Jon Guevara Banking System v0.50

Java NCIII 2025 Batch 3 Programming Course Completion Project

**Prepared by:**

**Jonathan Guevara**

**Java NCIII 2025 Batch 3**

**Github:**

https://github.com/guevarajf/java\_nciii\_2025batch3.git

**Folder:**

Project/BankingSystem/

|  |
| --- |
| **TABLE OF CONTENTS:**  [1 Executive Summary 1](#_Toc206675874)  [2 System Architecture 2](#_Toc206675875)  [3 Functional Specifications 3](#_Toc206675876)  [4 Technical Implementation 4](#_Toc206675877)  [5 User Interface Design 5](#_Toc206675878)  [6 Data Flow Architecture 11](#_Toc206675879)  [7 Java NCIII Course Completion Competency 12](#_Toc206675880)  [8 System Requirements and Deployment 13](#_Toc206675881) |

# Executive Summary

## System Overview

The **Jon Guevara Banking System v0.50** is a desktop application developed as a project for the Java NCIII 2025 Batch 3 Programming Course completion.

The system demonstrates the basic Java programming competencies through implementation of a banking System using Object-Oriented Programming principles, Swing GUI, and relational database integration.

## Primary Objectives

* Demonstrate mastery of Java Swing GUI development for course completion
* Implement role-based access control mechanisms as practical application
* Showcase database connectivity and transaction management skills
* Apply Model-View-Controller (MVC) architectural pattern in real-world context
* Exhibit professional software development practices acquired during the course

## Target Audience

This technical documentation serves as a reference for:

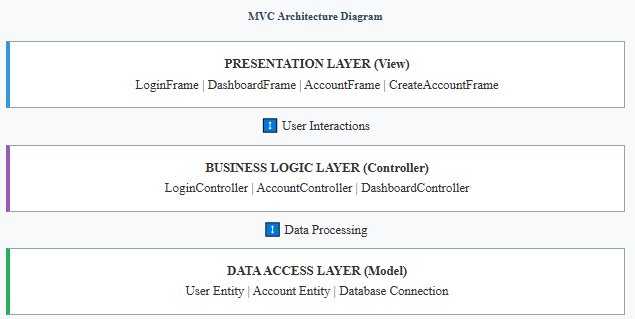
* Java NCIII course instructors evaluating course completion projects
* Academic evaluators assessing course completion requirements

# System Architecture

## Architectural Pattern

The application follows the Model-View-Controller (MVC) architectural pattern, ensuring clear separation of concerns and maintainable code structure.

**MVC Architecture Diagram**



## Folder Structure

BankingSystem/

├── Main.java

├── README.md

├── UserRoles.xlsx

├── bank.db

├── controller

│ ├── AccountController.java

│ ├── DashboardController.java

│ └── LoginController.java

├── lib

│ ├── javafx.base.jar

│ ├── javafx.controls.jar

│ ├── javafx.fxml.jar

│ ├── javafx.graphics.jar

│ └── sqlite-jdbc-3.50.3.0.jar

├── model

│ ├── Account.java

│ ├── Database.java

│ └── User.java

└── view

├── AccountFrame.java

├── ChangePasswordFrame.java

├── CreateAccountFrame.java

├── DashboardFrame.java

└── LoginFrame.java

## Technology Stack

* Programming Language: Java
* GUI Framework: Java Swing
* Database System: SQLite
* Database Connectivity: JDBC (Java Database Connectivity)
* Architecture Pattern: Model-View-Controller (MVC)
* Build System: Manual compilation with classpath dependencies

# Functional Specifications

## User Role Management System

### Role Hierarchy and Permissions

The system implements a four-tier role-based access control mechanism:

| **User Role** | **Create Account** | **Modify Password** | **Deposit Funds** | **Withdraw Funds** | **Balance Inquiry** | **Account Operations** | **System Logout** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Customer | **❌** | **❌** | **❌** | **❌** | **✅** | **❌** | **✅** |
| Teller | **✅** | **❌** | **✅** | **✅** | **✅** | **✅** | **✅** |
| Manager | **✅** | **✅** | **❌** | **❌** | **✅** | **✅** | **✅** |
| Admin | **✅** | **✅** | **✅** | **✅** | **✅** | **✅** | **✅** |

### Test System Credentials

|  |  |  |
| --- | --- | --- |
| Login | Password | Role |
| admin | admin | Admin |
| qqqq | 1111 | Customer |
| admin1 | Admin1@1 | Admin |
| zzzz | Zzzz1111@ | Manager |
| xxxx | Xxxx1111@ | Teller |

