Banking Transaction Management System (BTMS)

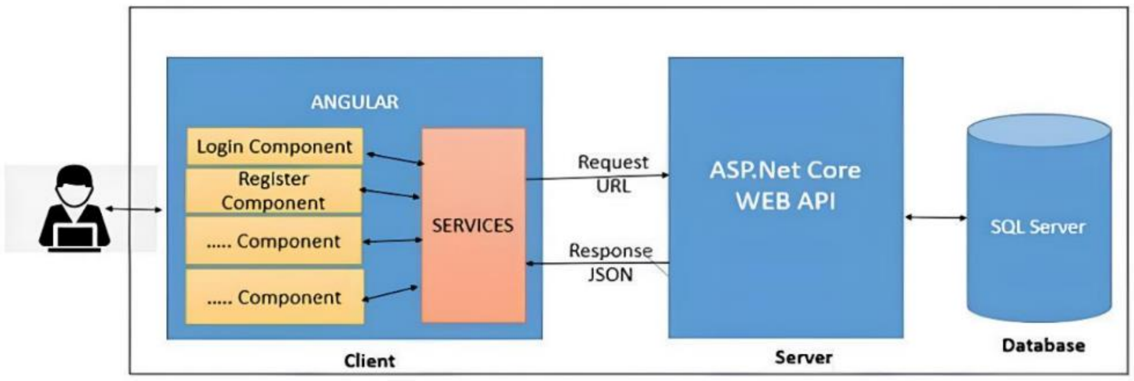
1. Overview

The Banking Transaction Management System (BTMS) is designed to manage various banking transactions including account creation and routine transactions (deposits and withdrawals) for customers. The system supports two user roles: Customer, and Manager, each with specific permissions and functionalities.

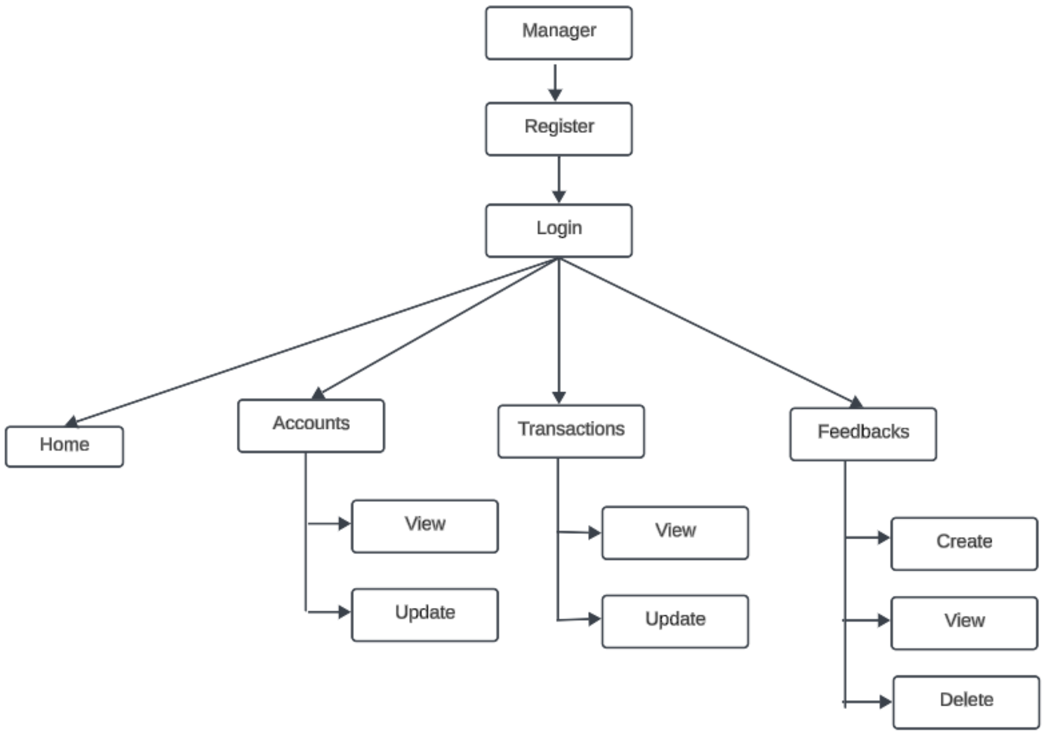
Users of the System:

1. **Customer**
2. **Manager**

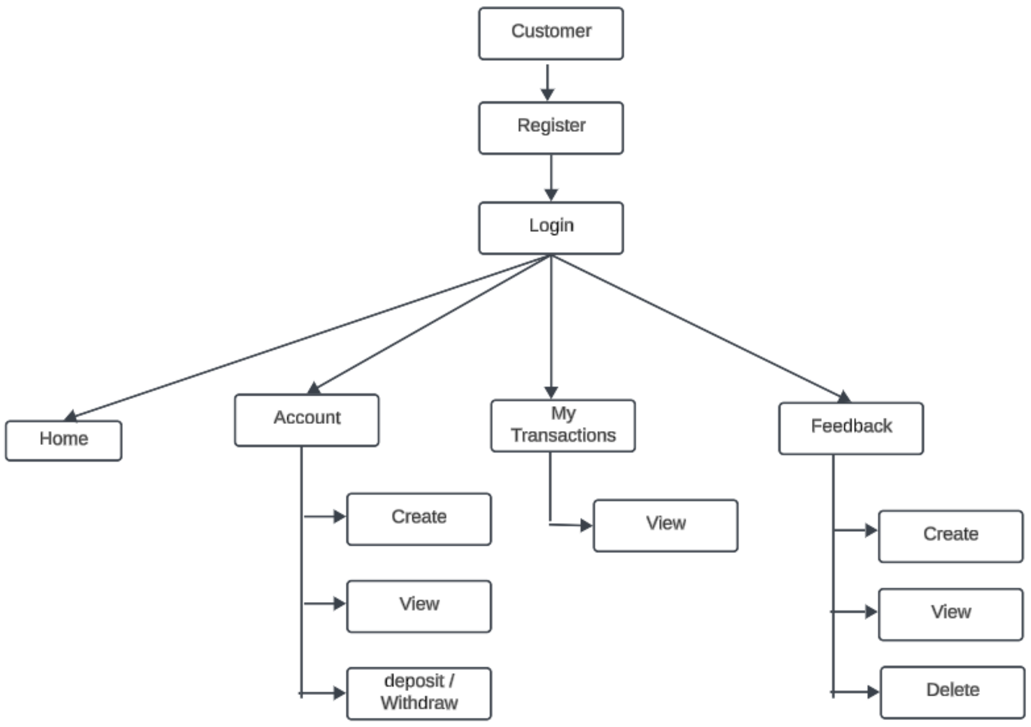
<u>System Architectural Diagram:</u>



<u>Manager Flow Diagram:</u>



Customer Flow Diagram:



Modules of the Application:

Manager:

- Register
- Login
- Dashboard
    - View all Account details and Approve (Edit).
    - View all Transaction details (Edit).
    - View feedback.

Customer:

- Register
- Login
- Dashboard
    - Create Account.
    - View My Accounts and make Deposits & withdrawals.
    - Add and view feedback (delete).

Technology Stack

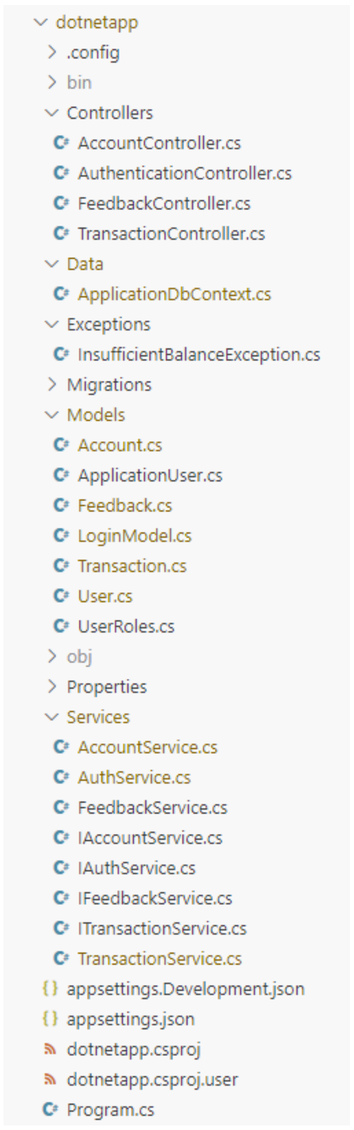
Front End	Angular 10+, HTML, CSS
Back End	.Net WebAPI, EF Core, Ms SQLServer Database

Application assumptions:

1. The login page should be the first page rendered when the application loads.
2. Manual routing should be restricted by implementing Auth Guard, utilizing the **canActivate** interface. For example, if the user enters as http://localhost:8080/dashboard or http://localhost:8080/user the page should not navigate to the corresponding page instead it should redirect to the login page.
3. Unless logged into the system, the user cannot navigate to any other pages.
4. Logging out must again redirect to the login page.

Backend Requirements:

Create folders for Models, Controllers, Services and configuration inside dotnetapp as mentioned in the below screenshot.



```
dotnetapp
  .config
  bin
  Controllers
    AccountController.cs
    AuthenticationController.cs
    FeedbackController.cs
    TransactionController.cs
  Data
    ApplicationDbContext.cs
  Exceptions
    InsufficientBalanceException.cs
  Migrations
  Models
    Account.cs
    ApplicationUser.cs
    Feedback.cs
    LoginModel.cs
    Transaction.cs
    User.cs
    UserRoles.cs
  obj
  Properties
  Services
    AccountService.cs
    AuthService.cs
    FeedbackService.cs
    IAccountService.cs
    IAuthService.cs
    IFeedbackService.cs
    ITransactionService.cs
    TransactionService.cs
  appsettings.Development.json
  appsettings.json
  dotnetapp.csproj
  dotnetapp.csproj.user
  Program.cs
```

ApplicationDbContext: (/Data/ApplicationDbContext.cs)

Namespace: dotnetapp.Data
Inside **Data** folder create **ApplicationDbContext** file with the following DbSet mentioned below

```
public DbSet<Account> Accounts { get; set; }
public DbSet<Transaction> Transactions { get; set; }
public DbSet<Feedback> Feedbacks { get; set; }
public  DbSet<User> Users { get; set; }
```

Model Classes:

Inside **Models** folder create all the model classes mentioned below.
Namespace: dotnetapp.Models

User (Models / User.cs)
This class stores user-related information in the system.
Properties:
- UserId (int): Unique identifier for each user.
- Email (string): Email address of the user. Required.
- Password (string): Password for the user's account. Required.
- Username (string): The username chosen by the user. Required.
- MobileNumber (string): The user's mobile number. Required.
- UserRole (string): The role of the user in the system, such as Manager or Customer. Required.

ApplicationUser (Models / ApplicationUser.cs)
This class represents a user within the application, inheriting from the IdentityUser class in ASP.NET Identity.
Properties:
- Name (string?): Optional field with a maximum length of 30 characters representing the user's name.

LoginModel (Models / LoginModel.cs)
This class stores credentials used during the login process for user authentication.
Properties:
- Email (string): The email address of the user. Required for authentication.
- Password (string): The password for the user's account. Required for authentication.

UserRoles (Models / UserRoles.cs)
This class defines constants for user roles in the application. It helps in differentiating between roles like Customer and Manager.
Constants:
1. User: Represents a standard customer. - **Customer**
2. User1: Represents a manager. - **Manager**

Account (Models / Account.cs)
This class represents a bank account in the system.
Properties:
- AccountId (int): Unique identifier for the account.
- UserId (int): The ID of the user who owns this account.

- AccountHolderName (string): Name of the account holder.
- AccountType (string): Type of account (e.g., Savings, Current).
- Balance (decimal): Current balance of the account.
- Status (string?): The status of the account (e.g., Active, InActive). This field is optional.
- ProofOfIdentity (string): A document or ID proof used for verification.
- DateCreated (DateTime?): Date when the account was created, optional.
- LastUpdated (DateTime?): Date when the account was last updated, optional.
- User (User?): A navigation property to associate the account with a specific user.

Transaction (Models / Transaction.cs)
This class represents a transaction within an account, such as a deposit, withdrawal, or transfer.
Properties:

- TransactionId (int): Unique identifier for the transaction.
- AccountId (int): The ID of the account involved in the transaction (e.g., sender's account in case of transfer).
- TransactionDate (DateTime): Date and time when the transaction occurred.
- TransactionType (string): Type of transaction (e.g., Deposit, Withdrawal).
- Amount (decimal): Amount involved in the transaction.
- Status (string?): Status of the transaction (e.g., Processing, Completed, Rejected). Optional.
- Description (string?): Optional description of the transaction.
- Account (Account?): A navigation property that associates the transaction with a specific account.

Feedback (Models / Feedback.cs)
This class stores user feedback submitted through the application.
Properties:

- FeedbackId (int): Unique identifier for each feedback.
- UserId (int): The ID of the user who provided the feedback.
- Comments (string): The feedback text provided by the user.
- DateProvided (DateTime): The date when the feedback was submitted.
- User (User?): A navigation property that associates the feedback with the user who submitted it.

Important note:
Implement database logic only in the service file functions **without using try-catch.** Use **try-catch** only in the controller files and call the service file functions inside it.

<u>Services</u>:
Inside "**Services"** folder create all the services file mentioned below.

Namespace: dotnetapp.Services

<u>AuthService</u> (Services / AuthService.cs):
The AuthService class is responsible for user authentication and authorization.

Functions:

1. **public async Task<(int, string)> Registeration(User model, string role):**
    a. Registers a new user with the provided details and assigns a role.

2. **public async Task<(int, string)> Login(LoginModel model):**
    a. Find user by email in the database.

    b. Check if user exists, if not return "**Invalid email".**
    c. If the user exists, check the password is correct, if not return "**Invalid password".**
    d. Logs in a user with the provided credentials and generates a JWT token for authentication.

3. **private string GenerateToken(IEnumerable<Claim> claims):**
    a. Generates a JWT token based on the provided claims.

<u>IAuthService</u> (Services / IAuthService.cs):

The **IAuthService** is an interface that defines methods for user registration and login.

Methods:

1. Task< (int, string)> Registration (User model, string role);
2. Task< (int, string)> Login (LoginModel model);

<u>AccountService</u> (Services / AccountService.cs)

The AccountService class is responsible for managing accounts, including creation, retrieval, updating, and deletion of account records.

<u>Constructor</u>:

```
public AccountService(ApplicationDbContext context)
    {
      _context = context;
    }
```

Functions:

1. **public Account CreateAccount(Account account):**
    - Creates a new account with the given details.
    - Sets the status to "InActive" upon creation.
    - Automatically assigns the current date and time to both DateCreated and LastUpdated fields.
    - Adds the new account to the database and saves changes.
    - Returns the created account.
2. **public Account GetAccountById(int accountId):**
    - Retrieves an account by its unique identifier (AccountId).
    - Searches for the account in the database, and returns it if found, otherwise returns null.
3. **public IEnumerable<Account> GetAllAccounts():**
    - Retrieves all the accounts stored in the database.
    - Returns the full list of accounts.
4. **public IEnumerable<Account> GetAccountsByUserId(int userId):**

   - Retrieves all accounts associated with a specific user (UserId).
   - Includes navigation to the User entity for related information.
   - Returns the list of accounts for the given user.
5. **public Account UpdateAccount(int accountId, Account updatedAccount):**
   - Finds an existing account by AccountId.
   - Updates the account’s balance, status, and LastUpdated timestamp with values from updatedAccount (if provided).
   - Saves the changes to the database and returns the updated account.
   - If the account does not exist, returns null.
6. **public bool DeleteAccount(int accountId):**
   - Deletes an account with the specified AccountId.
   - Searches for the account, and if found, removes it from the database and returns true.
   - Returns false if the account does not exist.

<u>IAccountService</u> (Services / IAccountService.cs)

The IAccountService interface defines the methods required for account management.

Methods:

1. **IEnumerable<Account> GetAllAccounts():**
   - Retrieves all accounts from the database.
2. **Account GetAccountById(int id):**
   - Retrieves an account by its unique identifier.
3. **Account CreateAccount(Account account):**
   - Creates a new account with the provided details.
4. **Account UpdateAccount(int id, Account account):**
   - Updates an account's details based on the provided AccountId and the updated account data.
5. **bool DeleteAccount(int id):**
   - Deletes an account based on its AccountId.
6. **IEnumerable<Account> GetAccountsByUserId(int userId):**
   - Retrieves all accounts associated with a specific user.

TransactionService (Services / TransactionService.cs)

The TransactionService class manages all operations related to account transactions, including creating, updating, and retrieving transactions, as well as processing the transaction logic (e.g., deposit and withdrawal).

Constructor:

```
public TransactionService(ApplicationDbContext context)
    {
        _context = context;
    }
```

Functions:

1. **public IEnumerable<Transaction> GetTransactions():**
   - Retrieves a list of all transactions from the database.
   - Includes related Account data for each transaction.
   - Returns a collection of Transaction objects.
2. **public IEnumerable<Transaction> GetTransactionsByUserId(int userId):**
   - Retrieves all transactions associated with a specific user.
   - Joins the Account table to find transactions where the UserId matches the provided user ID.
   - Returns a list of transactions for the given user.
3. **public Transaction CreateTransaction(Transaction transaction):**
   - Creates a new transaction for an account.
   - Validates the account by AccountId and ensures that it exists.
   - Checks for sufficient balance if the transaction is of type Withdrawal. If insufficient balance detected, throw insufficient balance custom Exception with message – “**Insufficient balance.”**
   - Only allows transaction types "Deposit" or "Withdrawal".
   - Assigns the initial status as "Processing" for amounts greater than 10,000, otherwise "Completed".
   - Saves the transaction to the database.
   - Automatically processes the transaction (i.e., updates balances) if no approval is required.
   - Returns the created transaction.
4. **public Transaction UpdateTransactionByManager(int transactionId, Transaction updatedTransaction):**
   - Finds a transaction by its TransactionId and includes the related Account.
   - Prevents updates on transactions already marked as "Completed."

    - Updates the transaction details (amount, type, description) based on the manager's input.
    - Allows the manager to approve or reject the transaction, updating its status accordingly.
    - If approved, processes the transaction logic (e.g., updating account balances).
    - Returns the updated transaction.

5. **public Transaction GetTransactionById(int transactionId):**
    - Retrieves a specific transaction by its TransactionId.
    - Includes the related Account data.
    - Returns the transaction if found, otherwise returns null.
6. **private void ProcessTransaction(Transaction transaction): - (Optional)**
    - Handles the actual deposit or withdrawal logic for the transaction.
    - Updates the account balance based on the transaction type:
        - Adds to the balance if the transaction is a deposit.
        - Deducts from the balance if it's a withdrawal (after checking sufficient balance). If insufficient balance detected, throw insufficient balance custom Exception with message – "**Insufficient balance.**"
    - Updates the transaction's description with details of the processed transaction.
    - Saves the updated account and transaction to the database.

ITransactionService (Services / ITransactionService.cs)

The ITransactionService interface defines the methods required for managing account transactions.

Methods:

1. **IEnumerable<Transaction> GetTransactions():**
    - Retrieves all transactions from the database.
2. **Transaction GetTransactionById(int transactionId):**
    - Retrieves a transaction by its unique identifier (TransactionId).
3. **Transaction CreateTransaction(Transaction transaction):**
    - Creates a new transaction for an account.
4. **Transaction UpdateTransactionByManager(int transactionId, Transaction updatedTransaction):**
    - Updates a transaction based on manager approval or rejection.
5. **IEnumerable<Transaction> GetTransactionsByUserId(int userId):**
    - Retrieves all transactions associated with a specific user.

FeedbackService (Services / FeedbackService.cs)

The FeedbackService class is responsible for managing user feedback, including retrieving, adding, and deleting feedback records from the database.

