ThiruEadu - Complete Working Documentation

1. Purpose and Scope

This document explains the complete working of the ThiruEadu personal finance desktop app, including:

- system architecture
- code-level module responsibilities
- frontend to backend interactions
- transaction and debt workflows
- database structure with ER diagram
- implementation notes from project exploration
- detailed application screenshots

This documentation is designed for handover, technical review, and future maintenance.

2. Product Summary

ThiruEadu is a local-first personal finance and debt-ledger application built with React + Tauri + Rust + SQLite.

Core capabilities:

- multi-account tracking (Bank, Cash, Wallet)
- person-centric debt ledger (virtual asset/liability accounts)
- transaction system covering expense, income, transfer, debt, settlement
- analytics dashboard for spending, income, payment modes, and people balances
- computed account balances from transactions (single source of truth)

3. Technology Stack

- Frontend: React 19, TypeScript, Vite
- State Management: TanStack React Query
- Desktop Bridge: Tauri v2 (`invoke` command bridge)
- Backend: Rust with service/repository layering
- Storage: SQLite (integer money in paise)
- Charts: Recharts

4. Architecture Overview

Architecture Overview

4.1 Layered Design

1. UI Components render views and collect user input.
2. Query hooks fetch and mutate data with cache invalidation.
3. `src/db.ts` transforms payloads (rupees to paise) and invokes Rust commands.
4. Rust handlers receive command calls.
5. Service layer validates and enforces business rules.
6. Repository layer performs SQL operations.
7. SQLite persists immutable financial records.

4.2 Key Design Principle

Balances are not stored in the `accounts` table. They are computed from transaction history:

$$ \text{balance(account)} = \sum \text{credits} - \sum \text{debits} $$

where:

- credits: rows where `dest_account_id = account.id`
- debits: rows where `source_account_id = account.id`

5. Codebase Exploration Summary

5.1 Frontend Core Files

- `src/App.tsx` : root state machine (views, filters, navigation, modals)
- `src/components/Dashboard.tsx` : analytics tabs and charts
- `src/components/AddTransactionModal.tsx` : expense/income, transfer, people tabs
- `src/components/PeopleView.tsx` : person cards and per-person ledger
- `src/components/AddAccountView.tsx` : account creation + linked payment modes
- `src/components/CategoriesView.tsx` : category management
- `src/components/PaymentModesView.tsx` : payment mode management
- `src/hooks/useQuery.ts` : query keys, fetch hooks, mutation hooks
- `src/db.ts` : typed bridge to Rust commands + amount conversion
- `src/calculations.ts` : financial analytics and ledger calculations

5.2 Backend Core Files

- `src-tauri/src/handlers/mod.rs` : Tauri command handlers
- `src-tauri/src/service/mod.rs` : validation and business logic
- `src-tauri/src/repository/mod.rs` : SQLx repositories
- `src-tauri/src/repository/database.rs` : schema setup + foreign key pragma
- `src-tauri/src/models/mod.rs` : request/response domain models

5.3 Current View State Machine

Views currently implemented in app flow:

- home
- transactions
- stats
- accounts
- people
- person-ledger
- add-account
- add-transaction
- categories
- payment-modes
- add-person

6. Data Model and ER Diagram

6.1 Main Tables

- `accounts` : real and virtual accounts
- `transactions` : immutable ledger entries with amount in paise
- `categories` : expense/income category definitions
- `payment_modes` : UPI/Cash/Card style mode entities
- `account_modes` : account to payment mode mapping
- `account_categories` : account to category mapping
- `audit_logs` : operation audit trail
- `budgets` : reserved table for budget module

6.2 Money Precision Rule

All monetary values are stored as integer paise in backend/database. Frontend display conversion is done only at render/boundary layer.

7. Transaction Dataflow and Interaction

7.1 Request Interaction Path

1. User submits transaction form in `AddTransactionModal`.
2. Frontend converts amount from rupees to paise in `src/db.ts`.
3. Frontend calls `invoke('create_transaction', { request })`.
4. Rust service validates:

- amount > 0
- account references valid
- transaction structure valid for type

5. Repository inserts immutable row in `transactions`.
6. React Query invalidates and refetches `accounts` + `transactions`.
7. Dashboard and lists update with recomputed balances.

7.2 Interaction Modes in App

- Expense/Income: debit/credit to real accounts
- Transfer: money movement between real accounts
- People flows:
- `DEBT_GIVEN`
- `DEBT_TAKEN`
- `DEBT_SETTLED_RECEIVE`
- `DEBT_SETTLED_PAY`

8. Business Workflows

8.1 Account Creation Workflow

1. User creates account in Add Account view.
2. Optional opening balance is accepted.
3. Account is stored, links to category and mode mappings are saved.
4. Opening balance path is represented by transaction logic (no direct balance write).

8.2 Expense Workflow

1. Pick account, category, mode, date, amount, title.
2. Submit and persist transaction.
3. Recomputed balances are visible across Home and Accounts.

8.3 People Ledger Workflow

1. Create person account (`VIRTUAL_ASSET` or `VIRTUAL_LIABILITY`).
2. Record lent/borrowed transactions with or without cash movement.
3. Settle partially or fully.

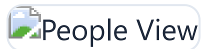
4. Person ledger shows running balance and settlement direction.

9. Screenshots and UI Exploration

9.1 Home Dashboard

9.2 Transactions View

9.3 People View



9.4 Person Ledger View

9.5 Analytics Overview

9.6 Analytics - Spending Tab

9.7 Analytics - Modes Tab

9.8 Accounts View

9.9 Categories Management View

9.10 Add Transaction - Expense/Income

9.11 Add Transaction - Transfer

9.12 Add Transaction - People

10. How We Interact with the App Internally

The app uses a strict interaction contract between layers.

- UI never writes directly to SQLite.
- UI calls typed helper functions in `src/db.ts`.
- `src/db.ts` maps frontend fields to backend command payloads.
- Tauri command handlers convert payloads into service calls.
- Services validate business constraints and route to repositories.
- Repositories execute SQL using parameterized queries.
- Query cache invalidation keeps UI state synchronized after mutations.

This pattern allows clear ownership:

- Presentation in React
- Rules in Rust service layer
- Persistence in repository/database layer

11. Security and Integrity Notes

- integer money handling avoids floating-point errors
- foreign keys enforced (`PRAGMA foreign_keys = ON`)
- no direct balance mutation; balances derived from ledger history
- SQL is parameterized through SQLx binding

- audit log table supports operation traceability

12. Reproducibility (Documentation Assets)

All generated assets are inside `project_documentation`.

Contents:

- `project_documentation/ThiruEadu_Complete_Documentation.md`
- `project_documentation/ThiruEadu_Complete_Documentation.pdf`
- `project_documentation/diagrams/*.svg`
- `project_documentation/screenshots/*.png`

Automation scripts:

- `scripts/capture-doc-screenshots.mjs`
- `scripts/build-documentation-pdf.mjs`

13. Conclusion

This project already contains strong architectural patterns for a fintech-grade local ledger:

- precise money model
- immutable transaction ledger
- derived balances
- layered command architecture
- clear UI separation

The documentation package provides both technical explanation and visual evidence (screenshots + vector diagrams) for full project exploration and handoff.