Technical Implementation of a Software Product

Course:	5G00FT06-3004 Software Project, Autumn 2024
Product:	Receipt Manager App
When implemented?	Autumn 2024
Members of the project team:	Jingjing Yang, Razib Hasan, Xiaosi Huang
Client/customer:	Maria Valli, Anne-Mari Stenbacka

Version History of this Document

Date	Modified by	Description of Changes
21.11.2024	Jingjing Yang	The first draft
25.11.2024	Xiaosi Huang	The second draft
26.11.2024	Razib Hasan	The third draft
06.11.2024	Jingjing Yang	The final version

Table of Contents

1. Product

1.1 Product Description

Receipt Manager is a Flutter-based application designed to help users efficiently manage their finances by scanning, categorizing, and visualizing receipts. The app integrates Firebase and third-party APIs for seamless user authentication, data storage, and financial insights.

1.2 Other Documentation

- Trello Broad: Product Backlog, Sprint Backlog
  https://trello.com/b/2Eayp9hg/receipt-tracker-app
- Working hours sheet

  https://tuni-my.sharepoint.com/:x:/r/personal/xiaosi_huang_tuni_fi/Documents/Work%20hours%20-%20Team%20C.xlsx?d=w5606e284919743848ad30f390066eb87&csf=1&web=1&e=2e1Kp2
- Figma UI templates used for design inspiration.

  https://www.figma.com/design/PqDB8zj748XUsuDSdeFGFF/Team-C?node-id=177-1256&t=fc0Go9yFtEzWRY9P-1
- Sprint Reports: Outlines the progress across five sprints.

  https://tuni-my.sharepoint.com/:w:/r/personal/jingjing_yang_tuni_fi/_layouts/15/Doc.aspx?sourcedoc=%7BF31711B6-6D68-4412-A834-95BCA4BF497C%7D&file=Sprint%20Report.docx&action=default&mobileredirect=true
- README: Detailed guide for setup and usage.

  https://github.com/jingjingyang0803/Receipt-Manager-App/blob/main/README.md
- DEMO video: Detailed guide for app usage.

  https://youtu.be/X_dJnOpCZHU

  https://youtu.be/F-EuRJqj0iY

1.3 Code and Data Availability

- GitHub Repository: https://github.com/jingjingyang0803/Receipt-Manager-App

- Access: Public repository.

1.4 Relevant Addresses

- Development/Test Environments: Local setup using emulators/simulators and physical devices across iOS/android platforms.
- There is no production deployment at this stage.

2. Implementational Decisions

2.1 Constraints, Demands, and Justifications

- **Environment**: Built using Flutter for cross-platform compatibility (Android and iOS).
- **Key Constraints**:
    - **Resource Limitations:**
        - The project relied on the **free version of Firebase**, which imposed restrictions such as the inability to customize email verification or password reset messages.
        - The design team lacked professional experience, resulting in a relatively simple **UI/UX Design**.
        - Limited funding meant the app could not be published to official app stores, requiring manual installation instead.
    - **OCR Development:**
        - Limited resources for OCR development led to the use of **Firebase Cloud Functions** and **Google Cloud Vision API**.
        - Time constraints and complexity of implementing advanced OCR features restricted functionality.
    - **Feature Scope:**
        - Due to time limitations, several advanced features were omitted, focusing instead on core functionality.
    - **Manual Categorization:**
        - Regional variations in merchant data and the complexity of implementing automatic categorization (e.g., machine learning models) led to prioritizing **manual category matching**.

