PROJECT REPORT

for

WALLET WATCH

Github repository(Click here)

Prepared by:

1. Kavya Gupta (IMT2023016)
2. Nainika Agrawal (IMT2023034)

December 7, 2025

Contents

1 Features and Functionalities

The app provides the following functionalities:

- **Login/ Register** – Authenticate the user via Firebase Authentication.
- **Forgot Password** – Reset user password via Firebase Authentication.
- **View Dashboard** – Displays an overview of the user's financial data.
- **Add/ Edit Transaction** – Record a new transaction or modify an existing one.
- **Delete Transaction** – Remove an existing transaction.
- **Filter / Search Transactions** – Search and filter transaction history.
- **Manage Insurance** – Add or update insurance details.
- **Manage Loans** – Add or update loan information.
- **Manage Subscriptions** – Add or modify subscriptions.
- **Create Budget** – Define a new budget
- **View Budget Progress** – Track budget utilization and progress.
- **Create Goal** – Set a new financial goal.
- **Delete Goal** – Remove a financial goal
- **Real-time Currency Conversion** – Accesses ExchangeRate API to provide live currency conversion and dynamically updates exchange rates.
- **View FAQs** – Shows frequently asked questions.
- **View Financial Tips** – Provides financial advice to the user.
- **Manage Profile** – Update user preferences and profile details.
- **Submit Feedback** – Send feedback to the application.

External Systems:

- **ExchangeRate API** – Provides live currency conversion and updates exchange rates dynamically.
- **Firestore Database** – Stores all user-specific data: transactions, budgets, goals, loans, insurance, subscriptions, and feedback.
- **Firebase Authentication** – Handles user sign-up, login, and password reset.

2 Running the App

2.1 Commands to Run the App

1. Please make sure that you have the following installed:
   - Flutter: https://docs.flutter.dev/get-started
   - Firebase Account: https://firebase.google.com/
   - A code editor like Visual Studio Code (https://code.visualstudio.com/) or Android Studio (https://developer.android.com/studio)
2. Clone the repository :

   git clone https://github.com/nainika0305/Wallet_Watch.git
3. Install dependencies

   flutter pub get
4. Configure Firebase
   - Set up a Firebase project from the Firebase Console(https://console.firebase.google.com/u/0/mfa)
   - Follow the steps to enable Firebase Authentication (email/password).
   - Configure Firebase Firestore to store transactions, budgets, and feedback.
   - Download and add the google-services.json (for Android) or GoogleService-Info.plist (for iOS) to your project.
5. Run the app - Ensure you have a device or emulator running, then execute the following command to run the app: flutter run
6. Additional Configuration
   - For Android, make sure to add the google-services.json file to the android/app directory.
   - For iOS, add the GoogleService-Info.plist file to the ios/Runner directory and ensure your app's deployment target is set to at least iOS 10.
7. Troubleshooting
   - If you encounter issues with Firebase, check your Firebase setup and ensure all configurations are correctly followed.
   - For issues related to dependencies, try running flutter clean and then flutter pub get again.

2.2 Operating Environment

Table 2.1: Operating Environment

Component	Specification
System Type	Client-Server (Mobile Client with Backend-as-a-Service)
Platform	Flutter SDK (cross-platform mobile framework)
Client Operating Systems	Android (API 21: Lollipop 5.0 and above) iOS (10.0 and above)
Backend Services	Firebase (BaaS)
Authentication	Firebase Authentication (Email/Password)
Database	Cloud Firestore (NoSQL distributed database)
Programming Language	Dart (Frontend), Firebase Security Rules (Backend)
State Management	Provider
Key Libraries	`fl_chart` (visualization), `flutter_colorpicker`, `intl` (formatting), `http` (API requests)
API Integrations	ExchangeRate API (for real-time currency conversion)
Development Tools	Android Studio / Visual Studio Code
Testing	`flutter_test` (Widget Testing)
Version Control	Git / GitHub

2.3 Design and Implementation Constraints

The design and implementation of the Wallet Watch application are guided by the following primary constraints:

- **Platform:** The application **must** be built using the Flutter framework and Dart language.
- **State Management:** The application **must** use the `Provider` package for state management, including for features like Dark Mode.
- **Backend:** The system **must** be a client-server application using Google Firebase as its Backend-as-a-Service (BaaS) for all authentication and database needs.
- **Database Model:** The system **must** use the Cloud Firestore (NoSQL) database. All data access is constrained to Firebase SDK methods (e.g., `.snapshots()`, `.add()`), not SQL.
- **Data Schema Constraint:** The system has an inconsistent fragmentation schema: `transactions` and `loans` are stored under the user's **email**, while `budgets` and `goals` are stored under the user's **Firebase UID**.
- **Business Logic Constraint:** All data aggregation and complex logic (like loan interest calculation, budget progress, and subscription due dates) **must** be performed on the **client-side** within the Dart code.
- **Network & Security:** The application requires a persistent internet connection for all real-time data synchronization and **must** be secured using Firebase Security Rules to ensure users only access their own data.

2.4 Assumptions and Dependencies

Assumptions: The user has a compatible smartphone (Android or iOS), stable internet connection, a valid email address to complete the registration process. All financial data, such as transactions and budgets, is assumed to be entered manually by the user, as the system does not auto-fetch from banks. The user is assumed to have basic knowledge of mobile app navigation (tapping buttons, filling forms, etc.). Application performance and response time are dependent on the user's device specifications and network strength.
Dependencies: The system's core functionality (authentication, data storage) is dependent on the availability and terms of service of Google Firebase. The currency conversion feature is dependent on the availability and accuracy of the third-party ExchangeRate API.

3 Project Phases

While developing the project, we adopted the Agile methodology as it allowed us to work in an iterative and incremental manner. This approach helped us ensure continuous improvement, maintain flexibility when requirements evolved, and incorporate new ideas and refinements as the project progressed. Each sprint focused on delivering a functional subset of the system, beginning with planning and analysis, followed by design, development, testing, and review activities. To avoid repetitive detailing, the list below highlights only the key features and milestones completed during each sprint, as the requirement gathering, design, implementation, and testing tasks were consistently performed across all iterations.

- **Sprint 0: Requirements Analysis, Setup & Planning**
  - Backlog Creation & Prioritization – Initial requirements were gathered, refined, and structured.
  - Environment Setup (Flutter + Firebase) – The development environment and tools were configured, ensuring the technology stack was ready for implementation.
- **Sprint 1: Core Foundations**
  - User Authentication – The basic authentication mechanism was implemented to support secure login and account access.
  - Basic UI & Navigation – Core screens and navigation flow were developed to provide the initial structure of the application.
  - Daily Standups & Review – Regular progress assessments and sprint evaluation helped align development efforts and address challenges early.
- **Sprint 2: Core Features**
  - Transaction Management Module – Functionality for recording, storing, and categorizing financial transactions was developed.
  - Budget Management Module – Budget tracking features were added to allow users to plan and monitor spending effectively.
  - Sprint Review & Retrospective 2 – Completed features were reviewed and improvements were planned for future sprints.
- **Sprint 3: Extended Features**
  - Dashboard Development – A visual dashboard was built to present key financial data in a simplified and user-friendly manner.
  - Loans, Subscriptions, and Insurance Features – Additional modules were integrated to enhance financial tracking and extend overall functionality.
  - Sprint Review & Retrospective 3 – The sprint outcomes were evaluated, and necessary enhancements were identified.
- **Sprint 4: Final Testing & Release**
  - Integration Testing & Bug Fixing – Comprehensive testing was performed to ensure all system components worked cohesively without issues.

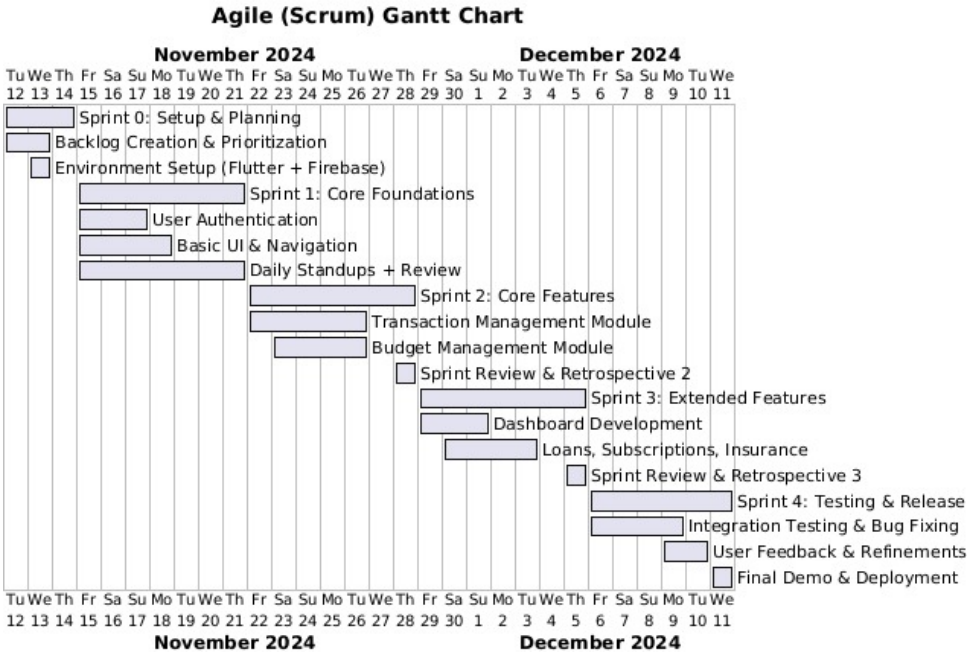

Figure 3.1: Gantt_Agile

- **Project Selection**
  - Although the core application had already been developed before the course began, we selected this project for our Software Engineering Lab.
  - The objective was not just to build a functional prototype, but to formally analyze, document, and evaluate its development process from a structured software engineering perspective.
- **Documentation and Engineering Activities**
  - A detailed Software Requirements Specification (SRS) was prepared to clearly define the scope, functional requirements, and constraints of the system.
  - Comprehensive design documentation was created, including UML diagrams, non UML diagrams architectural diagrams.
  - Architectural styles used in the system were identified and justified along with key design decisions that influenced modularity, scalability, and usability.
  - Testing scripts were prepared and executed to validate functionality, ensuring the system satisfied both functional and non-functional requirements.
- **Course Phases and Activities**
  - Requirement Analysis and preparation of the SRS.
  - Identification of architectural patterns and design principles.
  - Design Documentation
  - Testing phase involving preparation of test cases and execution of functional and integration tests.

4 Testing and Implementation Guide

4.1 Testing Strategy

To ensure the reliability and robustness of the *Wallet Watch* application, we adopted a multi-layered testing strategy often referred to as the **Testing Testing Pyramid**. This approach divides testing into two distinct categories to maximize code coverage and execution speed while minimizing dependencies on external networks.

The testing framework utilized includes:

- **Framework:** `flutter_test` (Standard Flutter testing SDK).
- **Database Simulation:** `fake_cloud_firestore` (To simulate a NoSQL database in memory).
- **Auth Simulation:** `firebase_auth_mocks` (To simulate user sessions without hitting Google servers).

4.1.1 Phase 1: Unit Testing (Logic & Validation)

The foundation of our testing suite consists of Unit Tests located in `test/services/`. These tests isolate individual service classes (e.g., `TransactionService`, `LoanService`) to verify business logic without testing the entire system.