## Core Banking Operations

### Account Management Functions

* **Account Creation**: Automated account generation during first deposit transaction
* **Balance Inquiry**: Real-time account balance retrieval with currency formatting
* **Transaction Processing**: Deposit and withdrawal operations with validation

### User Management Functions

* **User Authentication**: Credential verification against database records
* **User Registration**: New user account creation with role assignment
* **Session Management**: Secure session handling with user context preservation

# Technical Implementation

## Database Schema Design

The system utilizes a normalized relational database schema with two primary entities:

-- User Authentication Entity CREATE TABLE users ( id INTEGER PRIMARY KEY AUTOINCREMENT, username TEXT UNIQUE NOT NULL, password TEXT NOT NULL, role TEXT NOT NULL );

-- Banking Account Entity CREATE TABLE accounts ( id INTEGER PRIMARY KEY AUTOINCREMENT, owner TEXT UNIQUE NOT NULL, balance REAL NOT NULL DEFAULT 0.0 );

## Security Implementation

### SQL Injection Prevention

All database interactions utilize parameterized queries through PreparedStatement objects:

PreparedStatement ps = connection.prepareStatement( "SELECT role FROM users WHERE username=? AND password=?" ); ps.setString(1, sanitizedUsername); ps.setString(2, sanitizedPassword);

### Input Validation Framework

* Client-side Validation: Immediate user feedback for invalid inputs
* Server-side Validation: Business logic validation in controller layer
* Data Type Validation: Numeric format verification for monetary amounts
* Required Field Validation: Empty field detection and user notification

## Exception Handling Strategy

### Database Exception Management

try (

Connection connection = Database.connect();

PreparedStatement statement = connection.prepareStatement(sql)

) {

// Execute database operations here

} catch (SQLException exception) {

exception.printStackTrace();

return new TransactionResult(false, "Database error occurred", 0.0);

}

### User Interface Exception Handling

* **Graceful Error Recovery**: System continues operation despite non-critical errors
* **User-Friendly Messages**: Technical exceptions translated to comprehensible messages
* **Resource Cleanup**: Automatic resource deallocation using try-with-resources pattern

# User Interface Design

## Login Screen

A screenshot of a computer

AI-generated content may be incorrect.

## Dashboard

### Dashboard - Admin

A screenshot of a computer

AI-generated content may be incorrect.

### Dashboard – Manager

A screenshot of a computer

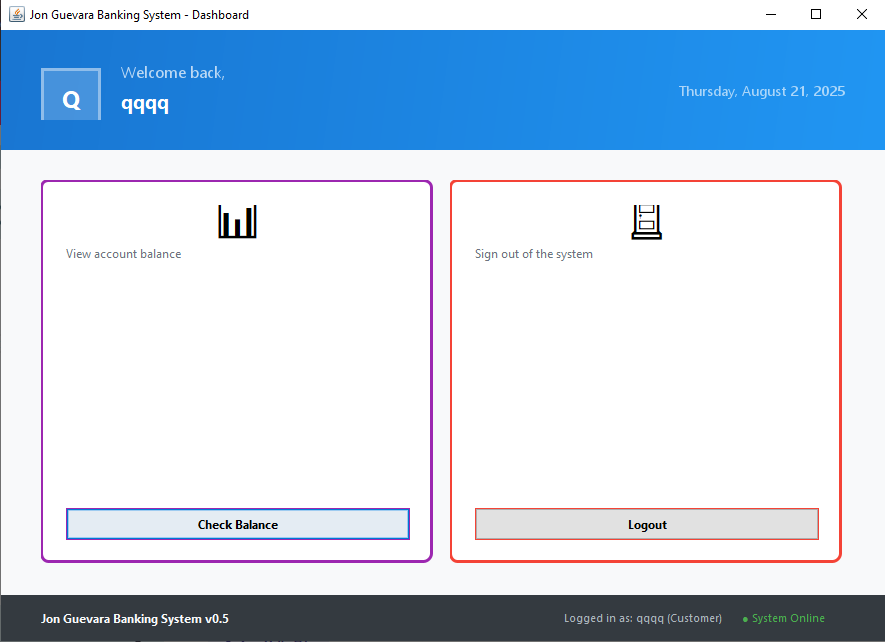
AI-generated content may be incorrect.

