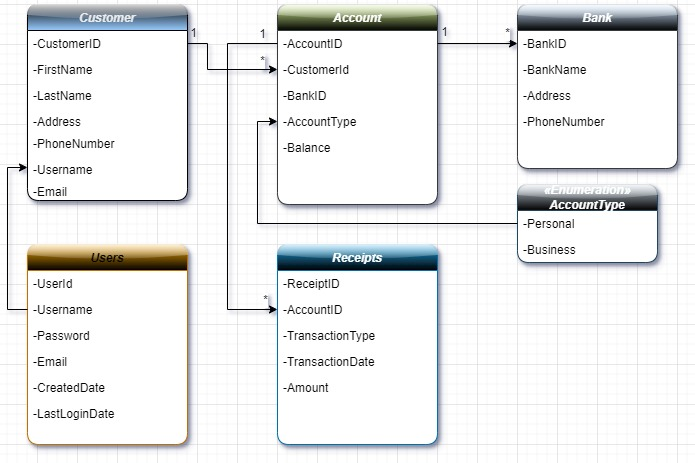
**Software Prototype**

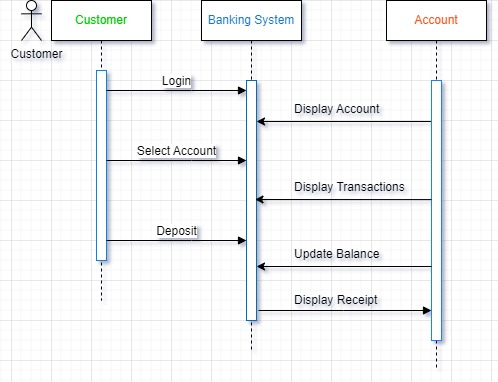
**Structural Diagrams: -Class Diagram**



The **Customer** entity is connected to the **Account** entity in a one, to many relationship, where a **customer** may have **accounts** while each account is linked to one **customer**. The **Account** entity is associated with the **Bank** entity in a one, to many relationship, where an account can be tied to one bank while a bank can have accounts.

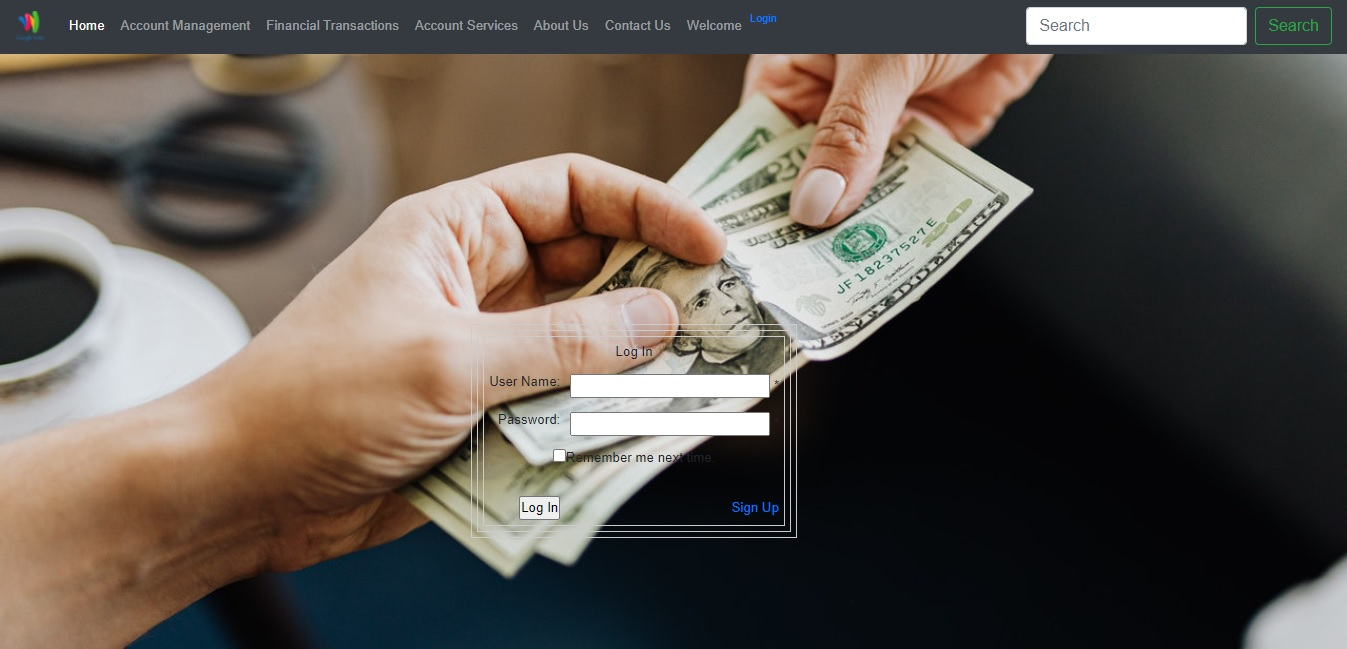
In the setup involving **Account** and **Receipts** classes if an account can possess receipts you'd employ a one, to many relationships. Essentially this implies that a single account can be linked to **receipts** whereas every receipt is tied to one account.

