

A PROJECT REPORT
ON
BANKING SYSTEM SIMULATION

Submitted in the partial fulfilment

For the award of course in

CORE JAVA

TO

<CODE INFINITE>

IT Training Service

Submitted by

S. HEMACHANDRAN ()

Under the guidance of

DR. R. MANSOOR AHAMAD

<CODE INFINITE> IT Training Service

#6c- 1st Floor, Chandrasekar Tower, Opp KGiSL,

Thudiyalur Rd, Saravanampatti, Coimbatore - 35

Ph : 0422-2666232 | Mobile : +91 99764 99765

info@codeinfinite.in | www.codeinfinite.in

JANUARY 2025

Declaration

I hereby declare that the project titled "Banking System Simulation" is my original work, completed as part of my academic practice to deepen my understanding of CRUD (Create, Read, Update, Delete) operations, database management, and system design.

I affirm that the work presented in this project is genuine and has not been copied or reproduced from any other source. Any references used have been acknowledged appropriately.

I take full responsibility for the authenticity and accuracy of the information, design, and implementation details contained in this project report. This work has not been submitted for any other course or purpose elsewhere.

Name: S HEMACHANDRAN

Course: CORE JAVA

Institution: CODE INFINITE

Date: 20 - 01 - 25

Acknowledgement

I would like to extend my heartfelt gratitude to my course trainer, Mr. Prakash V, for his invaluable guidance and unwavering support throughout the development of this Banking System Simulation. His expertise and advice were instrumental in deepening my understanding of JAVA, SQLite, and SWING. This foundation allowed me to effectively integrate object-oriented programming principles and database management concepts into the project.

I am also deeply grateful to Dr. Mansoor Ahamad R, the institution management head, for fostering a supportive educational environment and providing the resources that facilitated my learning journey. His dedication to creating an atmosphere where students can explore and excel is truly inspiring.

Furthermore, I wish to thank my peers and mentors for their thoughtful feedback and collaborative input during this project. Their innovative ideas and constructive suggestions helped refine the functionality and user interface of the application, ensuring a more polished and user-friendly product.

I also acknowledge the authors, developers, and contributors of various technical resources that provided valuable insights and guidance. Online communities, such as Stack Overflow and GitHub, along with official documentation for JAVA, SQLite, and SWING, were indispensable in addressing challenges and expanding my understanding of advanced programming concepts.

Finally, I would like to express my gratitude to my family and friends for their moral support and encouragement. Their confidence in my abilities motivated me to persevere through challenges and strive for excellence.

To everyone who contributed directly or indirectly to the completion of this project, I am deeply thankful. Your support and inspiration have made this experience both enriching and rewarding.

Abstract

The "Banking System Simulation" project is designed to provide a hands-on learning experience in implementing CRUD (Create, Read, Update, Delete) operations within a database-driven application. The primary goal of the project is to emulate the essential functionalities of a banking system, allowing users to create accounts, manage transactions, and retrieve data in an intuitive and efficient manner.

The project leverages the Java programming language, renowned for its robustness and platform independence, to handle core logic and database operations. By integrating MySQL as the relational database management system, the project ensures reliable and secure data storage. The Swing framework is utilized to develop a graphical user interface (GUI) that is both user-friendly and interactive, enabling seamless interaction between the user and the system.

Key functionalities of the project include:

- **Account Management:** Users can create new accounts, update account details, and delete accounts as needed.
- **Transaction Handling:** The system supports depositing, withdrawing, and transferring funds between accounts while maintaining accurate records of all transactions.
- **Data Retrieval and Analysis:** Users can query transaction histories and account summaries, providing insights into financial activities.

The project also incorporates advanced programming concepts such as the Stream API and Collection API. These features enable efficient data processing, sorting, and filtering, enhancing the overall performance and scalability of the application. By adhering to object-oriented programming (OOP) principles, the system is modular and easy to maintain, allowing for future enhancements and modifications.

In addition to its technical objectives, the project emphasizes the importance of designing applications that balance functionality with usability. The inclusion of real-time feedback mechanisms, error handling, and validation checks ensures that the system is reliable and secure for end users.

Overall, the "Banking System Simulation" serves as a comprehensive platform to demonstrate the integration of programming, database management, and GUI development. It not only fulfills its intended purpose as a practice project but also provides a solid foundation for developing more complex and feature-rich systems in the future.

Table of Content

S.no	TOPIC	Page. No
01	Introduction <ul style="list-style-type: none">○ Objectives of the Project○ Overview of CRUD in Banking Systems	07
02	Implementation <ul style="list-style-type: none">○ Coding Approach<ul style="list-style-type: none">▪ Key Functions and Classes▪ Challenges and Solutions○ Database Design<ul style="list-style-type: none">▪ Schema, Tables, and Relationships○ User Interface Design<ul style="list-style-type: none">▪ Layout, Colors, and Usability	08
03	Hardware Requirements <ul style="list-style-type: none">○ Minimum, Recommended, and High-End Specifications○ Peripheral and Network Requirements	10
04	Technologies Used <ul style="list-style-type: none">○ Programming Language○ Database Management○ GUI Framework○ Supporting Libraries and Tools	12

05	Special Features <ul style="list-style-type: none"> ○ Transaction Management ○ User-Friendly Interface ○ Scalability and Performance 	13
06	Project File Structure	14
07	Future Enhancements	14
08	Real-World Application	15
09	Sample Code	17
10	Sample Screen	35
11	Conclusion	39
12	References <ul style="list-style-type: none"> ○ Documentation and Tutorials ○ Online Communities and Libraries ○ Development Tools and APIs 	39

1. Introduction

Motivation for the Project

The project is designed to provide hands-on experience in implementing CRUD (Create, Read, Update, Delete) operations while solving a real-world problem: managing banking functionalities. Users can create accounts, store data securely, and practice modifying or deleting records. This serves as a practical learning environment for database interaction, user interface design, and secure transaction management, which are critical in modern software development.

