Banking System Database for Loan Processing

Luan Cao – SID: 017048694 – dao.cao@sjsu.edu
Clyde Joshua Delgado – SID: 017575766 – delgado.clydejoshua@sjsu.edu
May 9, 2025

Introduction

As a **project overview**, we developed a database management system for a **banking system** that focuses on managing customer accounts, tracking transactions, and handling loan applications and payments. The system will allow customers to manage their accounts and help manage finances easily by accurately displaying and updating data according to user transactions.

Objectives

The **primary goals** of the project are to design a relational database to store and manage customers, accounts, transactions, and loans; develop a web application for customers to manage and view their bank accounts and loans through API calls using the Spark Java Framework; and utilize the JDBC API to connect and interact with the MySQL database.

The web application will have the **key features** such as user authentication and perform certain actions that automatically perform create, retrieve, update, and delete operations, such as in transfers and applications.

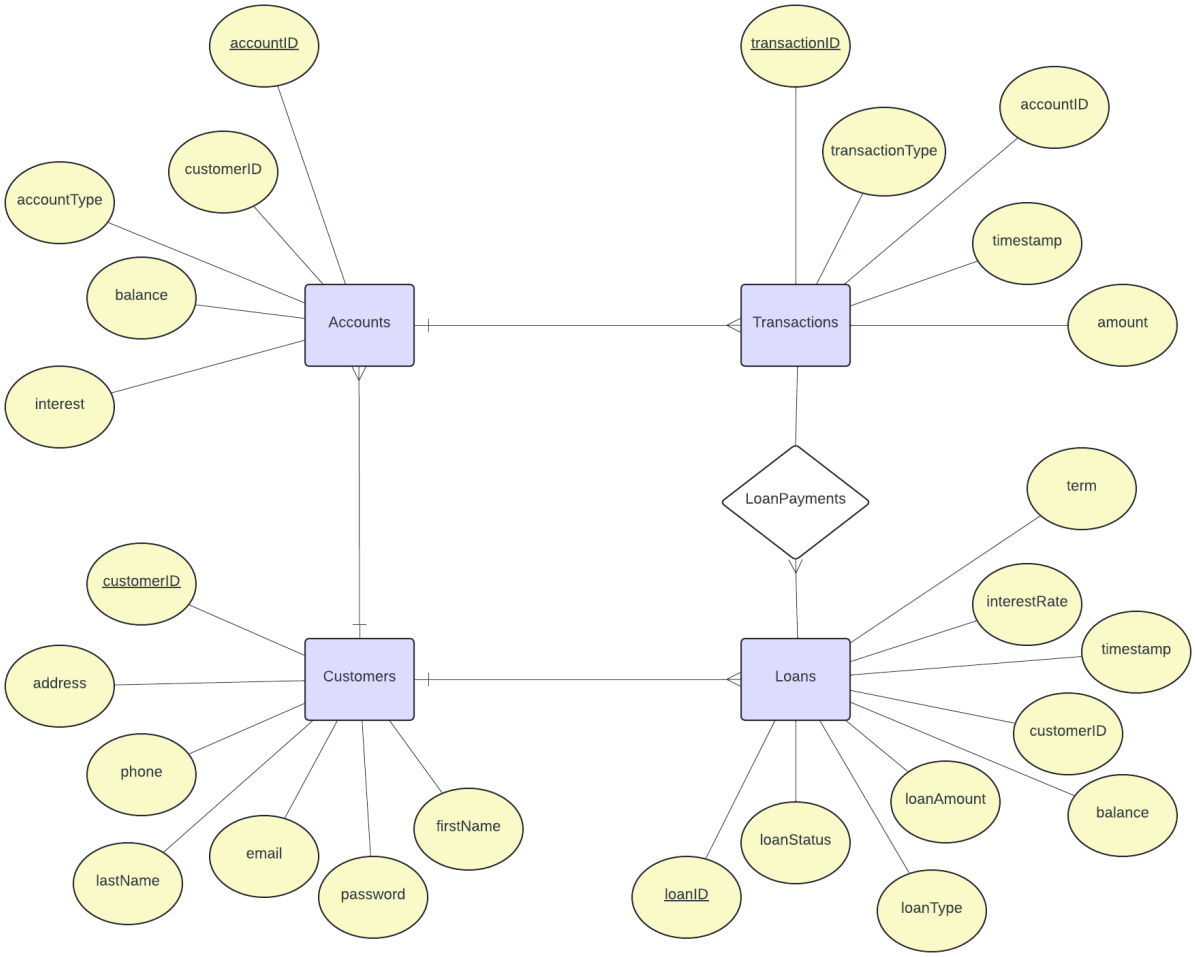
System Architecture

The system employs a **three-tier architecture** divided into the presentation, logic, and database layers. The **presentation layer** uses native HTML5, CSS3, and JavaScript as front-end technologies for data entry and display. The **logic layer** uses the Spark Java framework to handle API requests and responses and JDBC to execute queries. Lastly, the **database layer** stores all entities and relationships using MySQL. The following is the full list of the **technology stack:**

