email validates correctly.

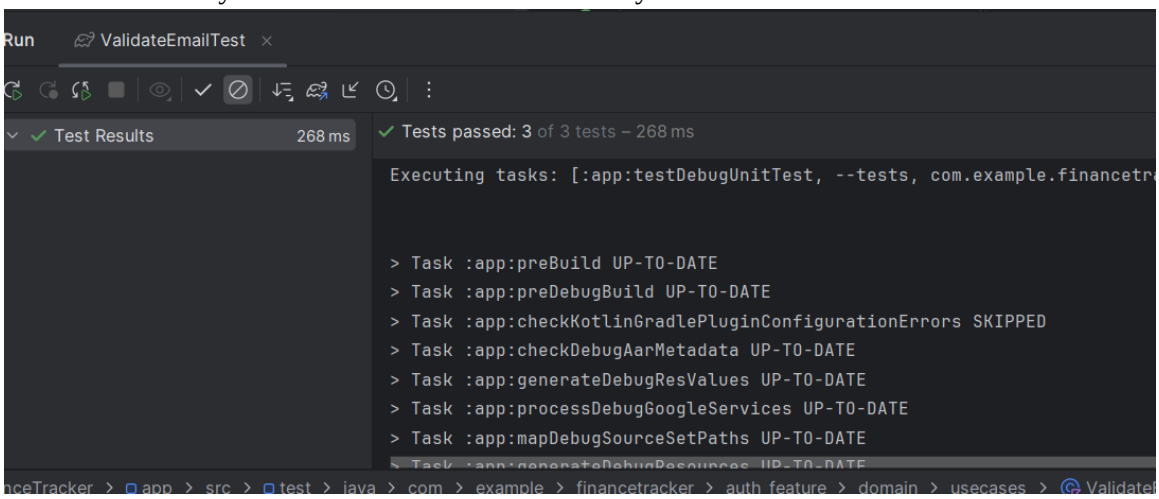


Figure 7-2 Validate Email Testcases

3. **Password Validation Tests**

Password validation tests check for strength and complexity requirements:

a. Passwords that are shorter than 8 characters prompt an error with the minimum required length.

b. Passwords that do not contain any special character prompt an error for at least one special character.

c. Passwords lacking an uppercase letter or digit invoke an error that necessitates at least one uppercase letter or digit value.

d. Passwords meeting all criteria pass validation without issues.

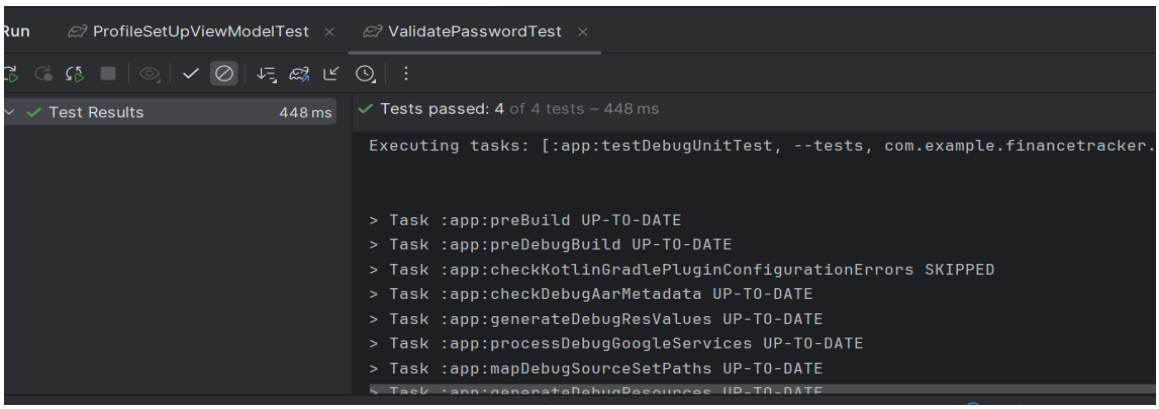
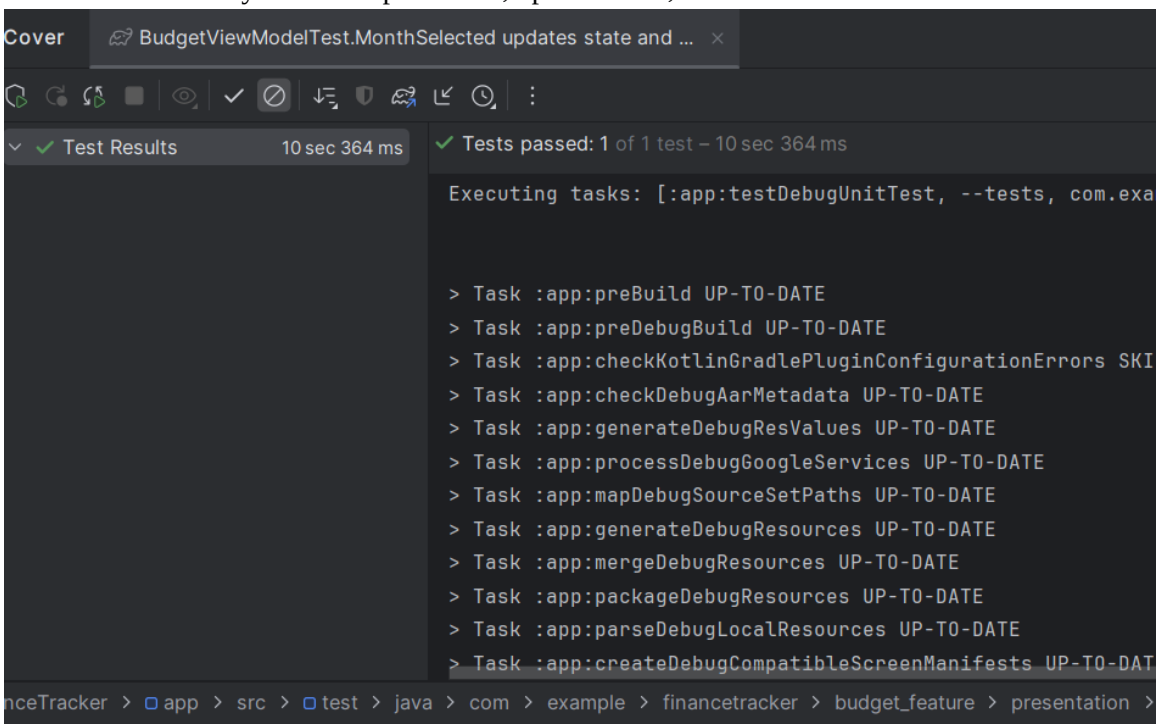


Figure 7-3 Validate Confirm Password Testcases

These unit tests ensure the fundamental user inputs are validated correctly for security and usability so as to prevent common user errors and making the app's authentication system more secure.

7.1.3.2 BudgetViewModel

The unit tests for the BudgetViewModel comprehensively cover various user interactions and state changes related to budget management in the finance tracker app. These tests ensure that the ViewModel correctly handles input events, updates state, and communicates validation results.



1. **Change Budget Input**
   This test verifies that when the user inputs a new budget amount, the ViewModel updates its internal state accordingly, reflecting the entered value immediately.

2. **Save Budget with Empty Input**
   This test ensures that attempting to save a budget without entering any amount results in a validation failure. The ViewModel emits an error event notifying that the budget field cannot be empty.

3. **Change Alert Threshold**
   The alert threshold represents the percentage at which users want to be notified about approaching budget limits. This test confirms that updating this threshold updates the ViewModel's state correctly.

4. **Toggle Receive Budget Alerts**
   This test checks that enabling or disabling budget alert notifications updates the ViewModel's state flag, reflecting user preferences for receiving alerts.

5. **Toggle Create Budget State**
   This test validates that changing the internal flag which indicates whether the user is currently creating a budget is properly updated in the state.

6. **Change Budget with Decimal Input**
   This test confirms the ViewModel can handle budget inputs with decimal values, ensuring state consistency when users input precise budget amounts.

7. **Save Budget with Valid Input**
   When a valid budget amount is entered, this test verifies that the ViewModel triggers the insertion of the budget into local storage and emits a success event. It also mocks the repository interactions to simulate a scenario where no existing budget for the current month is present. The test further asserts that the saved budget contains correct user ID, amount, and a valid timestamp.

8. **Month and Year Selection**
   This test simulates the user selecting a specific month and year for which the budget applies. It verifies that the ViewModel loads the corresponding budget data from local storage and updates its state with budget amount, currency symbol, alert threshold, and notification preferences. It also mocks user profile data to ensure the correct currency symbol is displayed.

Overall, these tests validate that the BudgetViewModel correctly processes user input, manages state transitions, handles validation, and interacts with use cases for persistent storage. This coverage is critical to ensure a smooth and reliable budgeting experience for users.

7.1.3.3 ProfileSetupViewModel

The test cases for the ProfileSetUpViewModel primarily focuses on validating the user profile setup process, ensuring that input validation, profile data submission, and error handling work as expected. The tests utilize mocking of use cases to isolate the ViewModel logic and use coroutine test dispatchers to manage asynchronous flows.

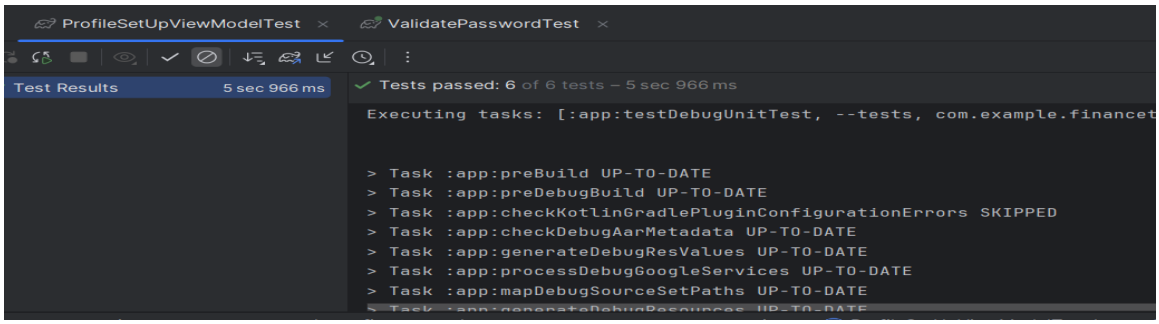


Figure 7-4 ProfileSetupViewModel Testcases

1. **Validate Fields Failure Sends Failure Event**
   This test verifies that when the user inputs invalid profile details (e.g., an invalid first name), the ViewModel correctly performs validation using the use cases and emits a failure event with the appropriate error message. It mocks the name validation to fail while other fields pass, ensuring the validation stops and notifies the user of the first failure encountered.

2. **Validate Names Event Emits Failure When Name Is Invalid**
   This test checks that when the name validation use case returns failure, the ViewModel emits a failure event specifically for the name validation step. It isolates name validation and confirms the ViewModel responds properly with the failure message.

3. **Validate Phone Number Event Emits Success When Phone Number Is Valid**
   This test confirms that when a valid phone number is provided, the ViewModel runs phone number validation and emits a success event indicating the phone number is valid. It mocks phone number validation to always succeed and asserts the corresponding success event is sent.

4. **Validate Country Event Emits Success When Country Is Valid**
   This test ensures that when a valid country is selected, the ViewModel triggers country validation use case and emits a success event. It mocks the country validation to succeed and verifies the event emitted reflects successful validation.

5. **Validate Fields Success Sends Success Event**
   This test verifies that when all profile fields (name, phone number, country, and currency) are valid, the ViewModel proceeds with updating the user profile, saving data to the local database, and keeping the user logged in. It mocks all related use cases to succeed and asserts the ViewModel emits a success event indicating that the profile setup was completed successfully.

6. **Validate Fields Throws Exception Sends Failure Event**
   This test simulates a scenario where all validations pass but an exception occurs during the saving process (e.g., database insertion error). It mocks the insert operation to throw an exception and verifies that the ViewModel catches the exception and emits a failure event with the error message from the exception.

7.1.3.4 Add Transactions Event (AddTransactionViewModel)

The unit tests for the AddTransactionViewModel comprehensively cover user interactions and validation logic involved in adding new financial transactions within the finance tracker app. These tests ensure that the ViewModel correctly processes user inputs, validates transaction details, manages data saving both locally and in the cloud, and emits appropriate success or failure events based on validation outcomes

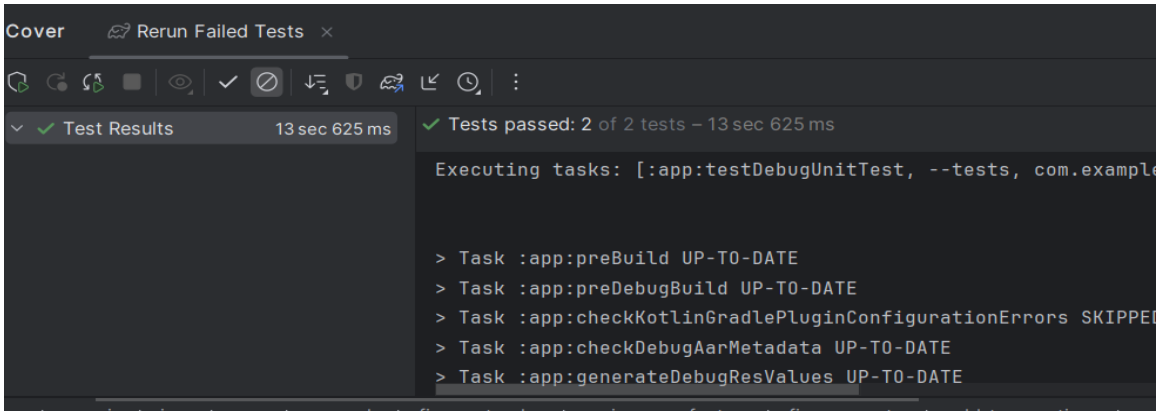


Figure 7-5 Add Transactions Event Testcases

1. **Add Transactions Sends Success Event When Validation Passes and Local + Cloud Save Succeeds**
   This test verifies that when all transaction input validations (empty field, price, and category) pass successfully, the ViewModel initiates the process to save the transaction locally and remotely. It mocks a positive internet connection and simulates successful insertion by returning a transaction ID. The test confirms that the ViewModel emits a success event, indicating the transaction was added correctly without errors.

2. **Add Transactions Sends Failure Event When Validation Fails**
   This test simulates a failure in input validation, such as when the transaction name is empty. The ViewModel receives a failed validation response with an appropriate error message. The test checks that the ViewModel emits a failure event containing the correct error details, ensuring that invalid transactions are not processed further or saved.

7.1.3.5 Add Saved Item (SavedItemViewModel)

The unit tests for the SavedItemViewModel thoroughly validate the behavior of the ViewModel responsible for adding and managing saved items in the finance tracker app. These tests ensure that

the ViewModel properly handles user inputs, performs validation checks on item details such as name and price, manages currency selection, and correctly responds to both successful and failed save operations. The tests also cover the ViewModel's ability to emit appropriate validation events signaling success or failure, thereby guaranteeing reliable user feedback and robust item management. More Figures for Add Saved Items Testcases are further mentioned in Appendices 12.3.8

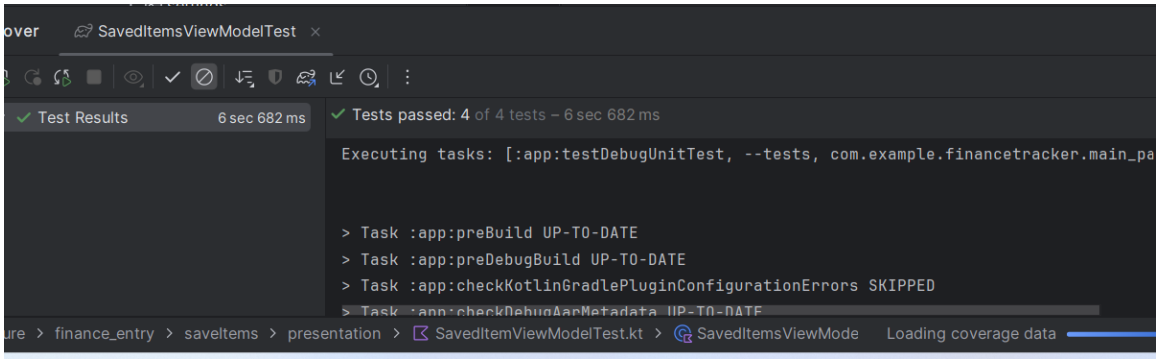


Figure 7-6 Add Save Item Testcases

1. **Submitting an Item with Valid Inputs Emits a Success Event**
   This test verifies that when all required inputs—item name, item price, and currency details—are valid and pass the validation checks, the ViewModel successfully processes the submission. It mocks positive internet connectivity and local save operations, then confirms that the ViewModel emits a success validation event, indicating the item was saved correctly.

2. **Submitting an Item with an Invalid Name Emits a Failure Event**
   This test simulates the scenario where the item name is empty or invalid. The validation logic detects this and returns a failure result with a descriptive error message ("Item name required"). The ViewModel reacts by emitting a failure validation event containing this message, thus preventing the item from being saved and providing user feedback.

3. **Submitting an Item with an Invalid Price Emits a Failure Event**
   This test addresses the case when the item price is missing or invalid. The validation detects the empty or incorrect price input and responds with a failure message ("Item price required"). The ViewModel subsequently emits a failure event with this message to inform the user and block the submission.

4. **Submitting an Item When Local Save Operation Fails Emits a Failure Event**
   This test covers error handling during the save operation itself. Even if all inputs are valid, a failure such as an exception during the local save process (e.g., database failure) is simulated. The ViewModel catches this exception and emits a failure event with the error message ("Local save failed"), ensuring that save errors are gracefully handled and communicated.

Overall, these unit tests guarantee that the SavedItemViewModel robustly validates input data, manages currency selection, handles both local persistence and cloud sync flags, and appropriately signals the UI with success or error states. This ensures a smooth and error-resistant user experience during the process of adding saved items in the app.