### Dashboard – Teller

A screenshot of a computer

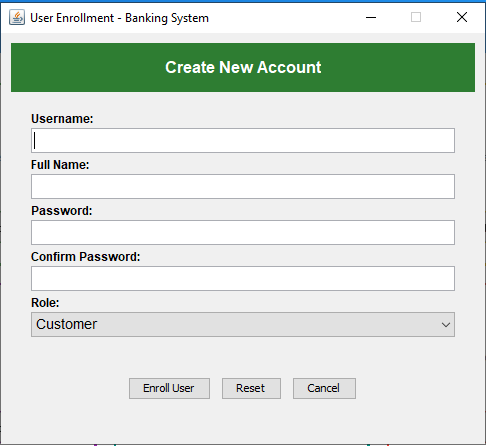
AI-generated content may be incorrect.

### Dashboard – Customer

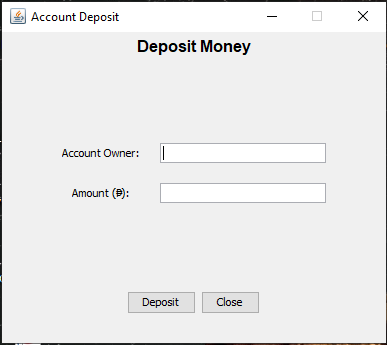


## User Interface

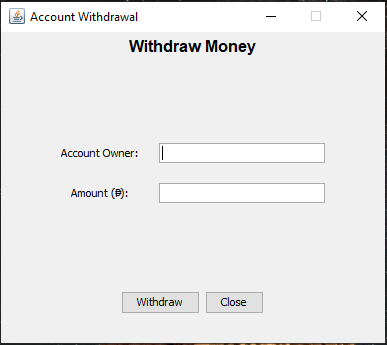
### Create New Account



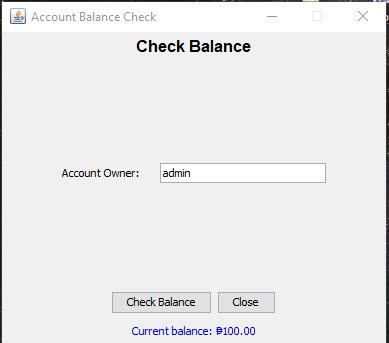
### Deposit Money



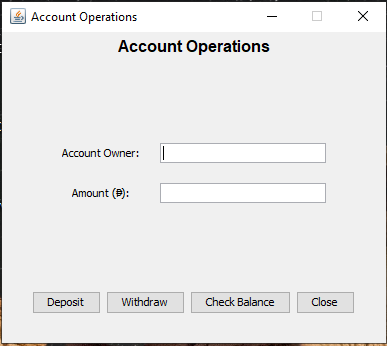
