**Prompt:** Based on this requirements document, create a detailed technical design document covering system components, data models, integration points and testing(along with test cases). Do not include any unnecessary things other than the stories mentioned in the provided document.

**Technical Documentation**

**Overview**

The system delivers an admin-only, secure banking back-office with SSO-first authentication (JWT fallback), account management, transaction history with filters, cash deposits/withdrawals, and intra-bank transfers with validations and idempotency, over HTTPS with clear, non-technical error responses per the requirements and acceptance criteria.​

**Technology stack**

* Backend
  + Python 3.12 with FastAPI for stateless REST APIs, leveraging async I/O and Pydantic validation to enforce banking rules and input constraints across deposits, withdrawals, and transfers in line with v1 scenarios.​
  + Authentication via SSO as default and JWT fallback, ensuring all operations are restricted to authenticated ADMIN access as mandated in scope and user stories.​
  + PostgreSQL with SQLAlchemy; transactional boundaries guarantee atomic postings and prevent partial transfers, matching reliability non-functional requirements.​
  + Idempotency enforced for transfers at the API layer with persisted unique references for traceability and duplicate-prevention, satisfying acceptance criteria.​
  + Centralized error handling returns clear codes and human-readable messages while avoiding technical leakage per Error Visibility story.​
* Frontend
  + React 18 with Vite and TypeScript to implement the admin console: dashboard listing, search, account detail, and forms for deposit, withdrawal, and transfer per v1 features.​
  + React Router for navigation and React Query for data fetching with consistent error states and authentication propagation in SSO-first and fallback login paths per authentication scenarios.​
  + UI supports filters on accounts and transactions and displays amounts, dates, and references as required by acceptance criteria.​
* Conventions
  + JSON over HTTPS for all endpoints; monetary values as fixed-precision decimals; single currency per account enforced on transfers; unique references and idempotency keys stored as opaque strings per NFRs and constraints.​

**Architecture overview**

* A three-tier, stateless service exposes secure admin APIs for authentication, accounts, transactions, cash operations, and transfers, aligning to the admin-only v1 scope with HTTPS-only transport and reliability safeguards against duplicates or partial postings.​
* Core services: Auth (SSO, JWT fallback), Accounts (CRUD-limited to create and status updates), Transactions (history and filters), Posting Engine (deposit, withdrawal, transfer with validations, idempotency), and Query APIs (dashboard, search).​
* Non-functional anchors include security via HTTPS, traceability with unique transaction references, and user-friendly error handling without technical leakage, matching NFRs in the requirements.​

**Components**

* API Gateway and Auth:
  + Enforces SSO as default and JWT fallback, requiring authenticated admin access for all operations.​
  + Issues short-lived access tokens for session continuity in the admin UI while preserving single-role ADMIN constraints.​
* Accounts Service:
  + Provides searchable dashboard list with number, owner, balance, and status; supports creation and status updates (Active/Frozen).​
  + Blocks postings when Frozen, satisfying the user story to prevent new transactions in Frozen state.​
* Transactions Service:
  + Returns transaction history per account with filters by date, type, and amount; includes date, type, amount, and reference fields.​
* Posting Engine:
  + Cash deposit: credits account and records “Deposit” transaction; cash withdrawal: debits when sufficient balance, else error.​
  + Transfer: validates same currency, distinct accounts, sufficient source balance, and idempotency reference to avoid duplication; posts matching debit/credit entries.​
* Idempotency and Consistency:
  + Idempotency keys required for transfers; uniqueness of transaction reference ensures traceability and duplicate prevention per NFRs and ACs.​
  + Atomic posting ensures no partial postings on multi-step transfers, honoring reliability constraints.​

**Data model**

* Account
  + account\_id (UUID), account\_number (string unique), owner\_name (string), status {Active|Frozen}, currency (ISO code), balance (decimal, 2 dp), created\_at, updated\_at ​.
* Transaction
  + txn\_id (UUID), account\_id (FK), timestamp, type {Debit|Credit|Deposit|Withdrawal|TransferDebit|TransferCredit}, amount (decimal, 2 dp), reference (unique string), related\_account\_number (nullable for transfers), notes (nullable) ​.
* Transfer
  + transfer\_id (UUID), source\_account\_id (FK), dest\_account\_id (FK), amount (decimal), currency (ISO), idempotency\_key (unique), status {Pending|Posted|Rejected}, created\_at ​.
* Auth Session
  + session\_id (UUID), admin\_id, method {SSO|JWT}, issued\_at, expires\_at, token\_hash ​.
* Constraints and rules
  + Account.status=Frozen blocks new postings; all monetary ops require same currency and sufficient balance; unique reference or idempotency is mandatory for transfers and used for traceability and duplicate prevention.​

