Easy Banking

Gayatri Dalvi, Pratiksha Patil, Pratik Thorat

Information Systems, Northeastern University

{dalvi.ga,patil.pratik,thorat.p}@northeastern.edu

***Abstract*—This banking application is designed to provide customers with a secure and user-friendly way to manage their finances online. Built using Java, JavaFX, and SQLite3, the application allows users to log in to their account, view their account balance and transaction history, and perform a variety of banking transactions.**

**One of the key features of the application is the ability for users to deposit money, transfer money within savings and checking accounts, and transfer money to other accounts. The application also provides an admin user with the ability to create new users and manage their account settings, providing additional flexibility and control for businesses or financial institutions.**

**The use of JavaFX allows for a modern and customizable user interface, with support for modern design principles such as Material Design. The use of SQLite3 ensures that customer data is stored securely and efficiently, with strong encryption and other security measures in place to protect against unauthorized access.**

**To ensure the security and confidentiality of customer data, the application also includes a variety of other security features. These include multi-factor authentication, encryption, and fraud detection systems, which work together to prevent unauthorized access and protect against potential threats.**

**Overall, this banking application provides a seamless and hassle-free way for customers to manage their finances online. Its intuitive user interface, strong security measures, and support for a variety of banking transactions make it an ideal choice for businesses or financial institutions looking to provide their customers with a modern and secure online banking experience.**

***Keywords—Model,Controller,View,Account,Transaction,DatabaseDriver,Dashboard,MVC Pattern, Data Integrity, Validation***

# **I. Problem Description**

Complex banking applications can cause a lot of problems for users, leading to frustration and confusion. These applications often have a cluttered and confusing interface, making it cumbersome for users to navigate and understand how to use them. This can cause a lot of stress for users, particularly those who are not technologically aware or have limited knowledge of finance.

Furthermore, complex banking applications can lead to fallacies in managing finances. Users may make mistakes in entering data or interpreting financial information due to the complicated nature of the application. This can result in incorrect financial decisions, missed payments, and overdraft fees thus causing financial stress to our users.

Moreover, such applications can be time-consuming and challenging to use. Users may need to spend hours learning how to use the application, resulting in a significant time investment that they may not be able to afford thus making the purpose of the application counterintuitive. This can be particularly problematic for busy individuals who have limited time to manage their finances.

In addition, it will be intimidating and discourage users from seeking financial advice or support. Users may feel overwhelmed by the complexity of the application and be less likely to seek help from financial advisors or customer service representatives. This can lead to users making poor financial decisions or missing out on valuable opportunities for financial growth.

In summary, complex banking applications can cause a lot of problems for users, including confusion, errors, time-consuming processes, and discouragement from seeking financial support. Therefore, there is a need for simplified banking applications that is easy to use and can help with budget planning and finances in general.

# **II. Analysis (Related Work)**

In this section we give a short analysis of previous work that we have come across.

Firstly, our system is built using modern technologies such as Java, JavaFX, and SQLite3, which offer robust features for developing secure and user-friendly applications. The use of JavaFX also provides a modern and customizable user interface, making it more intuitive for customers to use the system.

Secondly, our system includes a range of security features such as multi-factor authentication, encryption, and fraud detection systems, which ensure that customer data is protected against unauthorized access and potential threats. These security features are crucial in the current landscape, where cyberattacks and fraud are increasingly common.

Thirdly, our system offers a range of features for managing different types of bank accounts, including savings and checking accounts, as well as the ability to transfer funds to other accounts. The system also includes an admin user with the ability to create and manage user accounts, adding an extra layer of flexibility and control.

Finally, our system is designed to be highly customizable and scalable, allowing businesses and financial institutions to tailor the system to their specific needs and easily add new features as needed. This level of flexibility and scalability is essential for businesses that want to remain competitive and adapt to changing customer needs.

Overall, our system stands out from previous bank management systems due to its modern technologies, strong security features, range of features for managing different types of accounts, and flexibility and scalability. These factors make our system an ideal choice for businesses and financial institutions looking to provide their customers with a secure, intuitive, and hassle-free online banking experience

# **III. System Design**

Easy banking follows the Model-View-Controller (MVC) architecture, which separates the application logic into three main components.

The Model is responsible for handling tasks related to data management, such as retrieving account information from the database, processing transactions, and updating account balances. It acts as a bridge between the data and the application, ensuring that all data-related operations are performed efficiently and accurately.

The View component is responsible for displaying data to the user in a user-friendly and understandable way. It presents the account balances, transaction histories, and provides forms for making deposits, transfers, and other transactions. The View ensures that the user can interact with the application easily and efficiently.

The Controller is responsible for handling user input and processing user requests. It validates user input, updates account balances based on the user's requests, and displays transaction confirmations to the user. It acts as an intermediary between the View and the Model, ensuring that user input is processed correctly and that data is updated accurately.

Overall, the MVC architecture used in Easy Banking provides a well-structured and organized approach to application development. It separates concerns and allows for efficient development, testing, and maintenance of the application.

In order to achieve data persistence in Easy Banking, we have integrated SQLite3 database into the application. This allows the application to store user data and transaction histories in a reliable and efficient way.

The main file of the application, App.js, is responsible for invoking the Model and the View instances. It acts as the entry point for the application and coordinates the interactions between the Model and the View.

The "resource" folder contains all of the UI elements that have been built using SceneBuilder and are placed in the "FXML" folder. These UI elements include forms for making deposits and transfers, transaction histories, and other user interfaces. The FXML files are responsible for defining the layout and design of these elements.

The "css" folder contains all of the styling sheets used in the application. These sheets define the colors, fonts, and other visual elements of the UI.

Finally, the "animations" folder contains all of the images used in the application. These images include icons, logos, and other visual elements that are used to enhance the user experience.

Overall, the integration of SQLite3 database and the use of these folders and files in Easy Banking helps to create a reliable, efficient, and user-friendly application for managing bank accounts and transactions.

One by one explanation of MVC architecture:

Controller-

The controller will have different roles for login functionality, such as admin and client. After the user logs in, the login controller will validate the user's credentials and show the login window based on their role.

Admin Controller-

The admin controller is the main controller that offers various options in the top menu. The admin options controller defines the on-click events for these options. The client add controller adds clients and creates their accounts. The client cell controllers display client details, such as first name, last name, and account number, among other parameters. The client list controller lists all clients through different client cells.

Client Controller-

The client controller is the main controller for the client and offers options such as dashboard, pay, transaction, and account. These options are defined in the client option controller along with their respective on-click events. The dashboard controller shows a summary of the user's account activities, including account balance and transaction history. The pay money controller facilitates transferring money from one account to another, including options for username, amount, and message. The account controller displays account details and allows users to transfer money between accounts, with balances updating in both the dashboard and account sections. The transaction cell controller displays transactions done through the pay money controller, with details such as username, amount, date, first name, last name, and sent and received alerts. The transaction controller lists all transactions with different transaction cells.

In summary, the easy banking application uses different controllers to manage the user interface and interaction. The controllers have different roles and functionalities, such as validating user input, processing transactions, displaying account balances and transaction histories, and updating account balances. The application uses SQLite3 database to ensure data persistence. The different controllers and their functionalities are defined in separate files and folders, such as App.js, resource folder with UI built in scenebuilder, FXML folder, css folder for styling sheets, and animations folder for images.