Objectives of the Project

Master CRUD Operations: Build a strong foundation by implementing core CRUD functionalities across account and transaction data.

Enhance Data Security: Use password encryption and access control for managing sensitive financial records.

Simplify Real-Time Transactions: Enable live updates for balances and transaction logs, ensuring accuracy and user trust.

Improve Usability: Create an intuitive platform where even non-technical users can manage their financial tasks easily.

Bridge Learning with Real-World Applications: Apply CRUD in practical scenarios while designing scalable systems.

System Overview

Key Features and Functionality

The Banking System Application encompasses a range of features designed to simplify banking operations while maintaining high levels of security and reliability.

1. **Account Management:** Users can create new accounts, update account details, and view balances. This feature supports both personal and business accounts, enabling flexibility.
2. **Transaction Management:** The system enables users to perform deposits, withdrawals, and inter-account transfers. Every transaction is logged with timestamps and categorized for easy retrieval.
3. **Comprehensive Database:** The backend database ensures the safe storage of user details, account data, and transaction histories. The schema is optimized for quick data retrieval and minimal redundancy.
4. **Authentication Mechanism:** Secure login and logout processes are implemented to protect user data from unauthorized access.
5. **Cross-Platform Support:** The system's compatibility with Windows, macOS, and Linux makes it accessible to a broader audience.
6. **Reporting Features:** Users can generate transaction reports for specified time periods, aiding in financial analysis and planning.

Benefits of the System

1. **Time-Saving:** Automation of processes such as balance updates and transaction logging reduces manual effort.
2. **Enhanced Security:** Encrypted passwords and secure connections ensure that sensitive information remains protected.
3. **Scalability:** The system can accommodate growing data volumes and additional features without compromising performance.
4. **Customizability:** Users can tailor the interface and settings to align with their specific needs, enhancing usability.

This combination of features and benefits positions the Banking System Application as a versatile and dependable solution for diverse financial management needs.

2. Implementation

Coding Approach

The Banking System Application is structured using a layered architecture that promotes modularity and maintainability. Each layer has a specific role, ensuring clear separation of concerns and simplifying debugging and future enhancements.

Layers:

1. Data Access Layer (DAO) : Manages interactions with the SQLite database, including executing SQL queries and handling connections. Key classes include:

- `AccountDAO.java`: Handles CRUD operations related to user accounts.
- `TransactionDAO.java`: Processes transaction-related data.

2. Service Layer: Implements the core business logic and acts as a bridge between the DAO and presentation layers. For instance, `TransactionService` validates transaction details and coordinates database updates.

3. Presentation Layer: Developed using Java Swing, this layer provides the user interface. It consists of multiple forms and dashboards to guide user interactions.

Workflow Example: Transaction Processing

1. The user initiates a withdrawal through the GUI.
2. The service layer validates the user's credentials, checks the account balance, and ensures that the requested amount is available.
3. The DAO updates the account balance and logs the transaction details in the database.
4. A success message is displayed on the GUI.

Database Design

The application's database design adheres to best practices, ensuring data normalization and integrity. Below is a detailed schema description:

1. Accounts Table:

- a. Columns: `id` (Primary Key), `name`, `balance`
- b. Purpose: Stores user account details and balances.

2. Transactions Table:

- c. Columns: `id` (Primary Key), `account_id` (Foreign Key), `type`, `amount`, `timestamp`
- d. Purpose: Logs transactions such as deposits, withdrawals, and transfers.

3. Users Table:

- e. Columns: `id` (Primary Key), `username`, `password`
- f. Purpose: Maintains login credentials for user authentication.

User Interface Design

The application's GUI, developed using Java Swing, prioritizes simplicity and efficiency. Key features include:

- 1. **Dashboard:** Displays an overview of account balances, recent transactions, and quick action buttons.
- 2. **Login Page:** Provides secure access to the system with input validation and error messages for incorrect credentials.
- 3. **Transaction Forms:** Guides users through deposits, withdrawals, and transfers with clear instructions.
- 4. **Responsive Design:** Ensures compatibility across various screen resolutions.

Error Handling and Logging

To ensure reliability, the application incorporates robust error-handling mechanisms. Common issues addressed include:

- 1. **Database Connection Failures:** Informative messages guide users, while logs capture technical details for debugging.
- 2. **Invalid Inputs:** Input validation prevents errors such as negative transaction amounts or empty fields.
- 3. **System Exceptions:** Unhandled exceptions are logged, preserving critical data for analysis.

Logs are stored in a dedicated file, ensuring transparency and simplifying the debugging process.

3. Hardware Requirements

The Banking System Simulation is a lightweight software that primarily runs on personal computers. Below is a detailed list of the hardware requirements based on different system configurations to ensure smooth development, testing, and execution of the application.

