



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

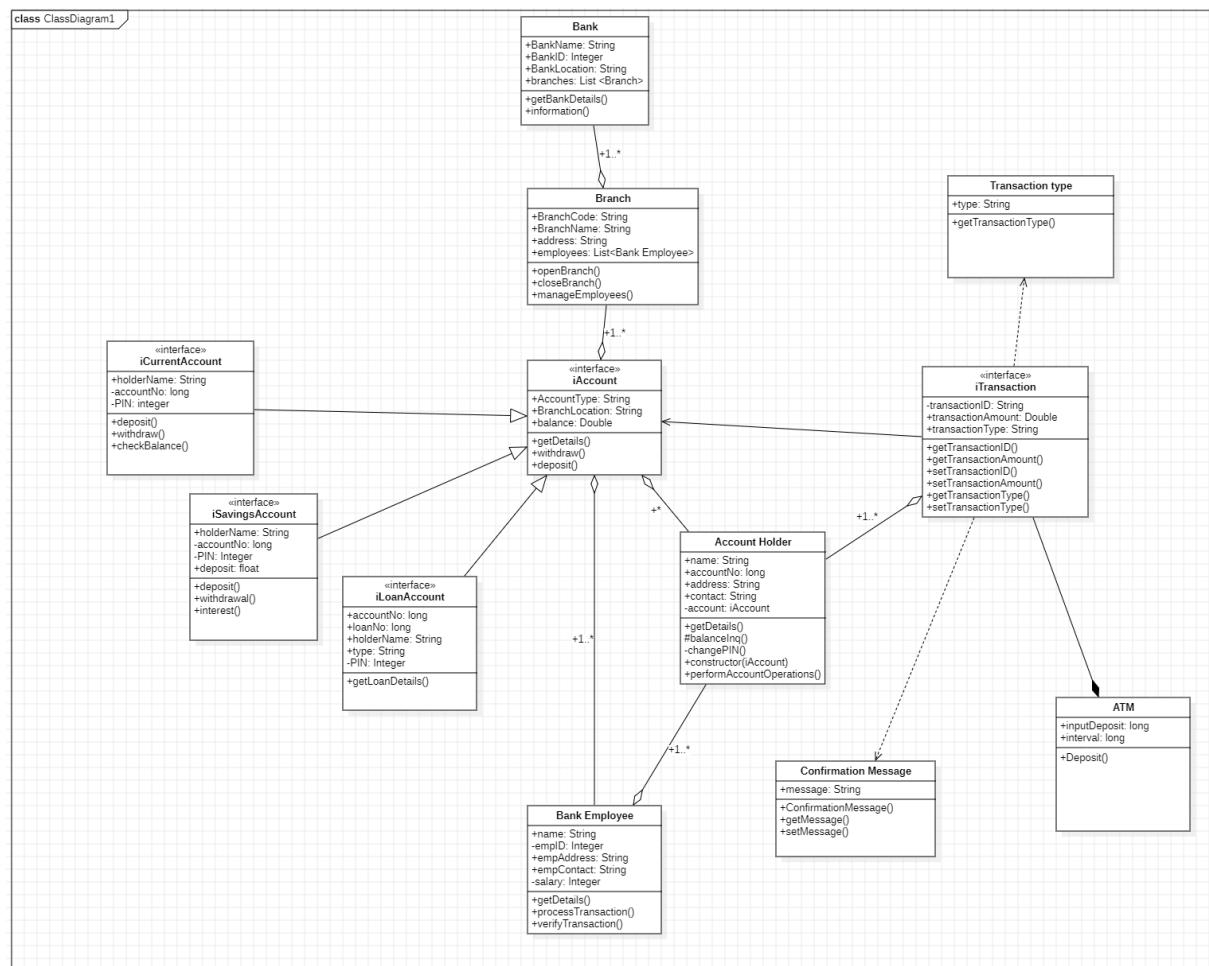
Frontend: HTML, CSS, AJAX

Models:

Class Diagram:

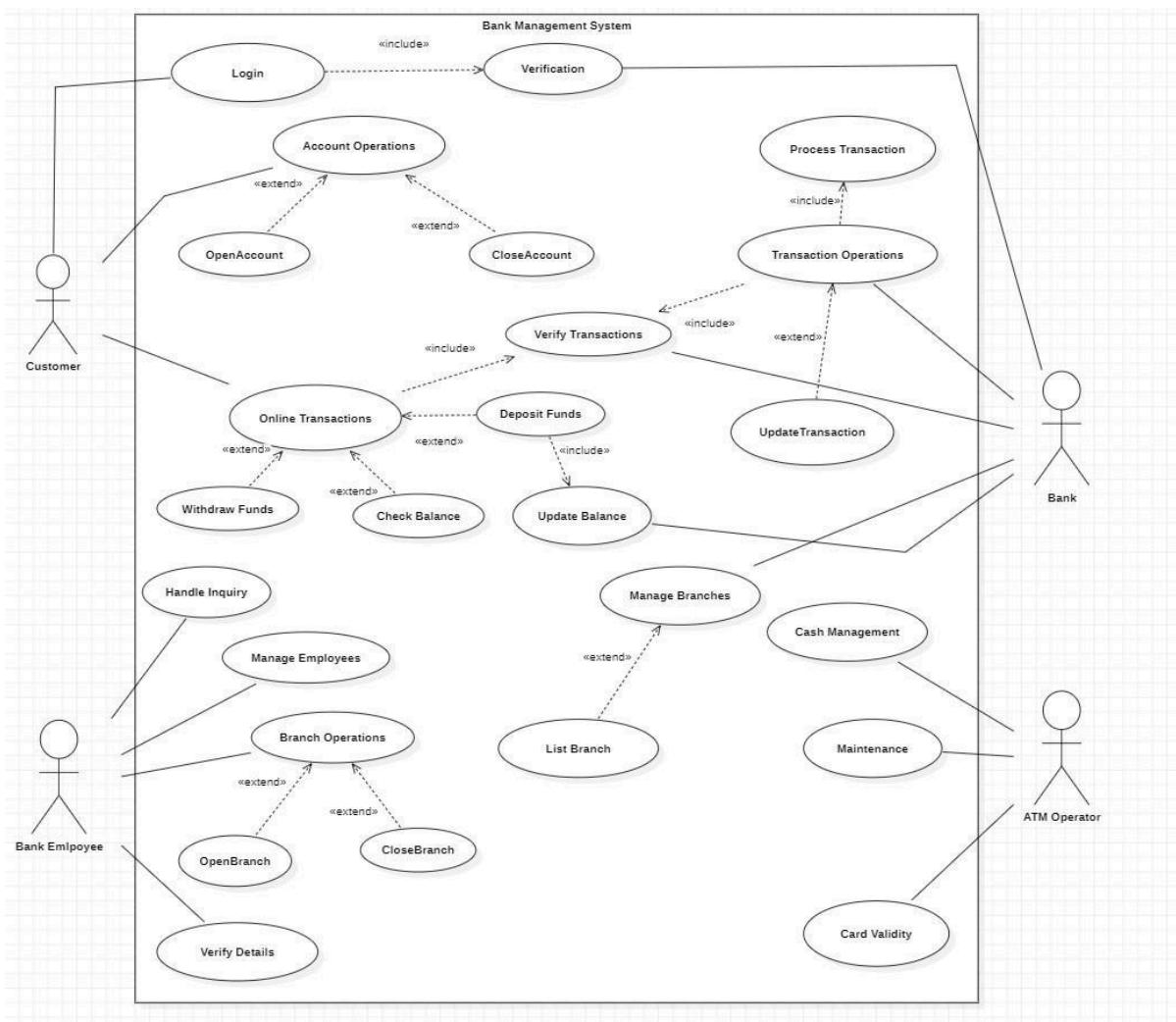
Classes:

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



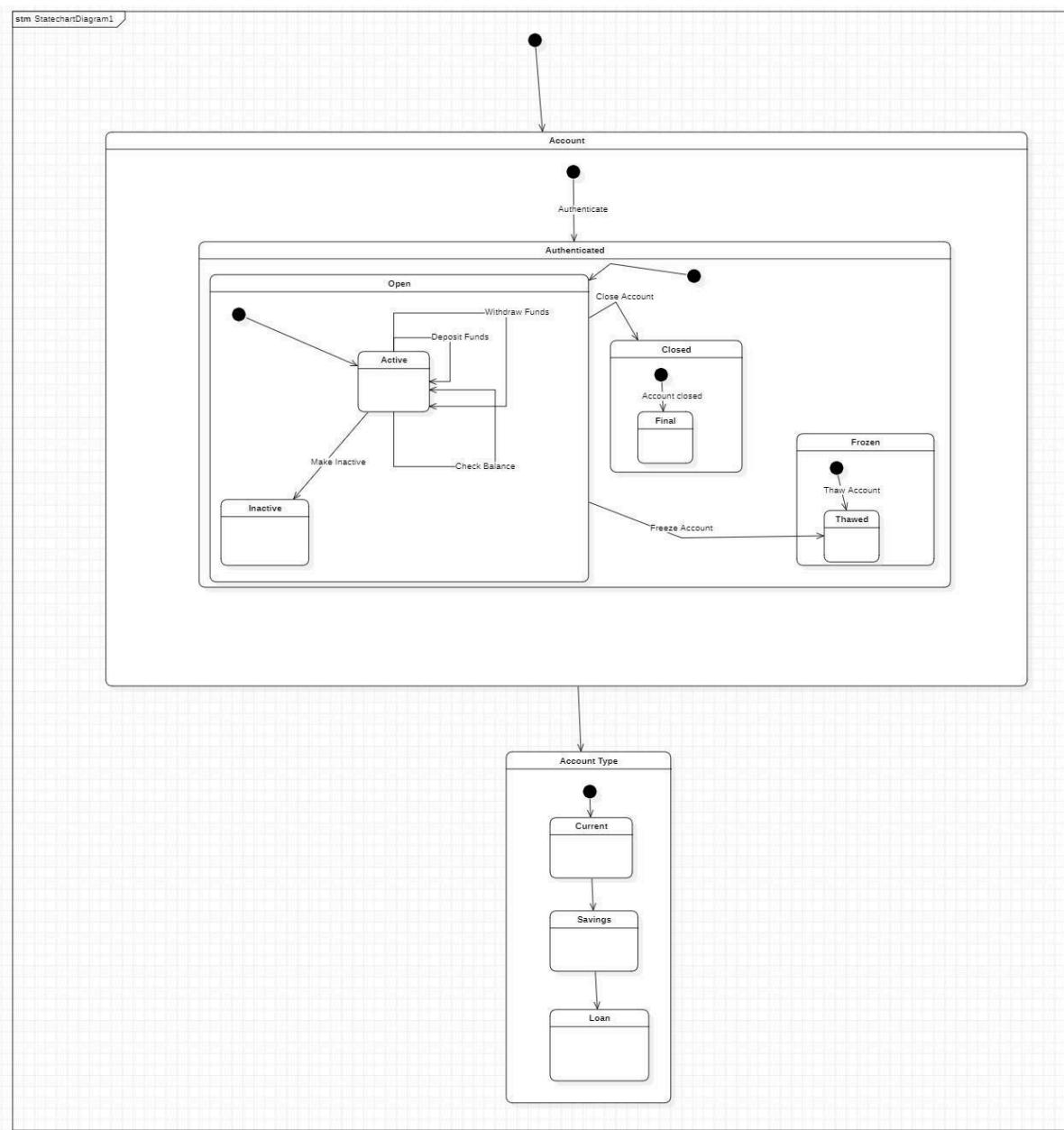
Individual Contributions of Team Members:

Functionality 1 (Done by Anoosh Damodar)

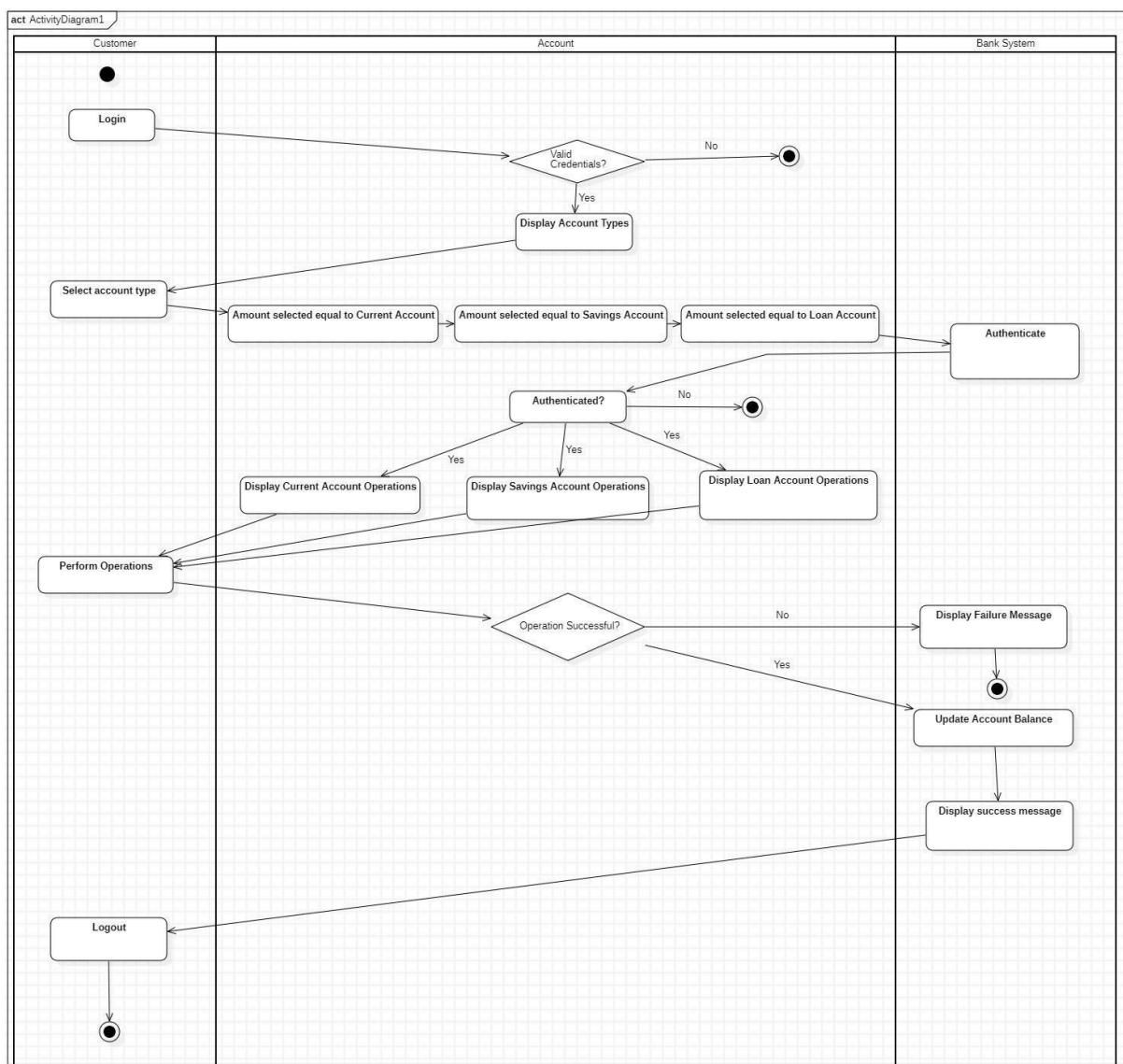
Account Operations

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

State Diagram:



Activity Diagram:

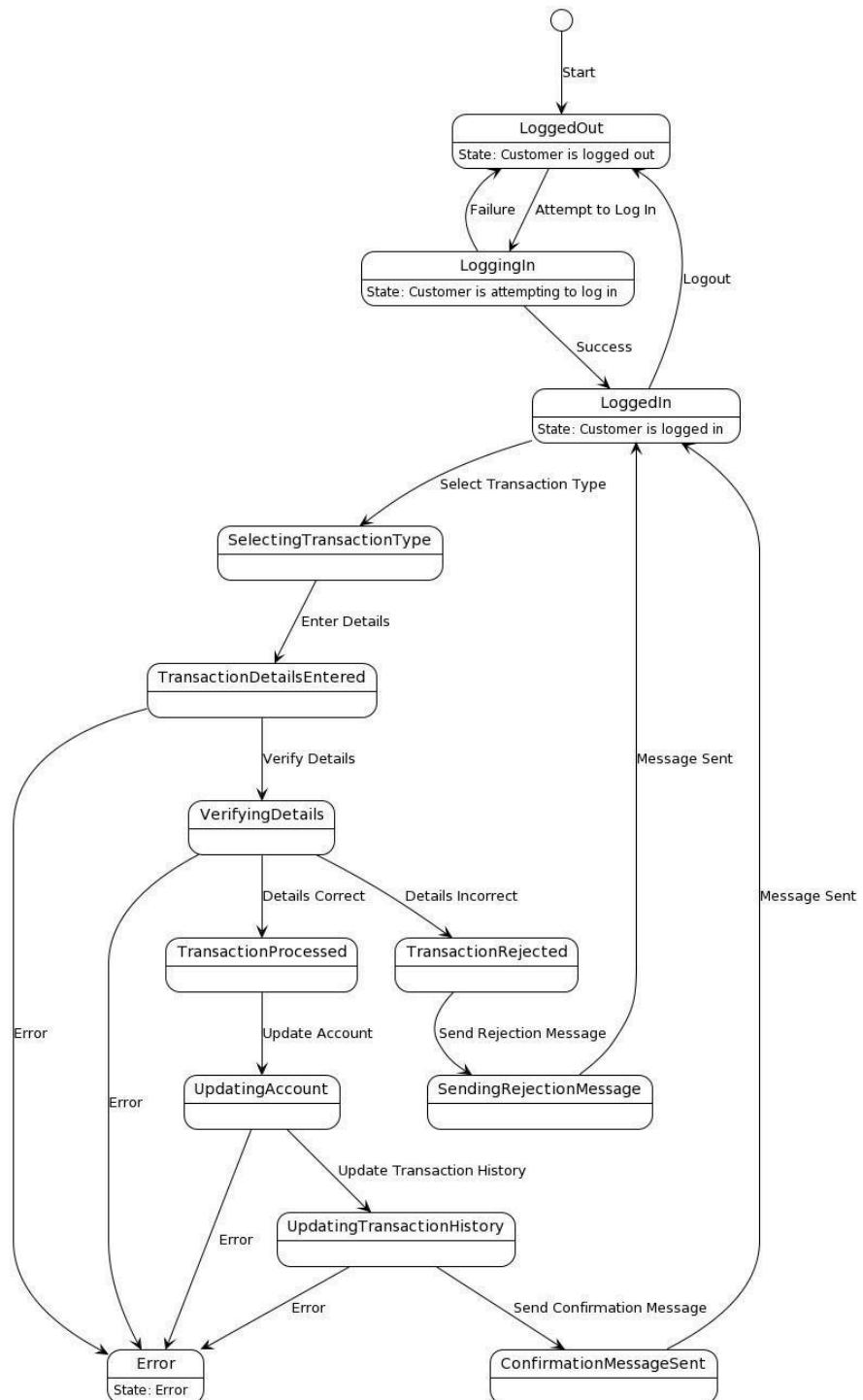


Functionality 2 (Done by Anirudh Revanur)

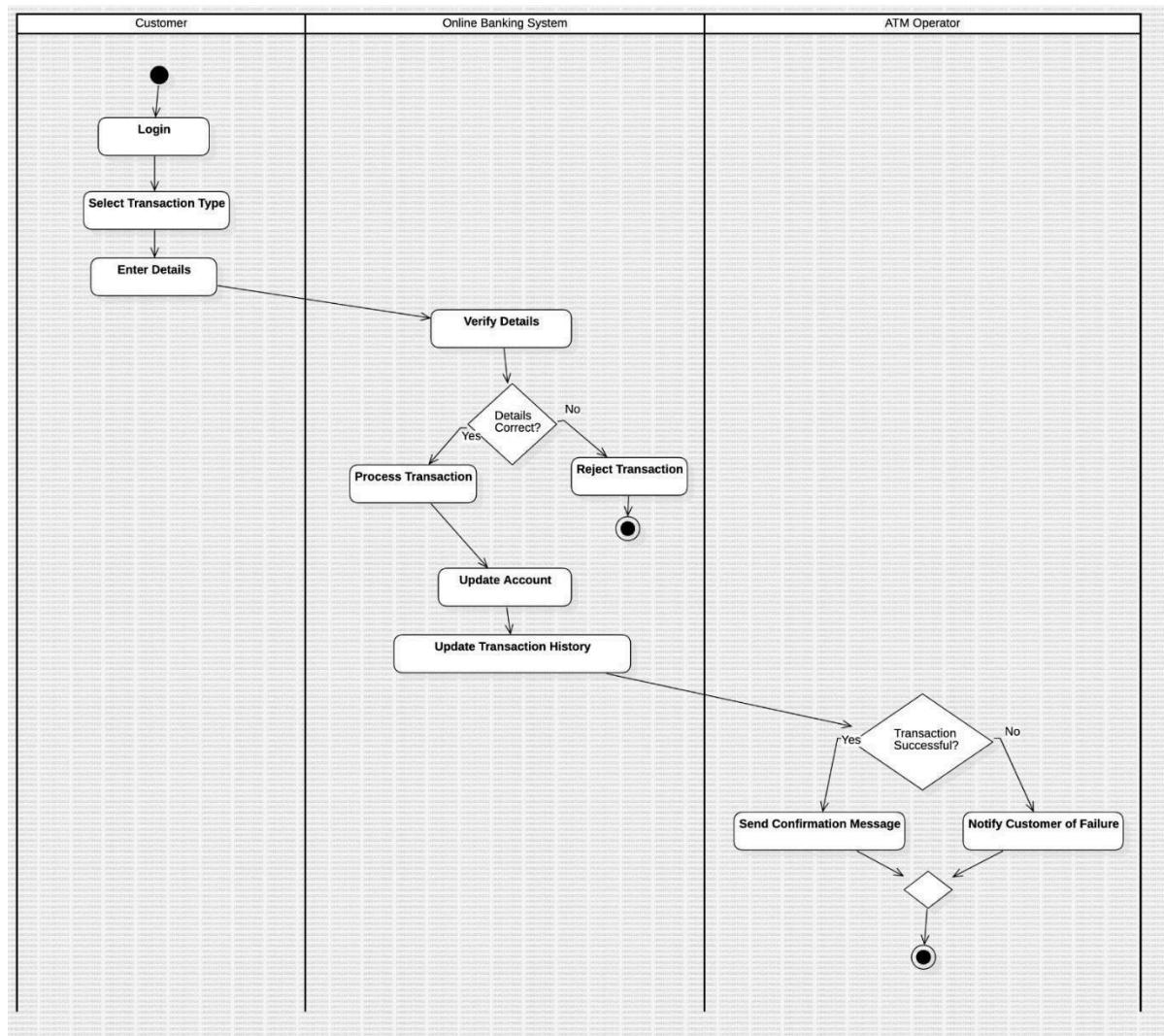
Online Transactions

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

State Diagram:



Activity Diagram:

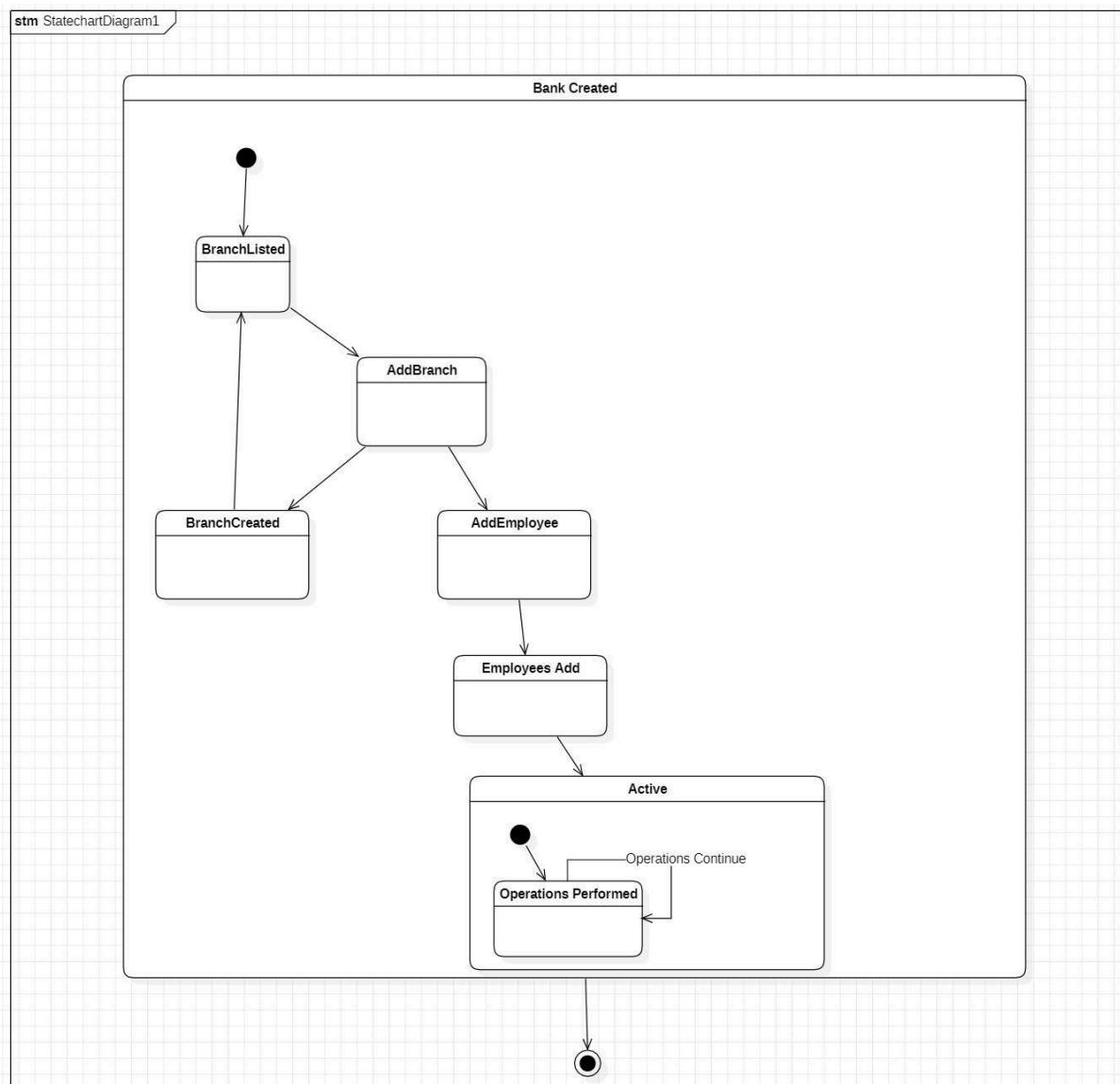


Functionality 3 (Done by Anshul Baliga)

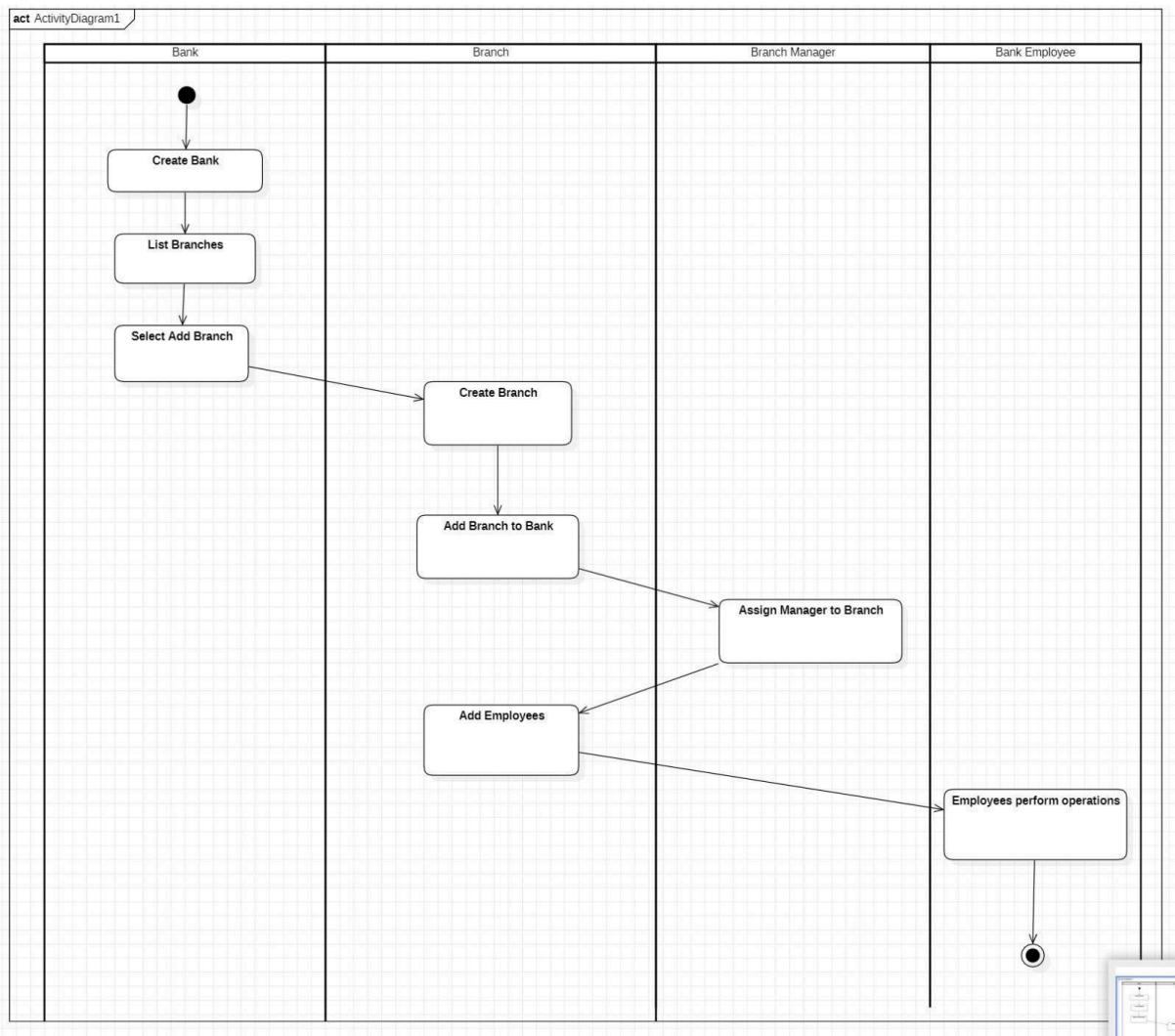
Branch Operations

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

State Diagram:



Activity Diagram:

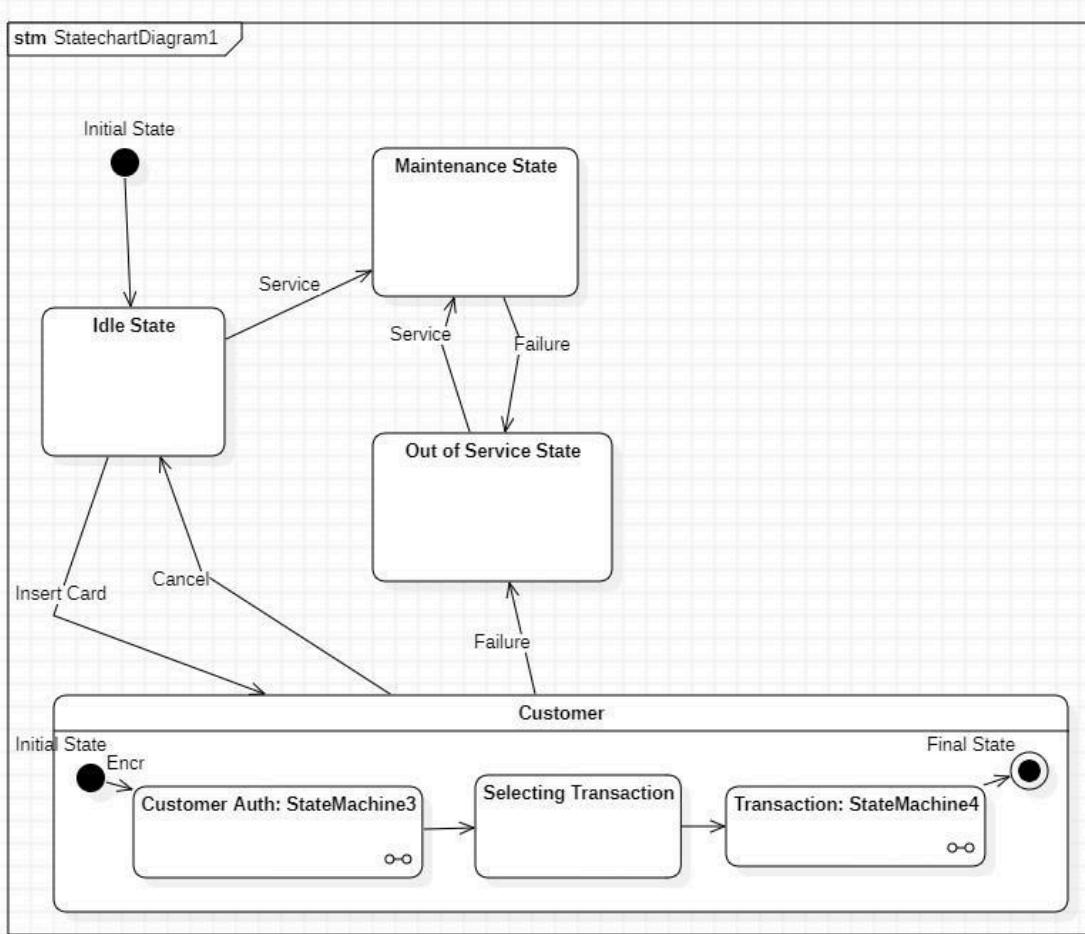


Functionality 4 (Done by Anirudh Lakhota)