**Behavioral Diagrams**: **-Sequence Diagram**



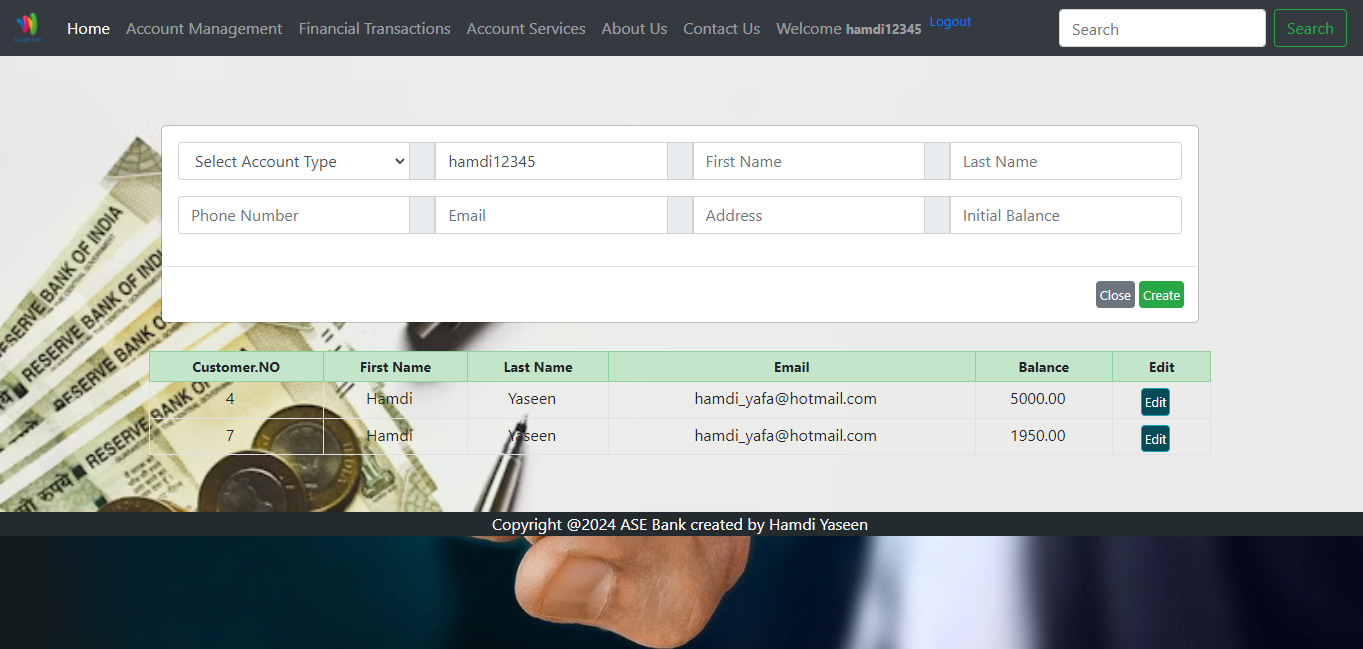
The process starts when the customer logs into the banking system. Subsequently, the banking system communicates with the account to exhibit account details and transaction history. Following that the customer chooses an account. Proceeds, with a deposit transaction. The banking system then adjusts the account balance. Issues a receipt. This diagram visually illustrates how elements in the banking system interact during actions, like logging in selecting an account making a deposit and receiving a receipt.