1. Minimum Hardware Requirements

- Processor:
 - Intel Core i3 (3rd generation or later) or AMD equivalent with at least 2 cores.
 - Rationale: Handles basic Python operations and GUI rendering efficiently.
- RAM:
 - 4 GB
 - Rationale: Provides sufficient memory for running lightweight applications and basic development tools like Python IDEs.
- Storage:
 - 256 GB HDD or SSD (with at least 5 GB free for software and project files).
 - Rationale: Stores project files, SQLite database, and Python installation.
- Display:
 - 1366x768 resolution monitor.
 - Rationale: Suitable for viewing the Swing-based GUI and running development tools.
- Graphics:
 - Integrated GPU (e.g., Intel HD Graphics).
 - Rationale: Swing applications don't require advanced graphics rendering.
- Input Devices:
 - Standard keyboard and mouse.

2. Recommended Hardware Requirements (For Optimal Performance)

- Processor:
 - Intel Core i5 (6th generation or later) or AMD Ryzen 3 (latest generation).
 - Rationale: Enhances performance when running multiple programs like Python IDEs, database tools, and testing environments simultaneously.
- RAM:
 - 8 GB
 - Rationale: Enables smooth multitasking, especially if you are using modern IDEs like PyCharm or Visual Studio Code alongside debugging tools.
- Storage:
 - 512 GB SSD
 - Rationale: Faster read/write speeds for project files and application execution, along with sufficient storage for additional resources.

- Display:
 - Full HD (1920x1080) monitor.
 - Rationale: Provides a more comfortable workspace for coding, debugging, and designing GUIs.
- Graphics:
 - Integrated or entry-level discrete GPU (e.g., NVIDIA GeForce GTX 1050).
 - Rationale: Ensures fluid GUI rendering and allows for smoother testing of other potential Java libraries with graphical components.

3. High-End Hardware Requirements (For Advanced Development and Future Scalability)

- Processor:
 - Intel Core i7 (10th generation or later) or AMD Ryzen 5/7 (latest generation).
 - Rationale: Ideal for running advanced simulations, performance testing, or integrating AI/ML components in future application updates.
- RAM:
 - 16 GB or higher
 - Rationale: Supports resource-intensive tasks like data analysis, running virtual environments, or multitasking across multiple tools.
- Storage:
 - 1 TB SSD (NVMe preferred).
 - Rationale: Provides ample storage for large datasets, libraries, and future extensions of the application.
- Display:
 - 4K monitor.
 - Rationale: Offers a premium coding and GUI design experience, particularly helpful for detailed visual debugging.
- Graphics:
 - NVIDIA RTX series (e.g., RTX 3060) or AMD Radeon RX series.
 - Rationale: Allows for integration of advanced GUI libraries or 3D graphical interfaces if planned in future updates.
- Peripheral Devices:
 - High-quality mechanical keyboard and ergonomic mouse for long development sessions.

Additional Hardware Considerations

1. External Backup Storage:
 - A 1 TB external HDD or SSD to back up project files and database snapshots.
2. Networking Requirements:
 - Reliable internet connection (broadband or fiber) for accessing resources, documentation, and cloud services.
3. Power Backup:
 - UPS (Uninterruptible Power Supply) with a minimum of 600 VA for uninterrupted development sessions during power outages.
4. Printer (Optional):
 - Useful for printing documentation, project reports, or user manuals if required.
5. Testing Device (Optional):
 - A basic laptop or PC with lower specifications to test application performance on less powerful machines.

By adhering to these hardware specifications, you can ensure a seamless development and user experience for the Expense Tracker Application.

4. Technologies Used

Programming Language

Java: Selected for its platform independence and extensive library support, making it ideal for this cross-platform application.

Frameworks and Libraries

1. Swing: Powers the GUI, enabling the creation of responsive and intuitive user interfaces.
2. JDBC: Facilitates database connectivity and SQL query execution.
3. MySQL Connector: Ensures seamless integration with SQLite.

Database Management System

SQLite: A lightweight yet powerful database engine that simplifies deployment and maintenance.

Development Tools

1. Integrated Development Environment (IDE) : IntelliJ IDEA and Eclipse were used for efficient coding and debugging.
2. Version Control: Git and GitHub facilitated collaboration and version management.
3. Build Tools: Maven automated dependency management and project builds.

5. Special Features

Security and Authentication

The application employs industry-standard security measures, including:

1. Password Encryption: User passwords are stored as encrypted hashes, safeguarding against breaches.
2. Session Management: Limits concurrent sessions to prevent misuse.

Transaction Management

1. Detailed Logs: Every transaction includes metadata such as timestamps and descriptions.
3. Error Recovery: Failed transactions trigger rollback mechanisms, preserving database consistency.

User-Friendly Interface

1. Simplified Navigation: Clearly defined menus and forms ensure a smooth user experience.
2. Custom Filters: Users can sort transactions by type, date, or amount, enhancing accessibility.

Scalability and Performance

1. The system's architecture is designed to accommodate new features and increased user loads. Techniques such as database indexing and query optimization ensure consistent performance.

6. Project File Structure

```
BankingSystem/
├── src/
│   ├── main/
│   │   ├── App.java
│   │   ├── dao/
│   │   │   ├── AccountDAO.java
│   │   │   ├── TransactionDAO.java
│   │   ├── models/
│   │   │   ├── Account.java
│   │   │   ├── Transaction.java
│   │   ├── services/
│   │   │   ├── AccountService.java
│   │   │   ├── TransactionService.java
│   │   ├── utils/
│   │   │   ├── DatabaseConnection.java
│   ├── resources/
│   │   ├── schma.sql
│   │   ├── data.sql
│   ├── lib/
│   │   ├── mysql-connector-j-9.1.0.jar
│   └── README.md
```

7. Future Enhancements

1. Cloud Integration

Implement cloud storage for user data, allowing cross-device access.

