Implementation Document

for

Brokeo

Version 1.0

Prepared by

Group Number:17 **Group Name: Runtimeзrror**

Anjali Patra	230148	anjalip23@iitk.ac.in
Aujasvit Datta	220254	aujasvitd22@iitk.ac.in
Bhavnoor Singh	230293	bhavnoor23@iitk.ac.in
Darshan Sethia	220326	darshands22@iitk.ac.in
Dhriti Barnwal	230364	dhritib23@iitk.ac.in
Marisha Thorat	230637	marishajt23@iitk.ac.in
Rudransh Verma	230881	rudranshv23@iitk.ac.in
Sanjna S	230918	sanjnas23@iitk.ac.in
Solanki Shrey Jigneshbhai	231017	shreyjs23@iitk.ac.in
Suryansh Verma	231061	suryanshv23@iitk.ac.in

Course: CS253

Mentor TA: Mr. Paras Ghodeshwar

Date: 28 March 2025

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Full Group	First draft of Implementation Document	28/03/25

1 Implementation Details

Programming Languages and Frameworks

1) Frontend (Mobile App):

We used **Flutte**r as the framework for building our mobile application. Flutter is a modern UI toolkit by Google that allows for cross-platform app development. The advantages of using Flutter are:

- Enables building for both Android and iOS using a single codebase, reducing development effort.
- Offers a rich set of pre-designed and customizable widgets for responsive and expressive UIs.
- Compiles directly to native ARM code, ensuring high performance and smooth animations.
- Supports hot reload, allowing us to instantly view changes during development.
- Simplifies UI consistency across devices and screen sizes without platform-specific tweaks.

2) State Management:

To manage state across the app, we used **Riverpod**, a modern and robust state management library for Flutter. We chose Riverpod over other state management solutions due to the following advantages:

- Offers compile-time safety and better type checking, helping catch errors early.
- Supports modular and testable code through the separation of business logic and ui.
- Enables fine-grained listening, which results in performance optimizations.
- Provides flexibility to define both synchronous and asynchronous providers.
- Avoids common pitfalls of earlier solutions like Provider, while maintaining a similar developer experience.

Databases and Backend Services

1) Database (Cloud Firestore):

For storing application data, we used **Cloud Firestore**, a NoSQL cloud-hosted database from Firebase. It offered the following advantages:

- Enables real-time synchronization across all clients, improving responsiveness.
- Maintains a local cache, ensuring the app works even in offline scenarios.
- Automatically handles scaling and performance optimization without manual setup.
- Simplifies integration with Firebase Authentication and other Firebase services.
- Eliminates the need to create a separate local database (like using sqflite) and custom backend (like Django), significantly reducing development complexity.

UI Development

1) Flutter Widgets (Frontend UI):

We built the entire UI using native **Flutter widgets**, which gave us control and flexibility in UI design. The advantages include:

- A comprehensive library of layout and material widgets for building intuitive interfaces.
- Declarative UI programming model makes the UI reactive to state changes.
- Uniform look and feel across different screen sizes and platforms.
- Strong community support and rich documentation.

2 Codebase

Github Repository : https://github.com/Runtime-Error-IITK/brokeo

Code Structure :

```
├── android/
├── assets/
├── build/
├── ios/
├── lib/
└── test/
```

The project follows the standard Flutter directory layout and separates concerns across different platforms and functionalities. The codebase and database interactions are integrated within the Flutter app, with a modular structure that supports platform-specific builds and testing.

Codebase:

The root directory contains the following major folders:

- **lib/**
  Contains the main source code for the application, including the UI, state management (via Riverpod), Firestore interactions, and business logic. This is the core of the app.
- **test/**
  Contains unit and widget tests written for the application. It is used to verify the correctness of individual components and features in isolation.
- **android/, ios/**
  These folders contain platform-specific code and configuration needed to build and run the app on respective platforms. They are mostly auto-generated by Flutter.
- **assets/**
  Stores static assets like images, fonts, and icons used in the application.
- **build/**
  This is a generated folder that contains compiled output and intermediate files. It is managed by Flutter's build system and is not manually edited.

