Architecture & Reasoning

a) System Overview

Nissmart is a micro-savings ledger platform exposing both a user-facing dashboard and an admin operations console. Users can select or create wallets, view their balances, and execute deposits, transfers, and withdrawals through RESTful endpoints. An admin summary endpoint aggregates wallet value, transaction counts, and volumes so the operations team can monitor risk and liquidity. Every write path uses an idempotency service to keep finances safe even under retries.

b) Architecture Components

Component	Responsibility
API Service (FastAPI)	Orchestrates endpoints under `/api`, handles validation, and wires idempotency across ledger operations. Each request runs through SWR-friendly fetchers on the frontend.
Database (SQLite)	Tracks Users, Accounts, Transactions, and ledger entries. Async SQLAlchemy sessions are shared via dependency injection.
Ledger/Transaction tables	Central `Transaction` model records deposits/transfers/withdrawals; ledgers are updated atomically via `LedgerService` and validation routines in `TransactionService`.
Dashboard service	The `/api/dashboard/admin` endpoint aggregates counts and amounts (deposits/transfers/withdrawals) and feeds the admin dashboard which renders KPIs, breakdown bars, and sparkline charts.
Background processing (Idempotency)	`IdempotencyService` caches operation outcomes, enforces lock acquisition, and prevents repeated writes when the same key/retry pair occurs.

c) Data Model Rationale

- **Users** store identity, email, and activation state. They drive ownership for accounts and transactions.

- **Accounts/Wallets** are tied to users (including treasury/escrow) with precise `Currency` enums. Balances include real and available amounts so withdrawals can guard funds.
- **Transactions** are typed (`deposit`, `transfer`, `withdrawal`) and store status, currency, amount, reference, and optional metadata. Indexed by user/account IDs and timestamp for history queries.
- **TransferRequests / Withdrawals** are modeled through `Transaction` rows with contextual fields (source/destination) rather than standalone tables; this keeps history centralized and simplifies reporting.

This structure prioritizes a single ledger table per operation, making it easy to sum balances per currency and preventing duplication of transactional data.

d) Transaction Safety

1. **Avoiding negative balances:** Both `LedgerService` and the validation layer check available balance before approving a withdrawal/transfer. Transfers debit the source account first and only credit once the debit is validated.
2. **Avoiding double-spend:** Idempotency keys block duplicate POSTs by caching responses (status and body) and returning them if the same key arrives again.
3. **Ensuring atomicity:** Database transactions wrap ledger updates so debit and credit operations happen together; any exception triggers rollback.
4. **Idempotency:** The frontend's `apiFetch` wrapper generates an `Idempotency-Key` for each mutation, while backend `IdempotencyService` enforces locks and caches the result keyed on the payload hash.
5. **Validation:** Inputs (amount > 0, destination user choices, currency selection) are validated both client-side (operations form) and server-side (Pydantic schemas) before processing.

e) Error Handling

- **Failed transfers:** The application rolls back the DB transaction and surfaces a 4xx or 5xx error message; the frontend shows a toast with the backend detail.
- **Failed withdrawals:** Similar path—validation rejects insufficient balance and the response explains the rejection; retries are safe because idempotency prevents duplicate debits.
- **Unexpected backend errors:** Global exception handlers bubble a clear JSON error; frontend notifications plus console logging help surface issues during demos.

f) Assumptions & Trade-offs

- SQLite suffices for this prototype, acknowledging a production deployment would use a more robust RDBMS.

- A single ledger table simplifies reporting but requires careful column naming to distinguish operation context.
- Idempotency and validation logic run inline rather than via background jobs; this keeps latency predictable for the test.