- **Solutions Tested**
    - **Email Customization:**
        - Customizing email content for verification and password reset was tested but found impossible with the free Firebase plan.
    - **OCR Libraries:**
        - Initially, **Google's ML Kit for Flutter** was tested, but conflicts with Firebase integration created challenges, especially during mobile deployment.
        - An alternative library, **Tesseract OCR**, was tried, but text recognition proved difficult to implement effectively on iOS.
        - Ultimately, **Firebase Cloud Functions** were combined with the Google Cloud Vision API, successfully resolving conflicts and enabling reliable OCR functionality.
    - **Auto-Categorization:**
        - Automatic receipt categorization was explored but found to be infeasible due to regional variations and limited machine learning resources. Manual categorization was chosen for its user-friendliness and simplicity.
- **Justifications**
    - **Affordability:** The project relied on free tools and services to stay within budget, prioritizing practicality over advanced features.
    - **Time Constraints:** Limited development time dictated a focus on core functionality and manual approaches over automated solutions.
    - **User Experience:** Manual categorization was selected to ensure a straightforward and reliable user experience, even without sophisticated AI tools.
    - **Deployment:** Without the funding required to publish the app to stores, manual installation remains the only viable option.

2.2 Languages, Libraries, and Tools Used

- **Languages**: Dart
- **Backend**: Firebase
- **Libraries**:

- firebase_core, firebase_auth, cloud_firestore: Core Firebase services.
- provider: For state management.
- logger: For structured logging during development.
- fl_chart: For data visualization.
- See more in the README.md file.

3. User Roles

3.1 Registered Users

- **Register and log in** with verifed email**.**
- Edit **user profiles.**
- Set preferred **currency** to display**.**
- Add and delete **categories.**
- Add, edit, and delete **receipts.**
- Set category-specific **budgets.**
- Check the **visualization** of expenses using bar charts, pie charts, and line charts.
- Set **filtering** options (date range, payment methods, categories to include) and **sorting order.**
- Search receipts from the list are based on some **queries** and criteria**.**

3.2 Admins

- The Firebase project is managed under **Jingjing Yang's Firebase account**, which grants her full administrative control over stored data.

4. Data Protection and Security

Sensitive Data

- The system stores sensitive data, including:
  - **User authentication data** (email and password, managed securely through Firebase Authentication).
  - **Financial data** stored in **Firestore**, tied to user-specific documents and collections.

- Access control is implemented through Firebase **security rules**, ensuring that authenticated users can only access their own data.

Backup Policy

- Data storage and real-time synchronization are managed by Firebase, which inherently provides **backup and recovery mechanisms**.
- Firebase's infrastructure ensures high availability and redundancy, reducing the risk of data loss.

Access and Ownership

- The Firebase project is managed under **Jingjing Yang's Firebase account**, which grants her full administrative control over stored data.
- Four main **Firestore collections** are used to organize user and financial data efficiently.

Security Measures

1. **Access Control Rules:**
    - Firebase security rules restrict access to authenticated users, ensuring that users can only view and modify their own data.
    - Password management (reset and change functionality) is also restricted to authenticated users for added security.
2. **Encryption:**
    - All data is encrypted in transit using HTTPS and encrypted at rest by Firebase to protect against unauthorized access.
3. **Authentication:**
    - User authentication is handled securely through Firebase Authentication.

Risks and Maintenance

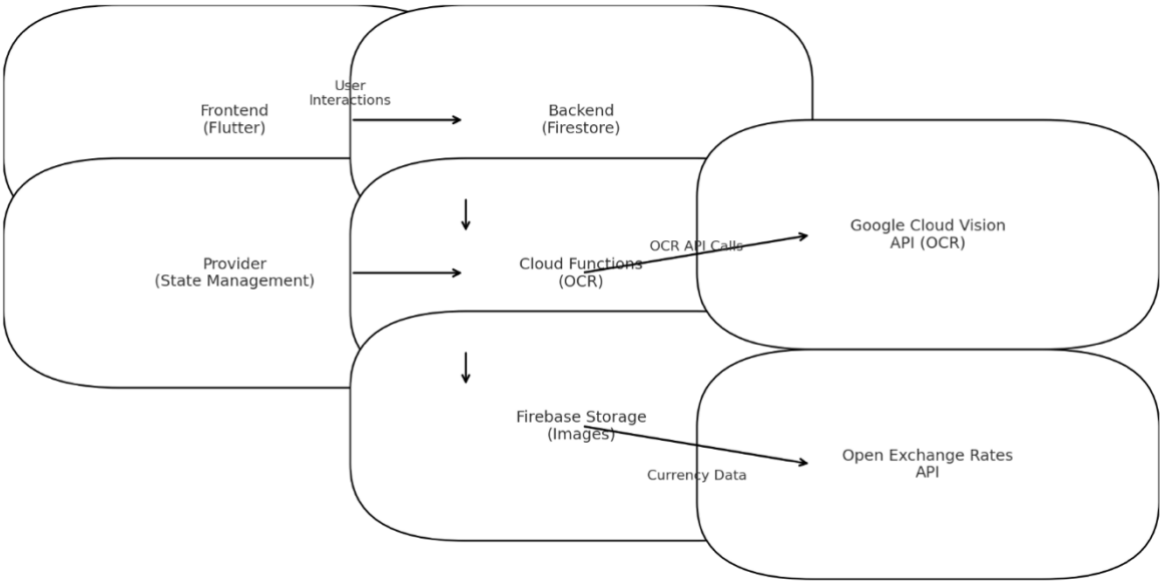
- **Sensitive Data Management:**
    - The system relies on the **owner's Firebase account**, making it important to secure account access (e.g., strong passwords, two-factor authentication).
- **Future Development Considerations:**
    - Future developers must adhere to existing **Firebase rules** and ensure that no modifications compromise data integrity or security.

