SRS Document

Banking System Application

A Web-Based Application

Nihal Badiger | Shailesh G. | Pranav Kumar | Anshaj Gupta

Contents

1. INTRODUCTION

Overview

The purpose of this Software Requirements Specification (SRS) document is to provide a detailed description of the Banking Application. The application is designed to facilitate basic banking operations such as user registration, login, deposits, withdrawals, fund transfers, and transaction history management. The document outlines the functional and non-functional requirements of the system, ensuring that the development team and stakeholders have a clear understanding of the project's scope and objectives.

This SRS serves as a foundation for the development, testing, and deployment phases of the project, ensuring that all requirements are met efficiently and effectively.

Scope

The Banking Application **is a web-based system built using the** Flask framework **and a** SQLite database**.** The application is targeted at individual users who want to manage their personal finances. Key functionalities of the system include:

- ✓ **User Registration**: Users can create an account by providing essential information like username, email, phone number, and password.
- ✓ **User Login**: Registered users can securely log into the application.
- ✓ **Dashboard**: A personalized dashboard that displays user account details and balance.
- ✓ **Deposit**: Users can add funds to their account balance.
- ✓ **Withdraw**: Users can withdraw funds from their account, ensuring they have sufficient balance.
- ✓ **Transfer**: Users can transfer funds to other registered users by specifying the recipient's username and account number.
- ✓ **Transaction History**: Users can view a detailed history of their transactions (deposits, withdrawals, and transfers).
- ✓ **Download Transaction History**: Users can export their transaction history in Excel format.
- ✓ **Logout**: Users can securely log out of the system.

All user data and transactions are securely stored in the SQLite database. This project is ideal for small-scale banking solutions or personal financial management.

Assumptions and Dependencies

The following assumptions and dependencies are considered during the development and deployment of the Banking Application:

Type	Description
Technological Assumptions	• The application will be hosted on a server capable of running Python 3.9+. • The application will rely on the Flask framework for the backend. • The frontend will use HTML and CSS for user interface design. • The database will be SQLite, which is suitable for lightweight, single-user applications.

User Assumptions	• Users will have a basic understanding of web applications. • Users will provide valid input for registration and transactions (e.g., usernames, account numbers, and amounts). • Users will use secure credentials for authentication.
Dependencies	• The system depends on the availability of required Python libraries, including Flask, Flask-SQLAlchemy, pandas, and openpyxl. • Internet access is required for the initial deployment and installation of dependencies. • The application assumes a stable hosting environment (e.g., Dockerized environment, local development server, or cloud hosting).
Performance Assumptions	• The application assumes moderate usage with limited concurrent users, as SQLite is not designed for high scalability. • The system will function optimally for personal banking purposes with a small number of transactions.
Security Assumptions:	• Passwords will be securely hashed and stored using *werkzeug.security.* • Users will log out properly to prevent unauthorized access. • Data validation will ensure proper inputs are processed, mitigating errors or misuse.

2. SYSTEM OVERVIEW

The Banking Application is a lightweight, web-based financial management system developed using the Flask framework. It enables users to securely perform fundamental banking operations, ensuring a smooth and user-friendly experience. The system is designed with simplicity and efficiency in mind, making it suitable for personal finance management or small-scale banking use cases.

Presentation Layer (Frontend)
Built using HTML and CSS.
Provides a responsive user interface for interacting with the application.
Users access the system through a web browser.

Application Layer (Backend)
Developed using the Flask framework (Python).
Handles all business logic, routing, and user authentication.
Processes user inputs, manages session data, and communicates with the database.

Data Layer (Database)
Uses SQLite as the database for lightweight data storage.
Stores user information, account details, and transaction records securely.

3. FUNCTIONAL REQUIREMENTS

Function	Description
User Registration	Users can create an account by providing a username, email, phone, and password.
Password Hashing	User passwords are securely hashed before storing them in the database.
User Login	Users can log in using valid credentials (username and password).
Account Number Generation	The system generates a unique account number for each user.
View Dashboard	Users can view their account balance and basic details on the dashboard.
Deposit Funds	Users can add funds to their account balance.
Withdraw Funds	Users can withdraw funds if they have sufficient balance.
Transfer Funds	Users can transfer funds to other registered users.
Transaction Logging	Every deposit, withdrawal, and transfer is logged in the system.
View Transaction History	Users can view all transactions related to their account.
Download Transaction History	Users can download transaction history as an Excel file.
Logout Functionality	Users can securely log out, ending their session.

