UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

Bank Management System

Submitted by:

Anirudh Lakhotia	**PES1UG21CS086**
Anirudh Revanur	**PES1UG21CS087**
Anoosh Damodar	**PES1UG21CS092**
Anshul Baliga	**PES1UG21CS095**

6th Semester B Section

Prof. Raghu B A Rao
Associate Professor

January - May 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement:

Our project is a Bank Management System, built using Java Spring Boot. It is a web application that allows users to manage the operations of a bank, its branches and its employees using a concise and user-friendly interface.

The system allows for the creation of new banks, new branches of a particular bank, adding bank employees to different branches and retrieving bank or branch details.

A customer can login and create their bank account and has access to a savings account, current account and loan account as a part of their main bank account. Customers can deposit or withdraw money through the system, check their current balance, take out loans and view their transaction history.

We also created an ATM interface where users can withdraw cash, view balance, etc.

Technologies used:

Backend: Java Spring Boot, Spring Data JPA, Maven, MySQL

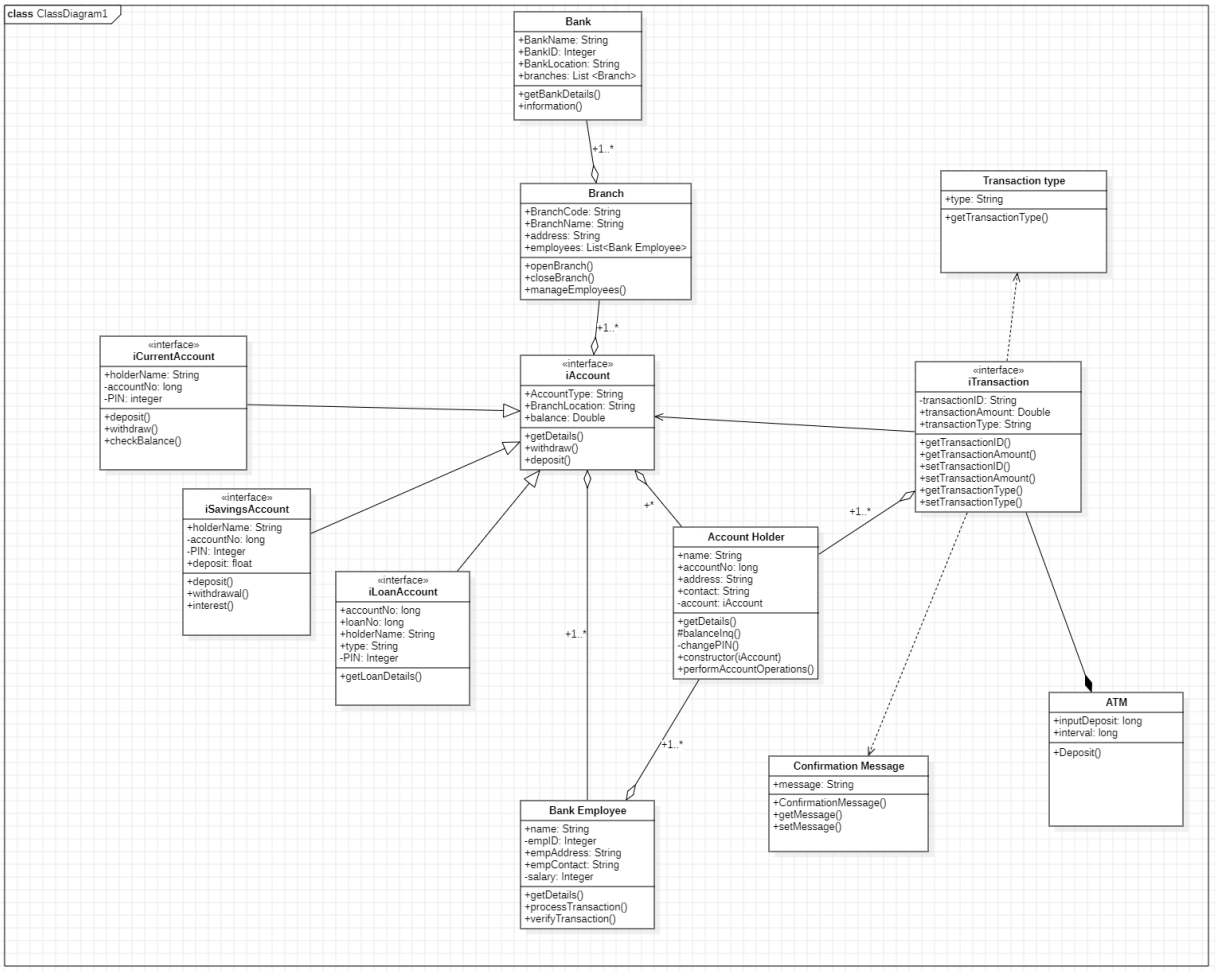
Frontend: HTML, CSS, AJAX

Models:

Class Diagram:

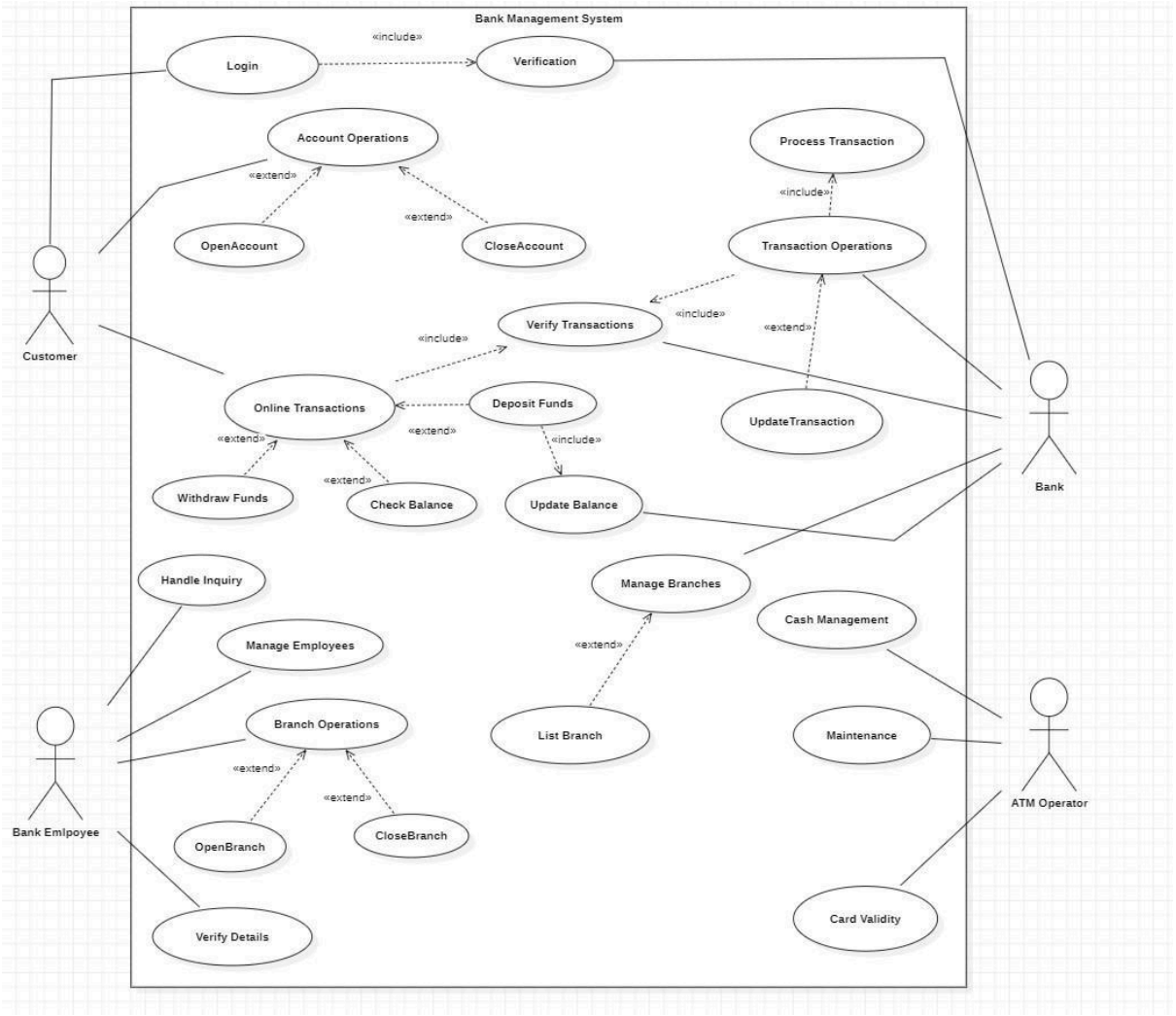
Classes:

Bank, Branch, Accounts, Current Account, Savings Account, Loan Account, Account holder, Transaction History, Online Transaction, Bank Employee, ATM Operator, Transaction Type, Confirmation Message



Use-case Diagram:

Banking operations for customers, bank employees, bank and ATM operators



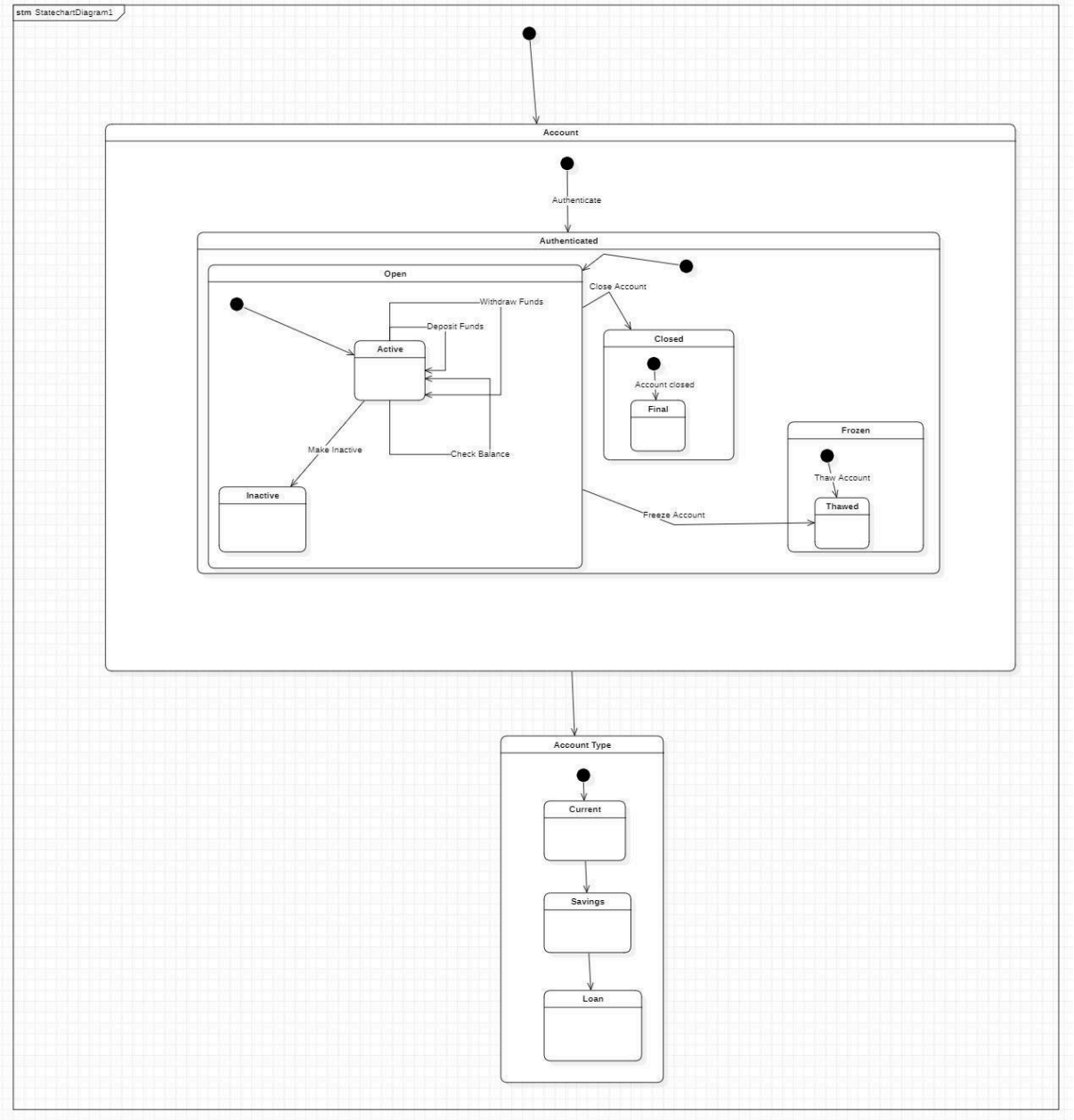
Individual Contributions of Team Members:

Functionality 1 (Done by Anoosh Damodar)

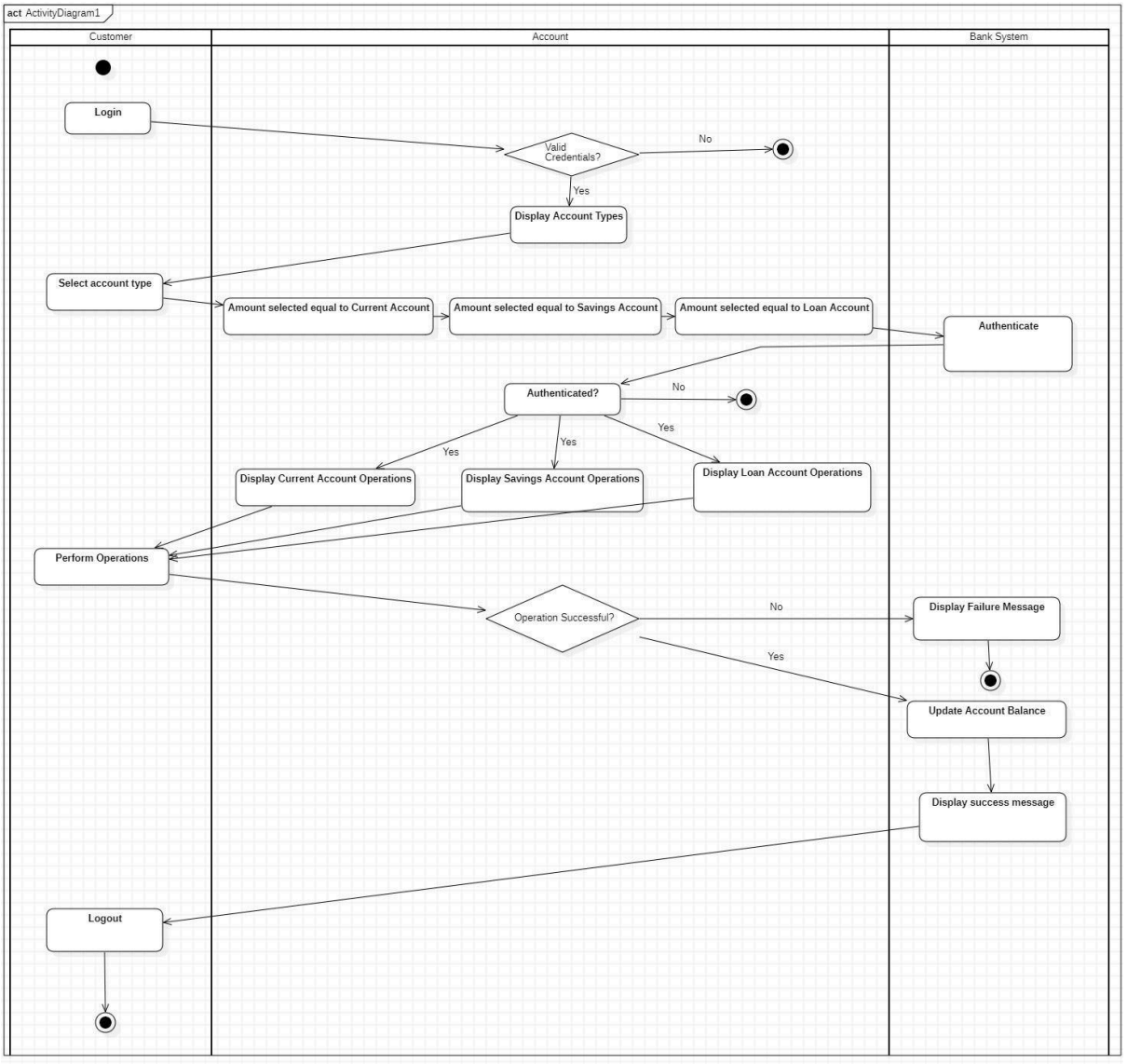
Account Operations

Managing accounts within the banking system, such as opening accounts, depositing funds, withdrawing funds, and checking balances.

State Diagram:



Activity Diagram:

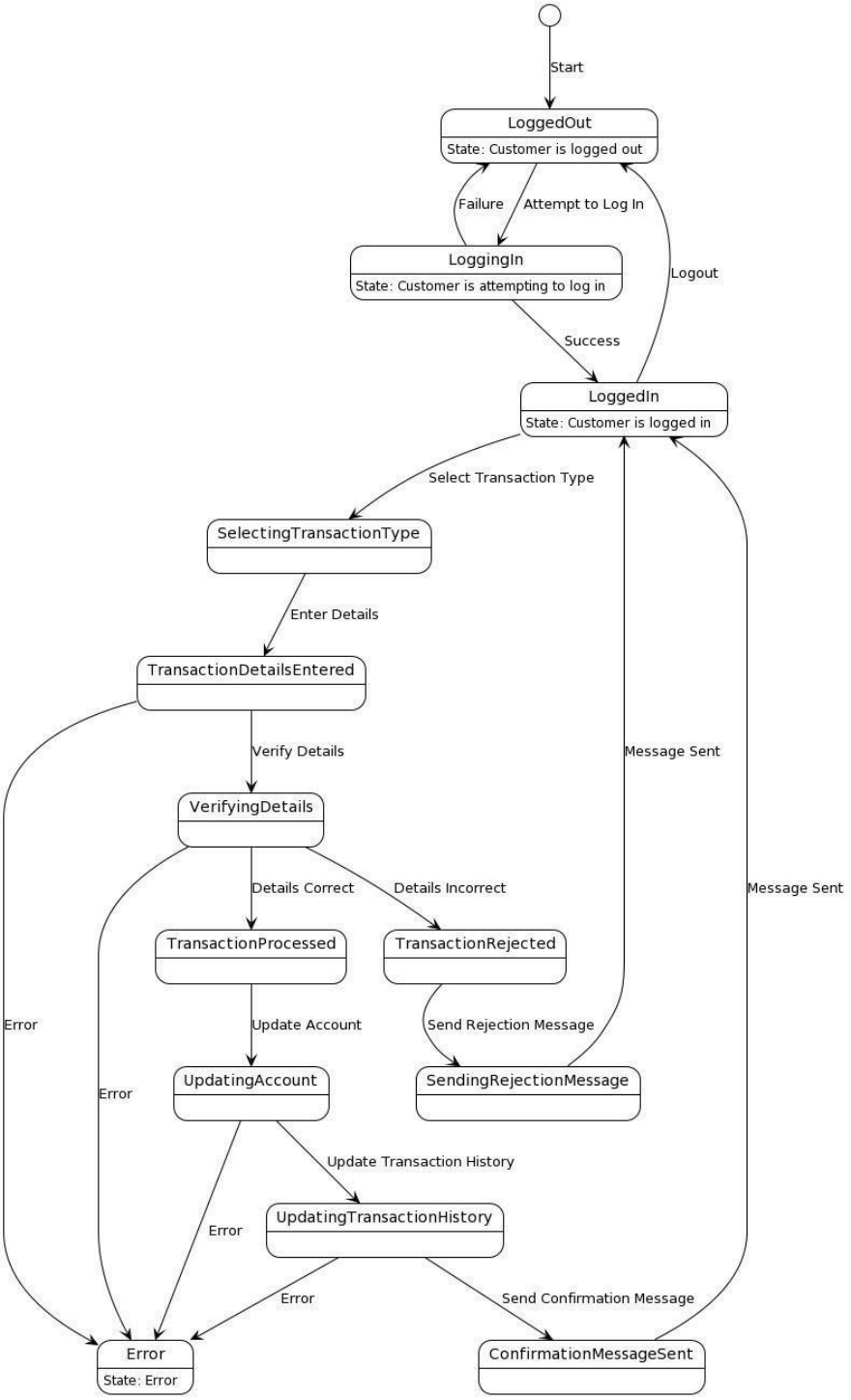


Functionality 2 (Done by Anirudh Revanur)