- Proper documentation for updating Firebase rules or extending collections must be maintained.

Relevant Documentation

- Firebase documentation for **security rules**, **authentication**, and **backup policies** can serve as a reference for further enhancements or troubleshooting.

5. Architectural Overview



5.1 Description of the System and Relations

Frontend

- **Technology**: Built using **Flutter**, the frontend manages user interactions and communicates with the backend through HTTP requests and Firebase integrations.

Backend

- **Firebase Firestore**:
    - Stores user and application data, including:
        - Receipts, categories, budgets, and user-specific settings.
    - Data is organized per user, ensuring security and privacy using Firebase Authentication and Firestore rules.
- **Firebase Cloud Storage**:
    - Used for storing images uploaded by users, such as:

        - Receipt photos (for OCR processing).
        - Profile images (user-specific resources).
    - Acts as the storage backend for receipt images used in OCR.
- **Cloud Functions**:
    - Executes server-side logic, specifically:
        - **OCR Processing**: Handles requests to the **Google Cloud Vision API** for extracting text from receipt images stored in Firebase Storage.

External APIs

- **Google Cloud Vision API**:
    - Processes **Optical Character Recognition (OCR)** for receipt images uploaded to Firebase Cloud Storage.
- **Open Exchange Rates API**:
    - Provides real-time exchange rates for currency conversion.

State Management

- **Provider with ChangeNotifier**:
    - Efficiently manages application state, including:
        - User settings, budgets, receipt data, and categories.

5.2 Firebase Configuration Details

1. Cloud Functions

- **Functions Deployed**:
    - annotateImageCallable (Callable Function): Handles OCR processing requests via Firebase and communicates with the Google Cloud Vision API.
    - annotateImageHTTP (HTTP Function): Provides a direct HTTP endpoint for OCR requests.
- **Purpose**:
    - Executes server-side logic, such as calling the Google Cloud Vision API for OCR tasks.

2. Machine Learning

- **Feature Used**: Text recognition for receipt image analysis.
- **Integration**:
    - The **Google Cloud Vision API** is utilized through Cloud Functions for OCR, extracting data from receipt images stored in Firebase Storage.

3. Firestore Database

- **Collections Used**:
    - budgets: Stores user budget information.
    - categories: Maintains category details for receipts.
    - receipts: Holds receipt details, including OCR results and metadata.
    - users: Stores user profiles and settings.
- **Data Structure**:
    - Data is organized per user, ensuring security and privacy using Firebase Authentication and Firestore rules.

4. Firebase Storage

- **Buckets**:
    - receipts/: Contains uploaded receipt images.
    - users/: Stores user profile pictures.
- **Purpose**:
    - Acts as the storage backend for receipt images used in OCR.