Key aspects validated during this phase:

1. **Input Validation:** Ensuring the system rejects invalid inputs, such as negative amounts for transactions or empty strings for categories.
2. **Data Integrity:** Verifying that floating-point calculations (e.g., currency conversion, tax estimation) maintain precision.
3. **Destructive Testing:** Deliberately attempting to break the system (e.g., overpaying a loan) to ensure proper exception handling.

```
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/auth_service_test.dart
00:02 +5: All tests passed!
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/budget_service_test.dart
00:04 +8: All tests passed!
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/financial_logic_test.dart
00:01 +10: All tests passed!
```

```
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/goal_service_test.dart
00:02 +6: All tests passed!
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/loan_service_test.dart
00:02 +8: All tests passed!
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/transaction_service_test.dart
00:02 +7: All tests passed!
PS C:\Users\Nainika\Desktop\money\Wallet_Watch> flutter test test/services/user_service_test.dart
00:02 +7: All tests passed!
```

4.1.2 Phase 2: System Integration Testing (User Journeys)

To verify that the modules function correctly as a cohesive application, we implemented a **System Integration Test** in `test/backend_test.dart`.

This test simulates a complete "User Journey" spanning the entire application lifecycle:

- **Profile Creation:** User registers and receives an initial balance.
- **Budgeting:** User sets a monthly spending limit.
- **Spending:** User records an expense.
- **Liability Management:** User takes a loan and performs a partial repayment.

4.2 Test Coverage & Traceability

Our testing suite was designed to ensure direct traceability to the Software Requirements Specification (SRS). Below is a summary of how critical Functional Requirements (FRs) are covered by automated tests.

Module	SRS Reference	Test File
User Authentication	FR1.1, FR1.2, FR1.3	`auth_service_test.dart`
Transaction Mgmt	FR2.1, FR2.2, NFR2.2	`transaction_service_test.dart`
Budgeting	FR3.1, FR3.3	`budget_service_test.dart`
Goals	FR5.0 (Implied)	`goal_service_test.dart`
Loans & Liabilities	Section 6 (Liabilities)	`loan_service_test.dart`
Financial Logic	FR5.7, FR (Tax Tool)	`financial_logic_test.dart`

Table 4.1: Traceability Matrix: Requirements to Tests

4.3 Execution Instructions

The following instructions detail how to set up the environment, run the automated test suite, and launch the application.

4.3.1 Prerequisites

Ensure the following are installed on the host machine:

- Flutter SDK (Version 3.0 or higher)

- Dart SDK
- An Android Emulator or iOS Simulator

4.3.2 Step 1: Installation

Clone the repository and install the required dependencies.

```
# 1. Navigate to the project directory
cd Wallet_Watch

# 2. Install dependencies
flutter pub get
```

4.3.3 Step 2: Running the Tests

We have configured the test runner to display detailed console output, allowing the examiner to visualize the "User Journey" steps in real-time.

To run the Complete System Integration Test:

```
flutter test test/backend_test.dart -r expanded
```

To run All Unit Tests (The Full Suite):

```
flutter test
```

4.4 Conclusion on Quality Assurance

By utilizing `fake_cloud_firestore`, we achieved a testing environment that is **deterministic**, **offline-capable**, and **fast**. The combination of granular Unit Tests for logic verification and a comprehensive System Integration Test for workflow validation ensures that *Wallet Watch* meets the high reliability standards required for financial software.

5 Extended Description

5.1 Product Perspective

The Wallet Watch application is a mobile-based personal finance management system designed to help users efficiently track their expenses, savings, and overall budget goals. It integrates multiple financial functionalities within a single distributed environment powered by Firebase backend services for real-time data synchronization and storage.

The system stores the following information:

- **Transaction Details:**
  Records each expense or income entry, including date, amount, category (e.g., groceries, travel, salary), mode of payment, and notes.
- **Budget Details:**
  Contains budget name, type (expense/savings), amount, timeline, color tag, and category. Tracks spending vs. remaining amount dynamically.
- **User Information:**
  Includes first name, last name, email, currency preference, and authentication details. Stored securely using Firebase Authentication and Firestore.
- **Subscription, Loan, and Insurance Data:**
  Keeps detailed records of recurring payments, outstanding loans, and insurance policies.
- **Reports and Analytics:**
  Generates visual representations of spending patterns, category-wise distribution, and currency conversions.

The system functions as a client–server application, where the Flutter-based front-end communicates with Firebase's distributed cloud database and authentication servers. It ensures smooth user experience across devices, with real-time data updates and offline access support.

Features were described in detail in the SRS document.

5.2 User Classes and Characteristics

Users of the system are individuals who wish to manually log, track, and manage their personal finances through a single mobile application. The system is designed to provide a comprehensive overview of transactions, budgets, and financial goals. The system supports one primary user class: the Registered User. There are no "Admin" or "Employee" roles with special privileges; all features are accessible to any user who has authenticated.

1. **Authentication & Account Management**
   - Register for a new account using a first name, last name, email, password, and preferred currency (register.dart).
   - Log in with existing email and password credentials (login_page.dart).
   - Receive password strength feedback during registration (register.dart).

- Reset a forgotten password via an email link (forgot_password.dart).
- Log out of the application (Profile.dart).

2. **Core Financial Management**
   - View a home dashboard summarizing totalMoney and transaction trends in a graph (Home.dart).
   - Add a new transaction (either Income or Expense), specifying a title, amount, category, date, time, mode of payment, and note (AddTransactionPage.dart).
   - View a list of all transactions from the Firestore database (Transactions.dart).
   - Filter the transaction list by type (Income/Expense), ascending date, or amount (Transactions.dart).
   - Add a new budget (either Expense or Savings), specifying a name, timeline, category, color, and total amount (AddBudgetPage.dart).
   - View all active budgets, separated by 'expense' and 'savings' types (budgets.dart).
   - Manually allocate funds to a budget, which updates its spending or allocation field (budgets.dart).
   - Add a new financial goal with a title and description (GoalsPage.dart).
   - View and delete existing financial goals (GoalsPage.dart).
3. **Liabilities & Subscriptions Management**
   - Add a new subscription (platform, amount, start date, note) (AddSubscriptions.dart).
   - View and delete existing subscriptions, with a dynamically calculated next payment date and progress bar (Subscriptions.dart).
   - Add a new insurance policy (title, provider, amount, term, due date, note) (AddInsurance.dart).
   - View and delete existing insurance policies, with a dynamically calculated "days remaining" progress bar (insurance.dart).
   - Add a new loan (title, amount, interest rate, tenure, start date) (AddLoans.dart).
   - View all loans and their repayment progress (amount paid vs. remaining) (Loans.dart).
   - Make a payment on an existing loan, which correctly updates the totalMoney, accruedInterest, amountPaid, and remainingAmount fields (Loans.dart).
4. **Utilities and Application Settings**
   - Change the application's base currency (ChangeCurrency.dart).
   - Toggle the application theme between Light and Dark mode (Settings.dart, darkmode.dart).
   - Use a real-time currency conversion tool (CurrencyConversion.dart).
   - Use a tax estimation tool (TaxEstimationTool.dart).
   - Export a financial report (ExportReport.dart).
   - View a static list of financial tips (Tips.dart).
   - View a static list of FAQs (FAQs.dart).
   - Submit app feedback (Feedback.dart).
   - View the app's Privacy Policy (privacypolicy.dart).

6 Flow

This section details the step-by-step logical execution of the Wallet Watch application.

6.1 User Authentication and Session Initialization

The entry point of the application. Wallet Watch uses `FirebaseAuth` for identity management and Firestore for user metadata persistence.

1. **App Launch:** The application launches by initializing the Flutter engine, loading core dependencies, and connecting to Firebase. After initialization, the user is greeted with an introductory screen that provides a brief overview of the application.
2. **Session Check:** The `FirebaseAuth` service checks for an existing persistence token.
   - **If Token Exists:** The user is automatically redirected to the *Dashboard*.
   - **If Null:** The user is directed to the *Register/Login* screen.
3. **Registration Flow:**
   - User inputs Name, Email, Password, and selects a **Base Currency**.
   - **Backend Action:** Firebase Auth creates a new UID. Simultaneously, a document is created in the `users` collection in Firestore with the initial `totalMoney` set to 0.0 and the selected currency.
4. **Login Flow:** User provides credentials. Upon success, the app retrieves the user's unique UID to query data in subsequent steps.

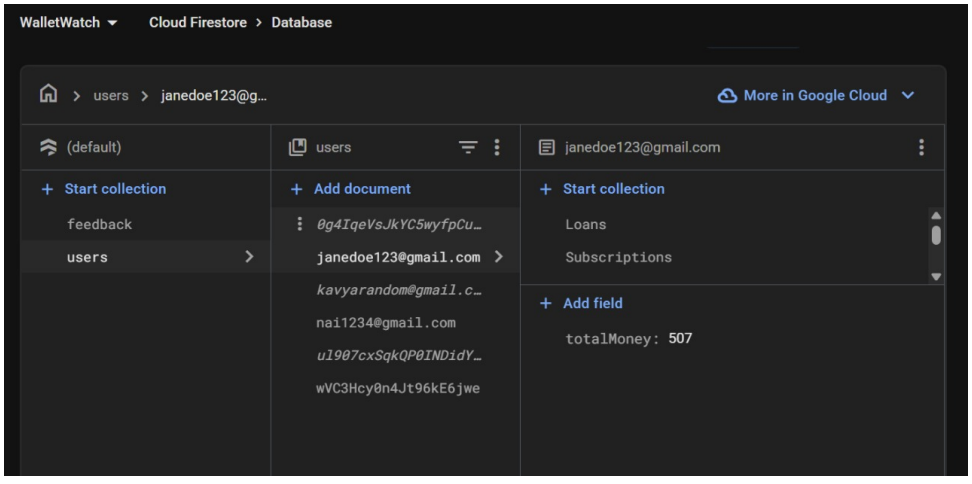

Figure 6.1: Users Collection

6.2 Dashboard and Data Synchronization

1. **Stream Initialization:** The `Home.dart` widget subscribes to a `StreamBuilder` listening to `users/{uid}`.
2. **Data Fetching:**
   - **Balance:** The `totalMoney` field is fetched.
   - **Graph Data:** A query is executed on the `transactions` sub-collection, filtered by the current month/week to populate the Income vs. Expense chart.
3. **Real-time Updates:** Any change made to the database (via other modules like Loans or Budgets) triggers an immediate UI rebuild via the Observer pattern, updating the Total Money and Graphs without manual refreshing.
4. **Dashboard overview:** The Dashboard provides a structured overview of the user's financial status, displaying the total balance, recent activity, and a real-time transaction chart. It also offers navigation shortcuts to core modules such as financial insights, goals, and transaction management through clearly defined interface elements.

6.3 Transaction Management

1. **Input:** User selects "Add Transaction," inputs Title, Amount, selects Category (e.g., Food, Salary), picks Date/Time, and chooses a Payment Mode (e.g., UPI, Cash) and adds a note(optional).
2. **Validation:** Client-side logic ensures the amount is positive and required fields are filled.
3. **Execution:**
   - **A:** A new document is added to the `transactions` sub-collection with the provided metadata.
   - **B:** The application reads the user's current `totalMoney`.
   - **C:** Logic dictates: If *Income*, add to total; if *Expense*, subtract from total.
   - **D:** A Firestore update call persists the new `totalMoney` value.
4. **Filtering and navigation:** The module also supports transaction filtering by type and date range, enabling users to analyze past activity more efficiently. Additional navigation controls allow access to related modules such as loans, subscriptions, and insurance records.