4. NON-FUNCTIONAL REQUIREMENTS

Function	Description
Performance	• API Response Time: The system should respond to prediction requests within 1 second under normal load. • Concurrent Requests: The system should handle at least 100 concurrent requests without performance degradation. • Scalability: The architecture must support scaling to handle larger datasets and higher user traffic.
Security	• Authentication: Secure the API using authentication methods Data Privacy: Ensure no sensitive data is exposed. Use HTTPS for secure communication. • Input Validation: Prevent malicious data input through strict validation rules.
Usability	• Ease of Use: Provide a user-friendly UI interface for testing and interacting with the APIs. • Clear Documentation: All endpoints and features should be well documented for seamless user understanding.
Reliability and Availability	• Uptime: The system must have a 99% uptime to ensure consistent availability.

	• Error Handling: Provide meaningful error messages and ensure the system gracefully recovers from unexpected failures.
Maintainability	• Modularity: The code base should follow modular design principles to facilitate updates and debugging. • Documentation: Include detailed inline comments and external documentation for easier maintenance.
Portability	• Platform Independence: The system should be deployable on any platform supporting Python, Flask, and relevant dependencies.

5. SYSTEM ARCHITECTURE

Technologies Used

Component	Technology	Description
Front End	HTML5, CSS3	Creates a responsive and user-friendly interface for users.
Backend	Flask (Python Framework)	Handles routing, business logic, and server-side processing.
Database	SQLite	Lightweight relational database for storing user and transaction data.
ORM	Flask-SQLAlchemy	Simplifies interaction with the database using Python objects.
Security	Werkzeug Security	Provides secure password hashing and validation.
Data Export	pandas, openpyxl	Facilitates exporting transaction history as Excel files.
Web Server	Flask Development Server	Used for local development and testing.
Deployment	Docker	Containerizes the application for seamless deployment

Data Flow

The **data flow** of the application can be broken down into the following steps:

User Registration:

- ✓ A new user registers by providing necessary credentials.
- ✓ An account number is auto-generatd incrementally.

User Login:

- ✓ Users log in using their username and password.
- ✓ Secure session management is established upon successful login.

Perform Banking Operations:

- ✓ Deposit Funds: Users can add money to their account.
- ✓ Withdraw Funds: Users can withdraw funds, ensuring sufficient balance.
- ✓ Transfer Funds: Users can send funds to other registered users.
- ✓ Transaction Logging: All transactions are recorded for future reference.

Transaction History and Export:

- ✓ Users can view their transaction history.
- ✓ Transaction history can be downloaded as an Excel file.

Logout:

- ✓ Users can log out, terminating the session securely.

Database Design

USER

Field	Type	Constraints	Description
Id	INTEGER	Primary Key, Auto-increament	Unique identifier for each user.
Username	STRING(80)	Unique, Not Null	User's unique username.
Email	STRING(120)	Unique, Not Null	User's email address.
Password	STRING(200)	Not Null	Hashed user password
Phone_number	STRING(10)	Unique, Not Null	User's phone number
Account_number	STRING(20)	Unique, Not Null	User's unique account number
Balance	Float	Default: 0	Current account balance
Status	STRING (20)	Default: 'Active'	Account Status
Account_type	STRING(20)	Default: 'Checking'	Type of User Account

TRANSACTION

Field	Type	Constraints	Description
Id	INTEGER	Primary Key, Auto-increament	Unique transaction identifier.
Sender_id	INTEGER	Foreign Key, Not Null	Reference to the sender's user ID.
Recipient_id	INTEGER	Foreign Key, Not Null	Reference to the recipient's user ID.
Amount	FLOAT	Not Null	Transaction amount
Timestamp	DATETIME	Default: Current Time	Timestamp of when the transaction occured

API Interface

Endpoint	Method	Parameter	Request Body	Response	Description
/api/register	POST	None	{ username, email, phone, password }	{ success: true, message: "Account created" }	Registers a new user and creates an account.
/api/login	POST	None	{ username, password }	{ token: "auth_token", user_id }	Logs in the user and returns an auth token.
/api/balance	GET	user_id	None	{ balance: 1000.0 }	Retrieves the user's current account balance.
/api/deposit	POST	Authorization (Header)	{ amount }	{ success: true,	Deposits a specified amount

				new_balance: 1300 }	into the account.
/api/withdraw	POST	Authorization (Header)	{ amount }	{ success: true, new_balance: 700 }	Withdraws funds if sufficient balance exists.
/api/transfer	POST	Authorization (Header)	{ recipient_account, amount }	{ success: true, message: "Transfer successful" }	Transfers funds to another user's account.
/api/transaction_history	GET	user_id, Authorization (Header)	None	[{ id, sender, recipient, amount, timestamp }]	Retrieves all transactions related to the user.
/api/export_transactions	GET	Authorization (Header)	None	Excel File	Exports the user's transaction history as an Excel file.

6. ENTITY RELATIONSHIP DIAGRAM