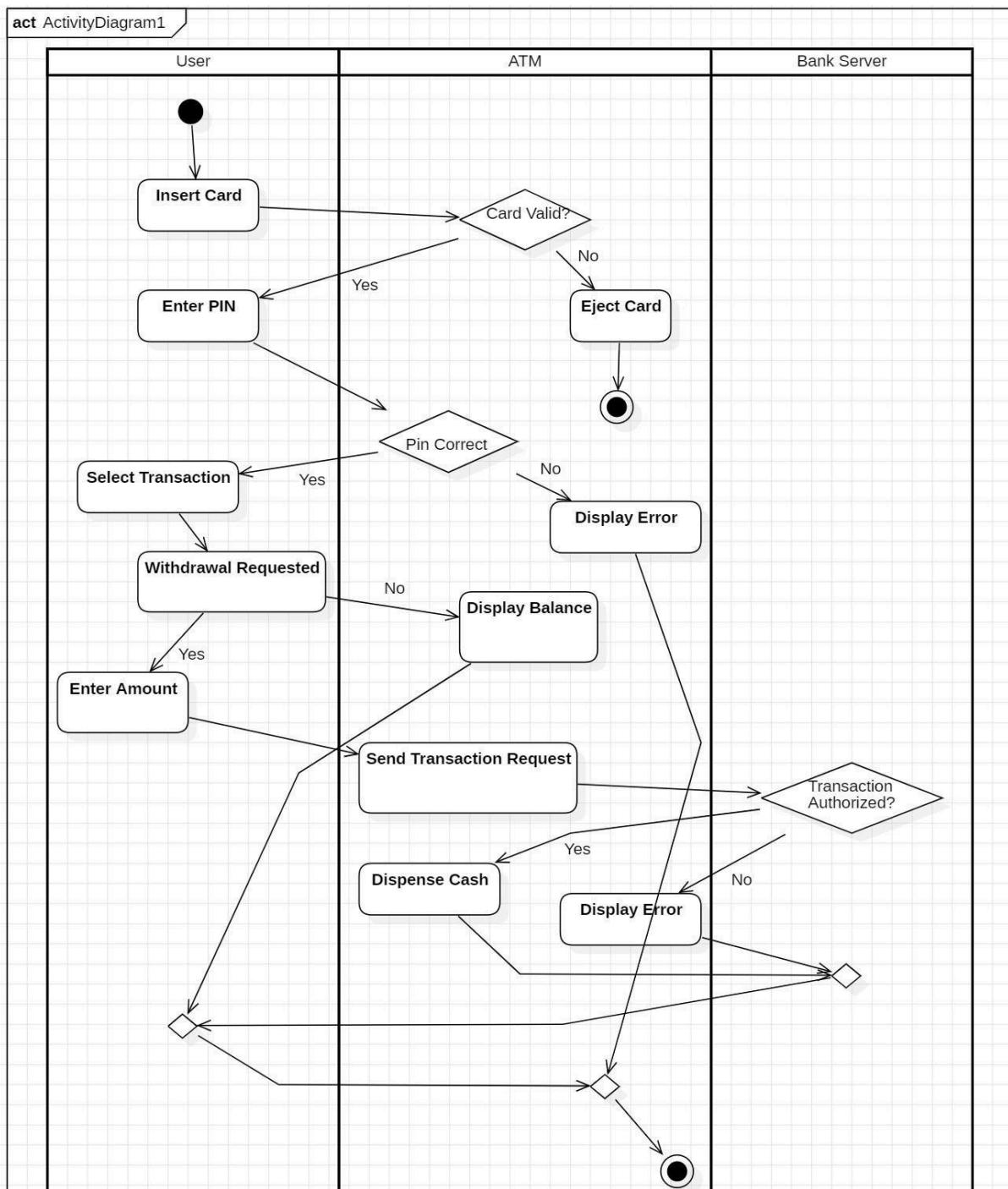
ATM Operations

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:

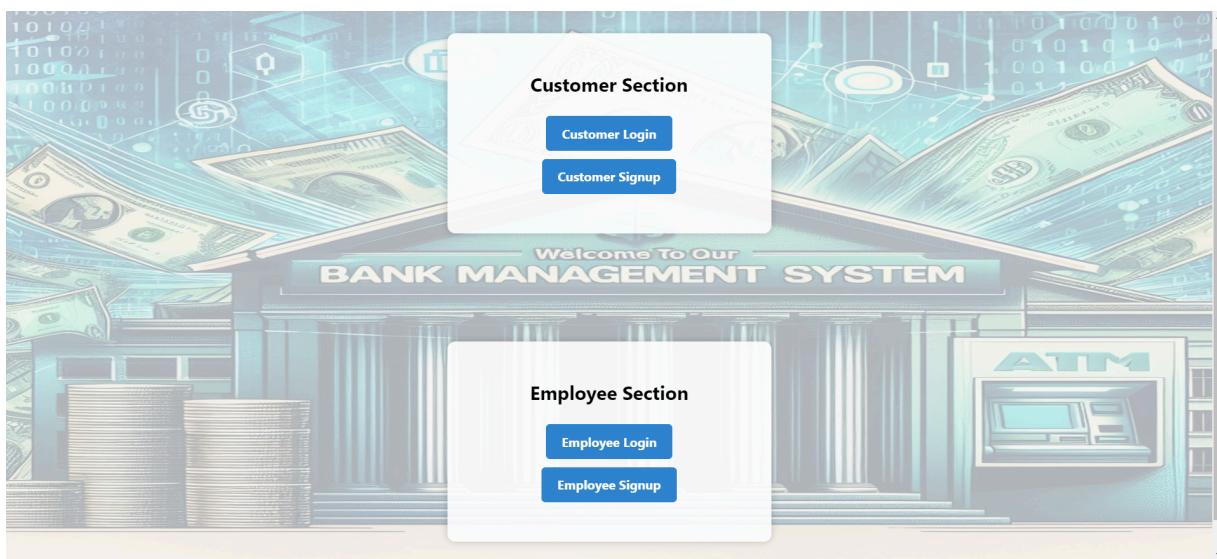
- In our system, we notify `BankEmployee` instances of changes in account statuses or transactions, sent to each 'iAccount'.