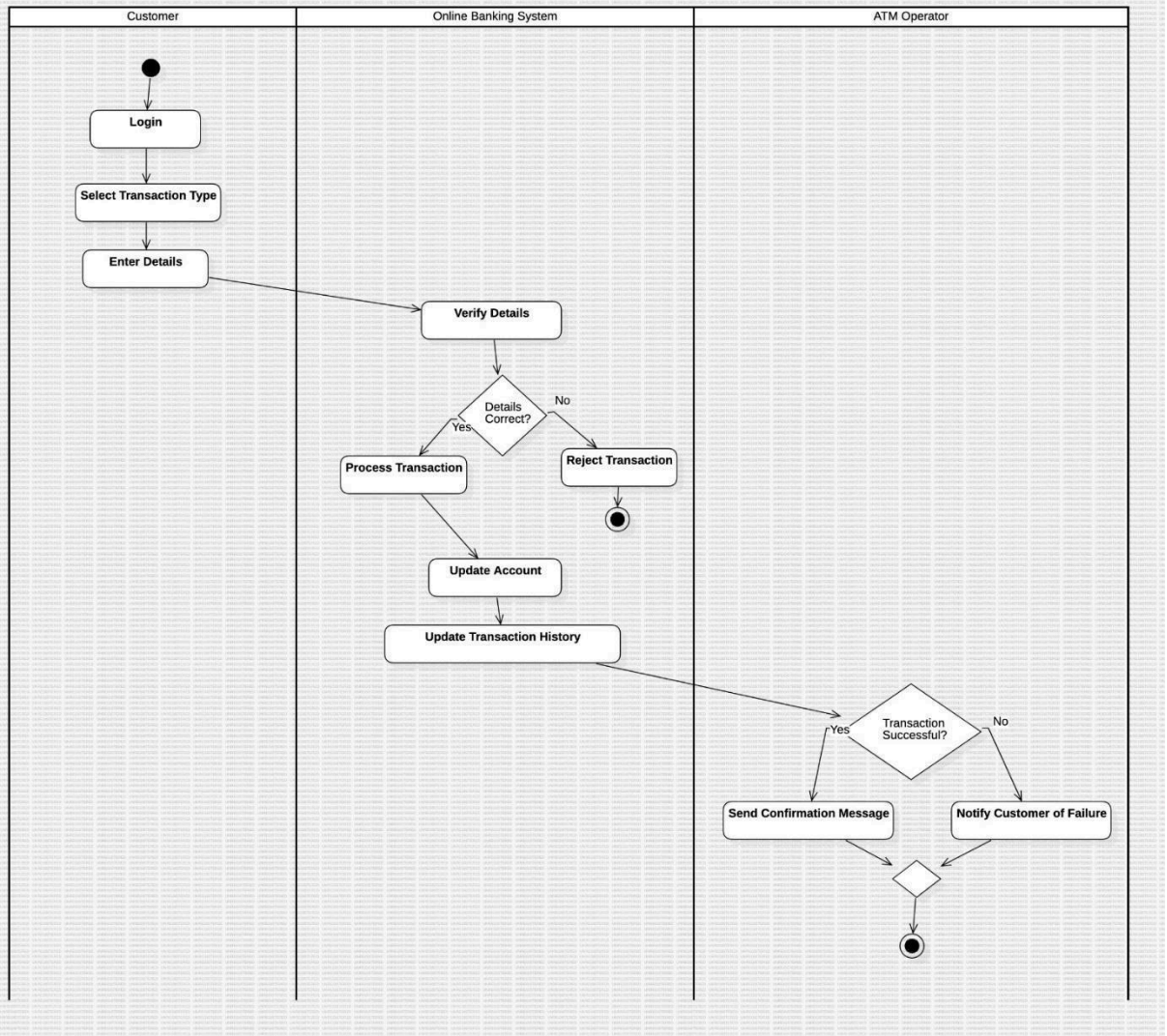
Online Transactions

Transferring funds, paying bills, viewing transaction history, and updating personal information.

State Diagram:



Activity Diagram:

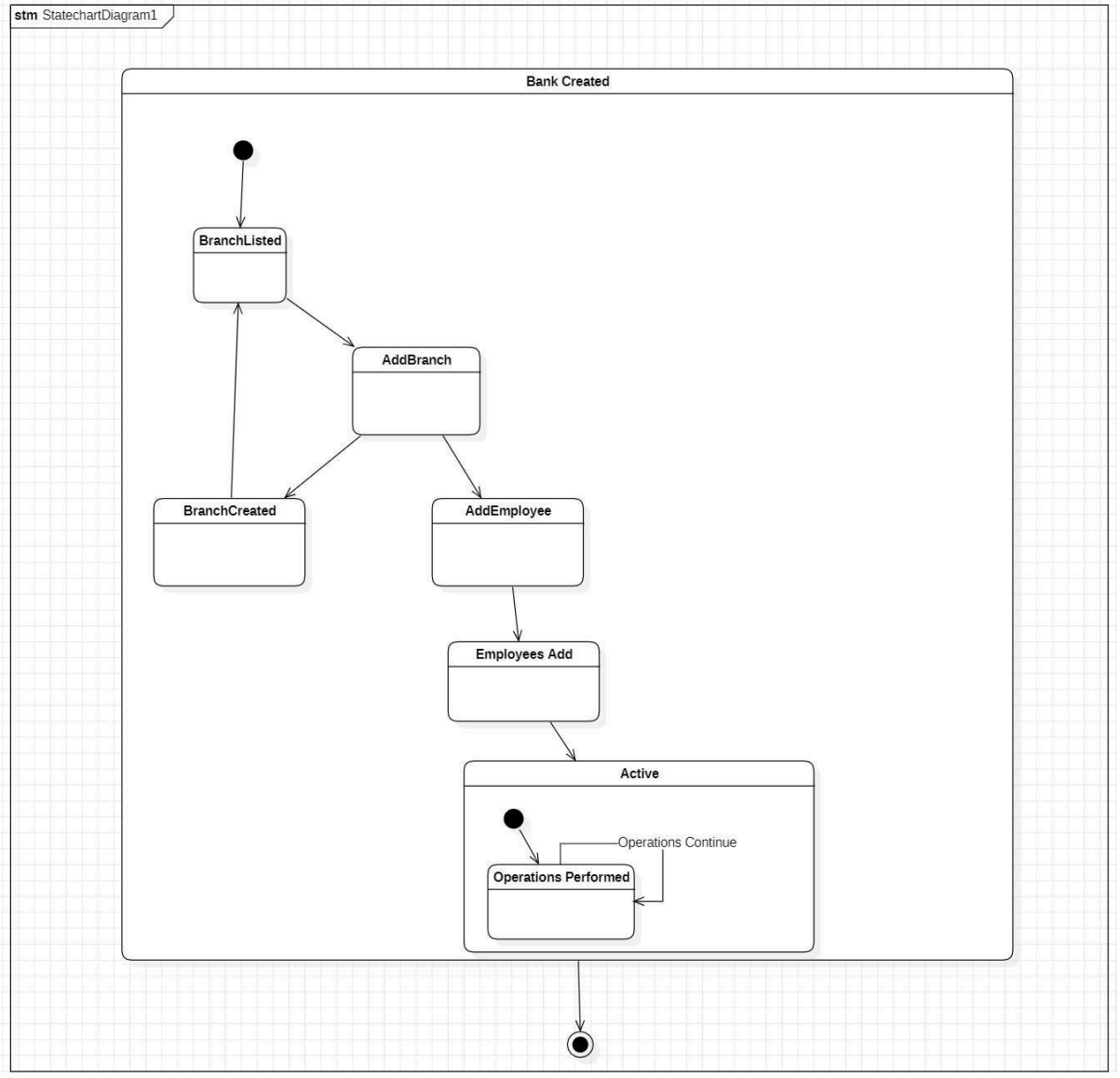


Functionality 3 (Done by Anshul Baliga)