Inside the lib/, The project is divided into two parts, the frontend and the backend based on the model view architecture format.

```
└── lib/
    ├── backend/
    ├── frontend/
    └── main.dart
```

lib/

This is the core directory of the Flutter application where the main logic, UI, models, and data handling are implemented. The folder is organized into logical subdirectories to separate concerns and improve maintainability:

- **`backend/`**
  Contains all backend-related code such as Firestore queries, Firebase integrations, and logic for reading/writing to the database. This includes provider declarations using Riverpod to expose backend data to the UI in a reactive and testable way.
- **`frontend/`**
  Includes all the UI-related code built using Flutter widgets. This folder is structured into screens, components, and layout files that represent the visual parts of the application. It consumes the providers defined in the backend to reflect live data changes.
- **`main.dart`**
  The entry point of the application. It initializes Firebase, sets up global providers, and runs the main app widget. It also handles high-level routing and theming.

The structure of the backend is as follows-

```
└── backend/
    ├── models/
    │   ├── category.dart
    │   ├── due.dart
    │   ├── merchant.dart
    │   ├── schedule.dart
    │   ├── sms.dart
    │   ├── split_user.dart
    │   ├── split_transaction.dart
    │   ├── transaction.dart
    │   └── user.dart
    └── services/
        ├── crud/
        │   ├── database_service.dart
        │   ├── category_service.dart
        │   ├── due_service.dart
        │   ├── merchant_service.dart
        │   ├── schedule_service.dart
        │   ├── sms_service.dart
        │   ├── split_user_service.dart
        │   ├── split_transaction_service.dart
        │   └── transaction_service.dart
        ├── managers/
        │   ├── category_manager.dart
        │   ├── due_manager.dart
        │   ├── merchant_manager.dart
        │   ├── schedule_manager.dart
        │   ├── sms_manager.dart
        │   ├── split_user_manager.dart
        │   ├── split_transaction_manager.dart
        │   └── transaction_manager.dart
        └── providers/
            ├── category_manager_provider.dart
            ├── due_manager_provider.dart
            ├── merchant_manager_provider.dart
            ├── schedule_manager_provider.dart
            ├── sms_manager_provider.dart
            ├── split_user_manager_provider.dart
            ├── split_transaction_manager_provider.dart
            └── transaction_manager_provider.dart
```

- **`models/`**
  This folder defines the data structures used across the application, representing individual entities such as Transaction, Category, Merchant, Due, etc. These are typically Dart classes that map directly to documents stored in Firestore.
- **`crud/`**
  This subdirectory contains all Create, Read, Update, and Delete operations for Firestore collections. Each file in this folder is typically responsible for interfacing with a specific collection (e.g., transactions, categories). This clear separation makes the code easier to maintain and scale as new database features are added.
- **`managers/`**
  Contains utility classes that handle lists of related objects (unlike the `models/` directory, which defines individual data types as singleton-like entities). These manager classes are responsible for managing collections of items in memory—such as lists of transactions, categories, or dues—and provide methods for insertion, deletion, filtering, and other batch operations.
- **`providers/`**
  Includes Riverpod providers that expose the app's state to the frontend. These providers act as bridges between the UI and backend, reacting to changes in the Firestore data or application state. They are designed to be composable, testable, and optimized for performance with features like auto-disposal and selective listening.

The structure of the frontend is as follows-

```
├── assets/
│   ├── icon.png
│   └── category_icons.png/
│       └── contains icons...
└── lib/
    └── frontend/
        ├── home_pages/
        │   └── home_page.dart
        ├── login_pages/
        │   ├── login_page1.dart
        │   ├── login_page2.dart
        │   └── login_page3.dart
        ├── profile_pages/
        │   ├── FAQs_page.dart
        │   ├── budget_page.dart
        │   ├── edit_profile_page.dart
        │   ├── privacy_policy_page.dart
        │   └── profile_page.dart
        ├── transactions_pages/
        │   ├── categories_page.dart
        │   ├── category_page.dart
        │   ├── merchants_page.dart
        │   └── transaction_detail_page.dart
        ├── split_pages/
        │   ├── choose_split_type.dart
        │   ├── choose_transactions.dart
        │   ├── manage_splits.dart
        │   ├── split_between.dart
        │   └── split_history.dart
        ├── analytics_pages/
        │   └── analytics_page.dart
        ├── notification_pages/
        │   └── notification_page.dart
        └── main.dart
```