Github link to the Codebase:

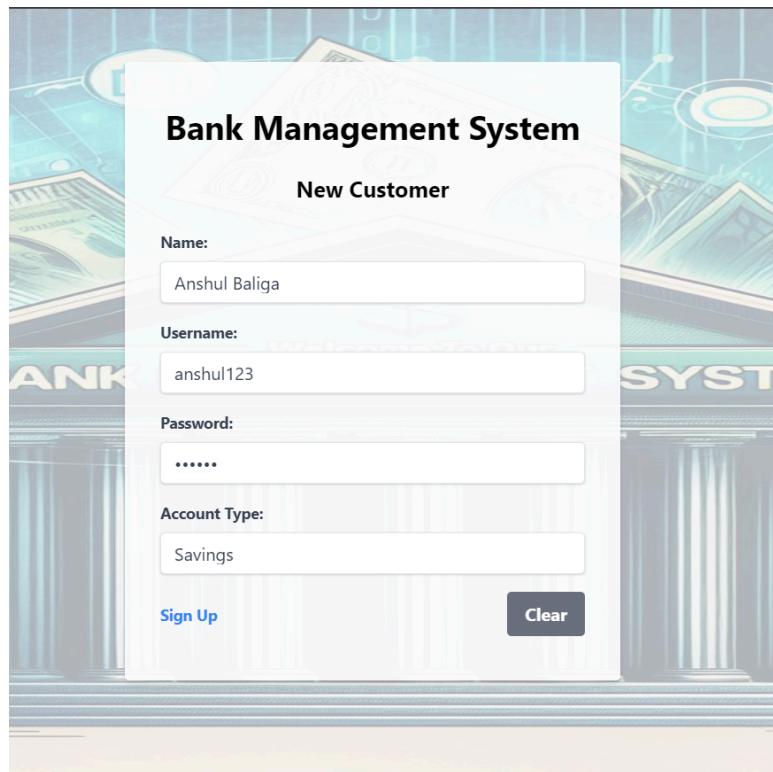
<https://github.com/anooshd7/Bank-Management-System-Java-Spring>

Screenshots with input values populated and output shown:

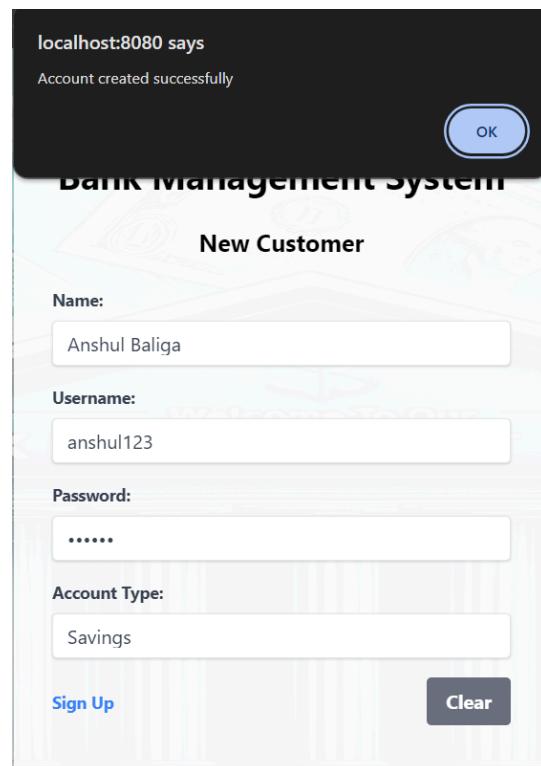
Home Page:



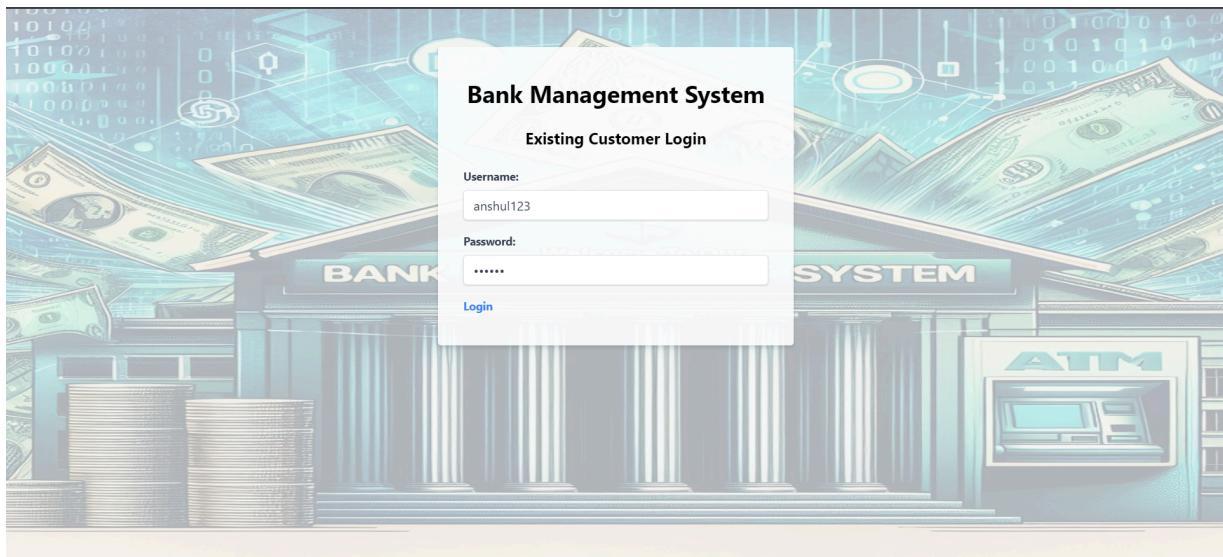
Customer Signup:



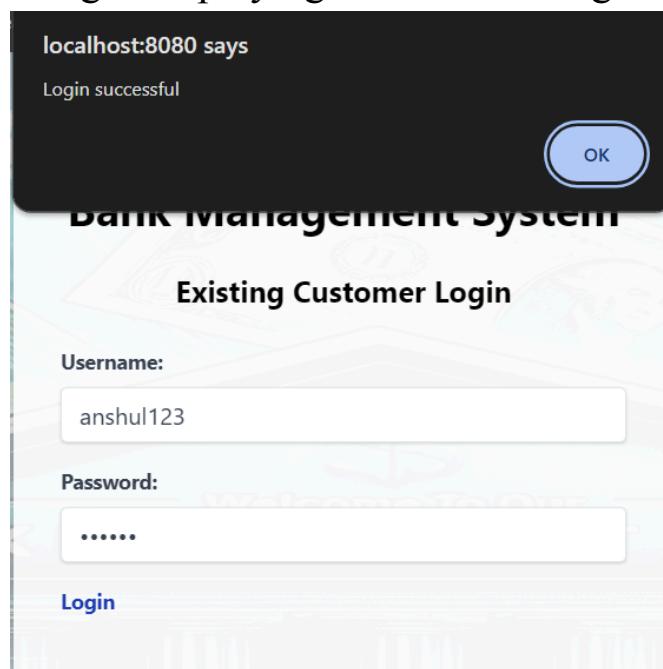
Confirmation message displaying ‘Successful Account Creation’:



Customer Login:



Confirmation messages displaying 'Successful Login':



Depositing money into an account:

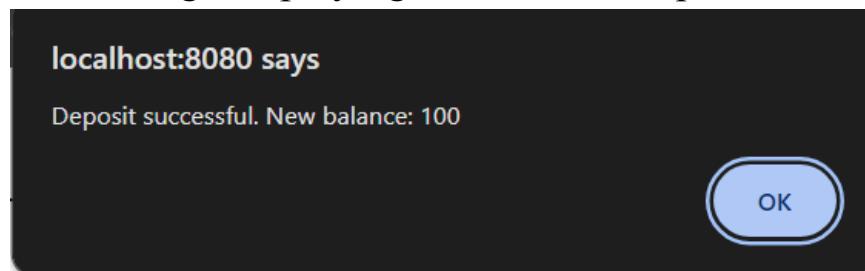
Account Management

Deposit

Amount:

Deposit

Confirmation message displaying ‘Successful Deposit’:



Deposit

Amount:

Deposit

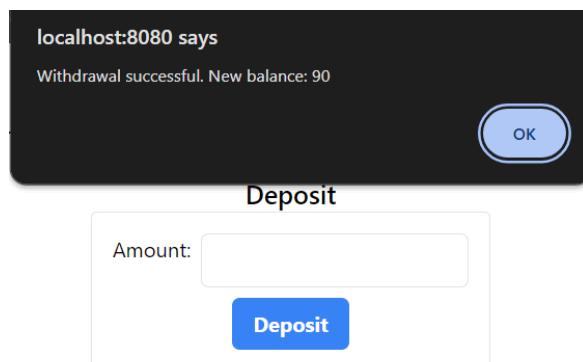
Withdrawing money:

Withdraw

Amount:

Withdraw

Confirmation message displaying ‘Successful Withdrawal’:

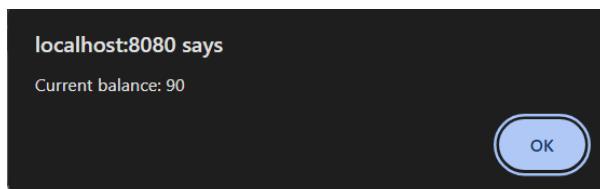


Withdraw

Amount:

Withdraw

Checking account balance:



Withdraw

Amount:

Withdraw

Check Balance

Check Balance

Display account details:

Get Account Details

Get Account Details

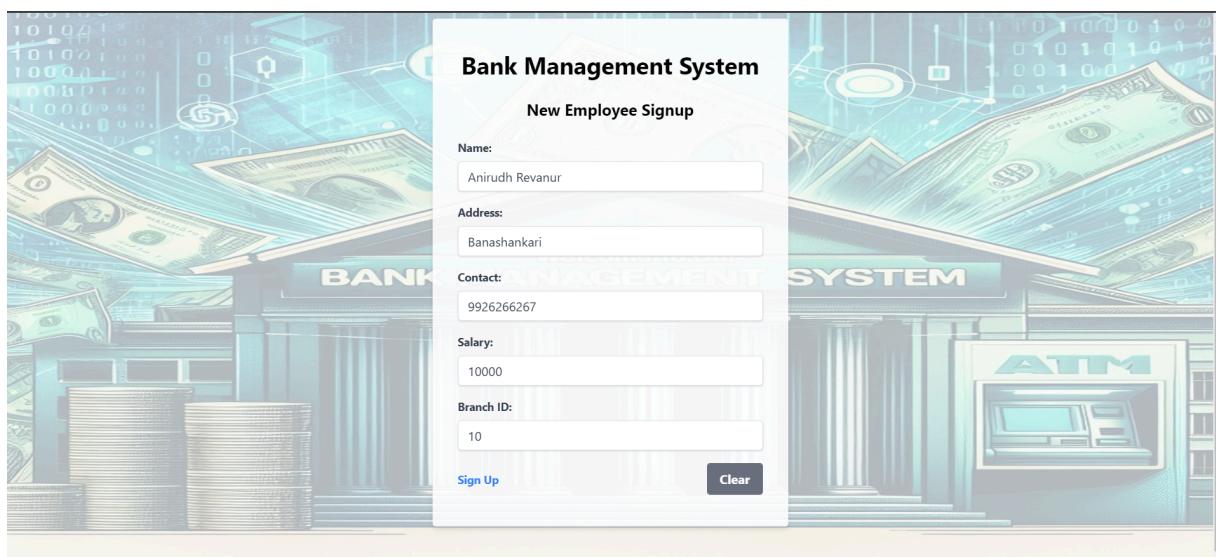
Account ID: 36

Name: Anshul Baliga

Balance: 90

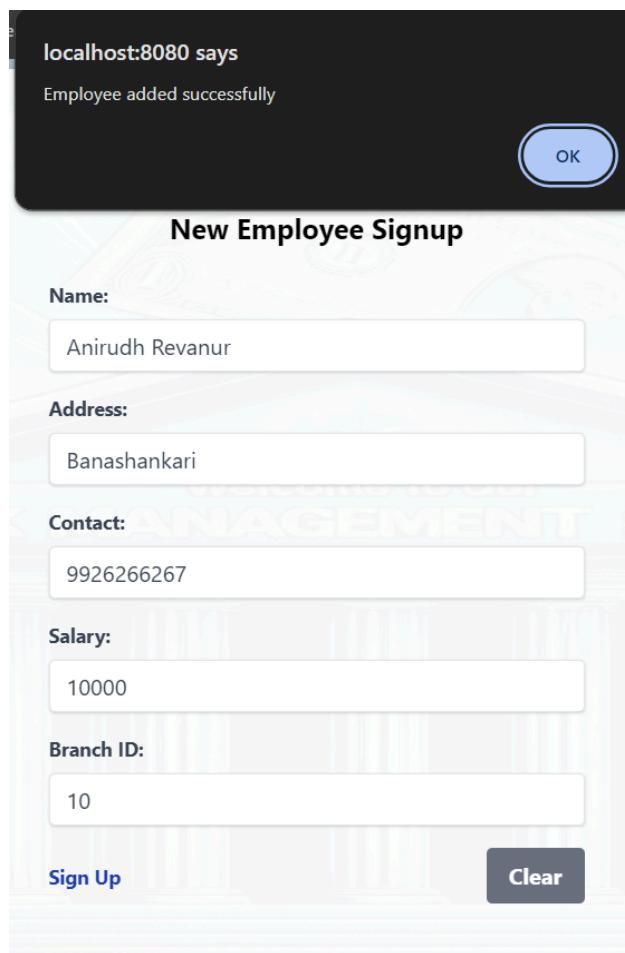
Interest Rate: 0.05

Employee Signup:



The image shows a screenshot of a web-based application for a 'Bank Management System'. The background features a stylized illustration of a bank building, stacks of coins, and US dollar bills, all set against a digital grid with binary code. In the center, a modal window titled 'Bank Management System' is displayed under the heading 'New Employee Signup'. The form contains six input fields: 'Name' (Anirudh Revanur), 'Address' (Banashankari), 'Contact' (9926266267), 'Salary' (10000), and 'Branch ID' (10). At the bottom of the form are two buttons: 'Sign Up' (in blue) and 'Clear' (in dark grey).

Confirmation message displaying ‘Successful addition of Employee’:



Display bank details:

The screenshot shows a form titled "Get Bank Details" with a "Bank ID" input field containing the value "2" and a "Get Bank Details" button. To the right of the form, under the heading "Bank Details", are the following details:

ID: 2
Bank Name: SBI
Location: Bengaluru
Number of Branches: 2

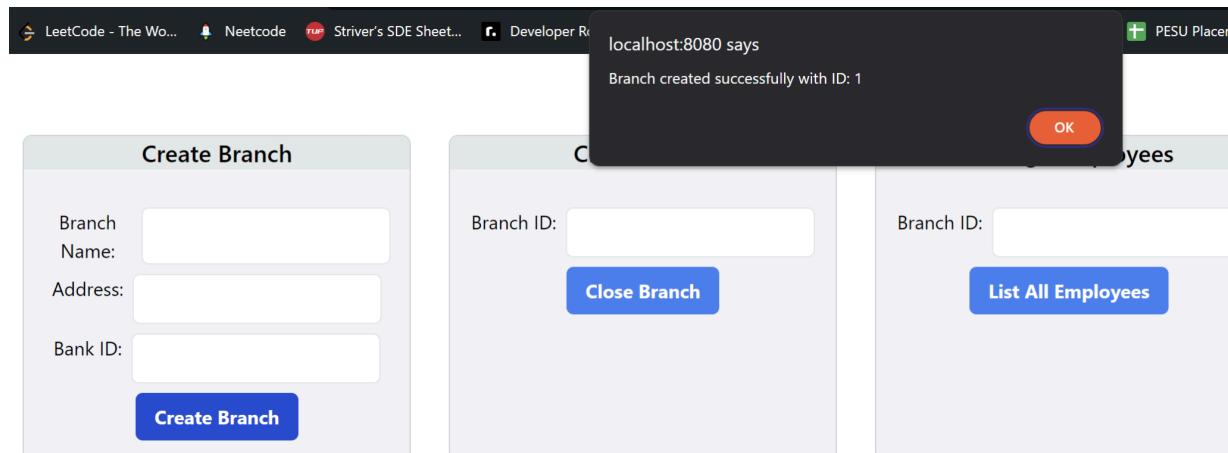
Creating a branch:

Create Branch

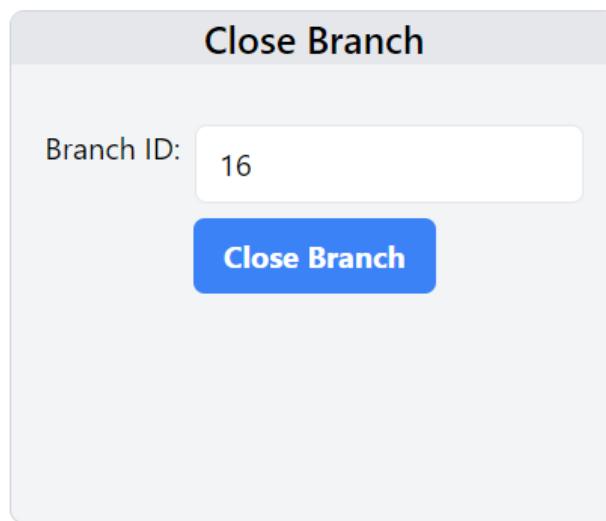
| | |
|--------------|--|
| Branch Name: | <input type="text" value="New Branch"/> |
| Address: | <input type="text" value="Indiranagar"/> |
| Bank ID: | <input type="text" value="2"/> |