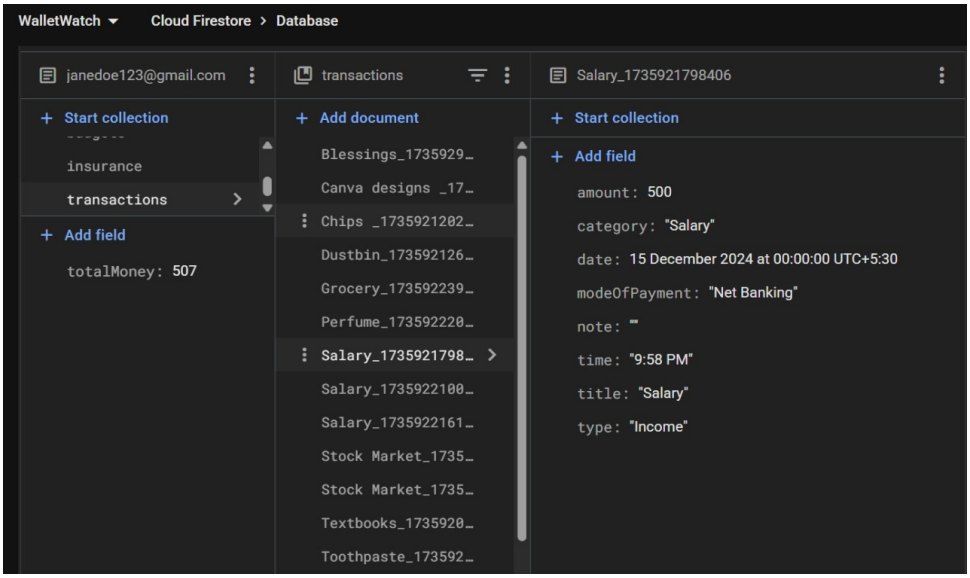

Figure 6.2: Transaction Collection

6.4 Budgeting and Allocation Strategy

1. **Budget Creation:** User defines a budget (e.g., "Groceries"), sets a limit, and assigns a color tag. This is stored in the `budgets` collection.
2. **Funds Allocation:**
   - User selects "Allocate Funds" on a specific budget card.
   - The system updates the `spending` or `allocation` field of that specific budget document.
3. **Progress Calculation:** The UI dynamically calculates the percentage:

$$\text{Progress} = \left(\frac{\text{Allocated or Spent}}{\text{Total Budget Limit}} \right) \times 100$$

   This drives the linear progress indicators shown on the Budgets screen.

6.5 Liability Management (Loans, Subscriptions, Insurance)

6.5.1 Loan Management

- **Creation:** A loan document is added under `users/{uid}/loans` with fields like `totalAmount`, `remainingAmount`, and `dueDate`.
- **Repayment:**
  - User enters a payment amount.
  - App computes:

```
remainingNew = remainingOld - payment
```

  - A batch update:
    1. Updates the loan's `remainingAmount`.
    2. Deducts the payment from the user's `totalMoney`.

- **Completion:** When `remainingAmount` reaches 0, the loan is marked as completed.

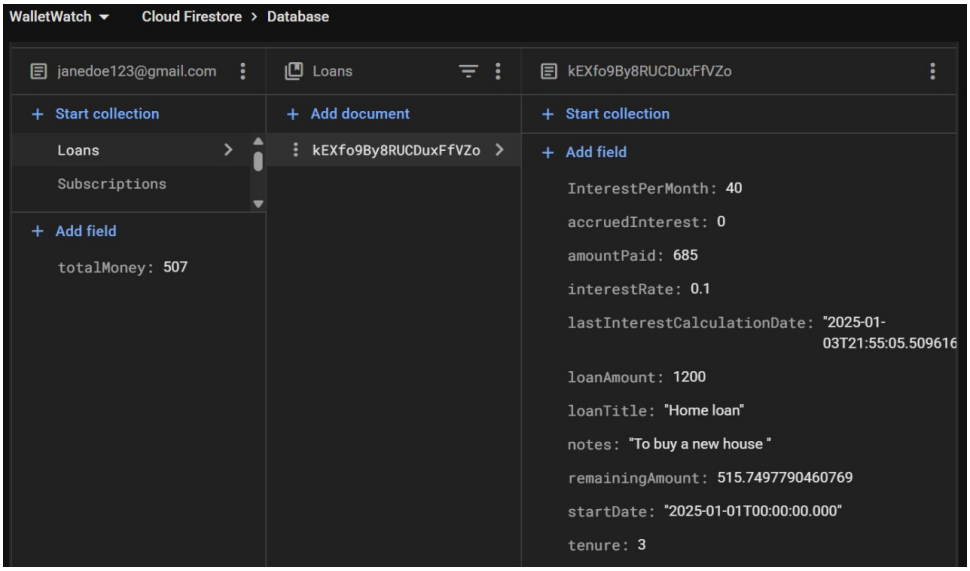

Figure 6.3: Loan collection

6.5.2 Subscription Management

- **Creation:** A subscription document is stored with fields like `amount`, `platform`, etc.
- **Recurring Charge:** When the billing date arrives:
  1. A transaction document is created under the `transactions` sub-collection representing the subscription charge.
  2. The subscription amount is deducted from `totalMoney`.
  3. `nextPaymentDate` is updated based on the billing cycle.

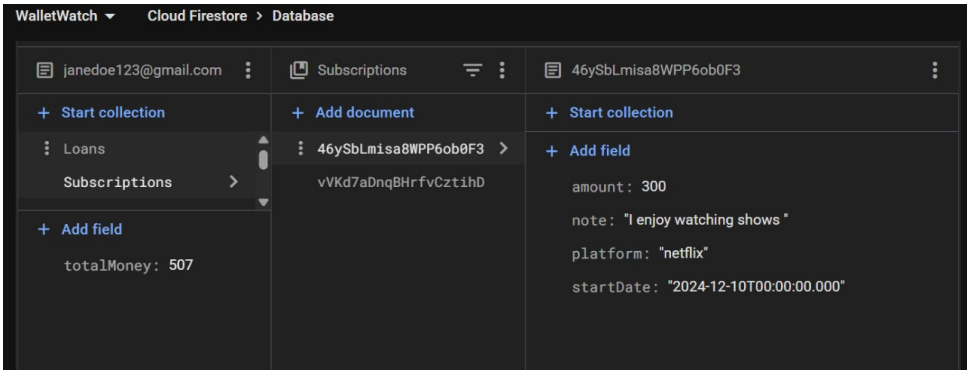

Figure 6.4: Subscription collection

6.5.3 Insurance Management

- **Creation:** Insurance entries include fields like `amount`, `dueDate`, etc
- **Premium Payment:** When due:
  1. A transaction entry is generated representing the insurance premium payment.

2. The premium amount is deducted from `totalMoney`.
3. `nextDueDate` is updated according to the premium cycle.

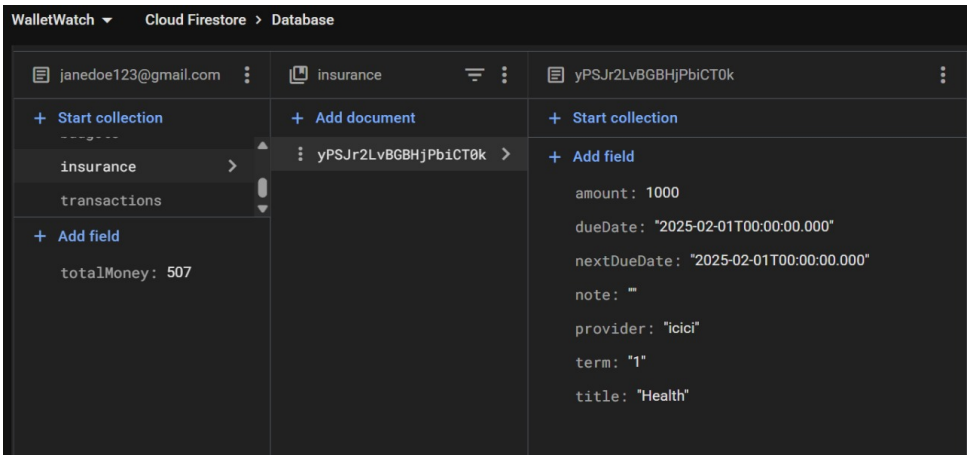

Figure 6.5: Insurance collection

6.6 Settings and Personalization

The Settings page allows users to customize their application experience and adjust account-specific preferences. These operations mostly affect local app behavior while keeping essential profile data synced through Firebase.

6.6.1 Profile Overview

At the top of the page, the user's name, email, and avatar initials are displayed. These details are updated automatically whenever profile data changes.

6.6.2 Settings Options

The settings menu includes the following core actions:

i) Theme Toggle (Light/Dark Mode)

- The user can switch between Light and Dark themes.

ii) Feedback and Rating

- Opens a dedicated feedback screen where the user can submit comments.
- Sends data to Firestore under a `feedback` collection.

iii) FAQs

- Redirects the user to a static FAQs screen explaining app usage, features, and troubleshooting steps.

iv) Goals

- Navigates to the Goals module where users track savings goals.

v) Currency Conversion

- Displays a conversion interface using live exchange rates.
- Fetches data through an HTTP GET request to the ExchangeRate API.

vi) Log Out

- Clears local persistence and redirects the user back to the login screen.

vii) Additional Links

Meet the Team navigates to an informational screen displaying developer details, app credits, and acknowledgments.

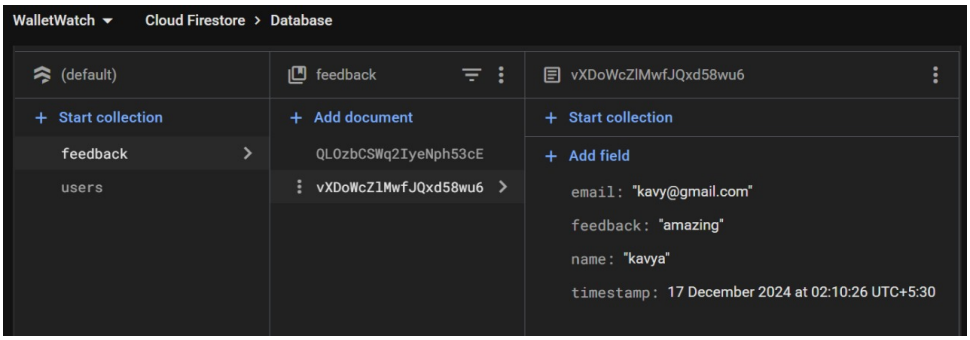

Figure 6.6: Feedback

7 Database Schema

The application uses a Data-Centered Architecture with Cloud Firestore. The database is organized into the following collections and documents.

`feedback` Collection

This is a top-level collection to store all feedback submitted by users.

- **Document ID:** Auto-generated (e.g., `vXDoWc...`)
- **Fields:**
  - `name`: String
  - `email`: String
  - `feedback`: String
  - `timestamp`: Timestamp

`users` Collection

This is the primary collection that holds all data for every user.

