Part 1 – System Design: Transaction Handling System

Table of Contents

1. High-Level Architecture Overview 2

1.1 State Machine Diagram of Transactions 2

1.2 Account Life Cycle Diagram 2

1.3 System sequence of user sending money to another account 3

2. Key Components & Responsibilities 3

3. Data Models 4

4. Technology Choices 4

5. Non-Functional Requirements 5

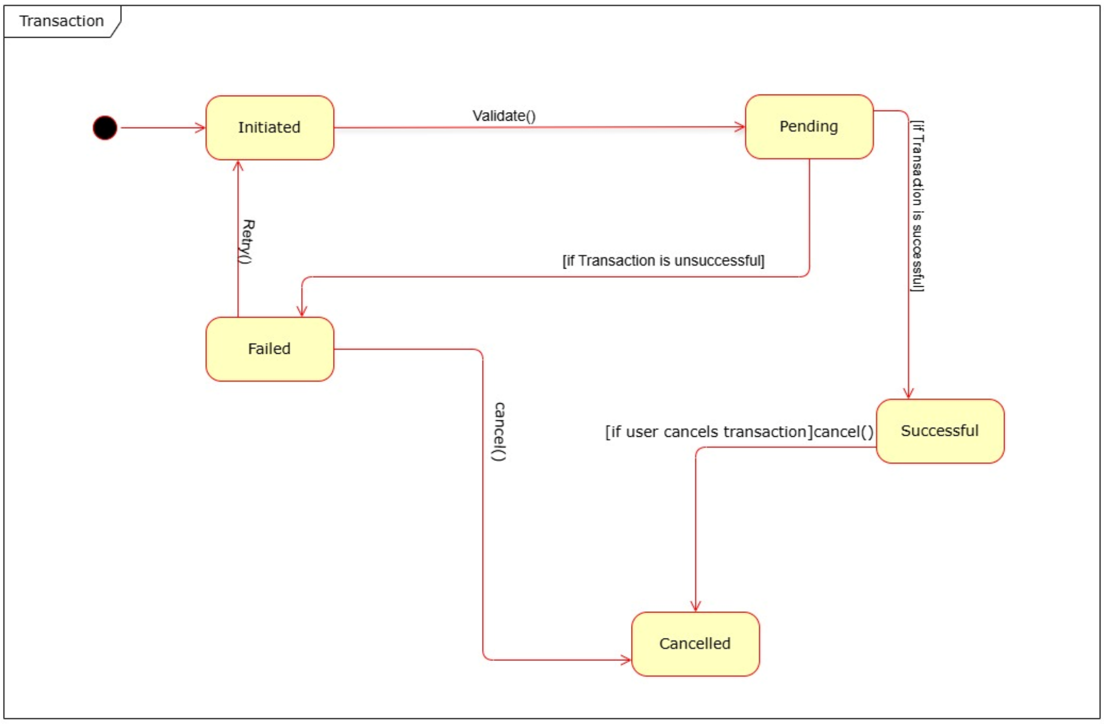