Constructor:

```
public FeedbackService(ApplicationDbContext context)
    {
      _context = context;
    }
```

Functions:

1. **public async Task<IEnumerable<Feedback>> GetAllFeedbacks():**
   - Retrieves all feedback records from the database.
   - Includes the related User data using Entity Framework's Include method.
   - Returns a list of Feedback objects asynchronously.
2. **public async Task<IEnumerable<Feedback>> GetFeedbacksByUserId(int userId):**
   - Retrieves all feedback submitted by a specific user.
   - Filters feedback records by the UserId property.
   - Returns a list of feedback entries for the given user ID asynchronously.
3. **public async Task<bool> AddFeedback(Feedback feedback):**
   - Adds a new feedback record to the database.
   - Saves the feedback asynchronously.
   - Returns true if the feedback is successfully added.
4. **public async Task<bool> DeleteFeedback(int feedbackId):**
   - Deletes an existing feedback record based on its FeedbackId.
   - Checks if the feedback exists; if not, it returns false.
   - If the feedback exists, it removes the feedback record from the database and saves the changes asynchronously.
   - Returns true if the feedback is successfully deleted.

IFeedbackService (Services / IFeedbackService.cs)

The IFeedbackService interface defines the methods required for managing feedback.

Methods:

1. **Task<IEnumerable<Feedback>> GetAllFeedbacks():**
   - Retrieves all feedback records.

2. **Task<IEnumerable<Feedback>> GetFeedbacksByUserId(int userId):**
   - Retrieves all feedback records associated with a specific user.
3. **Task<bool> AddFeedback(Feedback feedback):**
   - Adds a new feedback entry to the database.
4. **Task<bool> DeleteFeedback(int feedbackId):**
   - Deletes a feedback entry by its unique identifier (FeedbackId).

Controllers:
Inside "**Controllers**" folder create all the controllers file mentioned below.
Namespace: dotnetapp.Controllers

AuthenticationController (Controllers / AuthenticationController.cs):
This controller handles user authentication and registration requests.

Functions:

1. **public async Task<IActionResult> Login(LoginModel model)**
   a. Accepts login requests, validates the payload, and calls the authentication service to perform user login. Returns a JWT token upon successful login.

2. **public async Task<IActionResult> Register(User model):**
   a. Accepts registration requests, validates the payload, and calls the authentication service to register a new user. Returns a success message upon successful registration.

AccountController (Controllers / AccountController.cs)

The AccountController class manages accounts, handling various operations like creating, retrieving, updating, and deleting accounts. It interacts with the AccountService to perform these actions and secures access based on roles like "Manager" and "Customer."

Functions:

1. **[HttpGet] public IActionResult GetAllAccounts():**
   - Retrieves all accounts using the GetAllAccounts method from the AccountService.
   - This action is restricted to users with the "Manager" role.
   - Returns a 200 OK response with the list of accounts.
   - If no accounts are found, returns a 404 Not Found response.
2. **[HttpGet("user/{userId}")] public IActionResult GetAccountsByUserId(int userId):**
   - Retrieves all accounts for a specific user by their userId using the GetAccountsByUserId method from the AccountService.
   - This action is restricted to users with the "Customer" or "Manager" roles.
   - Returns a 200 OK response with the list of accounts for the given user.
   - If no accounts are found, returns a 404 Not Found response.
3. **[HttpPost] public IActionResult CreateAccount([FromBody] Account account):**

- Creates a new account with the provided account data.
- This action is restricted to users with the "Customer" or "Manager" roles.
- Ensures that the AccountType is either "Savings" or "Current."
- Checks if an account of the same type already exists for the user; if it does, returns a 400 Bad Request response.
- If the account is successfully created, returns a 201 Created response with the account details and the location of the created resource.
- If an unexpected error occurs, returns a 500 Internal Server Error response with the error message.

4. **[HttpGet("{id}")] public IActionResult GetAccountById(int id):**
    - Retrieves an account by its id using the GetAccountById method from the AccountService.
    - This action is restricted to users with the "Customer" or "Manager" roles.
    - Returns a 200 OK response with the account data if found.
    - If the account is not found, returns a 404 Not Found response.
5. **[HttpPut("{id}")] public IActionResult UpdateAccount(int id, [FromBody] Account account):**
    - Updates an existing account with the given id and updated account data.
    - This action is restricted to users with the "Manager" role.
    - Calls the UpdateAccount method from the AccountService to update the account.
    - If the account is successfully updated, returns a 200 OK response with the updated account data.
    - If the account is not found, returns a 404 Not Found response.
    - If there’s an InvalidOperationException, returns a 409 Conflict response with the error message.
    - If an unexpected error occurs, returns a 500 Internal Server Error response with the error message.
6. **[HttpDelete("{id}")] public IActionResult DeleteAccount(int id):**
    - Deletes an account with the specified id using the DeleteAccount method from the AccountService.
    - This action is restricted to users with the "Manager" role.
    - If the account is successfully deleted, returns a 204 No Content response.
    - If the account is not found, returns a 404 Not Found response.

TransactionController (Controllers / TransactionController.cs)

The TransactionController class manages transaction-related operations, providing endpoints for retrieving, creating, and updating transactions. It interacts with the TransactionService to handle these operations. Certain actions are restricted to specific roles such as "Manager" and "Customer."

Functions:

1. **[HttpGet] public IActionResult GetTransactions():**
   - Retrieves all transactions using the GetTransactions method from the TransactionService.
   - This action is restricted to users with the "Manager" role.
   - Returns a 200 OK response with the list of transactions.
2. **[HttpPost] public IActionResult CreateTransaction([FromBody] Transaction transaction):**
   - Allows customers to create a new transaction with the provided transaction data.
   - This action is restricted to users with the "Customer" role.
   - Calls the CreateTransaction method from the TransactionService to create the transaction.
   - If the transaction is successfully created, returns a 201 Created response with the transaction details and the location of the created resource.
   - If an unexpected error occurs, returns a 500 Internal Server Error response with the error message.
3. **[HttpPut("manager/{id}")] public IActionResult UpdateTransactionByManager(int id, [FromBody] Transaction updatedTransaction):**
   - Allows a manager to update an existing transaction with the given id and updated transaction data.
   - This action is restricted to users with the "Manager" role.
   - Calls the UpdateTransactionByManager method from the TransactionService to update the transaction.
   - If the update is successful, returns a 200 OK response with the updated transaction data.
   - If an invalid argument is provided, returns a 400 Bad Request response with the error message.
   - If an unexpected error occurs, returns a 500 Internal Server Error response with the error message.
4. **[HttpGet("customer/{userId}")] public IActionResult GetTransactionsByUserId(int userId):**
   - Retrieves all transactions for a specific user by their userId using the GetTransactionsByUserId method from the TransactionService.
   - This action is restricted to users with the "Customer" role.

    - Returns a 200 OK response with the list of transactions for the given user.

5. **[HttpGet("{id}")] public IActionResult GetTransactionById(int id):**
    - Retrieves a transaction by its id using the GetTransactionById method from the TransactionService.
    - Returns a 200 OK response with the transaction data if found.
    - If the transaction is not found, returns a 404 Not Found response.

FeedbackController (Controllers / FeedbackController.cs)

The FeedbackController class manages feedback-related operations and interacts with the FeedbackService to handle CRUD (Create, Read, Delete) actions for feedback. It provides role-based access to customers and managers for different operations.

Functions:

1. **[HttpGet] public async Task<ActionResult<IEnumerable<Feedback>>> GetAllFeedbacks():**
    - Retrieves all feedback entries using the GetAllFeedbacks method from the FeedbackService.
    - This action is restricted to users with the "Manager" role.
    - If feedbacks are retrieved successfully, returns a 200 OK response with the list of feedbacks.
    - If an error occurs during the process, returns a 500 Internal Server Error response with the exception message.
2. **[HttpGet("user/{userId}")] public async Task<ActionResult<IEnumerable<Feedback>>> GetFeedbacksByUserId(int userId):**
    - Retrieves feedbacks submitted by a specific user identified by userId using the GetFeedbacksByUserId method from the FeedbackService.
    - This action is restricted to users with the "Customer" role.
    - If feedbacks are retrieved successfully, returns a 200 OK response with the feedbacks for the given user.
    - If an error occurs, returns a 500 Internal Server Error response with the exception message.
3. **[HttpPost] public async Task<ActionResult> AddFeedback([FromBody] Feedback feedback):**
    - Allows customers to submit new feedback.
    - This action is restricted to users with the "Customer" role.
    - Calls the AddFeedback method from the FeedbackService to add the feedback.
    - If the feedback is added successfully, returns a 200 OK response with a success message "Feedback added successfully."

    - If an error occurs, returns a 500 Internal Server Error response with the exception message.
4. **[HttpDelete("{feedbackId}")] public async Task<ActionResult> DeleteFeedback(int feedbackId):**
    - Deletes a feedback entry by its feedbackId.
    - This action is restricted to users with the "Customer" role.
    - Calls the DeleteFeedback method from the FeedbackService to delete the feedback.
    - If the feedback is deleted successfully, returns a 200 OK response with a success message "Feedback deleted successfully."
    - If the feedback is not found, returns a 404 Not Found response with the message "Cannot find any feedback."
    - If an error occurs, returns a 500 Internal Server Error response with the exception message.

Exception:

Inside exception folder create the exception file named **InsufficientBalanceException (InsufficientBalanceException.cs).**

Purpose: The **InsufficientBalanceException** class provides a mechanism for handling exceptions related to transactions within the application.

Namespace: It should located within the **dotnetapp.Exceptions** namespace.

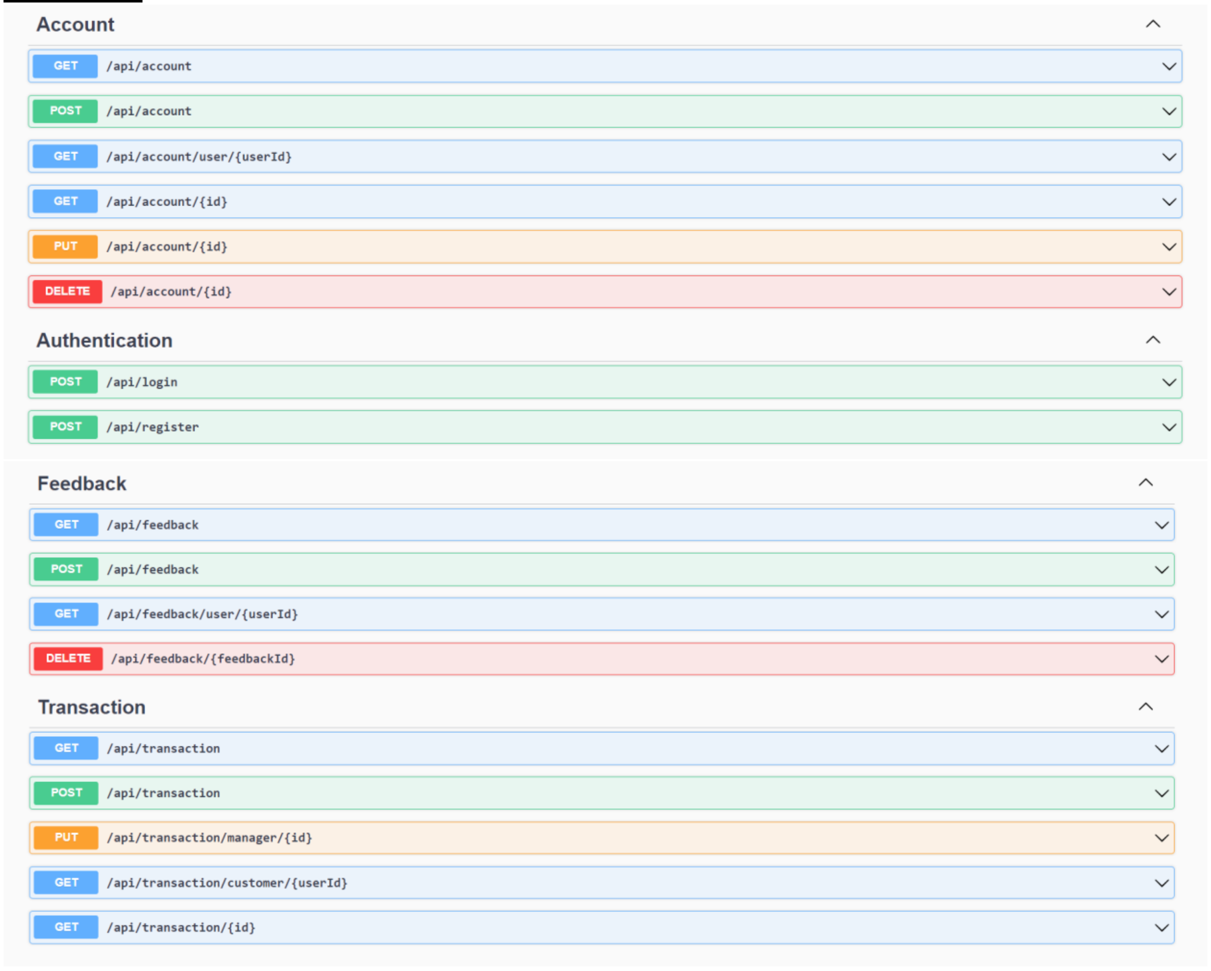
Inheritance: Inherits from the base **Exception** class, enabling it to leverage existing exception handling mechanisms.

Constructor: Contains a constructor that accepts a message parameter, allowing to specify custom error messages when throwing exceptions.

For example, you might throw a **InsufficientBalanceException**

1. When trying to **make withdrawal from insufficient balance account.**

Endpoints:



Account

GET /api/account

POST /api/account

GET /api/account/user/{userId}

GET /api/account/{id}

PUT /api/account/{id}

DELETE /api/account/{id}

Authentication

POST /api/login

POST /api/register

Feedback

GET /api/feedback

POST /api/feedback

GET /api/feedback/user/{userId}

DELETE /api/feedback/{feedbackId}

Transaction

GET /api/transaction

POST /api/transaction

PUT /api/transaction/manager/{id}

GET /api/transaction/customer/{userId}

GET /api/transaction/{id}

Note:

Ensure that JWT authentication is enabled for all endpoints, with authorization based on user roles. For example, the endpoint for Approving a account should only be accessible with a JWT present in the header. If not, return **Unauthorized**. Additionally, it should be restricted to customers with the "**Manager"** role. If a customer tries to access the Manager roles endpoints, it should return a **Forbidden** error.

1. Login: [Access for both Customer and Manager]

Endpoint name: "/api/login"

Method: POST

Request body: {

```
"Email": "string",
"Password": "string"
}
```

Response:

Status Code	**Response body**
200 (HttpStatusCode OK)	JSON object containing a JWT token. Example: { "Status": "Success",

	"token": "eyJhbGciOiJIUzI1NiiQWRtaW4iLCJqdGkiOiJmZTUyYzAxZS1" }
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

2. <u>Register</u>: [Access for both Customer and Manager]

Endpoint name: "/api/register"
Method: POST
Request body:

```
{
  "Username": "string",
  "Email": "user@example.com",
  "MobileNumber": "9876541221",
  "Password": "Pass@2425",
  "UserRole": "string"
}
```

Response:

Status Code	**Response body**
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

3. <u>Get all Accounts:</u> [Access for both Customer and Manager]

Endpoint name: "/api/account"

Method: GET

Response:

Status Code	**Response body**
200 (HttpStatusCode OK)	JSON object containing all the accounts.
500	JSON object containing Error message.

4. <u>Add Account:</u> [Access for both Customer and Manager]

Endpoint name: "/api/account"

Method: POST

Request body:

```
{
  "UserId": 0,
  "AccountHolderName": "string",
  "AccountType": "string",
  "Balance": 0,
  "Status": "string",
  "ProofOfIdentity": "string",
  "DateCreated": "2024-10-03T07:34:52.489Z",
  "LastUpdated": "2024-10-03T07:34:52.489Z"
}
```

Response:

Status Code	**Response body**
201 (HttpStatusCode Created)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

5. <u>Get Accounts by UserId:</u> [Access for only Customer]

Endpoint name: "/api/account/user/{userId}"

Method: GET

Parameter: userId

Response:

Status Code	**Response body**
200 (HttpStatusCode OK)	JSON object containing account details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

6. <u>Get specific Account by accountId:</u> [Access for both Customer and Manager]

Endpoint name: "/api/account/{id}"

Method: GET

Parameter: id

Response:

Status Code	**Response body**
200 (HttpStatusCode OK)	JSON object containing account details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

7. <u>Update Account:</u> [Access for only Manager]

Endpoint name: “/api/account/{id}”

Method: PUT

Parameter: id

Request body:

```
{
  "UserId": 0,
  "AccountHolderName": "string",
  "AccountType": "string",
  "Balance": 0,
  "Status": "string",
  "ProofOfIdentity": "string",
  "DateCreated": "2024-10-03T07:34:52.489Z",
  "LastUpdated": "2024-10-03T07:34:52.489Z"
}
```

Response:

Status Code	**Response body**
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

8. <u>Delete Account:</u> [Access for only Manager]

Endpoint name: “/api/account/{id}”

Method: DELETE

Parameter: id

Response:

Status Code	**Response body**
204 (HttpStatusCode NoContent)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

9. <u>Add Transaction:</u> [Access for only Customer]

Endpoint name: “/api/transaction”

Method: POST

Request body:

```
{
  "AccountId": 0,
  "TransactionDate": "2024-10-03T07:47:29.251Z",
  "TransactionType": "string",
  "Amount": 0,
  "Status": "string",
  "Description": "string"
}
```

Response:

Status Code	Response body
201 (HttpStatusCode Created)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

10. Get Transactions specific to Customer: [Access for only Customer]

Endpoint name: “/api/transaction/customer/{userId }”

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing transaction details.
500	JSON object containing Error message.

11. Get Transactions specific TransactionId [Access for both Customer & Manager]

Endpoint name: “/api/transaction/{id}”

Method: GET

Parameter: id

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing transaction details.
500	JSON object containing Error message.

12. <u>Get all Transactions:</u> [Access for only Manager]

Endpoint name: “/api/transaction”

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all Transactions.
500	JSON object containing Error message.

13. <u>Update Transaction:</u> [Access for only Manager]

Endpoint name: “/api/transaction/manager/{id}”

Method: PUT

Parameter: id

Request body:

```
{
  "AccountId": 0,
  "TransactionDate": "2024-10-03T07:53:56.444Z",
  "TransactionType": "string",
  "Amount": 0,
  "Status": "string",
  "Description": "string"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

14. <u>Get all feedbacks:</u> [Access for only Manager]

Endpoint name: “/api/feedback”

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all feedback.

500	JSON object containing Error message.

15. <u>Add feedback</u>: [Access for only Customer]

Endpoint name: “/api/feedback”

Method: POST

Request body:

```
{
  "UserId": 0,
  "Comments": "string",
  "DateProvided": "2024-07-07T12:28:56.927Z"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
500	JSON object containing Error message.

16. <u>Get feedback specific to Customer</u>: [Access for only Customer]

Endpoint name: “/api/feedback/user/{userId }”

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing feedback details specific to user.
500	JSON object containing Error message.

17. <u>Delete feedback</u>: [Access for only Customer]

Endpoint name: “/api/feedback /{feedbackId}”

Method: DELETE

Parameter: feedbackId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.

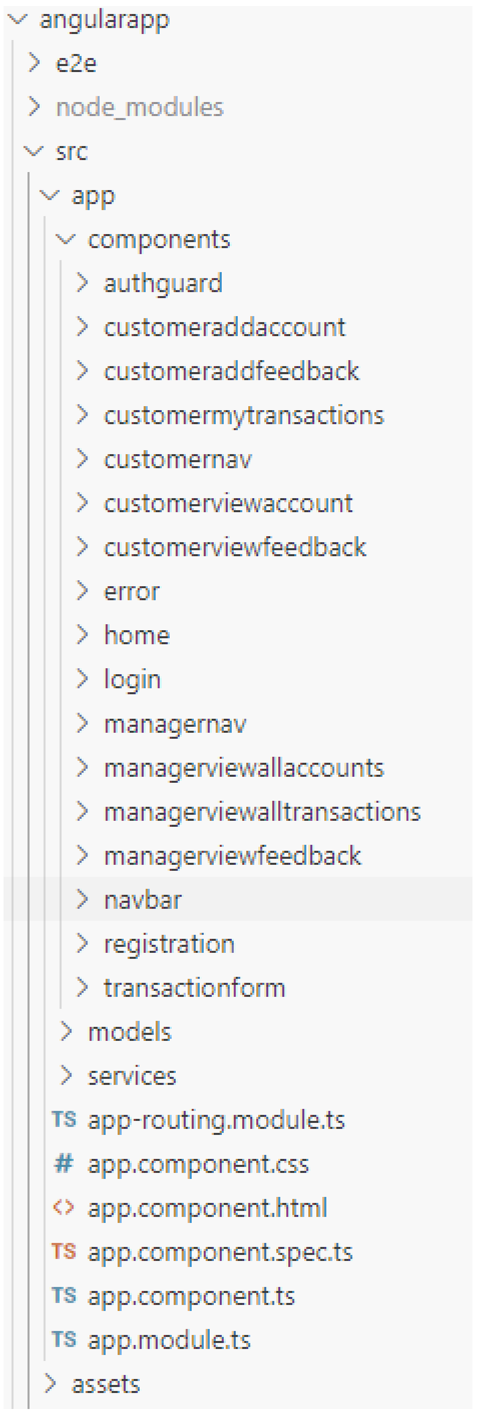
400 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

Frontend Requirements:

- Create a folder named components inside the app to store all the components. (Refer project structure screenshots).
- Create a folder named models inside app to store all the model interface.
- Create a folder named as services inside app to implement all the services.
- Create model interface referring the backend entities (User, Account, Transaction, Feedback) mentioned in the backend requirements accordingly.
- You can create your own components based on the application requirements.
- Import model files, services and components as required.