2. Mobile Application

Develop mobile apps using frameworks like Flutter or React Native.

2. AI-Powered Insights

Integrate machine learning models to provide spending analysis and recommendations.

3. Enhanced Reporting

Generate financial summaries in various formats (PDF, Excel) for detailed insights.

8. Real-World Applications

The Banking System Application can be used in diverse scenarios, including:

1. Personal Finance Management: Tracks individual spending and savings.
2. Small Businesses: Assists in managing payroll, expenses, and revenues.
3. Educational Use: Serves as a case study for teaching database management and software design principles.

Explanation of Key Components

1. **App.java**
 - The starting point of the application. It initializes the program, sets up configurations, and acts as the entry point for user interaction. It connects the database and service components with the graphical user interface (GUI).
2. **database/**
 - **BankingSystem.db**: The SQLite database file that stores all data related to accounts, transactions, and users.
 - **schema.sql**: Contains scripts for setting up the database schema, including tables for accounts, transactions, and user details.
 - **DatabaseConnection.java**: Handles the connection between the application and the SQLite database. It ensures proper connection management, error handling, and resource cleanup.
3. **dao/** (Data Access Object Layer)
 - **AccountDAO.java**: Manages all database operations related to user accounts, such as creating, reading, updating, and deleting account records.
 - **TransactionDAO.java**: Handles database queries for transactions, including deposits, withdrawals, transfers, and logging details.
4. **services/**
 - **AccountService.java**: Implements the business logic for account-related operations. It validates user input, ensures sufficient funds for withdrawals, and updates account balances.
 - **TransactionService.java**: Handles transaction-related operations. It processes deposits, withdrawals, and inter-account transfers, ensuring consistency in transaction logs.
5. **gui/** (Graphical User Interface)
 - Contains all GUI components organized by functionality, such as:
 - **LoginForm.java**: Provides a secure login interface for users.
 - **Dashboard.java**: Displays account summaries, transaction histories, and quick action buttons for deposits, withdrawals, and transfers.
 - **TransactionForms.java**: Handles specific user interactions for performing deposits, withdrawals, and transfers.
 - **themes/**: Includes CSS files or predefined styling configurations to customize the application's appearance, ensuring a visually appealing and consistent user experience.

6. **utils/**

- **ValidationUtils.java**: Provides utility functions for input validation, such as checking for valid account numbers and ensuring transaction amounts are positive.
- **DateUtils.java**: Includes helper methods for formatting and manipulating dates for transaction logging.
- **ExportUtils.java**: Handles exporting transaction and account data to external formats such as CSV or Excel.

7. **tests/**

- Contains automated test scripts for verifying the functionality and stability of the application:
 - **AccountServiceTest.java**: Unit tests for account-related operations.
 - **TransactionServiceTest.java**: Unit tests for transaction processing.
 - **IntegrationTests.java**: End-to-end tests to ensure seamless interaction between GUI, service, and database components.

8. **requirements.txt**

- A list of third-party libraries and dependencies required to run the project. Examples include:
 - **sqlite-jdbc**: For database connectivity.
 - **JUnit**: For testing the application.

9. **README.md**

- Provides an overview of the project, including its purpose, installation instructions, usage guidelines, and system requirements. It serves as a reference for developers and users.

10. **docs/**

- Contains additional documentation to support the application:
 - **UserManual.pdf**: A detailed guide for end-users on how to navigate and use the application.
 - **ProjectReport.pdf**: A comprehensive report on the development process, system design, and features of the application.

This structured approach ensures that the Banking System project is modular, easy to maintain, and scalable for future enhancements. Let me know if you'd like to adjust or add anything!

9. Sample Code

1. Main

BankingAppSwing.java

```
package main;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.sql.SQLException;
import javax.swing.*;
import main.models.Account;
import main.services.AccountService;
import main.services.TransactionService;

public class BankingAppSwing {

    private JFrame frame;

    private final AccountService accountService;

    private final TransactionService transactionService;

    private Container c;

    public BankingAppSwing() {

        accountService = new AccountService();

        transactionService = new TransactionService();

        initialize();

    }

    private void initialize() {
```

```

frame = new JFrame("Banking System");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

frame.setBounds(300, 90, 900, 600);

frame.setResizable(false);


c = frame.getContentPane();

c.setLayout(null);


JLabel title = new JLabel("Banking System");

title.setFont(new Font("Arial", Font.PLAIN, 30));

title.setSize(300, 30);

title.setLocation(300, 30);

c.add(title);


JButton createAccountButton = createMenuButton("Create Account", 100);

JButton viewAccountButton = createMenuButton("View Account", 150);

JButton depositButton = createMenuButton("Deposit Money", 200);

JButton withdrawButton = createMenuButton("Withdraw Money", 250);

JButton viewTransactionsButton = createMenuButton("View Transactions", 300);

JButton adminButton = createMenuButton("Admin", 350);

JButton exitButton = createMenuButton("Exit", 400);


createAccountButton.addActionListener(e -> openCreateAccountPanel());

viewAccountButton.addActionListener(e -> openViewAccountPanel());

depositButton.addActionListener(e -> openDepositPanel());

withdrawButton.addActionListener(e -> openWithdrawPanel());

viewTransactionsButton.addActionListener(e -> openViewTransactionsPanel());