**Main Page:**

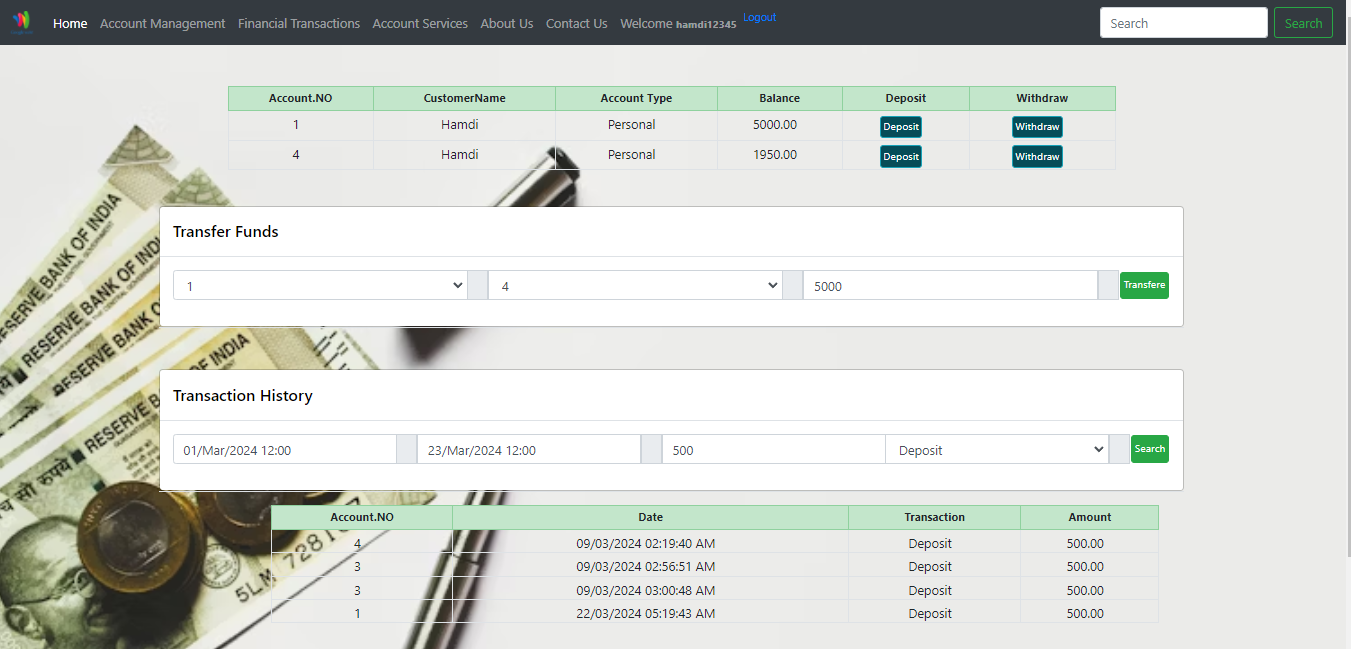


**Function Button Pages:**

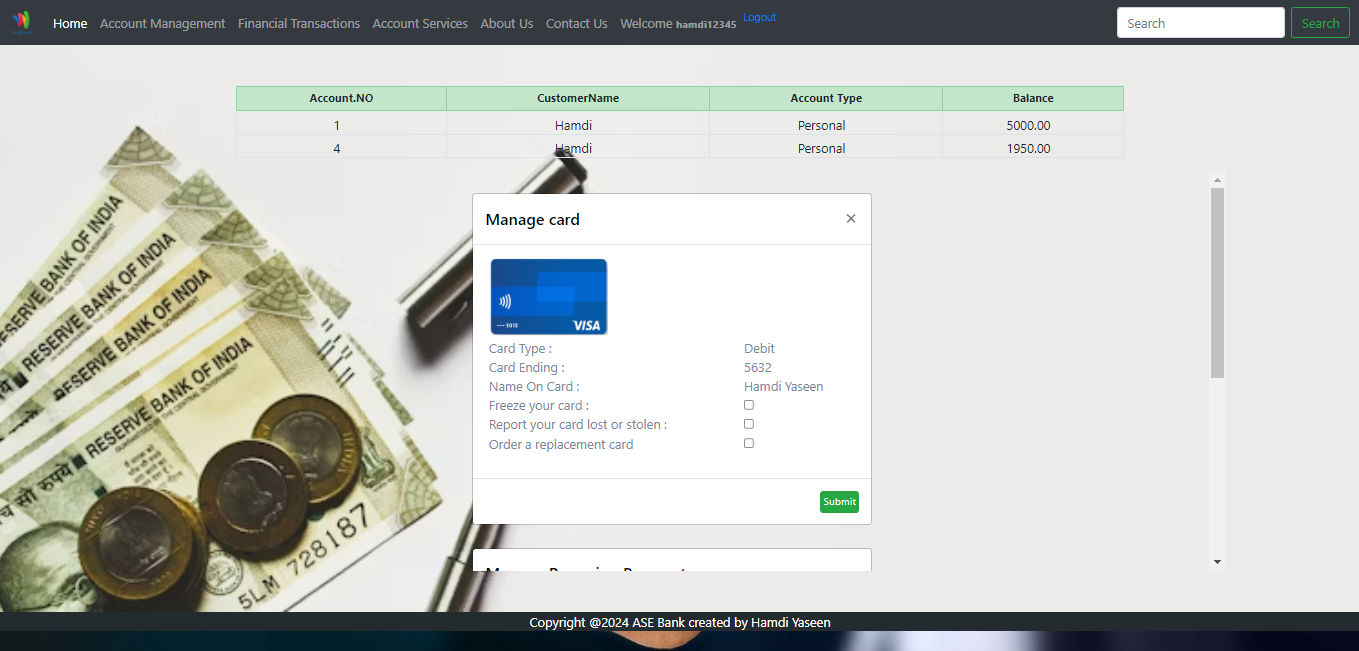
1. Account Management

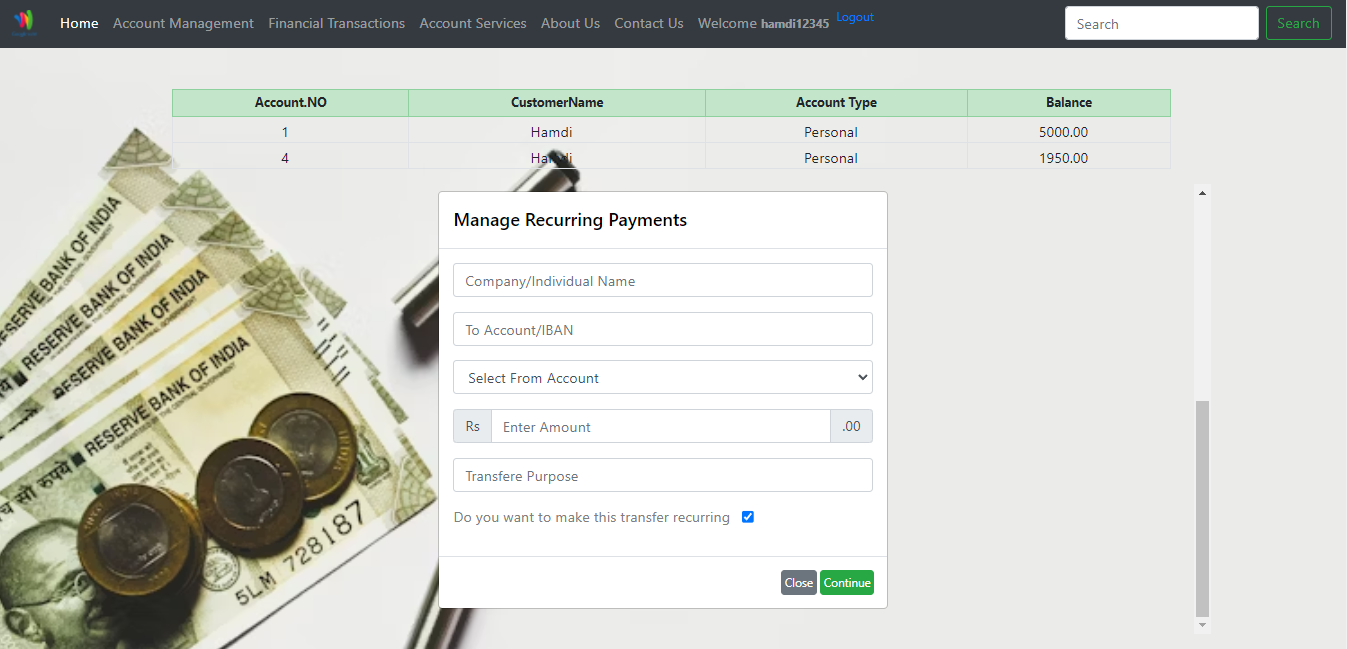


1. Transactions



1. Account Services





**Implementation and testing**: **Automated testing**

**Step 1: Establishing the Development Environment:**

It is imperative to possess the essential resources required for this task, such as:Visual Studio (or any other Integrated Development Environment compatible with C# and NET) NUnit Framework, NUnit Test Adapter (applicable when utilizing Visual Studio) Microsoft SQL Server

**Step 2: Establishing the Account Class:**

Initially, it is imperative to formulate a basic Account class encompassing a function designated for the purpose of depositing funds.

public class Account

{

public Account(decimal initialBalance)

{

Balance = initialBalance;

}

public void Deposit(decimal amount)

{

if (amount <= 0)

{

throw new ArgumentException("Deposit amount must be positive.");

}

Balance += amount;

}

public void Withdraw(decimal amount)

{

if (amount <= 0)

