Core Banking System (CBS) — Project Documentation

1. Title Page

- Project: Core Banking System (CBS)
- Backend: Node.js + Express
- Database: MySQL (mysql2/promise)
- Frontend: Static HTML (Bootstrap), Fetch API
- Author: Hassaan Tariq
- Date: December 5, 2025

2. Abstract

This project implements a simplified Core Banking System showcasing transactional integrity and auditability. It provides customer, account, and transaction management APIs and demonstrates Transaction Control Language (TCL) features including explicit transactions, row-level locking, savepoints, and controlled rollbacks. The system includes an admin-powered demo UI to visualize deposits, withdrawals (including error paths), transfers, and savepoint scenarios, as well as audit and activity reports.

3. Introduction (CBS Overview)

A Core Banking System handles the lifecycle of customers and accounts, and secure, atomic financial operations. This CBS demonstrates:

- Customer onboarding and account creation
- Deposits, withdrawals, and inter-account transfers
- Transaction logging and staff audit trails
- Safe concurrency via `SELECT ... FOR UPDATE`
- TCL demonstrations (COMMIT, ROLLBACK, SAVEPOINT) for educational purposes

Key components:

- `server/app.js`: Express app, mounts routers under `/api/*`, serves `public/` and `/health`
- `server/db.js`: MySQL connection pool via `mysql2/promise` with env-configurable credentials
- Routers/controllers: Customers, Accounts, Transactions, TCL demos, and a simple customer portal
- `public/`: Static demo pages including `tcl_complete.html`, audit and reports UIs

4. Objectives & Scope

- Objectives:
  - Implement core banking operations with transactional safety
  - Demonstrate TCL primitives (begin/commit/rollback, savepoints) with clear UI flows
  - Provide auditability through `AuditLog` and detailed `TransactionLog`
  - Offer basic reporting for recent activity and summaries
- Scope:
  - Educational demo (not production-grade)
  - Admin token-based attribution; simplified customer portal

    - Focus on clear transaction patterns and correctness under constraints

5. ERD/EERD Diagrams

Entities and relationships (high-level):

- `UserAccount (userid, fullname, password_hash, role)`
    - Used for staff/admin attribution in audit and transaction logs
- `Customer (customerid, fullname, email, phone)`
- `Branch (branchid, name, location)`
- `Account (accountno, customerid -> Customer, branchid -> Branch, balance, status)`
- `TransactionLog (id, accountno -> Account, type, amount, reference_account -> Account, performed_by -> UserAccount, created_at)`
- `AuditLog (id, userid -> UserAccount, action, description, created_at)`

Suggested diagram notes:

- `Account.customerid` and `Account.branchid` are foreign keys
- `TransactionLog.reference_account` supports transfer pairing (from/to sides)
- `AuditLog.userid` tracks staff actions

You can render an ERD using tools like MySQL Workbench or dbdiagram.io based on `sql/database_schema.sql`.

6. Database Design (Schema)

See `sql/database_schema.sql`. Highlights:

- Referential integrity via foreign keys
- Indexed primary keys for `Customer`, `Account`, `UserAccount`
- `TransactionLog` captures both sides of transfers using `reference_account`
- `AuditLog` records descriptive actions and the acting `userid`

Initialization:

```
mysql -u <user> -p < sql/database_schema.sql
```

Default env (override via `server/db.js`):

- `DB_HOST=127.0.0.1`, `DB_USER=root`, `DB_PASSWORD=root`, `DB_NAME=cbs_db`

7. Transaction Control Implementation (TCL)

Patterns used across controllers (e.g., `transactionController.js`, `tclController.js`):

- Explicit transactions:
    - `conn.beginTransaction()`
    - `conn.commit()`
    - `conn.rollback()`

- Row locking for concurrency safety:
  - `SELECT ... FOR UPDATE` before mutating balances
- Savepoints and partial rollbacks:
  - `conn.query('SAVEPOINT <name>')`
  - `conn.query('ROLLBACK TO SAVEPOINT <name>')`
- Auditability:
  - Insert into `AuditLog` after state-changing operations
  - `performed_by` attribution in `TransactionLog`
- Defensive validation:
  - Positive amounts, active accounts, sufficient funds checks

Examples:

- Deposit:
  - Lock account → increment balance → log transaction → audit → commit
- Withdrawal:
  - Lock account → validate funds → decrement balance → log → audit → commit
- Transfer:
  - Lock both accounts in consistent order → debit/credit → dual log entries → audit → commit
- Savepoint demo:
  - Withdraw step 1 → `SAVEPOINT` → attempt step 2 → if violation, `ROLLBACK TO SAVEPOINT` → log only valid steps → audit → commit

Admin attribution:

- `Authorization: admin_<userid>_<timestamp>` header parsed server-side (fallback to `userid` in body → default)
- Ensures logs show the current admin (e.g., Hassaan Tariq)

8. Possible Future Enhancements

- Strong auth:
  - JWT for admins/staff; password reset flows; role-based access
- Customer features:
  - Proper customer login, profile management, statements, notifications
- Ops & UX:
  - Pagination and filtering for audit/reports; advanced search
  - Pre-submit account existence checks in TCL UI; improved validation
- Banking features:
  - Scheduled transfers; interest accrual; fee processing; account states
- Reliability:
  - Idempotent endpoints; outbox pattern; retry logic; better error taxonomy
- Observability:
  - Structured logging, metrics, tracing; admin dashboards
- Security:
  - Least-privilege DB user; parameterized migrations; secrets management

---

Quick Run & Demo

- Start (dev): `npm run dev`
- Start (prod): `npm start`
- Health: `GET /health`
- TCL demo UI: open `public/tcl_complete.html`
- Reports: `public/audit_log.html`, `public/reports.html`
- Tests:

```
node test_cbs.js
node test_customers.js
```

References

- App entry: `server/app.js`
- DB pool: `server/db.js`
- Controllers: `server/controllers/*.js`
- Routes: `server/routes/*.js`
- Schema: `sql/database_schema.sql`
- Static UIs: `public/*.html`