```

```

        adminButton.addActionListener(e -> openAdminPanel());
        exitButton.addActionListener(e -> System.exit(0));

        frame.setVisible(true);
    }

    private JButton createMenuButton(String text, int yPos) {
        JButton button = new JButton(text);
        button.setFont(new Font("Arial", Font.PLAIN, 20));
        button.setSize(300, 40);
        button.setLocation(300, yPos);
        c.add(button);
        return button;
    }

    private void openCreateAccountPanel() {
        JPanel panel = createFormPanel("Create Account");

        JLabel typeLabel = createLabel("Account Type:", 100);
        String[] accountTypes = { "Savings", "Current" };
        JComboBox<String> typeComboBox = new JComboBox<>(accountTypes);
        typeComboBox.setFont(new Font("Arial", Font.PLAIN, 15));
        typeComboBox.setSize(200, 30);
        typeComboBox.setLocation(300, 100);
        panel.add(typeLabel);
        panel.add(typeComboBox);
    }

```

```

JLabel balanceLabel = createLabel("Initial Balance:", 150);
JTextField balanceField = createTextField(300, 150);
panel.add(balanceLabel);
panel.add(balanceField);

JButton createButton = createActionButton("Create", 200);
JButton backButton = createActionButton("Back to Main Menu", 250);

createButton.addActionListener(e -> {
    String type = (String) typeComboBox.getSelectedItem();
    double balance;
    try {
        balance = Double.parseDouble(balanceField.getText());
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(frame, "Please enter a valid number for the initial
balance.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
    }
    try {
        Account account = accountService.createAccount(type, balance);
        JOptionPane.showMessageDialog(frame, "Account created successfully: " + account);
        initialize();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
    }
});

```

```

        backButton.addActionListener(e -> initialize());

        panel.add(createButton);
        panel.add(backButton);

        switchPanel(panel);
    }

    private void openViewAccountPanel() {
        JPanel panel = createFormPanel("View Account");

        JLabel idLabel = createLabel("Enter Account ID:", 100);
        JTextField idField = createTextField(300, 100);
        panel.add(idLabel);
        panel.add(idField);

        JButton viewButton = createActionButton("View", 150);
        JButton backButton = createActionButton("Back to Main Menu", 200);

        viewButton.addActionListener(e -> {
            long accountId = Long.parseLong(idField.getText());
            try {
                Account account = accountService.getAccountDetails(accountId);

                if(account == null) {JOptionPane.showMessageDialog(frame, "Account Not Found!
");}
            }
        });
    }

```

```

        else {JOptionPane.showMessageDialog(frame, "Account Details: " + account);}

    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
    }
});

backButton.addActionListener(e -> initialize());

panel.add(viewButton);
panel.add(backButton);

switchPanel(panel);
}

private void openDepositPanel() {
    JPanel panel = createFormPanel("Deposit Money");

    JLabel idLabel = createLabel("Enter Account ID:", 100);
    JTextField idField = createTextField(300, 100);
    panel.add(idLabel);
    panel.add(idField);

    JButton verifyButton = createActionButton("Verify Account", 150);
    JButton backButton = createActionButton("Back to Main Menu", 200);

```

```

verifyButton.addActionListener((ActionEvent e) -> {
    long accountId = Long.parseLong(idField.getText());
    try {
        Account account = accountService.getAccountDetails(accountId);
        if(account == null){
            JOptionPane.showMessageDialog(frame, "Account not Found.");
        }

        else{
            JOptionPane.showMessageDialog(frame, "Account verified.");
            JPanel depositPanel = createFormPanel("Deposit Money - Account Verified");

            JLabel amountLabel = createLabel("Enter Amount:", 100);
            JTextField amountField = createTextField(300, 100);
            depositPanel.add(amountLabel);
            depositPanel.add(amountField);

            JButton depositButton = createActionButton("Deposit", 150);
            JButton backToMainButton = createActionButton("Back to Main Menu", 200);

            depositButton.addActionListener(ev -> {
                double amount = Double.parseDouble(amountField.getText());
                try {
                    transactionService.recordTransaction(accountId, amount, "Credit");
                    JOptionPane.showMessageDialog(frame, "Deposit successful!");
                    initialize();
                } catch (SQLException ex) {
                    JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",

```

```

JOptionPane.ERROR_MESSAGE);

    }

    });

    backToMainButton.addActionListener(ev -> initialize());

    depositPanel.add(depositButton);
    depositPanel.add(backToMainButton);

    switchPanel(depositPanel);
}

} catch (SQLException ex) {
    JOptionPane.showMessageDialog(frame, "Error: Account not found! " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
}

});

backButton.addActionListener(e -> initialize());

panel.add(verifyButton);
panel.add(backButton);

switchPanel(panel);
}

private void openWithdrawPanel() {

```



```
JPanel panel = createFormPanel("Withdraw Money");
```

```
JLabel idLabel = createLabel("Enter Account ID:", 100);
```

```
JTextField idField = createTextField(300, 100);
```

```
panel.add(idLabel);
```

```
panel.add(idField);
```

```
JButton verifyButton = createActionButton("Verify Account", 150);
```

```
JButton backButton = createActionButton("Back to Main Menu", 200);
```

```
verifyButton.addActionListener(e -> {
```

```
    long accountId = Long.parseLong(idField.getText());
```

```
    try {
```

```
        Account account = accountService.getAccountDetails(accountId);
```

```
        if(account == null){
```

```
            JOptionPane.showMessageDialog(frame, "Account not Found.");}
```

```
    else{
```

```
        JOptionPane.showMessageDialog(frame, "Account verified! ");
```

```
JPanel withdrawPanel = createFormPanel("Withdraw Money - Account Verified");
```

```
JLabel amountLabel = createLabel("Enter Amount:", 100);
```

```
JTextField amountField = createTextField(300, 100);
```

```
withdrawPanel.add(amountLabel);
```

```
withdrawPanel.add(amountField);
```

```
JButton withdrawButton = createActionButton("Withdraw", 150);
```

```

JButton backToMainButton = createActionButton("Back to Main Menu", 200);

withdrawButton.addActionListener((ActionEvent ev) -> {
    double amount = Double.parseDouble(amountField.getText());
    try {
        transactionService.recordTransaction(accountId, amount, "Debit");
        JOptionPane.showMessageDialog(frame, "Withdrawal successful!");
        initialize();
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
    }
});

backToMainButton.addActionListener(ev -> initialize());

withdrawPanel.add(withdrawButton);
withdrawPanel.add(backToMainButton);

switchPanel(withdrawPanel);
}
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(frame, "Error: Account not found! " +
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
}
});

backButton.addActionListener(e -> initialize());