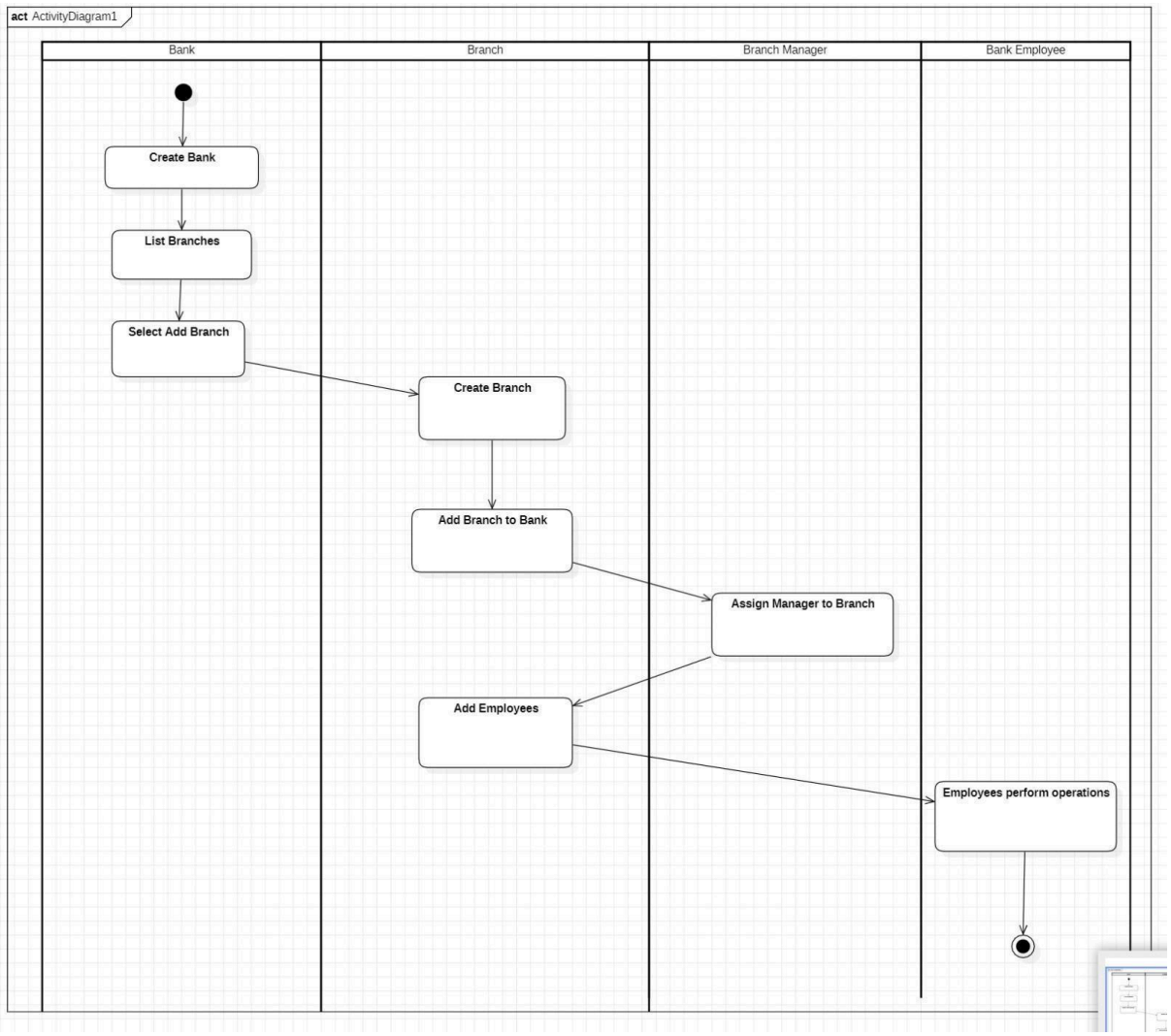
Branch Operations

Opening and closing branches, managing branch resources, displaying bank details and adding or removing employees from a branch

State Diagram:



Activity Diagram:

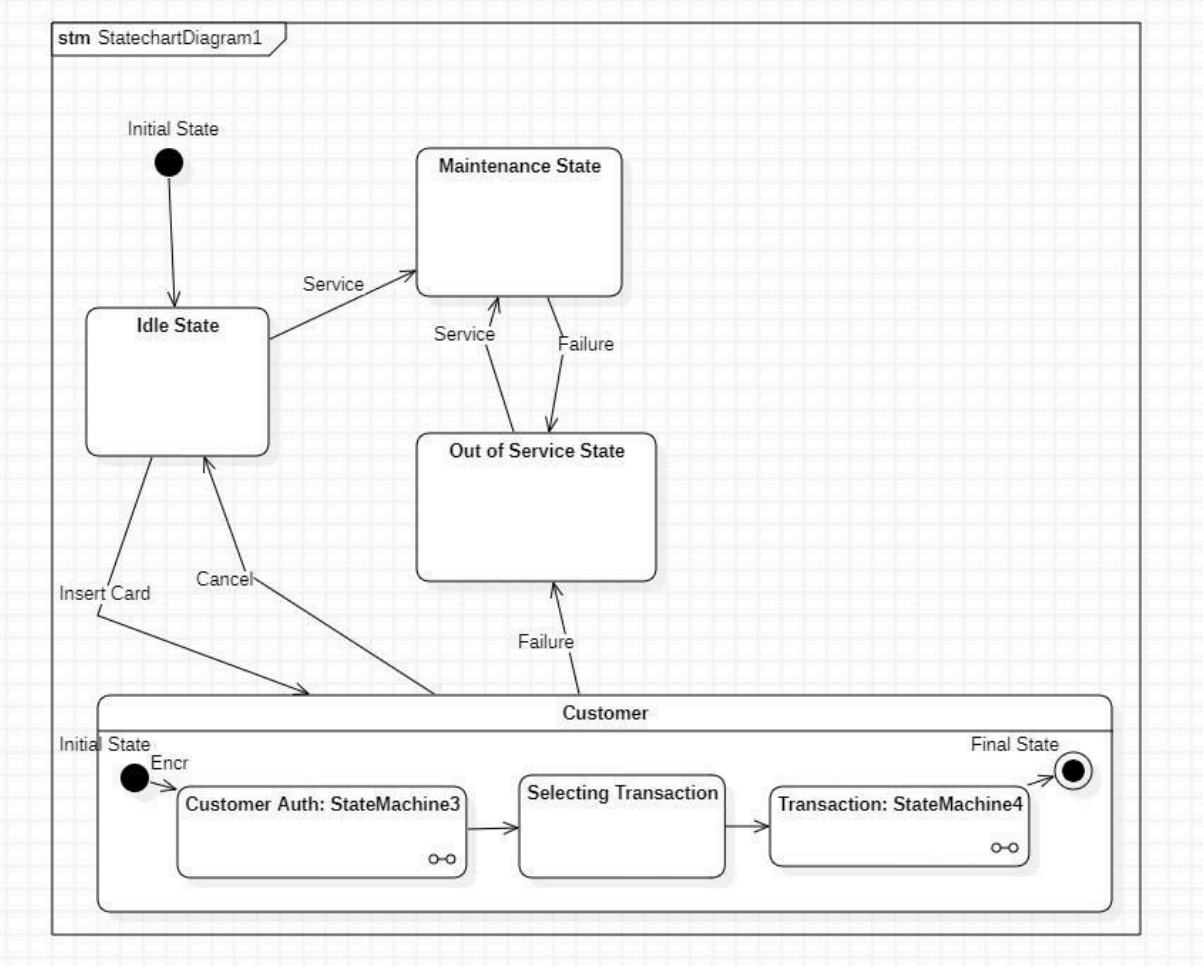


Functionality 4 (Done by Anirudh Lakhotia)

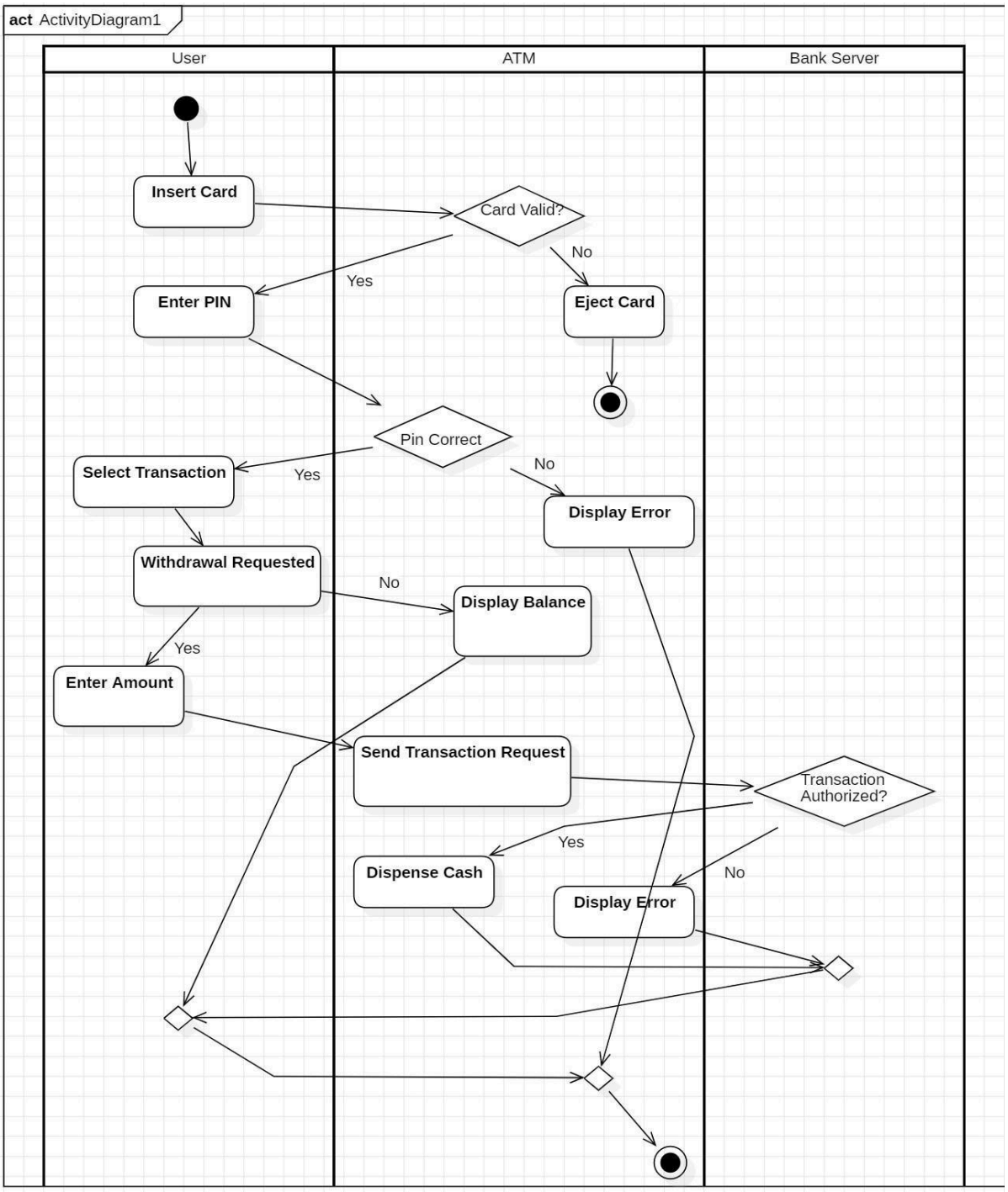
ATM Operations

Cash withdrawals, balance inquiries, and fund transfers outside of traditional bank branches