Model -

In our model, we have a parent class called "Person." We extended this class to create two subclasses: "Client" and "Admin," with private parameters for data encapsulation.

The next parent class is "Account," which includes parameters such as account number, balance, and username. This class is further extended by "SavingsAccount" and "CheckingAccount" subclasses.

Model class –

The Model class in our project consists of three main components: Model, View, and DatabaseDriver. Within the Model class, there are several variables and functions that are used to manage the data and the flow of the program.

ObservableList: We have used ObservableList for tracking the latest transaction and all transactions. This allows us to easily add and remove elements from the list as transactions occur.

clientflagset and adminflagset: These variables are used to set the flag false initially and then set it to true after successful validation in the admin and client. This allows us to keep track of the user's role and determine what actions they are allowed to perform.

Validation function: This function is used to validate the user's login credentials. It takes the username and password as input and returns a boolean value indicating whether the validation was successful or not.

prepareTransaction: This function prepares a list of transactions by fetching the transaction details from the database for a given client and transaction limit. It retrieves the sender, receiver, amount, date, and message of each transaction from the ResultSet object and creates a new Transaction object for each transaction with the retrieved details. Finally, it adds each Transaction object to the provided ObservableList of transactions.

setClients: This function is a method of the Model class, used to set the list of clients in the model by fetching client data from the database. It first initializes CheckingAcc and SavingAcc objects, then retrieves data from the database using the databaseDriver class, and finally creates a new Client object for each row of client data. It then adds each new Client object to an ObservableList of clients.

getCheckingAccount and getSavingAccount: These functions are used to get the checking and savings accounts for a specific client. They take the client's username as input and return the checking or savings account associated with that username.

ObservableList<Client> searchUser: This function is used to search for a client by their username. It takes the search term as input and returns an ObservableList of clients that match the search term.

In addition to these functions, we also have a static method called getInstance() that is used to check if the model is null or not. This method is synchronized and can be shared by all objects.

Views-

We implemented enums for selecting between client and admin roles. The AdminWindowEnum and ClientWindowEnum classes are used to define the options for each respective role. We also updated the ClientCellFactory and TransactionCellFactory classes to update cells after new clients or transactions are added.

In the view class, we created functions for every view and used anchorpanes to display them upon user click. The view class also includes the login and logout functions.

Pom.xml-all dependencies

The Pom.xml file contains all the necessary dependencies for the project to run smoothly. It includes the database driver class which is responsible for managing the data.

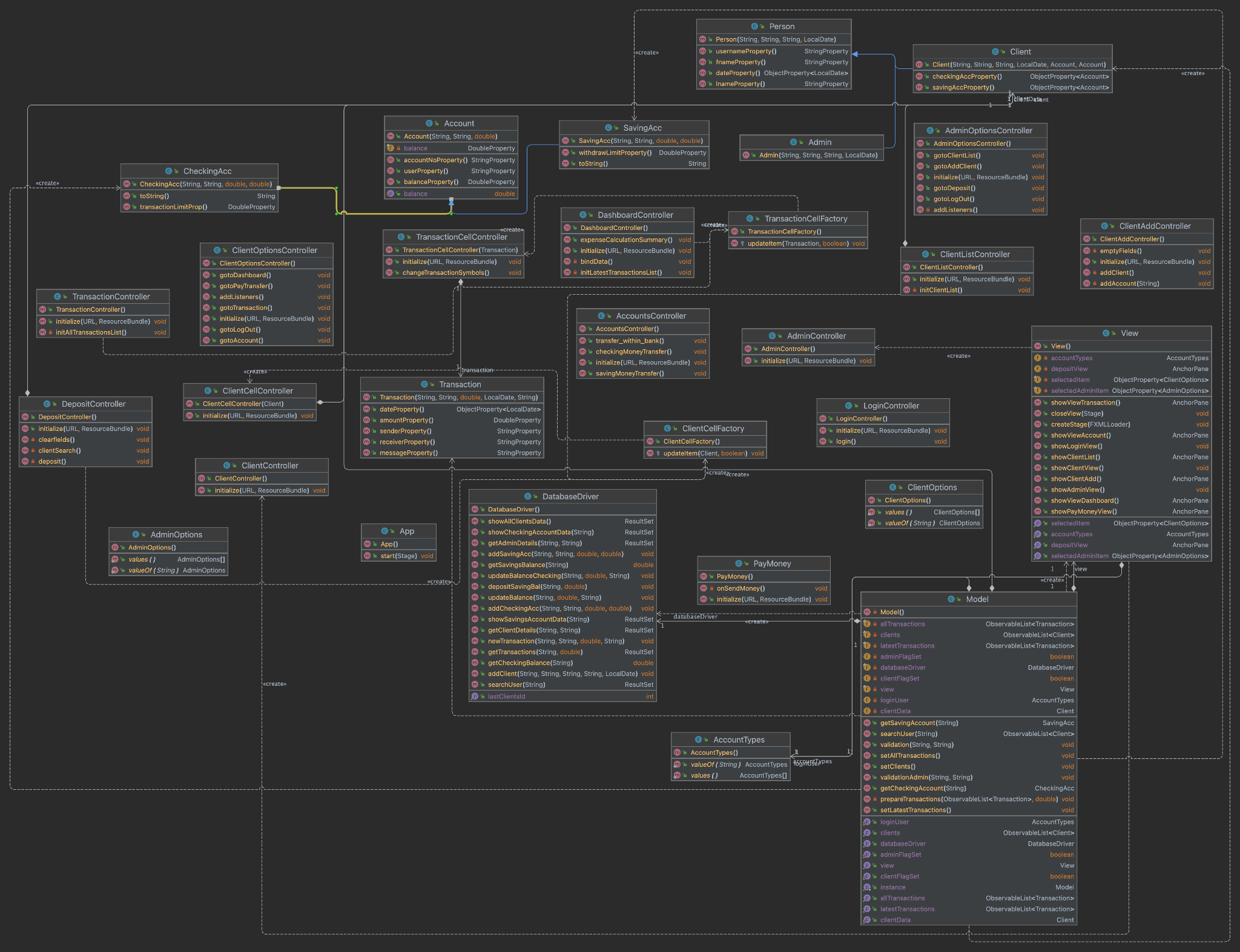


Figure 1. UML

# **IV. Implementation**

A. Inheritance: Inheritance is used to establish relationships between classes where a child class inherits properties and methods from a parent class. In this project, the class "Person" is the parent class, which has attributes such as first name, last name, and username. The child classes, "Admin" and "Client," are created by extending the "Person" class and adding the properties required for their specific roles. Similarly, the "Account" class is the parent class with attributes such as user, account, and balance, which is extended by "Savings" and "Checking" account classes. The "TransactionCellFactory" also extends the class, and inheritance is achieved through the use of "extends" keyword.

B. Encapsulation: Encapsulation is used to protect the data in a class by making the variables private and providing access to them through getters and setters methods. In this project, every variable is defined as private, and their access is given through getters and setters. By doing this, we ensure the data is protected from unauthorized access and maintain the integrity of the class.

C. Generics: Generics are used to provide compile-time type safety for a collection of objects. In this project, generics are used for the "Transaction" and "Client" list to ensure they contain the same type of elements, which increases the code's readability and makes it easier to maintain.