7.1.4 Test coverage:

While some key components such as the Register/Login page text fields and the BudgetViewModel have achieved full unit test coverage, other features like the ProfileSetupViewModel, SavedItemViewModel, and AddTransactionViewModel currently have partial coverage ranging from 46% to 57%. Overall, the project currently has approximately 10% unit test coverage. This limited coverage is primarily due to time constraints. Moving forward, there are plans to expand and improve the test suite by writing additional test cases to ensure more comprehensive validation and higher reliability across all features. More Test coverage figures are further mentioned in Appendices 12.5

Test Cases	Coverage for that feature
Validate the Register/Login Page Text Fields	100%
BudgetViewModel	100%
ProfileSetupViewModel	57%
Add Saved Item (SavedItemViewModel)	50%
Add Transactions Event (AddTransactionViewModel)	46%

Table 2 Test Cases Coverage

7.1.5 Manual Testing

The application was tested manually on a physical Android device to ensure seamless interaction between the UI and backend elements. During this phase, all the principal functionalities including user signup, login, budget management, profile setup, saved items, and processing of transactions were thoroughly exercised. The manual testing involved verifying the user interactions, input validation, navigation flows, and local as well as cloud data persistence. This interactive testing method facilitated the identification and closure of UI inconsistencies, functional faults, and performance problems in real usage scenarios with assurance of the app's stability and responsiveness before deployment.

7.2 Usability Testing

7.2.1 Purpose

For evaluating my finance tracker app, a combination of surveys and usability testing was used. These methods provided complementary insights into user experience and ensured thorough feedback for refinement. A diverse group of 20 participants was selected to ensure various perspectives were considered. They represented different age ranges, genders, and occupations. Usability testing offered in-depth qualitative insights by observing participants interact with app

features such as budget setting and notifications. Real-time challenges, like navigating graphical expense representations was identified. Additionally, it validated the intuitiveness and value of features like transaction graphs and secure authentication.

7.2.2 Recruitment and Orientation

Participants were selected based on their interest in financial management and familiarity with mobile apps. A diverse user base was targeted, considering factors like age, occupation, and tech-savviness. Participants for the survey were chosen based on their experience with tracking personal expenses and using mobile applications. This criterion ensured that respondents had the necessary background to provide insightful feedback. Once they agreed to participate, they were given a brief overview of the study's goals. The introduction emphasized that participation was voluntary, and confidentiality of responses was guaranteed. Participants were also made aware of the purpose of the survey and how their feedback would be used.

At the start of the session, participants were briefed on the purpose of the usability test. They were assured that the goal was to evaluate the app's functionality, not their performance. The confidentiality of their responses was emphasized. The survey was distributed to participants via a direct link to Google Forms. The survey itself was divided into various sections, with each segment addressing a different aspect of the expense tracker app, such as its usability, the effectiveness of features, and overall user satisfaction. The questions were a combination of multiple-choice items, Likert scale ratings, and open-ended responses, which allowed both quantitative and qualitative feedback. Clear and concise instructions were provided to ensure participants understood how to complete the survey. The collected data from the completed surveys were stored securely and handled with utmost care to maintain participant anonymity. Ethical guidelines were strictly followed, highlighting the voluntary nature of the survey and the respectful treatment of user data. These measures were in place to guarantee the reliability of the collected information while adhering to high standards of ethical research practice.

7.2.3 Introduction to the App and Display of Diff screens

At the beginning of the usability testing session, participants were provided with a mobile device that had a prototype version of the Expense Tracker app installed. This prototype included several key screens that represent the core features and functionality of the application. The purpose of this phase was to introduce participants to the app's user interface and ensure they understood the primary functions available on each screen before interacting with the app.

Each screen was presented to the participants, accompanied by a clear and concise explanation of its purpose and how it contributes to the overall user experience. Below is a breakdown of the main screens that were demonstrated:

i. **Dashboard Screen:**
   This was the first screen introduced, functioning as the app’s central hub. It provides a comprehensive overview of the user’s financial status, including total spending for the current month, the remaining amount in their budget, and the progress towards any defined savings goals. The layout is designed for quick insights, using progress bars and summary cards for intuitive understanding.

ii. Add Transaction Screen:

This screen enables users to manually add financial transactions such as income or expenses. Participants were shown how to input key transaction details including the amount, transaction type (Income or Expense), category, date and time, currency, and any optional notes. Users can also link previously saved items or select from recurring transactions. The goal is to make the process fast and straightforward to encourage frequent usage.

iii. View Records Screen:

This section allows users to review all previously added transactions. Transactions are listed chronologically with filtering and searching options, allowing users to sort records by category, date range, or amount. Tapping on an entry brings up detailed information and options to edit or delete the transaction.

iv. Charts Screen:

The Charts screen visually represents the user's spending behavior through various graphical formats, such as pie charts and bar graphs. For example, the pie chart displays a breakdown of spending by category (e.g., Rent, Food, Travel), helping users identify areas where they spend the most. These visual insights are useful for self-assessment and financial planning.

v. Settings Screen:

The Settings screen serves as the control panel for personalizing and configuring the app. Key options demonstrated include:

- Profile Management: View and edit personal details such as name, email, country, phone number, and base currency.
- Category Management: Add, remove, or edit income and expense categories based on personal preferences.
- Budget Settings: Configure or adjust monthly budget limits to track and control spending.
- Dark Mode: Toggle between light and dark themes to enhance visual comfort based on user preference or time of day.
- Logout: Securely sign out from the current session.

Each screen was carefully designed to align with user expectations and improve financial management experiences. Participants were encouraged to ask questions during the walkthrough to clarify any aspects before engaging in actual interaction tasks during the testing phase.

7.2.4 Usability Testing Scenario

7.2.4.1 Initial App Exploration and Profile Setup

- Objective: Evaluate first-time user experience and ease of setting up user profiles.

- Task: Participants set up accounts, create profiles, and configure their budgeting preferences.
- Feedback to Gather:
    - First Impressions: Assess the app's design, navigation, and user interface (e.g., color scheme, layout).
    - Ease of Setup: Evaluate how straightforward account creation and preference settings are.
    - Difficulties or Confusion: Note any unclear steps or issues encountered.

7.2.4.2 Adding and Categorizing Expenses

- Objective: Assess the ease of tracking expenses and setting categories.
- Task: Participants add multiple types of expenses (e.g., groceries, rent, entertainment) and organize them into categories.
- Feedback to Gather:
    - Ease of Use: Evaluate the simplicity of adding expenses and navigating categories.
    - Relevance of Categories: Check if the available categories cover user needs.
    - Suggestions for Improvement: Identify missing or redundant features.

7.2.4.3 Setting and Monitoring Budget Goals

- Objective: Evaluate the ease of setting and adjusting monthly budget goals.
- Task: Participants set a monthly budget and explore how to edit or reset it through the Settings screen.
- Feedback to Gather:
    - Ease of Use: Is the budget setup process simple and intuitive?
    - Clarity: Are users aware of how their budget is applied or tracked in the app?
    - Suggestions for Improvement: Allow category-wise budget limits, offer confirmation or guidance after budget is set

7.2.4.4 Visualizing Transactions and Budget Trends

- Objective: Evaluate the graphical representation of expenses and budget tracking.
- Task: Participants review pie charts, bar graphs, or other visualizations showing their spending habits.
- Feedback to Gather:
    - Design Appeal: Check if graphs are visually appealing and easy to understand.
    - Insights and Clarity: Assess whether the visuals provide actionable insights.
    - Suggestions for Visualization: Identify missing data points or alternative graph styles.

7.2.4.5 Managing Multiple User Profiles

- Objective: Test how the app supports multi-user functionality.
- Task: Participants switch between different user profiles and manage their individual expenses.
- Feedback to Gather:

- Ease of Switching Profiles: Evaluate how quickly users can switch accounts.
- Data Separation: Assess whether expenses are securely and properly managed for different users.
- User Experience: Gather insights into the intuitiveness of multi-user features.

7.2.5 Study Result

Here are eight survey questions tailored for evaluating Expense Tracker App. These are based on Don Norman's principles of Interaction Design (e.g., consistency, feedback, and usability) and Jordan's Four Pleasures Framework (physio-pleasure, socio-pleasure, psycho-pleasure, and ideo pleasure). These questions can help assess usability, design, and emotional appeal: (Norman, 2004) (Kefalidou G., 2016)

i. How intuitive and clear are the features of the app, such as adding transactions, viewing reports, or setting budgets?
(Rate from 1 - Not Intuitive at All to 5 - Extremely Intuitive)

ii. How consistent is the design across different sections of the app, such as the transaction view and graphical analysis?
(Rate from 1 - Very Inconsistent to 5 - Very Consistent)

iii. If you make a mistake, such as entering incorrect data or forgetting a passcode, how easy is it to recover and continue using the app?
(Rate from 1 - Very Difficult to 5 - Very Easy)

iv. How comfortable and accessible is the app to use, considering the font size, color scheme, and navigation design?
(Rate from 1 - Very Uncomfortable to 5 - Very Comfortable)

v. Does the app stimulate you intellectually, for example, through insights from transaction trends or the use of visual data representations?
(Yes/No) Please elaborate.

vi. Do you feel the app enhances your ability to manage finances and communicate financial goals with others (e.g., family, friends, or business partners)?
(Rate from 1 - Not at All to 5 - Significantly)

vii. How enjoyable is it to use the app in terms of screen readability, navigation, and touch responsiveness?
(Rate from 1 - Not Enjoyable to 5 - Extremely Enjoyable)

viii. Does the app align with your personal values, such as promoting financial discipline, technology innovation, or sustainability?
(Yes/No) Please explain.

The rest of the questions have been pasted in the appendix section for the in-depth references of the questionnaire.

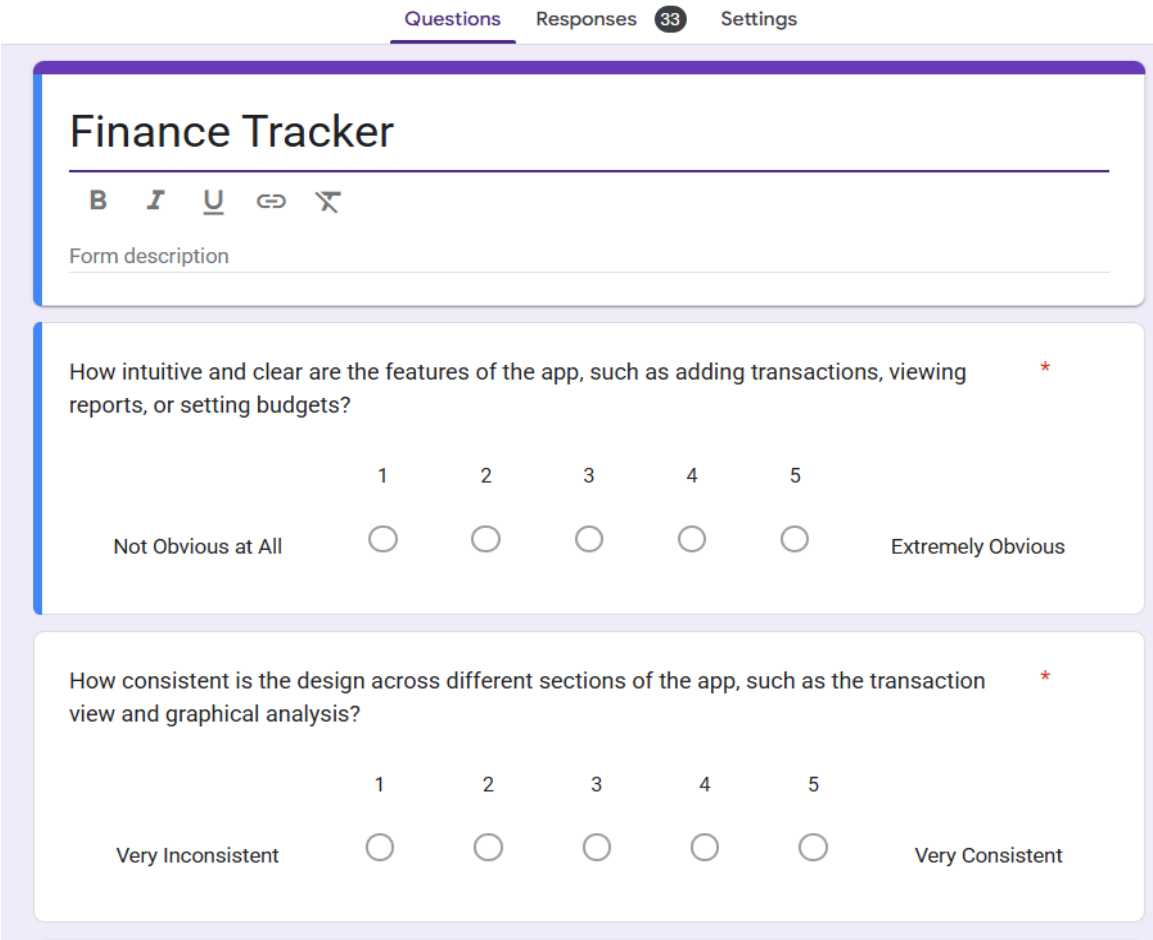


Figure 7-7 Google Form Questions

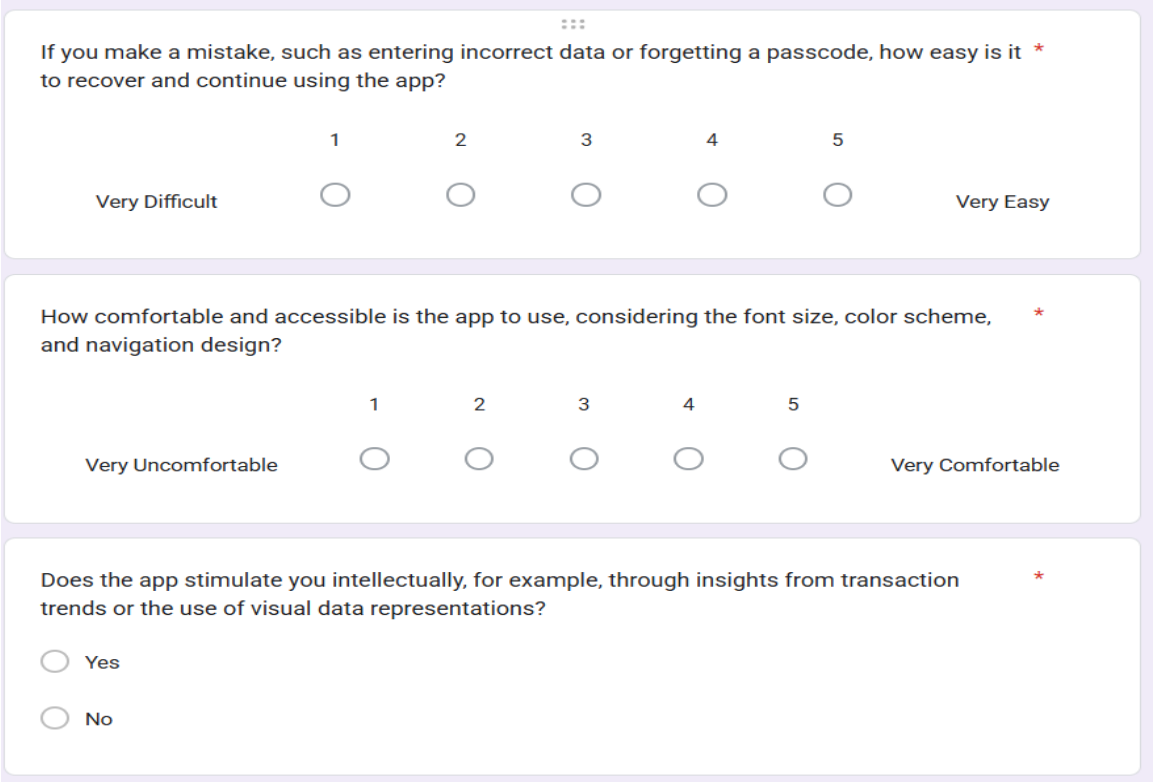


Figure 7-8 Google Form Questions

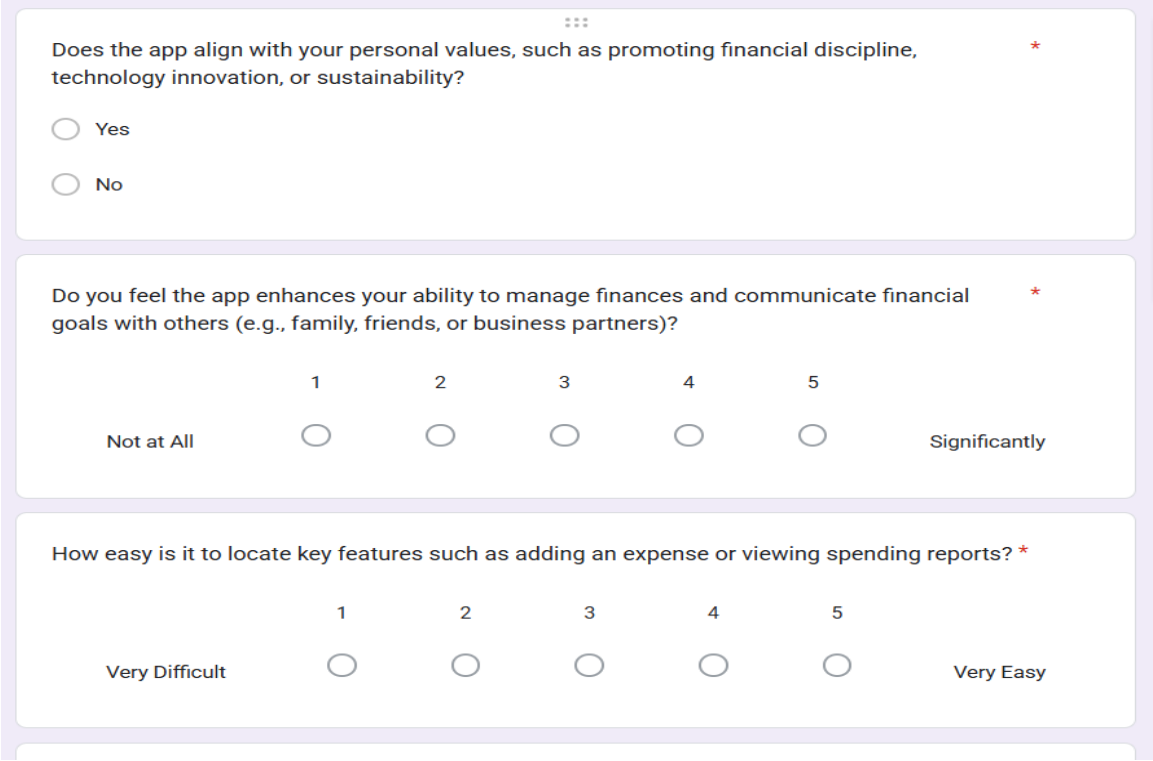


Figure 7-9 Google Form Questions

Results based on the survey questions:

Based on the feedback results from 20 participants, here's an analysis of the responses to each question using both descriptive statistics for rating questions and frequency counts for yes/no questions:

1. Affordance (Norman's Principle) (Norman, 2004) (Kefalidou G., 2016)

- Equal Split: 23 participants said "Yes", and 5 said "No"
- Implication: The significant majority who found the app's affordances clear indicates that the design elements effectively communicate their intended use, making the app easy to navigate.