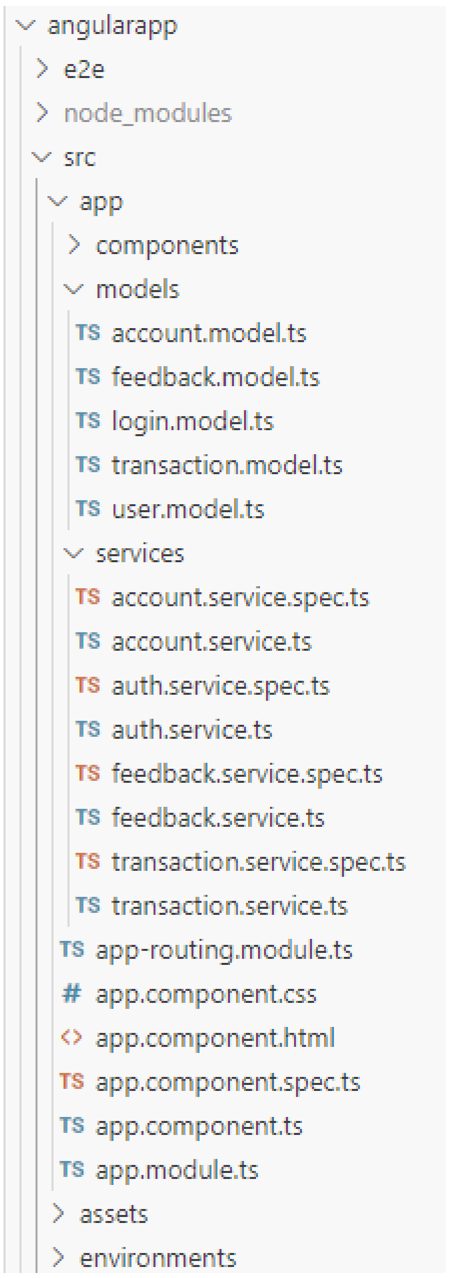
Project Folder Screenshot:

Components:



```
angularapp
  e2e
  node_modules
  src
    app
      components
        authguard
        customeraddaccount
        customeraddfeedback
        customermytransactions
        customernav
        customerviewaccount
        customerviewfeedback
        error
        home
        login
        managernav
        managerviewallaccounts
        managerviewalltransactions
        managerviewfeedback
        navbar
        registration
        transactionform
      models
      services
      app-routing.module.ts
      app.component.css
      app.component.html
      app.component.spec.ts
      app.component.ts
      app.module.ts
    assets
```

Services and Models:



```
angularapp
  e2e
  node_modules
  src
    app
      components
      models
        account.model.ts
        feedback.model.ts
        login.model.ts
        transaction.model.ts
        user.model.ts
      services
        account.service.spec.ts
        account.service.ts
        auth.service.spec.ts
        auth.service.ts
        feedback.service.spec.ts
        feedback.service.ts
        transaction.service.spec.ts
        transaction.service.ts
      app-routing.module.ts
      app.component.css
      app.component.html
      app.component.spec.ts
      app.component.ts
      app.module.ts
    assets
    environments
```

Frontend Models:

User Model:

```
class User {
UserId?: number;
Email: string;
Password: string;
Username: string;
MobileNumber: string;
UserRole: string;
}
```

Login Model:

```
class Login {
Email: string;
Password: string;
}
```

Transaction Model:

```
interface Transaction {
    TransactionId: number;
    AccountId: number;
    TransactionDate: Date;
    TransactionType: string;
    Amount: number;
    Status?: string;
    Description?: string;
    Account?: Account;
  }
```

Account Model:

```
class Account {
    AccountId?: number;
    UserId: number;
    AccountHolderName: string;
    AccountType: string;
    Balance: number;
    Status: string;
    ProofOfIdentity: string;
    DateCreated?: Date;
    LastUpdated?: Date;
}
```

Feedback Model:

```
class Feedback {
  FeedbackId?: number;
  UserId: number;
  Comments: string;
  DateProvided: Date;
}
```

Frontend services:

- Declare a public property apiUrl to store the backend URL in all the services.
- For example, public apiUrl = 'http://localhost:8080'. Instead of 'localhost', replace it with the URL of your workspace port 8080 URL.
- For the API's to be used please refer the API Table.
- Authorized token to be passed in headers for all end points.

1. AuthService(auth.service.ts):

Create a service name as **auth** inside app/services folder to implement the following functions.

Methods Overview:

- register(user: User): Observable<any>:
  - Use this method to register a new user. It sends a POST request to the '/api/register' endpoint with the user data provided as the body.

- login(login : Login): Observable<any>:
  - This method is used to authenticate a user by logging them in. It sends a POST request to the '/api/login' endpoint with the user's email and password. Upon successful login, it stores the JWT token in localStorage and updates the user's role and ID using BehaviorSubjects.

2. **AccountService (services/account.service.ts)**

The AccountService class is an Angular service responsible for managing account-related operations. It interacts with the backend API to perform CRUD (Create, Read, Update, Delete) operations on account data. This service ensures that all requests include the necessary authorization token stored in localStorage.

Functions:

1. **createAccount(account: Account): Observable<any>**
   - **Description:** This method creates a new account.
   - **Implementation:** Sends a POST request to the /api/account endpoint with the account data provided as the body.
   - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
2. **getAccountById(accountId: number): Observable<Account>**
   - **Description:** Retrieves a specific account by its unique identifier.
   - **Implementation:** Sends a GET request to the /api/account/:accountId endpoint, replacing :accountId with the provided account ID.
   - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
3. **getAccountByUserId(userId: number): Observable<Account[]>**
   - **Description:** Fetches all accounts associated with a specific user identified by userId.
   - **Implementation:** Sends a GET request to the /api/account/user/:userId endpoint, replacing :userId with the provided user ID.
   - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
4. **getAllAccounts(): Observable<Account[]>**
   - **Description:** Retrieves all accounts from the server.
   - **Implementation:** Sends a GET request to the /api/account endpoint to fetch the list of accounts.
   - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
5. **updateAccount(account: Account): Observable<any>**
   - **Description:** Updates an existing account with new data.
   - **Implementation:** Sends a PUT request to the /api/account/:accountId endpoint, replacing :accountId with the account's unique identifier, and includes the updated account data in the request body.
   - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.

3. **TransactionService (services/transaction.service.ts)**

The TransactionService class is an Angular service designed to manage transaction-related operations within the application. It communicates with the backend API to perform CRUD (Create, Read, Update, Delete) operations on transaction data. The service ensures that all requests are authorized by including a token stored in localStorage.

Functions:

1. **getAllTransactions(): Observable<Transaction[]>**
   - **Description:** Fetches all transactions from the server.
   - **Implementation:** Sends a GET request to the /api/transaction endpoint to retrieve a list of all transactions.
   - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
2. **addTransaction(transaction: Transaction): Observable<any>**
   - **Description:** Adds a new transaction to the system.
   - **Implementation:** Sends a POST request to the /api/transaction endpoint with the transaction data provided as the body.

    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
3. **updateTransaction(transaction: Transaction): Observable<any>**
    - **Description:** Updates an existing transaction using its unique identifier.
    - **Implementation:** Sends a PUT request to the /api/transaction/manager/:transactionId endpoint, where :transactionId is replaced with the transaction's ID, and includes the updated transaction data in the request body.
    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
4. **getTransactionsByUserId(userId: number): Observable<Transaction[]>**
    - **Description:** Retrieves all transactions associated with a specific user identified by userId.
    - **Implementation:** Sends a GET request to the /api/transaction/customer/:userId endpoint, replacing :userId with the provided user ID.
    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.

4. **FeedbackService (services/feedback.service.ts)**
The FeedbackService class is an Angular service designed to manage feedback-related operations within the application. It interacts with the backend API to perform operations related to user feedback, including submitting new feedback, retrieving existing feedback, and deleting feedback entries. The service ensures that all requests are authorized by including a token stored in localStorage.
Functions:

1. **sendFeedback(feedback: Feedback): Observable<Feedback>**
    - **Description:** Sends user feedback to the server.
    - **Implementation:** Sends a POST request to the /api/feedback endpoint with the feedback data provided as the body.
    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
2. **getAllFeedbacksByUserId(userId: string): Observable<Feedback[]>**
    - **Description:** Retrieves all feedback submitted by a specific user identified by userId.
    - **Implementation:** Sends a GET request to the /api/feedback/user/:userId endpoint, where :userId is replaced with the provided user ID.
    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
3. **deleteFeedback(feedbackId: string): Observable<void>**
    - **Description:** Deletes a specific feedback entry identified by feedbackId.
    - **Implementation:** Sends a DELETE request to the /api/feedback/:feedbackId endpoint, where :feedbackId is replaced with the feedback ID to be deleted.
    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.
4. **getFeedbacks(): Observable<Feedback[]>**
    - **Description:** Fetches all feedback from the server.
    - **Implementation:** Sends a GET request to the /api/feedback endpoint to retrieve a list of all feedback entries.
    - **Authorization:** Includes the authorization token prefixed with 'Bearer' from localStorage in the request headers.

Validations:

Client-Side Validation:

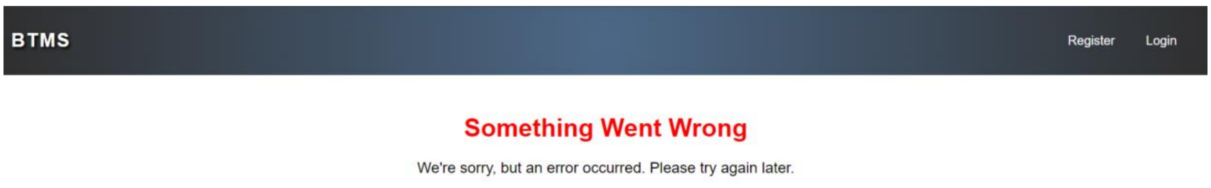
- Implement client-side validation using HTML5 attributes and JavaScript to validate user input before making API requests.

- Provide immediate feedback to users for invalid input, such as displaying error messages near the input fields.

Server-Side Validation:
- Implement server-side validation in the controllers to ensure data integrity.
- Validate user input and API responses to prevent unexpected or malicious data from affecting the application.
- Return appropriate validation error messages to the user interface for any validation failures.

Exception Handling:
- Implement exception handling mechanisms in the controllers to gracefully handle errors and exceptions. Define custom exception classes for different error scenarios, such as API communication errors or database errors.
- Log exceptions for debugging purposes while presenting user-friendly error messages to users. Record all the exceptions and errors handled store in separate table “ErrorLogs”.

Error Pages:
Create custom error pages for different HTTP status codes (e.g., 404 Not Found, 500 Internal Server Error) to provide a consistent and user-friendly error experience. Ensure that error pages contain helpful information and guidance for users.
Thus, create a reliable and user-friendly web application that not only meets user expectations but also provides a robust and secure experience, even when faced with unexpected situations. Error page has to be displayed if something goes wrong.



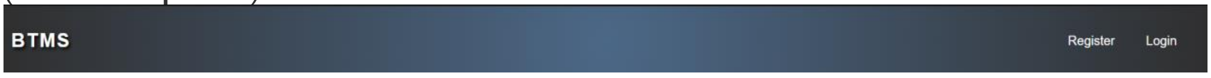
Something Went Wrong

We're sorry, but an error occurred. Please try again later.

Frontend Screenshots:
• All the asterisk (*) marked fields are mandatory in the form. Make sure to mark all the field names with * symbol followed by the validations.

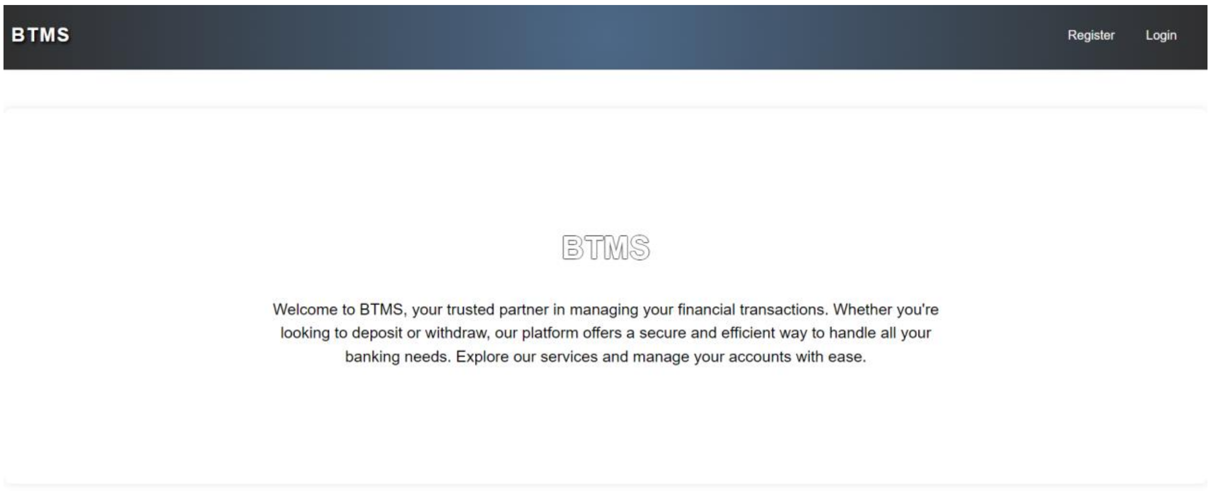
Navigation Bar:

(**navbar component**)



Component displays a title along with router links to "Register" and "Login".

Landing Page:
(**home component**)



BTMS

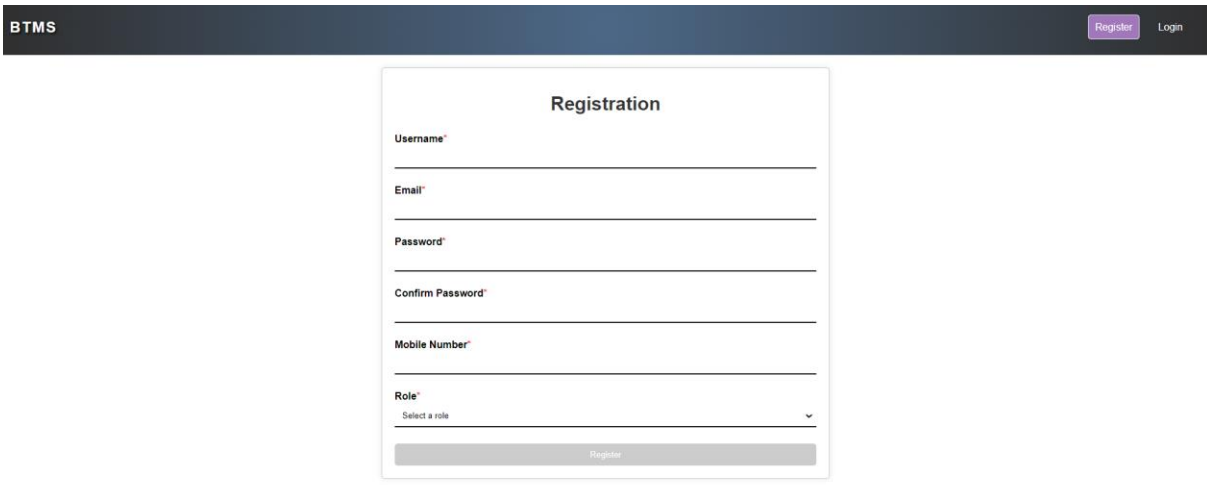
Welcome to BTMS, your trusted partner in managing your financial transactions. Whether you're looking to deposit or withdraw, our platform offers a secure and efficient way to handle all your banking needs. Explore our services and manage your accounts with ease.

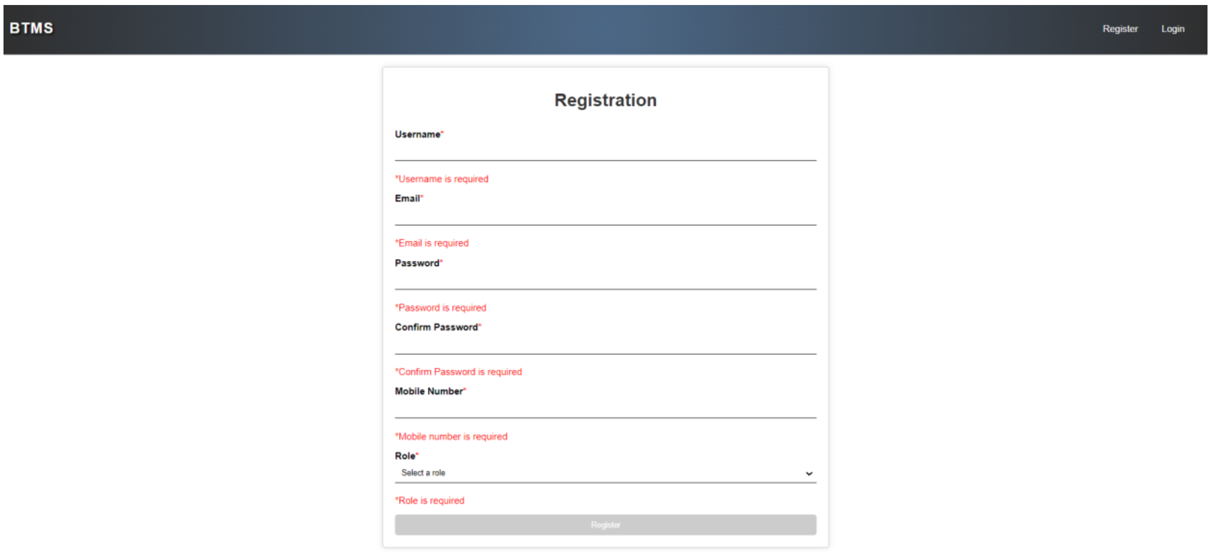
Component features a heading "BTMS" accompanied by an introductory message that provides an overview of the application.

Application side (Manager and Customer):

Registration Page:
(**registration component**)

Clicking "Register" in the navbar displays the registration page for both roles. Please refer to the screenshots below for validations.