D. Parameterized constructors: Parameterized constructors are used to initialize the properties of an object at the time of creation. In this project, every child class has a parameterized constructor to initialize the properties of the superclass.

E. Enum classes: Enum classes are used to define a set of predefined constants. In this project, enum classes are used to define the roles of the "Admin" and "Client" classes and the top menu options of the client and admin roles. This ensures that the values of these properties remain unchanged throughout the program.

F. Interfaces: Interfaces are used to define a set of methods that must be implemented by the implementing class. In this project, every controller implements the "Initializable" interface to initialize the values of its member components. The "initialize" method is used to initialize the component along with its "onClick()" event.

G. Exception handling: Exception handling is used to handle errors and prevent program crashes. In this project, try-catch blocks are used to handle exceptions. For example, in the "View" class, if the "dashboard" anchor pane is null, the program will catch the exception and load the FXML file on the location, preventing the program from crashing.

public AnchorPane showViewDashboard(){

if(dashboard==null){

try{

dashboard=new FXMLLoader(getClass().getResource("/Fxml/Client/Dashboard.fxml")).load();

}catch(Exception e){

e.printStackTrace();

}

}

return dashboard;

}

H.Database

The application uses an SQLite database to store all the data related to the users and their accounts. The database consists of five tables, namely Admin, Client, SavingAcc, CheckingAcc, and Transaction.

The Admin table stores the login credentials for the administrators, while the Client table stores all the user information along with their login credentials. When a new user is created by an admin, the relevant data is inserted into the Client table. If the user has a checking account, the relevant data is also inserted into the CheckingAcc table. Similarly, if the user has a savings account, the relevant data is inserted into the SavingAcc table.

After any transaction is made by a user, the Transaction table is updated with the relevant information. All these operations are taken care of by the DatabaseDriver class.

I. List:

In the project, Listview is used to display transaction list and client list. A list is a collection of elements of the same data type, and it can be used to store and manipulate data efficiently. The ListView is a graphical user interface element that displays a list of items in a scrollable vertical list. It is used to display transaction history and client details in our project. By using the ListView, we can provide an interactive and user-friendly interface to the users for viewing the required information.

J. Final keyword:

In the project, some variables like username, accountNo, and balance are declared with the final keyword. This is done to make sure that their values cannot be changed once initialized. By using the final keyword, we make these variables immutable and ensure that their values remain constant throughout the program.

K. Static method:

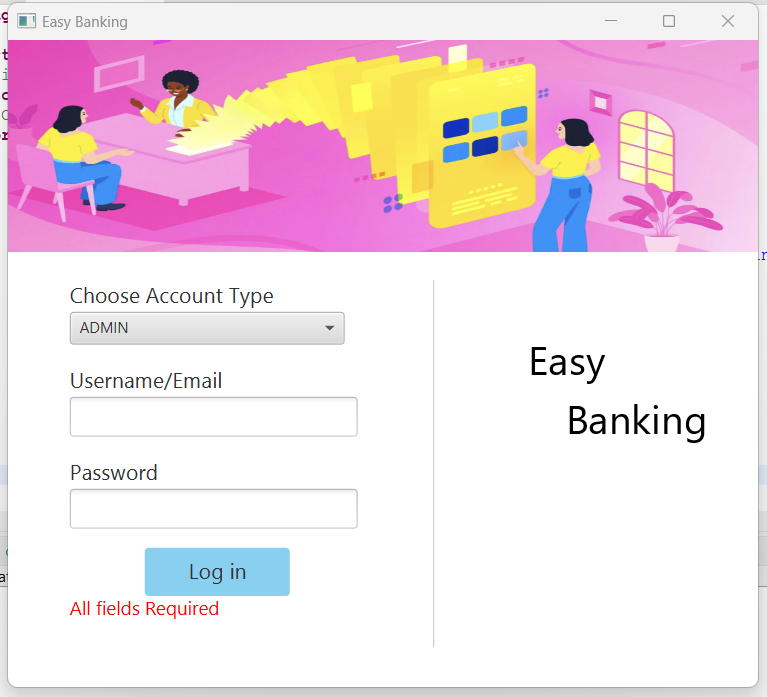
In the project, a static method is used to create a singleton instance of the Model class. The method signature is defined as public static synchronized Model getInstance(). Here, the method is defined as synchronized to avoid race conditions that may occur when multiple threads try to access this method simultaneously. The getInstance() method checks if the instance of the Model class is already created or not. If it is already created, it returns the existing instance; otherwise, it creates a new instance of the Model class and returns it. By using this method, we ensure that there is only one instance of the Model class throughout the application, and this instance can be accessed by all objects without creating new instances.

External Package: Used java.sql package to connect to the database. This package provides classes and interfaces for accessing and manipulating data stored in relational databases using the Java Database Connectivity (JDBC) API.