- IntelliJ IDEA (development IDE)
- Java (JDK 11+)
- Java Database Connectivity (JDBC)
- Spark Java Framework
- MySQL 8.0
- HTML5
- CSS3
- JavaScript

Database Design

Schema Diagram:



Normalization:

+ Accounts(AccountID, CustomerID, AccountType, Balance, Interest)
+ Customers(CustomerID, FirstName, LastName, Address, Phone, Email, Password)
+ Transactions(TransactionID, TransactionType, AccountID, Timestamp, Amount)
+ Loans(LoanID, LoanStatus, LoanType, LoanAmount, Balance, CustomerID, Timestamp, InterestRate, Term)
+ LoanPayments(TransactionID, LoanID)

Entity Descriptions:

- **Customers:** Stores client's personal details.
- **Accounts:** Links to Customers, records account type, and balance.
- **Transactions:** Records debit/credit entries for each Account.
- **Loans:** Tracks loan applications, amounts, statuses, interest rates, and application dates.

Attributes and Constraints:

Customer table

Field	SQL type	Java type	Description	Constraint
customerID	INTEGER	integer	Primary key. Autoincrement field for each CustomerID record to ensure that each record has a unique CustomerID	NOT NULL
firstName	VARCHAR(30)	String	Customer's first name	NOT NULL
lastName	VARCHAR(30)	String	Customer's last name	NOT NULL
address	VARCHAR(40)	String	Customer's address	NOT NULL
phone	CHAR(10)	String	Customer's phone number	NOT NULL, phone should be ten digit and each character should be in range [0-9]
email	VARCHAR(30)	String	Customer's email	NOT NULL, email should be in pattern %_@__%.__%
password	VARCHAR(30)	String	Customer's password	NOT NULL

Account table

Field	SQL type	Java type	Description	Constraint
accountID	INTEGER	integer	Primary key. Autoincrement field for each AccountID record to ensure that each record has a unique AccountID	NOT NULL
customerID	INTEGER	integer	Foreign Key. CustomerID that link with an AccountID	NOT NULL
accountType	CHAR(20)	String	Type of the Account	NOT NULL, accountType should be CHECKING or SAVING
balance	DOUBLE	double	Amount of money in an Account	NOT NULL, Balance > 0
interest	DECIMAL(4,3)	double	Interest rate	NOT NULL, Interest > 0

Transaction table

Field	SQL type	Java type	Description	Constraint
transactionID	INTEGER	integer	Primary key. Autoincrement field for each TransactionID record to ensure that each record has a unique TransactionID	NOT NULL
transactionType	CHAR(20)	String	Type of the transaction	NOT NULL
accountID	INTEGER	integer	Foreign key. Account ID number that link with the TransactionID	NOT NULL

timestamp	TIMESTAMP	String	Date of the Transaction	NOT NULL
amount	DOUBLE	double	Amount of money in the Transaction	NOT NULL, amount > 0

Loan table

Field	SQL type	Java type	Description	Constraint
loanID	INTEGER	integer	Primary key. Autoincrement field for each LoanID record to ensure that each record has a unique LoanID	NOT NULL
loanStatus	VARCHAR(10)	String	The status of the loan	NOT NULL. Status is one of these values: PENDING, APPROVED, PAID.
loanType	VARCHAR(10)	String	The type of the loan	NOT NULL. Type is one of these values: HOME, STUDENT, PERSONAL, AUTO.
loanAmount	DOUBLE	double	The principal amount of the loan	NOT NULL. Amount > 0
balance	DOUBLE	double	The remaining amount of the loan	NOT NULL. Balance > 0
customerID	INTEGER	integer	Foreign key. The CustomerID that link with the LoanID	NOT NULL
timestamp	TIMESTAMP	String	The date when the Loan is established	NOT NULL
InterestRate	DECIMAL(4,3)	double	The rate of the Loan	NOT NULL. Interest rate > 0
term	INTEGER	integer	Years for the loan to be paid	NOT NULL. term > 0

LoanPayment table:

Field	SQL type	Java type	Description	Constraint
transactionID	INTEGER	integer	Primary key and foreign key reference to Transaction table.	NOT NULL
loanID	INTEGER	integer	Primary key and foreign key reference to Loan table.	NOT NULL

Functional Requirements

The user will be the bank customer using the bank system on a web browser to manage their accounts, funds, and loans. In the system, they'll be able to view their profile, accounts, and loans; apply for a loan; pay for a loan; and transfer funds.