- **Document ID:** The user's email address (e.g., `janedoe123@gmail.com`)
- **Fields (on user document):**
  - `totalMoney`: double
  - (other fields like `firstName`, `lastName`, etc.)
- **Sub-collections:** Each user document contains its own sub-collections:
  - `transactions` **Sub-collection**
    - **Fields:**
      - `amount`: double
      - `category`: String
      - `date`: Timestamp
      - `modeOfPayment`: String
      - `note`: String
      - `time`: String
      - `title`: String
      - `type`: String
  - `Subscriptions` **Sub-collection**
    - **Fields:**
      - `platform`: String
      - `amount`: double

        - `startDate`: String
        - `note`: String
- `Loans` **Sub-collection**
    - **Fields:**
        - `loanTitle`: String
        - `loanAmount`: double
        - `interestRate`: double
        - `tenure`: int
        - `startDate`:
        - `remainingAmount`: double
        - `amountPaid`: double
        - `InterestPerMonth`: double
        - `accruedInterest`: double
        - `lastInterestCalculationDate`: String
        - `notes`: String
- `insurance` **Sub-collection**
    - **Fields:**
        - `title`: String
        - `provider`: String
        - `amount`: double
        - `term`: String
        - `dueDate`:
        - `nextDueDate`:
        - `note`: String