Database:

```
├── categories/
│   ├── name
│   ├── categoryId
│   └── budget
├── dues/
│   ├── dueId
│   ├── amount
│   ├── merchantId
│   └── categoryId
├── merchants/
│   ├── merchantId
│   ├── name
│   └── categoryId
├── schedules/
│   ├── scheduleId
│   ├── amount
│   ├── merchantId
│   ├── categoryId
│   ├── timePeriod
│   └── dates
├── transactions/
│   ├── transactionId
│   ├── amount
│   ├── date
│   ├── merchantId
│   ├── categoryId
│   └── smsID
└── split_transactions/
    ├── transactionId
    ├── amount
    ├── date
    ├── merchantId
    ├── categoryId
    ├── smsID
    └── splitUsers
```

Our application uses **Cloud Firestore** as its database, structured as a set of collections with documents representing entities in the app. Below is an overview of each collection and its fields:

- **categories :** Represents different spending categories defined by the user (e.g., Food, Utilities). Each document contains:
  - name : The name of the category.
  - categoryId : A unique identifier for the category.
  - budget : The monthly budget assigned to the category.
- **dues :** Stores dues that need to be settled with merchants. Each document includes:
  - dueId : A unique identifier for the due.
  - amount : The outstanding amount.
  - merchantId : A reference to the associated merchant.
  - categoryId : A reference to the related category.
- **merchants:** Stores details about merchants involved in transactions. Each document includes:
  - merchantId : A unique identifier for the merchant.
  - name : The merchant's name.
  - categoryId : A reference to the default category associated with this merchant.
- **schedules:** Used to manage recurring or scheduled payments. Each document contains:
  - scheduleId : A unique identifier for the schedule.
  - amount : The recurring payment amount.
  - merchantId : The merchant involved in the schedule.
  - categoryId : The relevant category for the scheduled expense.
  - timePeriod : Frequency of the schedule (e.g., weekly, monthly).
  - dates : List of due dates or upcoming payment instances.
- **transactions :** Stores all regular (non-split) transaction records. Each document includes:
  - transactionId : A unique identifier for the transaction.
  - amount : The amount of the transaction.
  - date : The date of the transaction.
  - merchantId : The merchant involved in the transaction.
  - categoryId : The category the transaction falls under.
  - smsID : (Optional) Identifier linking the transaction to a parsed SMS.
- **split_transactions :** Represents transactions that are split among multiple users. Each document includes:
  - transactionId : A unique identifier for the transaction.
  - amount : The total amount of the transaction.
  - date : The date the transaction occurred.
  - merchantId : Reference to the merchant involved.
  - categoryId : Reference to the category.
  - smsID : (Optional) SMS-based identifier for tracking.
  - splitUsers : A list of references to documents in the users collection, indicating with whom the transaction is shared.

3 Completeness

Implemented Features

1. Sign-up/Sign-in

- Users can sign in using their phone number and OTP (One-Time Password).
- Ensures smooth multi-device access by securely managing user sessions.

2. Features Implemented in Home Section

This section is structured to provide users with clear insights and easy navigation:

- **Monthly Spending Overview** – Displays the total amount spent in the current month.
- **Transactions** – Lists all recent transactions, categorized for easy tracking.
- **Categories** – Highlights spending across different categories with a breakdown of expenses.
- **Scheduled Payments** – Shows upcoming bill payments and subscriptions.
- **Split Expenses** – Provides an overview of shared transactions, balances, and pending settlements.
- **Budget Tracking** – Displays set budgets and their usage with alerts for approaching limits.
- **Notification Section** – Lists down all the notifications received.

