**Online Banking Application**

Introduction

The project aims to develop an Online Banking Application entirely from Java.

I have used Spring Boot to create the backend of the system and HTML to create basic Submission Forms.

System Requirements

* Java 17
* MySQL

**Note:** Connect the MySQL locally to this project and change the credentials present in ***src/main/resources/application.yml*** to connect this application to your database.

Running The Project

* Create a local MySQL database and connect it to the application with the necessary credentials.
* Run the OnlineBankingApplication.java from ***src/main/java/com/example/onlinebankingapplication/OnlineBankingApplication.java***
* While accessing the Api URLs make sure to replace {iD} with the id of the customer. Eg:

[http://localhost:8080/customer/{iD}/account/transaction](http://localhost:8080/customer/%7biD%7d/account/transaction)

should be used as

<http://localhost:8080/customer/1/account/transaction>

if you want to make a transaction for an account connected

with Customer Id 1.

Entity Relationship Diagram

A screenshot of a computer

Description automatically generated

Customer Entity:

* Contains the basic details of the customer- First name, Last Name & Email Address
* It is the parent entity and all the other entities depends on it.
* It contains **customerId** of Long type as a Primary key.

Account Entity:

* Contains the account details of the customer.
* It has a One-to-One relationship with the **Customer** Entity, implying that a customer can open only one account under his credentials.
* This can be modified to have a Many-to-One relationship if we want to increase the functionality of the application.
* It has **accountId** of Long type as Primary key.
* It contains **customerId** of Long type as Foreign Key.
* Account Entity has a bidirectional One to Many relationship with **Transaction** Entity.

Transaction Entity:

* Contains the transaction details made by the account.
* It has a Many to One Relationship with the **Account** Entity, implying that an account can have many transactions under it.
* It has **transactionId** of Long type as Primary key.
* It has **accountId** of Long type as Foreign Key.
* The **runningBalance** attribute shows the updated balance of the account after the transaction has been made.

Contact Entity:

* Contains the contact details of the customer.
* It has a One-to-One relationship with the **Customer** Entity.
* It contains **contactId** of Long type as a Primary key.
* It contains **customerId** of Long type as a Foreign key.

Address Entity:

* Contains the Address details of the customer.
* It has a One-to-One relationship with the **Customer** Entity.
* It contains **addressId** of Long type as a Primary key.
* It contains **customerId** of Long type as a Foreign key.

REST APIs

* Save Customer API

Api URL: <http://localhost:8080/save-customer>

This Api saves the customer into the database. Situated in the Customer Controller the **saveCustomerForm()** handles the GET request to the /save-customer endpoint which returns a view which renders the HTML content. The view that is returned by the method is **CustomerForm.html** which parses this template and carry out various template expressions to render the form. This form takes in customer details.

A screenshot of a computer

Description automatically generated

Fig 1: Customer Details Form

After hitting submit we are then redirected to the same /save-customer endpoint but with a POST request. The **saveCustomer()** method then receives the **Customer** object populated by the previous form and saves that customer in the database.

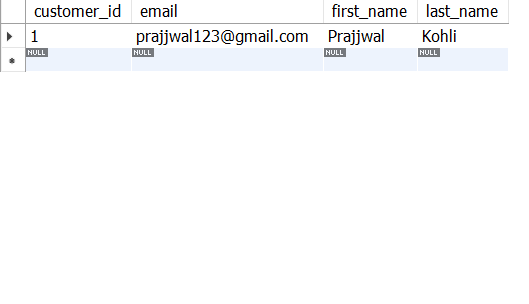


Fig 2: Customer Table

* Create Account API

Api URL: [http://localhost:8080/customer/{iD}/create-account](http://localhost:8080/customer/%7biD%7d/create-account)

This Api is handled by the Account Controller. The **createAccountForm()** handles the GET request to /create-account endpoint which return a view named AccountForm.html view which is rendered by the Thymeleaf same as **CustomerForm**. This form also takes in the address and the contact information along with account details.

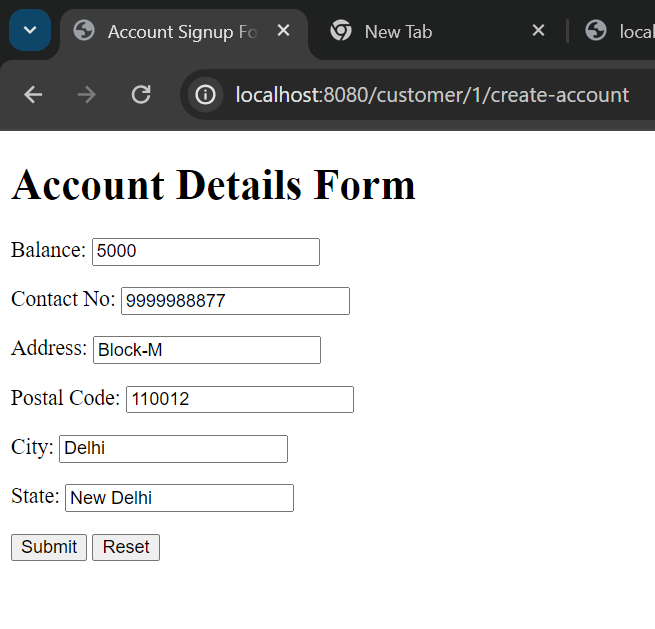


Fig 3: Account Details Form

After hitting submit we get redirected to the same /create-account endpoint with a POST request. The **createAccount()** which handles the POST calls to the /create-account endpoint receives objects of **Account, Contact, Address.**

**Contact** & **Address** which were populated with the previous form are then saved into the database and simultaneously linked with customer ID present in Api URL.

