Functional High-Level Design (HLD) for Apartment Complex Account Management System

1. Introduction & Scope

Overview

The Apartment Complex Account Management System is designed to provide comprehensive financial management capabilities for apartment complexes, enabling tracking of income, expenses, assets, and liabilities. The system will ensure accurate financial record-keeping, prevent overdrafts, and generate essential financial reports.

Key Business Goals

- Streamline financial operations for apartment complex management
- Maintain accurate and real-time account balances
- Prevent transactions that would result in negative cash/bank balances
- Generate actionable financial reports for decision making
- Ensure data integrity and security of financial information

Functional Requirements Coverage

- Account creation and management (Income, Expense, Asset, Liability)
- Transaction recording with debit/credit journal system
- Balance tracking and profit/loss calculation
- Comprehensive reporting capabilities
- Negative balance prevention

Non-Functional Requirements

- Performance: <2s response for key operations
- Security: Role-based access control and audit logging
- Usability: Intuitive interface with clear error messaging
- Data Integrity: Immutable transaction records

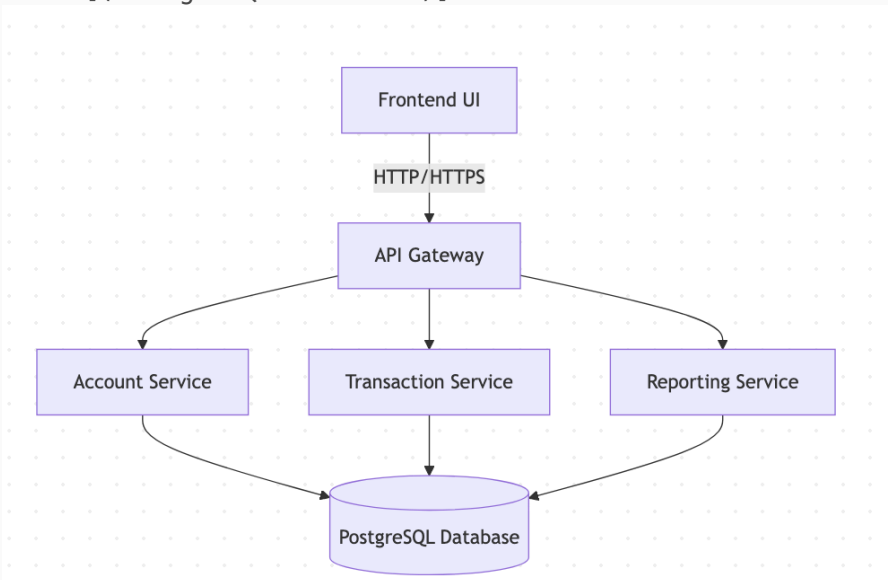
2. System Architecture

High-Level Modules/Components

![System Architecture Diagram]

text

```
mermaid
graph TD
    A[Frontend UI] -->|HTTP/HTTPS| B[API Gateway]
    B --> C[Account Service]
    B --> D[Transaction Service]
    B --> E[Reporting Service]
    C --> F[Database]
    D --> F
    E --> F
    F[(PostgreSQL Database)]
```



Component Breakdown

1. **Frontend Layer**
    - React-based responsive web application
    - State management using Redux
    - Formik for form handling
    - Chart.js for data visualization
2. **API Layer**
    - RESTful API Gateway (Node.js/Express)

    - JWT authentication
    - Request validation
    - Rate limiting
3. **Service Layer**
    - Account Service: CRUD operations for accounts
    - Transaction Service: Journal entry processing, balance validation
    - Reporting Service: Financial report generation
4. **Data Layer**
    - PostgreSQL relational database
    - Redis cache for frequently accessed data
    - Backup and replication configuration

Interaction Flow

- Client makes request → API Gateway authenticates and routes → Appropriate service processes request → Database operation → Response returned to client

Deployment View

- **Cloud Infrastructure**: AWS EC2 instances
- **Containerization**: Docker containers managed via Kubernetes
- **CI/CD**: GitHub Actions pipeline with automated testing
- **Monitoring**: Prometheus + Grafana dashboard