BTMS Register Login

Registration

Username*

*Username is required

Email*

*Email is required

Password*

*Password is required

Confirm Password*

*Confirm Password is required

Mobile Number*

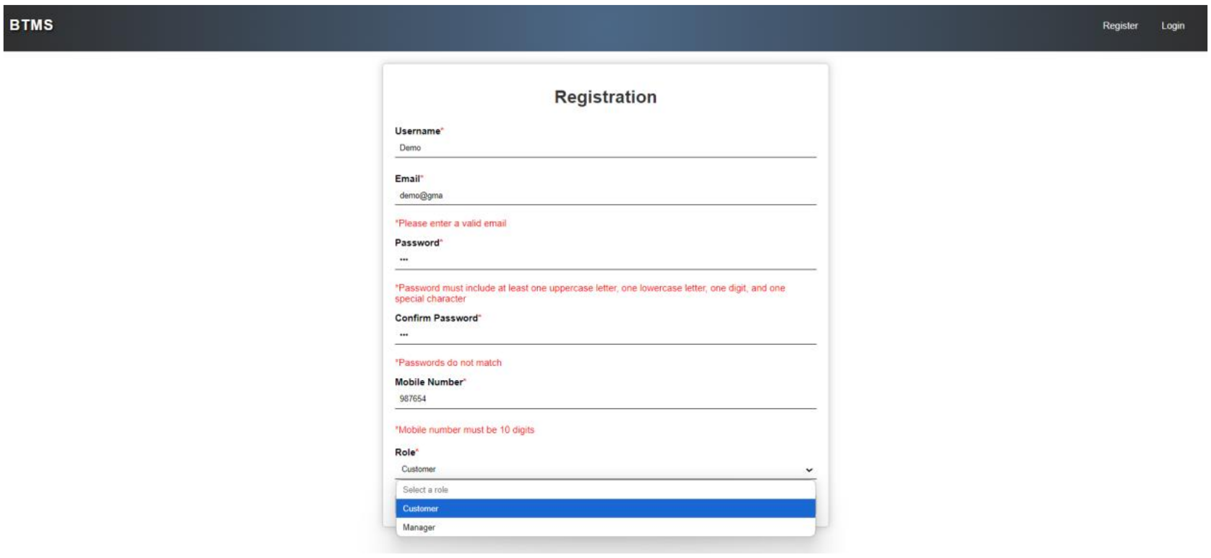
*Mobile number is required

Role*

Select a role

*Role is required

Register


BTMS Register Login

Registration

Username*

Demo

Email*

demo@gma

*Please enter a valid email

Password*

•••

*Password must include at least one uppercase letter, one lowercase letter, one digit, and one special character

Confirm Password*

•••

*Passwords do not match

Mobile Number*

987654

*Mobile number must be 10 digits

Role*

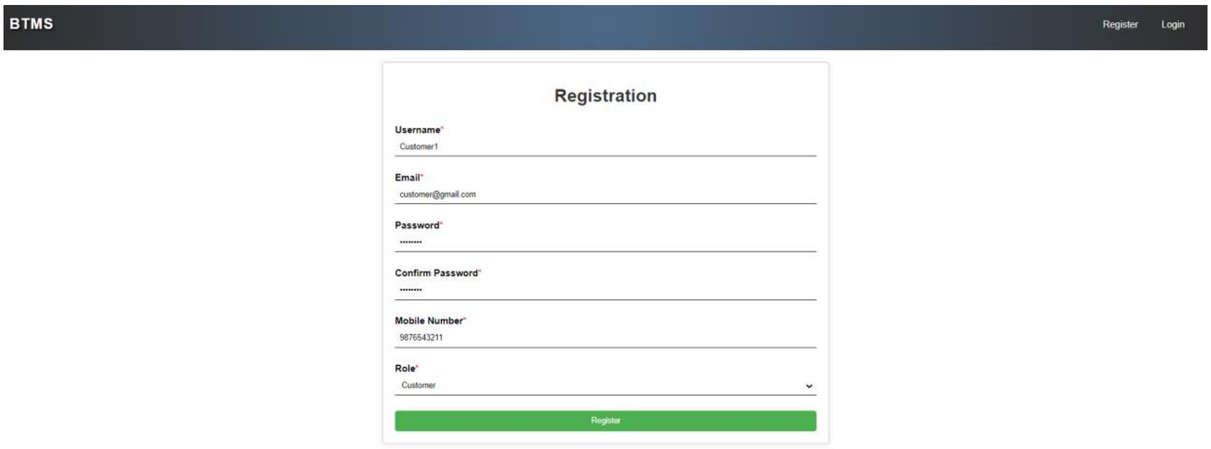
Customer

Select a role

Customer

Manager

The “Register” button should remain disabled until all validations are cleared. Once all validations are met, the “Register” button will be enabled.


BTMS Register Login

Registration

Username*

Customer1

Email*

customer@gmail.com

Password*

••••••••

Confirm Password*

••••••••

Mobile Number*

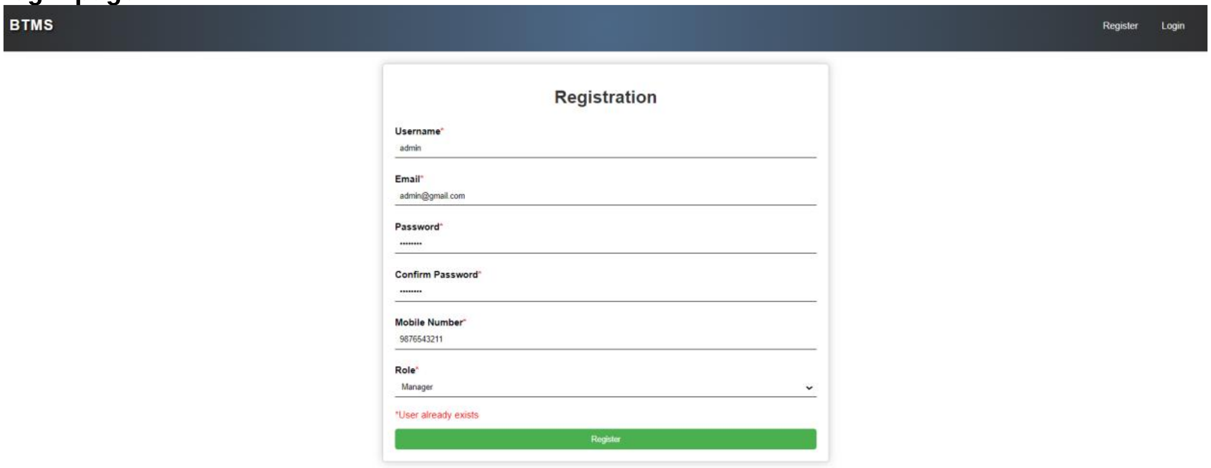
9876543211

Role*

Customer

Register

When the "Register" button is clicked, upon successful submission, the user must be navigated to the **login page**.


BTMS
Register Login

Registration

Username*
admin

Email*
admin@gmail.com

Password*
••••••••

Confirm Password*
••••••••

Mobile Number*
9876543211

Role*
Manager

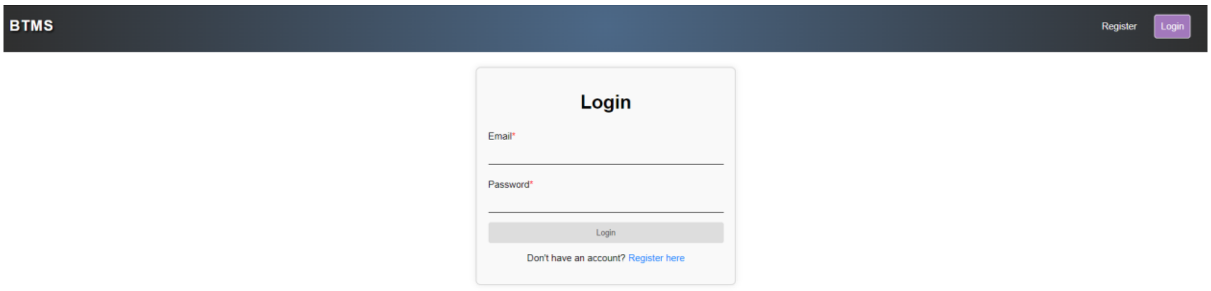
*User already exists

Register

If a user attempts to register with an existing email, a message stating "User already exists" will be displayed.

Login Page
(**login component**)

This page is used for logging in to the application. On providing the valid email and password, the user will be logged in.


BTMS
Register Login

Login

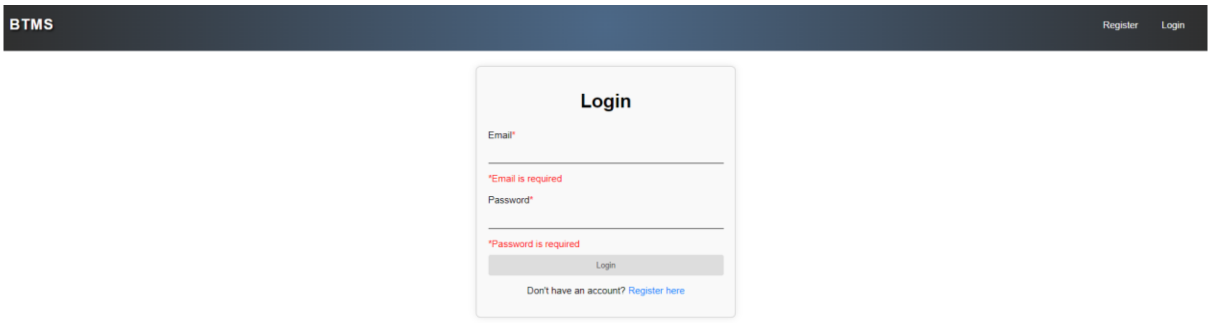
Email*

Password*

Login

Don't have an account? Register here

Perform validations for email and password fields.


BTMS
Register Login

Login

Email*

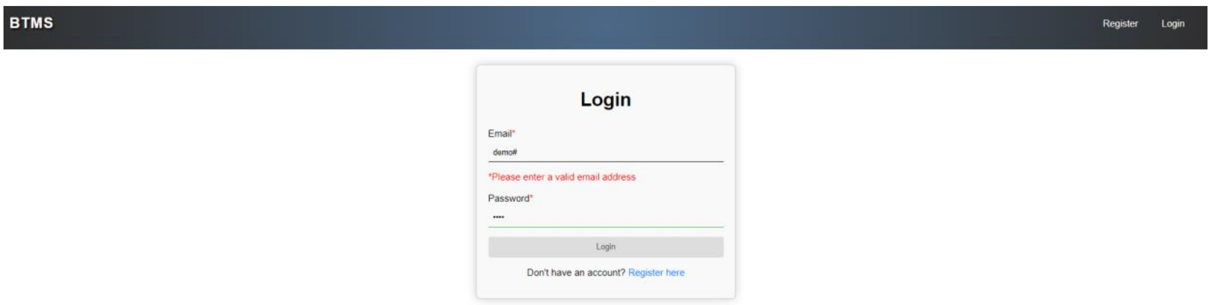
*Email is required

Password*

*Password is required

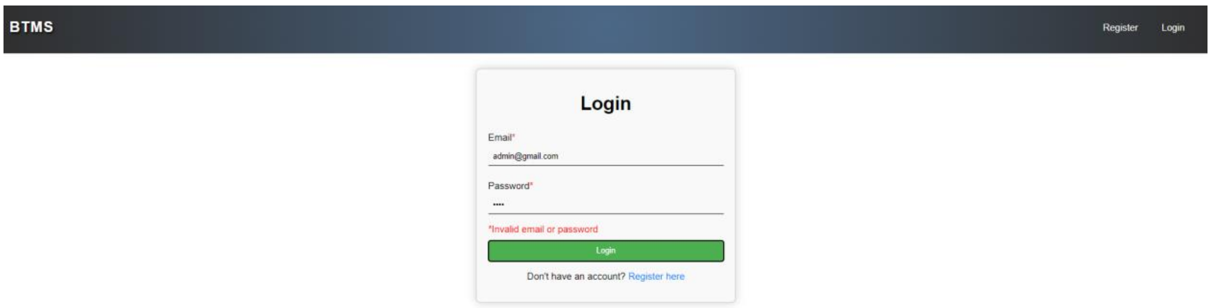
Login

Don't have an account? Register here



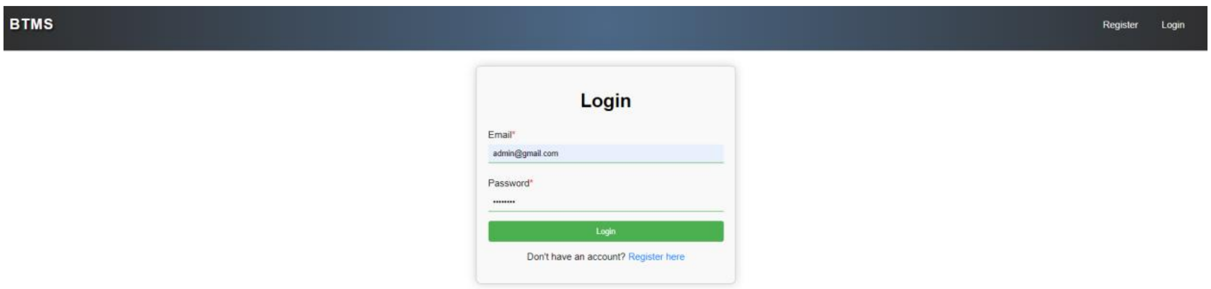
Email*
demo#
*Please enter a valid email address
Password*
••••

The email field uses regex validation. If the input does not meet the standard email format, it will display the message "Please enter a valid email address."



Email*
admin@gmail.com
Password*
••••
*Invalid email or password

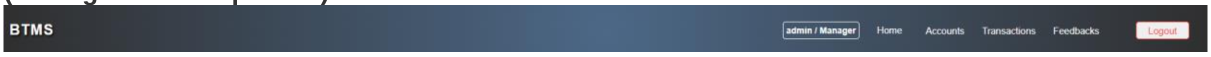
If a non-registered or incorrect email or password is used to log in, it will display the message "Invalid email or password."



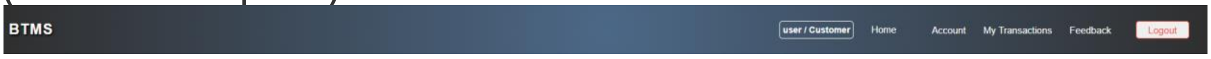
Email*
admin@gmail.com
Password*
••••••••

Upon clicking the 'Login' button and successfully authenticating, the page will be navigated to either the **managernav** component or the **customernav** component, depending on the user's role (Manager or Customer).

(managernav component)

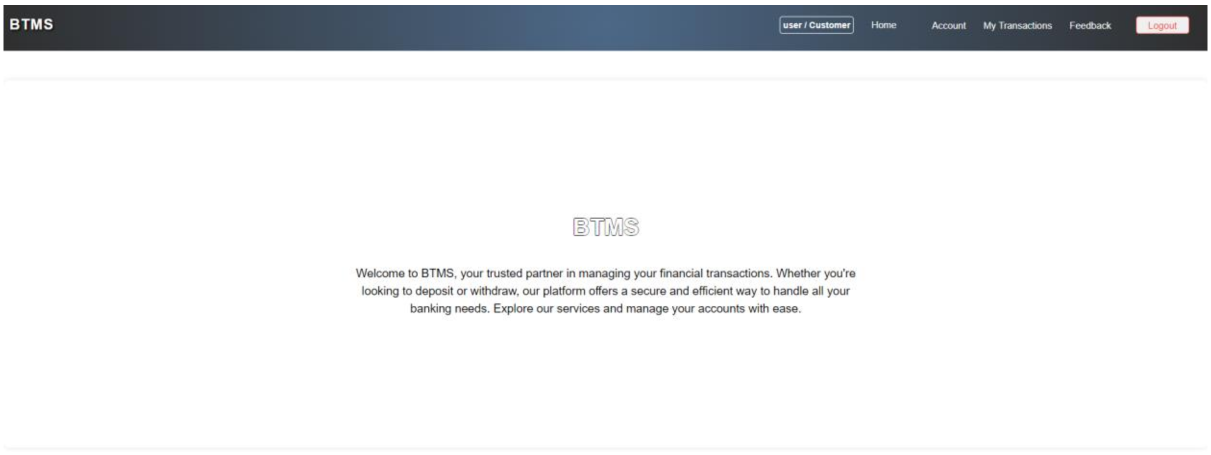


(customernav component)



Customer side:

Home Component: This page is used to display the information about the bank application. On clicking the 'Home' tab, customer can view the information about the application.


BTMS

user / Customer · Home · Account · My Transactions · Feedback · Logout

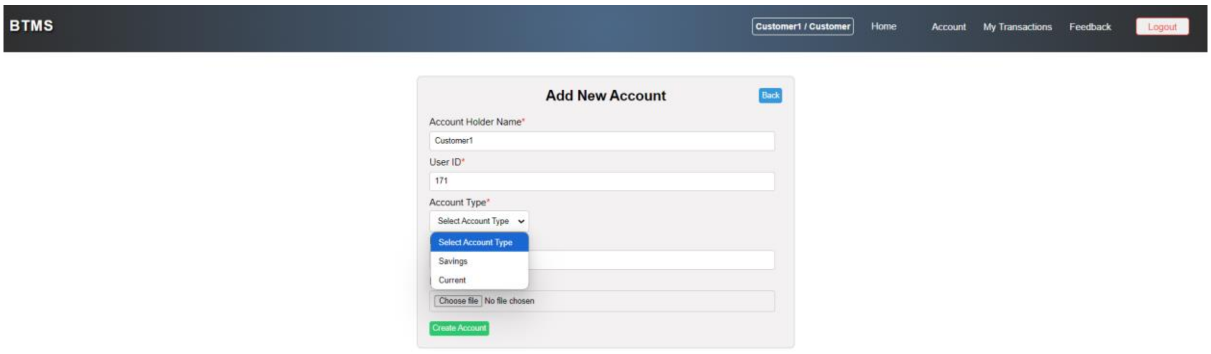
BTMS

Welcome to BTMS, your trusted partner in managing your financial transactions. Whether you're looking to deposit or withdraw, our platform offers a secure and efficient way to handle all your banking needs. Explore our services and manage your accounts with ease.

Upon successful login. If the user is a regular user, the **(customernav component)** will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.

Customer Add Account: npx ng g c components/Customeraddaccount

Clicking on "Accounts -> Add Accound" from the navbar will navigate to the **Customeraddaccount component**, which display form to fill the account creation details.


BTMS

Customer1 / Customer · Home · Account · My Transactions · Feedback · Logout

Add New Account

Back

Account Holder Name*: Customer1

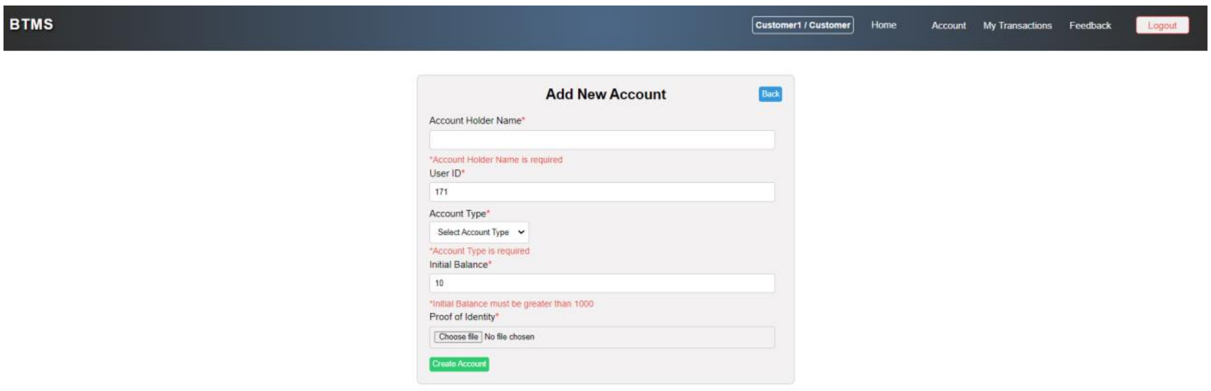
User ID*: 171

Account Type*: Select Account Type

Select Account Type
Savings
Current

Choose file No file chosen

Create Account


BTMS

Customer1 / Customer · Home · Account · My Transactions · Feedback · Logout

Add New Account

Back

Account Holder Name*:

*Account Holder Name is required

User ID*: 171

Account Type*: Select Account Type

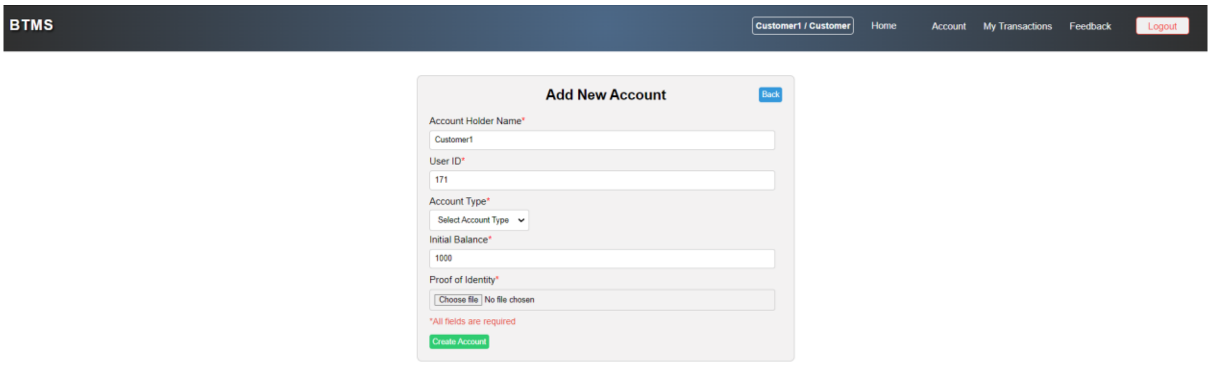
*Account Type is required

Initial Balance*: 10

*Initial Balance must be greater than 1000

Proof of Identity*: Choose file No file chosen

Create Account

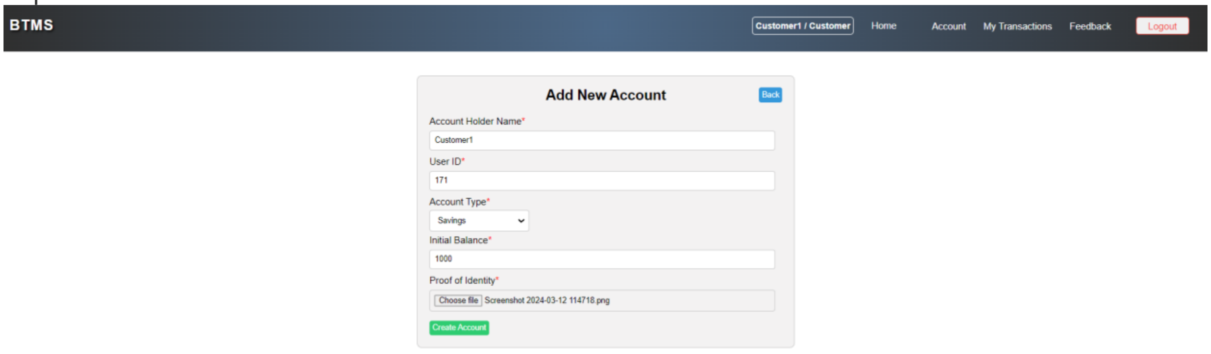

BTMS
Customer1 / Customer | Home | Account | My Transactions | Feedback | Logout

Add New Account

Back

Account Holder Name*
Customer1
User ID*
171
Account Type*
Select Account Type
Initial Balance*
1000
Proof of Identity*
Choose file | No file chosen
*All fields are required
Create Account

UserId should be prepopulated and should be read only mode.
Implement all the validations has shown in the above Screen Shots.