The **createDate** & **accountNo** fieds are automatically created with **setCurrentDate()** and **setRandomAccountNo()** methods in the **AccountServiceImpl** class. Finally, the **Account** is stored into the database with and is linked with the respective customer ID.

A screenshot of a computer

Description automatically generated

Fig 4: Contact Table

A screenshot of a computer

Description automatically generated

Fig 5: Address Table

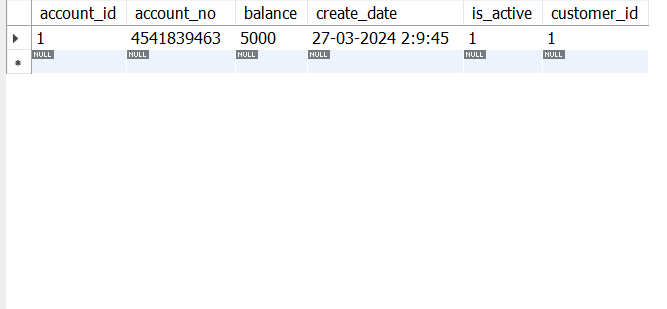


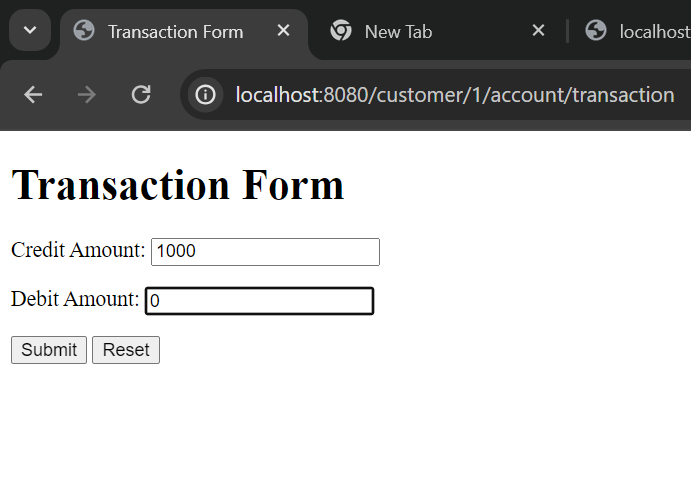
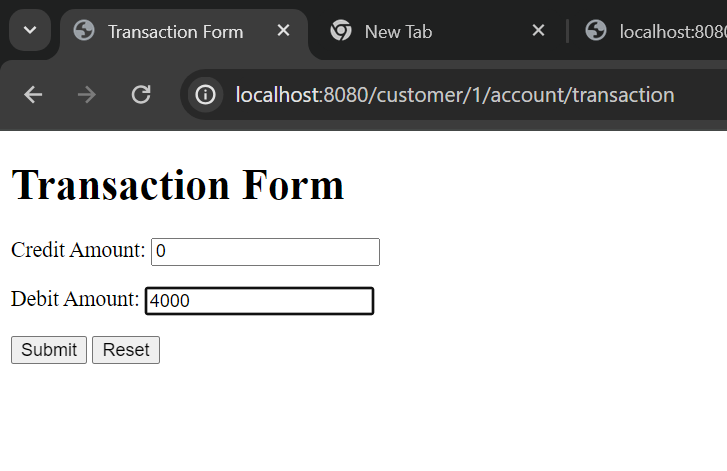
Fig 6: Account Table

* Transaction API

Api URL: [http://localhost:8080/customer/{iD}/account/transaction](http://localhost:8080/customer/%7biD%7d/account/transaction)

This Api helps in creating a transaction whether its debit or credit. It then modifies the balance of the account according to the transaction done and records the time of the transaction.

The GET request to /transaction endpoint is handled by the **createTransactionForm()** which returns a view of **TransactionForm.html**. After submitting the form we are then redirected to the same endpoint with a POST request.

a) b)

Fig 7: Two transactions are done-

a) Amount of ₹1000 is credited into the account

b) Amount of ₹4000 is debited from the account

This POST request is handled by **createTransaction()** method which receives the populated **Transaction** object. Now this Transaction object is saved into the database after linking it with the respective account.

A screenshot of a computer

Description automatically generated

Fig 8: Both transactions are stored in the Transaction table.

* Show Transaction History API

Api URL: [http://localhost:8080/customer/{iD}/account/transaction-history](http://localhost:8080/customer/%7biD%7d/account/transaction-history)

This Api shows the transaction history of the account. Its mechanism is same as the Check Balance API, the only difference is that at the end it invokes **getTransactionList()** to the found account and returns it to the user in the JSON format.

A screenshot of a computer

Description automatically generated

Fig 9: JSON response of /transaction-history endpoint

* Check Balance API

Api URL: [http://localhost:8080/customer/{iD}/account/balance](http://localhost:8080/customer/%7biD%7d/account/balance)

This Api is handled by the **checkBalance()** method in Account Controller. The method searched the Customer table using the “iD” sent as the path variable in the Api URL. We then invoke **findByCustomer()** method present in **AccountRepository.** This method returns the **Account** object for a given **Customer** or returns *null* if no account exists. We then invoke **getBalance()** to this found account and returns it to the user in the JSON format.

A screenshot of a computer

Description automatically generated

Fig 10: JSON response of /balance endpoint