Feature Descriptions:

- **Login and logout:** customers are authenticated when logging in and creating a session. The session persists until the user logs out.
- **Dashboards**: customers can view their account and loan on a dashboard. Specific dashboards offer more specific details about balances, transactions, interest, and such.
- **Funds Transfer:** customers can transfer funds from one account to another. Customers may transfer to other customers' accounts, but they may not take from other accounts.
- **Apply Loan:** customers may apply for a variety of loans with variable terms and interest rates. Minimum amounts are in place though the application does not go through any inspection or approval process.
- **Pay Loan:** customers may pay for a loan using funds directly from their account. They must have enough balance to pay for the loan.
- **Delete Loan:** customers are prompted to delete a loan whenever the balance hits zero.
- **Transactions**: customers may view their transactions by visiting an account page. Transactions are displayed from newest to oldest.
- **Transaction Filter:** customers may filter their transactions between certain dates and amounts by filling out the filter inputs. All values are optional so users may only add one filter.
- **Address and contact info change:** customers may visit their profile and update their address or contact information, such as phone and email.

Non-Functional Requirement

The following non-functional requirements were added to enhance the user experience and provide some form of security.

- **Performance:** to enhance performance, two indices are added for the accountID and timestamp of Transactions as these columns are often referred to by queries.
- **Security:** Customers are authenticated before viewing their accounts. Input validation was added on all forms to preserve the consistency of the database.
- **Accessibility:** The application offers a responsive web interface on devices with smaller screen size.

Implementation Details

Code Structure

The code are separated into two main categories, the logic and presentation layers. Under src are folders java and resources.

The java folder holds all the .java files required for the logic layer. It is further divided into sub folders: db, model, controller, service, and dao. The db folder only holds the Database class, which creates the connection for the application. The model folder contains all the classes or objects that are used to translate from Java objects to json objects.The controller folder contains

all the controller code which manages the routes and parsing of data from the request before passing it to a service as well as issuing success or fail response. The service folder contains the services which calls the data access objects (DAO). Lastly, the dao folder contains all the dao which interacts with the database using JDBC.

Code Snippets

Upon launching the application the **database connection** is automatically established in Database.java using the following code snippets (not in sequence):

```
import java.sql.DriverManager;
private static Connection connection;
connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/bank",
"root", {database password});
```

Using this connection, the web application is started from Java's main function using the following code snippets:

```
port(4567);
Database.connect();
CustomerController.setupRoutes(Database.getConnection());
// Other controllers
```

This connection is used to setup the routes for each controller. Each route will have a http method and path which will be fetched by the javascript. The following snippet is an example of a route:

```
get("/account/:id", (request, response) -> {
      response.type("application/json");
      int id = Integer.parseInt(request.params(":id"));
      return // some response;
});
```

The next code snippets are regarding CRUD operations and transactions. These operations are enclosed within a try catch blocks to catch any error during processing of the statements. Here's an example of an SQL statement being executed and retrieved within AccountDao.java:

```
String sql = "SELECT * FROM Accounts WHERE customerID = ?";
try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, id);
      ResultSet rs = pstmt.executeQuery();
```

```
        while (rs.next()) { accounts.add(new Account(//rs
attributes)); }
} catch (SQLException e) { e.printStackTrace(); }
```

When multiple queries are required for a specific feature, such as funds transfer for example, auto commit is set to false which signals the start of a transaction. When an error occurs, the SQLExeption is caught and the system sends a rollback command, reverting any changes during the transaction. If no error occurs, it sends a commit command. Finally, always return auto commit to true.

```
try {
        conn.setAutoCommit(false); // start transaction
        ! CRITICAL SECTION !
        // multiple prepared statements and queries
        conn.commit(); // commit transaction
} catch (SQLException e) {
        conn.rollback(); // if something goes wrong, rollback
        e.printStackTrace();
} finally {
        conn.setAutoCommit(true); // return auto commit to true.
}
```

Error Handling

As much as possible, input data are filled, cleaned, and formatted correctly before executing queries. This avoids as much errors as possible. In the case where an SQLExeption or error was to occur it is caught but does not cause the server to stop or fail.