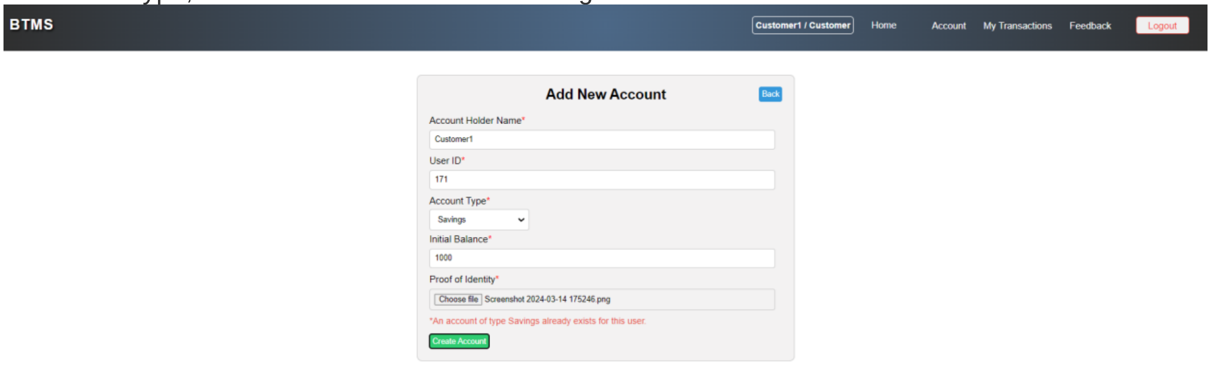

BTMS
Customer1 / Customer | Home | Account | My Transactions | Feedback | Logout

Add New Account

Back

Account Holder Name*
Customer1
User ID*
171
Account Type*
Savings
Initial Balance*
1000
Proof of Identity*
Choose file | Screenshot 2024-03-12 114718.png
Create Account

On clicking the Create Account button with valid data, will show success pop up & navigate to **customerviewaccount** Component. If the customer has been already created a account with the mentioned type, then it should show error message

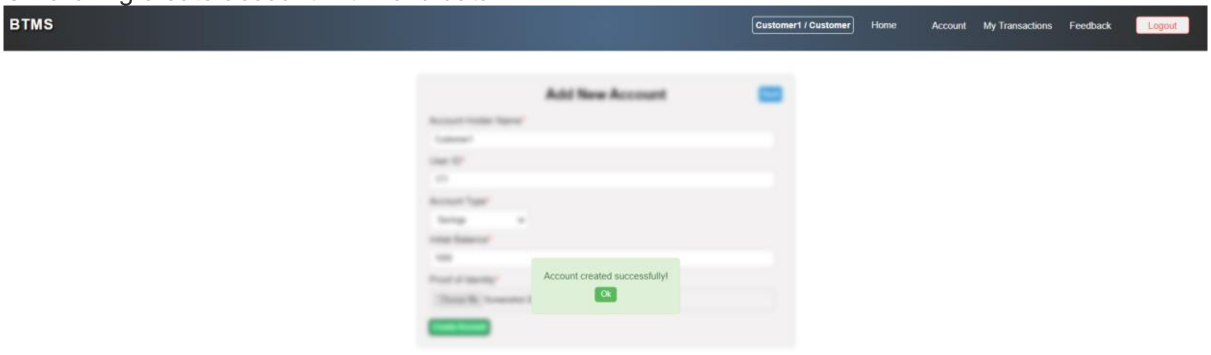

BTMS
Customer1 / Customer | Home | Account | My Transactions | Feedback | Logout

Add New Account

Back

Account Holder Name*
Customer1
User ID*
171
Account Type*
Savings
Initial Balance*
1000
Proof of Identity*
Choose file | Screenshot 2024-03-14 175246.png
*An account of type Savings already exists for this user.
Create Account

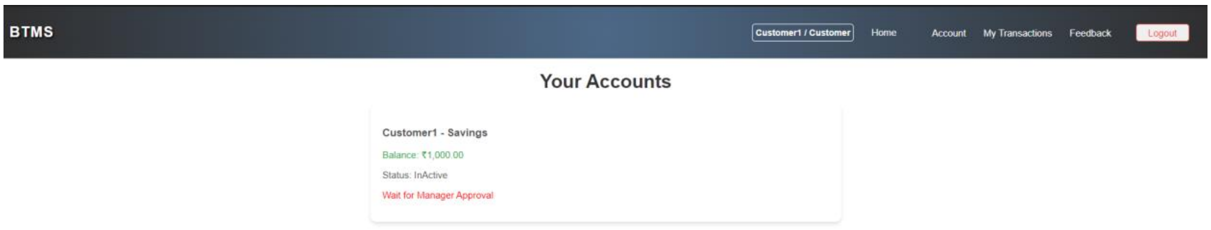
On clicking create account with valid data…


BTMS
Customer1 / Customer | Home | Account | My Transactions | Feedback | Logout

Account created successfully!
Ok

customerviewaccount Component – npx ng g c components/customerviewaccount

Display all the accounts related to the customer.



Your Accounts

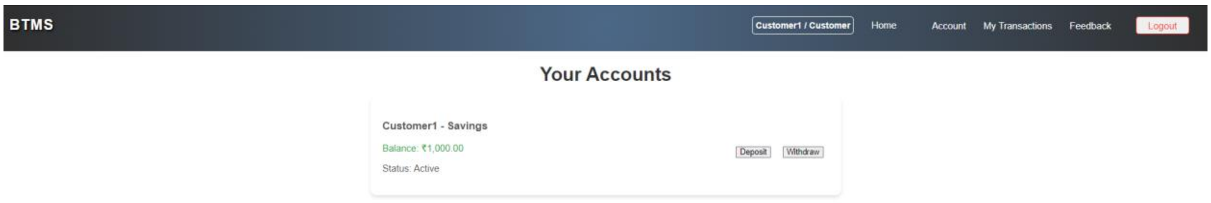
Customer1 - Savings

Balance: ₹1,000.00

Status: InActive

Wait for Manager Approval

If Account is Approved by manager then, Deposit & Withdrawal button should be displayed



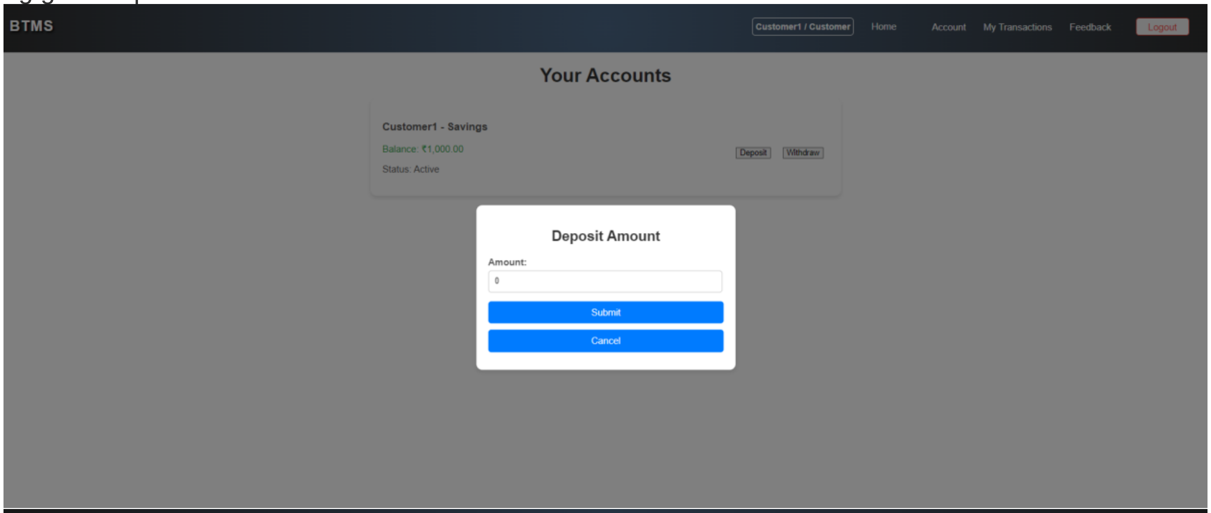
Your Accounts

Customer1 - Savings

Balance: ₹1,000.00

Status: Active

On clicking the Deposit Button or Withdrawal Button, it should call **transactionform** component – npx ng g c components/transactionform



Your Accounts

Customer1 - Savings

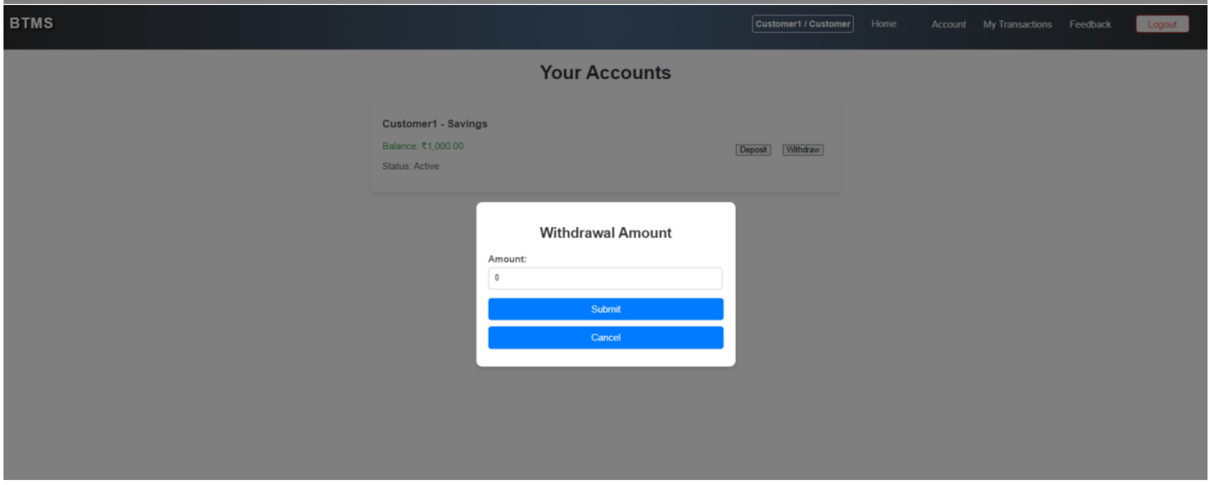
Balance: ₹1,000.00

Status: Active

Deposit Amount

Amount:

0



Your Accounts

Customer1 - Savings

Balance: ₹1,000.00

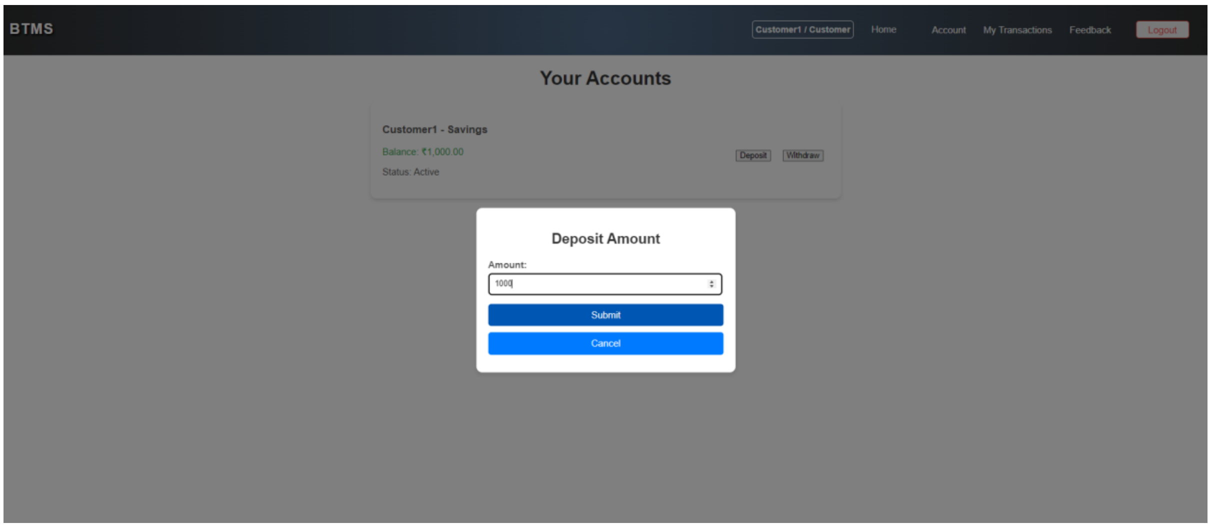
Status: Active

Withdrawal Amount

Amount:

0

On entering the valid Amount less than 10000 on deposit or withdrawal, it will process the transaction & reflect in the account



Your Accounts

Customer1 - Savings

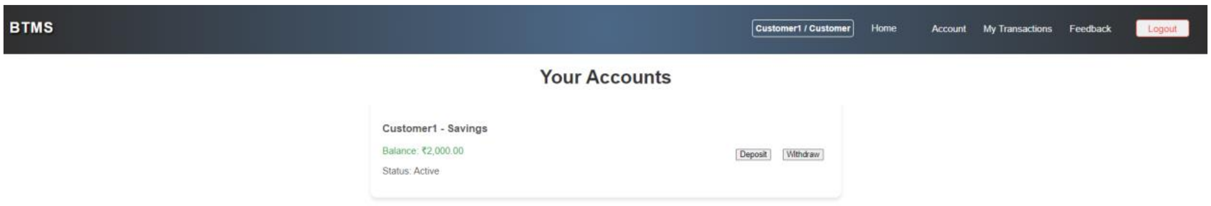
Balance: ₹1,000.00

Status: Active

Deposit Amount

Amount: 1000

Deposited 1000 into the account directly:



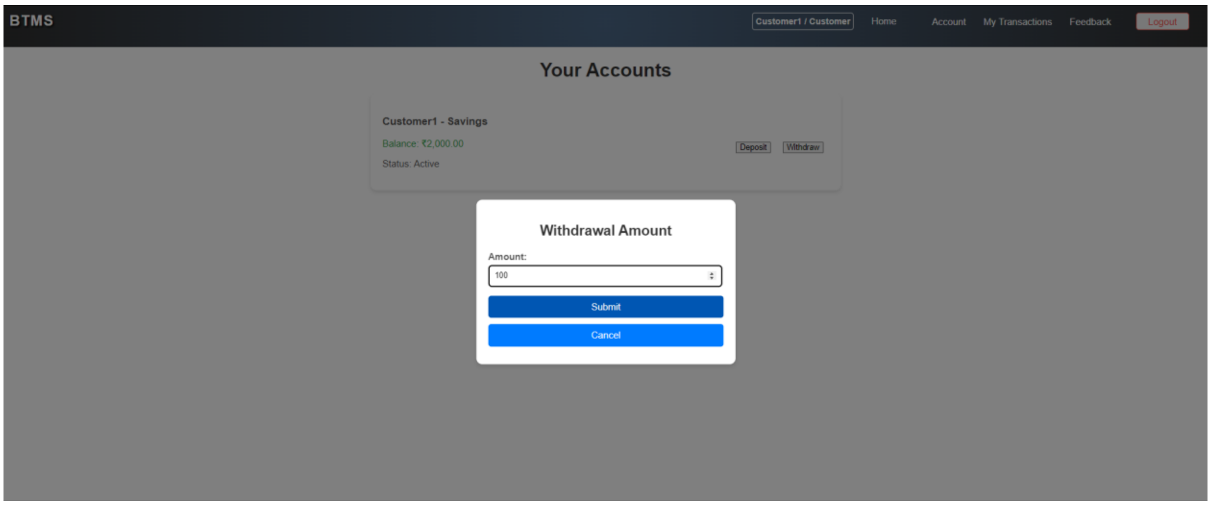
Your Accounts

Customer1 - Savings

Balance: ₹2,000.00

Status: Active

Withdrawal of amount 100 from the account will reflect in saving account:



Your Accounts

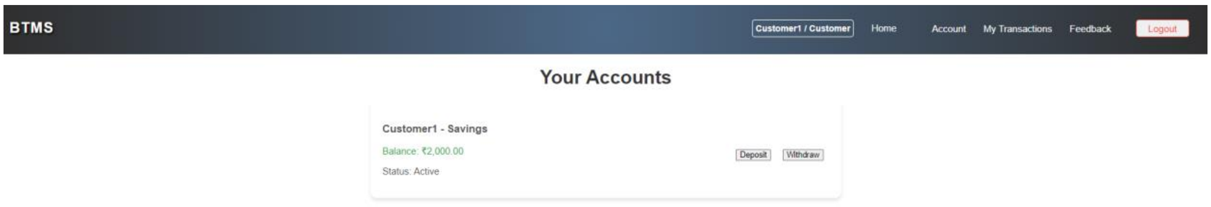
Customer1 - Savings

Balance: ₹2,000.00

Status: Active

Withdrawal Amount

Amount: 100



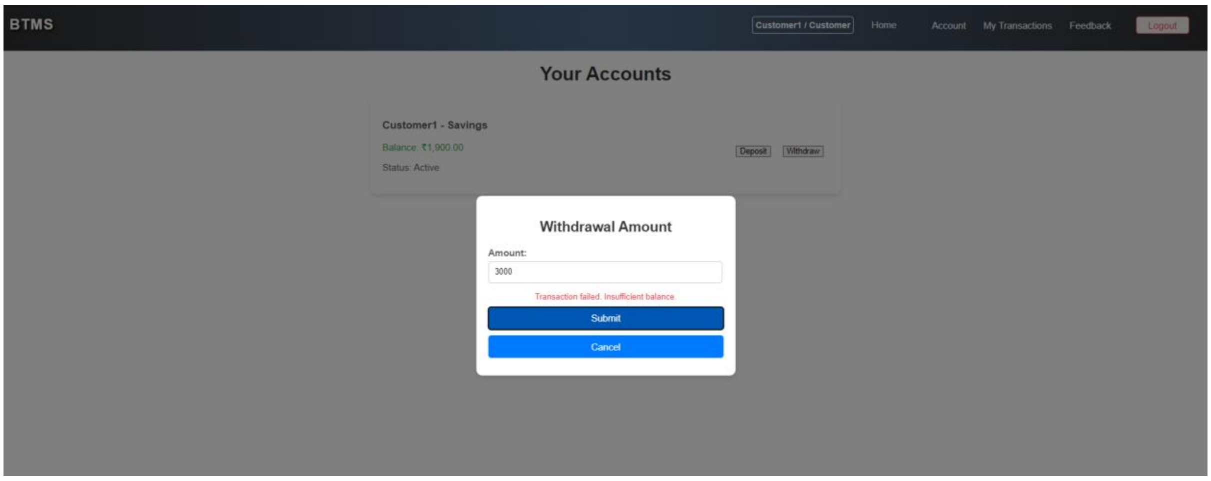
Your Accounts

Customer1 - Savings

Balance: ₹2,000.00

Status: Active

If customer tries to withdraw amount higher than his balance, then display exception message:



Your Accounts

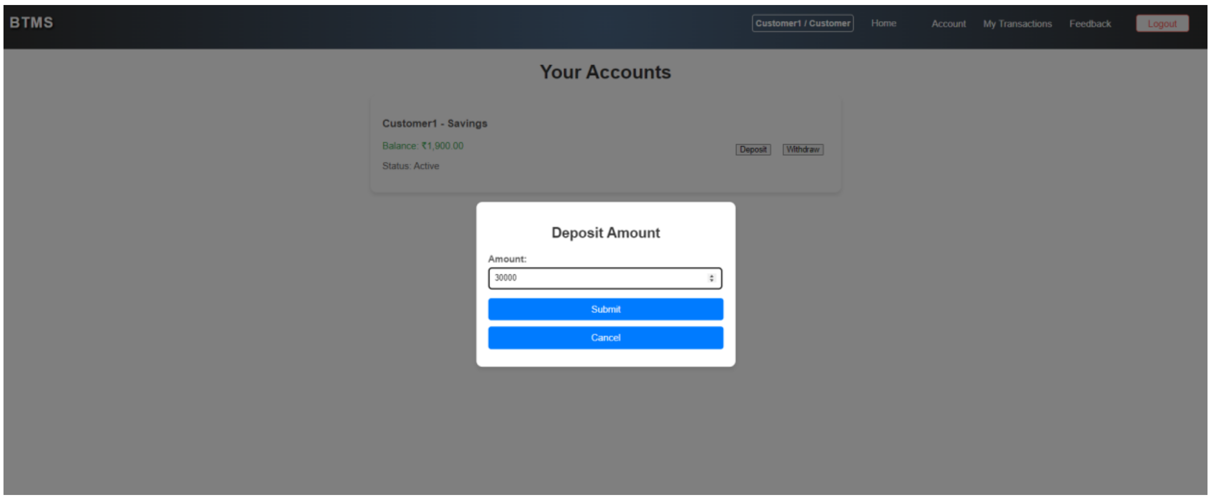
Customer1 - Savings
Balance: ₹1,900.00
Status: Active

Withdrawal Amount

Amount:
3000

Transaction failed. Insufficient balance.

If deposit or withdrawal is higher than 10000, then manager needs to approve the transaction. It will not directly reflect the account till manager approves.



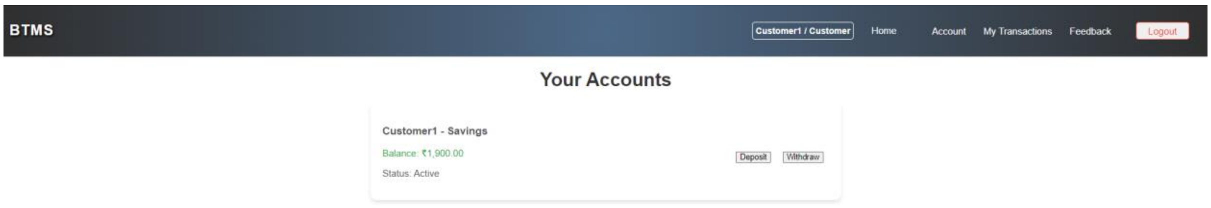
Your Accounts

Customer1 - Savings
Balance: ₹1,900.00
Status: Active

Deposit Amount

Amount:
30000

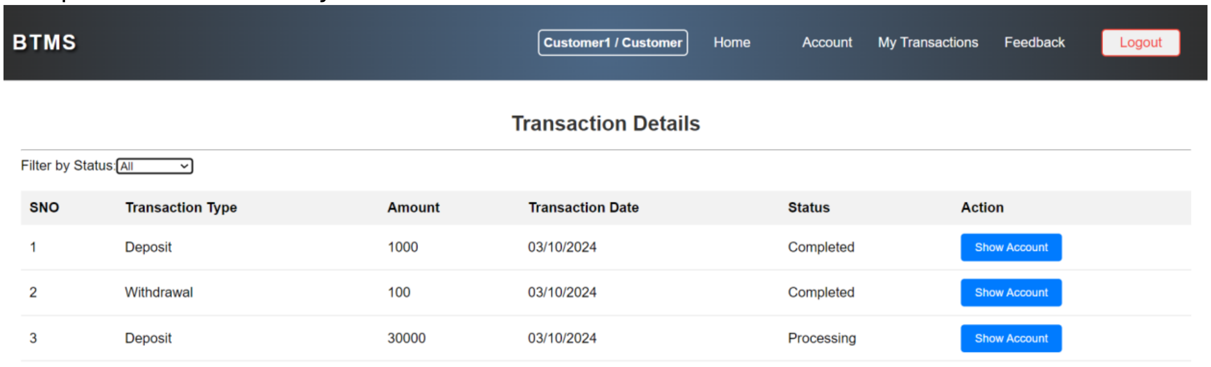
Balance is not increased, till manager approves:



Your Accounts

Customer1 - Savings
Balance: ₹1,900.00
Status: Active

On clicking the My Transactions option in customernav, will call the **customermytransactions** Component to display all the Transactions related to that customer with filter option. Npx ng g c components/customermytransactions

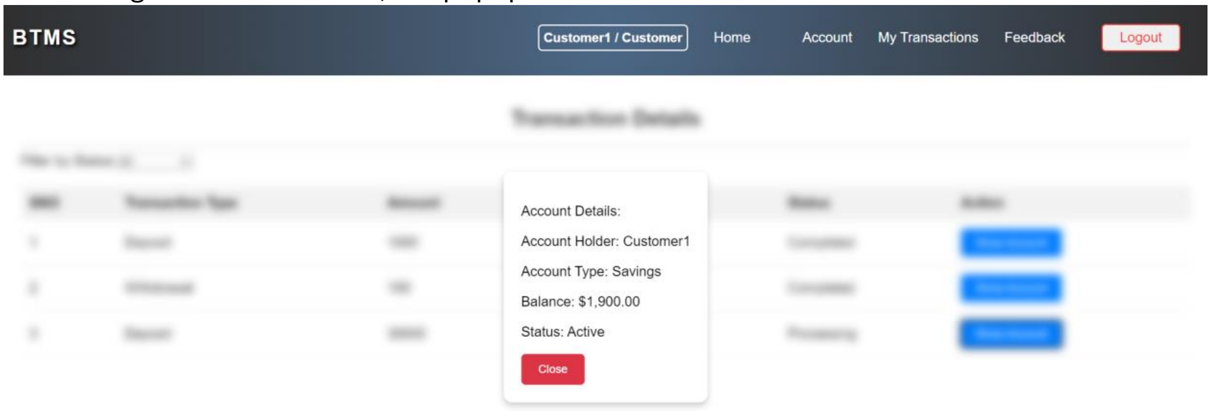


Transaction Details

Filter by Status: All

SNO	Transaction Type	Amount	Transaction Date	Status	Action
1	Deposit	1000	03/10/2024	Completed	Show Account
2	Withdrawal	100	03/10/2024	Completed	Show Account
3	Deposit	30000	03/10/2024	Processing	Show Account

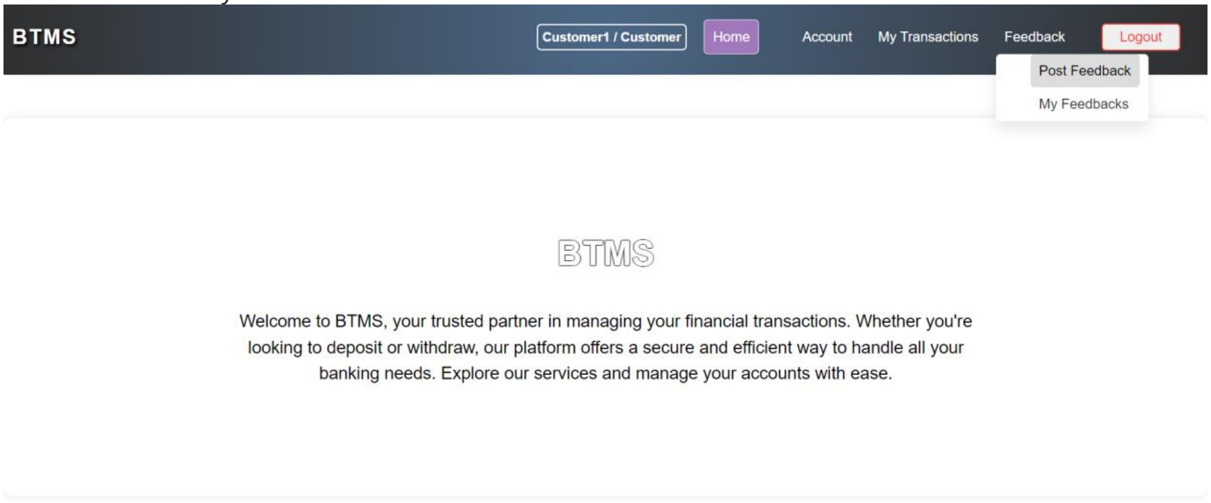
On clicking the Show Account, will popup the account details:



Account Details:
Account Holder: Customer1
Account Type: Savings
Balance: $1,900.00
Status: Active

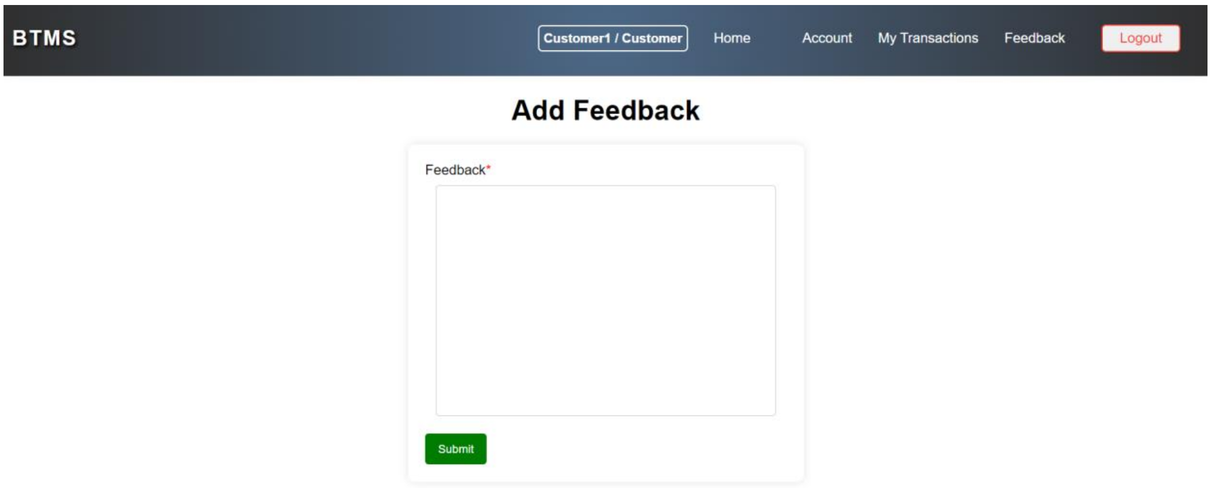
Customer Feedback:

On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".

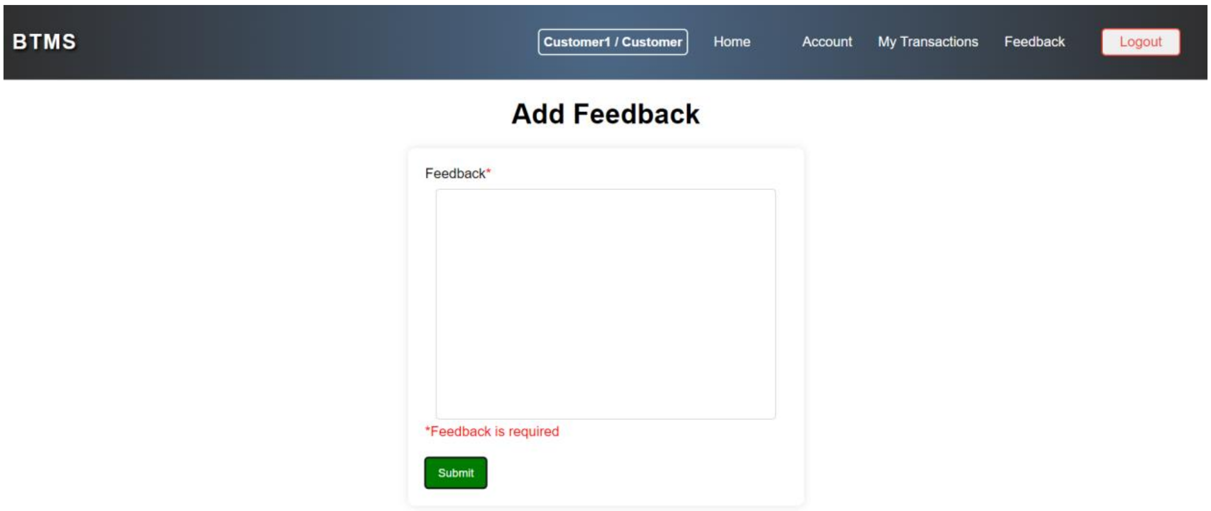


Welcome to BTMS, your trusted partner in managing your financial transactions. Whether you're looking to deposit or withdraw, our platform offers a secure and efficient way to handle all your banking needs. Explore our services and manage your accounts with ease.

Clicking on "Post Feedback" will navigate to the **customeraddfeedback component**, which displays the form to post feedback with heading as "Add Feedback"

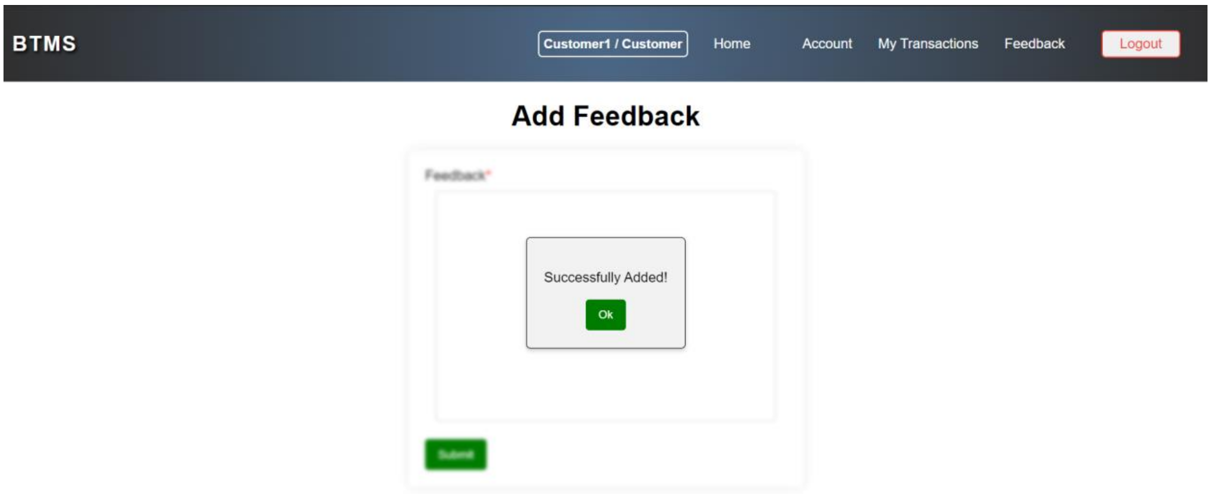


Clicking the "Submit" button with empty textarea will display a validation message stating "Feedback is required".



*Feedback is required

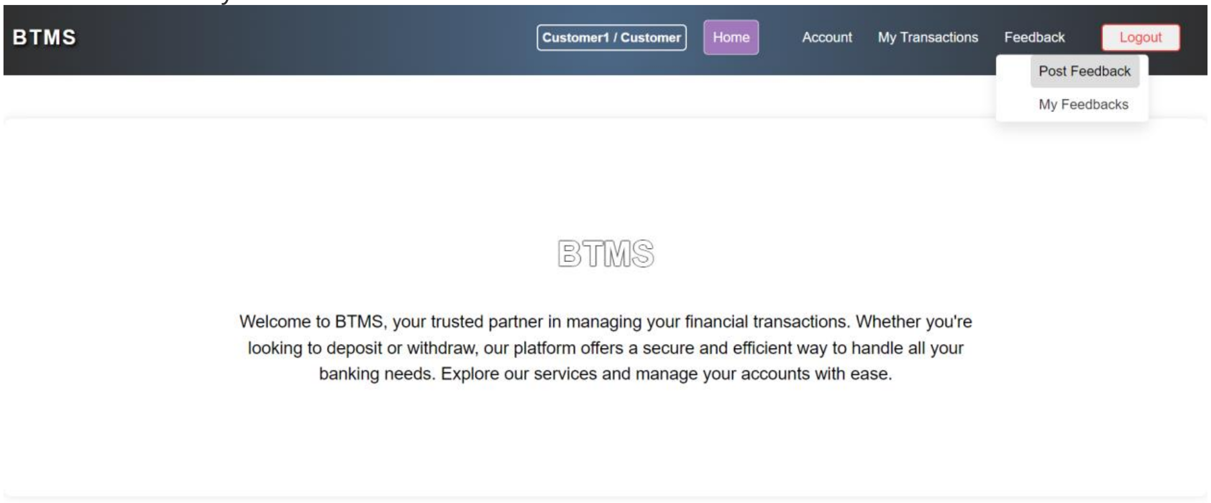
Upon clicking the "Submit" button, if the operation is successful, a popup message saying "Successfully Added!" should be displayed.



Successfully Added!

Clicking the "Ok" button will close the popup, and the same **customeraddfeedback component** will be displayed.

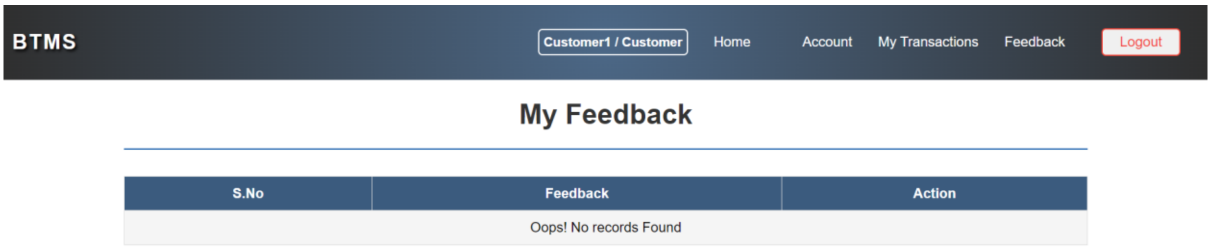
Customer view Feedback:
On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".



Welcome to BTMS, your trusted partner in managing your financial transactions. Whether you're looking to deposit or withdraw, our platform offers a secure and efficient way to handle all your banking needs. Explore our services and manage your accounts with ease.