8 Diagrams

8.1 Usecase Diagram

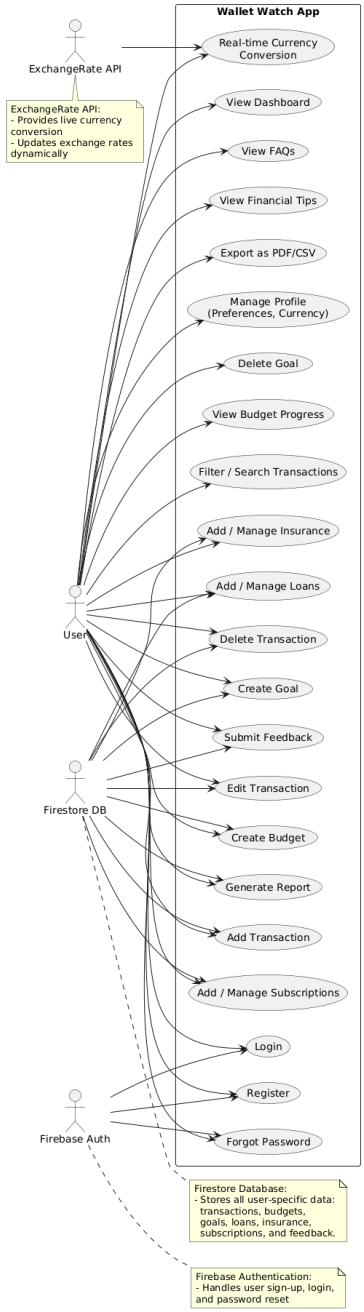

Figure 8.1: Usecase Diagram

8.2 Class Diagram

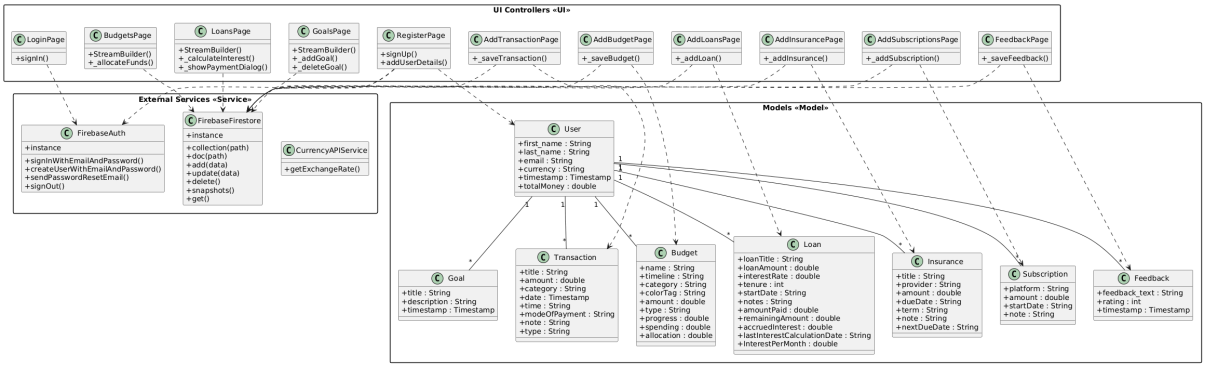

Figure 8.2: Class Diagram

8.3 Object Diagram

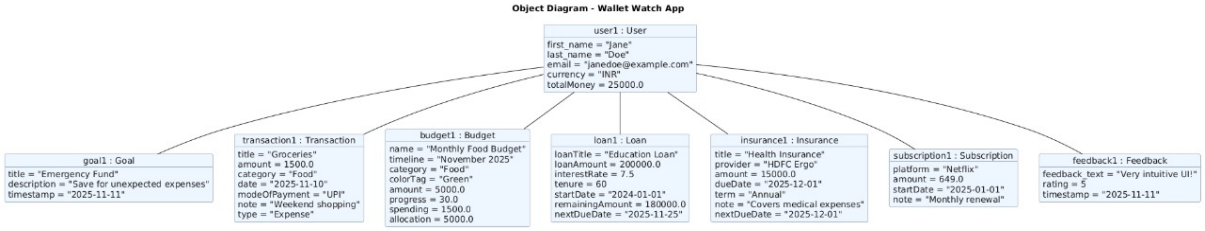

Figure 8.3: Object Diagram

8.4 ER Diagram

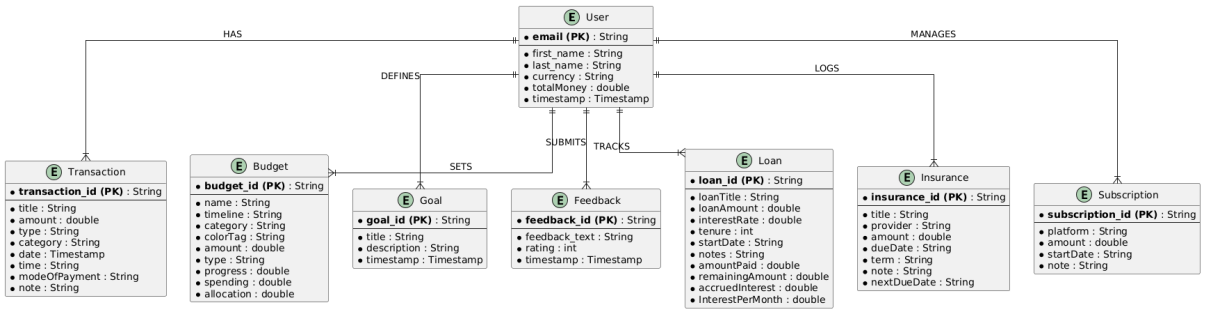

Figure 8.4: ER Diagram

8.5 Activity Diagrams

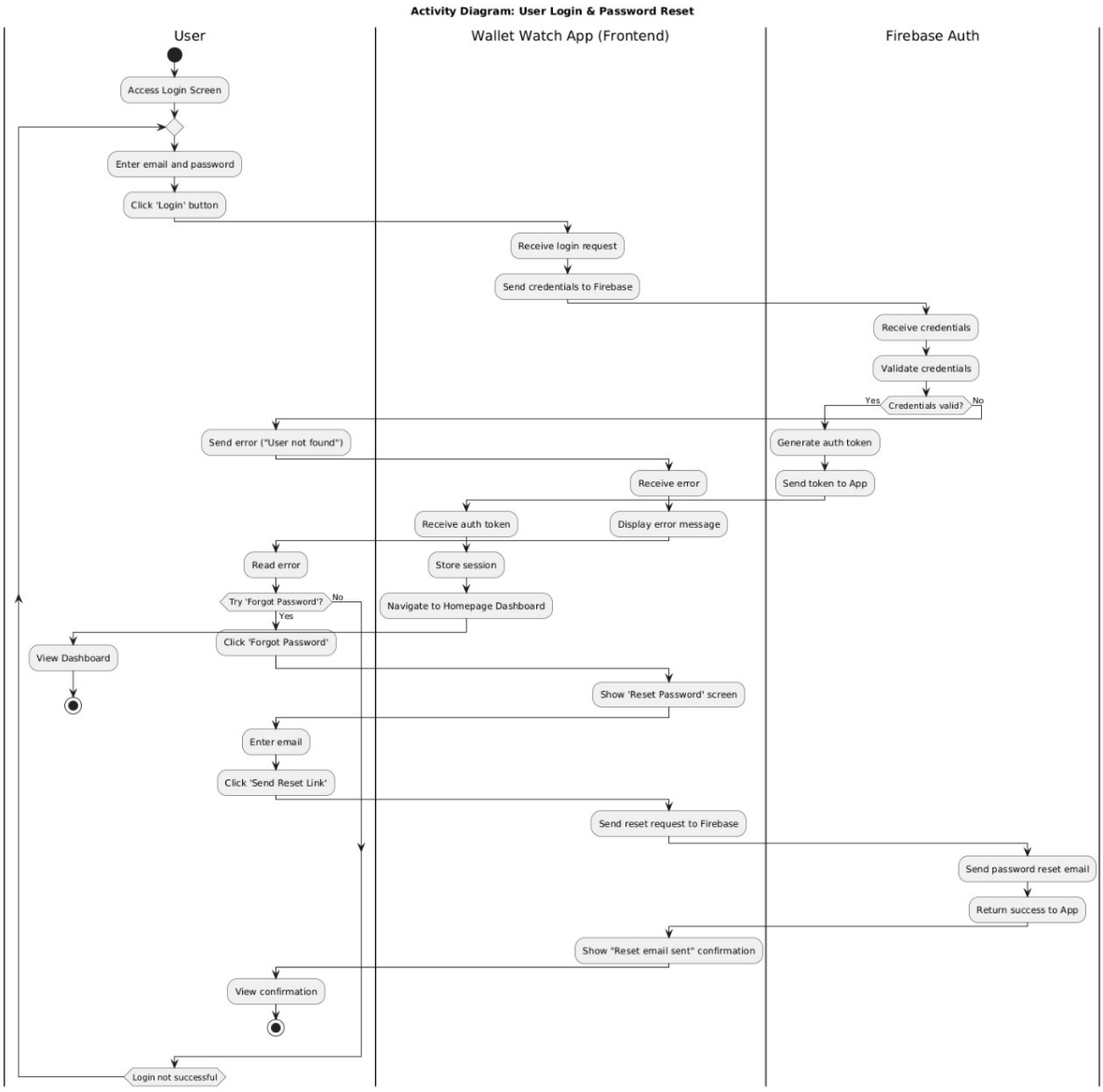

