Mini Banking Application

System Design & Architecture Documentation

Technology Stack: MERN (MongoDB, Express, React, Node.js)

Prepared for Developer & System Architecture Review

Table of Contents

1. Introduction

This document provides a comprehensive and professionally structured architectural overview of the Mini Banking Application built using the MERN stack. It follows industry documentation standards similar to AWS, Google Cloud, and MongoDB technical design papers.

2. System Overview

The Mini Banking App simulates core banking operations such as account creation, balance management, and fund transfers. It does not integrate with external payment gateways; instead, it focuses on database integrity, secure authentication, transaction logs, and deterministic financial operations.

3. High-Level Architecture

The architecture is divided into three primary layers: frontend (React), backend (Node.js + Express), and database (MongoDB). Each layer communicates via REST APIs over HTTPS.

[Client (React)]
|
| HTTPS (JWT Authentication)
v
[API Server - Node.js/Express]
|
| Mongoose ORM
v
[MongoDB Database]

4. Component Responsibilities

Frontend (React)

• Authentication UI • Dashboard with real-time balance • Transfer interface • Transaction history with filters • Responsive and modular UI components

Backend (Node.js + Express)

• REST API handling • User authentication and token management • Business logic for account and transaction operations • Validation and security middleware • MongoDB transaction sessions for atomic operations

Database (MongoDB)

MongoDB stores persistent financial and user information using structured collections. Document relationships are implemented using ObjectId references.

5. Database Schema & Relationships

Users (1:1 → Accounts)

- _id
- name
- email
- password (hashed)
- accountId (FK)
- createdAt

Accounts (1:N → Transactions)

- _id
- userId (FK)
- balance
- accountNumber
- createdAt

Transactions (Many-to-One)

- _id
- fromAccount (FK)
- toAccount (FK)
- amount
- type (CREDIT | DEBIT | TRANSFER)
- createdAt

Session Logs (N:1 → Users)

- _id
- userId (FK)
- ip
- loginAt

6. API Specification

POST /auth/register → Create user + account
POST /auth/login → Authenticate user & return JWT
GET /account/me → Fetch account + user details
GET /account/balance → Retrieve current balance
POST /account/transfer → Transfer funds between accounts
GET /transactions → Transaction history with filters

7. Transactional Logic (ACID)

1. Validate user identity and JWT token
2. Validate sender's account balance
3. Start MongoDB session and initiate transaction
4. Deduct amount from sender's account
5. Credit amount to receiver's account
6. Insert transaction log
7. Commit the session (ACID compliance)

8. Aggregation Pipelines

User + Account (JOIN)

```
db.users.aggregate([
  { $lookup: {
      from: "accounts",
      localField: "accountId",
      foreignField: "_id",
      as: "account"
  }},
  { $unwind: "$account" }
]);
```

Monthly Summary

```
db.transactions.aggregate([
  { $group: {
      _id: { month: { $month: "$createdAt" }, year: { $year: "$createdAt" }},
      totalAmount: { $sum: "$amount" },
      count: { $sum: 1 }
  }},
  { $sort: { "_id.year": 1, "_id.month": 1 }}
]);
```