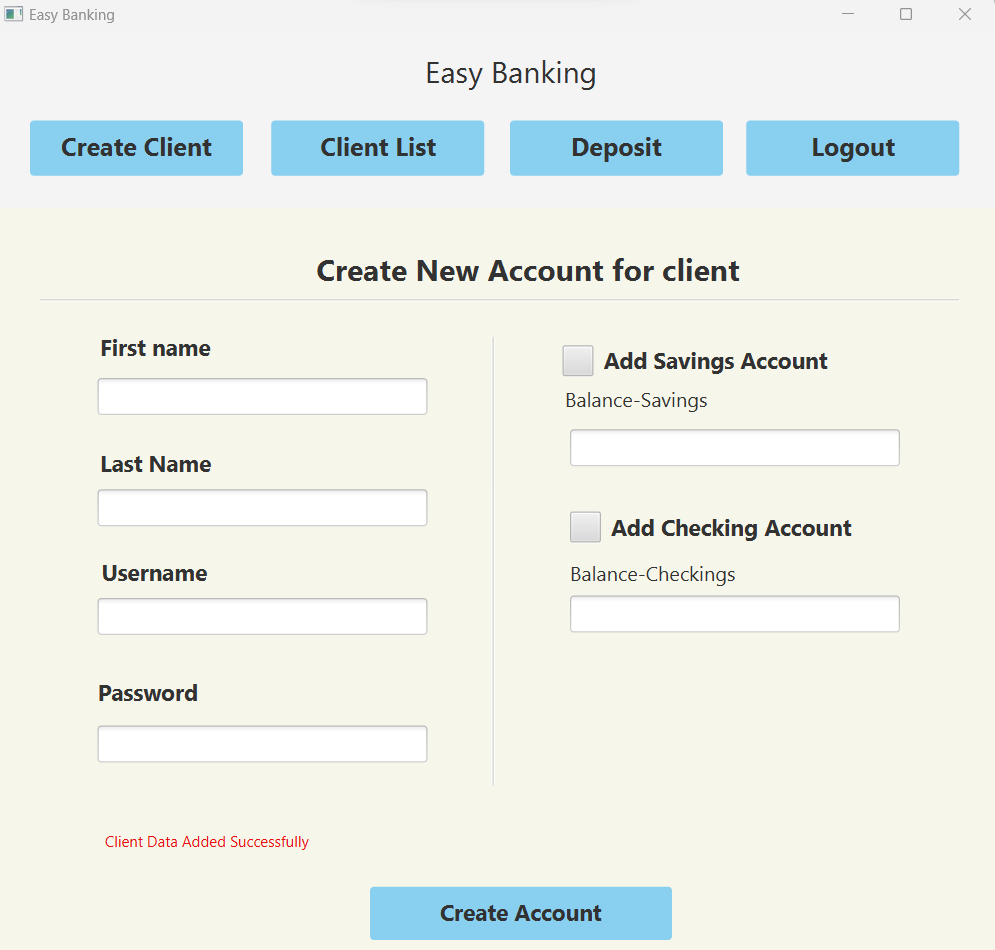
# **V. Evaluation**

The screenshots of sample run and the explanation

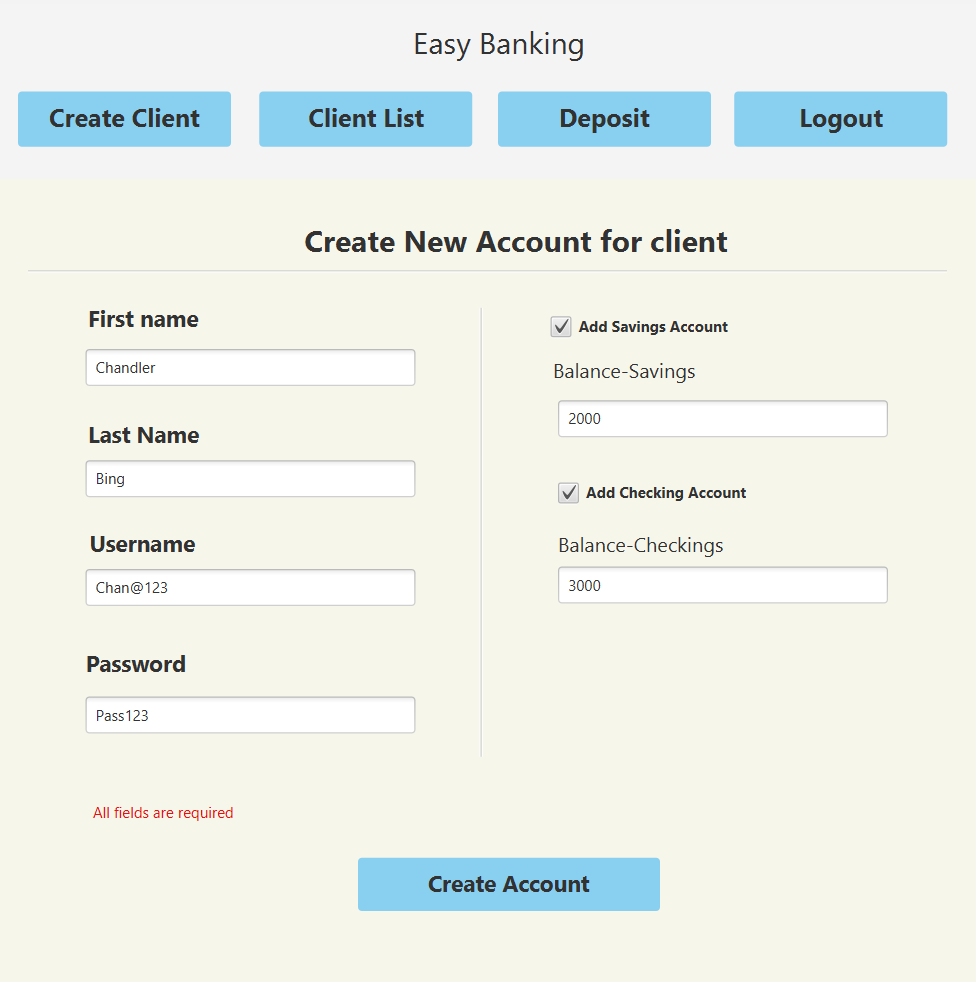
1) Login Page - Data validation for Username and Password. It shows User not found if we don't have user credentials in Database.



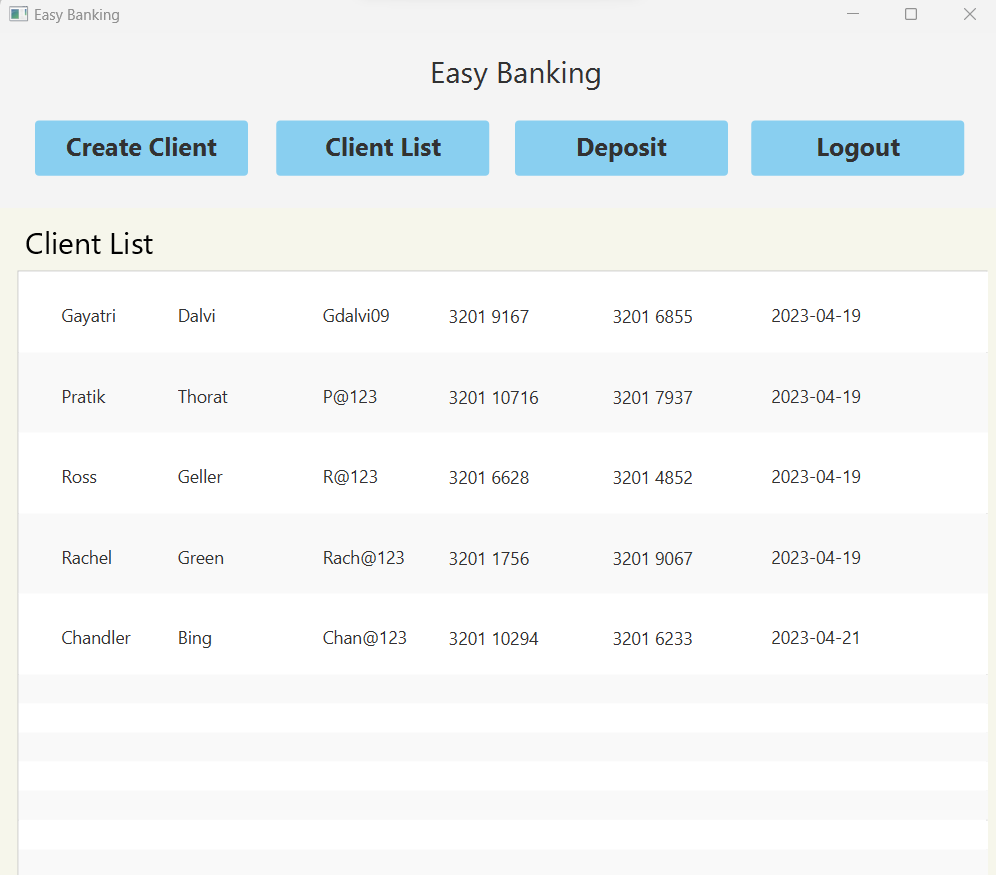
2) Create Client - Upon logging in, the initial page presented is for creating a client. Only the administrator is authorized to create new users.