5. Remote Config

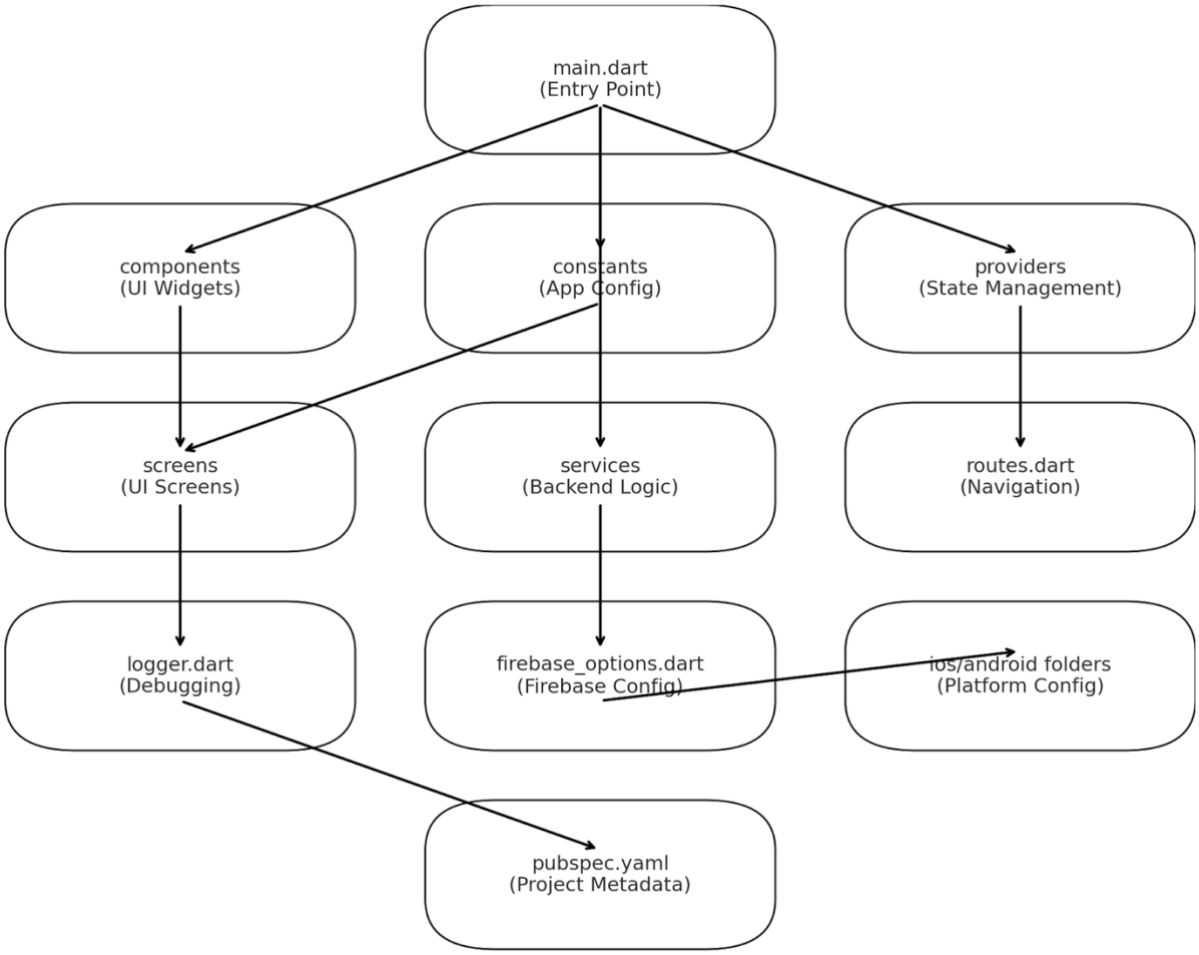
- **Parameters Used**:
    - API_KEY: Stores the Open Exchange Rates API key for currency conversion.
- **Purpose**:
    - Provides dynamic configuration, enabling seamless updates to API keys without requiring app updates.

6. Authentication

- **User Management**:

- Firebase Authentication is used for user sign-up, login, and profile management.
- Security rules ensure that authenticated users can only access their data.