2. Consistency (Norman's Principle) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.64
- Most Frequent Rating: 4 (9 times)
- Variability: Ratings are somewhat spread out, with a lean towards higher ratings.

3. Mapping (Norman's Principle) (Norman, 2004) (Kefalidou G., 2016)

- Equal Split: 24 participants said "Yes", and 4 said "No"
- Implication: The majority of users found the app's mapping to be intuitive, meaning that the relationships between controls and their effects are well- aligned with user expectations.

4. Constraints (Norman's Principle) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.54
- Most Frequent Rating: 5 (10 times)
- Variability: Ratings ranged from 1 to 5, showing a wide range of experiences.

5. Visibility (Norman's Principle) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.57
- Most Frequent Rating: 4,5 (8 times)
- Variability: Ratings ranged from 1 to 5, showing a wide range of experiences.

6. Feedback (Norman's Principle) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.57
- Most Frequent Rating: 4 (11 times)
- Variability: Ratings ranged from 1 to 5, showing a wide range of experiences.

7. Physio Pleasure (Jordan's Pleasure Framework) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.46
- Most Frequent Rating: 4,5(8 times each)
- Variability: Opinions vary widely, indicating differing levels of physical comfort and ease of use.

8. Ideo Pleasure (Jordan’s Pleasure Framework) (Norman, 2004) (Kefalidou G., 2016)

- Equal Split: 21 participants said "Yes", and 7 said "No"
- Implication: This split suggests that the app's intellectual stimulation is polarizing among users.

9. Pyscho Pleasure (Jordan’s Pleasure Framework) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.46
- Most Frequent Rating: 4 (9 times)
- Variability: A noticeable number of participants found the app enjoyable and found control over their finances.

10. Socio Pleasure (Jordan’s Pleasure Framework) (Norman, 2004) (Kefalidou G., 2016)

- Mean Rating: 3.57
- Most Frequent Rating: 5 (10 times)
- Variability: A noticeable number of participants found the app helpful and found control over their finances.

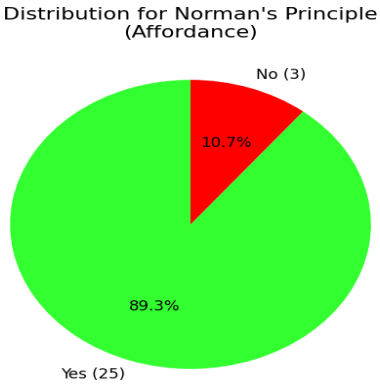


Figure 7-10 Survey Result (Affordance)

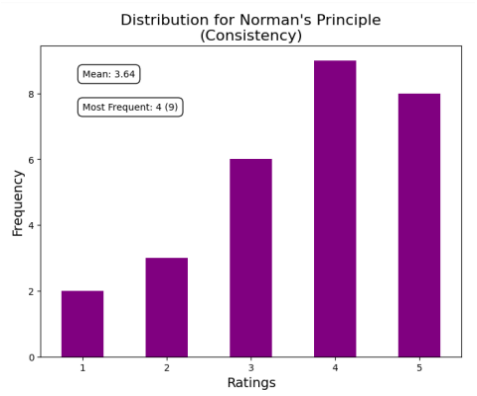


Figure 7-11 Survey Result (Consistency)

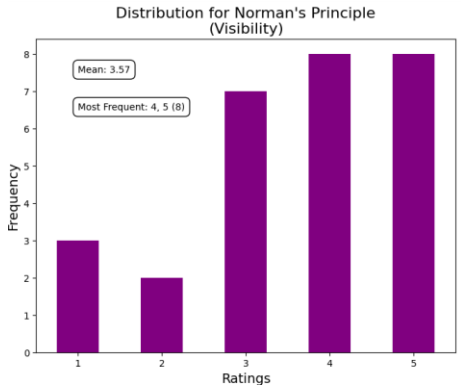


Figure 7-13 Survey Result (Visibility)

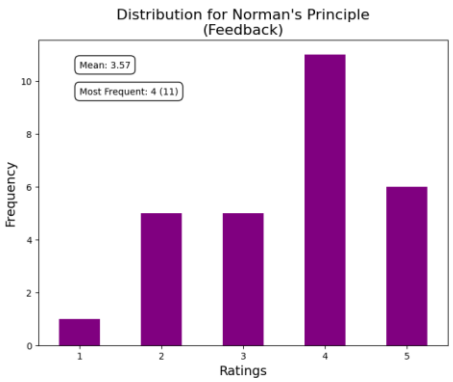


Figure 7-13 Survey Result (Feedback)

More graphical Figures can be found in Appendix 12.2

8 Challenges and Solutions

During the development of the application, several technical, UI/UX, and performance-related challenges emerged. Each obstacle was carefully analyzed, and effective solutions were implemented to ensure a seamless user experience and maintain application stability.

8.1 Technical Challenges

8.1.1 Country API Service Downtime:

The RESTful API used to fetch country data experienced intermittent downtimes, disrupting the country selection process during profile setup. To mitigate this issue, a local fallback mechanism was introduced using preloaded country data, ensuring uninterrupted access even when the external API was unavailable.

8.1.2 Exchange Rate API Call Limits:

The exchange rate service enforced a daily request limit, which made it impractical to perform live conversions frequently. To overcome this, a scheduled background task was implemented using WorkManager, configured to run once daily at 6:00 AM. This task fetches the latest exchange rates and stores them in the local Room database. All currency conversions throughout the day are then performed using the locally stored data, reducing API dependency while maintaining accuracy.

8.1.3 Forgot Password Redirection:

Upon requesting a password reset, users received a link via email that opened in a web browser rather than redirecting to the application. Since deep linking for password resets was not supported directly through Firebase, this limitation remained a known issue. As a temporary solution, clear instructions were provided to guide users through resetting their password via the web.

8.2 UI/UX Challenges

8.2.1 Onboarding Redirection for First-Time Users:

There was initially no mechanism to differentiate between new and returning users, leading to confusion during onboarding. This was resolved by introducing a persistent flag (profileUpdated)

stored in SharedPreferences. The app checks this flag on launch and redirects first-time users to the account setup flow accordingly.

8.2.2 Theme Switching (Dark Mode):

Implementing seamless dark mode switching was challenging, especially when users changed the theme from the settings or used the system default. The issue stemmed from the need to maintain UI state across different screens. A shared ViewModel was used between the MainActivity and SettingsScreen, allowing theme changes to propagate instantly without requiring reinitialization.

8.2.3 Displaying User Name in Settings:

The settings screen initially displayed "null null" for the user's name due to the ViewModel not having updated state after login. This was resolved by explicitly reinitializing the user profile data on the Home screen after login, ensuring state consistency across the app.

8.2.4 View Transactions Screen Crash:

A crash occurred on the View Transactions screen due to a conflict between LazyColumn and a scrollableState modifier. The issue was traced back to layout constraints. To fix this, a fixed height was applied to the LazyColumn, stabilizing the UI and preventing crashes.

8.2.5 Incorrect Navigation in Records View (Row Table):

While navigating to specific pages in the records view (e.g., different tabs or table pages), the app consistently landed on the first screen. This was resolved by passing the appropriate page number as a navigation argument, ensuring users are directed to the correct screen as intended.

8.3 Performance Challenges

8.3.1 Device Compatibility with Credential Saving API:

The application used the Android Credential Saving API to offer a smoother login experience by automatically filling in saved credentials. However, this feature is only supported on newer Android versions and devices with the appropriate Google Play Services configuration. On older devices, the bottom sheet for selecting saved credentials did not appear, causing the login process to silently cancel. This led to confusion, as there was no feedback to the user. To handle this, additional checks were added to detect unsupported devices, and the app now falls back to the manual login flow with appropriate messaging when the credential save prompt is unavailable.

9 Future Work

The current version of the Finance Tracker application realizes its primary goals through facilitating sign-ups, tracking transactions, managing budgets, and presenting informative visualizations. However, application development is never complete, especially in a domain as unpredictable and user-centered as individual finances. As with constant feedback, technology enhancements, and shifting user needs, the application needs to be continually reassessed and fine-tuned. Future work will be centered on expanding the features of what is currently in place, making continued advances in the technical foundation, enhancing user experience, exploring backend integrations, and ensuring scalability to ensure long-term success.

9.1 Feature Enhancements

One of the core areas of future development is enhancing the feature set of the app to give users a smarter and smoother experience.

Recurring Transactions Feature:

The recurring transaction switch is one of the primary future features. In most real-life situations, users encounter recurring spends or revenues—like rent, electricity bills, EMIs, or wages. They have to currently input these manually every month. Now with the addition of a toggle feature, users can label any transaction as recurring, and the app will automatically post it on the same date every month. This not only saves time but also reduces the risk of forgetting to record habitual financial transactions. The periodic transactions can also be managed from a separate page where users can make changes or delete them according to need.

Visualization Feature:

Besides recurring entries, the visualization ability of the app will also be expanded. Whereas bar charts and pie charts take a snapshot of spending or income by categories, users require a view that is more detailed of financial trends. A line chart will be added to display daily expenditure or income behavior over a period. This will enable users to spot anomalies or spikes in the daily spend from a visual perspective, encouraging improved budgeting discipline. For example, if a user is seeing excessively high weekend spends, they might adjust their spending behavior accordingly.

Budget Feature:

Also, the budgeting system will be enhanced. At present, the app does provide the functionality to set a rough monthly budget, but different users have different spends they want to prioritize. To make the users more specific, a category-based budgeting functionality will be introduced. This would enable users to reserve certain budget amounts for areas like Food, Transport, Entertainment, etc. By this, users would not only be reminded when they exceed their overall monthly budget but also when they cross certain limits in specific areas. This fine-grained budgeting would help users track spending better and develop healthier financial habits.

9.2 Technical Improvements

While the app is already built based on the principles of Clean Architecture, it's always possible to improve the internal organization in terms of being more maintainable, responsive, and scalable. With growing codebase and additional features, it becomes increasingly critical to adhere to a clear separation of concerns.

One of the technical goals for future development is to restructure parts of the project more strictly to Clean Architecture principles. This means going back to current domain, data, and presentation layers and refactoring them to be loosely coupled and modularized. Services and use cases will be redesigned to enable better code reuse, and duplicate or redundant code will be removed or abstracted as reusable utilities and components

Also, emphasis will be put on creating a more testable architecture. When there is a clearly defined domain layer and proper dependency injection (e.g., using Dagger Hilt), UI testing and unit testing can be performed more effectively. Clean and testable architecture prevents future changes and updates from inadvertently breaking present functionalities. Clean and testable architecture also provides safer collaboration if different developers are working on the project simultaneously.

9.3 User Experience (UX) Enhancements

User experience is the heart of any successful application. No matter how solid the backend or how extensive the feature set, if the interface is clunky or unsightly, users may be dissuaded from using the application. With that in mind, several UI and UX enhancements are on the way.

It is to be kept in mind that even most universal applications—such as WhatsApp, Instagram, or YouTube—are occasionally updating their user interfaces for better interaction and fulfilling evolving requirements. The same approach will be followed here. In subsequent releases, efforts will be made to rethink UI elements for better consistency, usability, and responsiveness. More fluid transitions, modern layout techniques, and more intuitive navigation patterns, for example, will be adopted to enhance the usability.

One of the most significant developments planned along these lines is a shift towards the use of Canvas APIs within Jetpack Compose in order to draw custom and dynamic graphics. Instead of relying heavily on default charting libraries, using Canvas allows more interactive and animated graphical elements, especially in the analytics domains. This will not only improve the visual display of the app, but also provide end users with more useful and interactive mechanisms for interacting with their financial data.

In addition, efforts will be put in to add access features such as larger font sizes, high contrast schemes, and support for screen readers so that the app is accessible and usable to a broad base of users.

9.4 Integration with External Services

Today, the app employs Firebase for user authentication, cloud storage, and real-time sync services. While Firebase offers a speedy and reliable way to incorporate backend capabilities without worrying about the server, it comes with some constraint in flexibility, custom logic, and data flow ownership.

In the near future, a major course of development will be the creation and integration of a custom backend system. Spring Boot (Java/Kotlin) or Ktor (Kotlin) technologies are the choices for this development. These backend frameworks offer powerful APIs, database management, and authentication features so that there can be complete control over how data is processed, validated, and sent to the app.

By having a backend developed within the company, it is possible to add functionalities such as customized notifications, logging, advanced user analysis, and even AI-driven recommendations for expenses. Further, reliance upon third-party systems would be minimized, making the app more solid, customizable, and cost-effective in the long term.

9.5 Scalability and Cloud Optimization

As the application gains more users and handles more financial data, it will be important that the infrastructure scale without loss of performance. In future releases, there will be an in-depth exploration of cloud service optimization, especially data retrieval and synchronization-related services.

Multi-device sync efforts will be made to allow easy use of the app on multiple phones or tablets with up-to-date information. Caching methods will be investigated and optimized to reduce latency, and data retrieval operations will be optimized using pagination and indexing to make it faster on large transaction histories.

As for database optimization, queries should be light, speedy, and optimized for mobile use. Redundant reads and writes will be minimized to save both bandwidth as well as processing cycles. If a custom backend is adopted, it would be implemented with scalable architecture—e.g., maybe with microservices, containerization (e.g., Docker), or load-balancing techniques for faster performance when there are large volumes of users.

Finally, as part of cloud optimization, there will be cost analysis to ensure that usage and storage fees are kept at acceptable levels while delivering the best user experience.

10 Conclusion

This project successfully developed an Android-based personal finance management application aimed at helping individuals manage their financial activities with greater ease and accuracy. The application provides a secure, intuitive platform for users to track their income, monitor their expenses, set financial goals, and visualize their financial data through interactive charts. Through the integration of modern technologies such as Kotlin, Jetpack Compose, Firebase Authentication, Firestore, and Room Database, the app offers both offline accessibility and real-time cloud synchronization, ensuring a reliable user experience.

The implementation of key features like budget management, multi-currency support, data visualization, and secure authentication addresses major challenges faced by users in managing their day-to-day finances. In addition, user-centric features such as reminders, currency conversion, and data export enhance the overall usability and functionality of the application.

Testing and evaluation through user feedback demonstrated high levels of satisfaction with the app's usability, design, and performance, while also highlighting areas for potential future enhancements. Despite challenges faced during the development, appropriate solutions were applied to ensure the delivery of a robust and scalable application.

In conclusion, this project not only achieved its initial objectives but also laid a strong foundation for future improvements. Further work can be undertaken to enhance personalization, introduce AI-driven recommendations, expand platform support, and provide deeper financial insights to users. The project demonstrates how thoughtfully designed digital tools can empower users to develop better financial habits, promoting long-term financial literacy and well-being.

11 References

[1] Oswal, S., & Koul, S. (2013). Big data analytic and visualization on mobile devices. In *Proc. Nat. Conf. New Horizons IT-NCNHIT* (p. 223).

[2] Wong, C. K., & Salleh, M. N. M. (2023). Personal Finance and Budgeting Mobile Application,"CashSave". *Applied Information Technology And Computer Science*, *4*(1), 1372-1387.

[3] Imawan, R., Putra, W. P., Alqahtani, R., Milakis, E. D., & Dumchykov, M. (2025). Enhancing Financial Literacy in Young Adults: An Android-Based Personal Finance Management Tool. *Journal of Hypermedia & Technology-Enhanced Learning*, *3*(1), 64-88.

[4] French, D., McKillop, D., & Stewart, E. (2021). The effectiveness of smartphone apps in improving financial capability. In *Financial literacy and responsible finance in the fintech era* (pp. 6-22). Routledge.

[5] Girdhar, G., Kumar, S., Bhardwaj, A., & Sharma, M. (2024, September). Design and Development of Expense App. In *2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET)* (pp. 1-4). IEEE.

[6] Google Developers, *Official Android Documentation*, URL: https://developer.android.com.

[7] Google Firebase, *Firebase Documentation*, URL: https://firebase.google.com/docs.

[8] JetBrains, *Kotlin Coroutines Guide*, URL: https://kotlinlang.org/docs/coroutines-overview.html.

[9] Google Material Design, *Material Design Guidelines*, URL: https://material.io/design.

[10] Google Developers, *Room Database Guide*, URL: https://developer.android.com/training/data-storage/room.

[11] P. Lackner, *Philipp Lackner [YouTube Channel]*, YouTube. URL: https://www.youtube.com/@PhilippLackner.