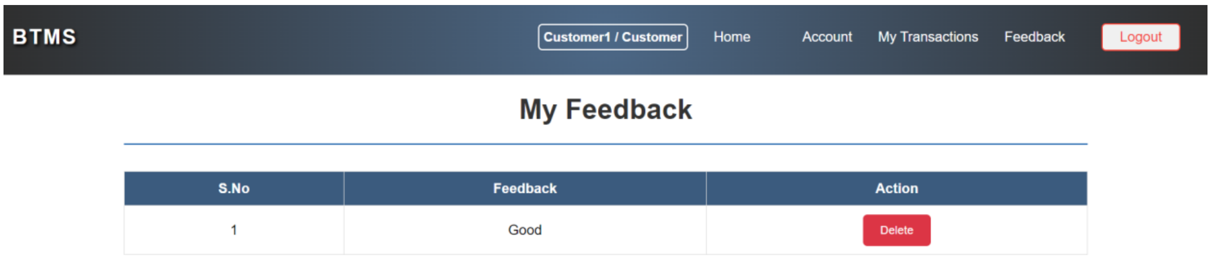
Clicking on "My Feedbacks" will navigate to the **customerviewfeedback component**, which displays posted feedback with heading as "My Feedback"

If no data is available, then "Oops! No records Found" should be displayed.



My Feedback

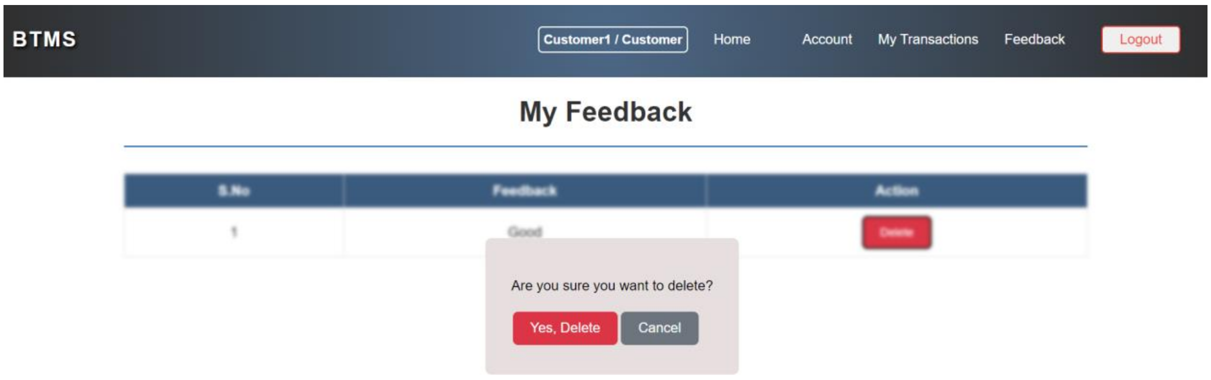
S.No	Feedback	Action
Oops! No records Found		



My Feedback

S.No	Feedback	Action
1	Good	Delete

On clicking the "Delete" button, a pop-up should be displayed with confirmatory message to delete the data.

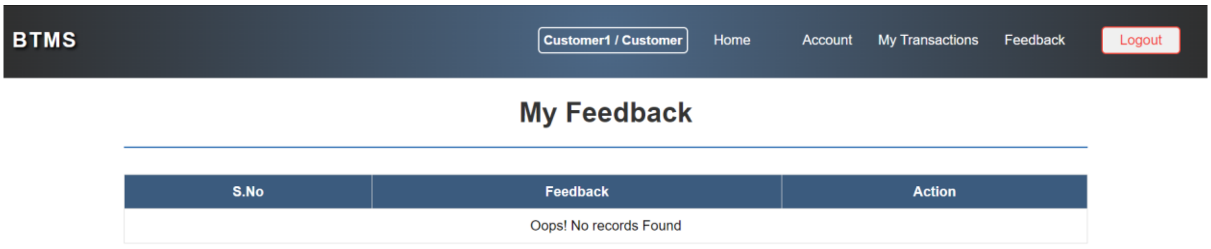


My Feedback

S.No	Feedback	Action
1	Good	Delete

Are you sure you want to delete?

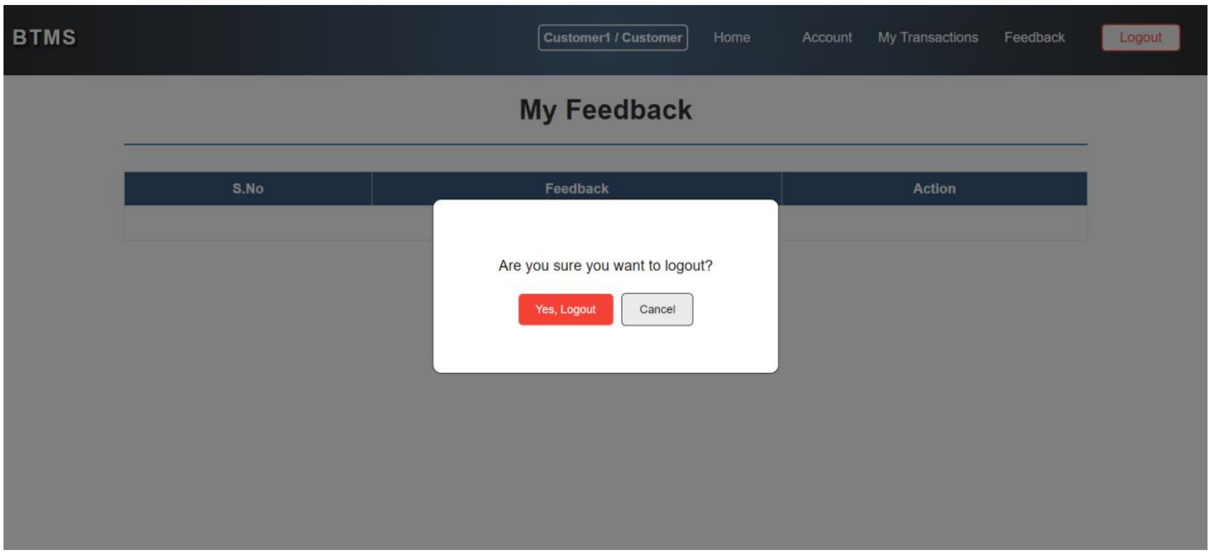
Clicking "Yes, Delete" will delete the feedback posted, and the change will be automatically reflected.



My Feedback

S.No	Feedback	Action
Oops! No records Found		

On clicking the "Logout" button, a pop-up should be displayed with confirmatory message to logout the user.

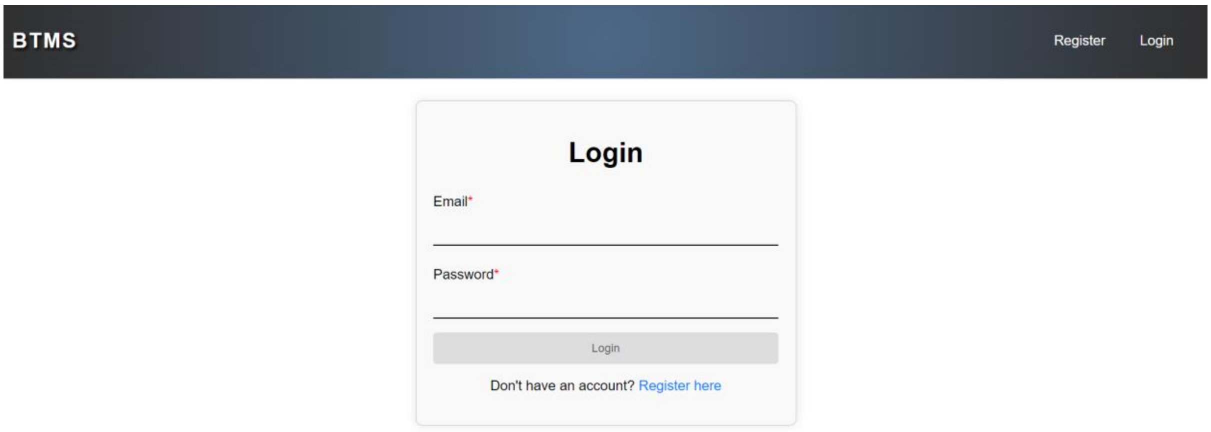


My Feedback

S.No	Feedback	Action

Are you sure you want to logout?

Clicking "Yes, Logout" will navigate to the login component.


BTMS

Register Login

Login

Email*

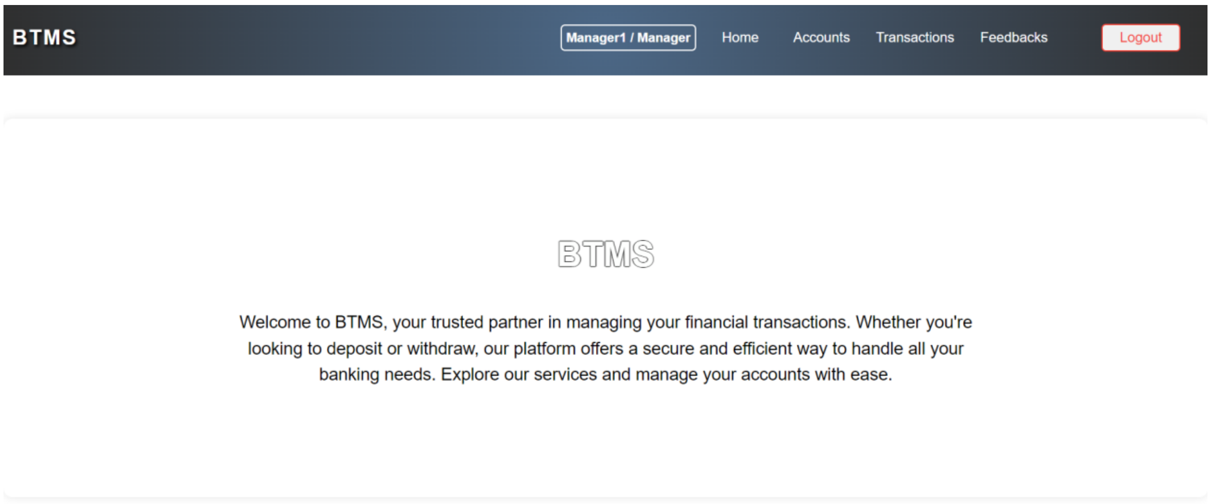
Password*

Login

Don't have an account? Register here

<u>Manager side:</u>

Home Component: This page is used to display the information about the Bank application. On clicking the '**Home**' tab, user can view the information about the application.


BTMS

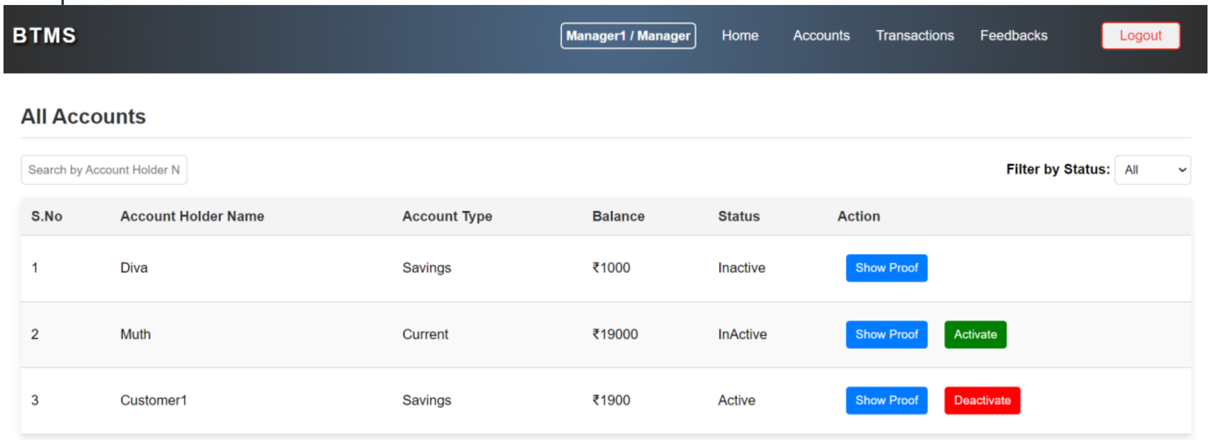
Manager1 / Manager Home Accounts Transactions Feedbacks Logout

BTMS

Welcome to BTMS, your trusted partner in managing your financial transactions. Whether you're looking to deposit or withdraw, our platform offers a secure and efficient way to handle all your banking needs. Explore our services and manage your accounts with ease.

Upon successful login, if the user is an Manager, the (managernav component) will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role

On clicking the Accounts option in the managernav, will call the **managerviewallaccounts** Component.



All Accounts

Search by Account Holder N

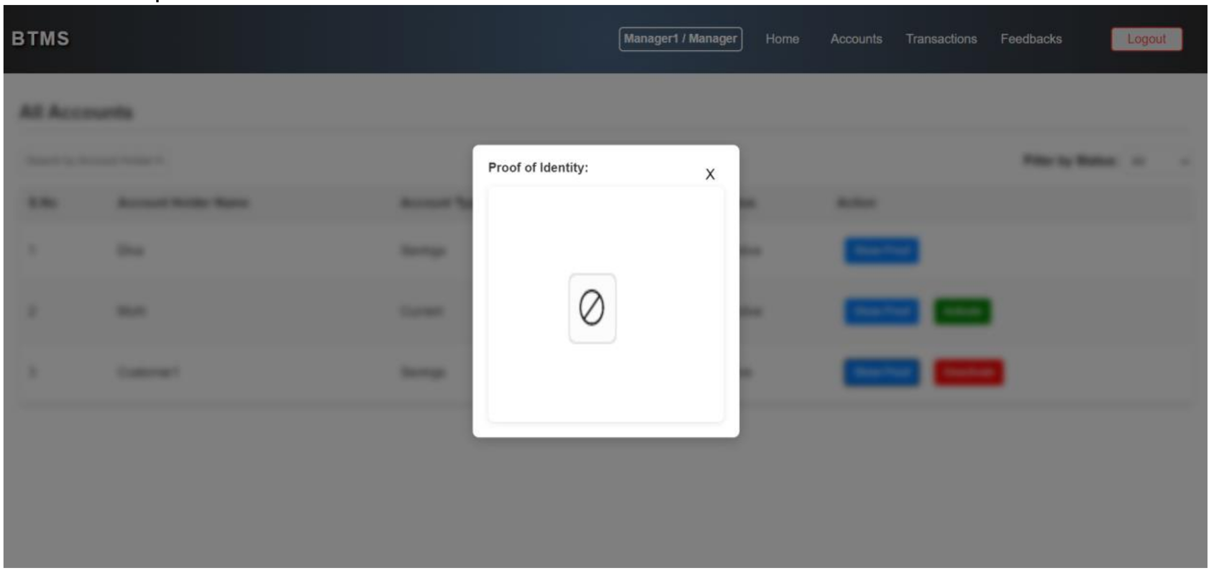
Filter by Status: All

S.No	Account Holder Name	Account Type	Balance	Status	Action
1	Diva	Savings	₹1000	Inactive	Show Proof
2	Muth	Current	₹19000	InActive	Show Proof Activate
3	Customer1	Savings	₹1900	Active	Show Proof Deactivate

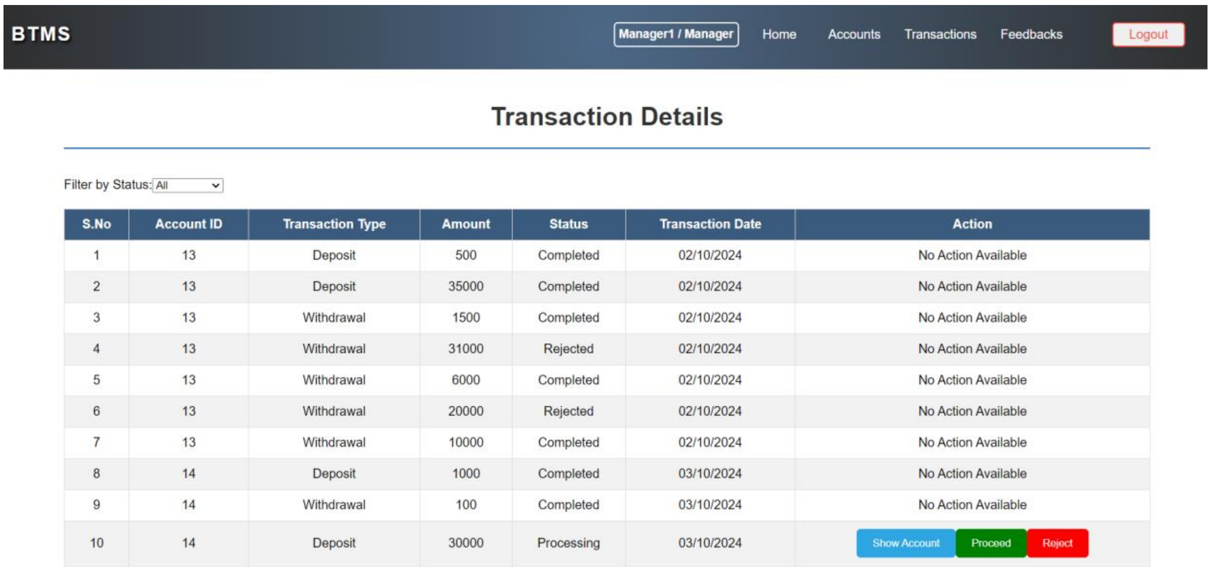
Managerviewallaccounts Components will display all the Accounts with actions Show proof, Activate, & Deactivate options for each account. If Account Status is Active then display Activate button & vice versa...

There should be Search by Account Hander name option & Filter by Status options.

On clicking the Show Proof button, should display the proof uploaded by customer in popup with close option:



On clicking the Transactions option in the managernav, will call the **managerviewalltransactions** Component.- npx ng g c components/managerviewalltransactions



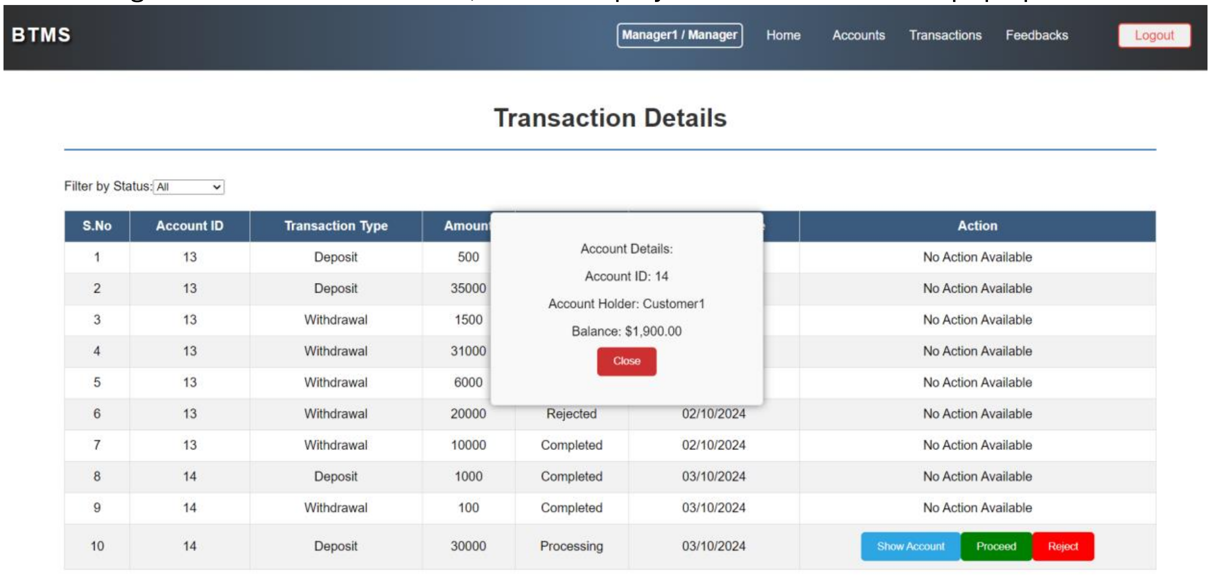
Transaction Details

Filter by Status: All

S.No	Account ID	Transaction Type	Amount	Status	Transaction Date	Action
1	13	Deposit	500	Completed	02/10/2024	No Action Available
2	13	Deposit	35000	Completed	02/10/2024	No Action Available
3	13	Withdrawal	1500	Completed	02/10/2024	No Action Available
4	13	Withdrawal	31000	Rejected	02/10/2024	No Action Available
5	13	Withdrawal	6000	Completed	02/10/2024	No Action Available
6	13	Withdrawal	20000	Rejected	02/10/2024	No Action Available
7	13	Withdrawal	10000	Completed	02/10/2024	No Action Available
8	14	Deposit	1000	Completed	03/10/2024	No Action Available
9	14	Withdrawal	100	Completed	03/10/2024	No Action Available
10	14	Deposit	30000	Processing	03/10/2024	Show Account Proceed Reject

Should display all the transactions by all the customers with options for status "Processing" – Show Account, Proceed, & Reject.