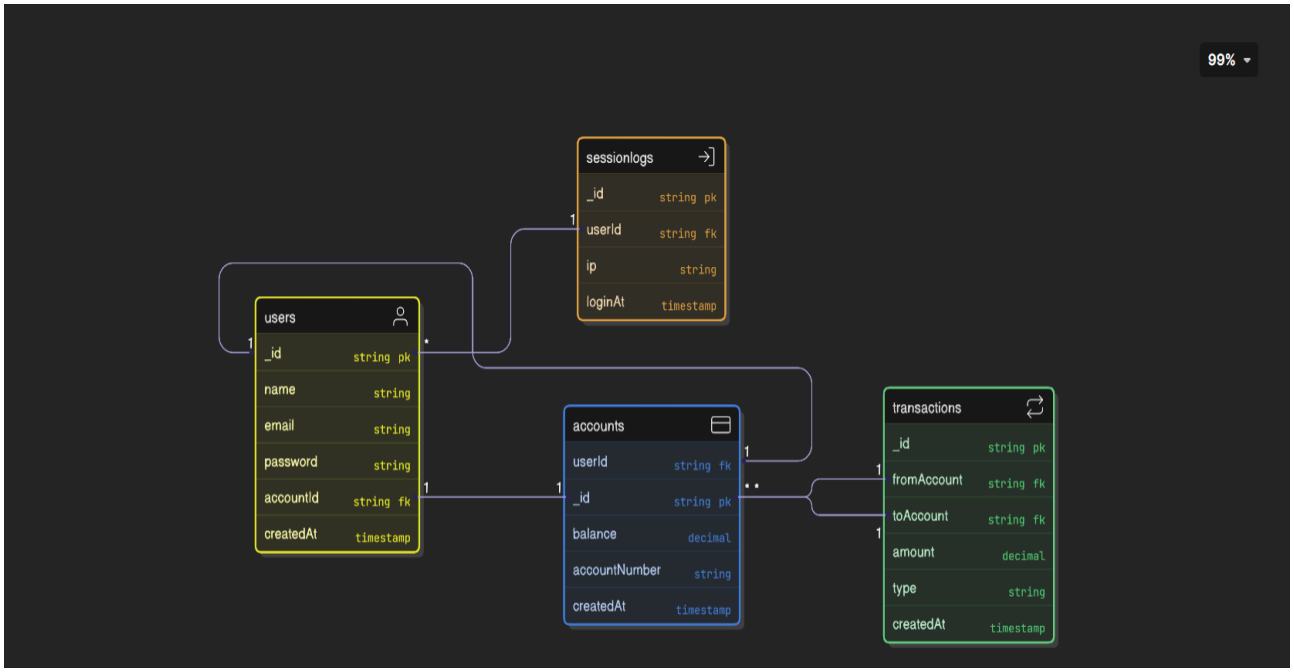
9. Folder Structure

```
backend/
  config/
  controllers/
  models/
  routes/
  middleware/
  server.js

frontend/
  components/
  pages/
  context/
  hooks/
  App.js
```

10. Development Workflow

1. Setup server + MongoDB connection
2. Build authentication system
3. Implement protected routes
4. Add account handling logic
5. Implement financial transactions with ACID guarantees
6. Build frontend UI + API integration
7. Add transaction filters & aggregation data
8. Final Q


99%
sessionlogs
_id string pk
userId string fk
ip string
loginAt timestamp
users
_id string pk
name string
email string
password string
accountId string fk
createdAt timestamp
accounts
userId string fk
_id string pk
balance decimal
accountNumber string
createdAt timestamp
transactions
_id string pk
fromAccount string fk
toAccount string fk
amount decimal
type string
createdAt timestamp

Title: Mini Banking Backend Security Checklist (Beginner-Friendly)

---

1. Password Security

- Never store passwords as plain text.
- Use bcrypt to hash passwords.

```
const bcrypt = require('bcrypt');
const hashedPassword = await bcrypt.hash(password, 10);
```

- Compare hashed passwords when logging in.

```
const match = await bcrypt.compare(inputPassword, hashedPassword);
```

2. Authentication

- Use JWT to manage sessions.
- Issue a token after login and protect routes.

```
const jwt = require('jsonwebtoken');
const token = jwt.sign({ userId: user._id }, 'secret_key', { expiresIn:
'1h' });
```

3. HTTPS

- Always use HTTPS in production to encrypt data in transit.

4. Input Validation

- Validate user input to prevent attacks.
- Use libraries like Joi or express-validator.
- Examples:
- Email format validation.
- Amount > 0 for deposits/withdrawals.

5. Authorization

- Ensure users can only access their own data.
- Example check: `req.user.id === transaction.userId`

6. Limit Requests

- Implement rate limiting to prevent brute force attacks.

```
const rateLimit = require('express-rate-limit');
const limiter = rateLimit({ windowMs: 15*60*1000, max: 100 });
app.use(limiter);
```

7. MongoDB Security

- Never expose your DB publicly.
- Use strong credentials.
- Enable authentication in MongoDB.
- Never commit credentials to GitHub.

8. Logging

- Maintain an audit log for deposits, withdrawals, and transfers.
- Helps track suspicious activity.

9. Environment Variables

- Store secrets in a `.env` file.

```
DB_URI=mongodb+srv://username:password@cluster.mongodb.net/mydb
JWT_SECRET=supersecretkey
```

Tip: Start simple, follow this checklist, and gradually improve security as you learn more.