# BANK ACCOUNT MANAGEMENT SYSTEM

**CONTEXT OF PROJECT**

-Provides an interface for users to interact with their accounts.

-Handles the core functionality like processing transactions, updating balances, and managing data.

-Can be built using languages like Python, Java, or c#.

## OBJECTIVE OF PROJECT

**Primary Objective:**

- Develop a user-friendly banking application that allows members to manage their checking and savings accounts, including viewing account balances, making deposits, and withdrawing funds.

**Secondary Objective:**

- Ensure all transactions are securely logged and stored in a database, with data integrity and reliability.

## 3)SCOPE OF THE PROJECT

a). Core Features

- User Management:
- Create and store user profiles (members) with basic information such as name and email.
- Link each user to a checking account and a savings account.
- Account Management:
- Each user will have two types of accounts: a checking account and a savings account.
- Users can view account information, including account type, balance, and transaction history.
- Transaction Management:

- Deposits: Users can deposit money into their checking or savings accounts.
- Withdrawals: Users can withdraw money from their checking or savings accounts, provided they have sufficient funds.
- Balance Inquiry: Users can check the current balance of their accounts.
- Data Persistence:
- All transactions and account balances will be stored in a database (e.g., MSSQL SERVER).
- Transaction history will be maintained for auditing and review.

b). Technical Scope

- Backend Development:
- Implemented using C#, handling core business logic like creating accounts, managing transactions, and interacting with the database.
- Database Management:
- Use a relational database (MSSQL SERVER) to store user profiles, account details, and transaction records.
- Ensure the database schema includes tables for members, accounts, and transactions.
- Interface:
- A command-line interface (CLI) for user interaction.
- Optional: Expand to a web interface using Flask if required.
- Error Handling:
- Implement basic error handling, such as ensuring deposits are positive amounts and preventing withdrawals that exceed account balances.
- Security:
- Basic security considerations, like ensuring transactions are accurately logged and users can only access their own account information.

c). Project Scope Management

- Deliverables:
- A C# .NET application with a CLI interface.
- A relational database schema with tables for members, accounts, and transactions.
- Documentation detailing how to set up, run, and use the application.

- Timeline:
- The project could be developed and tested over a few weeks, depending on the developer's experience and the specific requirements.

## FUNCTIONAL ASPECT OF THE PROJECT

### a) User Authentication

- Members should be able to log in using a secure authentication method (e.g., username and password, multi-factor authentication).

### b) View Account Information

- Display account details such as account number, type (checking/savings), and current balance.

### c) Perform Transactions

- Deposits: Allow members to deposit money into their checking or savings accounts.
- Withdrawals: Allow members to withdraw money from their checking or savings accounts, with checks for sufficient balance.
- Transfer between Accounts: Optionally, allow transfers between checking and savings accounts.

### d) Transaction History

- Maintain a record of all transactions for each account.
- Display transaction history to the user upon request.

### e) Exit and Save

- When the user exits the application, ensure all transactions are saved to the database.

- Ensure data integrity by implementing rollback mechanisms if saving fails.

## NON-FUNCTIONAL ASPECT OF THE PROJECT

### a) Performance

- Response Time: The application should provide quick responses to user actions, such as logging in, viewing account information, and processing transactions. Ideally, all operations should be completed within 2-3 seconds.
- Throughput: The system should be able to handle multiple transactions per second, especially during peak usage times.
- Scalability: The application should be scalable to accommodate an increasing number of users and transactions without significant degradation in performance.

### b) Security

- Data Protection: Sensitive data, including user credentials and account information, must be encrypted both at rest and in transit.
- Authentication and Authorization: The system must ensure that only authorized users can access their accounts and perform transactions. Implement strong password policies and possibly account lockout mechanisms after repeated failed login attempts.
- Audit and Logging: The application should log all significant events (e.g., login attempts, transactions) for audit purposes. Logs should be protected from tampering and stored securely.

### c) Usability

- User Interface: The interface should be intuitive, user-friendly, and accessible to all users, including those with disabilities. This includes clear navigation, well-labelled buttons, and informative error messages.
- Ease of Use: The system should require minimal training or guidance to use, with consistent and simple workflows for all operations.
- Accessibility: The application should comply with accessibility standards (e.g., WCAG) to ensure that users with disabilities can effectively use the system.

### d) Reliability

- Availability: The application should have high availability, ideally 99.9% uptime, ensuring that users can access their accounts and perform transactions almost all the time.
- Fault Tolerance: The system should be able to handle failures gracefully, with mechanisms in place for data recovery in case of system crashes or unexpected errors.
- Backup and Recovery: Regular backups of the database should be performed to prevent data loss. The system should support efficient recovery procedures in case of data corruption or loss.

e) Portability

- Cross-Platform Compatibility: If the application has a frontend, it should be compatible across different platforms (e.g., web browsers, mobile devices) and operating systems (e.g., Windows, macOS, iOS, Android).
- Deployment Flexibility: The application should be easy to deploy on different environments, whether on-premise or in the cloud.

## TECHNICAL ASPECTS OF THE PROJECT

- Development Environment:
- IDE or Text Editor: Tools like Visual Studio for writing and editing code, debug, and build code, and then publish an app.

- Version Control System: Git and GitHub for managing code versions and collaboration.
- Database Management System (DBMS):

- MSSQLSERVER: For a lightweight, file-based database, suitable for small applications.


- MySQL: For a more robust, scalable database solution if the application grows.

**NECESSARY RESOURCES**

a) Human Resources

- Software Developer(s):
- Responsible for designing, coding, testing, and deploying the application.
- Proficient in .NET C#.
- Database Administrator (DBA):
- Manages the database design, optimization, backup, and security.
- Ensures the database is properly integrated with the application.
- Project Manager:
- Oversees the project timeline, budget, and resource allocation.
- Coordinates between different teams and stakeholders.
- QA Engineer/Tester:
- Tests the application to ensure it meets the requirements and is free of bugs.
- Performs both manual and automated testing.
- UI/UX Designer (optional):
- If a more sophisticated interface is desired, a designer can create wireframes, mock-ups, and user flows.
- Ensures the application is user-friendly and visually appealing.

b) Physical Resources

- Computers/Servers:
- Developer machines with sufficient processing power and memory.
- A server for hosting the application if it’s not local or cloud-based.
- Backup Storage:
- External hard drives or cloud storage solutions like Google Drive or AWS S3 for backing up the database and code.