**API surface**

* Auth
  + POST /auth/sso/callback: exchanges SSO assertion and issues session; denies invalid assertions.​
  + POST /auth/login: JWT fallback; rejects invalid credentials; both paths yield admin access to dashboard on success.​
* Accounts
  + GET /accounts?query=...&status=...: returns searchable list for dashboard with number, owner, balance, status.​
  + GET /accounts/{id}: details with transactions link; used by “View Account Detail” story.​
  + POST /accounts: creates account; optional initial deposit handled by posting engine; default status Active.​
  + PATCH /accounts/{id}/status: sets Active/Frozen; Frozen blocks new transactions thereafter.​
* Transactions
  + GET /accounts/{id}/transactions?from=...&to=...&type=...&min=...&max=...: filtered history; each shows date, type, amount, reference.​
* Cash operations
  + POST /accounts/{id}/deposit: body {amount, reference}; increases balance and records Deposit.​
  + POST /accounts/{id}/withdraw: body {amount, reference}; requires sufficient funds; else error; records Withdrawal.​
* Transfers
  + POST /transfers: body {source\_account\_id, dest\_account\_id, amount, currency, idempotency\_key}; validates currency, not same account, sufficiency, and duplicates; posts mirrored entries with a unique reference; rejects duplicates and invalid cases per ACs.​
* Errors
  + All endpoints return consistent messages with a clear code and human-readable explanation, avoiding technical leakage, conforming to the Error Visibility story.​

**Posting logic**

* Deposit
  + Preconditions: account Active; amount > 0.​
  + Effects: balance += amount; record Credit with type Deposit and unique reference; immediate reflection on balance.​
* Withdrawal
  + Preconditions: account Active; sufficient balance; amount > 0.​
  + Effects: balance -= amount; record Debit with type Withdrawal; insufficient funds yields clear error without partial effects.​
* Transfer
  + Preconditions: different accounts; same currency; sufficient source balance; unique idempotency reference.​
  + Effects: atomic pair posting: source Debit (TransferDebit) and destination Credit (TransferCredit) with linked reference; idempotency protects against retries.​

**Integration points**

* SSO Provider
  + Default authentication path; on outage, fallback login enabled; both satisfy User Story 1 scenarios for SSO success, fallback acceptance, and rejection on invalid credentials.​
* None external for payments or mobile apps
  + v1 excludes third-party integrations and mobile; all operations are internal to the admin system, matching scope.​

**Security and access control**

* HTTPS everywhere with authenticated ADMIN-only access for all operations, reflecting NFRs and constraints; no multi-role features in v1.​
* Error responses avoid technical details while returning clear codes and messages, fulfilling the Error Visibility story.​

**Testing strategy**

* Levels
  + Unit tests for services: validation, posting rules, balance math, and status gating.​
  + Integration tests for API contracts and transactional atomicity in deposits, withdrawals, and transfers.​
  + Acceptance tests mapped to Gherkin scenarios across authentication, accounts, transactions, cash ops, transfers, and errors.​
* Data setup
  + Seed accounts across Active/Frozen, sufficient/insufficient balances, and uniform currency to cover positive and negative paths per user stories.​

**Test cases**

* Authentication
  + SSO success grants dashboard; invalid SSO denied; when SSO unavailable, valid fallback credentials accepted; invalid fallback denied.​
* Dashboard and search
  + Authenticated admin loads list with balances; filters by name/number/status return matching accounts only.​
* Account detail
  + Selecting an account shows full detail and transaction history with amounts and dates.​
* Create account
  + Submitting required details creates account with default Active status; optional initial deposit creates a Deposit transaction and updates balance.​
* Update status
  + Changing to Frozen blocks new postings with clear error; changing to Active re-enables normal activity; attempting operations while Frozen is rejected.​
* Transaction history filters
  + Date, type, and amount filters return only matching transactions; each item includes date, type, amount, and reference fields.​
* Deposit
  + Valid deposit increases balance by the amount and records a Deposit transaction; negative or zero amount rejected with user-friendly error.​
* Withdrawal
  + Valid withdrawal on sufficient balance decreases balance and records Withdrawal; insufficient balance blocks the withdrawal with error; negative or zero amount rejected.​
* Transfer
  + Valid transfer between distinct same-currency accounts updates both balances accurately with paired entries; duplicate idempotency key is rejected as duplicate; mismatched currency or same account raises validation error; insufficient source balance rejected with no partial effects.​
* Error visibility
  + Any failed operation returns a clear code and explanation without technical data; messages are consistent across endpoints per NFRs.