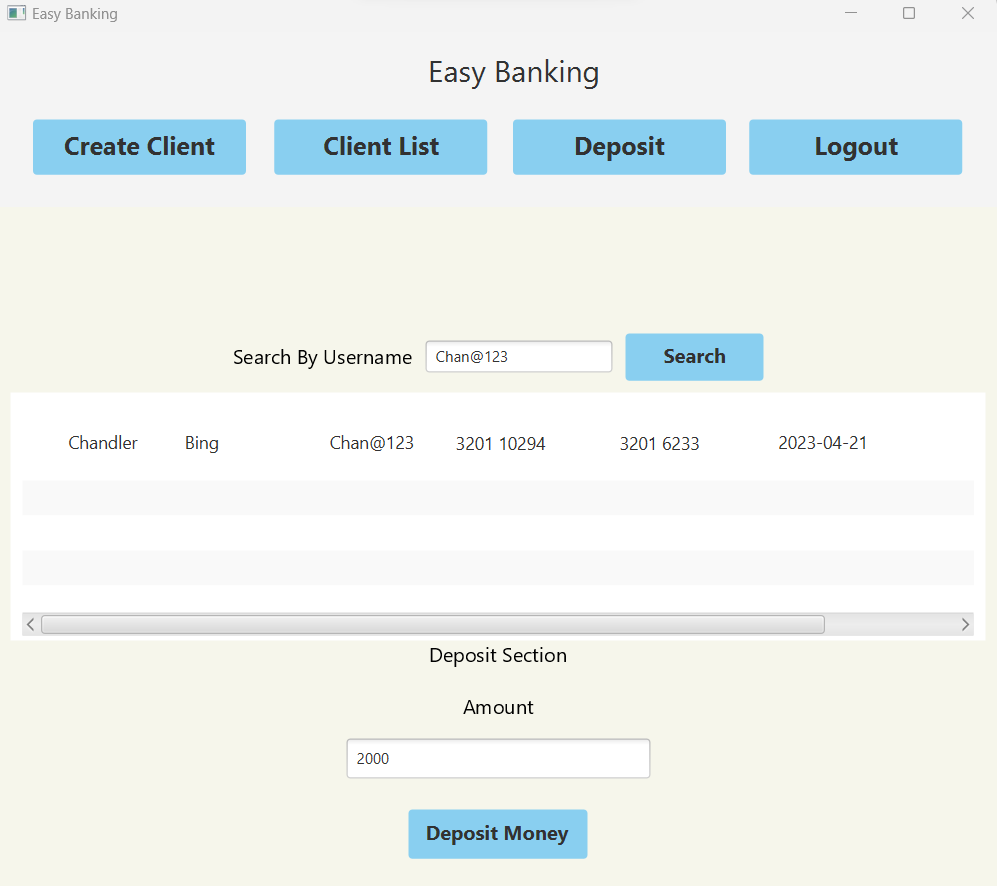
3) Data validation is implemented for each field, and once the admin adds and saves user account details, the values are stored in a specific table(Create Operation).



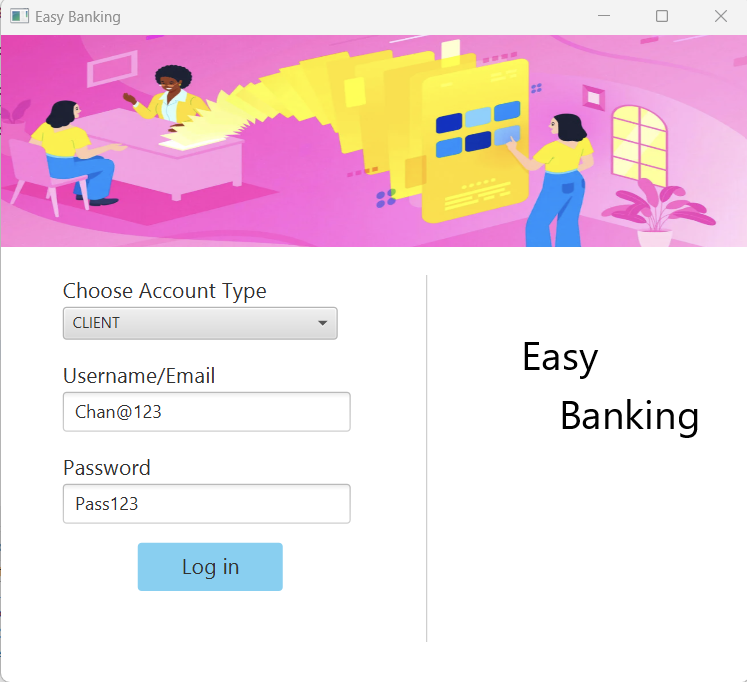
4) After client details are saved you can see it in Client List showing View Operation



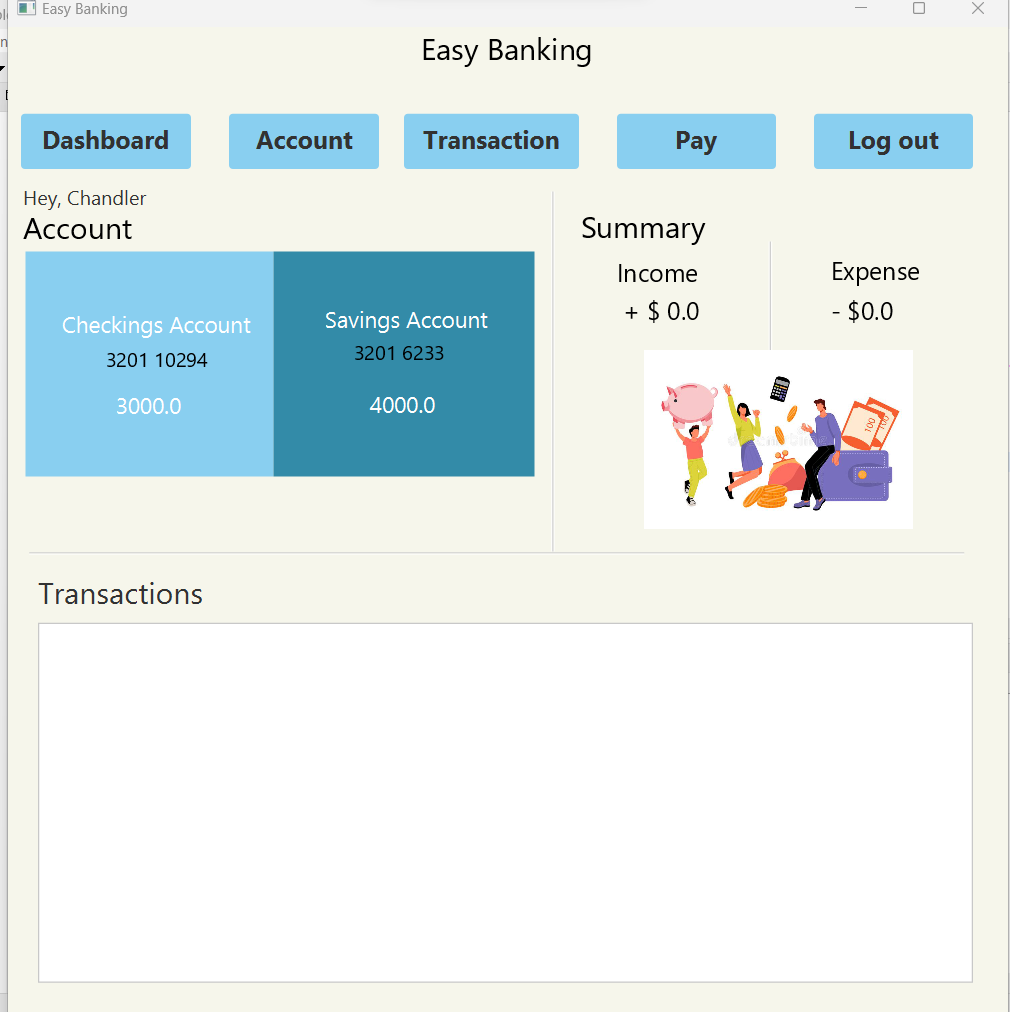
5) Deposit Section - In the Deposit section, the admin can perform a search operation to find the desired user and deposit an amount into their account.