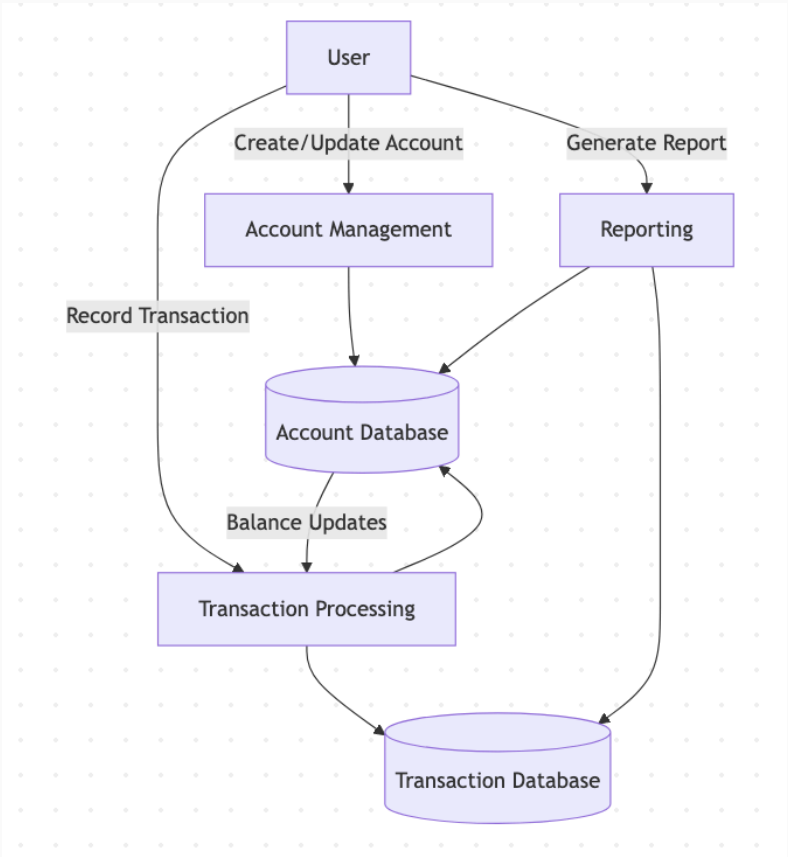
3. Data Flow & Process Diagrams

Level-1 Data Flow Diagram (DFD)

text

```
mermaid
flowchart TD
    A[User] -->|Create/Update Account| B[Account Management]
    A -->|Record Transaction| C[Transaction Processing]
    A -->|Generate Report| D[Reporting]
    B --> E[(Account Database)]
    C --> E
    C --> F[(Transaction Database)]
```

```
    D --> E
    D --> F
    E -->|Balance Updates| C
```