Testing and Validation

Test Cases:

- **Login:** User must be correctly authenticated given their email and password.
- **Update address or contact information:** changing address or contact information must reflect on the customer's information upon refresh.
- **Loan application:** applying for a loan must add entry in the loan table.
- **Loan payment:** paying for a loan must reduce the balance of the loan,
- **Delete loan:** deleting a loan must remove the loan the database. It must also remove all related rows in the loan payment table.
- **Transfer funds:** transferring funds must add 2 entries in the transactions table and update the corresponding accounts.
- **Dashboards:** must retrieve accounts, transactions, or loans and display the information accurately.

- **Filter transaction:** filtering transactions must retrieve and display the correct subset of transactions. It must accommodate optional fields.

Sample Data: All tables were initialized with at least 15 entries and no null values**.** The following are the breakdown of each table:

- 15 customers with different names and information.
- 18 accounts with various balances. Some customers have two accounts, checking and savings.
- 31 transactions of different types and values.
- 16 loans of different types and values.
- 16 loanPayments as unique pairs of transaction and loan ids.

Test Results: Each feature worked as expected. All CRUD operations were performed and reflected the correct entries in the database; cascading delete works as intended when a loan is deleted; search filters returned accurate subsets even with partial conditions; error messages were displayed on invalid inputs.

Challenges and Solutions

One of the challenges was designing the schema of the database. As the application was developed, the initial schema we designed did not fit the needs of the application. We iteratively improved the schema to better suite the application. One example is date versus timestamp where timestamp was more suited for the application to correctly sort transactions on the same day.

Another challenge we faced was seamlessly integrating the presentation, logic, and database layers while learning the technologies on the go. We encountered new errors which took time and understanding to resolve. Certain errors were invisible especially those involving null values or queries that produce empty sets. Overtime, bug investigation was easier once those errors were encountered.

Future Enhancements

Improvements could also be done on scalability of the system by **allowing new customers to sign up** and setup their initial accounts. We may also add another user type, employee, which could perform actions such as approve customer loans.

As with interest rates, the system may **add earned interest on accounts and loans** when the month ends. This requires thorough checking to ensure it is working as intended.

Regarding utility, the main **dashboard could display more insights** regarding the user's accounts such as their balances over a course of time. The user interface could give more meaningful feedback to the users whenever an error occurs.

Conclusion

In summary, we had **designed and built a banking database system** will all of the intended key functionalities. We **learned how to design a database** that would fit our user's use case and learned **how to make an API that utilizes JDBC for the backend**. These were achieved with us collaborating during design, reporting, and development. Creating a database system was a learning experience. Learning new technologies has its learning curve but the knowledge and skills we gained are useful for our careers.

If we were to redo the project, we would have designed the system more carefully and thoroughly to have an easier time implementing the system. In conclusion, the project helped reinforce the topics that we have learned throughout the semester.

Appendix

Appendix A: Database Schema SQL

```sql
CREATE TABLE Customers (
  customerID   INT            NOT NULL AUTO_INCREMENT,
  firstName    VARCHAR(30)    NOT NULL,
  lastName     VARCHAR(30)    NOT NULL,
  address      VARCHAR(40)    NOT NULL,
  phone        CHAR(10)       NOT NULL CHECK (phone REGEXP '^[0-9]{10}$'),
  email        VARCHAR(30)    NOT NULL CHECK (email LIKE '%_@_%._%'),
  password     VARCHAR(30)    NOT NULL,
  PRIMARY KEY (CustomerID)
);
```

Appendix B: Sample Queries

Create and Update Example (Transfer Funds)

```
"UPDATE Accounts SET balance = balance - ? WHERE accountID = ? AND
customerID = ?"

"UPDATE Accounts SET balance = balance + ? WHERE accountID = ?"

"INSERT INTO Transactions (transactionType, accountID, timestamp, amount)
VALUES ('SENT', ?, CURRENT_TIMESTAMP, ?)"

"INSERT INTO Transactions (transactionType, accountID, timestamp, amount)
VALUES ('RECEIVED', ?, CURRENT_TIMESTAMP, ?)"
```

Retrieve Example (Account Transactions)

```
SELECT * FROM Transactions t INNER JOIN Accounts a ON t.accountID =
a.accountID
WHERE a.customerID = ? AND t.accountID = ? AND t.timestamp BETWEEN ? AND ?
AND t.amount BETWEEN ? AND ? ORDER BY t.timestamp DESC"
```

Retrieve Example 2 (Login Authentication)

```
"SELECT * FROM Customers WHERE email = ? AND password = ?"
```

Delete Example (Loan Deletion)

```
"DELETE FROM Loans WHERE loanID = ? AND customerID= ? AND balance = 0"
```