```

```

        panel.add(verifyButton);
        panel.add(backButton);

        switchPanel(panel);
    }

    private void openViewTransactionsPanel() {
        JPanel panel = createFormPanel("View Transactions");

        JLabel idLabel = createLabel("Enter Account ID:", 100);
        JTextField idField = createTextField(300, 100);
        panel.add(idLabel);
        panel.add(idField);

        JButton viewButton = createActionButton("View", 150);
        JButton backButton = createActionButton("Back to Main Menu", 200);

        viewButton.addActionListener(e -> {
            long accountId = Long.parseLong(idField.getText());
            try {
                var transactions = transactionService.getTransactionsByAccountId(accountId);
                if (transactions.isEmpty()) {
                    JOptionPane.showMessageDialog(frame, "No transactions found!");
                } else {
                    String[] columnNames = { "Transaction ID", "Account ID", "Amount", "Type",
"Date" };

```

```

        Object[][] data = transactions.stream()
            .map(t -> new Object[] { t.getTransactionId(), t.getAccountId(), t.getAmount(),
t.getType(), t.getDate() })
            .toArray(size -> new Object[size][]);

        JTable table = new JTable(data, columnNames);
        JScrollPane scrollPane = new JScrollPane(table);
        scrollPane.setBounds(50, 250, 800, 300);

        panel.add(scrollPane);
        panel.revalidate();
        panel.repaint();
    }
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
}
});

backButton.addActionListener(e -> initialize());

panel.add(viewButton);
panel.add(backButton);

switchPanel(panel);
}

```

```

private void openAdminPanel() {
    JPanel panel = createFormPanel("Admin Panel");

    JLabel userLabel = createLabel("Username:", 100);
    JTextField userField = createTextField(300, 100);
    panel.add(userLabel);
    panel.add(userField);

    JLabel passLabel = createLabel("Password:", 150);
    JPasswordField passField = new JPasswordField();
    passField.setFont(new Font("Arial", Font.PLAIN, 15));
    passField.setSize(200, 30);
    passField.setLocation(300, 150);
    panel.add(passLabel);
    panel.add(passField);

    JButton loginButton = createActionButton("Login", 200);
    JButton backButton = createActionButton("Back to Main Menu", 250);

    loginButton.addActionListener(e -> {
        String username = userField.getText();
        String password = new String(passField.getPassword());
        if (username.equals("admin") && password.equals("admin123")) {
            openAdminOptionsPanel();
        } else {
            JOptionPane.showMessageDialog(frame, "Invalid credentials!");
        }
    });
}

```

```

    }
});

backButton.addActionListener(e -> initialize());

panel.add(loginButton);
panel.add(backButton);

switchPanel(panel);
}

private void openAdminOptionsPanel() {
JPanel panel = createFormPanel("Admin Options");

JButton viewAccountsButton = createActionButton("View All Accounts", 100);
JButton deleteAccountButton = createActionButton("Delete Account", 150);
JButton backButton = createActionButton("Back to Main Menu", 200);

viewAccountsButton.addActionListener(e -> {
    try {
        var accounts = accountService.getAllAccounts();
        if (accounts.isEmpty()) {
            JOptionPane.showMessageDialog(frame, "No accounts found!");
        } else {
            String[] columnNames = { "Account ID", "Type", "Balance" };
            Object[][] data = accounts.stream()
                .map(a -> new Object[] { a.getAccountId(), a.getAccountType(), a.getBalance() })

```

```

        .toArray(Object[][]::new);

        JTable table = new JTable(data, columnNames);
        JScrollPane scrollPane = new JScrollPane(table);
        scrollPane.setBounds(50, 250, 800, 300);

        panel.add(scrollPane);
        panel.revalidate();
        panel.repaint();
    }
} catch (SQLException ex) {
    JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
}
});

deleteAccountButton.addActionListener(e -> {
    String accountId = JOptionPane.showInputDialog(frame, "Enter Account ID to delete:");
    try {
        accountService.deleteAccountById(Long.parseLong(accountId));
        JOptionPane.showMessageDialog(frame, "Account deleted successfully!");
    } catch (SQLException ex) {
        JOptionPane.showMessageDialog(frame, "Error: " + ex.getMessage(), "Error",
JOptionPane.ERROR_MESSAGE);
    }
});