Figure 8.5: Activity_login_passwordreset

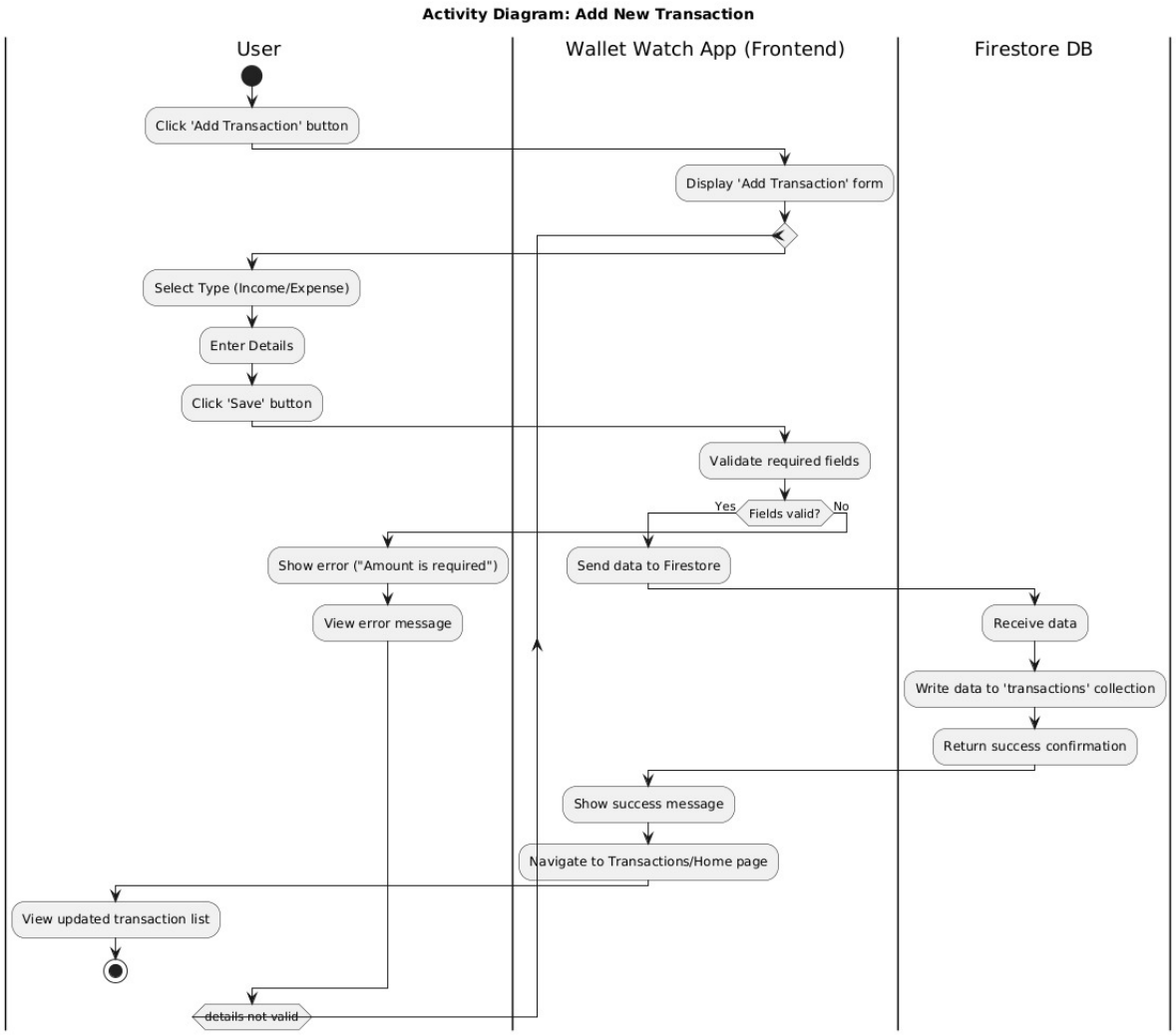

Figure 8.6: Activity_transaction

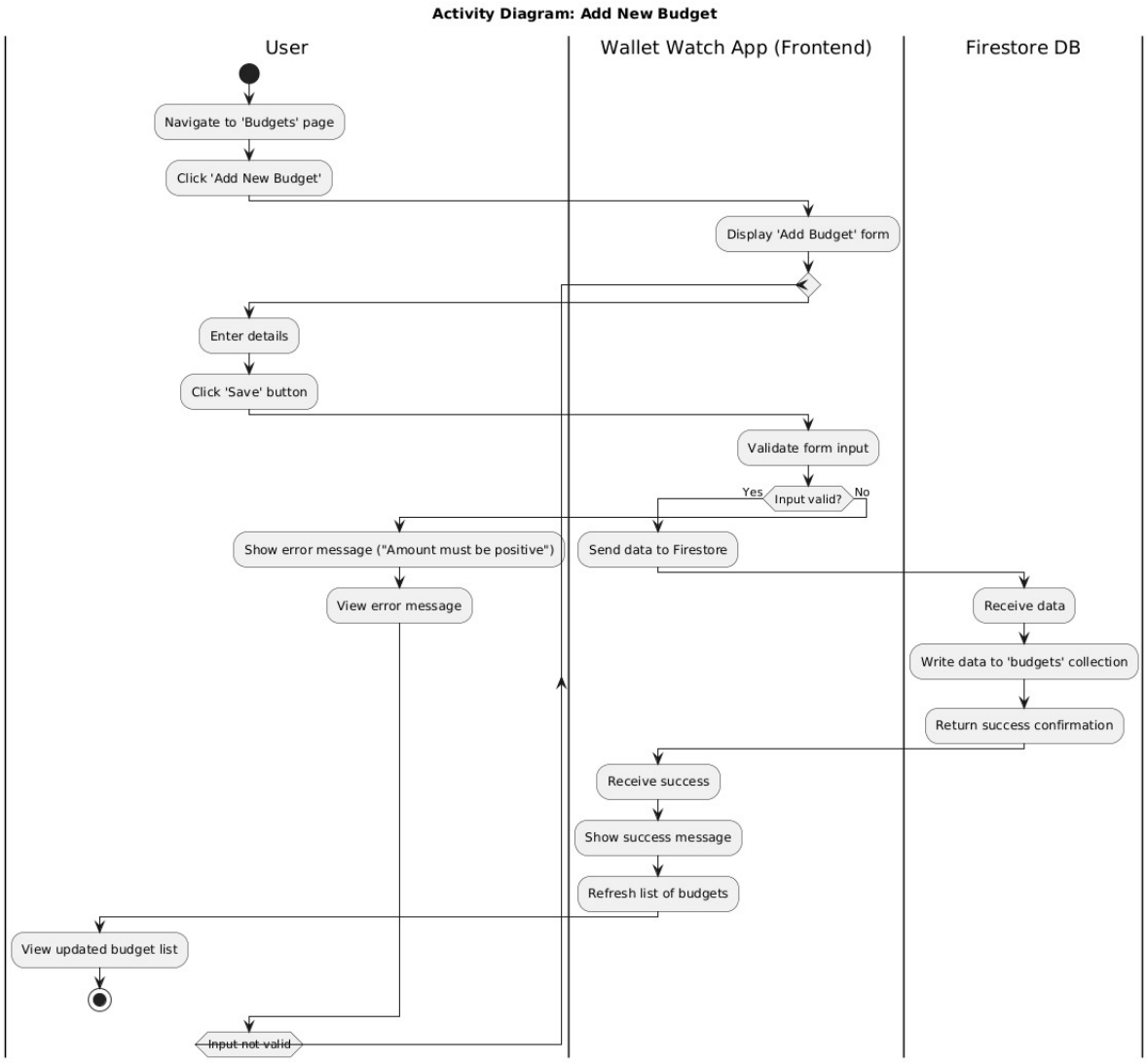

Figure 8.7: Activity_budget

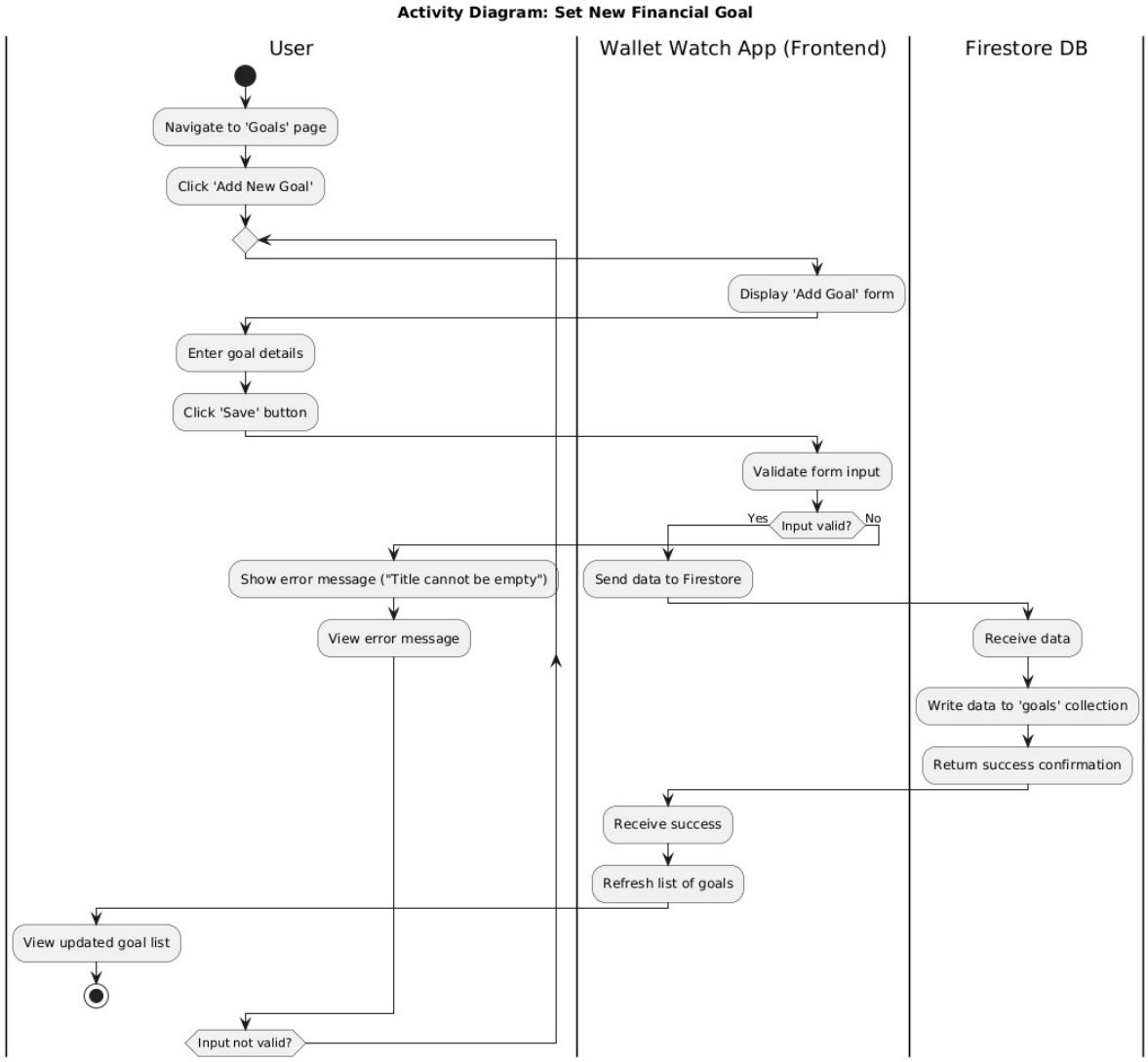

Figure 8.8: Activity_goal

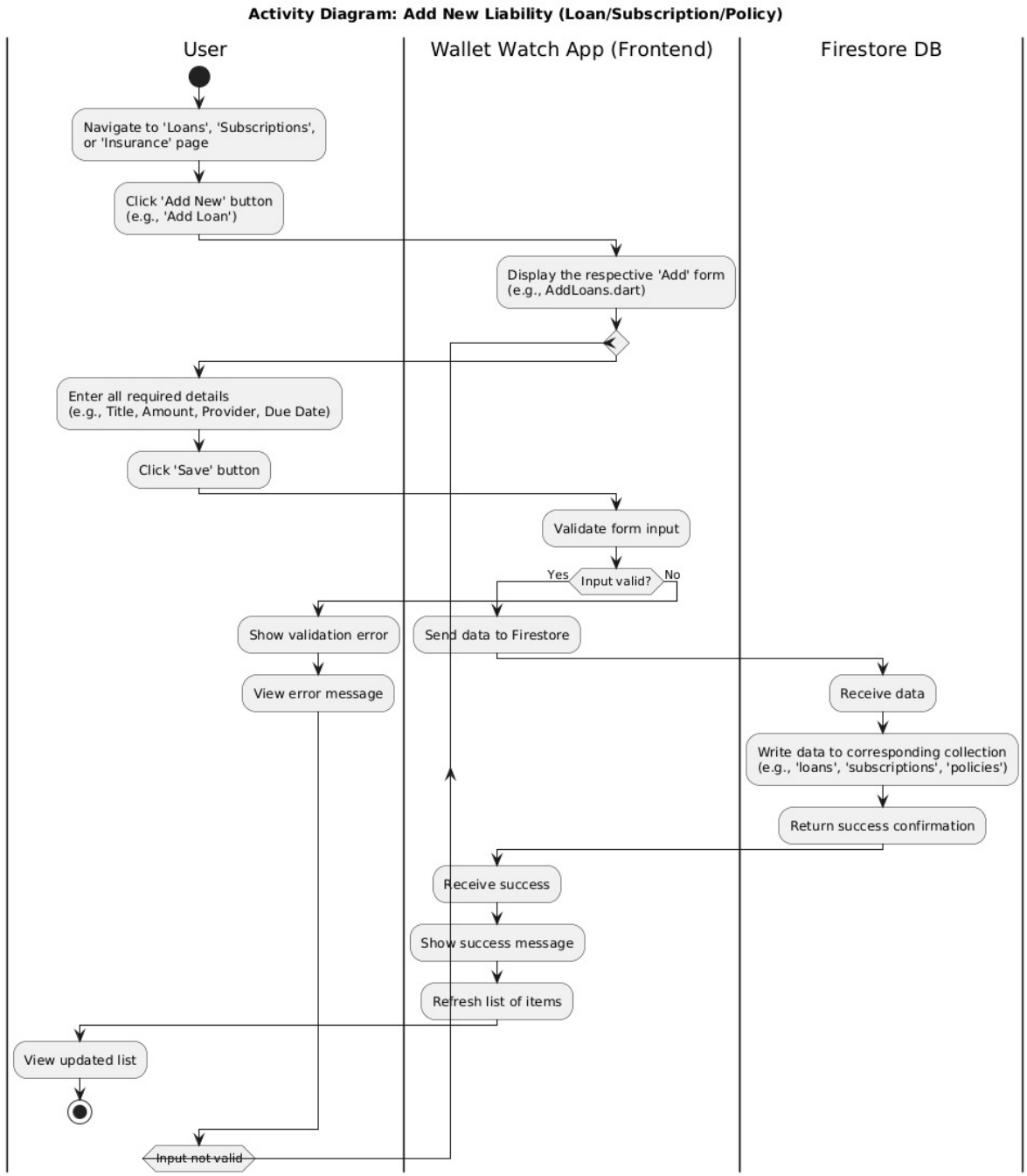

Figure 8.9: Activity_loan_sub_insurancepolicy

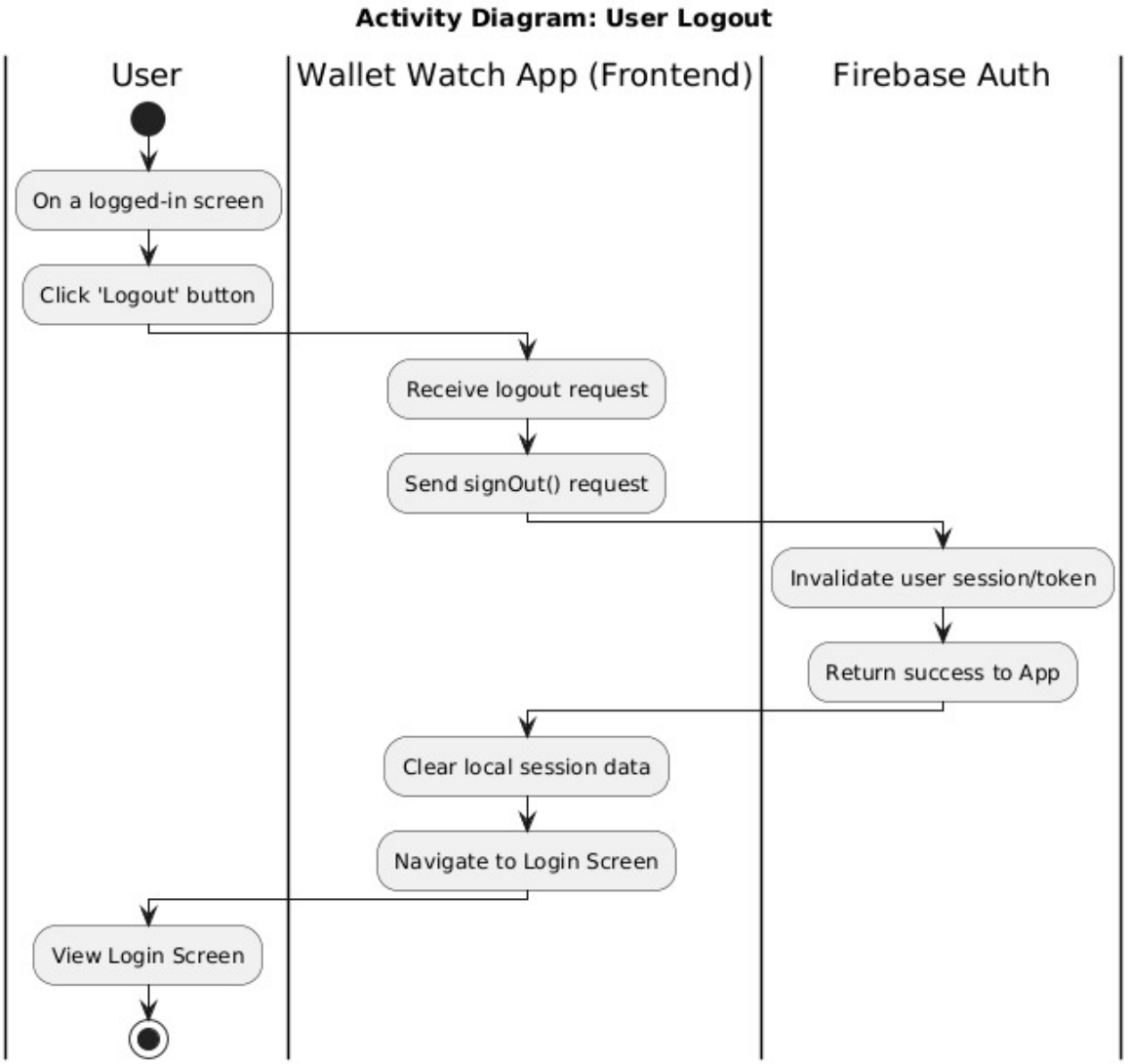

Figure 8.10: Activity_login_logout

8.6 Sequence Diagrams