### Withdraw Money



### Check Balance



### Account Operation



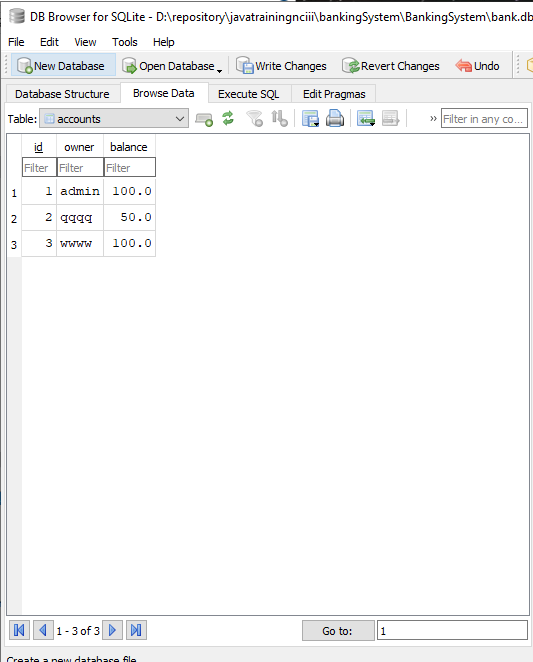
## Database View

### Table - Users

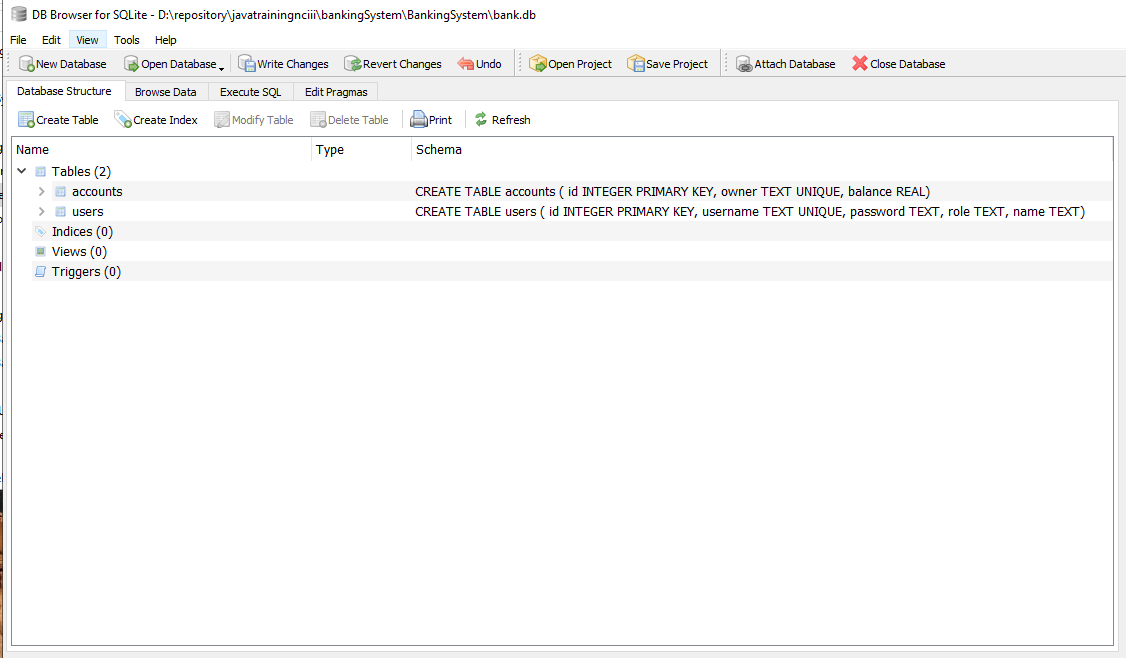
A screenshot of a computer

AI-generated content may be incorrect.

### Table - Account



### Schema



# Data Flow Architecture

## Application Initialization Sequence

A diagram of a process

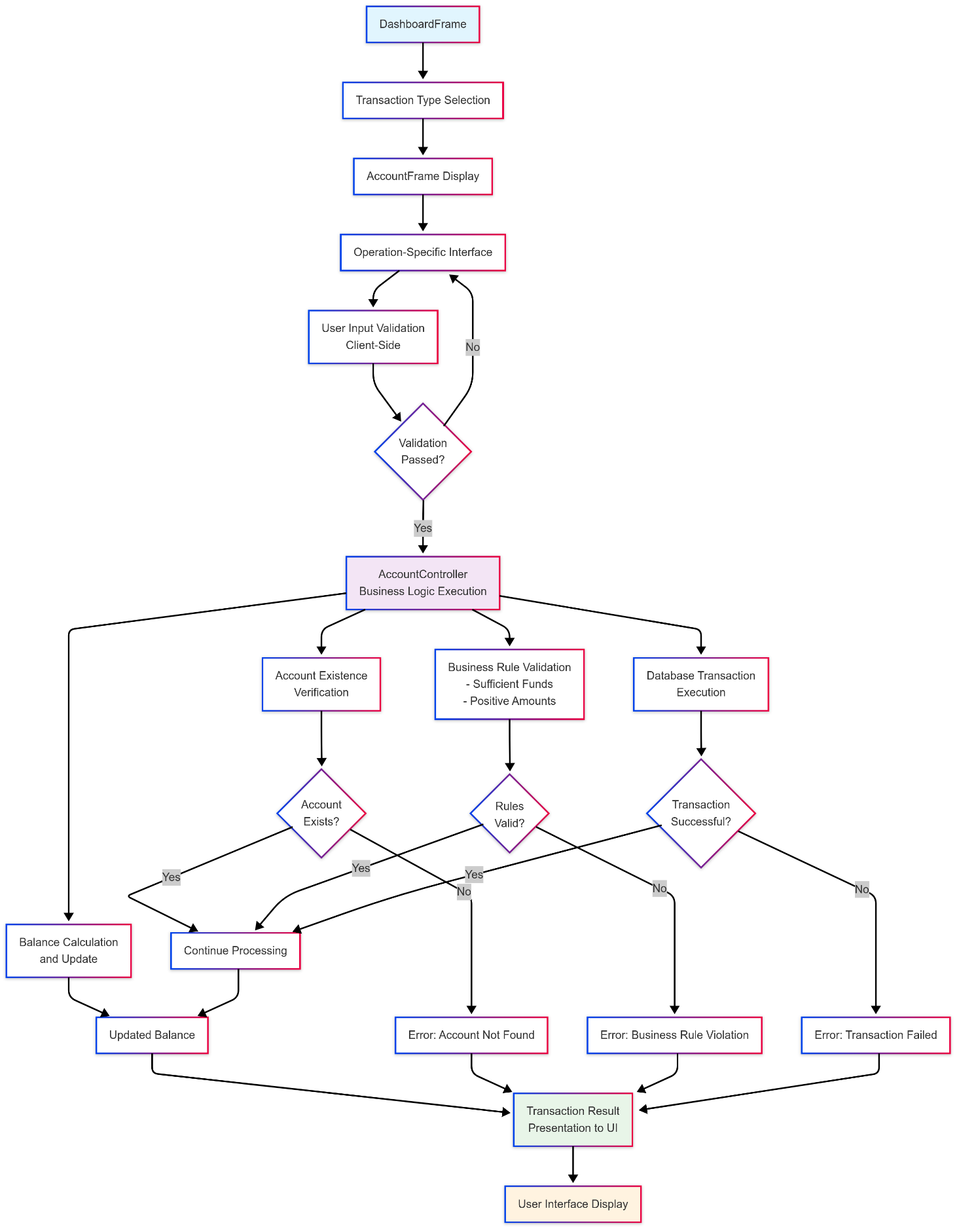
AI-generated content may be incorrect.