```

```

        backButton.addActionListener(e -> initialize());

        panel.add(viewAccountsButton);
        panel.add(deleteAccountButton);
        panel.add(backButton);

        switchPanel(panel);
    }

    private JPanel createFormPanel(String title) {
        JPanel panel = new JPanel(null);
        panel.setSize(900, 600);

        JLabel titleLabel = new JLabel(title);
        titleLabel.setFont(new Font("Arial", Font.PLAIN, 25));
        titleLabel.setSize(400, 30);
        titleLabel.setLocation(100, 30);
        panel.add(titleLabel);

        return panel;
    }

    private JLabel createLabel(String text, int yPos) {
        JLabel label = new JLabel(text);
        label.setFont(new Font("Arial", Font.PLAIN, 20));
        label.setSize(200, 30);
        label.setLocation(100, yPos);
    }

```



```
        return label;
    }
}
```

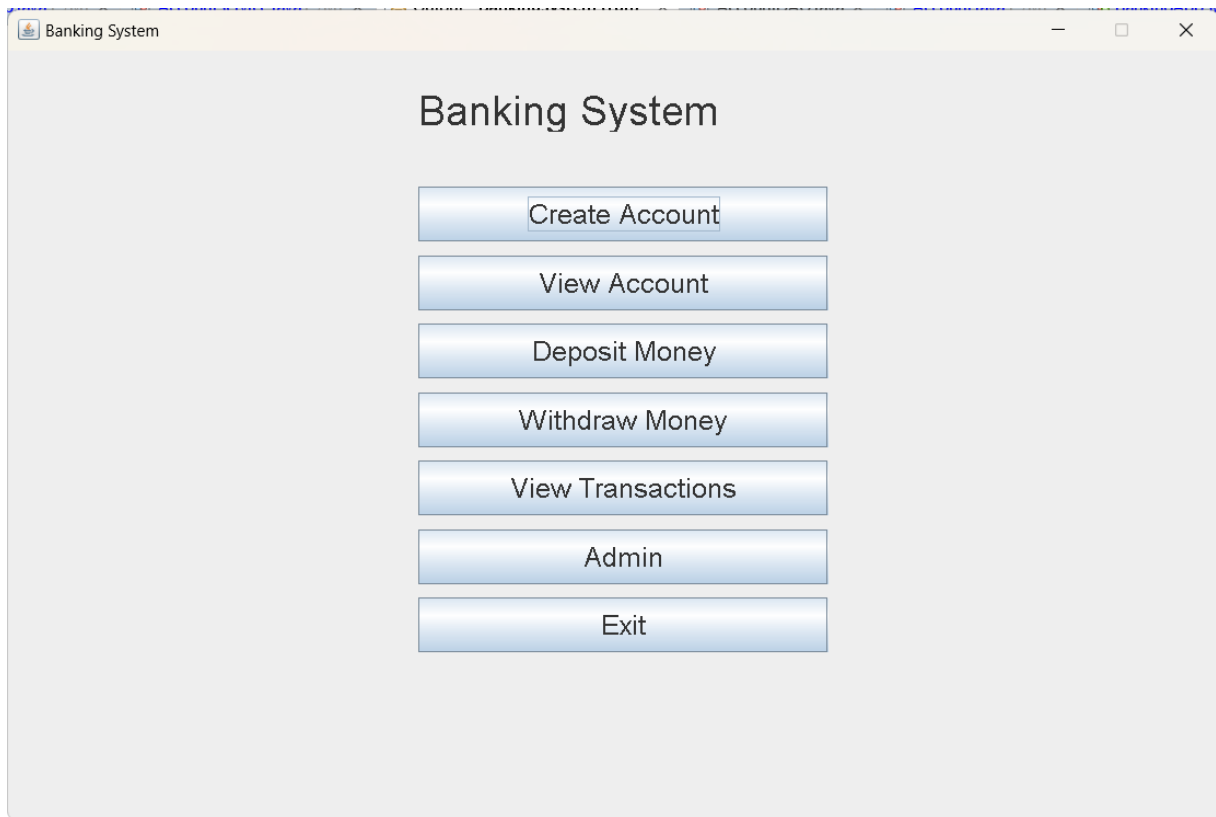
```
private JTextField createTextField(int xPos, int yPos) {
    JTextField textField = new JTextField();
    textField.setFont(new Font("Arial", Font.PLAIN, 15));
    textField.setSize(200, 30);
    textField.setLocation(xPos, yPos);
    return textField;
}
```

```
private JButton createActionButton(String text, int yPos) {
    JButton button = new JButton(text);
    button.setFont(new Font("Arial", Font.PLAIN, 15));
    button.setSize(200, 30);
    button.setLocation(300, yPos);
    return button;
}
```