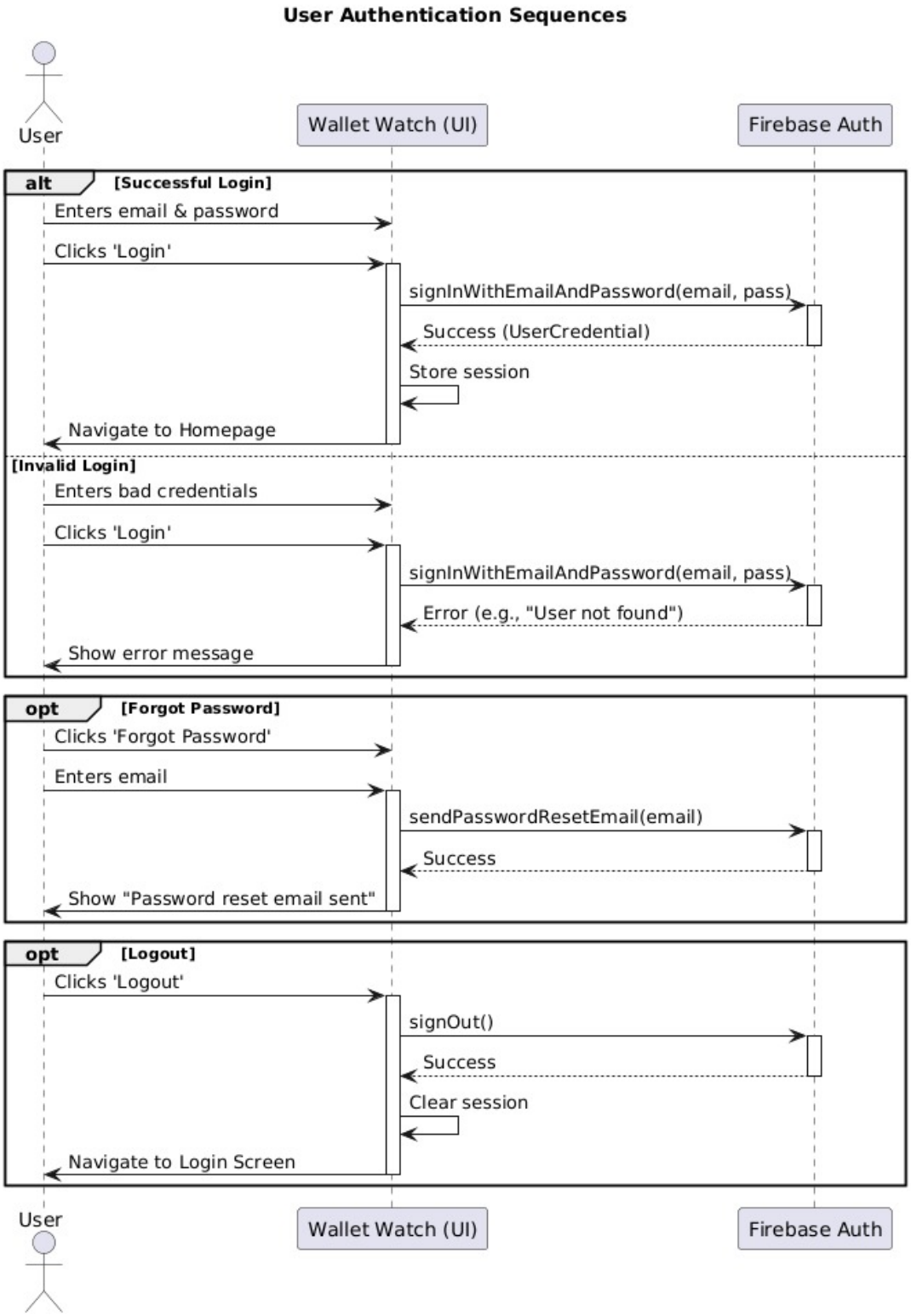

Figure 8.11: Sequence_authentication

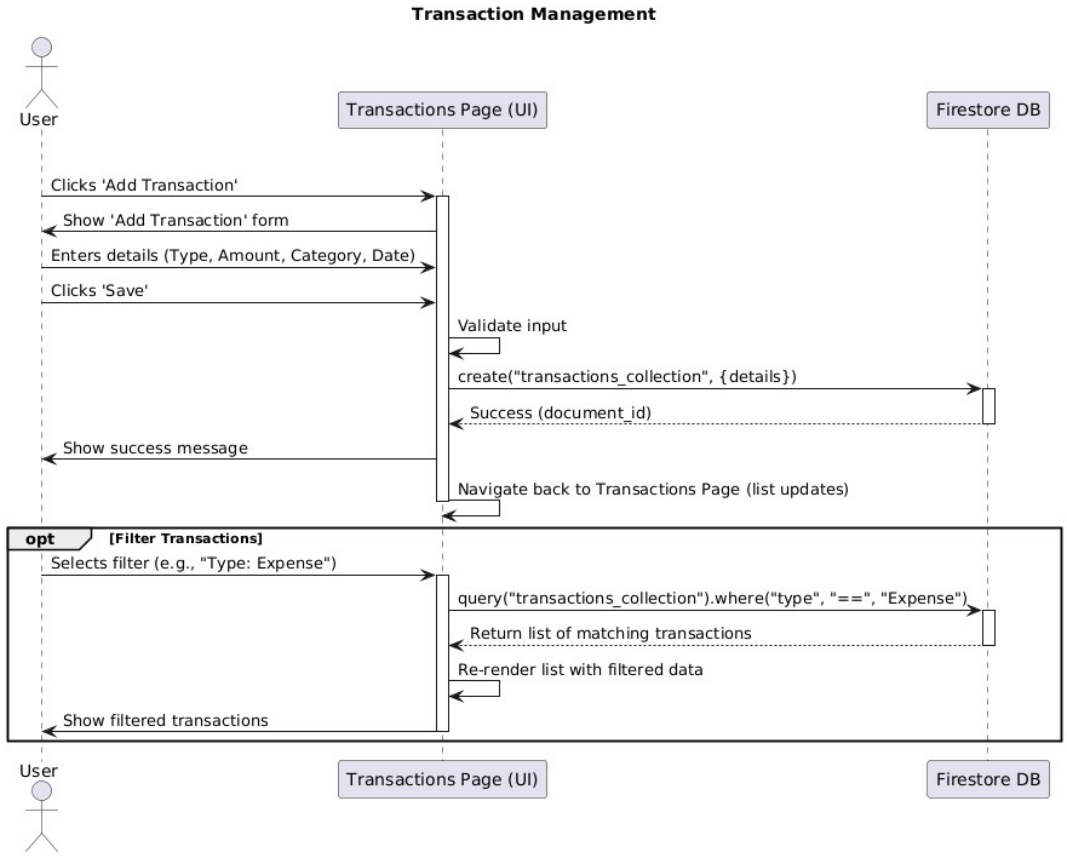

Figure 8.12: Sequence_transaction

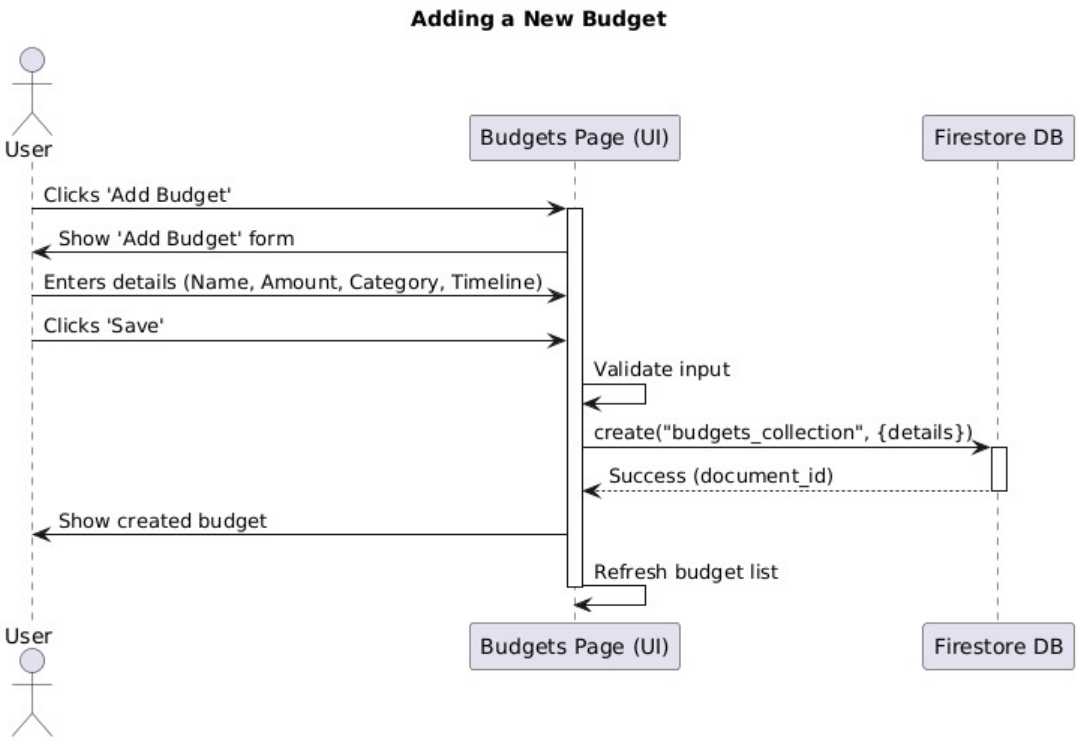

Figure 8.13: Sequence_budget

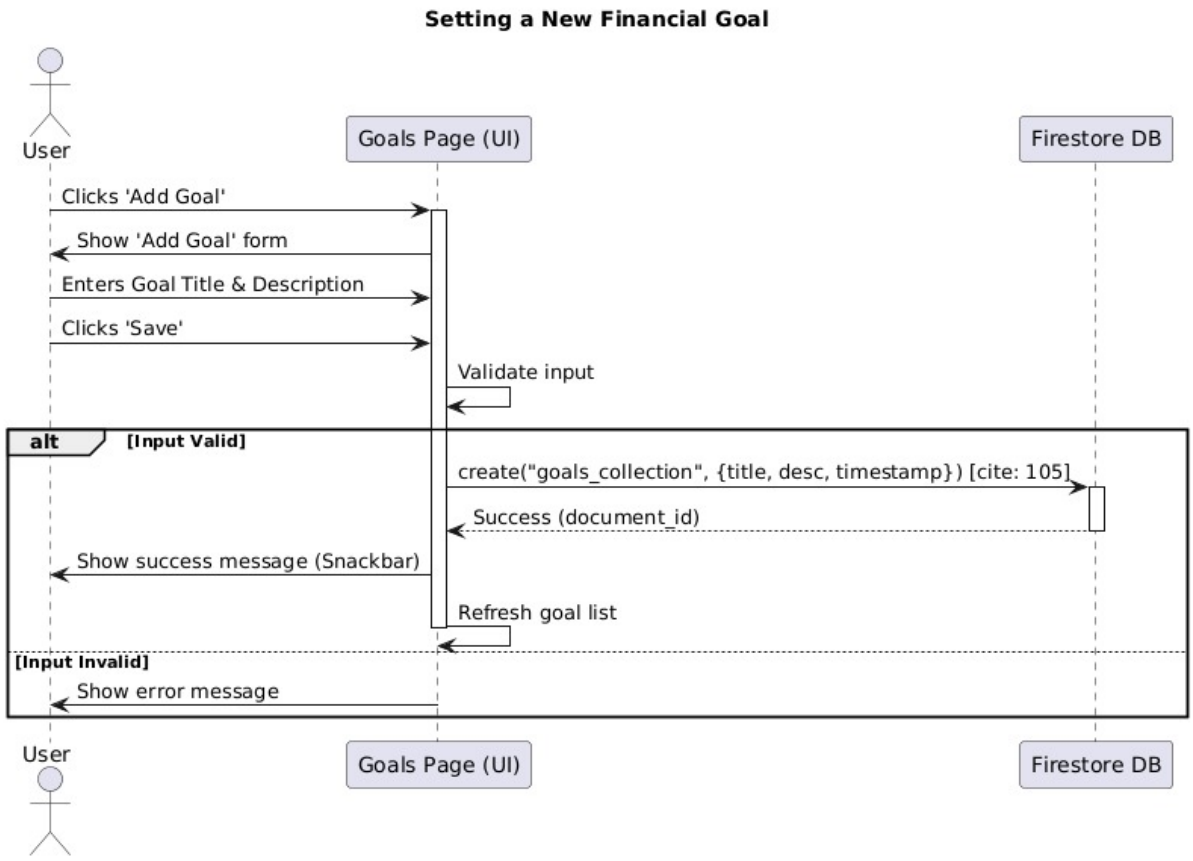

Figure 8.14: Sequence_goals

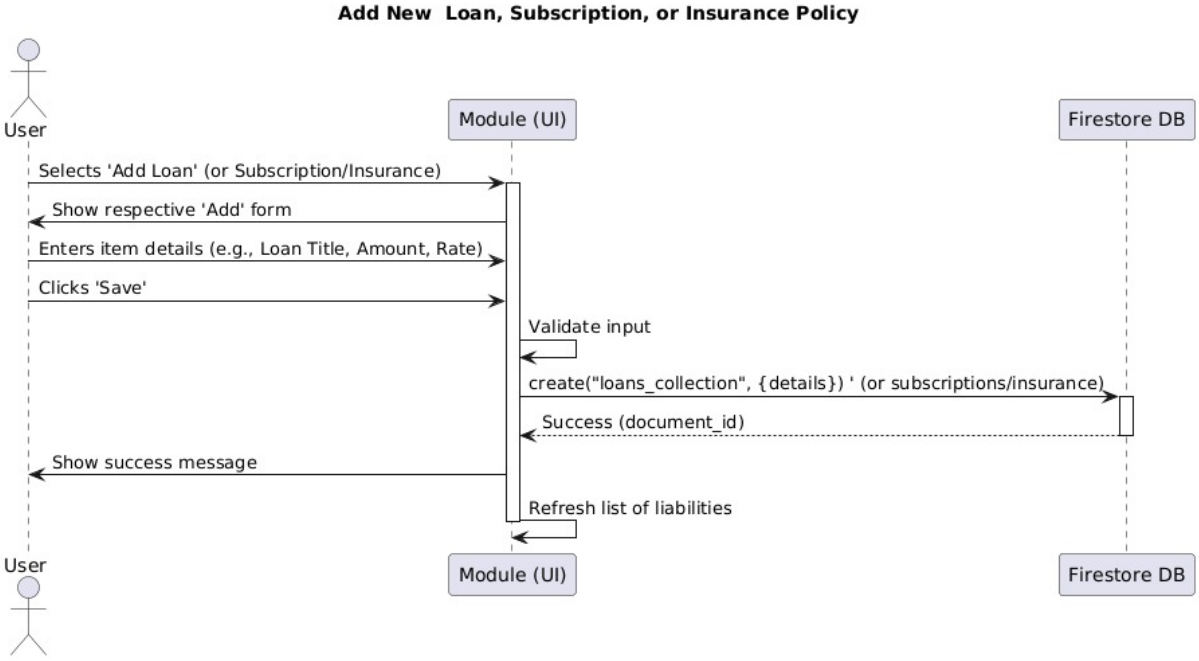

Figure 8.15: Sequence_loan_subscription_insurance

8.7 State Diagrams

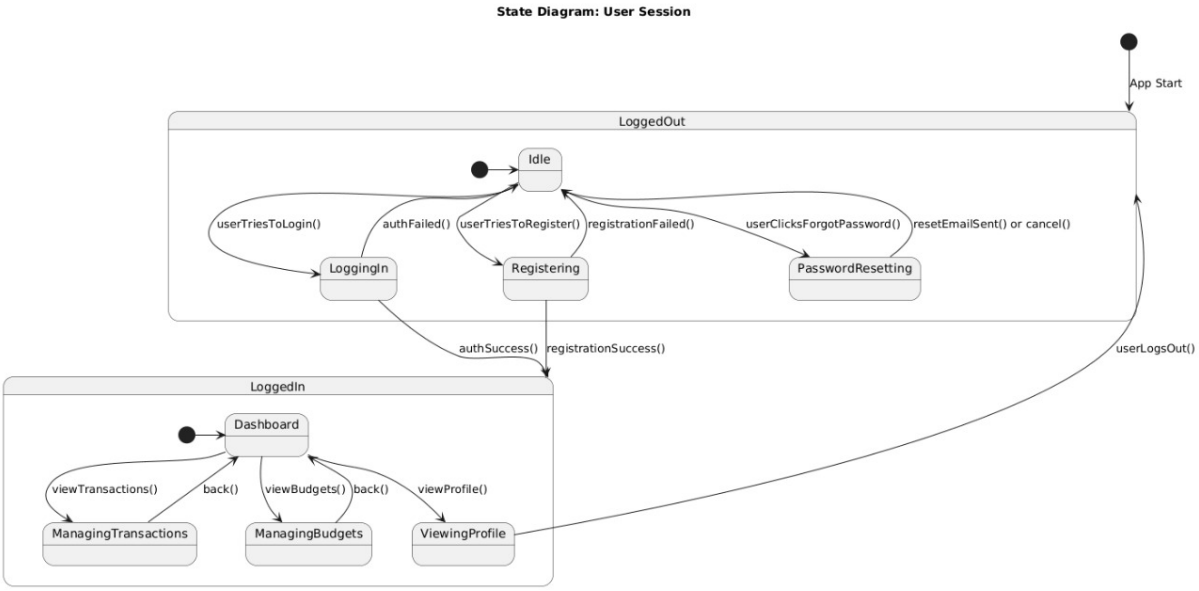