6. Code Structure



6.1 Folder Organization

1. ios/android folders:

- Contain Firebase platform-specific configurations for running the app on iOS or Android.

2. pubspec.yaml (Project Metadata):

- Declares dependencies and project settings, indirectly affecting all modules.

3. firebase_options.dart (Firebase Config)

- **Purpose**: Contains Firebase configuration details for initializing Firebase services in the app.
- **Notes**:
  - This file is auto-generated when Firebase is integrated with the project.
  - Includes keys and settings specific to the Firebase project.

4. main.dart (Entry Point)

- **Purpose**: The entry point of the Flutter application.
- **Features**:
  - Initializes the app and its dependencies.
  - Sets up global providers for state management.
  - Configures named routes using routes.dart.

5. routes.dart

- **Purpose**: Manages the app's navigation flow.
- **Features**:
  - Defines named routes for all screens.
  - Centralizes navigation logic, making it easier to update or modify routes.

6. constants

- **Purpose**: Stores static values and configurations used throughout the application.
- **Example**:
  - app_colors.dart: Defines theme colors and ensures consistent styling across the app.

7. components (UI Widgets)

- **Purpose**: Contains reusable UI widgets across multiple screens, ensuring consistency and reducing code duplication.
- **Examples**:
  - **Custom buttons**: custom_button.dart;
  - **Popups**: logout_popup.dart;

- **Form fields**: custom_input_field.dart;
- **Other utilities**: custom_search_bar.dart, expense_item_card.dart;

8. screens (UI Screens)

- **Purpose**: Contains the main UI screens of the application.
- **Files**:
  - home_page.dart: The app's dashboard or landing page.
  - receipt_list_page.dart: Displays a list of receipts.
  - add_update_receipt_page.dart: Allows users to add or edit receipts.
  - budget_page.dart, set_budget_page.dart: Manage and configure budgets.
  - category_page.dart: Handles category management.
  - report_page.dart: Shows expense analysis with charts.
  - settings_page.dart: Provides user settings like currency and profile updates.
  - login_page.dart, signup_page.dart: Authentication pages.
  - forgot_password_page.dart, email_sent_page.dart, verification_link_page.dart: Supporting screens for password recovery and account verification.

9. services (Backend Logic)

- **Purpose**: Handles backend interactions and utility functions.
- **Files**:
  - auth_service.dart: Manages Firebase Authentication operations (e.g., login, signup).
  - budget_service.dart: Handles CRUD operations for budgets.
  - category_service.dart: Manages categories in Firestore.
  - currency_service.dart: Integrates with APIs for currency conversion.
  - receipt_service.dart: Processes receipt data (e.g., storing, retrieving, and OCR parsing).
  - storage_service.dart: Handles Firebase Storage operations, such as saving images.
  - user_service.dart: Manages user profile data in Firestore.

10. providers (State Management)

- **Purpose**: Manages application state using the Provider package.
- **Files**:
  - authentication_provider.dart: Handles user authentication state.
  - budget_provider.dart: Manages budget-related state.
  - category_provider.dart: Manages category data and state.
  - currency_provider.dart: Handles currency settings and conversions.
  - receipt_provider.dart: Tracks receipt-related operations.
  - user_provider.dart: Manages user-specific data like profile information.