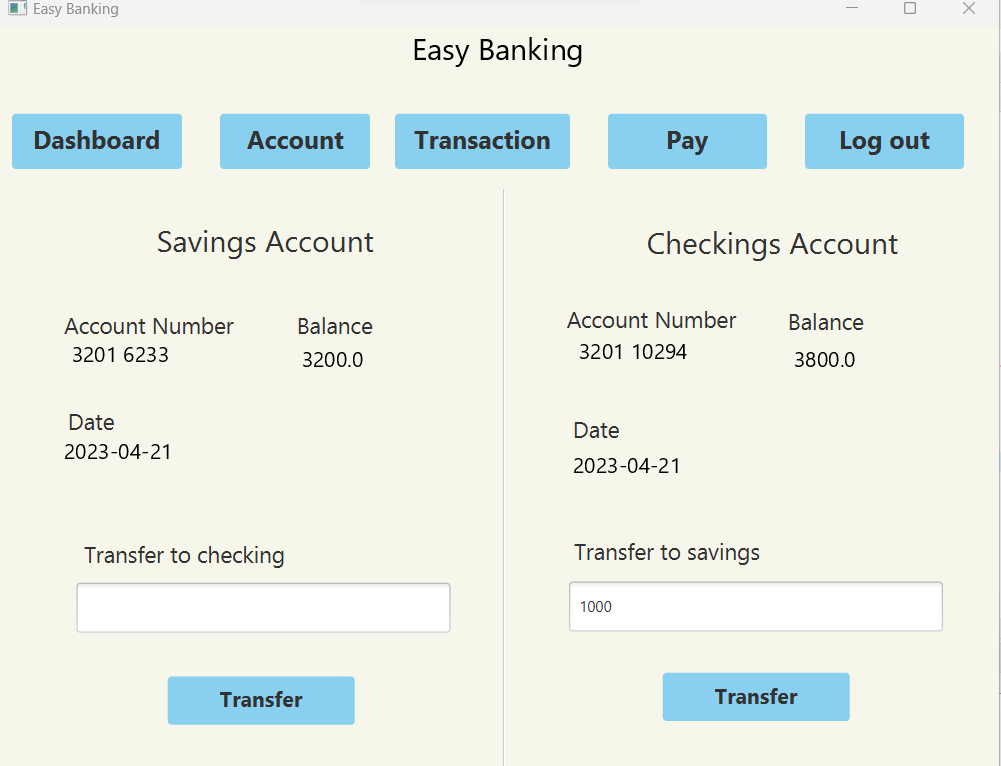
6) To add the client's username and password, you need to select the client from the dropdown menu and then enter the relevant credentials.



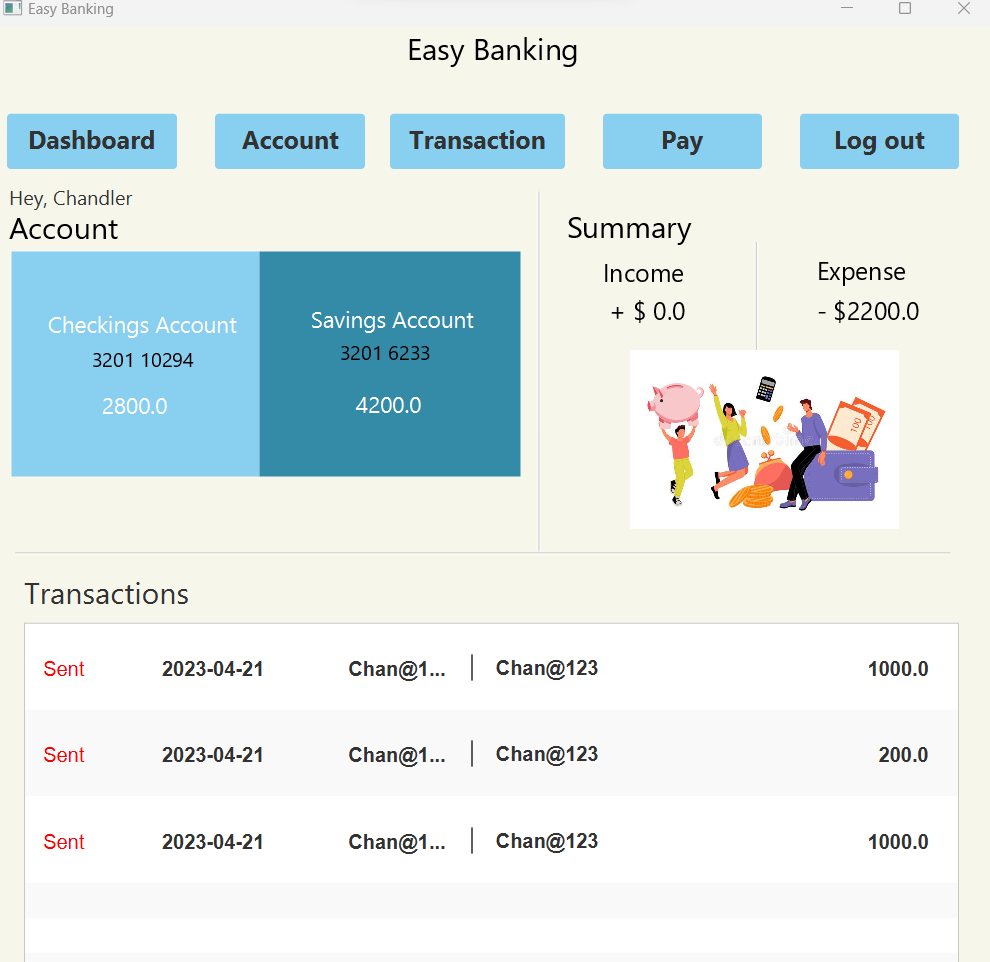
7) The client dashboard consists of three sections: one displays the username of the currently logged-in user, another shows the savings and checking account information along with their details, and the third section includes a summary of income and expenses. Additionally, there is a Transaction section that presents the latest transactions,this section shows Select Operation.