On clicking the Show Account button, should display the account details in pop up:



Transaction Details

Filter by Status: All

Account Details:

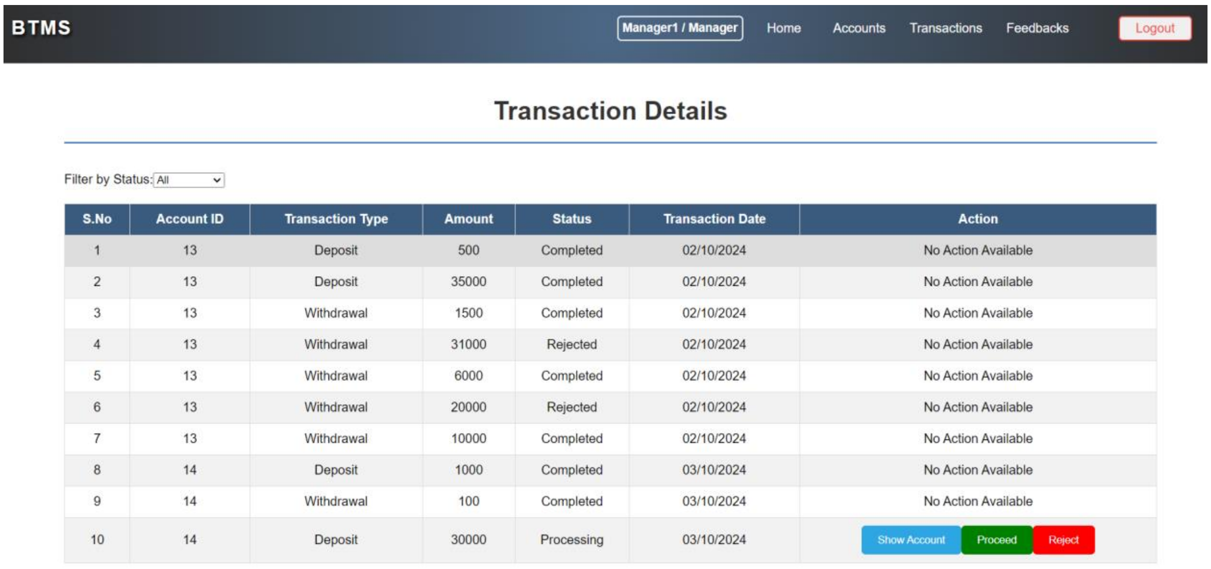
Account ID: 14

Account Holder: Customer1

Balance: $1,900.00

S.No	Account ID	Transaction Type	Amount	Status	Transaction Date	Action
1	13	Deposit	500			No Action Available
2	13	Deposit	35000			No Action Available
3	13	Withdrawal	1500			No Action Available
4	13	Withdrawal	31000			No Action Available
5	13	Withdrawal	6000			No Action Available
6	13	Withdrawal	20000	Rejected	02/10/2024	No Action Available
7	13	Withdrawal	10000	Completed	02/10/2024	No Action Available
8	14	Deposit	1000	Completed	03/10/2024	No Action Available
9	14	Withdrawal	100	Completed	03/10/2024	No Action Available
10	14	Deposit	30000	Processing	03/10/2024	Show Account Proceed Reject

On clicking the close, popup will get closed.



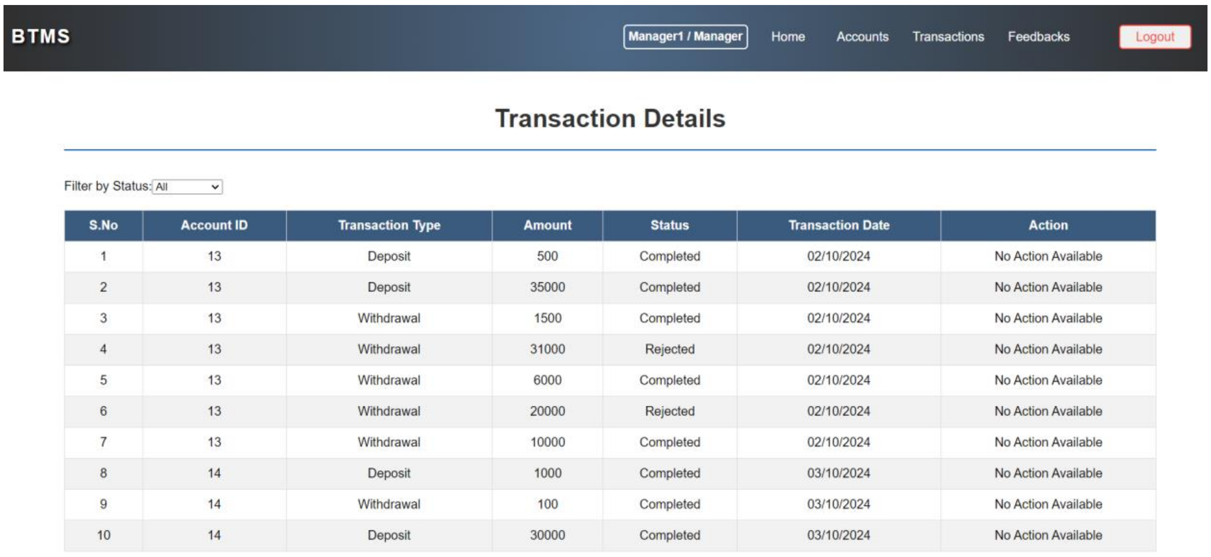
Transaction Details

Filter by Status: All

S.No	Account ID	Transaction Type	Amount	Status	Transaction Date	Action
1	13	Deposit	500	Completed	02/10/2024	No Action Available
2	13	Deposit	35000	Completed	02/10/2024	No Action Available
3	13	Withdrawal	1500	Completed	02/10/2024	No Action Available
4	13	Withdrawal	31000	Rejected	02/10/2024	No Action Available
5	13	Withdrawal	6000	Completed	02/10/2024	No Action Available
6	13	Withdrawal	20000	Rejected	02/10/2024	No Action Available
7	13	Withdrawal	10000	Completed	02/10/2024	No Action Available
8	14	Deposit	1000	Completed	03/10/2024	No Action Available
9	14	Withdrawal	100	Completed	03/10/2024	No Action Available
10	14	Deposit	30000	Processing	03/10/2024	Show Account Proceed Reject

When the "Proceed" button is clicked, the transactions should be processed, and when the "Reject" button is clicked, the transactions should be declined.

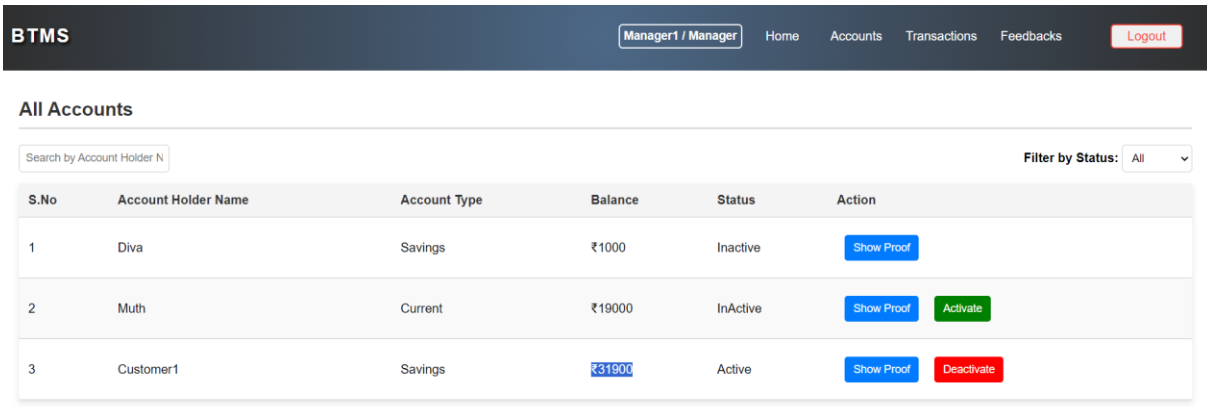
Should get reflected in account...



Transaction Details

Filter by Status: All

S.No	Account ID	Transaction Type	Amount	Status	Transaction Date	Action
1	13	Deposit	500	Completed	02/10/2024	No Action Available
2	13	Deposit	35000	Completed	02/10/2024	No Action Available
3	13	Withdrawal	1500	Completed	02/10/2024	No Action Available
4	13	Withdrawal	31000	Rejected	02/10/2024	No Action Available
5	13	Withdrawal	6000	Completed	02/10/2024	No Action Available
6	13	Withdrawal	20000	Rejected	02/10/2024	No Action Available
7	13	Withdrawal	10000	Completed	02/10/2024	No Action Available
8	14	Deposit	1000	Completed	03/10/2024	No Action Available
9	14	Withdrawal	100	Completed	03/10/2024	No Action Available
10	14	Deposit	30000	Completed	03/10/2024	No Action Available



All Accounts

Search by Account Holder N

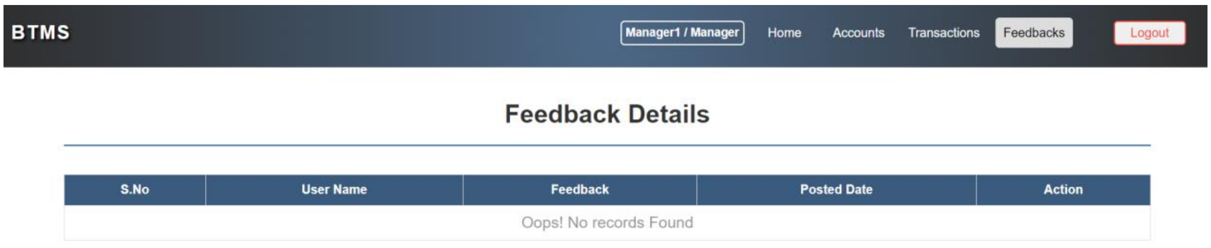
Filter by Status: All

S.No	Account Holder Name	Account Type	Balance	Status	Action
1	Diva	Savings	₹1000	Inactive	Show Proof
2	Muth	Current	₹19000	InActive	Show Proof Activate
3	Customer1	Savings	₹31900	Active	Show Proof Deactivate

Manager View Feedbacks:

Clicking on "Feedbacks" in the navbar will navigate to the **managerviewfeedback component**, which displays all feedbacks posted by all customers.

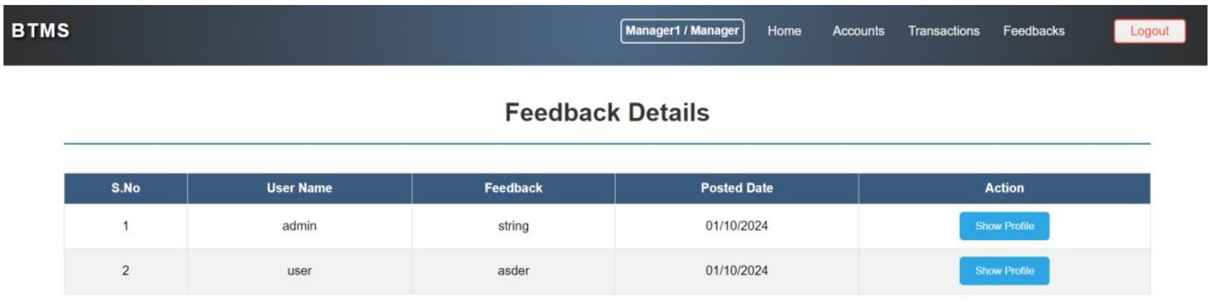
If no data is available, then "Oops! No records Found" should be displayed.



Feedback Details

S.No	User Name	Feedback	Posted Date	Action
Oops! No records Found				

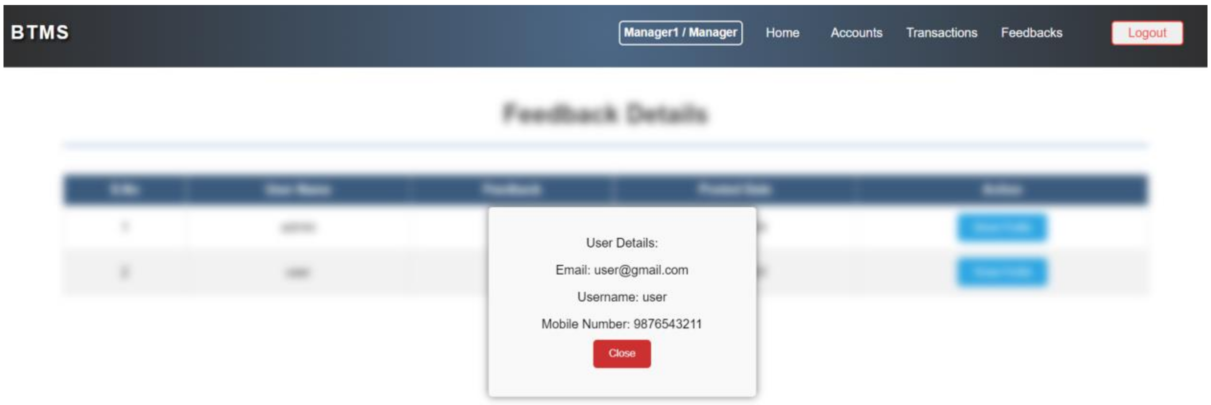
If data is available, the feedbacks posted by all users will be displayed in table format.



Feedback Details

S.No	User Name	Feedback	Posted Date	Action
1	admin	string	01/10/2024	Show Profile
2	user	asder	01/10/2024	Show Profile

Clicking on "Show Profile" will display additional details about the user in pop-up modal.



User Details:

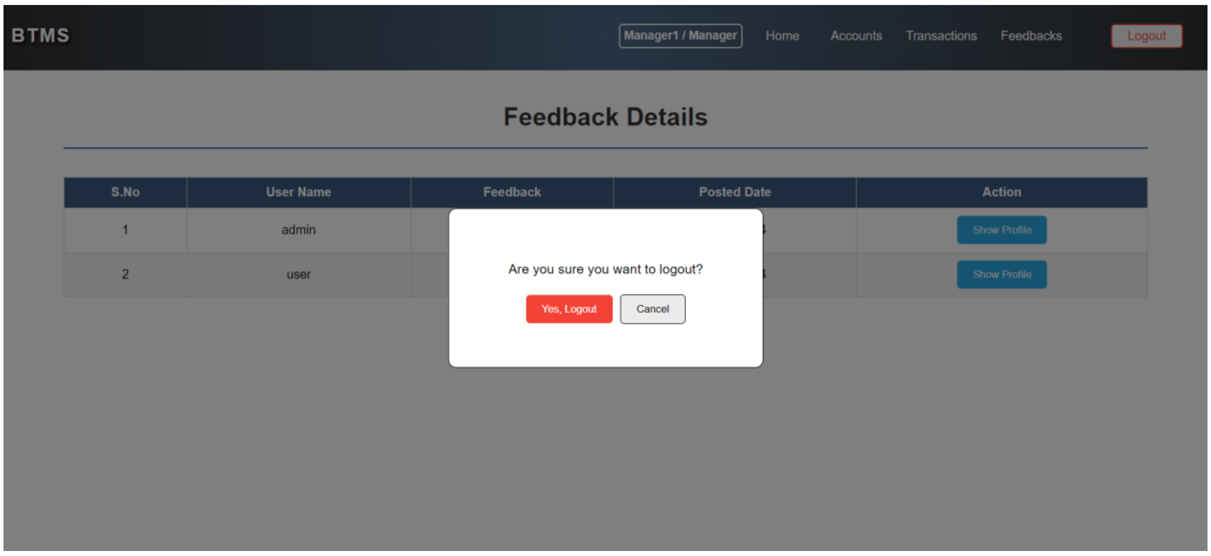
Email: user@gmail.com

Username: user

Mobile Number: 9876543211

On clicking the "Close" button, will close the pop-up modal displayed.

On clicking the "Logout" button, a pop-up should be displayed with confirmatory message to logout the user.

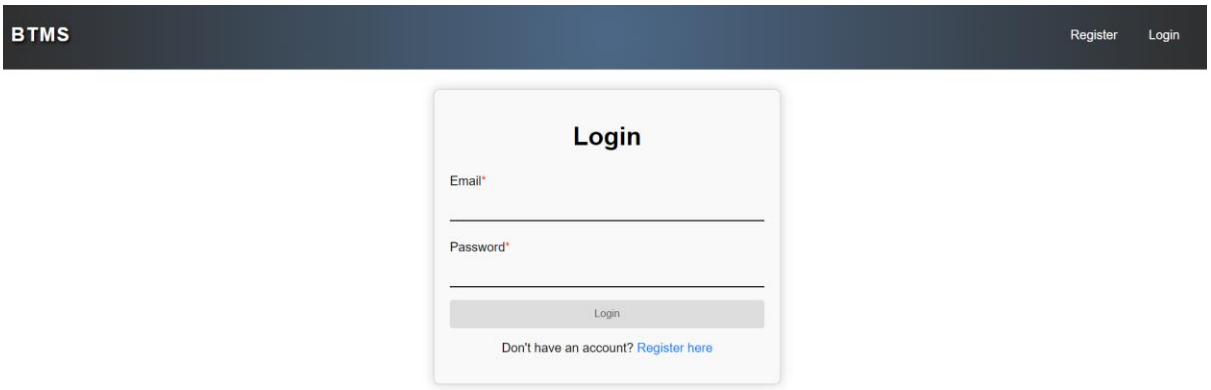


Feedback Details

S.No	User Name	Feedback	Posted Date	Action
1	admin			Show Profile
2	user			Show Profile

Are you sure you want to logout?

Clicking "Yes, Logout" will navigate to the login component.



Login

Email*

Password*

Login

Don't have an account? Register here

Platform Prerequisites (Do's and Don'ts):

1. The angular app should run in port 8081.
2. The dotnet app should run in port 8080.
3. To incorporate .Net Security into the application, use JWT authentication within the project workspace.

Key points to remember:

1. The id (for frontend) and attributes(backend) mentioned in the SRS should not be modified at any cost. Failing to do may fail test cases.
2. Remember to check the screenshots provided with the SRS.
3. Strictly adhere to the proper project scaffolding (Folder structure), coding conventions, method definitions and return types.
4. Adhere strictly to the endpoints mentioned in API endpoints section.
5. Don't delete any files in a project environment.

HOW TO RUN THE PROJECT:

BACKEND:
Open the terminal and follow the commands below.

- **cd dotnetapp**

Select the dotnet project folder

- **dotnet restore**

This command will restore all the required packages to run the application.

- **dotnet run**

To run the application in port 8080

- **dotnet build**

To build and check for errors

- **dotnet clean**

If the same error persists clean the project and build again

To work with Entity Framework Core:
Install EF using the following commands:
dotnet new tool-manifest

dotnet tool install --local dotnet-ef --version 6.0.6

dotnet dotnet-ef --To check the EF installed or not

dotnet dotnet-ef migrations add "InitialSetup" --command to setup the initial creation of tables mentioned in DBContext

dotnet dotnet-ef database update --command to update the database

To Work with SQLServer:
(Open a New Terminal) type the below commands
sqlcmd -U sa
password: **examlyMssql@123**

```
>use DBName
>go
```

```
1> insert into TableName values(" "," ",...)
2> go
```

Note:

1. Please ensure that the application is running on port 8080 before clicking the "Run Test Case" button.
2. Database Name should be **appdb**
3. **Use the below sample connection string to connect the Ms SQL Server**

connectionString = "User ID=sa;password=examlyMssql@123; server=localhost;Database=appdb;trusted_connection=false;Persist Security Info=False;Encrypt=False";

FRONTEND:

Open the terminal and follow the commands below.
Step 1:
• **Use "cd angularapp**" command to go inside the angularapp folder
• Install Node Modules - "**npm install**"

Step 2:
• Write the code inside src/app folder
• Create the necessary components
• To create Service: "**npx ng g s services/<service name>**"
• To create Component: "**npx ng g c components/<component name>**"

Step 3:
• Click the **Run Test Case** button to run the test cases

Note:
• Click PORT **8081** to view the result / output.
• If any error persists while running the app, delete the node modules and reinstall them.