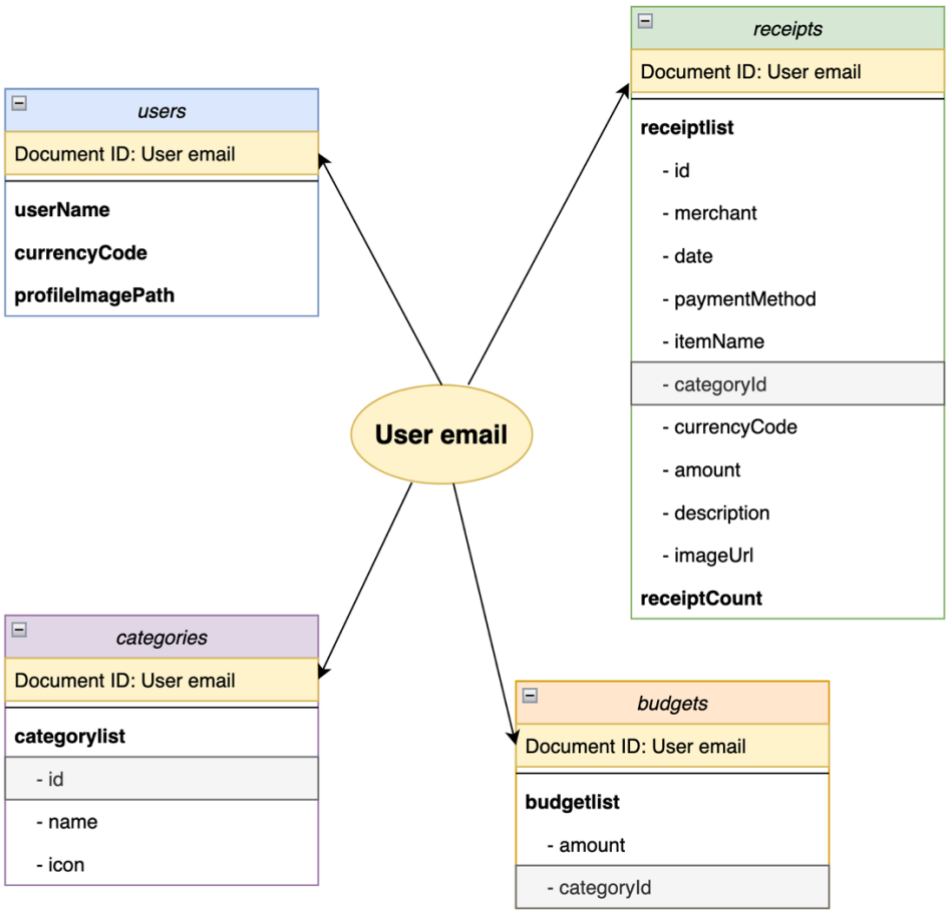
11. logger.dart

- **Purpose**: Manages logging throughout the app.
- **Features**:
  - Uses the logger package for structured logs.
  - Useful for debugging and tracking app activities.

6.2 Notable Patterns

MVC architecture with a Provider-based state management approach.

7. Data Storage Solution



7.1 General Design of Firestore Collection

- **All collections (budgets, categories, receipts, and users) use the user's email as the document ID.**
- **The user email is:**
    - **Retrieved from Firebase Authentication.**
    - **Used as the primary key to link data across collections for each authenticated user.**

7.2 Collection Details

a. users

- **Document ID: User email (e.g., jingjing.yang@tuni.fi)**
- **Fields:**
    - Username: User's display name.
    - profileImagePath: Path or URL to the user's profile picture.

    - currencyCode: User's preferred default currency (e.g., "EUR").
- **Purpose:**
    - Stores general user profile data.

b. categories

- **Document ID: User email (e.g., jingjing.yang@tuni.fi)**
- **Fields:**
    - **categorylist: An array of category objects, each containing:**
        - **id: Unique identifier for the category.**
        - **name: Name of the category (e.g., "Food").**
        - **icon: Visual representation of the category (emoji as string).**
- **Purpose:**
    - Manages expense categories for the user.
- **Relationships:**
    - Referenced by both budgets and receipts through categoryId.

c. budgets

- **Document ID: User email (e.g., jingjing.yang@tuni.fi)**
- **Fields:**
    - **budgetlist: An array of budget objects, each containing:**
        - **amount: Budget amount for a specific category.**
        - **categoryId: Reference to a category in categories.**
- **Purpose:**
    - **Stores user-defined budget limits for specific categories.**
- **Relationships:**
    - **Linked to categories via categoryId.**

d. receipts

- **Document ID: User email (e.g., jingjing.yang@tuni.fi)**
- **Fields:**
    - **receiptCount: Total number of receipts for the user.**

    - **receiptlist: An array of receipt objects, each containing:**
    - id: Unique identifier for the receipt.
    - amount: Value of the transaction.
    - categoryId: Reference to a category in categories.
    - currencyCode: Currency of the transaction (e.g., "EUR").
    - date: Date of the receipt.
    - description: Optional notes or details.
    - imageUrl: Path or URL to the receipt's image.
    - itemName: Name of the item purchased.
    - merchant: Name of the merchant or vendor.
    - paymentMethod: Payment method used (e.g., "PayPal").
- **Purpose:**
    - **Manages transaction records for the user.**
- **Relationships:**
    - Linked to categories via categoryId.