6. Failure Handling 6

Summary 6

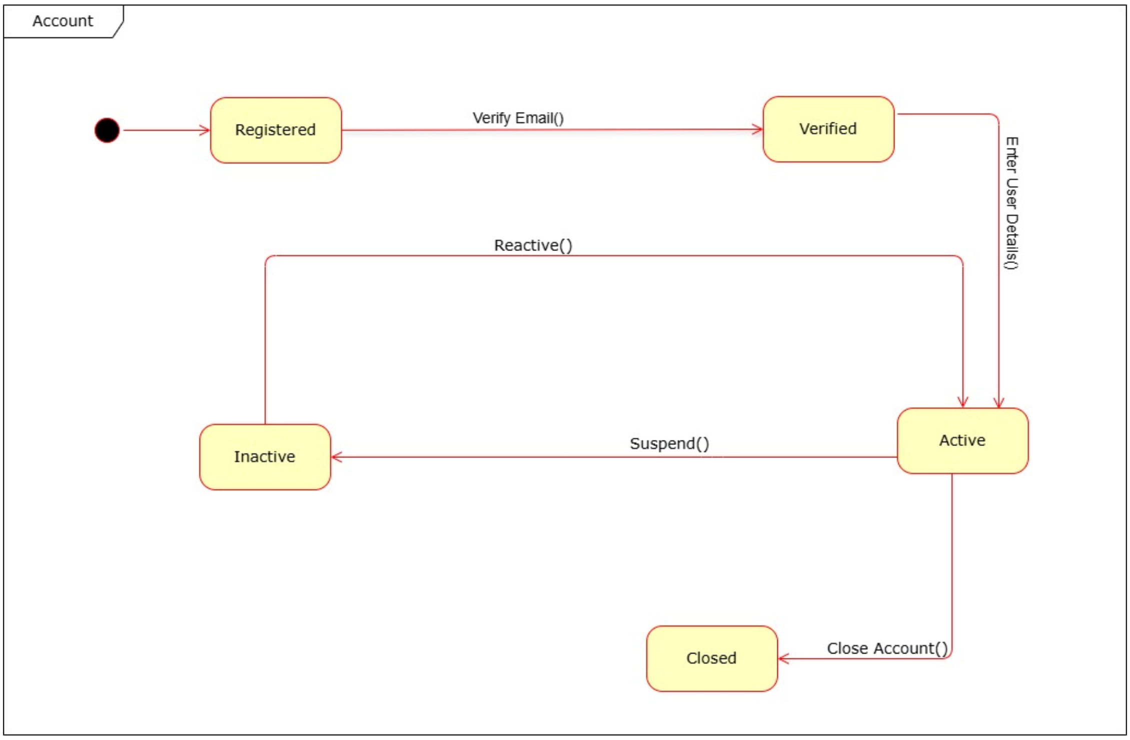
1. High-Level Architecture Overview

The system handles **user-initiated transactions** (e.g., sending money, making purchases, or transferring points). It should be **secure**, **reliable**, and **scalable** — suitable for a fintech-style use case.

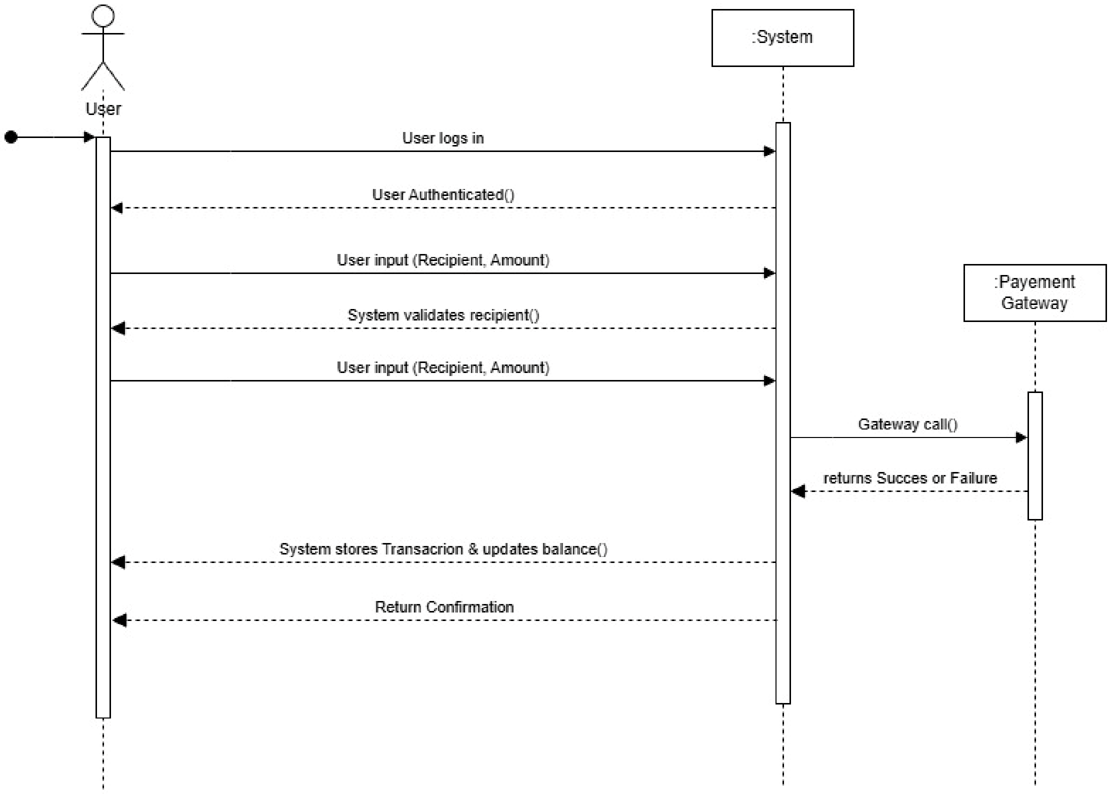
1.1 State Machine Diagram of Transactions


Transaction
Initiated
Validate()
Pending
[if Transaction is successful]
Retry()
[if Transaction is unsuccessful]
Failed
cancel()
[if user cancels transaction]cancel()
Successful
Cancelled

1.2 Account Life Cycle Diagram



1.3 System sequence of user sending money to another account


User
:System
:Payement Gateway
User logs in
User Authenticated()
User input (Recipient, Amount)
System validates recipient()
User input (Recipient, Amount)
Gateway call()
returns Succes or Failure
System stores Transacrion & updates balance()
Return Confirmation

2. Key Components & Responsibilities

Component	Responsibility
Client App	Allows users to initiate/view transactions. Handles UI and input validation.
API Gateway	Routes requests, handles rate limiting, authentication token validation.
Auth Service	Manages user login, sessions, and JWT tokens.
Transaction Service	Processes new transactions, deducts balances, ensures atomic DB operations.
Notification Service	Sends out confirmations via SMS/email.
Database	Stores users, transactions, audit logs, and balances.
External APIs	Communicates with payment providers or banks to finalize transactions.

3. Data Models

🧑‍💻 User

```
{
  "user_id": "uuid",
  "name": "string",
  "email": "string",
  "password_hash": "string",
  "balance": "float",
  "created_at": "timestamp"
}
```

💳 Transaction

```
{
  "transaction_id": "uuid",
  "sender_id": "uuid",
  "recipient_id": "uuid",
  "amount": "float",
  "status": "pending | success | failed",
  "timestamp": "timestamp",
  "description": "string"
}
```

📝 Record

```
{
  "log_id": "uuid",
  "event": "string",
  "user_id": "uuid",
  "timestamp": "timestamp",
  "metadata": "json"
}
```

4. Technology Choices

Layer	Tech Stack	Why?
Frontend	React or Flutter	Cross-platform, fast dev cycle
API Gateway	NGINX or AWS API Gateway	Routing, throttling, and SSL termination

Backend	Node.js (Express) or Python (FastAPI)	Lightweight, async, fast to build
Database	PostgreSQL	Relational integrity, supports transactions
Auth	JWT with bcrypt	Secure, stateless authentication
Notifications	Twilio, SendGrid	Reliable third-party services
External APIs	REST over HTTPS	For bank/payment integration
Deployment	Docker + AWS ECS or Heroku	Scalable and easy to manage

5. Non-Functional Requirements

Security

- Use HTTPS for all comms
- Hash passwords with bcrypt
- Validate and sanitize all inputs
- Use JWT for secure stateless auth
- Store sensitive data encrypted at rest

Reliability

- Use transactions in the DB to ensure atomicity
- Retry logic for failed API calls
- Log all events to enable post-mortems
- Health checks + monitoring

Scalability

- Stateless backend services behind a load balancer
- Use caching for frequent reads (e.g., Redis)
- Horizontal scaling for both web and DB tiers
- Partition data for high-throughput workloads

6. Failure Handling

Failure Type	**How It's Handled**
External API fails	Retry logic + circuit breaker fallback
DB write fails	Rollback transaction + return meaningful error
Auth token invalid	401 Unauthorized + redirect to login
Rate-limiting triggered	429 Too Many Requests with retry-after headers
Unexpected crash	Error logged + monitoring alert (via Datadog/Sentry/etc.)

Summary

This system is designed to be **modular**, **secure**, and **scalable**. Each service has a clear role, and the architecture supports growth and real-world challenges like third-party failures and high traffic. Technologies are chosen for developer speed and long-term maintainability.