Figure 8.16: State_User_Session

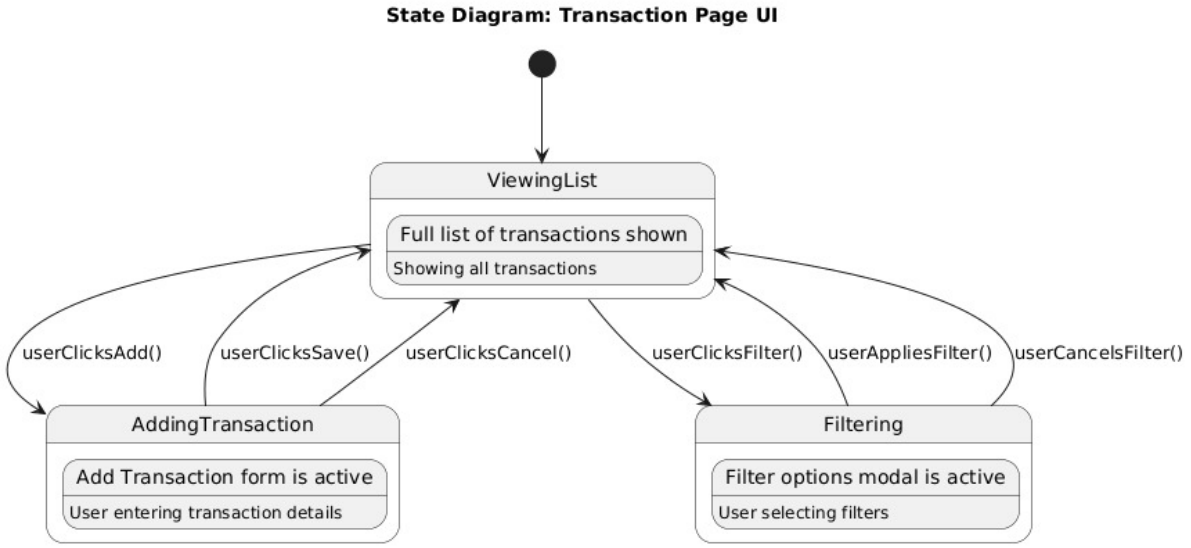

Figure 8.17: State_Transaction

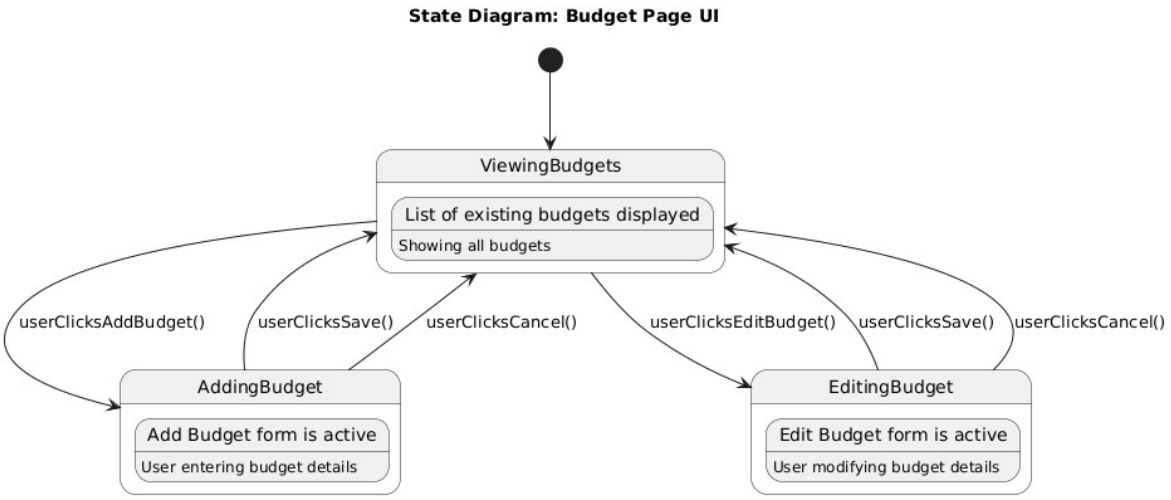

Figure 8.18: State_Budget

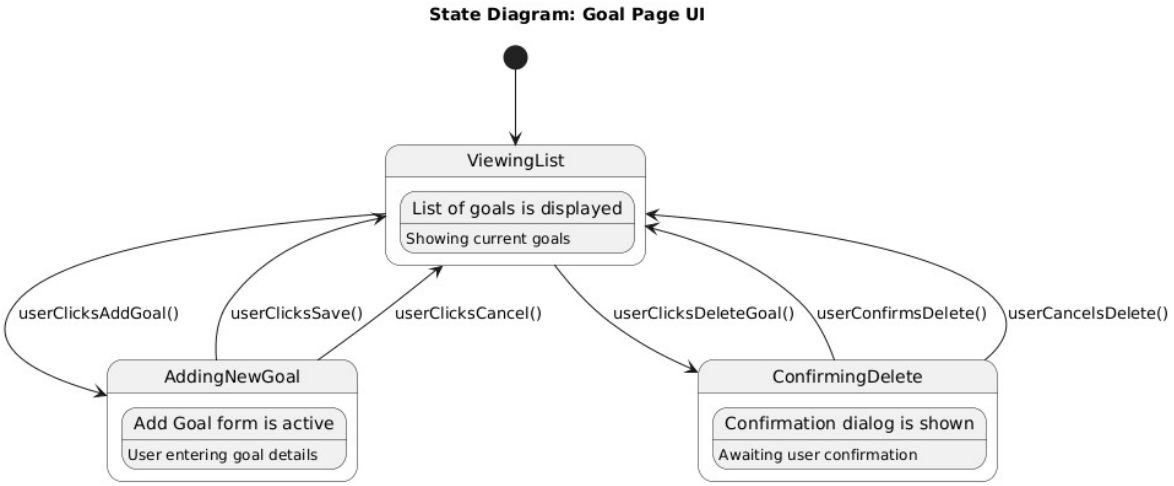

Figure 8.19: State_Goal

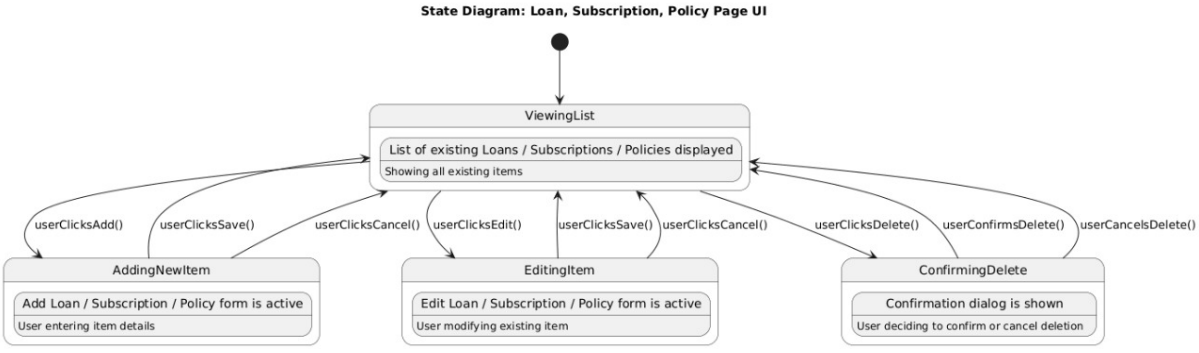

Figure 8.20: State_Loan_Sub_InsurancePolicy

8.8 Gantt Charts

We used the Agile Methodology

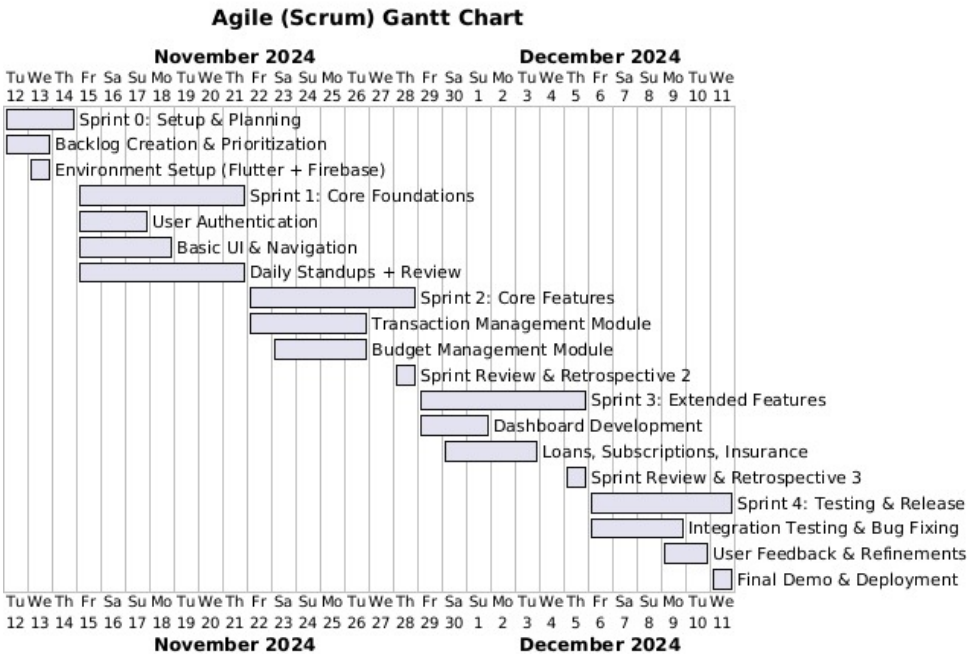

Figure 8.21: Gantt_Agile