7.3 Collection Relationships

- **The user's email serves as the primary key across all collections.**
- **Each collection (budgets, categories, receipts, users) is independent but logically linked by the user's email, which is consistent with Firebase Authentication.**
- **Data flow:**
    - Collection “categories” provides categoryId used in budgets and receipts.
    - Collection “receipts” track transactions, while budgets enforce category-specific limits.
    - Collection “users“ contains non-functional user information, such as profile details and currency for display (with amount converted).

8. Maintenance

8.1 Responsibilities

Team members manage code updates and app maintenance.

8.2 Installation

For Developers

1. Clone the repository from GitHub.
2. Run flutter pub get to fetch all dependencies.
3. Execute flutter run to build and run the app.
4. Refer to the README.md file in the repository for any additional setup instructions.

For Android Users

1. Download the APK file (app-release.apk) from the submission folder.
2. Enable "Install Unknown Apps" in your Android phone's settings if prompted.
3. Install the APK file and launch the app.

For iPhone Users

- Publishing the App:
  - To officially publish the app or distribute it to other devices, an Apple Developer Program subscription is required ($99/year). This option is not currently implemented.
- Alternative: USB Installation (Free Option):
  1. Connect your iPhone to a macOS computer via USB.
  2. Use Xcode to build and install the app directly on your iPhone.
  3. Trust the developer certificate on your iPhone when prompted.
  4. Note: The app will remain functional for 7 days due to limitations of free provisioning, after which it needs to be reinstalled.

8.3 Emergency Procedures

Restore data from Firestore backups if required.

Restore the codebase from the tagged zip file.

8.4 Modifications

Follow the README.md for adding dependencies or updating features.

9. Instructions for Basic Usage

See the demo video and user manual in the README.md for detailed steps.

10. Missing Features, Known Problems, and Future Development

Missing Features

- **Budget Notifications**: Notify users when they are nearing or exceeding their budgets.
- **Year-to-Year Comparison**: Provide a visual comparison of expenses and savings across different years.
- **Multi-User Account Support**: Enable multiple users, such as family members, to manage accounts together.
- **Batch Delete Limitations**: Currently, users must delete expenses or records one at a time, which can be time-consuming and inefficient for managing multiple entries.
- **Lack of Goal Tracking**: The system does not provide a way for users to set financial goals or track progress, limiting long-term financial planning.

Known Problems

- **No Feedback System**: There is no built-in mechanism for collecting and processing user feedback, making it challenging to address user concerns or improve the application effectively.
- **Limited Responsiveness for Different Screen Sizes**: The application does not fully adapt to various screen sizes, which can lead to a suboptimal user experience on devices with smaller or larger displays.
- **Text Extraction and Matching Limitations:**
    - The text extraction from receipts and matching with receipt fields is currently based on a simple algorithm rather than AI-powered methods. This limits the accuracy and flexibility of field matching.
    - The current method works well for many cases, and **manual adjustments** have been enabled in the app to allow users to correct any discrepancies in the extracted data and ensure accurate field matching.

Future Development

- **Recurring Expenses**: Introduce functionality to schedule and manage recurring expenses, such as monthly bills, to simplify tracking.

- **Data Export**: Allow users to export their financial data in various formats (e.g., CSV, PDF).
- **Enhanced User Experience with Animations and Sound Effects**: Add animations for smooth transitions and interactive button sound effects to improve user engagement and satisfaction.
- **Theme Customization**: Allow users to customize the app's theme, including options for light/dark modes, color schemes, and fonts to enhance personalization.