[12] Shah, V. (2025). Finance Tracker [Google Form]. URL: https://docs.google.com/forms/d/e/1FAIpQLSdFxKfUDtfoqlzX43mCwaZVH270Qh2yDwKKG3O3J4cn4GRiIg/viewform?usp=sf_link

12 Appendices

12.1 Survey Form

Do the app's visual indicators (e.g., icons, labels) help you quickly understand your current financial status? *

- ◯ Yes
- ◯ No

How satisfied are you with the immediate feedback provided after adding an expense or setting a budget? *

	1	2	3	4	5	
Very Dissatisfied	◯	◯	◯	◯	◯	Very Satisfied

Does the app notify you effectively when you approach or exceed your budget? *

- ◯ Yes
- ◯ No

Does the app effectively prevent you from entering duplicate expenses or invalid data? *

- ◯ Yes
- ◯ No

How helpful are the app's prompts or warnings when an action (e.g., entering an expense) is incomplete or incorrect? *

	1	2	3	4	5	
Not Helpful	◯	◯	◯	◯	◯	Extremely Helpful

How intuitive is the process for adding expenses, setting budgets, or categorizing transactions? *

	1	2	3	4	5	
Not Intuitive	◯	◯	◯	◯	◯	Very Intuitive

Do the app's design elements, such as buttons or dropdown menus, clearly suggest their functions? *

- Yes
- No

How consistent are the design and interaction patterns across different sections of the app? *

	1	2	3	4	5	
Very Inconsistent	○	○	○	○	○	Very Consistent

Do similar tasks (e.g., editing or deleting an entry) follow the same interaction logic throughout the app? *

- Yes
- No

If you make a mistake, such as entering the wrong amount, how easy is it to correct the error? *

	1	2	3	4	5	
Very Difficult	○	○	○	○	○	Very Easy

Does the app provide clear error messages when something goes wrong (e.g., when syncing data)? *

- Yes
- No

How visually appealing do you find the app's overall design and color scheme? *

	1	2	3	4	5	
Not Appealing	○	○	○	○	○	Very Appealing

Does the app avoid unnecessary complexity, focusing on the most essential features? *

- Yes
- No

How easy is it to discover advanced features, such as data visualization or export options? *

	1	2	3	4	5	
Very Difficult	○	○	○	○	○	Very Easy

Does the app suggest useful features or provide tips for better financial management? *

- Yes
- No

Does the app make managing your finances feel intellectually stimulating, such as through insightful spending analysis? *

- Yes
- No

Does the app allow or encourage sharing financial goals or insights with others (e.g., family budgets or spending comparisons)? *

- Yes
- No

Do you feel a sense of accomplishment or control over your finances when using the app? *

	1	2	3	4	5	
Not at All	○	○	○	○	○	Extremely Empowering

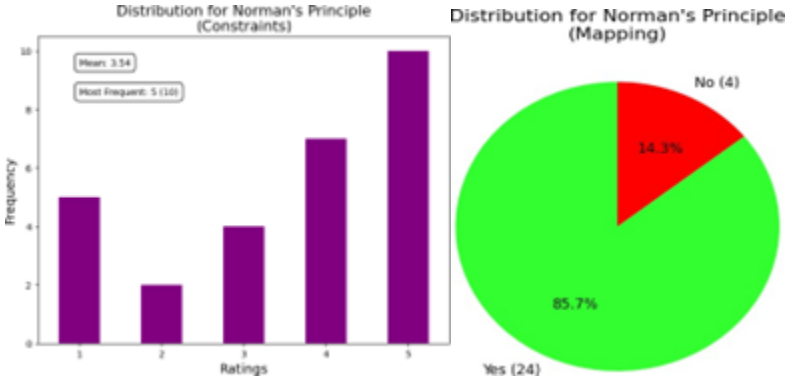
Do you feel a sense of accomplishment or control over your finances when using the app? *

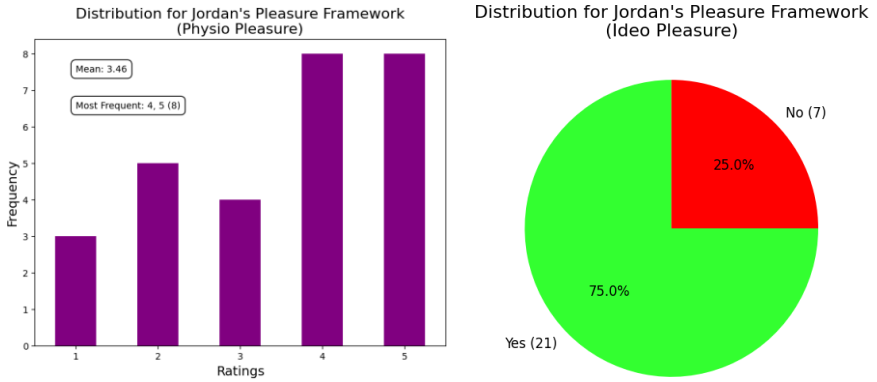
	1	2	3	4	5	
Not at All	○	○	○	○	○	Extremely Empowering

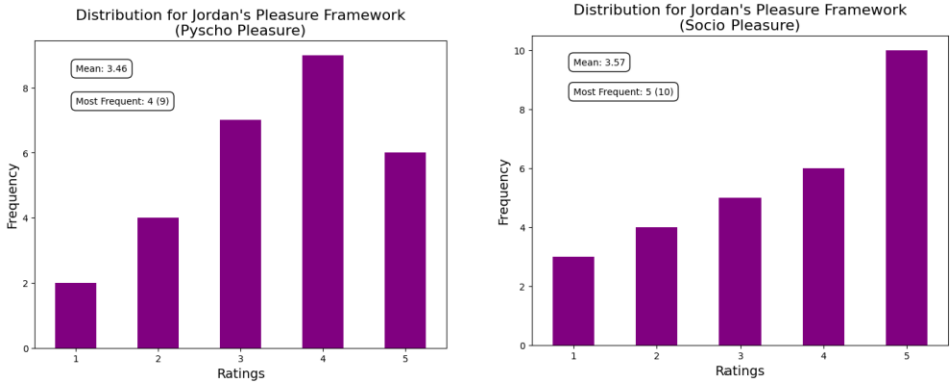
How comfortable is it to use the app in terms of screen readability, navigation, and touch responsiveness? *