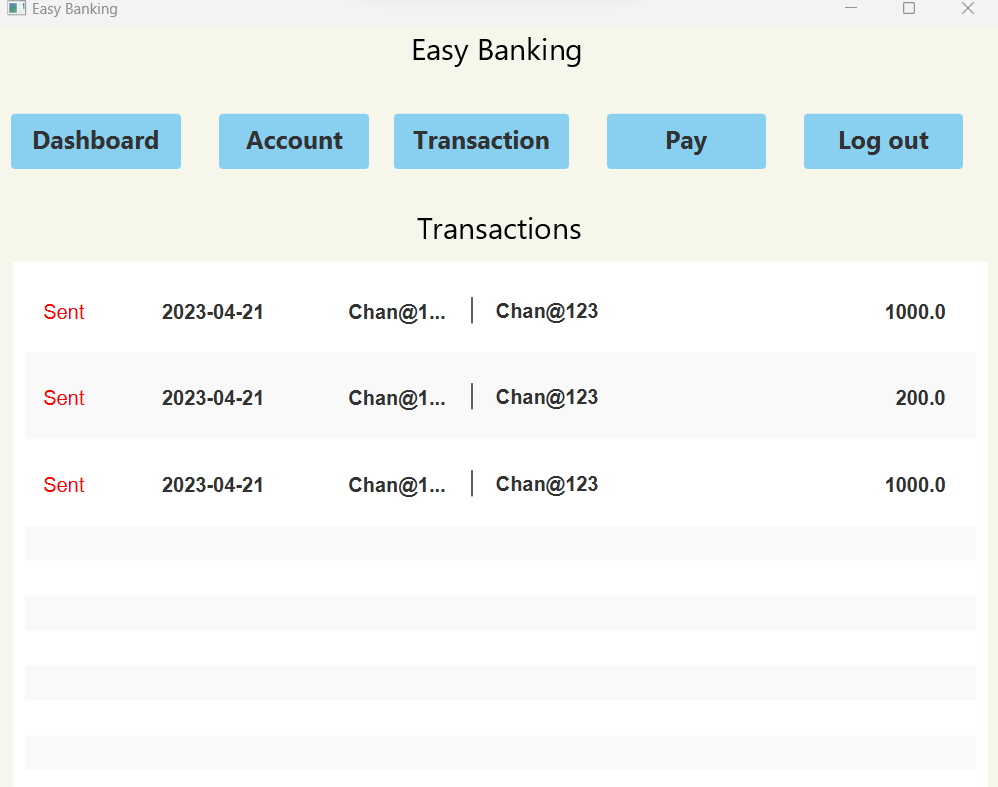
8) Account section - On the dashboard, both Saving Account and Checking Account are displayed. If the user wishes to transfer money between the two accounts, they can do so by selecting the appropriate account and entering the desired amount in the textbox. The account balance will be updated accordingly and the user can see the changes immediately(Update Operation).



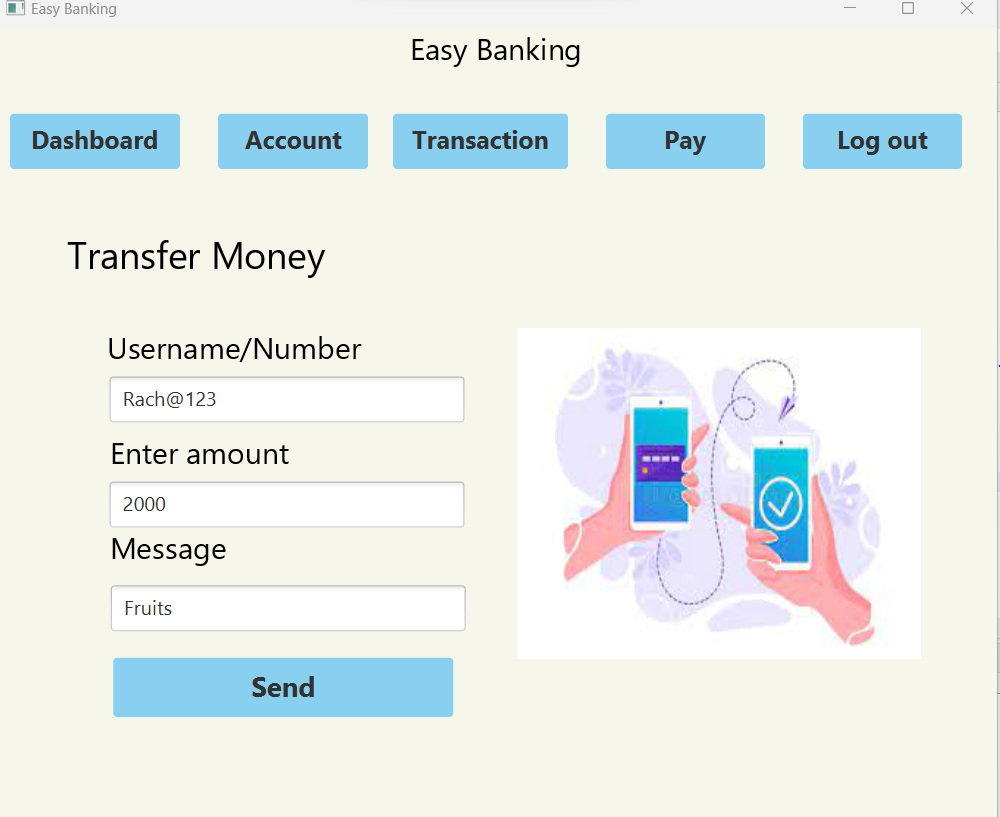
9)Updated Dashboard after update operation



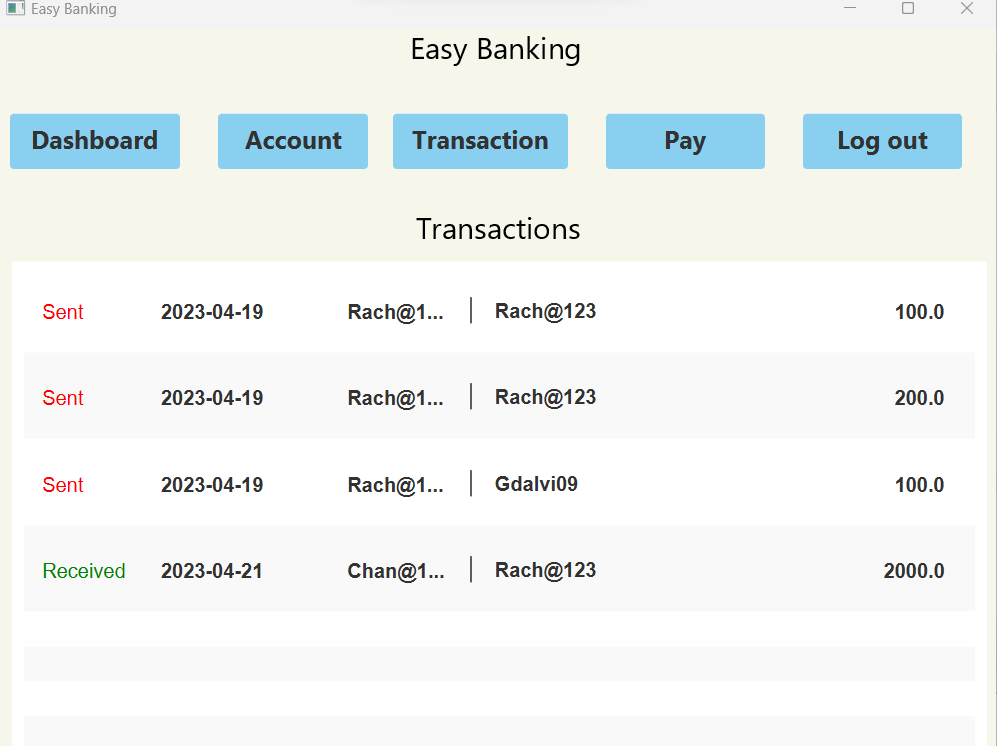
10) Transaction section - Displays all transaction



11) Pay transfer - To transfer funds to another user, you need to enter the recipient's username along with the desired amount and a message.