Key Process Flows

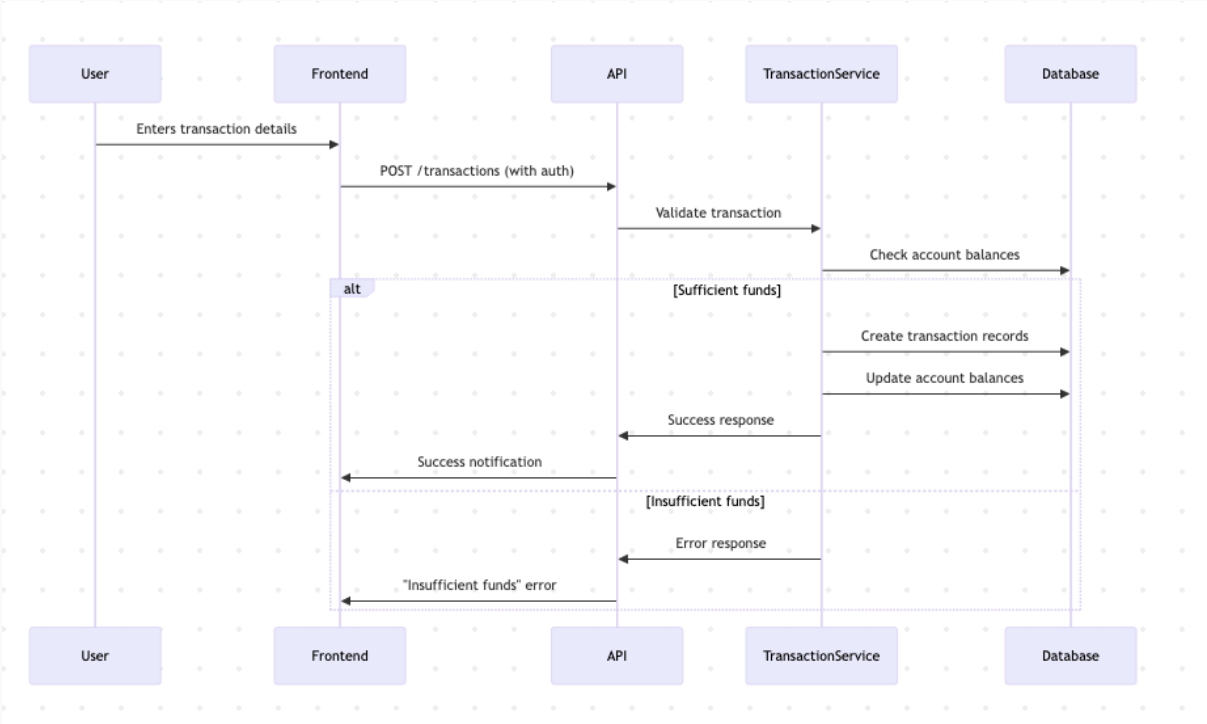
Transaction Recording Sequence

text

```
mermaid
sequenceDiagram
    participant User
    participant Frontend
    participant API
    participant TransactionService
    participant Database

    User->>Frontend: Enters transaction details
    Frontend->>API: POST /transactions (with auth)
    API->>TransactionService: Validate transaction
    TransactionService->>Database: Check account balances
    alt Sufficient funds
        TransactionService->>Database: Create transaction records
        TransactionService->>Database: Update account balances
```

```
        TransactionService->>API: Success response
        API->>Frontend: Success notification
    else Insufficient funds
        TransactionService->>API: Error response
        API->>Frontend: "Insufficient funds" error
    End
```



Monthly Profit/Loss Calculation Flow

1. Reporting service queries all income transactions for month
2. Reporting service queries all expense transactions for month
3. System sums amounts for each category
4. Calculates net profit/loss (Income - Expenses)
5. Formats results for display

4. Technology Stack & Design Choices

Frontend

- **Framework**: React.js (with TypeScript)
- **State Management**: Redux Toolkit
- **UI Components**: Material-UI
- **Form Handling**: Formik with Yup validation

- **Charting**: Chart.js
- **Justification**: Robust ecosystem, strong typing, reusable components

Backend

- **Language**: Node.js (Express)
- **API Design**: RESTful principles
- **Authentication**: JWT with refresh tokens
- **Caching**: Redis for frequently accessed data
- **Justification**: JavaScript full-stack consistency, strong middleware ecosystem

Database

- **Primary**: PostgreSQL
- **Schema**: Relational design with ACID compliance
- **Backup**: Daily snapshots + WAL archiving
- **Justification**: Financial data requires strict consistency, complex queries

Infrastructure

- **Web Server**: Nginx (reverse proxy + load balancing)
- **Containerization**: Docker
- **Orchestration**: Kubernetes (for scaling)
- **CI/CD**: GitHub Actions
- **Monitoring**: Prometheus + Grafana

5. APIs & Integrations

Core API Endpoints

- **Account Management**
  - `POST /accounts` - Create new account
  - `GET /accounts` - List all accounts
  - `GET /accounts/{id}` - Get account details
  - `PUT /accounts/{id}` - Update account
  - `GET /accounts/types` - List account types

- **Transaction Management**
    - `POST /transactions` - Record new transaction
    - `GET /transactions` - List transactions (filterable)
    - `POST /transactions/void/{id}` - Void a transaction
- **Reporting**
    - `GET /reports/balances` - Current balances
    - `GET /reports/profit-loss` - Monthly P&L
    - `GET /reports/cashflow` - Cash position

Authentication

- JWT-based authentication
- Role-based access control (Admin, Accountant, Viewer)
- Rate limiting (100 requests/minute per user)

Future Integration Points

- Bank API integration for automated reconciliation
- Payment gateway for online collections
- Accounting software export (QuickBooks format)