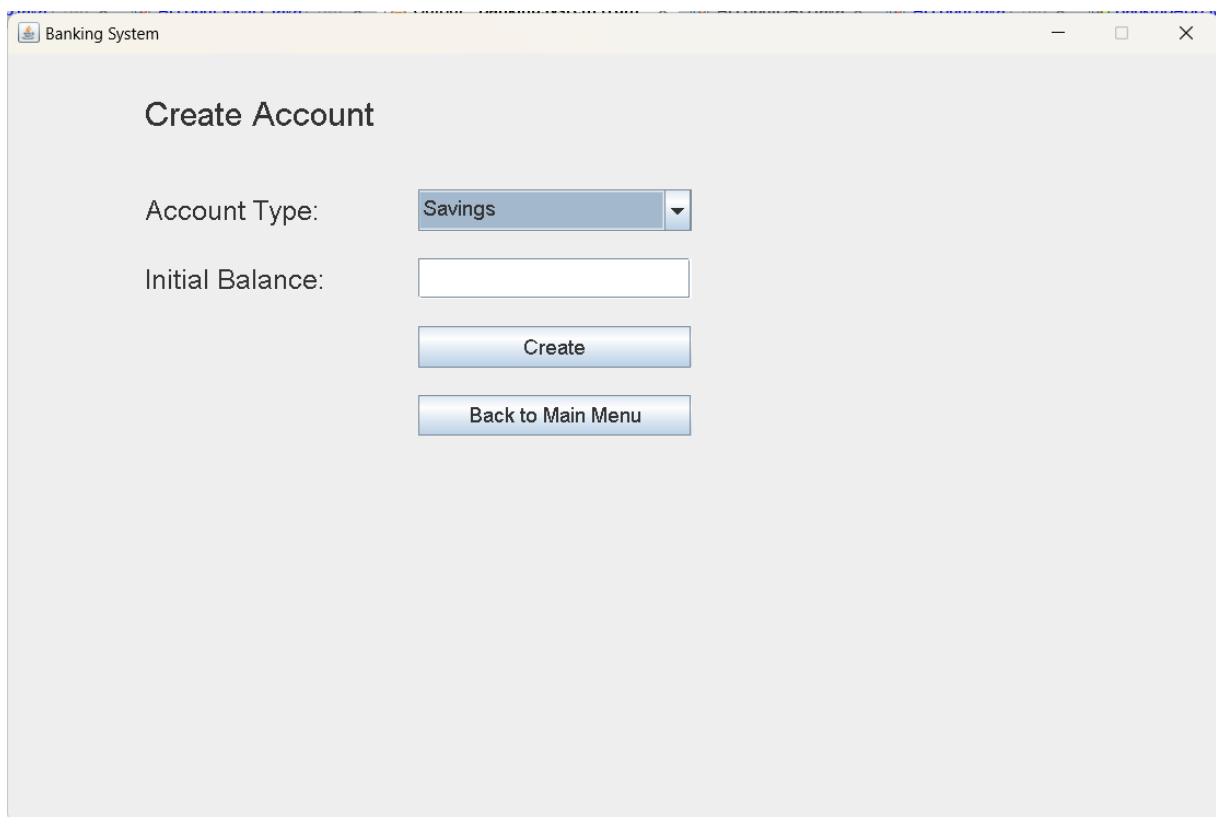
```
// private JButton createActionButton(String text, int containerWidth, int yPos) {
//     JButton button = new JButton(text);
//     button.setFont(new Font("Arial", Font.PLAIN, 15));
//     button.setSize(200, 30);
//     int xPos = (containerWidth - button.getWidth()) / 2;
//     button.setLocation(xPos, yPos);
//     return button;
// }
```

```
private void switchPanel(JPanel panel) {  
    frame.getContentPane().removeAll();  
    frame.getContentPane().add(panel);  
    frame.revalidate();  
    frame.repaint();  
}  
  
public static void main(String[] args) {  
    SwingUtilities.invokeLater(BankingAppSwing::new);  
}  
}
```

10. Sample Screen:



A screenshot of a software window titled "Banking System". The window has a light gray background and a title bar with standard Windows window controls (minimize, maximize, close). In the center of the window, the text "Banking System" is displayed in a large, bold, black font. Below this title, there is a vertical stack of eight blue buttons with white text, each with a slight gradient and a shadow effect. The buttons are labeled: "Create Account", "View Account", "Deposit Money", "Withdraw Money", "View Transactions", "Admin", and "Exit".



A screenshot of a software window titled "Banking System" showing the "Create Account" screen. The window has a light gray background and a title bar with standard Windows window controls. The text "Create Account" is displayed in a large, bold, black font at the top left. Below this, there are two labels: "Account Type:" and "Initial Balance:". The "Account Type:" label is followed by a blue dropdown menu with a white arrow pointing down, showing the word "Savings". The "Initial Balance:" label is followed by a white text input field with a thin gray border. Below these input fields, there are two blue buttons with white text. The first button is labeled "Create" and the second button is labeled "Back to Main Menu".

Banking System

View Account

Enter Account ID:

Banking System

Deposit Money

Enter Account ID:

Banking System

Withdraw Money

Enter Account ID:

Verify Account

Back to Main Menu

Banking System

View Transactions

Enter Account ID:

View

Back to Main Menu

Banking System

Admin Panel

Username:

Password:

Login

Back to Main Menu

Banking System

Admin Options

View All Accounts

Delete Account

Back to Main Menu

11. Conclusion

The Banking System Application delivers a comprehensive solution for financial management, blending security, usability, and scalability. By leveraging modern technologies and adhering to industry standards, the project achieves its objectives and lays the foundation for future growth.

This project has been a valuable learning experience, encompassing software design, database development, and user interface design. It showcases the potential of technology in revolutionizing traditional banking practices, offering users a reliable and efficient tool for managing their finances.

12. References

To develop and enhance the Expense Tracker Application, the following resources and tools were utilized:

1. Programming Language:

- JAVA documentation: [Java Documentation - Get Started](#)

2. Database:

- SQLite documentation: <https://sqlite.org>

3. GUI Framework:

- Swing tutorial: [javax.swing \(Java Platform SE 8 \)](#)

4. Online Communities:

- Stack Overflow: <https://stackoverflow.com>
- GeeksforGeeks: <https://geeksforgeeks.org>

5. Learning Resources:

- TutorialsPoint: <https://tutorialspoint.com>
- W3Schools: <https://w3schools.com>

6. Version Control:

- GitHub: Used for code versioning and collaboration: <https://github.com>

These resources collectively supported the creation, testing, and improvement of the application