

## Practice 3 - Backend

### Account Transfer System with Balance Validation in Node.js

**Objective:** Learn how to implement a secure money transfer API in Node.js and MongoDB without using database transactions. This helps you understand dependent multi-document updates, balance validation, and proper error handling.

#### **Concept Overview:**

In financial systems, fund transfers require consistent updates to multiple user accounts. This exercise teaches logical validation and sequential updates to ensure accuracy without database transactions.

#### **Steps / Procedure:**

##### **Step 1: Initialize Project**

```
mkdir account-transfer-system
cd account-transfer-system
npm init -y
npm install express mongoose body-parser
```

##### **Step 2: Create server.js**

```
const express = require('express'); const
mongoose = require('mongoose'); const
bodyParser = require('body-parser'); const
app = express(); const PORT = 3000;

app.use(bodyParser.json());

mongoose.connect('mongodb://localhost:27017/bankDB', {
  useNewUrlParser: true,   useUnifiedTopology: true });

const userSchema = new mongoose.Schema({
  name: String,   balance: Number });

const User = mongoose.model('User', userSchema);

app.post('/create-users', async (req, res) => {
  try {    await User.deleteMany({});    const
  users = await User.insertMany([    { name:
  'Alice', balance: 1000 },
    { name: 'Bob', balance: 500 }    ]);
  res.status(201).json({ message: 'Users created', users });
} catch (err) {    res.status(500).json({ message: 'Error
creating users' });    });

app.post('/transfer', async (req, res) => {  try {
const { fromUserId, toUserId, amount } = req.body;
const sender = await User.findById(fromUserId);
const receiver = await User.findById(toUserId);
```

```

    if (!sender || !receiver) {      return
    res.status(404).json({ message: 'User not found' });    }

    if (sender.balance < amount) {    return res.status(400).json({
message: 'Insufficient balance' });    }

    sender.balance -= amount;
    receiver.balance += amount;

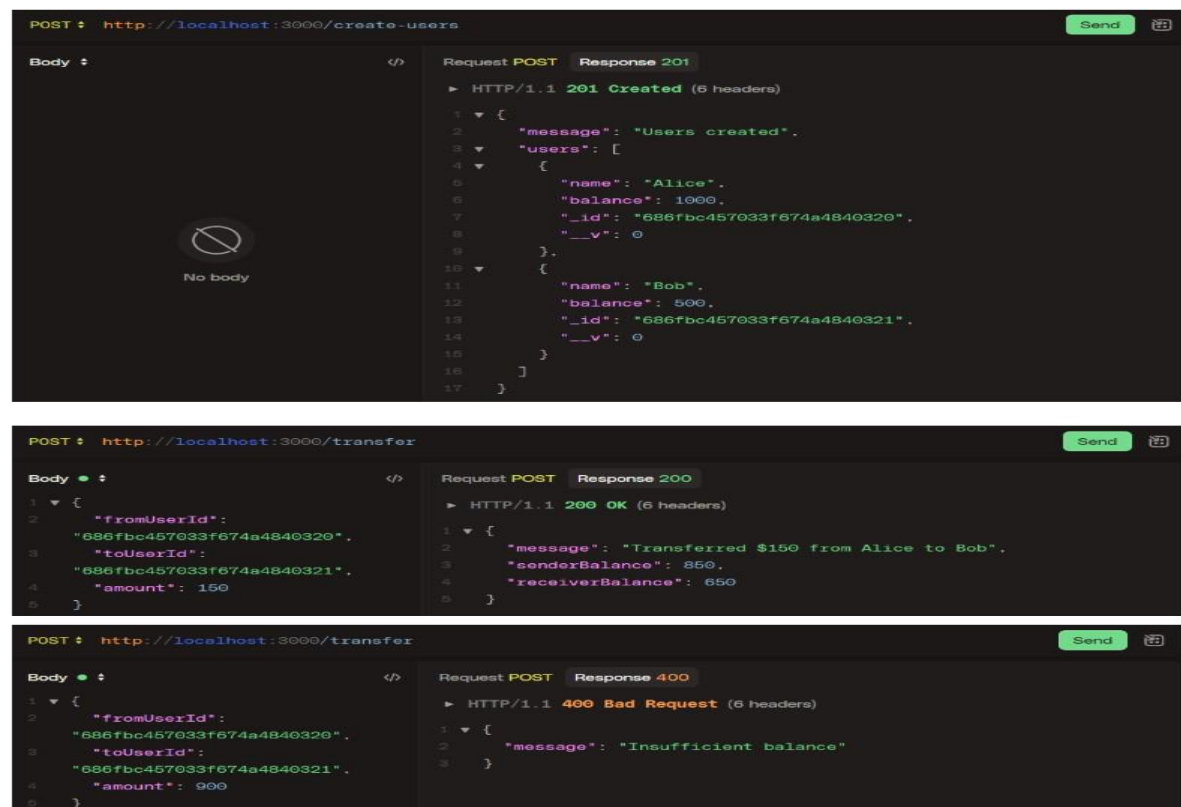
    await sender.save();
    await receiver.save();

    res.status(200).json({      message: `Transferred ${amount} from
${sender.name} to ${receiver.name}`,      senderBalance: sender.balance,
receiverBalance: receiver.balance    });    } catch (error) {
res.status(500).json({ message: 'Transfer failed', error: error.message });    }
});

app.listen(PORT, () => {    console.log(`Server running on
http://localhost:${PORT}`); });

```

### Expected Output:



The image displays three screenshots of a REST client interface, showing the results of API requests and responses for a user transfer system.

**First Screenshot:** A POST request to `http://localhost:3000/create-users` is shown. The response is a 201 status code, indicating that users were successfully created. The response body contains a message "Users created" and an array of two user objects: Alice (balance: 1000) and Bob (balance: 500).

**Second Screenshot:** A POST request to `http://localhost:3000/transfer` is shown. The request body contains the following JSON: `{ "fromUserId": "686fbc457033f674a4840320", "toUserId": "686fbc457033f674a4840321", "amount": 150 }`. The response is a 200 status code, indicating that the transfer was successful. The response body contains a message "Transferred \$150 from Alice to Bob", the sender's balance (850), and the receiver's balance (650).

**Third Screenshot:** A POST request to `http://localhost:3000/transfer` is shown. The request body contains the following JSON: `{ "fromUserId": "686fbc457033f674a4840320", "toUserId": "686fbc457033f674a4840321", "amount": 900 }`. The response is a 400 status code, indicating an "Insufficient balance" error.

### Result:

The system successfully transfers funds between users with proper validation and error handling. Logical checks ensure consistent data even without transactions.