6. High-Level Database Design

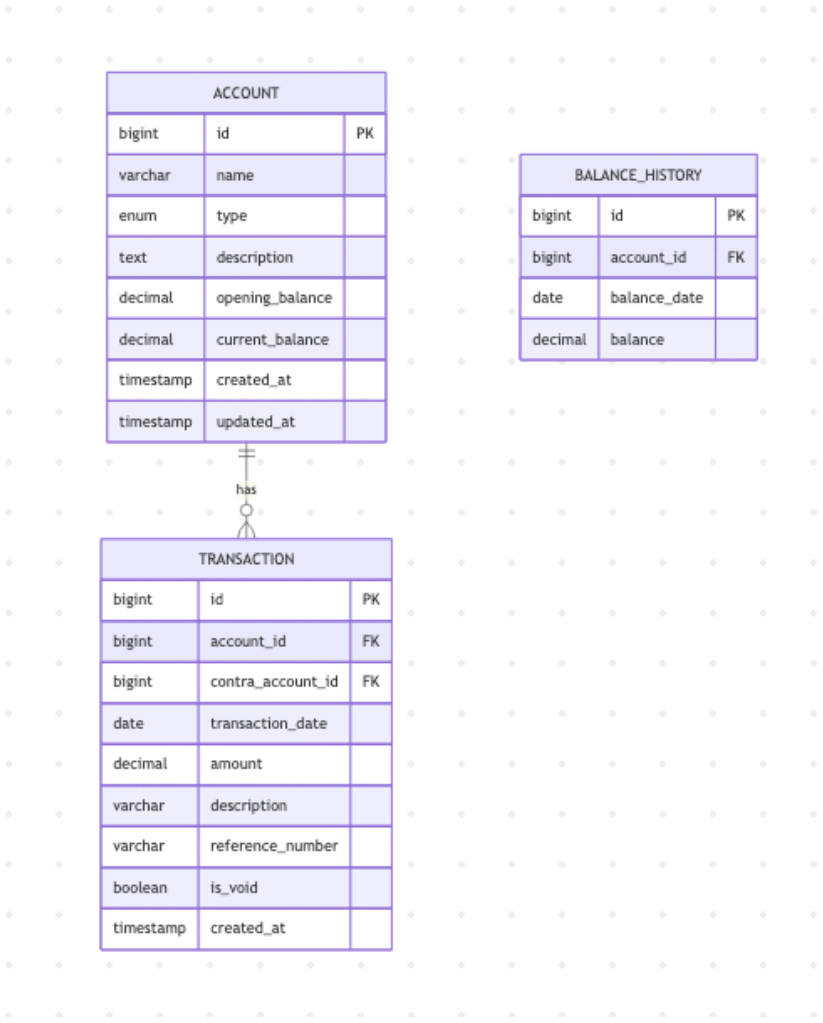
Entity-Relationship Diagram (ERD)

text

```
mermaid
erDiagram
    ACCOUNT ||--o{ TRANSACTION : has
    ACCOUNT {
        bigint id PK
        varchar name
        enum type
        text description
        decimal opening_balance
        decimal current_balance
        timestamp created_at
        timestamp updated_at
    }
    TRANSACTION {
```

```
        bigint id PK
        bigint account_id FK
        bigint contra_account_id FK
        date transaction_date
        decimal amount
        varchar description
        varchar reference_number
        boolean is_void
        timestamp created_at
    }

    BALANCE_HISTORY {
        bigint id PK
        bigint account_id FK
        date balance_date
        decimal balance
    }
```



Key Tables

1. **Accounts**
    - All financial accounts (Income, Expense, Asset, Liability)
    - Track current balance and type
2. **Transactions**
    - Journal entries with debit/account pairs
    - Immutable records (void flag instead of deletion)
3. **Balance History**
    - Daily snapshots for audit trail
    - Enables historical reporting

Data Storage Strategy

- OLTP design optimized for transaction processing
- Indexes on frequently queried fields (account_id, transaction_date)
- Partitioning by date for transaction table at scale
- Regular vacuum and analyze maintenance

7. Security & Compliance

Authentication/Authorization

- JWT-based authentication with 15-minute expiry
- Refresh tokens with 7-day expiry
- Role-based access control:
    - Admin: Full access
    - Accountant: Create/read transactions
    - Viewer: Read-only

Data Protection

- TLS 1.2+ for all communications
- AES-256 encryption for data at rest
- Sensitive fields encrypted in database (e.g., bank account details)

Audit & Compliance

- Immutable audit log of all financial transactions
- User activity logging for sensitive operations
- Daily backups with 30-day retention
- GDPR-ready data subject request processing

8. Scalability & Performance

Horizontal Scaling

- Stateless services allow unlimited horizontal scaling
- Database read replicas for reporting queries
- Connection pooling for database efficiency

Performance Optimization

- Redis cache for:
    - Account balances
    - Frequently accessed reports
    - Reference data (account types, etc.)

Database Optimization

- Query optimization using EXPLAIN ANALYZE
- Appropriate indexing strategy
- Batch processing for end-of-month reports

Expected Metrics

- API response times: <500ms for 95% of requests
- Concurrent users: 50+ with linear scaling
- Transaction throughput: 1000+/month with room to scale

9. Assumptions & Open Questions

Assumptions

- Single currency (INR) operation initially
- Manual bank reconciliation process to start

- On-premise deployment acceptable initially
- 5 concurrent users peak load initially

Open Questions

- Requirement for multi-branch support?
- Need for tenant-facing payment portal?
- Integration with existing property management system?
- Mobile app requirement?

Risks

- Complex transaction validation logic may impact performance
- Audit requirements may necessitate additional storage
- Bank integration may require regulatory compliance

Conclusion

This HLD provides a comprehensive technical blueprint for implementing the Apartment Complex Account Management System that meets all functional requirements while addressing non-functional needs for security, performance, and scalability. The design leverages modern technologies and architectural patterns to create a robust foundation that can evolve with future enhancements.

This response is AI-generated, for reference only.