## 6.2 Authentication Process Flow

A diagram of a computer

AI-generated content may be incorrect.

## 6.3 Transaction Processing Workflow



# Java NCIII Course Completion Competency

## Object-Oriented Programming Mastery

### Encapsulation Implementation

* **Private Field Access**: All entity fields declared with private visibility modifiers
* **Controlled Access Methods**: Public getter methods providing controlled field access
* **Data Validation**: Setter methods implementing business rule validation
* **Information Hiding**: Internal implementation details concealed from client code

### Composition and Aggregation

* **Controller-Model Relationships**: Controllers utilize model classes for data operations
* **View-Controller Dependencies**: User interface components delegate to controller objects
* **Loose Coupling**: Interface-based communication between architectural layers

### Static Method Utilization

* **Utility Methods**: Stateless operations implemented as static methods
* **Factory Patterns**: Database connection creation through static factory methods
* **Helper Functions**: Currency formatting and validation utilities

## Database Integration

### JDBC API Utilization

* **Connection Management**: Proper database connection lifecycle management
* **PreparedStatement Usage**: Parameterized query execution for security
* **ResultSet Processing**: Efficient data retrieval and processing
* **Transaction Management**: Atomic operation handling with rollback capabilities

### SQL

* **Data Definition Language (DDL)**: Table creation with appropriate constraints
* **Data Manipulation Language (DML)**: INSERT, UPDATE, SELECT operation expertise
* **Query Optimization**: Efficient query structure for optimal performance
* **Data Integrity**: Constraint enforcement and referential integrity maintenance

# System Requirements and Deployment

## Technical Prerequisites

* Java Runtime Environment: JRE
* Operating System: Cross-platform compatibility (Windows, Linux, macOS)
* Memory Requirements: Minimum 512MB RAM, recommended 1GB RAM
* Storage Requirements: 50MB disk space for application and database files

## Installation and Configuration

### Compilation Process

1. Navigate to project root directory cd BankingSystem/
2. Compile all Java source files with SQLite dependency

javac -cp ".;lib\sqlite-jdbc-3.50.3.0.jar" model\\*.java controller\\*.java view\\*.java Main.java

1. Execute application with proper classpath configuration

java -cp ".;lib\sqlite-jdbc-3.50.3.0.jar" Main

### Database Initialization

* **Automatic Schema Creation**: Database tables created automatically on first execution
* **Default Data Population**: Administrative user account created during initialization
* **File Location**: Database file (bank.db) created in application root directory
* **Backup Considerations**: Database file can be copied for backup purposes

\*\*\*\* END OF DOCUMENT \*\*\*\*