3. Features Implemented in Transactions Section

- This page shows the amount spent in a particular month and how much users can spend in the future based on their budget.
- It contains three sections:
  - **Transactions** – Displays all recent transactions with date, merchant, and other details. Users can also add a new transaction.
  - **Categories** – A pie chart showing the top spending categories, along with category-wise spending amounts and their limits. Clicking on a category provides a detailed breakdown of spending in that category. Users can also add new categories.
  - **Merchants** – Provides a merchant-wise breakdown of money spent and the number of times money was spent at each merchant. Clicking on a merchant displays detailed spending insights.

4. Features Implemented in Analytics Section

- **Spending Trends** – Users can view expenses on daily, weekly or monthly intervals through bar graph (for spends) and line chart (for transactions) of both inflow and outflow.
- **Data Export** – Users can share their financial data as a csv file through WhatsApp or other platforms.

5. Features Implemented in Split Section

- **Summary Overview** – Displays the total balance, total owed, and total are owed amounts.

- **Split History** – Shows a list of all users involved in shared transactions. Clicking on a user displays:
    - All past splits with that user.
    - An option to settle up the split and update the transaction status.
- **Add Split** – Users can create new shared expenses and assign contributors.

6. Features Implemented in Profile Section

- **Personal Details:** Users can manage their name, mobile number, and email.
- **Permissions Management:** Users can view and manage their app permissions (e.g., SMS access, notifications).
- **Budget Settings:** Users can set and modify their total budget and category-wise budgets.
- **Help & Support:** Provides access to FAQs, help resources, and app information.

Future Development Plans

To enhance user experience and expand functionality, we plan to introduce several improvements to Brokeo. Our focus will be on making financial management more efficient and insightful.

1. Feature Enhancements

- Integrate AI-driven financial insights to provide personalized spending analysis and smart budgeting sugggestions.
- Enhance SMS detection and transaction categorization to minimize manual data entry.
- Introduce OCR-based receipt scanning for effortless expense logging.
- Implement savings and investment tracking to help users monitor financial growth, interest earnings, and debts.

2. UI & UX Improvements

- Introduce dark mode and custom themes for a personalized app interface.
- Enhance the dashboard with interactive visual reports and financial health insights.
- Integrate voice command support, allowing users to log expenses and check balances hands-free.

3. Performance & Backend Enhancements

- Enhance data encryption and introduce multi-factor authentication for added protection.
- Improve offline functionality with auto-sync features, allowing expense tracking even without an internet connection.

4. Integration & Expansion

- Integrate bank accounts and payment gateways for real-time transaction updates.
- Allow financial data exports to Excel, Google Sheets, and accounting software.
- Introduce a community feature, where users can share budgeting strategies and financial tips.

With these improvements, we plan on making Brokeo a smarter way to manage finances. Our goal is to create a tool that not only helps users track their spending but also makes budgeting and expense sharing effortless. As we continue to refine Brokeo, we'll listen closely to user feedback, ensuring that every update brings real value and makes financial management smoother.

Appendix A - Group Log

Date	Timings	Duration	Minutes
12th Feb	10:00-11:00	1hr	● Discussed the basic framework of the documents, the components to be present and divided it individually amongst ourselves
15th Feb	10:00-11:00	1hr	● Decided database schema as well as backend technology to use
28th Feb	16:00-18:30	2hrs 30mins	● Built individual components ● Discussed on how to properly combine those components
10th Mar	21:00-23:00	2hrs	● Revisions of the components and finalisation of frontend
17th Mar	20:00-01:00	5hrs	● Working on backend ● Creation of riverpod providers
21st Mar	20:00-23:00	3hrs	● Completion of frontend ● Work on backend and its integration
24th Mar	20:00-21:30	1hr 30mins	● Started work on the Implementation Document and split up the work ● Discussed minor changes to the application and its beautification
27th Mar	21:00-00:00	3hrs	● Completion of the Implementation Document ● Completion of application
28th Mar	15:00-16:00	1hr	● Final Proofread of the Document