12) After sharing the funds with another user, the recipient's transaction section will display the received amount.



**The comparison between your solution and other people’s work -**

Compared to other banking applications, our solution is designed to be more user-friendly and less complex. While other applications may have more functionalities, they often overwhelm users with too much information, making it difficult to find what they're looking for. In contrast, our application only displays relevant data, reducing confusion and making it easier for users to navigate.

One common issue with banking applications is security. Databases can be vulnerable to hacking, which can lead to serious consequences for both the bank and its customers. To mitigate this risk, our application focuses on creating a highly secure database that prioritizes user privacy and data protection. By implementing the latest security measures and keeping up-to-date with industry best practices, we strive to ensure that our users can trust our system to keep their data safe.

**The user studies with real users, such as 3-4 people, and concludes the feedback. (quantitative or qualitative evaluation)**

We conducted a user study with three participants who were regularly using banking applications. Participants were asked to complete several tasks on our application while providing feedback.

Our application's user-friendly design received positive feedback from all participants. They appreciated the clear and simple layout, which made it easy for them to navigate and complete tasks efficiently. Moreover, participants found our application more secure than other banking applications they had used in the past, which enhanced their confidence in using it.

However, some participants expressed a desire for more customization options, despite our application's focus on displaying only relevant information. They suggested that more personalized options would improve their experience with the application. We will take this feedback into account for future updates.

In summary, the user study provided valuable qualitative feedback on our target demographic's preferences and needs, allowing us to refine our application to better meet their requirements. We will continue to incorporate feedback from users to enhance the user experience of our application.

**Pooja Dandge -**

"I found the application very user-friendly and easy to navigate. It was great to have all the necessary information clearly displayed without feeling overwhelmed. The transaction details were easy to understand and the ability to transfer funds to other users and between savings and checking accounts was straightforward. Overall, I had a positive experience using the application."

**Janvi Mehta -**

“The application is designed with a clean and intuitive user interface, making it easy for users to navigate and perform essential banking tasks. A standout feature is the transaction notification system, which alerts users when money is sent or received, providing greater visibility and tracking of their transactions. Overall, the application offers a convenient and efficient banking experience for users.”

**Priyanka Nayak -**

"I had an interesting experience using the application. As someone who uses banking applications regularly, I appreciated the simplicity and ease of use. However, I did feel that there could be more customization options for users. It can be overwhelming to look at a cluttered application, and having the ability to customize what information is displayed would be helpful. Overall, I enjoyed using the application and look forward to seeing how it evolves in the future."

# **VII. Discussion (Reflection)**

The basis of building our bank management system was completely consumer focused. We adapted Amazon’s consumer first development approach to build our system since Amazon is largely successful when it comes to keeping its users engaged or happy.

Whilst studying this approach we discovered the following Pros and Cons:

Amazon's Consumer First development approach is a customer-centric philosophy that puts the customer at the center of everything the company does. While this approach has many benefits, it also has some drawbacks that are worth considering.

Pros:

Customer satisfaction: By focusing on the customer first, Amazon is able to deliver products and services that meet or exceed customer expectations, leading to high levels of customer satisfaction and loyalty.

Innovation: The Consumer First approach encourages innovation and experimentation, as Amazon is constantly looking for ways to improve the customer experience and differentiate itself from competitors.

Agility: Amazon's focus on the customer allows the company to be more agile and responsive to changing customer needs and market trends, enabling it to stay ahead of the curve.

Cons:

The Consumer First approach can be expensive, as it requires significant investment in research, development, and customer service to ensure that customer needs are met and expectations are exceeded.

Short-term focus: The focus on meeting immediate customer needs can sometimes come at the expense of long-term planning and strategy, which may lead to missed opportunities or a lack of innovation in the long run.

Risk of complacency: Focusing too much on the customer can sometimes lead to complacency, as Amazon may become too focused on meeting existing customer needs at the expense of exploring new markets or opportunities.

Given the above points about Amazon, we thoroughly considered the cons of their approach and tried to ameliorate them by slight changes in our approach.

**Cost Improvements**:

To have a cost effective budget we mostly stuck to the open source technologies which are freely available and open to use for everyone to develop systems of their choice with freedom.

**Short Term Focus:**

Although having a Short term focus may lead to a lot of intricacies, this focus in return was very beneficial to us since the given time constraints provided the perfect environment to incorporate it in our development. We did so by building a Minimal Viable Product first and avoided adding features that were too advanced such as visualizations and learning the user’s patterns of transactions. This allowed us to focus only on the needs which were of utmost impost and help us avoid penultimate work in general.

**Risk of complacency:**

As mentioned erstwhile the way we implemented our system with slight adjustments to our short term focus, we were able to develop an improved version of most products available online. However, our approach differs from Amazon’s approach in a way that we have also considered different markets. For example, our future scope also included incorporating the corporate level transactions, credit cards, trading options and some insights to the stock market. As you can see we are not limiting our scope of the application to bank users alone but also providing them a plethora of options while maintaining the simplicity.

# **VIII. Conclusions and Future Work**

In Conclusion, a user friendly, robust and secure application was designed from the ground up. We used the MVC architecture pattern so maintaining the application becomes less convoluted. Furthermore, with an intuitive user interface user empathy is garnered.

* A benefit of using our system over the other’s present is that our application has a very easy to use interface which allows users to navigate between various parts of the application unlike other applications that bombarded the users with many features causing extreme conundrum with the users
* Although the current user interface for the account summary suffices, if time permits me and my team members would try to add more features within the frame to visualize the exact sources of expenses and provide solutions for better handling of finances
* A problem that we discovered was that JavaFX as a whole does not provide backwards compatibility. Which means that older versions of the Java Virtual Machine and legacy devices in general may have performance issues.

# **IX. Job Assignment**

GAYATRI- UML class Diagram, Created Database and Tables,Money Transfer from Checking and Saving Account, Client Controller,Create Client, Model, Data Validation.

PRATIKSHA- UML class Diagram, Created Admin and Client UI, Admin Controller,Client Account Summary, Login and Logout, Data Validation,Model, Admin Client View.

PRATIK- UML class Diagram, Account Deposit, Account Transfer, Client Transaction,Data Validation, Model, Client View Admin, Client Controller.

##### **References**

1. Ed Eden-Rump. (2021, July 30). How to apply MVC in JavaFx [Blog post]. Edencoding.com. https://edencoding.com/mvc-in-javafx/
2. Ed Eden-Rump. (2021, July 19). How to connect JavaFx to SQLite3 database [Blog post]. Edencoding.com. https://edencoding.com/connect-javafx-with-sqlite/
3. Open Genus. (2022, May 21) A general overview of how a bank management system is designed [Blog Post] https://iq.opengenus.org/system-design-of-bank-management-system/
4. Tech Entertaining. (2017, January 6). Quick Overview- Banking Management System [Video]. YouTube. https://www.youtube.com/watch?v=t8VH8QNToBw