State Diagram:



Activity Diagram:



Architecture Patterns, Design Principles, and Design Patterns:

Architecture Patterns:

1. Layered Architecture:
    - In our Bank Management System, The `Bank`, `Branch`, and `AccountHandler` classes are a part of the business logic layer, while `iAccount` and its implementations belong to the model layer. `iTransaction` and its implementations could be part of a service or integration layer

2. Microservices Architecture:
    - Each major component (`Bank`, `Branch`, `AccountHandler`, `iTransaction`) is developed as a separate microservice, responsible for their own function

3. Model-View-Controller (MVC):
    - `Bank` and `Branch` are part of the Model, user interface screens as Views, and actions performed on the UI (like creating an account) as Controllers

4. Event-Driven Architecture (EDA):
    - Operations like transactions trigger events (`iTransaction` implementations) that are handled by different parts of the system, such as updating account balances or sending confirmation messages

5. Repository Pattern:
    - Each `iAccount` implementation has a corresponding repository class as shown in the ‘Repository’ folder in our github for CRUD operations.

Design Principles:

SOLID:

1. Single Responsibility Principle:
    - The Bank class manages bank-related information, while the Account class and its subclasses handle account-specific operations

2. Open/Closed Principle:
    - The use of inheritance for Account types (CurrentAccount, SavingsAccount, and LoanAccount) allow new account types to be added without modifying existing code

3. Liskov Substitution Principle:
    - `iSavingsAccount`, `iLoanAccount`, and `iCurrentAccount` are inherited from a common base class `iAccount` and can be used anywhere, for each user that signs up to our system

4. Interface Segregation Principle:
    - Clients should not be forced to depend on interfaces they do not use. Separate interfaces (`iSavingsAccount`, `iLoanAccount`, `iCurrentAccount`) ensure this

5. Dependency Inversion Principle:
    - Higher level classes, ‘Bank’ and ‘Branch’ depend on interfaces like lower level classes `iAccount` and `iTransaction`, not their implementations.

GRASP:

1. Information Expert:
    - Bank class has attributes like BankName, BankId, and BankLocation, which makes it an information expert for managing bank-related data and operations (getDetails(), manageInformation(), manageBranches())

2. Low Coupling:
    - The TransactionHistory class is responsible for managing transaction-related operations, while the AccountHolder class handles account holder details, which reduces the coupling between these classes

3. High Cohesion:
    - The Account class and its subclasses (CurrentAccount, SavingsAccount, and LoanAccount) exhibit high cohesion by encapsulating account-specific data and operations within their respective classes

4. Polymorphism:
    - The inheritance hierarchy of Account types (CurrentAccount, SavingsAccount, and LoanAccount) inheriting from the base Account class demonstrates polymorphism

5. Controller:
    - `AccountHandler` acts as a controller, handling the input and delegating work to other components.

Design Patterns:

❖ Creational Pattern:

1. Singleton Pattern:
    - In our Bank Management System, a `Bank` class representing the entire bank system acts as a singleton

2. Factory Method:
    - Makes interfaces act amongst each other, like `AccountHandler` might use this to create `iAccount` instances

3. Builder:
    - Used for creating complex `BankEmployee` objects with various attributes

❖ Structural Pattern:

1. Facade:
    - `AccountHandler` acts as a facade, simplifying account operations for the client

2. Flyweight:
    - Used for `ConfirmationMessage` objects that share similar data, when triggered

❖ Behavioral Pattern:

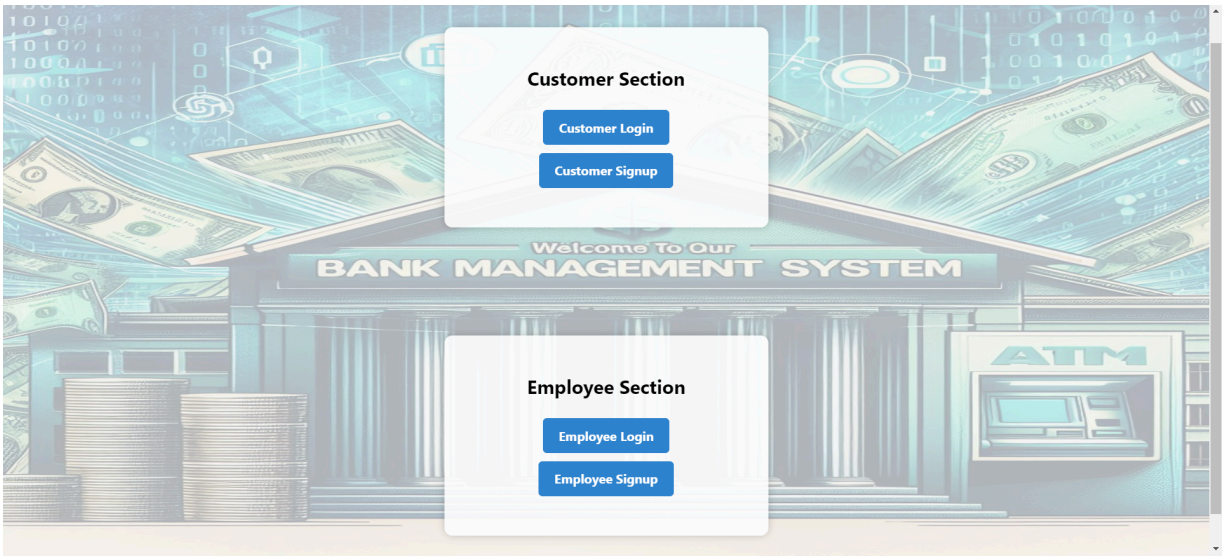
1. Observer:
    - In our system, we notify `BankEmployee` instances of changes in account statuses or transactions, sent to each ‘iAccount’.

Github link to the Codebase:

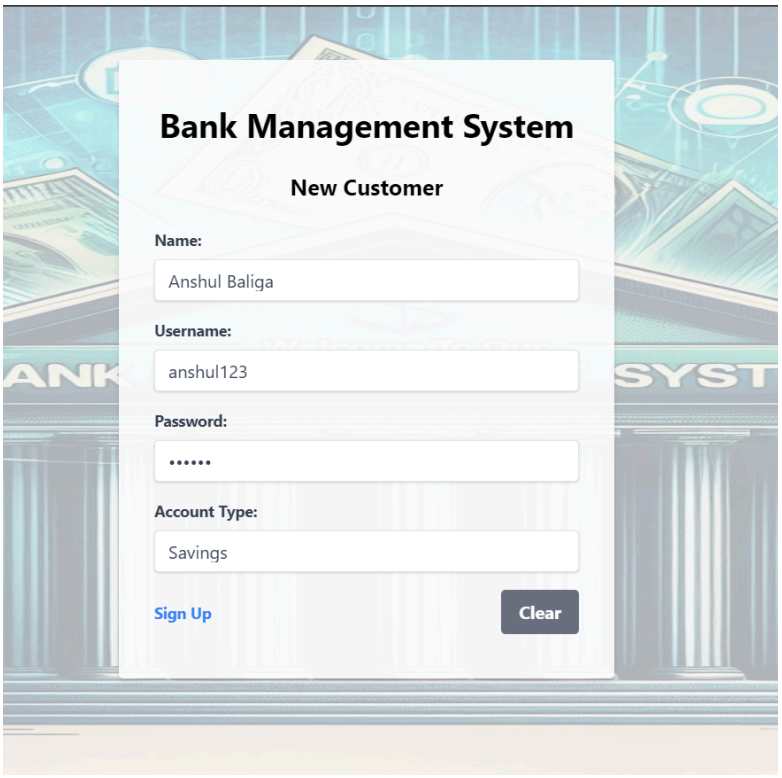
https://github.com/anooshd7/Bank-Management-System-Java-Spring

Screenshots with input values populated and output shown:

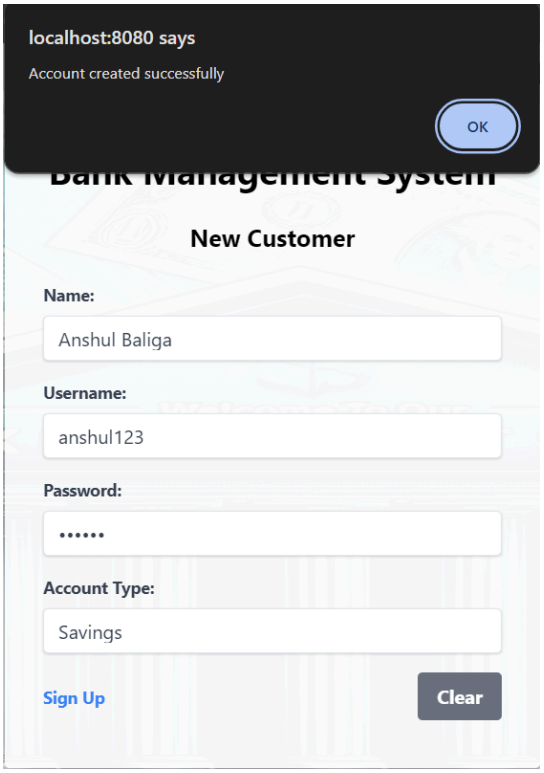
Home Page:



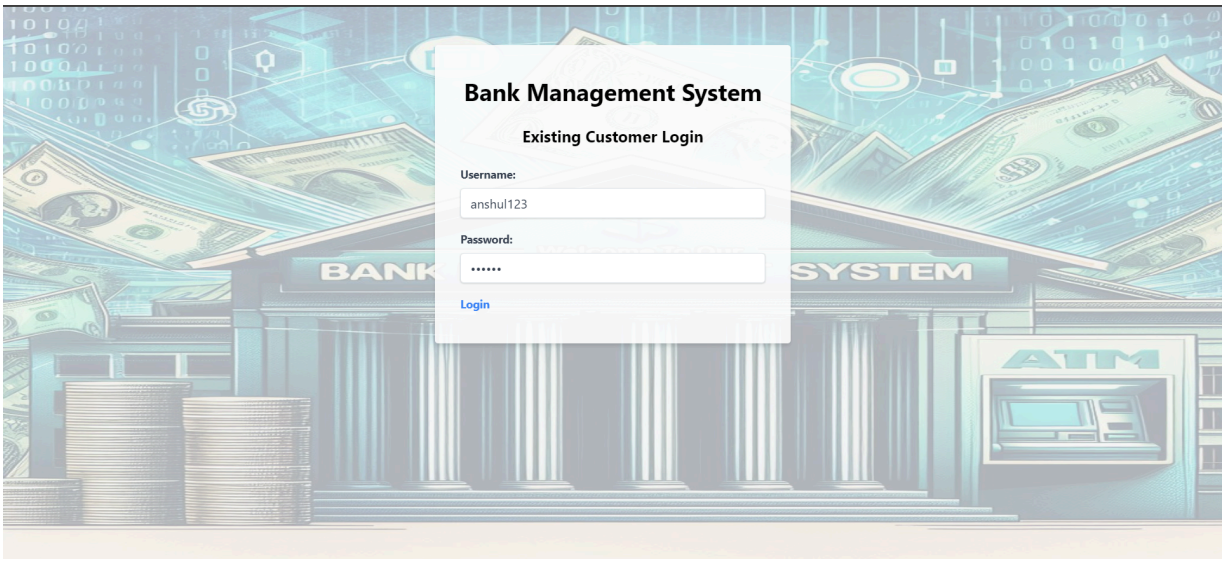
Customer Signup:



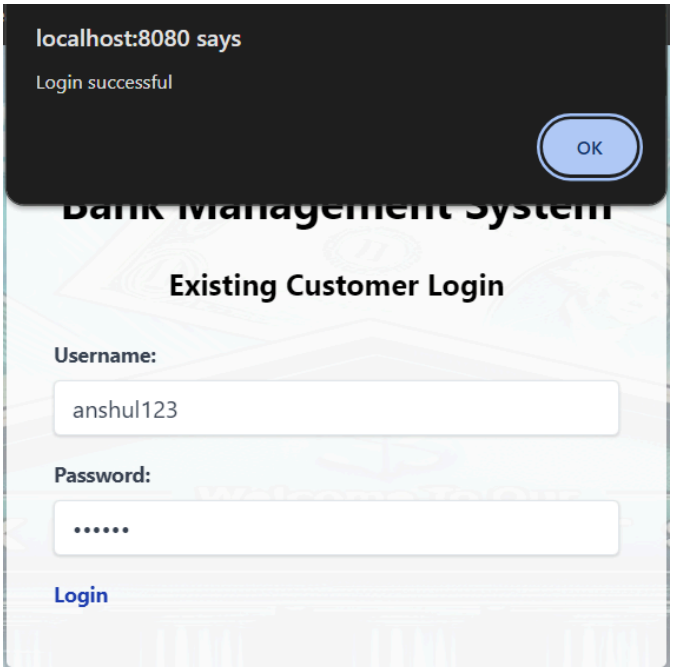
Confirmation message displaying ‘Successful Account Creation’:



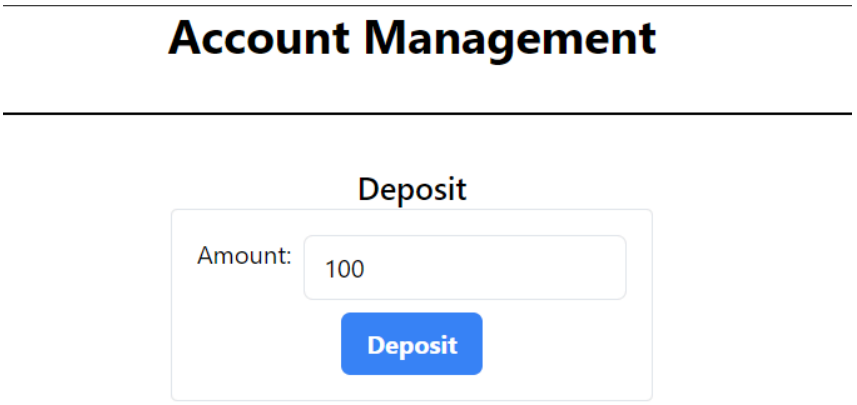
Customer Login:



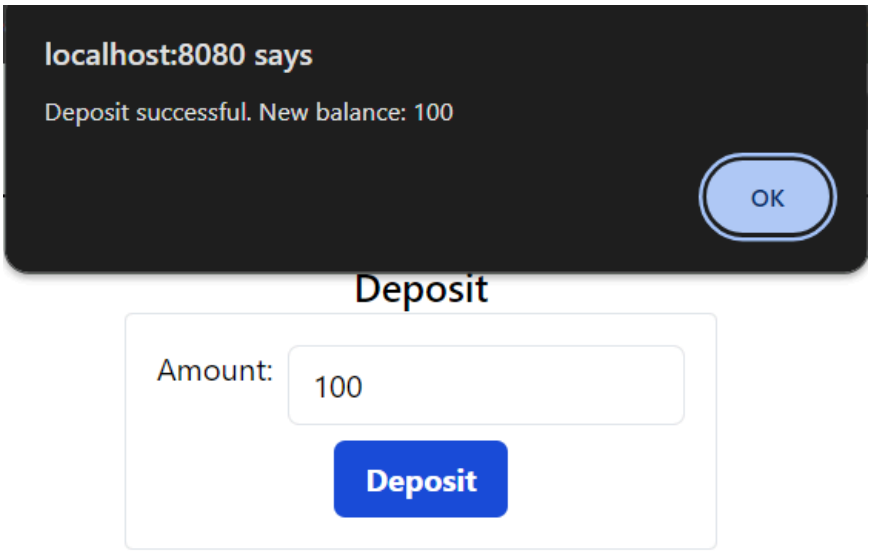
Confirmation messages displaying ‘Successful Login’:



Depositing money into an account:

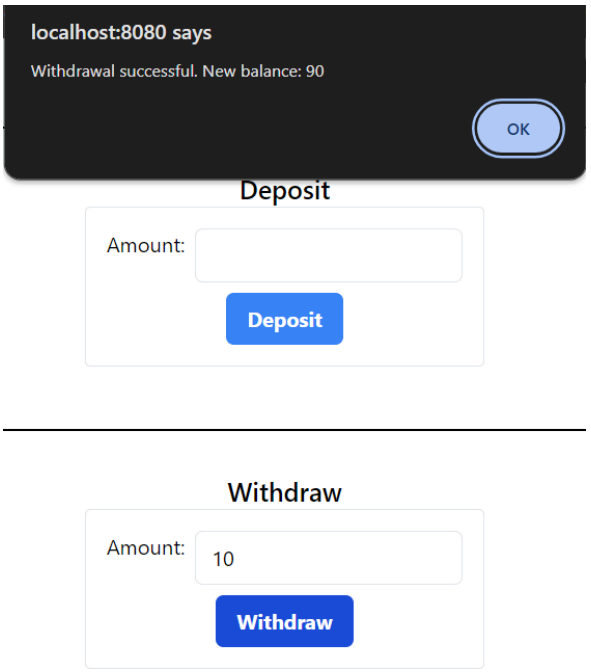


Confirmation message displaying ‘Successful Deposit’:

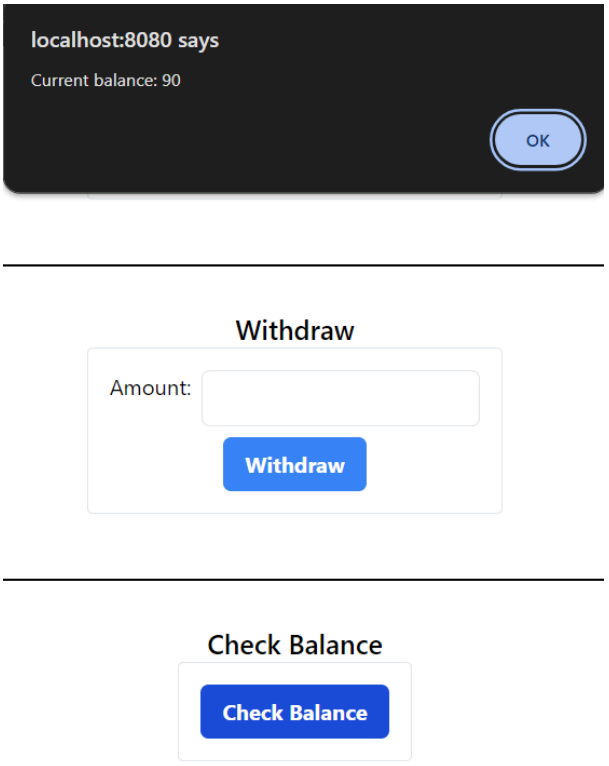


Withdrawing money:

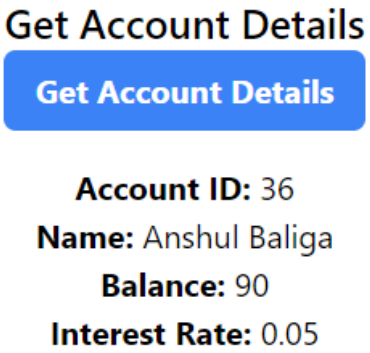
Confirmation message displaying ‘Successful Withdrawal’:



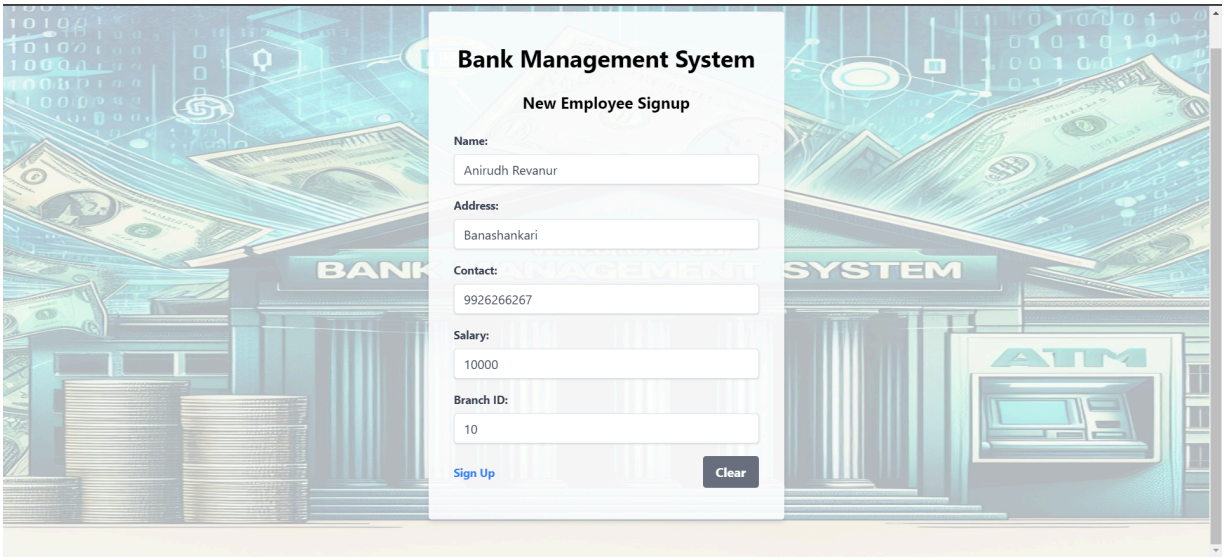
Checking account balance:



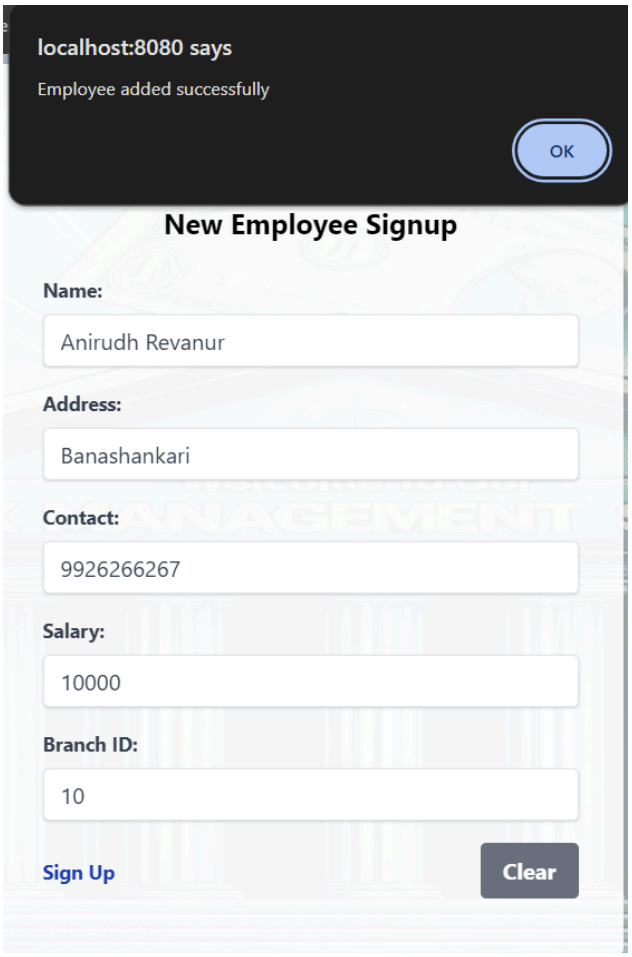
Display account details:



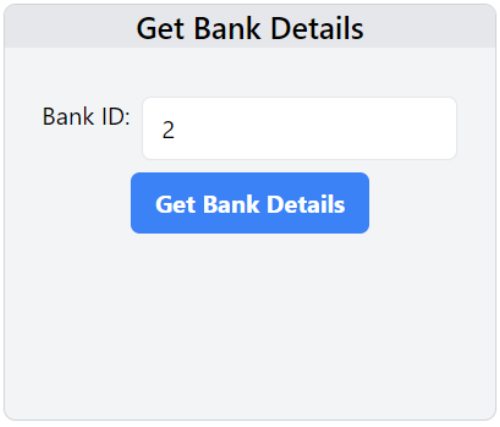
Employee Signup:

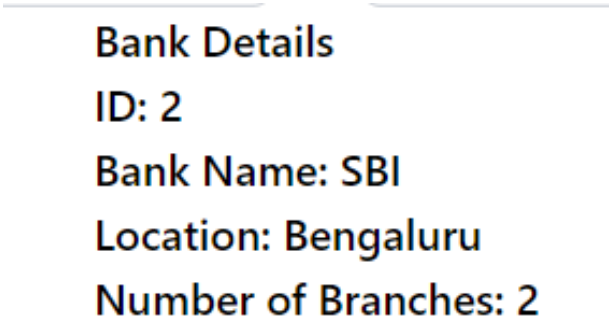


Confirmation message displaying ‘Successful addition of Employee’:

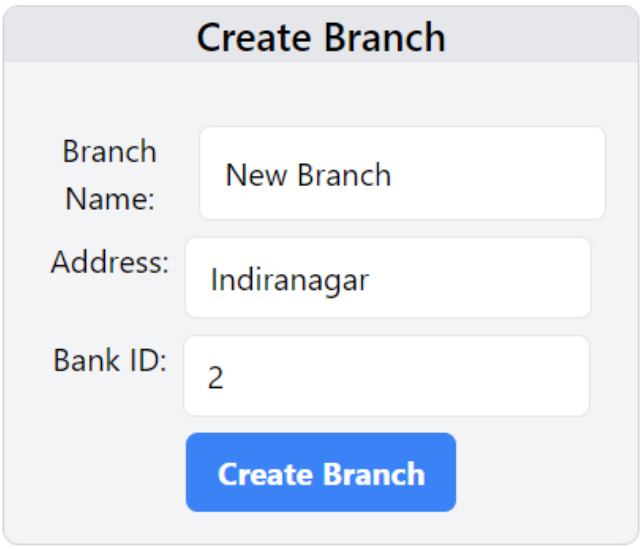


Display bank details:

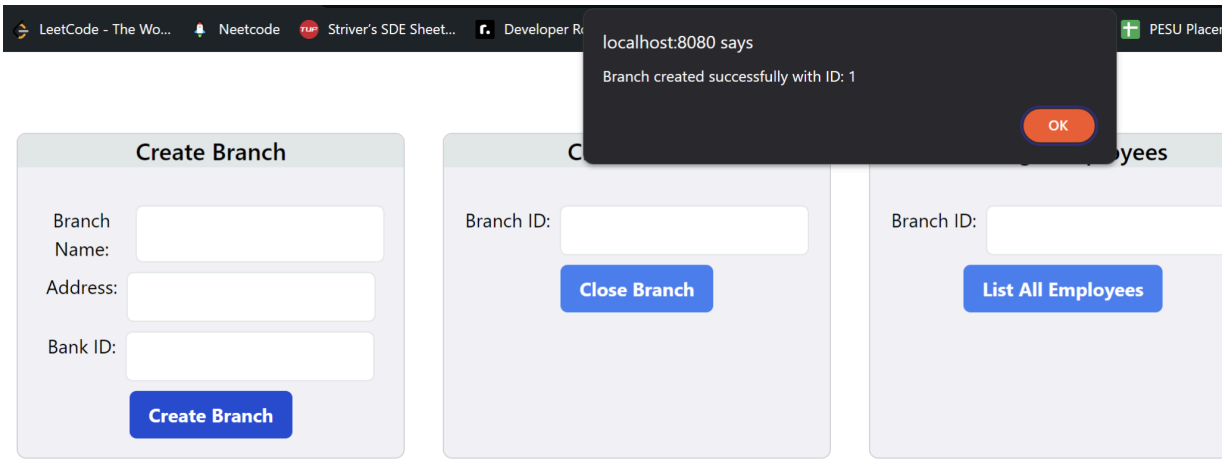




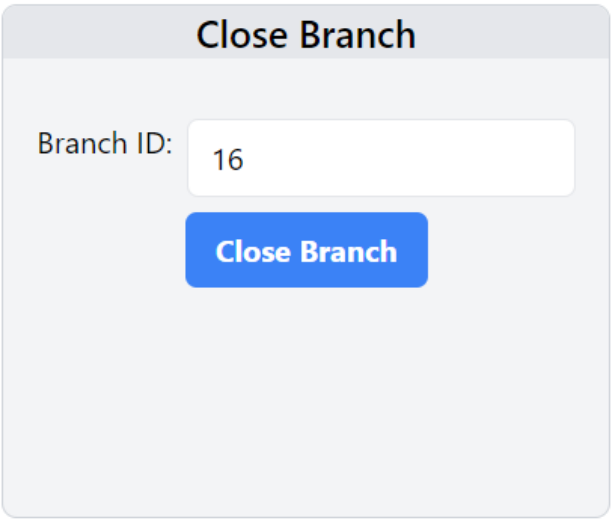
Creating a branch:



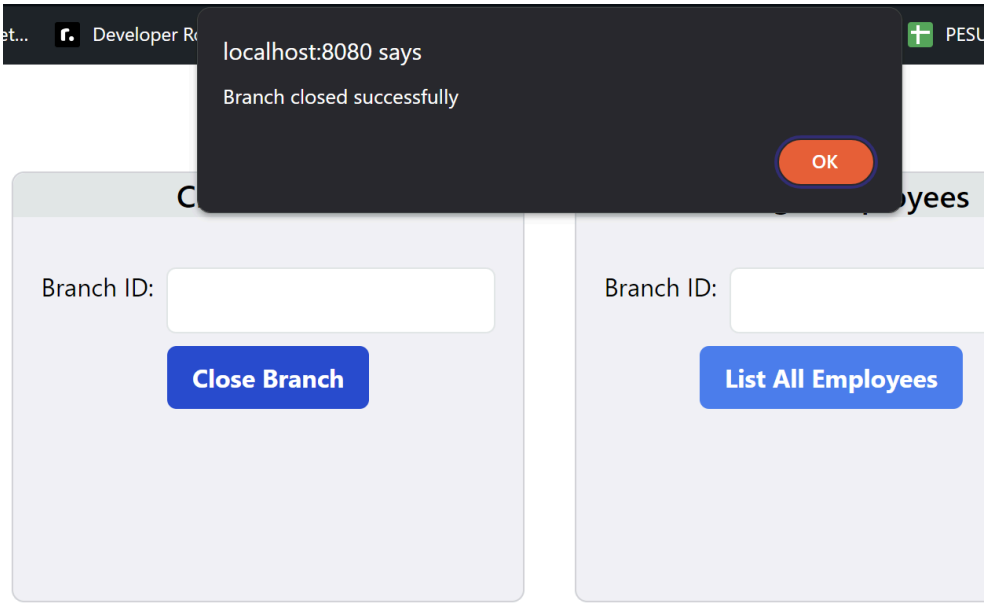
Confirmation message displaying ‘Successful creation of Branch’:



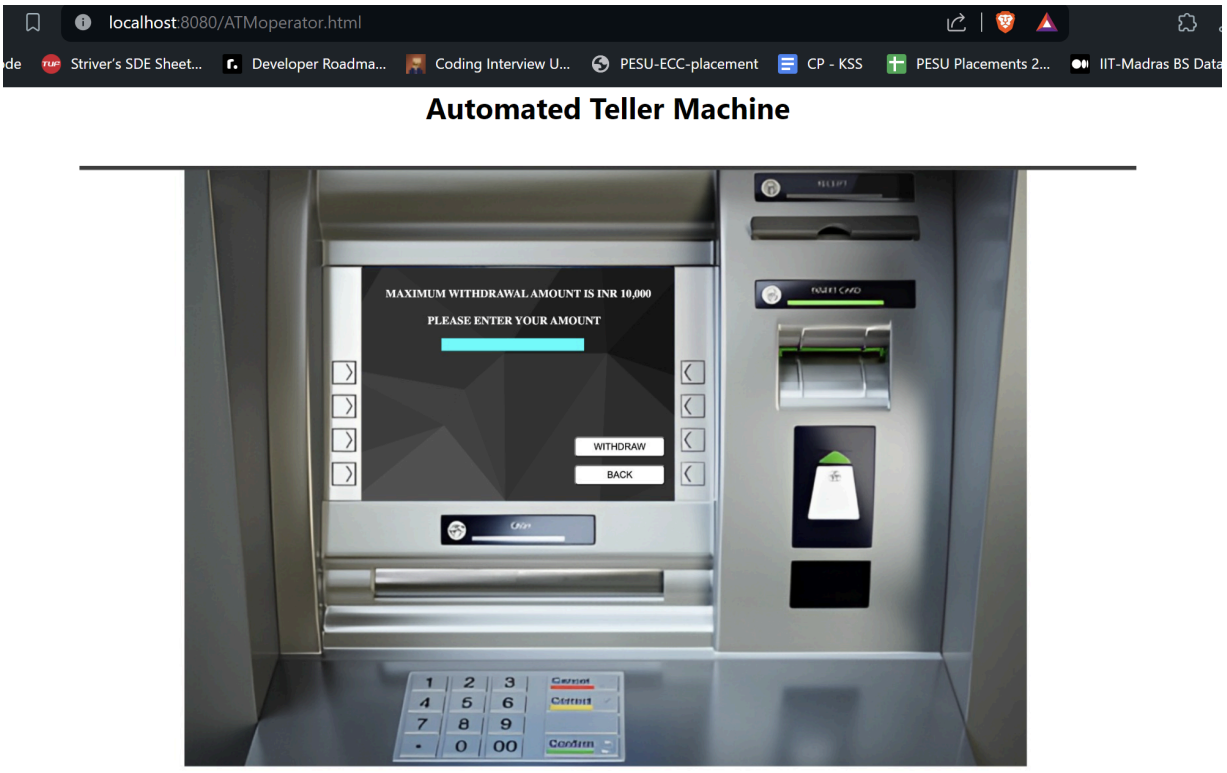
Closing a branch:



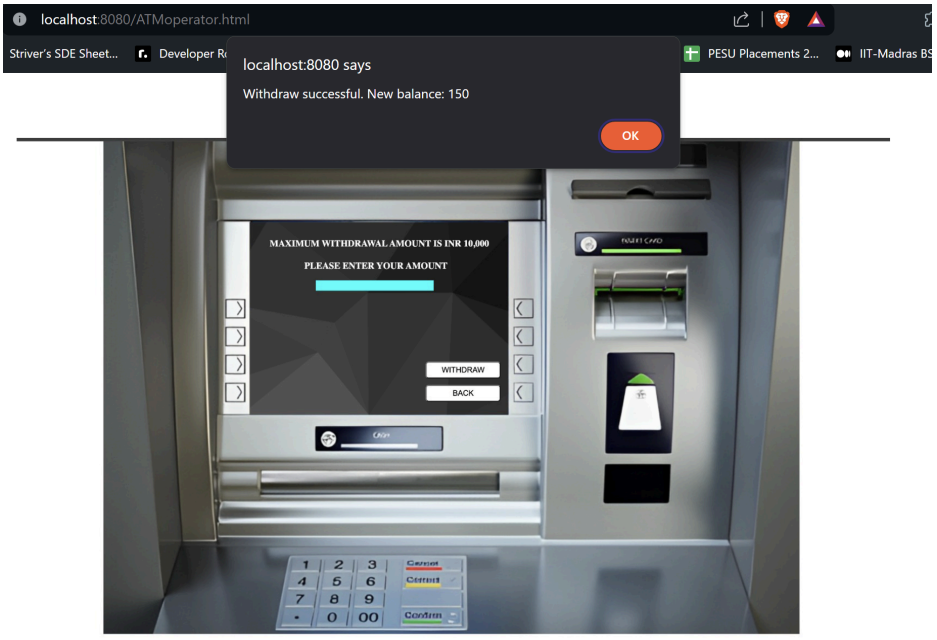
Confirmation message displaying ‘Successful closing of Branch’:



Depositing in the ATM:

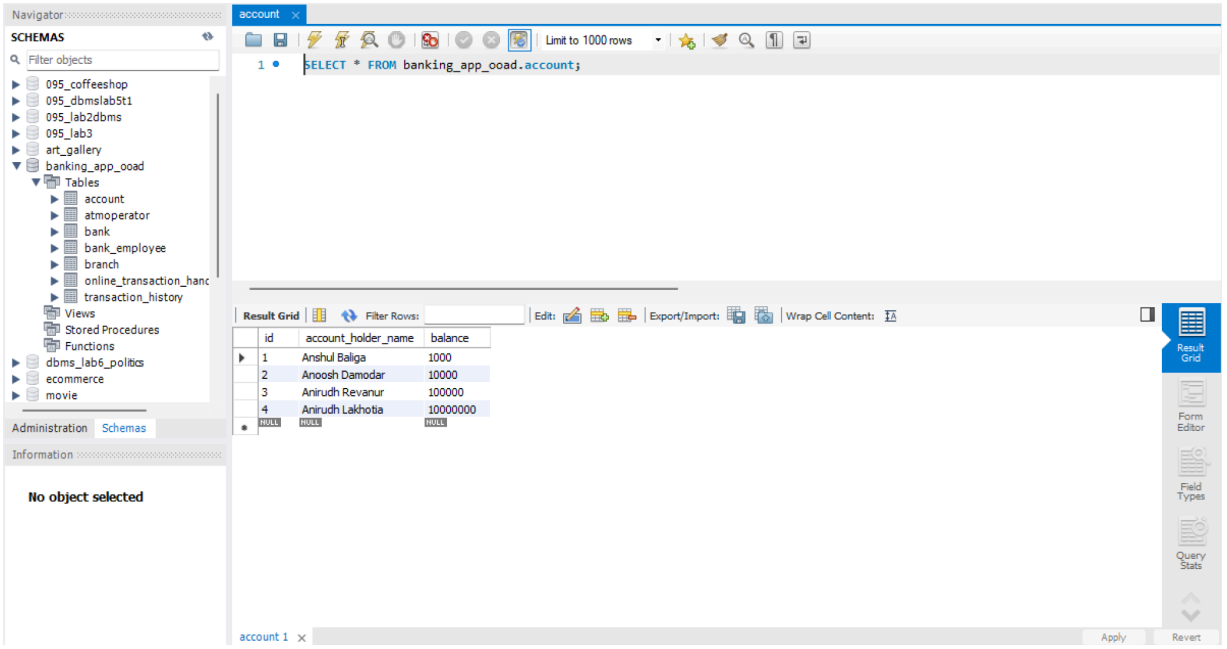


Confirmation message displaying ‘Successful Withdrawal’:



Database management:

Stores all the accounts entered:



Store Bank Details: