Account Transfer System with Balance Validation in Node.js

1. Aim

The primary aim of this project is to implement a secure and reliable API endpoint for simulating financial transactions (account transfers) in a Node.js/Express environment. The core focus is on enforcing **business logic integrity** by performing mandatory balance validation and demonstrating how to handle transactional success and failure conditions.

2. Objectives

Upon completion of this project, the following objectives should be met:

1. **Endpoint Implementation:** Successfully implement a POST /transfer API endpoint to accept transaction requests.
2. **Input Validation:** Implement robust checks for required fields (sourceAccount, destinationAccount, amount) and data types.
3. **Balance Validation:** Ensure the source account has sufficient funds to cover the transfer amount before proceeding with the transaction.
4. **Transaction Simulation:** Simulate the atomic nature of a financial transfer (debit the source, credit the destination) and handle potential failure states gracefully.
5. **Error Handling:** Provide clear and specific error messages to the client based on the validation failure (e.g., insufficient funds, invalid account).

3. Theory

A. Transaction Integrity and ACID Properties

In banking, ensuring **Transaction Integrity** is paramount. This concept is often governed by the **ACID** properties, even when only simulated locally:

- **Atomicity:** The transaction must either fully complete (all steps succeed) or entirely fail (no changes are saved). A transfer involves two atomic steps: debiting the source and crediting the destination. If one fails, both must be undone.
- **Consistency:** The transaction must bring the system from one valid state to another. The total money in the system should not change unless external funds are involved.
- **Isolation:** Concurrent transactions should not interfere with each other (simulated by sequential processing in this project).
- **Durability:** Once committed, the transaction must persist (simulated by updating the in-memory data store).

B. Business Logic Validation

This type of validation goes beyond checking if a field is present (syntactic validation). It enforces the rules of the business:

- **Self-Transfer Prevention:** The source and destination accounts must not be the same.
- **Positive Amount:** The transfer amount must be greater than zero.
- **Sufficient Funds Check (Balance Validation):** The current balance of the source account must be greater than or equal to the requested transfer amount. This check must occur **before** any database write operation.

4. Procedure

The project uses Node.js and the Express framework to create a simple financial service.

1. **Environment Setup & Dependencies:** Initialize the project and install Express: npm install express.
2. **Mock Data Store:** Define an in-memory JavaScript object to simulate a database containing bank accounts and their balances.
3. **Endpoint Creation:** Set up the Express server and define the POST /transfer route.
4. **Middleware Setup:** Use express.json() to parse incoming JSON request bodies.
5. **Validation Logic (The core):**
   - Extract and sanitize inputs (source, destination, amount).
   - Perform necessary business logic checks (source ≠ destination, amount > 0).
   - Implement the crucial balance check against the source account's mock balance.
6. **Transfer Execution:** If all validations pass, update the mock data by subtracting from the source and adding to the destination.
7. **Error Reporting:** Return a specific 400 Bad Request status for validation errors and a 200 OK status for success.
8. **Server Execution:** Run the server using: node transfer_server.js.

5. Code

The complete Node.js/Express server implementation is provided in the transfer_server.js file, which includes the mock data store and the validated transfer endpoint.

6. Output

A. Server Output (Initial State)

The console will show the initial mock balances when the server starts.

```
Server running at http://localhost:3000
Current Account Balances:
  123456: $1000.00
  987654: $500.00
  112233: $5000.00
```

B. HTTP Request/Response (Transfer Scenarios)

Test Case	Request (Body)	Status Code	Response Body
1. Success	{ "sourceAccount": "123456", "destinationAccount": "987654", "amount": 100 }	200 OK	{"message":"Transfer successful. New Balance (123456): $900.00"}
2. Insufficient Funds	{ "sourceAccount": "987654", "destinationAccount": "112233", "amount": 600 }	400 Bad Request	Transfer failed: Insufficient funds in source account 987654.
3. Invalid Input (Amount)	{ "sourceAccount": "123456", "destinationAccount": "987654", "amount": -50 }	400 Bad Request	Validation failed: Amount must be a positive number.
4. Self-Transfer	{ "sourceAccount": "123456", "destinationAccount": "123456", "amount": 10 }	400 Bad Request	Validation failed: Source and destination accounts cannot be the same.

7. Learning Outcomes

1. **Business Logic Implementation:** Mastered the implementation of critical financial rules (balance checks, positive amounts) directly in the application layer, demonstrating the difference between simple request validation and complex business validation.
2. **Atomic Transaction Handling:** Understood the concept of atomicity by ensuring the twin operations (debit and credit) are linked, providing consistent state management for financial data.
3. **Robust Error Mapping:** Learned to map specific business failures (e.g., "Insufficient Funds") to appropriate HTTP status codes and informative client messages, crucial for effective API development.