RainyDays

Emergency Fund Manual Tracker

**Background**

Managing shared emergency savings across individuals, families, or small groups is often prone to error when using spreadsheets or ad-hoc notes. A mobile-first app that supports manual, transparent, and collaborative tracking with real-time sync and offline-first capabilities will reduce friction and increase trust among participants.

This design targets working individuals (18+), families, and small saving groups who want a private, manual tracker for joint emergency funds without bank integrations (at least for now).

**Requirements**

**Must Have**

* User Registration & Profiles – secure accounts, profile data, avatars.
* Emergency Fund Creation & Participation – create multiple funds, invite participants (keeper, contributor)
* Manual Transactions – deposits, withdrawals, edit with notes and categories
* Group Statistics – Member rankings who contributed the most, progress, most active, and total this month
* History – All activities like withdraw/deposit
* Goal Tracking – target amount, dynamic progress visualization
* Offline-first – local write queue, sync-on-contact, conflict handling
* PIN/Biometric Lock – device-level quick lock for app.

**Should Have**

* Push / in-app notifications (contribution reminders, low-balance alerts, group member contributes)

**Could Have**

* Expense Categorization and tagging
* Savings insights / tips
* Theming / Dark mode
* Bank integration
* Investment features

**Method**

**High-level approach**

* Mobile client: React Native (Typescript). Use react-query (TanStack Query) for REST data caching and a Socket.IO client for real-time events and presence.
* Backend: Spring Boot exposing REST APIs + WebSocket. Server is authoritative: client send events, server validates and persists.
* Database: PostgreSQL. Use JPA and Liquibase.
* Real-time scaling: Redis for pub/subs and session store so multiple backend instances can broadcast to connected clients,
* Offline + Sync: local persistent store (SQLite via react-native-sqlite-storage). Each change includes a UUID and client timestamp for idempotency.
* Conflict resolution: server applies operations in arrival order; operations are idempotent via client-sent UUIDs and server-granted sequence IDs. For conflicting edits (two edits to same transactions), server produces a merge policy: last-write-by-server-timestamp with audit trail and optionally store both versions.

**Offline / Sync strategy**

* Client stores a local write-head queue (WAQ) of actions with clientUuid + client timestamp.
* On reconnect, client sends /sync with sequential changes: server checks for duplicates using clientUuid and returns canonical objects and any conflicts.
* Conflict policy: server accepts operations in arrival order. If two edits change same transaction field, server will keep the later server timestamp version and store the previous in events for audit. For critical conflicts deletes vs edits, server returns conflict response with both versions and a conflict flag; client UI shows “conflict detected” allowing the owner to choose resolution.

**Idempotency & Ordering**

* Every client-generated action includes clientUuid to prevent double-applying.
* Server assigns sequence\_id (or uses event id) for total ordering for the fund’s event stream.

**Security**

* Authentication: JWT tokens (short-lived access, long-lived refresh). Socket.IO will use the JWT on connect.
* Authorization: verify user is a participant of fund for fund-scoped actions.
* Transport: TLS for REST and WSS for sockets.
* Rate limiting & Abuse prevention per IP. 50 request per minute.
* Data privacy: Zero-knowledge not required (since app plain amounts), but encrypt sensitive fields at rest if desired.

**Gathering Results**

**Key metric to measure success** – Active funds created / month – DAU/MAU for participating users - % of transactions created while offline (indicator of offline usage) – Mean time to sync on reconnect – Conflict rate (how often server returns conflicts) – Crash/error rates in mobile app

**Validations steps post-lunch** – Run a usability study with 10-20 families to observe flows around contributions and conflict resolution. – Load test backend with simulated multi-user fund activity to ensure WebSocket throughput and Redis pub/sub scales. – Monitor for data-loss bugs via end-to-end tests.