**MongoDB: Transaction, Replication**

**1. Replica Set Setup**

**Objectives:**

* Three-node replica set (1 primary, 2 secondaries)
* 20 s replication delay on secondaries, to observe lag

**1.1 Prepare three data directories**

mkdir -p ~/mongo/rs0-{0,1,2}

**1.2 Launch three mongod processes**

mongod --replSet rs0 --port 27017 --dbpath ~/mongo/rs0-0 --bind\_ip localhost --fork --logpath ~/mongo/rs0-0/log.txt

mongod --replSet rs0 --port 27018 --dbpath ~/mongo/rs0-1 --bind\_ip localhost --fork --logpath ~/mongo/rs0-1/log.txt

mongod --replSet rs0 --port 27019 --dbpath ~/mongo/rs0-2 --bind\_ip localhost --fork --logpath ~/mongo/rs0-2/log.txt

**1.3 Initiate and configure the set**

Connect via mongosh --port 27017 and run:

rs.initiate({

\_id: "rs0",

members: [

{ \_id: 0, host: "localhost:27017" },

{

\_id: 1,

host: "localhost:27018",

priority: 0,

secondaryDelaySecs: 20

},

{

\_id: 2,

host: "localhost:27019",

priority: 0,

secondaryDelaySecs: 20

}

]

});

* Secondaries will lag by ~20 seconds.
* Use rs.status() to verify.

**2. Build the Bank System in TypeScript**

**Core Requirements:**

* **BankDB** database with an **Accounts** collection
* Money-transfer function using multi-document transactions
* Demonstrate success & forced failure

Below is a self-contained script—bank.ts—that:

1. Connects to your replica set
2. Seeds two accounts
3. Runs a successful transfer
4. Runs a failing transfer (simulates error)
5. Cleans up

**Prerequisites:**

npm install mongodb @types/node

tsc --init

Make sure tsconfig.json has "esModuleInterop": true.

// bank.ts

import { MongoClient, ObjectId, ClientSession } from "mongodb";

const uri = "mongodb://localhost:27017,localhost:27018,localhost:27019/?replicaSet=rs0";

const client = new MongoClient(uri);

interface Account {

\_id?: ObjectId;

name: string;

balance: number;

}

async function withTransaction(fn: (session: ClientSession) => Promise<void>) {

const session = client.startSession();

try {

await session.withTransaction(async () => {

await fn(session);

}, {

readConcern: { level: "local" },

writeConcern: { w: "majority" },

readPreference: "primary"

});

} finally {

await session.endSession();

}

}

async function transferMoney(

fromId: ObjectId,

toId: ObjectId,

amount: number

) {

const db = client.db("BankDB");

const accounts = db.collection<Account>("Accounts");

await withTransaction(async (session) => {

const from = await accounts.findOne({ \_id: fromId }, { session });

if (!from || from.balance < amount) {

throw new Error("Insufficient funds");

}

await accounts.updateOne(

{ \_id: fromId },

{ $inc: { balance: -amount } },

{ session }

);

await accounts.updateOne(

{ \_id: toId },

{ $inc: { balance: amount } },

{ session }

);

console.log(

`Transferred $${amount} from ${fromId} to ${toId} successfully.`

);

});

}

async function main() {

await client.connect();

const db = client.db("BankDB");

const accounts = db.collection<Account>("Accounts");

await accounts.deleteMany({}); // clean slate

// Seed two accounts

const [aliceRes, bobRes] = await Promise.all([

accounts.insertOne({ name: "Alice", balance: 1000 }),

accounts.insertOne({ name: "Bob", balance: 500 })

]);

const aliceId = aliceRes.insertedId;

const bobId = bobRes.insertedId;

console.log("Seeded accounts:", aliceId, bobId);

// 2.3.1 Successful transfer

try {

await transferMoney(aliceId, bobId, 200);

} catch (e) {

console.error("Unexpected failure:", e);

}

// 2.3.2 Failed transfer (simulate error)

try {

await transferMoney(aliceId, bobId, 2000);

} catch (e) {

console.error("Expected failure:", e.message);

}

// Show final balances

const final = await accounts.find().toArray();

console.table(final.map(a => ({

name: a.name,

balance: a.balance

})));

await client.close();

}

main().catch(console.error);

**How to run:**

tsc bank.ts

node bank.js

You should see:

1. Seeded account IDs
2. “Transferred $200 … successfully.”
3. “Expected failure: Insufficient funds”
4. A table showing Alice = 800, Bob = 700

**Deliverables**

* **Replica-set commands** and rs.initiate() config (above).
* **bank.ts** TypeScript file (above).

This satisfies **all** parts of Exercise 1: replica-set setup, delayed secondaries, database/model creation, transaction-backed transfers, success and forced-failure tests.