{

throw new ArgumentException("Withdrawal amount must be positive.");

}

if (Balance < amount)

{

throw new InvalidOperationException("Insufficient funds.");

}

Balance -= amount;

}

}

**Step 3: Write a Unit Test for the Deposit Method:**

Subsequently, an evaluation will be conducted to ascertain the accuracy of the account balance update carried out by the Deposit method.

using NUnit.Framework;

using System;

namespace BankingSystem.Tests

{

public class AccountTests

{

public void Deposit\_PositiveAmount()

{

// Arrange

var account = new Account(100); // Initial balance is 100

// Act

account.Deposit(50); // Try to deposit 50

// Assert

Assert.AreEqual(150, account.Balance, "Balance should be updated to 150 after deposit.");

}

public void Deposit\_NegativeAmount()

{

// Arrange

var account = new Account(100);

// Act & Assert

var ex = Assert.Throws<ArgumentException>(() => account.Deposit(-50));

Assert.That(ex.Message, Is.EqualTo("Deposit amount must be positive."));

}

}

}

**SQL Stored procedure**

ALTER PROCEDURE [dbo].**[DepositCashWithReceipt]**

@AccountID INT,

@Amount DECIMAL(18, 2)

AS

BEGIN

SET NOCOUNT ON;

declare @ErrorMessage VARCHAR(255);

SET @ErrorMessage = NULL;

-- Check if the account exists

IF NOT EXISTS (SELECT 1 FROM Account WHERE AccountID = @AccountID)

BEGIN

SET @ErrorMessage = 'Account does not exist.';

RETURN;

END

-- Update the balance

UPDATE Account

SET Balance = Balance + @Amount

WHERE AccountID = @AccountID;

-- Generate receipt

DECLARE @ReceiptID INT;

INSERT INTO Receipts (AccountID, TransactionType, Amount, TransactionDate)

VALUES (@AccountID, 'Deposit', @Amount, GETDATE());

SET @ReceiptID = SCOPE\_IDENTITY();

-- Output receipt information

SELECT 'Receipt generated successfully. Receipt ID: ' + CAST(@ReceiptID AS VARCHAR(10))

AS ReceiptInfo;

END;

**Step 4: Run the Test**

By employing NUnit Test Adapter within Visual Studio or utilizing a command-line utility such as dotnet test, one has the capability to execute the test cases. The successful execution of these tests is indicative of the accurate implementation of the Account class and deposit functionality.

**Explanation of the Test:**

Arrange: We initialize the object under test, in this case, an Account object.

Act: Perform the action that we want to test, which is the Deposit method.

Assert: Check if the outcome is as expected.

This methodology of testing aids in verifying the accuracy of fundamental features independently, guaranteeing that each element functions as expected prior to their integration into more extensive systems. These individual tests play a pivotal role in upholding the quality of code and serve as a fundamental element of any automated testing approach.

Validate (Assert function): Confirm that the account balance has been accurately modified by cross-referencing it against the anticipated balance. I used it as testing just in the code.

A stored procedure operates effectively, upholding the credibility of monetary transactions within the banking infrastructure. The utilization of automated tests for stored procedures aids in the early detection of errors and guarantees the dependability of database functions.

ALTER PROCEDURE [dbo].**[WithdrawCashWithRealTimeUpdate]**

@AccountID INT,

@Amount DECIMAL(18, 2),

@ErrorMessage VARCHAR(255)OUTPUT

AS

BEGIN

SET NOCOUNT ON;

SET @ErrorMessage = NULL;

-- Check if the account exists

IF NOT EXISTS (SELECT 1 FROM Account WHERE AccountID = @AccountID)

BEGIN

SET @ErrorMessage ='Account does not exist.';

RETURN;

END;

-- Check if sufficient balance is available

DECLARE @CurrentBalance DECIMAL(18, 2);

SELECT @CurrentBalance = Balance FROM Account WHERE AccountID = @AccountID;

IF @CurrentBalance < @Amount

BEGIN

SET @ErrorMessage ='Insufficient balance.';

RETURN;

END;

-- Update the balance

UPDATE Account

SET Balance = Balance - @Amount

WHERE AccountID = @AccountID;

-- Generate receipt

DECLARE @ReceiptID INT;

INSERT INTO Receipts (AccountID, TransactionType, Amount, TransactionDate)

VALUES (@AccountID, 'Withdraw', @Amount, GETDATE());

SET @ReceiptID = SCOPE\_IDENTITY();

-- Output success message

PRINT 'Withdrawal successful. Updated balance: ' + CAST((@CurrentBalance - @Amount) AS VARCHAR(20));

END;