Create Branch

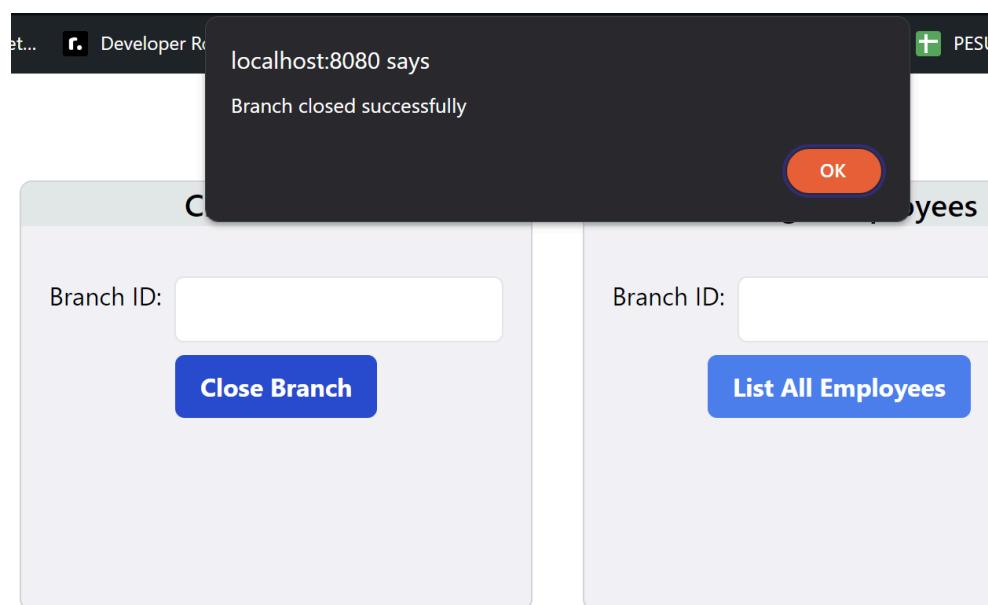
Confirmation message displaying ‘Successful creation of Branch’:



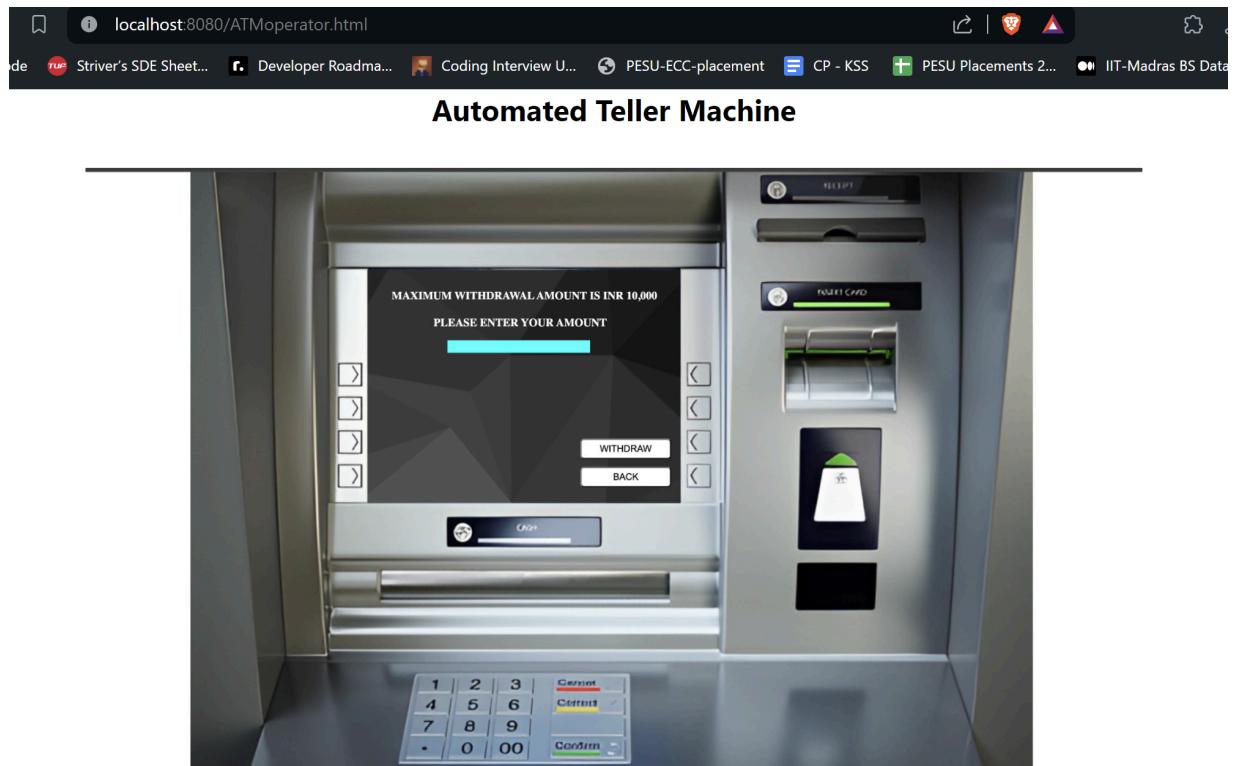
Closing a branch:



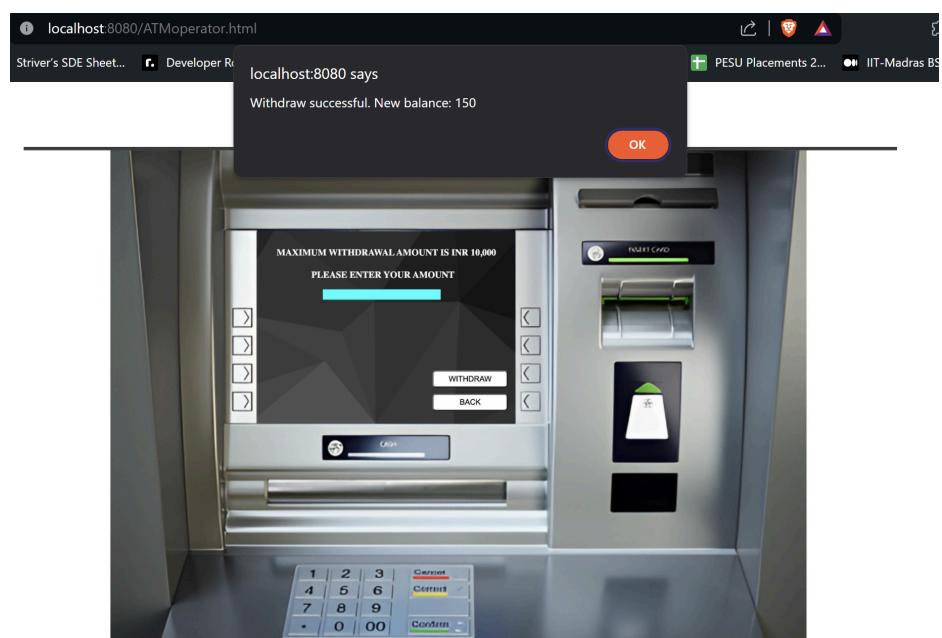
Confirmation message displaying 'Successful closing of Branch':



Depositing in the ATM:



Confirmation message displaying ‘Successful Withdrawal’:



Database management:

Stores all the accounts entered:

The screenshot shows the Oracle SQL Developer interface. The left sidebar displays the 'Schemas' tree, with 'banking_app_oad' expanded to show tables like account, atmoperator, bank, etc. The central pane shows a query editor with the following SQL statement:

```
1 • | SELECT * FROM banking_app_oad.account;
```

The results grid displays the following data:

| | id | account_holder_name | balance |
|---|------|---------------------|---------|
| ▶ | 1 | Anshul Baliga | 1000 |
| ▶ | 2 | Anoosh Damodar | 10000 |
| ▶ | 3 | Anirudh Revanur | 100000 |
| ▶ | 4 | Anirudh Lakhota | 1000000 |
| * | NULL | NULL | NULL |

Store Bank Details:

The screenshot shows the Oracle SQL Developer interface. The left sidebar displays the 'Schemas' tree, with 'banking_app_oad' expanded to show tables like bank, bank_employee, branch, etc. The central pane shows a query editor with the following SQL statement:

```
1 • | SELECT * FROM banking_app_oad.bank;
```

The results grid displays the following data:

| | id | bank_name | location |
|---|------|-----------|----------|
| ▶ | 1 | HDFC Bank | India |
| ▶ | 2 | HDFS | Japan |
| * | NULL | NULL | NULL |