	1	2	3	4	5	
Very Uncomfortable	○	○	○	○	○	Very Comfortable

12.2 Survey Result

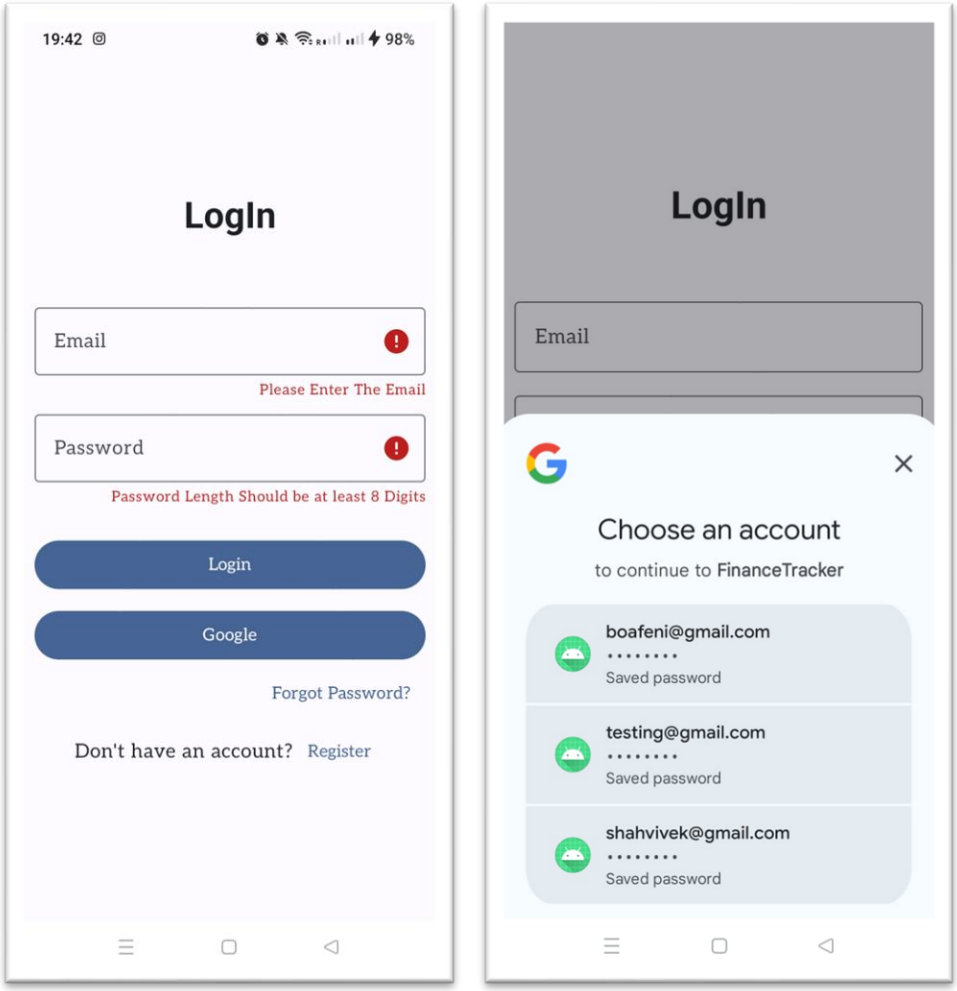




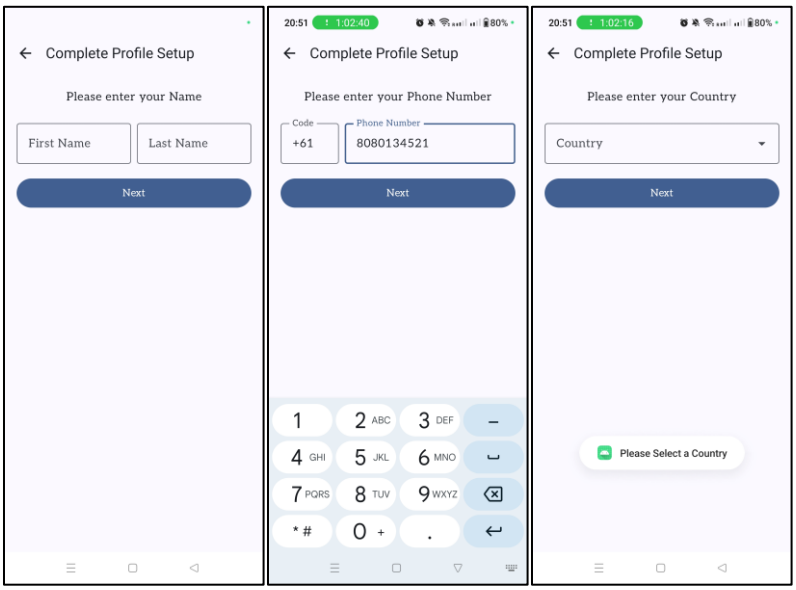


12.3 Front-End UI

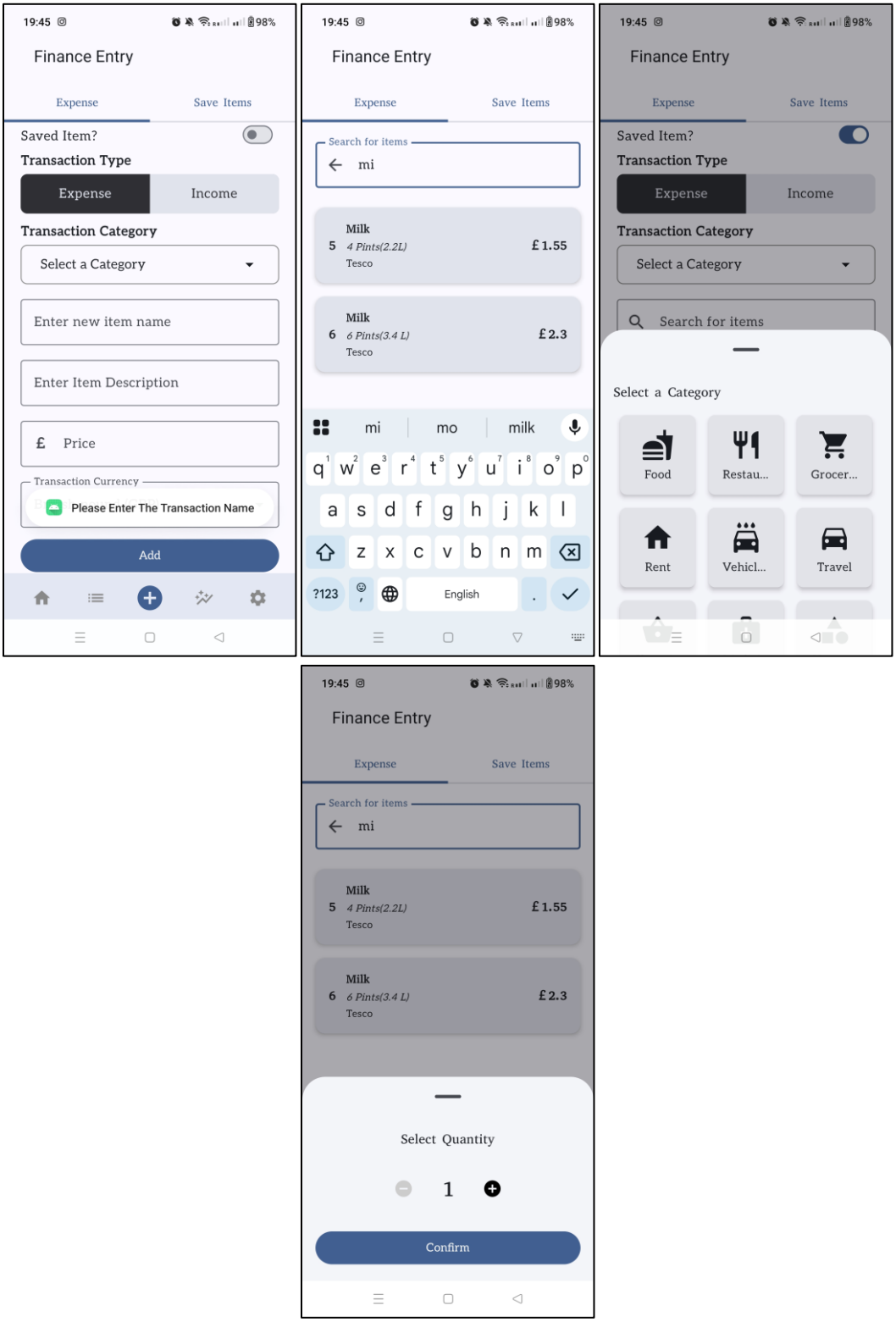
12.3.1 Login/Register Page



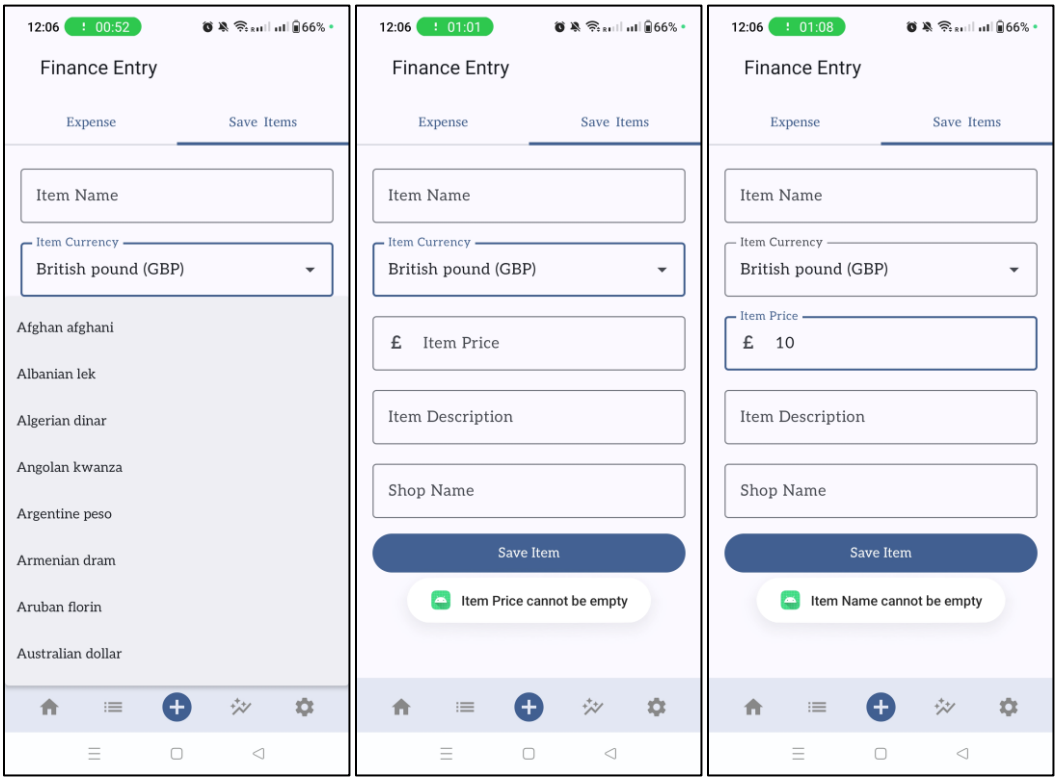
12.3.2 Profile Setup Page



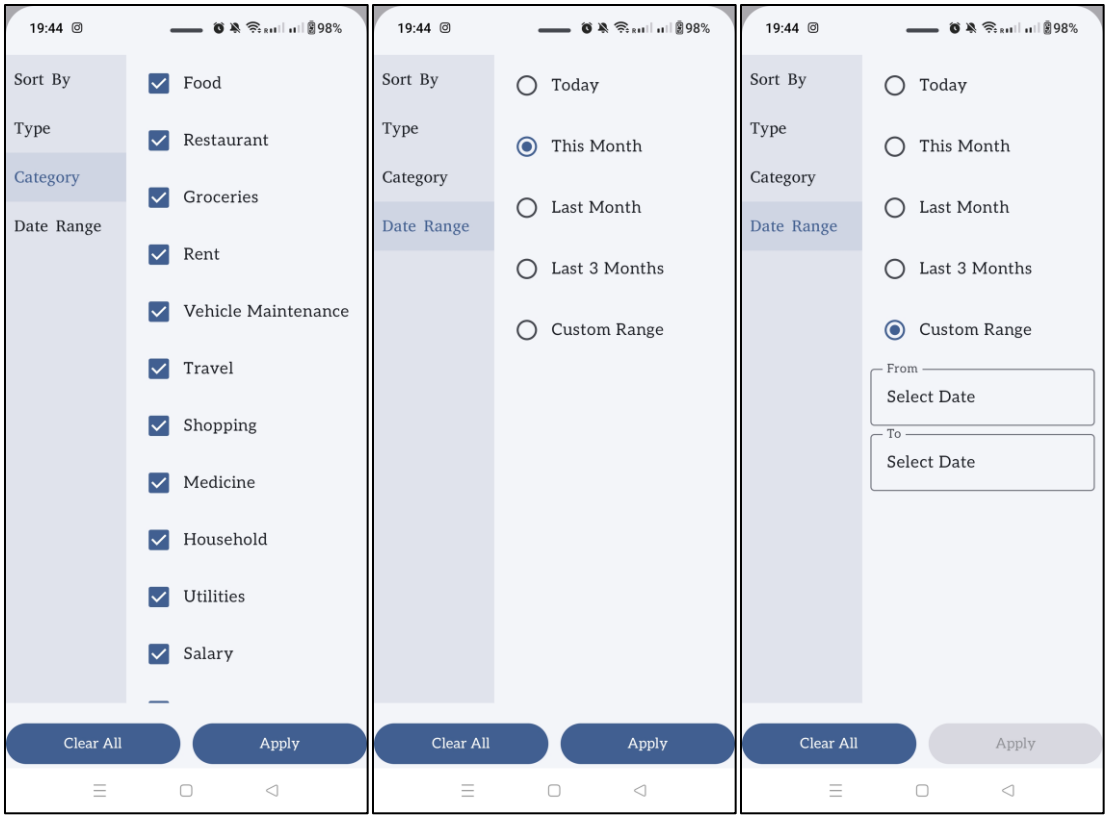
12.3.3 Add Transaction Page



12.3.4 Add Saved Item Page

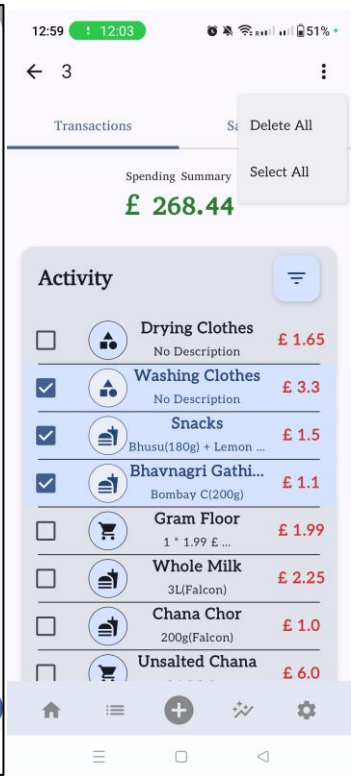


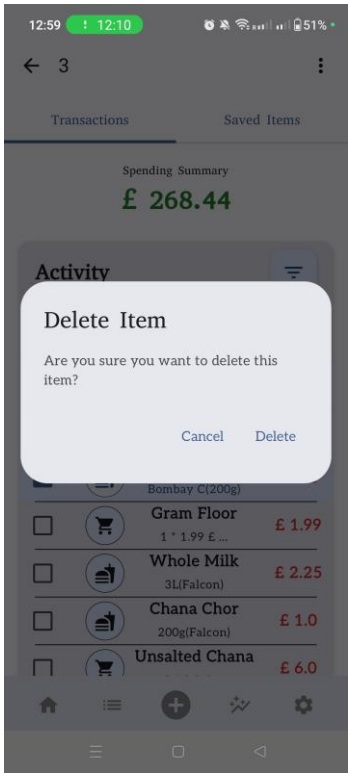
12.3.5 Transactions Records



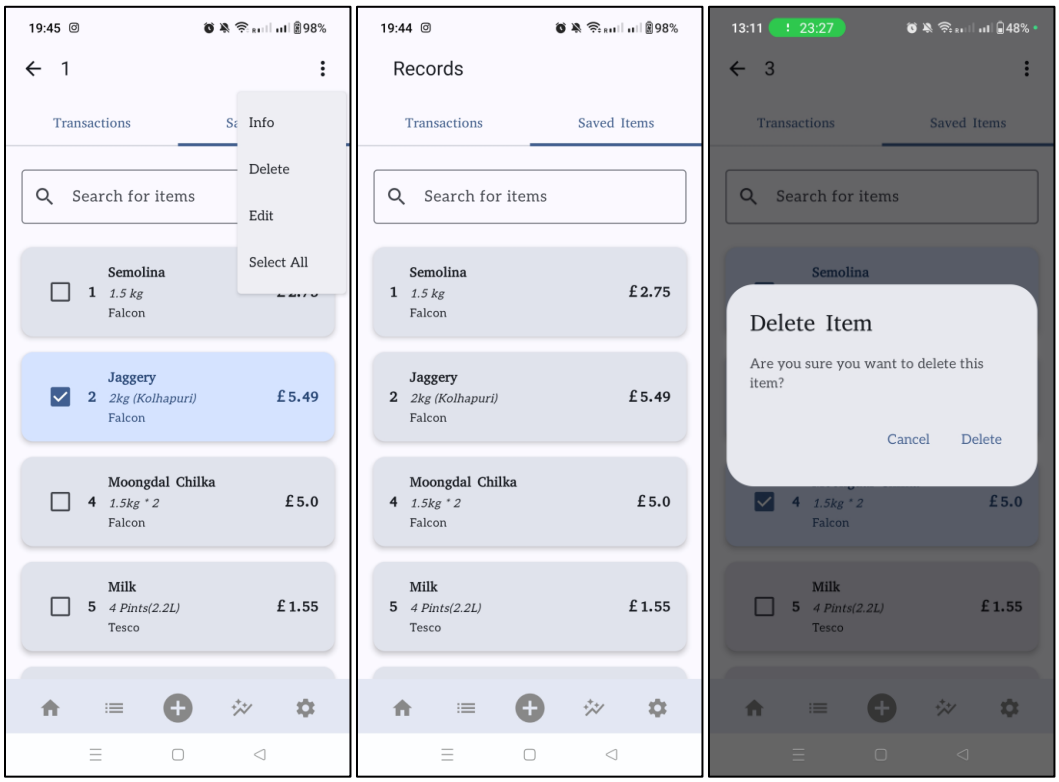

19:44
98%
Sort By
Type
Category
Date Range
Ascending
Descending
Clear All
Apply


19:44
98%
Sort By
Type
Category
Date Range
Income
Expense
Both
Clear All
Apply

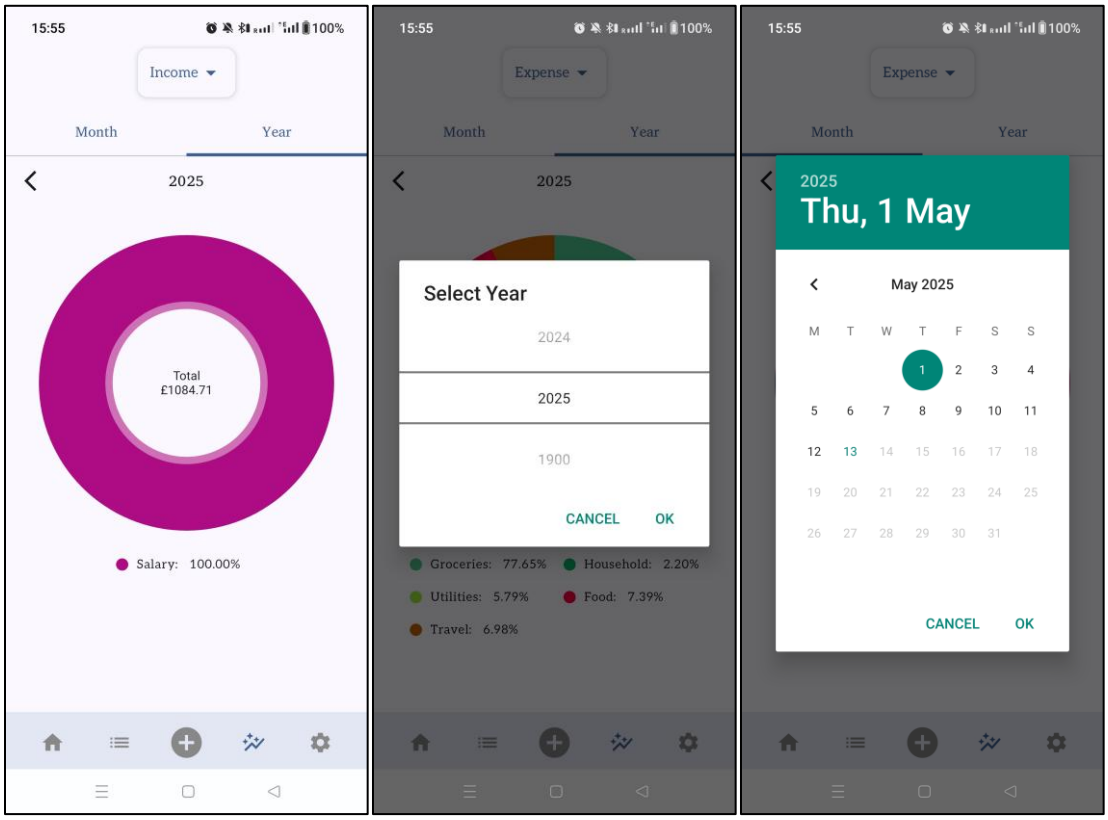

12:59
12:03
51%
3
Transactions
Delete All
Select All
Spending Summary
£ 268.44
Activity
Drying Clothes
No Description
£ 1.65
Washing Clothes
No Description
£ 3.3
Snacks
Bhusu(180g) + Lemon ...
£ 1.5
Bhavnagri Gathi...
Bombay C(200g)
£ 1.1
Gram Floor
1 * 1.99 £ ...
£ 1.99
Whole Milk
3L(Falcon)
£ 2.25
Chana Chor
200g(Falcon)
£ 1.0
Unsalted Chana
£ 6.0


12:59
12:10
51%
3
Transactions
Saved Items
Spending Summary
£ 268.44
Activity
Delete Item
Are you sure you want to delete this item?
Cancel
Delete
Bombay C(200g)
Gram Floor
1 * 1.99 £ ...
£ 1.99
Whole Milk
3L(Falcon)
£ 2.25
Chana Chor
200g(Falcon)
£ 1.0
Unsalted Chana
£ 6.0

12.3.6 Saved Item Records

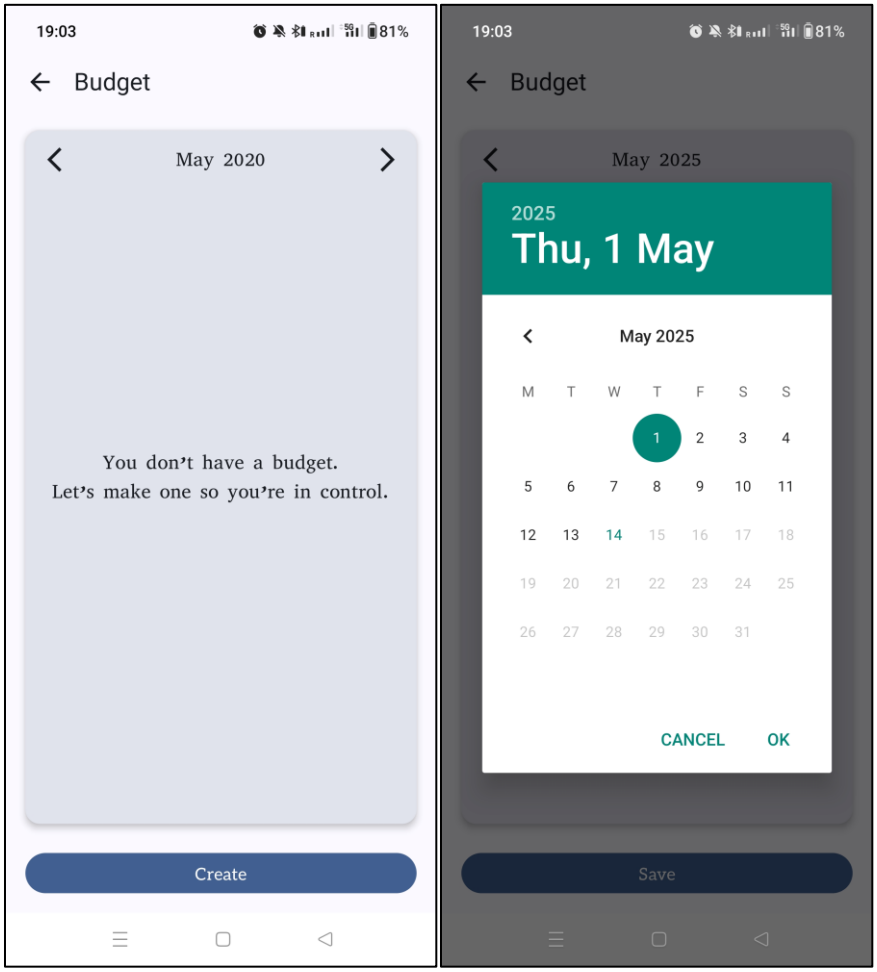


12.3.7 Charts Screen

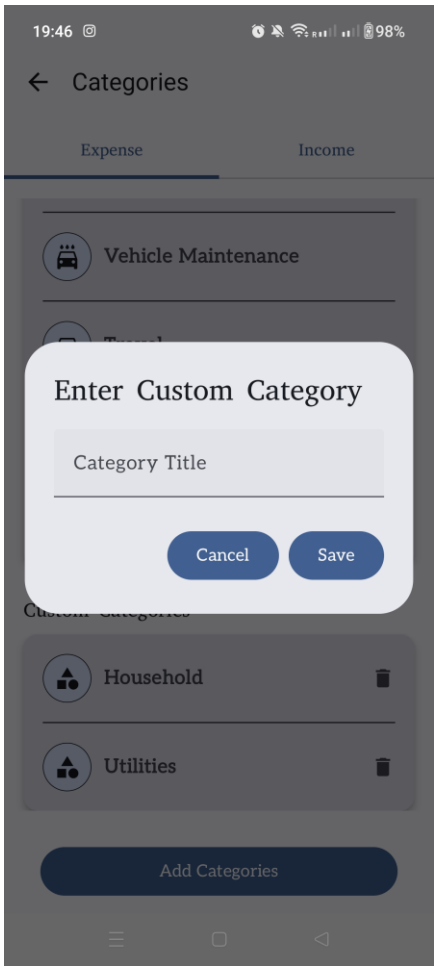


12.3.8 Budget Screen

```
if (states.createBudgetState) {
    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ) {
        NoBudgetMessage()
    }

} else {

    Box(
        modifier = Modifier.fillMaxSize(),
        contentAlignment = Alignment.Center
    ) {
        Column(
            modifier = Modifier,
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.Start
        ) {
```



12.3.9 Categories Screen



12.4 Back-End Setup

12.4.1 Database Setup

```
@Dao
interface BudgetDao {

    @Query("SELECT * FROM monthly_budgets WHERE userId = :userId AND month = :month AND year = :year LIMIT 1")
    suspend fun getBudgetForMonth(userId: String, month: Int, year: Int): BudgetEntity?

    @Query("SELECT * FROM monthly_budgets WHERE userId = :userId AND cloudSync == 0")
    fun getAllUnSyncedBudget(userId: String): Flow<List<BudgetEntity>>

    @Upsert
    suspend fun insertBudget(budget: BudgetEntity)
}
```

```
@Database(
    entities = [BudgetEntity::class],
    version = 3
)
abstract class BudgetDatabase: RoomDatabase() {
    abstract val budgetDao: BudgetDao
    companion object{
        const val DATABASE_NAME = "budget_db"
    }
}
```

```
@Entity(
    tableName = "monthly_budgets",
    indices = [Index(value = ["userId", "month", "year"], unique = true)]
)
data class BudgetEntity(
    @PrimaryKey val id: String,
    val userId: String,
    val amount: Double,
    val month: Int, // 1 to 12
    val year: Int,  // e.g., 2025
    val updatedAt: Long,
    val cloudSync: Boolean,
    val receiveAlerts: Boolean,
    val thresholdAmount: Float
)
```

```kotlin
val BUDGET_MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL( sql: "ALTER TABLE monthly_budgets ADD COLUMN cloudSync INTEGER NOT NULL DEFAULT 0")
    }
}

val BUDGET_MIGRATION_2_3 = object : Migration(2,3) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL( sql: "ALTER TABLE monthly_budgets ADD COLUMN receiveAlerts INTEGER NOT NULL DEFAULT 0")
        db.execSQL( sql: "ALTER TABLE monthly_budgets ADD COLUMN thresholdAmount REAL NOT NULL DEFAULT 0.0")
    }
}
```

```kotlin
@Dao
interface CategoryDao {

    @Query(" SELECT * FROM CategoryEntity WHERE (type = :type AND uid = :uid) OR (type = :type AND isCustom = 0)")
    fun getCategories(type: String, uid: String): Flow<List<CategoryEntity>>

    @Query(" SELECT * FROM CategoryEntity WHERE (type = :type AND uid = :uid AND isCustom = 1)")
    fun getCustomCategories(type: String, uid: String): Flow<List<CategoryEntity>>

    @Query(" SELECT * FROM CategoryEntity WHERE (type = :type AND isCustom = 0)")
    fun getPredefinedCategories(type: String): Flow<List<CategoryEntity>>

    @Upsert
    suspend fun insertCategories(categories: List<CategoryEntity>)

    @Upsert
    suspend fun insertCategory(category: CategoryEntity)

    @Query("SELECT COUNT(*) FROM CategoryEntity")
    suspend fun getCategoryCount(): Int

    @Query("DELETE FROM CategoryEntity WHERE categoryId = :categoryId")
    suspend fun deleteCustomCategory(categoryId: Int)
}
```

```kotlin
@Database(
    entities = [CategoryEntity::class],
    version = 3
)
abstract class CategoryDatabase: RoomDatabase() {
    abstract val categoryDao: CategoryDao
    companion object{
        const val DATABASE_NAME = "category_db"
    }
}
```

```kotlin
@Entity
data class CategoryEntity(

    @PrimaryKey(autoGenerate = true)
    val categoryId: Int,
    val uid: String?,
    val name: String,
    val type: String,
    val icon: String,
    val isCustom: Boolean
)
```

```kotlin
val CATEGORY_MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        // Step 1: Create a temporary table with the new schema
        db.execSQL( sql: """
            CREATE TABLE category_temp(
                uid TEXT NOT NULL PRIMARY KEY,
                name TEXT NOT NULL,
                type TEXT NOT NULL,
                icon TEXT NOT NULL,
                isCustom INTEGER NOT NULL
            )
        """)

        // Step 2: Copy data from the old table to the new table
        db.execSQL( sql: """
            INSERT INTO category_temp(uid, name, type, icon, isCustom)
            SELECT
                ROW_NUMBER() OVER (ORDER BY name) AS uid,   -- Generate a random UID for each category
                name,
                type,
                icon,
                0 AS isCustom  -- Assuming predefined categories are not custom, set to 0 (false)
            FROM CategoryEntity
        """)

        // Step 3: Drop the old table
        db.execSQL( sql: "DROP TABLE CategoryEntity")

        // Step 4: Rename the new table to the old table's name
        db.execSQL( sql: "ALTER TABLE category_temp RENAME TO CategoryEntity")
```

```kotlin
val CATEGORY_MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(db: SupportSQLiteDatabase) {

        db.execSQL( sql: "DROP TABLE CategoryEntity")


        db.execSQL(
            """
            CREATE TABLE IF NOT EXISTS CategoryEntity (
                categoryId INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                uid TEXT NOT NULL,
                name TEXT NOT NULL,
                type TEXT NOT NULL,
                icon TEXT NOT NULL,
                isCustom INTEGER NOT NULL
            )
            """.trimIndent()
        )
    }
}
```

```kotlin
@Entity
data class DeletedSavedItemsEntity(
    @PrimaryKey
    val itemId: Int,
    val userUID: String
)
```

```kotlin
@Dao
interface DeletedSavedItemsDao {

    @Upsert
    suspend fun insertDeletedSavedItems(deletedSavedItemsEntity: DeletedSavedItemsEntity)

    @Query("SELECT * FROM DeletedSavedItemsEntity WHERE userUid = :uid")
    fun getAllDeletedSavedItems(uid: String): Flow<List<DeletedSavedItemsEntity>>

    @Query("DELETE FROM DeletedSavedItemsEntity WHERE itemId =:itemId")
    suspend fun deleteDeletedSavedItemsByIds(itemId: Int)

}
```

```kotlin
@Dao
interface DeletedTransactionDao {

    @Upsert
    suspend fun insertDeletedTransaction(deletedTransactionsEntity: DeletedTransactionsEntity)

    @Query("SELECT * FROM DeletedTransactionsEntity WHERE userUid = :uid")
    fun getAllDeletedTransactions(uid: String): Flow<List<DeletedTransactionsEntity>>

    @Query("DELETE FROM DeletedTransactionsEntity WHERE transactionId =:transactionId")
    suspend fun deleteSelectedDeletedTransactionsByIds(transactionId: Int)



}
```

```kotlin
@Entity
data class DeletedTransactionsEntity(
    @PrimaryKey(autoGenerate = true)
    val transactionId: Int,
    val userUid: String,
)
```

```kotlin
@Database(
    entities = [SavedItemsEntity::class, DeletedSavedItemsEntity::class],
    version = 3
)
abstract class SavedItemsDatabase: RoomDatabase() {

    abstract val savedItemsDao: SavedItemsDao
    abstract val deletedSavedItemsDao: DeletedSavedItemsDao

    companion object{
        const val DATABASE_NAME = "save_items_db"
    }
}
```

```kotlin
@Entity
data class SavedItemsEntity(
    @PrimaryKey(autoGenerate = true)
    val itemId: Int,
    val itemName: String,
    val itemCurrency: String,
    val itemPrice: Double?,
    val itemDescription: String?,
    val itemShopName: String?,
    val userUID: String,
    val cloudSync: Boolean
)
```

```kotlin
interface SavedItemsDao {

    @Upsert
    suspend fun insertSavedItems(savedItemsEntity: SavedItemsEntity)

    @Insert
    suspend fun insertSavedItemReturningId(savedItemsEntity: SavedItemsEntity): Long

    @Query("SELECT * FROM SAVEDITEMSENTITY WHERE userUID = :userUID")
    fun getAllSavedItems(userUID: String): Flow<List<SavedItemsEntity>>

    @Query("SELECT * FROM SAVEDITEMSENTITY WHERE userUID = :userUID AND cloudSync == false")
    fun getAllNotSyncedSavedItems(userUID: String): Flow<List<SavedItemsEntity>>

    @Query("SELECT * FROM SAVEDITEMSENTITY WHERE itemId = :itemId")
    fun getSavedItemById(itemId: Int): SavedItemsEntity

    @Query("DELETE FROM SavedItemsEntity WHERE itemId =:savedItemsId")
    suspend fun deleteSelectedSavedItemsByIds(savedItemsId: Int)

    @Query("UPDATE SavedItemsEntity SET cloudSync = :syncStatus WHERE itemId = :id")
    suspend fun updateCloudSyncStatus(id: Int, syncStatus: Boolean)
}
```

```kotlin
val SAVED_ITEM_MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        // Create a new table with autoGenerate for itemId and copy the data

        // Drop the old table
        db.execSQL( sql: "DROP TABLE `SavedItemsEntity`")


        db.execSQL( sql: """
            CREATE TABLE IF NOT EXISTS `SavedItemsEntity` (
                `itemId` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                `itemName` TEXT NOT NULL,
                `itemCurrency` TEXT NOT NULL,
                `itemPrice` REAL,
                `itemDescription` TEXT,
                `itemShopName` TEXT,
                `userUID` TEXT NOT NULL
            )
        """)
    }
}
```

```
val SAVED_ITEM_MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(db: SupportSQLiteDatabase) {

        // 1. Rename the existing table
        db.execSQL( sql: "ALTER TABLE `SavedItemsEntity` RENAME TO `SavedItemsEntity_old`")

        // 2. Create new table with `cloudSync` column
        db.execSQL(
            sql: """
            CREATE TABLE IF NOT EXISTS `SavedItemsEntity` (
                `itemId` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                `itemName` TEXT NOT NULL,
                `itemCurrency` TEXT NOT NULL,
                `itemPrice` REAL,
                `itemDescription` TEXT,
                `itemShopName` TEXT,
                `userUID` TEXT NOT NULL,
                `cloudSync` INTEGER NOT NULL DEFAULT 0
            )
            """
        )

        // 3. Copy the data from old table to new table, set `cloudSync = 0`
        db.execSQL(
            sql: """
            INSERT INTO `SavedItemsEntity` (
                itemId,
                itemName,
                itemCurrency,
```

```
@Database(
    entities = [TransactionsEntity::class, DeletedTransactionsEntity::class],
    version = 3
)
@TypeConverters(DateTypeConverters::class)
abstract class TransactionDatabase: RoomDatabase() {

    abstract val transactionDao: TransactionDao
    abstract val deletedTransactionDao: DeletedTransactionDao

    companion object{
        const val DATABASE_NAME = "transaction_db"
    }
}
```

```kotlin
@Entity
data class TransactionsEntity(
    @PrimaryKey(autoGenerate = true)
    val transactionId: Int,
    val transactionName:String,
    val amount: Double,
    val currency: String?,
    val convertedAmount: Double?,
    val exchangeRate: Double?,
    val transactionType: String,
    val category: String,
    val dateTime: Long,
    val userUid: String,
    val description: String?,
    val isRecurring: Boolean,
    val cloudSync: Boolean,

)
```

```kotlin
@Dao
interface TransactionDao {

    @Upsert
    suspend fun insertTransaction(transactionsEntity: TransactionsEntity)

    @Insert
    suspend fun insertTransactionReturningId(transactionsEntity: TransactionsEntity): Long

    @Query("SELECT * FROM TransactionsEntity WHERE userUid = :uid")
    fun getAllTransactions(uid: String): Flow<List<TransactionsEntity>>

    @Query("SELECT * FROM TransactionsEntity WHERE userUid = :uid AND cloudSync == false")
    fun getAllLocalTransactions(uid: String): Flow<List<TransactionsEntity>>

    @Query("SELECT * FROM TransactionsEntity WHERE transactionId = :transactionId")
    fun getAllTransactionsById(transactionId: Int): TransactionsEntity

    @Query("DELETE FROM TransactionsEntity WHERE transactionId =:transactionId")
    suspend fun deleteSelectedTransactionsByIds(transactionId: Int)

    @Query("UPDATE TransactionsEntity SET cloudSync = :syncStatus WHERE transactionId = :id")
    suspend fun updateCloudSyncStatus(id: Int, syncStatus: Boolean)


}
```

```kotlin
val TRANSACTIONS_MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL(
            sql: """CREATE TABLE TransactionsEntity (
                transactionId INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                transactionName TEXT NOT NULL,
                amount REAL NOT NULL,
                currency TEXT,
                convertedAmount REAL,
                exchangeRate REAL,
                transactionType TEXT NOT NULL,
                category TEXT NOT NULL,
                dateTime INTEGER NOT NULL,
                userUid TEXT NOT NULL,
                description TEXT,
                isRecurring INTEGER NOT NULL,
                cloudSync INTEGER NOT NULL
            )"""
        )
    }
}

val TRANSACTIONS_MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(db: SupportSQLiteDatabase) {
        // Create the new table for deleted transactions
        db.execSQL("""
            CREATE TABLE IF NOT EXISTS `DeletedTransactionsEntity` (
                `transactionId` INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
                `userUid` TEXT NOT NULL
            )
        """.trimIndent())
    }
```

```kotlin
@Dao
interface UserProfileDao {

    @Query("SELECT * FROM UserProfileEntity where uid = :uid")
    fun getUserProfile(uid: String): UserProfileEntity

    @Upsert
    suspend fun insertUserProfile(userProfileEntity: UserProfileEntity)

}
```

```kotlin
@Database(
    entities = [UserProfileEntity::class],
    version = 2
)
abstract class UserProfileDatabase:RoomDatabase() {

    abstract val userProfileDao: UserProfileDao
    companion object{
        const val DATABASE_NAME = "user_profile_db"
    }
}
```

```kotlin
@Entity
data class UserProfileEntity(

    @PrimaryKey
    val uid: String,
    val firstName: String ,
    val lastName: String ,
    val email: String,
    val baseCurrency: String?,
    val country: String ,
    val callingCode: String,
    val phoneNumber: String,
    val profileSetUpCompleted: Boolean
)
```

```kotlin
val USER_PROFILE_MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportSQLiteDatabase) {

        // Create a temporary table with the new schema
        db.execSQL( sql: """
            CREATE TABLE user_profile_temp(
                uid TEXT NOT NULL PRIMARY KEY,
                firstName TEXT NOT NULL,
                lastName TEXT NOT NULL,
                email TEXT NOT NULL,
                baseCurrency TEXT,
                country TEXT NOT NULL,
                callingCode TEXT NOT NULL,
                phoneNumber TEXT NOT NULL,
                profileSetUpCompleted INTEGER NOT NULL
            )
        """)

        // Copy data from the old table to the new table
        // If `uid` is null, insert a random string for `uid`
        db.execSQL( sql: """
            INSERT INTO user_profile_temp(uid, firstName, lastName, email, b
            SELECT
                email AS uid,  -- Use email as uid since `uid` column doesn'
                firstName,
                lastName,
                email,
```

12.4.2 Local Core Features

12.4.2.1 Local Storage of Country Details:

```
override suspend fun doWork(): Result {
        Log.d( tag: "WorkManagerCountries",  msg: "Received countries: ${countries.size}")

        val countryEntities = countries.map { country ->
            try {
                val entity = country.toEntity()
                Log.d( tag: "WorkManagerCountries",  msg: "Mapped to Entity: $entity")
                entity
            } catch (e: Exception) {
                Log.e( tag: "WorkManagerCountries",  msg: "Error mapping country: $country", e)
                throw e
            }
        }

        Log.d( tag: "WorkManagerCountries",  msg: "Inserting countries into Room...")
        countryDao.insertAll(countryEntities)

        val insertedData = countryDao.getAllCountries()
        Log.d( tag: "WorkManagerCountries",  msg: "Countries inserted successfully: ${insertedData.size}")
        Result.success()

    } catch (e: Exception) {
        Log.e( tag: "WorkManagerCountries",  msg: "Unexpected error occurred", e)
        Result.failure()
    }
}
```

```
override suspend fun insertCountries() {

    val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.CONNECTED) // Ensures work runs only when connected
        .build()

    val workRequest = OneTimeWorkRequestBuilder<PrepopulateCountryDatabaseWorker>()
        .setConstraints(constraints)
        .setBackoffCriteria( // Set retry strategy
            BackoffPolicy.LINEAR,
             backoffDelay: 30, TimeUnit.SECONDS // Minimum delay before retry
        )
        .addTag( tag: "PrepopulateCountries") // Add a tag for tracking logs
        .build()

    Log.d( tag: "WorkManagerCountries",  msg: "WorkManager enqueued: $workRequest")
    workManager.enqueueUniqueWork(
         uniqueWorkName: "PrepopulateCountries",
        ExistingWorkPolicy.KEEP,  // Ensures it runs even after app restart
        workRequest
    )
}
```

```kotlin
class InsertCurrencyRatesLocalPeriodically(
    private val currencyRatesLocalRepository: CurrencyRatesLocalRepository
) {
    suspend operator fun invoke(){
        return currencyRatesLocalRepository.insertCurrencyRatesLocalPeriodically()
    }
}
```

```kotlin
class InsertCurrencyRatesLocalOneTime(
    private val currencyRatesLocalRepository: CurrencyRatesLocalRepository
) {
    suspend operator fun invoke(){
        return currencyRatesLocalRepository.insertCurrencyRatesLocalOneTime()
    }
}
```

```kotlin
    applicationScope.launch {
        predefinedCategoriesUseCaseWrapper.insertPredefinedCategories()
        setupAccountUseCasesWrapper.insertCountryLocallyWorkManager()
        viewRecordsUseCaseWrapper.deleteMultipleTransactionsFromCloud()
        viewRecordsUseCaseWrapper.deleteMultipleSavedItemCloud()
        settingsUseCaseWrapper.saveMultipleTransactionsCloud()
        settingsUseCaseWrapper.saveMultipleSavedItemCloud()
        budgetUseCaseWrapper.saveMultipleBudgetsToCloudUseCase()
    }
```

12.4.2.2 Exchange Rate Management:

```
return try {
    Log.d(TAG, msg: "Fetching exchange rates for base currency: $baseCurrencyCode")
    val currencyRates = try {
        api.getExchangeRates(baseCurrency = baseCurrencyCode)
    } catch (e: Exception) {
        Log.e(TAG, msg: "Error fetching exchange rates from API: ${e.message}")

        return if (e is java.net.UnknownHostException || e is java.net.ConnectException) {
            Log.d(TAG, msg: "No Internet. Retrying in 30 seconds...")
            Result.retry()  // Retry after backoff delay (30s)
        } else {
            Result.failure()
        }
    }

    val currencyRatesEntity = CurrencyRatesMapper.fromCurrencyResponseToEntity(currencyRates)
    currencyRatesDao.insertCurrencyRates(currencyRatesEntity)
    Log.d(TAG, msg: "Currency rates inserted successfully.")
    userPreferences.setCurrencyRatesUpdated(true)
    Log.d(TAG, msg: "Set Currency rates updated to TRUE successfully.")

    Result.success()
} catch (e: Exception) {
    Log.e(TAG, msg: "Error inserting currency rates: ${e.message}")
    Result.failure()
}
```

```
override suspend fun insertCurrencyRatesLocalOneTime() {
    if (!userPreferences.getCurrencyRatesUpdated()) {
        Log.d( tag: "WorkManagerCurrencies", msg: "Currency rates is not updated One Time Request Started.")
        val workRequest = OneTimeWorkRequestBuilder<PrepopulateCurrencyRatesDatabaseWorker>()
            .setConstraints(
                Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.CONNECTED)
                    .build()
            )
            .setBackoffCriteria(
                BackoffPolicy.EXPONENTIAL,
                backoffDelay: 30, TimeUnit.SECONDS
            )
            .addTag( tag: "PrepopulateCurrencyRates")
            .build()

        workManager.enqueueUniqueWork(
            uniqueWorkName: "PrepopulateCurrencyRates",
            ExistingWorkPolicy.KEEP,
            workRequest
        )

        Log.d( tag: "WorkManagerCurrencyRates", msg: "One-time WorkManager task successfully enqueued.")
    } else {
        Log.d( tag: "WorkManagerCurrencyRates", msg: "Currency rates already updated. Skipping one-time worke
    }
}
```

```
override suspend fun insertCurrencyRatesLocalPeriodically() {
    val workRequest = PeriodicWorkRequestBuilder<PrepopulateCurrencyRatesDatabaseWorker>(
        repeatInterval: 24, TimeUnit.HOURS
    )
        .setInitialDelay(calculateInitialDelay(), TimeUnit.MILLISECONDS) // Start at 6:00 AM
        .setConstraints(
            Constraints.Builder()
                .setRequiredNetworkType(NetworkType.CONNECTED) // Only run if internet is available
                .build()
        )
        .setBackoffCriteria(
            BackoffPolicy.EXPONENTIAL, // Ensures exponential retry delay
            backoffDelay: 10, TimeUnit.MINUTES       // Starts retrying after 10 minutes, doubles each time
        )
        .addTag( tag: "CurrencyUpdateWorker")
        .build()

    workManager.enqueueUniquePeriodicWork(
        uniqueWorkName: "CurrencyUpdateWorker",
        ExistingPeriodicWorkPolicy.CANCEL_AND_REENQUEUE, // Ensures it re-enqueues correctly if rescheduled
        workRequest
    )

    Log.d( tag: "WorkManagerCurrencyRates",  msg: "Scheduled daily currency update at 6:00 AM")
}
```

```
private fun calculateInitialDelay(): Long {
    val now = Calendar.getInstance()
    val targetTime = Calendar.getInstance().apply {
        set(Calendar.HOUR_OF_DAY, 6)
        set(Calendar.MINUTE, 0)
        set(Calendar.SECOND, 0)
    }

    if (now.after(targetTime)) {
        targetTime.add(Calendar.DAY_OF_YEAR,  amount: 1) // Schedule for next day if time has passed
    }

    return targetTime.timeInMillis - now.timeInMillis
}
```

```
class InsertCurrencyRatesLocalOneTime(
    private val currencyRatesLocalRepository: CurrencyRatesLocalRepository
) {
    suspend operator fun invoke(){
        return currencyRatesLocalRepository.insertCurrencyRatesLocalOneTime()
    }
}
```

```
class InsertCurrencyRatesLocalPeriodically(
    private val currencyRatesLocalRepository: CurrencyRatesLocalRepository
) {
    suspend operator fun invoke(){
        return currencyRatesLocalRepository.insertCurrencyRatesLocalPeriodically()
    }
}
```

```
applicationScope.launch {
    predefinedCategoriesUseCaseWrapper.insertPredefinedCategories()
    setupAccountUseCasesWrapper.insertCountryLocallyWorkManager()
    viewRecordsUseCaseWrapper.deleteMultipleTransactionsFromCloud()
    viewRecordsUseCaseWrapper.deleteMultipleSavedItemCloud()
    settingsUseCaseWrapper.saveMultipleTransactionsCloud()
    settingsUseCaseWrapper.saveMultipleSavedItemCloud()
    budgetUseCaseWrapper.saveMultipleBudgetsToCloudUseCase()
}
```

```
fun updateCurrencyRates(){
    viewModelScope.launch(Dispatchers.IO) {
        val userProfile = setupAccountUseCasesWrapper.getUserProfileFromLocalDb(userId)

        if(((!oldBaseCurrency.isNullOrEmpty() && oldBaseCurrency != _profileSetUpStates.value.selectedBaseCu
            setupAccountUseCasesWrapper.setCurrencyRatesUpdated(isUpdated = false)
            setupAccountUseCasesWrapper.insertCurrencyRatesLocalOneTime()
        }

    }
}
```

12.4.2.3 Predefined and Custom Categories:

```
override suspend fun insertPredefinedCategories() {
    val workRequest = OneTimeWorkRequestBuilder<PrepopulateCategoryDatabaseWorker>()
        .build()
    workManager.enqueueUniqueWork(
        uniqueWorkName: "prepopulate_db",
        ExistingWorkPolicy.KEEP,
        workRequest
    )
}
```

```
override suspend fun insertCategories(categories: List<Category>) {
    return categoryDao.insertCategories(
        categories = categories.map {
            it.toEntity()
        }
    )
}

override suspend fun insertCategory(category: Category) {
    return categoryDao.insertCategory(
        category = category.toEntity()
    )
}
```

```
class InsertPredefinedCategories(
    private val categoryRepository: CategoryRepository
) {
    suspend operator fun invoke() {
        return categoryRepository.insertPredefinedCategories()
    }
}
```

```
class InsertCustomCategories(
    private val categoryRepository: CategoryRepository
) {
    suspend operator fun invoke(category: Category) {
        return categoryRepository.insertCategory(category)
    }
}
```

12.4.2.4 Transaction Sync Workflow:

```
override suspend fun doWork(): Result {
    Log.d( tag: "WorkManagerUploadTransactions",  msg: "Worker started Upload All Transactions To Cloud")

    val userId = userPreferences.getUserIdLocally() ?: return Result.failure()

    return try {
        val allLocalTransactions = addTransactionUseCasesWrapper.getAllLocalTransactions(uid = userId).firs

        Log.d( tag: "WorkManagerUploadTransactions",  msg: "userId $userId ")
        Log.d( tag: "WorkManagerUploadTransactions",  msg: "allLocalTransactions $allLocalTransactions")



        val cloudSync = userPreferences.getCloudSync()

        if (allLocalTransactions.isEmpty() || !cloudSync) {
            Log.d( tag: "WorkManagerUploadTransactions",  msg: "No transactions to sync.")
            Log.d( tag: "WorkManagerUploadTransactions",  msg: "cloudSync: $cloudSync")
            return Result.failure()

        }
        else{

            allLocalTransactions.forEach { transaction ->
                val transactionId = transaction.transactionId
                val transactionWithId = transaction.copy(transactionId = transactionId, cloudSync = true)

                addTransactionUseCasesWrapper.saveSingleTransactionCloud(userId = userId, transaction)
                addTransactionUseCasesWrapper.insertTransactionsLocally(transactionWithId)
            }
            Log.d( tag: "WorkManagerUploadTransactions",  msg: "All local transactions inserted to cloud successful
            Result.success()
        }
    } catch (e: Exception) {
        Log.e( tag: "WorkManagerUploadTransactions",  msg: "Error during sync: ${e.message}")
        e.printStackTrace()
        Result.retry()
    }
}
```

```
    applicationScope.launch {
        predefinedCategoriesUseCaseWrapper.insertPredefinedCategories()
        setupAccountUseCasesWrapper.insertCountryLocallyWorkManager()
        viewRecordsUseCaseWrapper.deleteMultipleTransactionsFromCloud()
        viewRecordsUseCaseWrapper.deleteMultipleSavedItemCloud()
        settingsUseCaseWrapper.saveMultipleTransactionsCloud()
        settingsUseCaseWrapper.saveMultipleSavedItemCloud()
        budgetUseCaseWrapper.saveMultipleBudgetsToCloudUseCase()
    }
}
```

12.4.2.5 Saved Items Synchronization:

```
override suspend fun doWork(): Result {

    return try {
        val allLocalSavedItems = savedItemsUseCasesWrapper.getAllNotSyncedSavedItemUseCase(userUID = userId).firs

        Log.d( tag: "WorkManagerUploadSavedItems",  msg: "userId $userId ")
        Log.d( tag: "WorkManagerUploadSavedItems",  msg: "allLocalSavedItems $allLocalSavedItems")



        val cloudSync = userPreferences.getCloudSync()

        if (allLocalSavedItems.isEmpty() || !cloudSync) {
            Log.d( tag: "WorkManagerUploadSavedItems",  msg: "No savedItems to sync.")
            Log.d( tag: "WorkManagerUploadSavedItems",  msg: "cloudSync: $cloudSync")
            return Result.failure()

        }
        else{

            allLocalSavedItems.forEach { savedItems ->
                val itemId = savedItems.itemId
                val savedItemWithId = savedItems.copy(itemId = itemId, cloudSync = true)

                savedItemsUseCasesWrapper.saveSingleSavedItemCloud(userId = userId,savedItems = savedItemWithId)
                savedItemsUseCasesWrapper.saveItemLocalUseCase(savedItemWithId)

            }
            Log.d( tag: "WorkManagerUploadSavedItems",  msg: "All local saved items inserted to cloud successfully.
            Result.success()
        }
```

```
    applicationScope.launch {
        predefinedCategoriesUseCaseWrapper.insertPredefinedCategories()
        setupAccountUseCasesWrapper.insertCountryLocallyWorkManager()
        viewRecordsUseCaseWrapper.deleteMultipleTransactionsFromCloud()
        viewRecordsUseCaseWrapper.deleteMultipleSavedItemCloud()
        settingsUseCaseWrapper.saveMultipleTransactionsCloud()
        settingsUseCaseWrapper.saveMultipleSavedItemCloud()
        budgetUseCaseWrapper.saveMultipleBudgetsToCloudUseCase()
    }
}
```

12.4.2.6 Transaction Deletion Handling:

```
class DeleteAllTransactionsFromCloudDatabaseWorker @AssistedInject constructor(
    override suspend fun doWork(): Result {

        val userId = userPreferences.getUserIdLocally() ?: return Result.failure()

        return try {

            Log.d( tag: "WorkManagerDeletedTransactions", msg: "UserId: $userId")
            val allDeletedTransactions = viewRecordsUseCaseWrapper.getAllDeletedTransactionByUserId(userId = userId)

            Log.d( tag: "WorkManagerDeletedTransactions", msg: "deletedTransactions $allDeletedTransactions")


            if (allDeletedTransactions.isEmpty() ) {
                Log.d( tag: "WorkManagerDeletedTransactions",  msg: "No deleted transactions to sync.")
                return Result.success()

            }
            else{
                allDeletedTransactions.forEach { deletedTransaction ->
                    viewRecordsUseCaseWrapper.deleteTransactionCloud(userId = userId, transactionId = deletedTransac
                    viewRecordsUseCaseWrapper.deleteDeletedTransactionsByIdsFromLocal(transactionId = deletedTransac
                }
                Log.d( tag: "WorkManagerDeletedTransactions",  msg: "All Cloud transactions deleted from cloud success


                Result.success()
            }
        } catch (e: Exception) {
            Log.e( tag: "WorkManagerDeletedTransactions",  msg: "Error during sync: ${e.message}")
            e.printStackTrace()
```

```
    applicationScope.launch {
        predefinedCategoriesUseCaseWrapper.insertPredefinedCategories()
        setupAccountUseCasesWrapper.insertCountryLocallyWorkManager()
        viewRecordsUseCaseWrapper.deleteMultipleTransactionsFromCloud()
        viewRecordsUseCaseWrapper.deleteMultipleSavedItemCloud()
        settingsUseCaseWrapper.saveMultipleTransactionsCloud()
        settingsUseCaseWrapper.saveMultipleSavedItemCloud()
        budgetUseCaseWrapper.saveMultipleBudgetsToCloudUseCase()
    }
}
```

12.4.2.7 Saved Item Deletion Sync:

```
erride suspend fun doWork(): Result {
  Log.d( tag: "WorkManagerDeletedSavedItems",  msg: "Worker started Deleted All SavedItems From Cloud")

  val userId = userPreferences.getUserIdLocally() ?: return Result.failure()

  return try {

      Log.d( tag: "WorkManagerDeletedSavedItems", msg: "UserId: $userId")
      val allDeletedSavedItems = viewRecordsUseCaseWrapper.getAllDeletedSavedItemsByUserId(userId = userId).first()

      Log.d( tag: "WorkManagerDeletedSavedItems", msg: "deletedSavedItems $allDeletedSavedItems")


      if (allDeletedSavedItems.isEmpty() ) {
          Log.d( tag: "WorkManagerDeletedSavedItems",  msg: "No deleted saved items to sync.")
          return Result.success()

      }
      else{
          allDeletedSavedItems.forEach { deletedSavedItems ->
              viewRecordsUseCaseWrapper.deleteSavedItemCloud(userId = userId, itemId = deletedSavedItems.itemId ?:
              viewRecordsUseCaseWrapper.deleteDeletedSavedItemsById(itemId = deletedSavedItems.itemId ?: 0)
          }
          Log.d( tag: "WorkManagerDeletedSavedItems",  msg: "All Cloud saved items deleted from cloud successfully.")


          Result.success()
      }
  } catch (e: Exception) {
```

```
class DeleteMultipleSavedItemCloud(
    private val savedItemsRemoteRepository: SavedItemsRemoteRepository,
) {

    suspend operator fun invoke(){
        savedItemsRemoteRepository.deleteMultipleSavedItemsFromCloud()
    }
}
```

12.4.2.8 Budget Management:

```
    val userId = budgetUseCaseWrapper.getUIDLocally() ?: return Result.failure()

    return try {
        val allLocalBudgets = budgetUseCaseWrapper.getAllUnSyncedBudgetLocalUseCase(userId = userId).first()

        Log.d( tag: "WorkManagerUploadBudgets",  msg: "userId $userId ")
        Log.d( tag: "WorkManagerUploadBudgets",  msg: "allLocalTransactions $allLocalBudgets")



        val cloudSync = userPreferences.getCloudSync()

        if (allLocalBudgets.isEmpty() || !cloudSync) {
            Log.d( tag: "WorkManagerUploadBudgets",  msg: "No budgets to sync.")
            Log.d( tag: "WorkManagerUploadBudgets",  msg: "cloudSync: $cloudSync")
            return Result.failure()

        }
        else{

            allLocalBudgets.forEach { budget ->
                val budgetId = budget.id
                val budgetWithId = budget.copy(id = budgetId, cloudSync = true)

                budgetUseCaseWrapper.saveBudgetToCloudUseCase(userId = userId, budget = budgetWithId)
                budgetUseCaseWrapper.insertBudgetLocalUseCase(budget = budgetWithId)
            }
            Log.d( tag: "WorkManagerUploadBudgets",  msg: "All local budgets inserted to cloud successfully.")
            Result.success()
```

```
class SaveMultipleBudgetsToCloudUseCase(
    private val  budgetRemoteRepository: BudgetRemoteRepository
) {
    suspend operator fun invoke(){
        return budgetRemoteRepository.uploadMultipleBudgetsToCloud()
    }
}
```

12.4.3 Cloud Integration Core Features

12.4.3.1 Firebase Authentication Google / Register and Login

```
suspend fun resetPasswordWithCredential() : ResetPasswordWithCredentialResult {
    return try {
        val credentialResponse = credentialManager.getCredential(
            context = activity,
            request = GetCredentialRequest(
                credentialOptions = listOf(GetPasswordOption())
            )
        )

        val credential = credentialResponse.credential as? PasswordCredential
        val authResult = firebaseAuth.signInWithEmailAndPassword(credential!!.id, credential.password).await(
        return ResetPasswordWithCredentialResult.CredentialLoginSuccess( email: authResult.user?.email ?: "Unkn
    }catch (e:CreateCredentialCancellationException){
        e.printStackTrace()
        ResetPasswordWithCredentialResult.Cancelled
    } catch (e:CreateCredentialException){
        e.printStackTrace()
        ResetPasswordWithCredentialResult.CredentialLoginFailure
    } catch (e:Exception){
        e.printStackTrace()
        ResetPasswordWithCredentialResult.UnknownFailure
    }
}
```

```
suspend fun resetPasswordWithEmail(email: String) : ResetPasswordWithEmailResult {
    return try {
        firebaseAuth.sendPasswordResetEmail(email).await()
        ResetPasswordWithEmailResult.EmailSuccess
    } catch (e:FirebaseAuthException){
        e.printStackTrace()
        ResetPasswordWithEmailResult.AuthFailure
    } catch (e:CancellationException){
        e.printStackTrace()
        ResetPasswordWithEmailResult.Cancelled
    } catch (e:Exception){
        e.printStackTrace()
        ResetPasswordWithEmailResult.UnknownFailure
    }
}
```

12.4.3.2 Firestore Database for sync

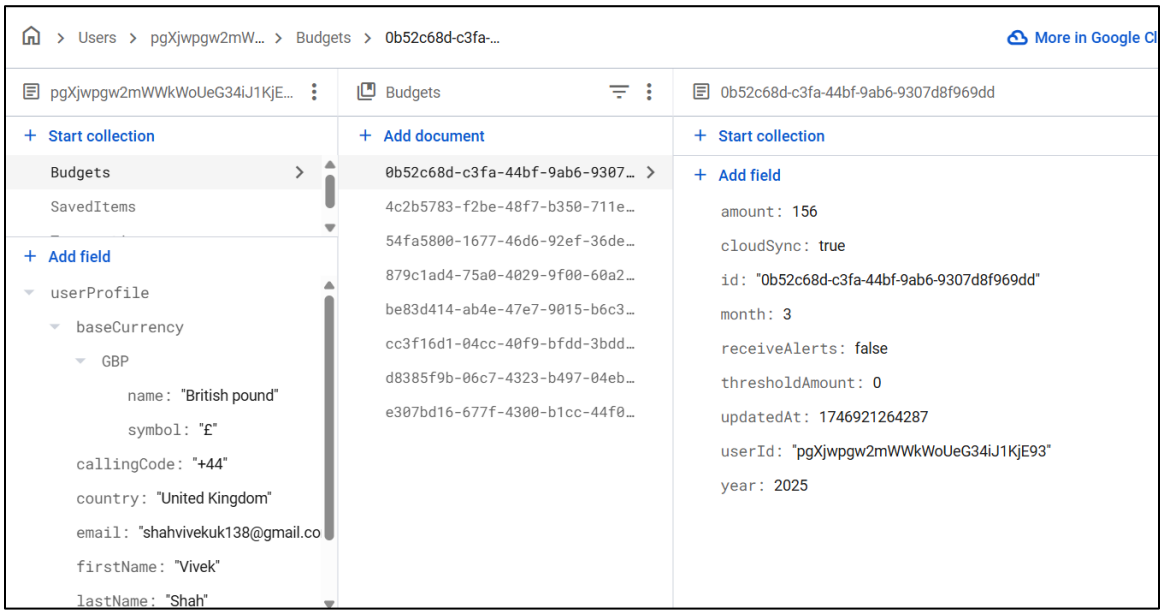
```
override suspend fun saveUserProfile(userId: String, profile: UserProfile) {
    try {
        // Force a Firestore network call
        firestore.collection( collectionPath: "Users").document(userId)
            .get(Source.SERVER) // Ensures Firebase tries fetching from the server
            .await()

        // Now perform the write operation
        firestore.collection( collectionPath: "Users").document(userId)
            .set(mapOf("userProfile" to profile), SetOptions.merge())
            .await()

    } catch (e: Exception) {
        Log.d( tag: "RemoteRepository", msg: "save user error ${e.localizedMessage}")
        Log.d( tag: "RemoteRepository", msg: "save user print stack ${e.printStackTrace()}")
        throw Exception("No internet connection. Profile update failed.")
    }
}
```

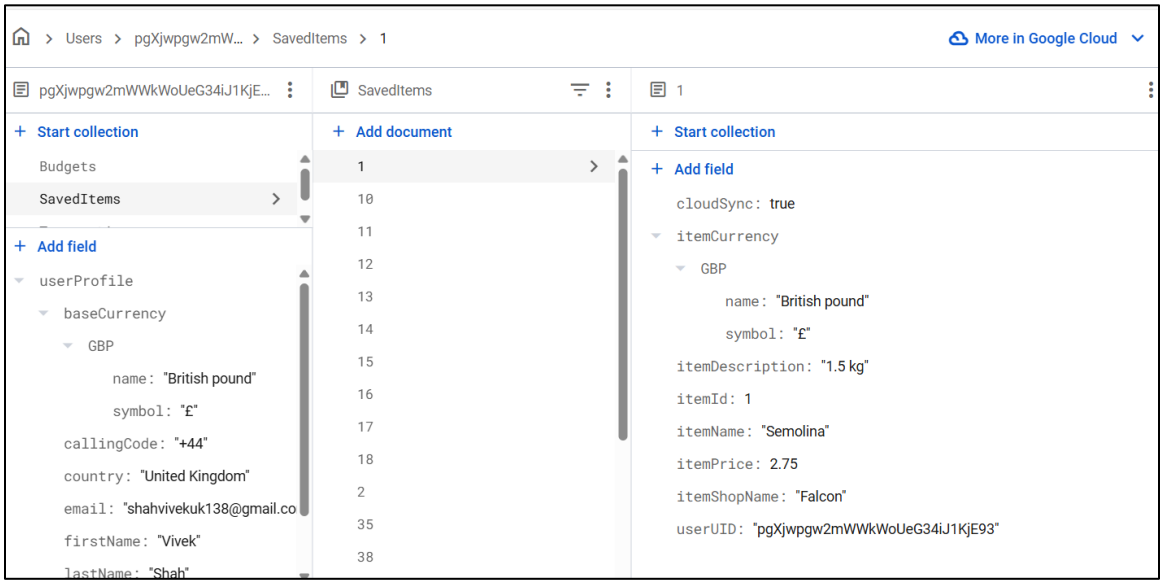
```
override suspend fun cloudSyncSingleTransaction(
    userId: String,
    transactions: Transactions,
    updateCloudSync: suspend (Int, Boolean) -> Unit
) {
    try {

        val transactionId = transactions.transactionId?.toString() ?: firestore.collection( colls
            .document(userId)
            .collection( collectionPath: "Transactions")
            .document().id

        firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "Transactions")
            .document(transactionId)
            .set(transactions.copy(cloudSync = true)) // Save with cloudSync = true
            .await()

        // Update local Room DB
        transactions.transactionId?.let { updateCloudSync(it, true) }

    }catch (e: Exception){

        // If failed, make sure cloudSync remains false
        transactions.transactionId?.let { updateCloudSync(it, false) }
    }
}
```

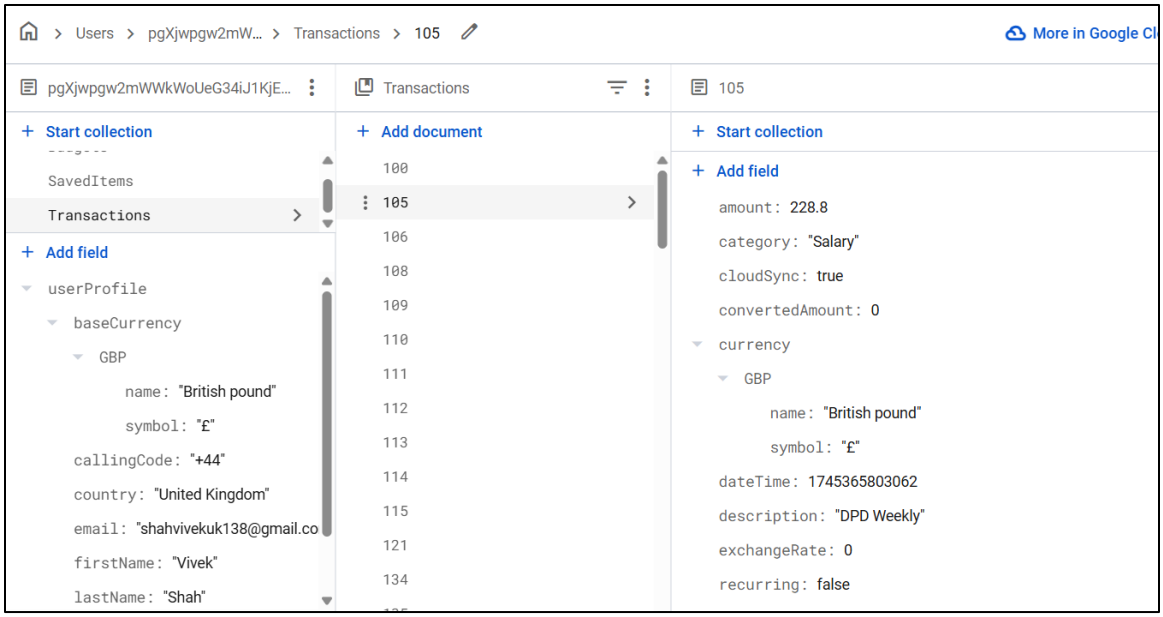
```
override suspend fun cloudSyncSingleSavedItem(
    userId: String,
    savedItems: SavedItems,
    updateCloudSync: suspend (Int, Boolean) -> Unit
) {
    try {

        val itemId = savedItems.itemId?.toString() ?: firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "SavedItems")
            .document().id

        firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "SavedItems")
            .document(itemId)
            .set(savedItems.copy(cloudSync = true)) // Save with cloudSync = true
            .await()

        // Update local Room DB
        savedItems.itemId?.let { updateCloudSync(it, true) }

    }catch (e: Exception){

        // If failed, make sure cloudSync remains false
        savedItems.itemId?.let { updateCloudSync(it, false) }
    }
}
```

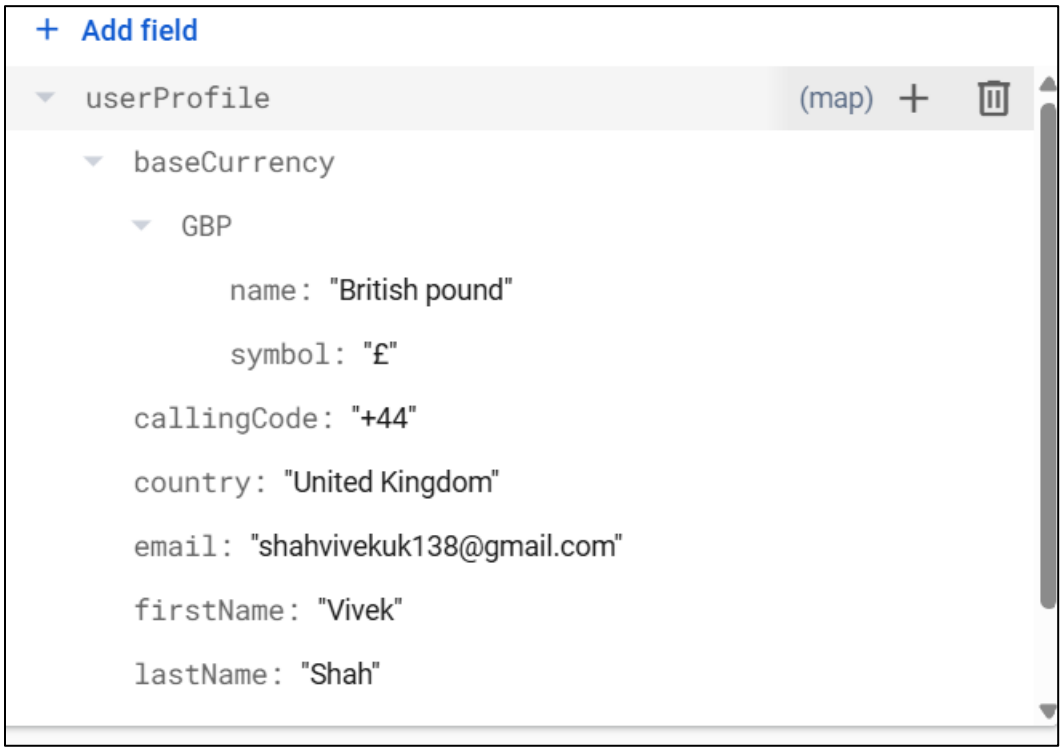
```
class BudgetRemoteRepositoryImpl(
    private val firestore: FirebaseFirestore,
    private val context: Context,
): BudgetRemoteRepository {
    override suspend fun uploadSingleBudgetToCloud(userId: String, budget: Budget) {

        val budgetId = budget.id ?: firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "Budgets")
            .document().id

        firestore.collection( collectionPath: "Users")
            .document(userId)
            .collection( collectionPath: "Budgets")
            .document(budgetId)
            .set(budget.copy(cloudSync = true)) // Save with cloudSync = true
            .await()
```

12.4.3.3 Explain structure of cloud data.





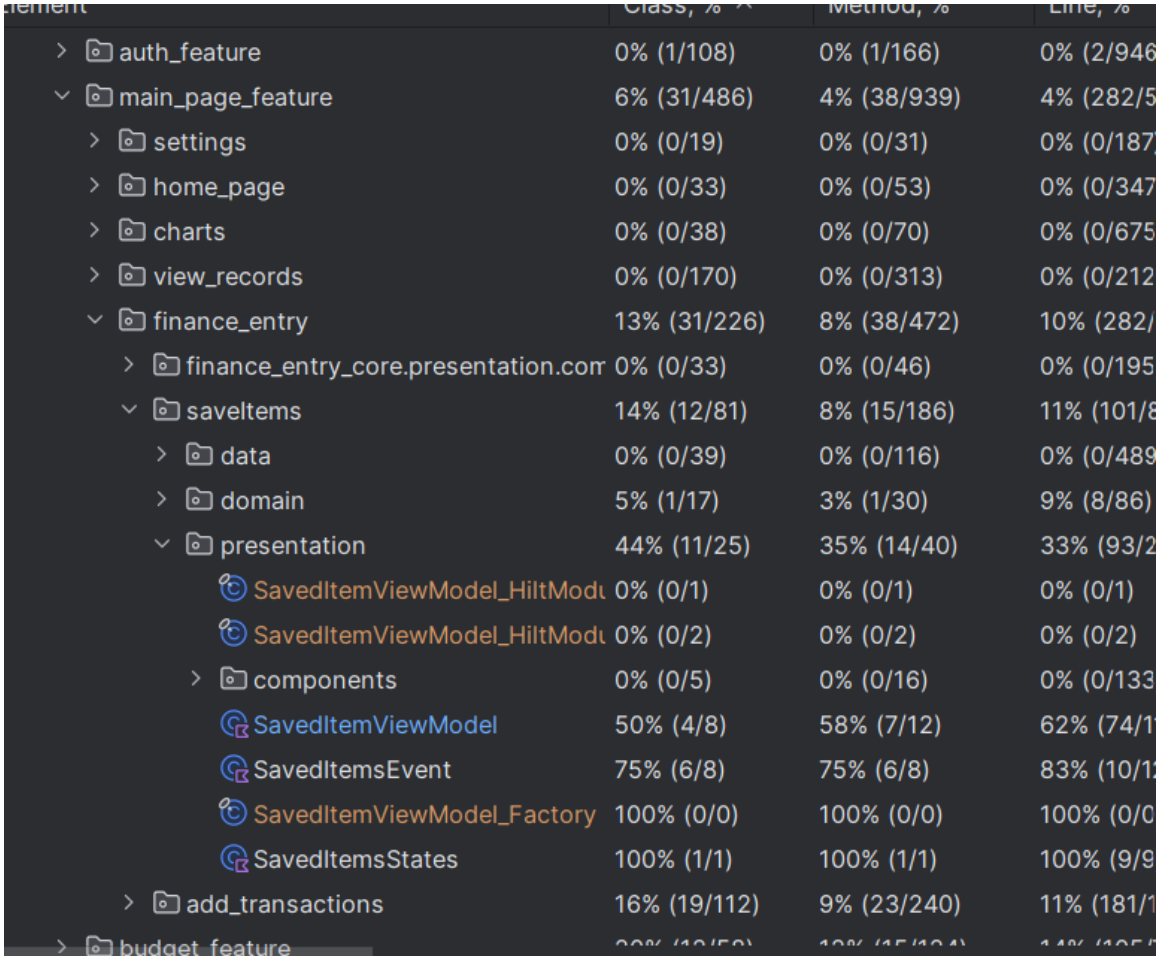

Users > pgXjwpgw2mW... > Transactions > 105
More in Google Cl
pgXjwpgw2mWWkWoUeG34iJ1KjE...
Transactions
105
+ Start collection
SavedItems
Transactions
+ Add field
userProfile
baseCurrency
GBP
name: "British pound"
symbol: "£"
callingCode: "+44"
country: "United Kingdom"
email: "shahvivekuk138@gmail.co
firstName: "Vivek"
lastName: "Shah"
+ Add document
100
105
106
108
109
110
111
112
113
114
115
121
134
+ Start collection
+ Add field
amount: 228.8
category: "Salary"
cloudSync: true
convertedAmount: 0
currency
GBP
name: "British pound"
symbol: "£"
dateTime: 1745365803062
description: "DPD Weekly"
exchangeRate: 0
recurring: false


+ Add field
userProfile
(map)
baseCurrency
GBP
name: "British pound"
symbol: "£"
callingCode: "+44"
country: "United Kingdom"
email: "shahvivekuk138@gmail.com"
firstName: "Vivek"
lastName: "Shah"

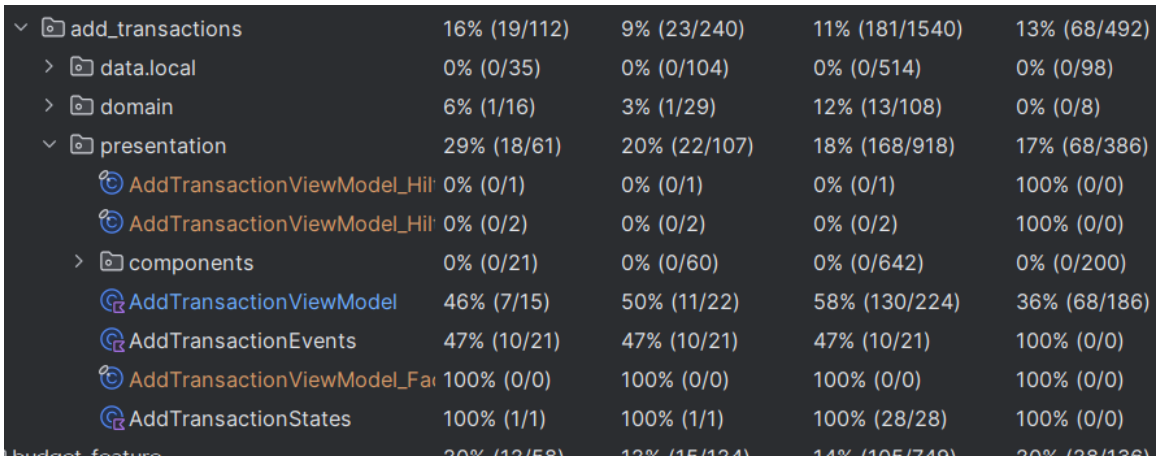
12.4.4 API Integration

```
object ApiClient {
    const val BASE_URL_COUNTRY = "https://restcountries.com/"
    const val BASE_URL_CURRENCY_RATES = "https://v6.exchangerate-api.com/v6/"
    const val API_KEY = "21f5e9959aaff358c6ed28bc"
}
```

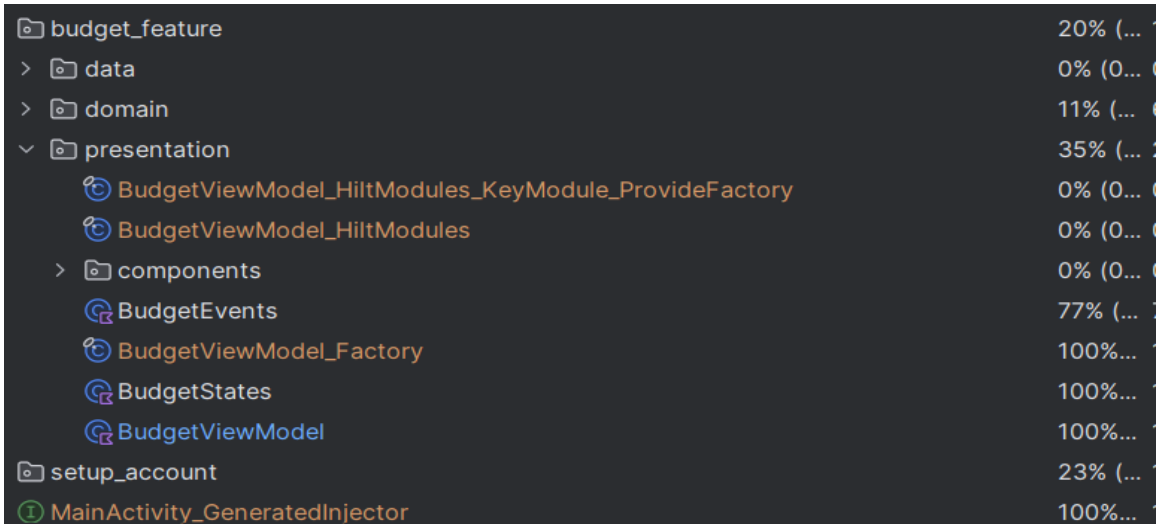
12.5 Test Coverage



Element	Class, %	Method, %	Line, %
auth_feature	0% (1/108)	0% (1/166)	0% (2/946
main_page_feature	6% (31/486)	4% (38/939)	4% (282/5
settings	0% (0/19)	0% (0/31)	0% (0/187
home_page	0% (0/33)	0% (0/53)	0% (0/347
charts	0% (0/38)	0% (0/70)	0% (0/675
view_records	0% (0/170)	0% (0/313)	0% (0/212
finance_entry	13% (31/226)	8% (38/472)	10% (282/
finance_entry_core.presentation.com	0% (0/33)	0% (0/46)	0% (0/195
saveItems	14% (12/81)	8% (15/186)	11% (101/8
data	0% (0/39)	0% (0/116)	0% (0/489
domain	5% (1/17)	3% (1/30)	9% (8/86)
presentation	44% (11/25)	35% (14/40)	33% (93/2
SavedItemViewModel_HiltModu	0% (0/1)	0% (0/1)	0% (0/1)
SavedItemViewModel_HiltModu	0% (0/2)	0% (0/2)	0% (0/2)
components	0% (0/5)	0% (0/16)	0% (0/133
SavedItemViewModel	50% (4/8)	58% (7/12)	62% (74/1
SavedItemsEvent	75% (6/8)	75% (6/8)	83% (10/1
SavedItemViewModel_Factory	100% (0/0)	100% (0/0)	100% (0/0
SavedItemsStates	100% (1/1)	100% (1/1)	100% (9/9
add_transactions	16% (19/112)	9% (23/240)	11% (181/1
budget_feature			



Element	Class, %	Method, %	Line, %
add_transactions	16% (19/112)	9% (23/240)	11% (181/1540)	13% (68/492)
data.local	0% (0/35)	0% (0/104)	0% (0/514)	0% (0/98)
domain	6% (1/16)	3% (1/29)	12% (13/108)	0% (0/8)
presentation	29% (18/61)	20% (22/107)	18% (168/918)	17% (68/386)
AddTransactionViewModel_Hil	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
AddTransactionViewModel_Hil	0% (0/2)	0% (0/2)	0% (0/2)	100% (0/0)
components	0% (0/21)	0% (0/60)	0% (0/642)	0% (0/200)
AddTransactionViewModel	46% (7/15)	50% (11/22)	58% (130/224)	36% (68/186)
AddTransactionEvents	47% (10/21)	47% (10/21)	47% (10/21)	100% (0/0)
AddTransactionViewModel_Fa	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
AddTransactionStates	100% (1/1)	100% (1/1)	100% (28/28)	100% (0/0)
budget_feature				


budget_feature 20% (...
data 0% (0...
domain 11% (...
presentation 35% (...
BudgetViewModel_HiltModules_KeyModule_ProvideFactory 0% (0...
BudgetViewModel_HiltModules 0% (0...
components 0% (0...
BudgetEvents 77% (...
BudgetViewModel_Factory 100%...
BudgetStates 100%...
BudgetViewModel 100%...
setup_account 23% (...
MainActivity_GeneratedInjector 100%...