



Bhartiya Vidya Bhavan's
Sardar Patel Institute of Technology
(Autonomous Institute Affiliated to University of Mumbai)
Department of Computer Engineering

Bank Management System

By
Adityavardhan Iyengar(2024300084)
Tattva Jain (2024300093)
Soham Jagushte(2024300091)

Guided by
Prof Jotsna Bhagat

Course Project

Database Management System (S.Y.2025-2026)

ABSTRACT

The SecureBank Online Banking System is a full-stack, database-driven application designed to simulate real-world banking operations with an emphasis on security, data integrity, and efficient user interaction. The project integrates MySQL for structured financial data management, Node.js/Express.js for backend API logic, and a responsive HTML–CSS–JavaScript interface for end users and administrators.

The database architecture includes core tables such as **users**, **accounts**, **transactions**, **audit_log**, and **bank_staff**, all maintained through strict relational integrity using primary and foreign keys. Key DBMS concepts—including **DML operations**, **stored procedures**, **views**, **triggers**, and **transaction control (COMMIT, ROLLBACK, FOR UPDATE)**—are applied to support secure authentication, account management, withdrawals, deposits, and administrative workflows.

Backend connectivity is implemented using a **MySQL connection pool**, environment-secured credentials, parameterized queries, and modular routing. Real-time financial operations ensure atomicity, while audit logging provides transparency and regulatory-level accountability. The front-end adopts a clean and responsive design, allowing customers and administrators to perform tasks such as viewing accounts, requesting loans, managing services, and handling approvals without page reloads.

Overall, SecureBank demonstrates a comprehensive application of DBMS principles within a modern web development environment, delivering a secure, scalable, and user-friendly online banking platform.

TABLE OF CONTENTS

Chapter 1: Creation of Database

- 1.1 Creating the Database
- 1.2 Creating the Users Table
- 1.3 Creating the Accounts Table
- 1.4 Creating the Transactions Table
- 1.5 Creating the Audit Log Table
- 1.6 Creating the Bank Staff Table
- 1.7 Stored Procedure: Secure User Creation
- 1.8 Stored Procedure: Money Transfer
- 1.9 Assigning Privileges
- 1.10 Creating Indexes

Chapter 2: Data Manipulation

- 2.1 Creating a New User
- 2.2 Retrieving Login Information
- 2.3 Creating a Bank Account
- 2.4 Retrieving User Accounts
- 2.5 Depositing Money
- 2.6 Recording Deposit Transaction
- 2.7 Withdrawing Money
- 2.8 Transfer via Stored Procedure
- 2.9 Viewing All Transactions
- 2.10 Admin Updating Staff Role
- 2.11 Logging Critical Actions
- 2.12 Deleting a User Account

Chapter 3: Operations on Database

- 3.1 Ensuring Unique & Secure User Creation
- 3.2 Dynamic Account Operations
- 3.3 Money Transfer with Transaction Control
- 3.4 Maintaining Audit Logs
- 3.5 Automatic Cleanup with Foreign Keys

Chapter 4: Views

- 4.1 Purpose of Views
- 4.2 Creating Account Overview View
- 4.3 Benefits of Views

Chapter 5: Triggers

- 5.1 Purpose of Triggers
- 5.2 Trigger for Transaction Logging
- 5.3 Trigger Example
- 5.4 Advantages of Triggers

Chapter 6: Database Connectivity

- 6.1 Objectives
- 6.2 Installing & Importing Drivers
- 6.3 Configuring the Connection Pool
- 6.4 Using Environment Variables
- 6.5 Executing Parameterized Queries
- 6.6 Transaction Management (Commit/Rollback)
- 6.7 Error Handling & Stability

Chapter 7: Front-End

- 7.1 Objective of Front-End Design
- 7.2 Technology Stack Used
- 7.3 Application Structure
- 7.4 Core Components
- 7.5 Styling with CSS

Chapter 1:

CREATION OF DATABASE

1. CREATION OF DATABASE

A database forms the backbone of any banking or financial management application.

In the **Bank Management System**, the database named **bank_management** stores essential information including user credentials, bank accounts, financial transactions, audit logs, and staff roles.

This chapter provides a detailed explanation of the SQL queries used to construct the entire system database. Each subsection contains code followed by explanatory theory, matching the format of the sample provided.

1.1 Creating the Database

The following commands create the database that will contain all tables, indexes, stored procedures, and user privileges.

It ensures that all banking-related data is organized under a single logical structure.

```
CREATE DATABASE IF NOT EXISTS bank_management;
USE bank_management;
```

Explanation

This query checks if the database already exists to prevent duplication and then selects it as the working database. All subsequent table creation and data manipulation commands will belong to this database.

1.2 Creating the Users Table

The **users** table is the core entity for authentication.

It stores secure user login credentials along with profile details and timestamps.

```
CREATE TABLE users (
    user_id INT PRIMARY KEY AUTO_INCREMENT,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    full_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
```

```
    phone VARCHAR(20),  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    last_login TIMESTAMP  
);
```

Explanation

This table stores sensitive user information:

- **username & email** are unique to avoid duplication.
 - **password** is stored in hashed form.
 - **created_at** and **last_login** help track account creation and login activity.
 - This table forms the foundation for all user-accessible modules in the system.
-

1.3 Creating the Accounts Table

Bank accounts are linked to users, enabling savings, checking, and fixed deposit management.

```
CREATE TABLE accounts (  
    account_id INT PRIMARY KEY AUTO_INCREMENT,  
    user_id INT,  
    account_type ENUM('savings', 'checking', 'fixed_deposit') NOT  
NULL,  
    account_number VARCHAR(20) UNIQUE NOT NULL,  
    balance DECIMAL(15, 2) DEFAULT 0.00,  
    interest_rate DECIMAL(5, 2),  
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,  
    FOREIGN KEY (user_id) REFERENCES users(user_id)  
);
```

Explanation

This table ensures:

- Each bank account belongs to exactly **one user** (via foreign key).
 - Balance and interest rate fields track financial values.
 - Unique **account_number** ensures no duplication.
 - Supports multiple account types used in real banking systems.
-

1.4 Creating the Transactions Table

Every money movement is recorded in this table.

```
CREATE TABLE transactions (
    transaction_id INT PRIMARY KEY AUTO_INCREMENT,
    account_id INT,
    transaction_type ENUM('deposit', 'withdrawal', 'transfer') NOT
NULL,
    amount DECIMAL(15, 2) NOT NULL,
    description TEXT,
    transaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (account_id) REFERENCES accounts(account_id)
);
```

Explanation

This table ensures a complete financial trail:

- Deposits, withdrawals, and transfers are all logged.
 - Every transaction is tied to an account for traceability.
 - Timestamping allows chronological reporting and security audits.
-

1.5 Creating the Audit Log Table

Audit logs record sensitive actions to enhance security and accountability.

```
CREATE TABLE audit_log (
    log_id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT,
    action VARCHAR(100) NOT NULL,
    details TEXT,
    ip_address VARCHAR(45),
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Explanation

This table tracks:

- Who performed an action
- What action was taken
- From which IP address
- At what time

Audit logging is essential for regulatory compliance in financial institutions.

1.6 Creating the Bank Staff Table

This table tracks employees such as admins, tellers, and managers.

```
CREATE TABLE bank_staff (
    staff_id INT PRIMARY KEY AUTO_INCREMENT,
    user_id INT,
    role ENUM('admin', 'manager', 'teller') NOT NULL,
    department VARCHAR(50),
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Explanation

- Staff accounts are connected to user accounts for authentication.
 - Role-based access ensures secure internal operations.
 - Departments allow organizational structure tracking.
-

1.7 Stored Procedure: Secure User Creation

Stored procedures reduce repeated code and ensure secure user handling.

```
DELIMITER //

CREATE PROCEDURE create_user(
    IN p_username VARCHAR(50),
    IN p_password VARCHAR(255),
    IN p_full_name VARCHAR(100),
    IN p_email VARCHAR(100),
    IN p_phone VARCHAR(20)
)
BEGIN
    INSERT INTO users (username, password, full_name, email, phone)
    VALUES (p_username, p_password, p_full_name, p_email, p_phone);
END //

DELIMITER ;
```

Explanation

This stored procedure standardizes how users are added to the system. It ensures consistency and reduces the risk of SQL injection.

1.8 Stored Procedure: Money Transfer

```
DELIMITER //
```

```

CREATE PROCEDURE transfer_money(
    IN from_account_id INT,
    IN to_account_id INT,
    IN amount DECIMAL(15, 2)
)
BEGIN
    DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
        ROLLBACK;
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Transaction failed';
    END;

    START TRANSACTION;

    UPDATE accounts SET balance = balance - amount
    WHERE account_id = from_account_id
    AND balance >= amount;

    UPDATE accounts SET balance = balance + amount
    WHERE account_id = to_account_id;

    INSERT INTO transactions (account_id, transaction_type, amount,
    description)
    VALUES (from_account_id, 'transfer', -amount, 'Transfer sent'),
           (to_account_id, 'transfer', amount, 'Transfer received');

    COMMIT;
END //

DELIMITER ;

```

Explanation

This procedure ensures safe monetary transfer by:

- Using **transactions** for atomic updates.

- Preventing overdrafts.
 - Logging both the sending and receiving operations.
 - Automatically rolling back in case of failure.
-

1.9 Assigning Privileges

```
CREATE USER 'bank_admin'@'localhost' IDENTIFIED BY
  'strong_admin_password';
CREATE USER 'bank_staff'@'localhost' IDENTIFIED BY
  'strong_staff_password';
CREATE USER 'bank_app'@'localhost' IDENTIFIED BY
  'strong_app_password';

GRANT ALL PRIVILEGES ON bank_management.* TO 'bank_admin'@'localhost';

GRANT SELECT, INSERT, UPDATE ON bank_management.users TO
  'bank_staff'@'localhost';
GRANT SELECT, INSERT, UPDATE ON bank_management.accounts TO
  'bank_staff'@'localhost';
GRANT SELECT, INSERT ON bank_management.transactions TO
  'bank_staff'@'localhost';

GRANT SELECT, INSERT, UPDATE ON bank_management.users TO
  'bank_app'@'localhost';
GRANT SELECT, INSERT ON bank_management.transactions TO
  'bank_app'@'localhost';
GRANT SELECT, UPDATE ON bank_management.accounts TO
  'bank_app'@'localhost';

FLUSH PRIVILEGES;
```

Explanation

Privileges ensure **role-based access control**, crucial in banking systems.
Admins maintain full control, while staff and applications only receive limited, purpose-specific permissions.

1.10 Creating Indexes

```
CREATE INDEX idx_users_username ON users(username);  
CREATE INDEX idx_accounts_user_id ON accounts(user_id);  
CREATE INDEX idx_transactions_account_id ON transactions(account_id);  
CREATE INDEX idx_audit_log_user_id ON audit_log(user_id);
```

Explanation

Indexes dramatically improve query speed, especially for:

- Login lookups
 - Account searches
 - Transaction reports
 - Security audits
-

Chapter 2:

DATA MANIPULATION

2. DATA MANIPULATION

Data Manipulation Language (DML) operations power all dynamic functionalities of the Bank Management System.

This chapter explains how data is inserted, retrieved, updated, and deleted across modules such as user registration, account handling, transaction processing, and administrative operations.

Each section follows the same format as your sample:

code → explanation → code → explanation, with 3–4 pages of total content.

2.1 Creating a New User

```
INSERT INTO users (username, password, full_name, email, phone)
VALUES ('admin', 'hashed_password_here', 'System Administrator',
'admin@bankapp.com', '1234567890');
```

Explanation

When a user registers or an admin adds a new staff member, their details are stored securely. Passwords must always be hashed using bcrypt or Argon2 for protection.

2.2 Retrieving Login Information

```
SELECT user_id, username, password, full_name
FROM users
WHERE username = 'admin';
```

Explanation

During login, the system fetches user credentials and verifies the password using `bcrypt.compare()`.

This ensures secure authentication without exposing raw passwords.

2.3 Creating a Bank Account

```
INSERT INTO accounts (user_id, account_type, account_number, balance, interest_rate)
VALUES (1, 'savings', 'ACC1001', 5000.00, 4.5);
```

Explanation

A user may have multiple accounts.

The system generates unique account numbers and initializes balances and interest rates.

2.4 Retrieving User Accounts

```
SELECT account_id, account_type, balance, created_at
FROM accounts
WHERE user_id = 1;
```

Explanation

This query retrieves all accounts owned by a user to display on their dashboard.

2.5 Depositing Money

```
UPDATE accounts
SET balance = balance + 1000
WHERE account_id = 101;
```

Explanation

Deposits are straightforward balance updates.

A corresponding transaction entry is also created for record-keeping.

2.6 Recording the Deposit Transaction

```
INSERT INTO transactions (account_id, transaction_type, amount, description)
```

```
VALUES (101, 'deposit', 1000, 'User deposit');
```

Explanation

This preserves the financial trail and supports statements, reporting, and audits.

2.7 Withdrawing Money

```
UPDATE accounts  
SET balance = balance - 500  
WHERE account_id = 101 AND balance >= 500;
```

Explanation

The balance check ensures no overdraft occurs.
This is a crucial security measure in banking.

2.8 Performing a Transfer Using Stored Procedure

```
CALL transfer_money(101, 102, 2000);
```

Explanation

The stored procedure performs atomic deduct/add operations and logs both sides of the transfer into the transactions table.

2.9 Viewing All Transactions for an Account

```
SELECT transaction_type, amount, description, transaction_date  
FROM transactions  
WHERE account_id = 101  
ORDER BY transaction_date DESC;
```

Explanation

This query allows users and staff to review all financial actions in chronological order.

2.10 Admin Updating Staff Role

```
UPDATE bank_staff  
SET role = 'manager'  
WHERE staff_id = 2;
```

Explanation

Administrators can update staff privileges and internal roles to maintain operational workflow.

2.11 Logging Critical Actions

```
INSERT INTO audit_log (user_id, action, details, ip_address)  
VALUES (1, 'Updated Staff Role', 'Staff ID 2 promoted to manager',  
'192.168.1.10');
```

Explanation

Every sensitive action is recorded to support security audits and compliance requirements.

2.12 Deleting a User Account

```
DELETE FROM users WHERE user_id = 10;
```

Explanation

Deleting a user removes their access, but foreign keys ensure data integrity for related accounts and transactions.

Chapter 3:

OPERATIONS ON DATABASE

3. OPERATIONS ON DATABASE

The Bank Management System performs various essential database operations to ensure data correctness, security, and consistent financial processing. These operations include enforcing constraints, updating balances, performing secure transactions, and managing cascading relationships across multiple tables such as users, accounts, and transactions.

These SQL operations form the core business logic supporting real-time banking activities like fund transfers, account creation, and logging administrative actions.

3.1 Ensuring Unique and Secure User Creation

To prevent duplicate admin creation and preserve data integrity, the following INSERT operation uses:

- ON DUPLICATE KEY UPDATE
- UNIQUE constraints on username and email

```
INSERT INTO users (username, password, full_name, email, phone)
VALUES ('admin',
'$2b$10$5QqZk3.TcyoBR3aQjf5yj0sgHoZB7gz2wn3sUD1RJVmH09aV75n1i',
'System Administrator', 'admin@bankapp.com', '1234567890')
ON DUPLICATE KEY UPDATE username = username;
```

This ensures the admin user is created only once.

3.2 Dynamic Account Operations

The accounts table supports multiple banking products such as:

- savings
- checking
- fixed deposits

A typical query to retrieve all accounts belonging to a user:

```
SELECT account_id, account_type, account_number, balance
```

```
FROM accounts  
WHERE user_id = 1;
```

This allows displaying a user's bank portfolio in the dashboard.

3.3 Secure Money Transfer Using Transaction Control

The system uses a stored procedure to maintain atomicity during fund transfers.

```
CALL transfer_money(101, 202, 5000.00);
```

Internally, this ensures:

- balance deduction only if sufficient funds exist
- automatic rollback on failure
- transaction record creation
- This prevents partial or inconsistent updates in financial operations.

3.4 Maintaining Audit Logs for Security

To track critical actions (logins, transfers, admin operations):

```
INSERT INTO audit_log (user_id, action, details, ip_address)  
VALUES (1, 'login', 'Admin logged in', '127.0.0.1');
```

This supports monitoring, compliance, and forensic tracking.

3.5 Automatic Cleanup via Foreign Keys

To prevent orphaned data:

```
ALTER TABLE accounts  
ADD CONSTRAINT fk_user_delete  
FOREIGN KEY (user_id) REFERENCES users(user_id);
```

This ensures referential integrity across all banking modules.

Chapter 4:

VIEWS

4. VIEWS

Views provide simplified access to combined financial data without exposing sensitive details such as passwords. In the Bank Management System, views help administrators quickly analyze account and transaction information.

4.1 Purpose of Views in the System

Views are used to:

- Simplify complex JOIN queries
- Improve security by hiding internal fields
- Support reporting dashboards and analytics
- Provide real-time summarized information

4.2 Creating a View for Account and Transaction Monitoring

The following view consolidates accounts and recent activity:

```
CREATE VIEW v_account_overview AS
SELECT
    a.account_id,
    a.account_number,
    a.account_type,
    a.balance,
    u.full_name,
    u.email,
    t.transaction_type,
    t.amount,
    t.transaction_date
FROM accounts a
JOIN users u ON a.user_id = u.user_id
LEFT JOIN transactions t ON a.account_id = t.account_id;
```

Admins can now run:

```
SELECT * FROM v_account_overview;
```

This avoids repeating multiple joins in backend queries.

4.3 Benefits of Using Views

- Protects sensitive data (passwords never exposed)
- Improves performance on repeated queries
- Enables faster decision-making for staff
- Supports clean abstraction for dashboards

Chapter 5:

TRIGGERS

5. TRIGGERS

Triggers automate backend database operations when INSERT, UPDATE, or DELETE actions occur. In banking, triggers are crucial for maintaining security, transparency, and data accuracy.

5.1 Purpose of Triggers in the Banking System

Triggers help:

- Auto-record sensitive financial events
- Maintain audit and compliance logs
- Prevent manual oversight
- Synchronize account activities in real-time

5.2 Designing Trigger for Transaction Logging

Whenever a transaction is inserted, the system logs it automatically:

```
DELIMITER $$
```

```
CREATE TRIGGER log_transaction
AFTER INSERT ON transactions
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (user_id, action, details, ip_address)
    VALUES (
        (SELECT user_id FROM accounts WHERE account_id =
        NEW.account_id),
        'transaction',
        CONCAT('Transaction of ', NEW.amount, ' ',
        NEW.transaction_type),
        'system'
    );
END$$

DELIMITER ;
```

5.3 Trigger in Action Example

Running:

```
INSERT INTO transactions (account_id, transaction_type, amount, description)
VALUES (10, 'deposit', 5000, 'Initial deposit');
```

Automatically results in:

```
INSERT INTO audit_log (...)
```

No manual intervention required.

5.4 Advantages of Triggers

- Zero-error automation
- Real-time monitoring
- Enhanced transparency
- Regulatory compliance support

Chapter 6:

Database Connectivity

6. DATABASE CONNECTIVITY

Database connectivity forms the foundational communication layer between the backend components of the Bank Management System and the MySQL database. Every functional aspect—such as authentication, account management, withdrawal processing, administrative privileges, transaction logging, and dashboard analytics—depends on this stable database interface.

The system achieves this connectivity using a centralized configuration module that establishes a MySQL connection pool, loads environment-based security credentials, and exposes a unified interface for database operations across all backend modules.

6.1 Objective of Database Connectivity

The database connectivity layer in the system is structured to accomplish the following objectives:

- 1. To establish a secure and persistent communication channel between the backend and MySQL.**

This is implemented in services.js, which initializes and exports the database pool used by all modules.

- 2. To ensure optimized performance under concurrent requests through a connection pool.**

The connection pool in services.js enables multiple parallel operations such as account queries, transaction inserts, and dashboard counts.

- 3. To safeguard database credentials through environment variables.**

The credentials are loaded through server.js using the statement:
`require('dotenv').config();`

- 4. To provide consistent access to database operations across various routes.**

Modules such as accounts.js, admin.js, and dashboard.js all import the shared pool and execute parameterized queries.

- 5. To support multi-step, atomic financial operations through transaction controls.**

This is implemented in the withdrawal logic using row locking, commit, and rollback.

Code References

- services.js – Connection pool setup

- accounts.js, admin.js, dashboard.js – Modules using the shared pool
- server.js – Loads environment variables

6.2 Installing and Importing Database Drivers

The system employs the mysql2 library to establish communication with the MySQL server. The library is installed using:

```
npm install mysql2
```

Within services.js, the driver is imported:

```
const mysql = require('mysql2');
```

This import enables the backend to construct a connection pool and execute asynchronous SQL queries using the promise() interface.

Code Reference

- services.js, line importing MySQL driver

6.3 Configuring Connection Pool

The MySQL connection pool is configured centrally inside services.js.

This pool maintains multiple active connections, enabling efficient parallel query execution without repeatedly creating and closing connections.

```
const pool = mysql.createPool({  
  host: process.env.DB_HOST,  
  user: process.env.DB_USER,  
  password: process.env.DB_PASSWORD,  
  database: process.env.DB_NAME,  
});  
module.exports = pool.promise();
```

This exported pool is used throughout the backend for all database interactions involving users, accounts, transactions, and administrative data.

Code Reference

- services.js — Pool creation and export

6.4 Using Environment Variables for Security

To prevent exposure of sensitive database credentials in the codebase, the system relies on environment variables stored in a .env file.

These variables are loaded at runtime by:

```
require('dotenv').config();
```

(server.js)

The connection pool then accesses these variables while establishing the connection to the database:

```
host: process.env.DB_HOST,  
user: process.env.DB_USER,  
password: process.env.DB_PASSWORD,  
database: process.env.DB_NAME
```

(services.js)

This approach ensures portability between development, testing, and production environments.

Code References

- server.js – Loading environment variables
- services.js – Using environment variables in pool configuration

6.5 Executing Parameterized Queries Across Backend Modules

The system executes SQL operations through parameterized queries, which enhance security by preventing injection attacks and improve consistency across backend files.

Examples include:

In accounts.js:

```
const [accounts] = await pool.query(
  "SELECT balance FROM accounts WHERE account_id = ? AND user_id = ?",
  [accountId, req.user.id]
);
```

In admin.js:

```
await pool.query(
  "UPDATE users SET role = ? WHERE user_id = ?",
  [role, userId]
);
```

In dashboard.js:

```
const [rows] = await pool.query(
  "SELECT COUNT(*) AS total FROM transactions"
);
```

These queries demonstrate a uniform pattern of secure and efficient database interaction without exposing raw SQL values.

Code References

- accounts.js, admin.js, dashboard.js — Parameterized SQL queries

6.6 Transaction Management

The system includes explicit transaction control mechanisms for sensitive financial operations, such as withdrawals, where multiple dependent queries must succeed or fail as a single unit.

Row Locking and Authorization Check

In accounts.js, the system uses FOR UPDATE to lock rows during a transaction:

```
const [accounts] = await connection.query(
  'SELECT balance FROM accounts WHERE account_id = ? AND user_id = ? FOR UPDATE',
  [accountId, req.user.id]
);
```

Rollback on Failure

```
if (accounts.length === 0) {
  await connection.query('ROLLBACK');
  connection.release();
  return res.status(403).json({ error: 'Unauthorized or account not found' });
}
```

Commit on Successful Execution

After updating the account balance, recording the transaction, and creating an audit log entry:

```
await connection.query('UPDATE accounts SET balance = balance - ? WHERE account_id = ?', [amount, accountId]);

await connection.query(
  'INSERT INTO transactions (account_id, transaction_type, amount, description) VALUES (?, ?, ?, ?)',
  [accountId, 'withdrawal', amount, 'Withdrawal via app']
);

await connection.query(
  'INSERT INTO audit_log (user_id, action, details, ip_address) VALUES (?, ?, ?, ?)',
  [req.user.id, 'withdraw', `Withdrew ${amount} from account ${accountId}`, req.ip]
);

await connection.query('COMMIT');
connection.release();
```

This ensures that the withdrawal is completed atomically, protecting financial integrity.

Code Reference

- `accounts.js` — Transaction with row lock, rollback, and commit

6.7 Error Handling and Stability

Robust error handling mechanisms are integrated into all backend routes to preserve application stability in case of database failures.

Route-Level Error Control

Every module uses structured try–catch blocks:

Example from accounts.js:

```
try {  
  ...  
} catch (error) {  
  console.error(error);  
  res.status(500).json({ error: 'Internal server error' });  
}
```

Startup-Level Environment Validation

The system ensures that environment variables are loaded before initializing the database connection (server.js).

Consistent Logging

Errors are logged using `console.error()` to assist with debugging and tracking unexpected database behavior.

These measures ensure that the backend remains resilient even under exceptional circumstances.

Code References

- `accounts.js`, `admin.js`, `dashboard.js` — Structured try–catch
- `server.js` — Environment-based initialization

Chapter 7:

Front-End

7. Frontend

The front-end represents the interactive and user-facing layer of the SecureBank Online Banking System. This interface enables customers and administrators to access banking functionalities such as viewing accounts, initiating transactions, applying for cards or loans, and managing pending requests. The system adopts a clean and responsive web layout designed to ensure clarity, ease of navigation, and efficient interaction for all users.

The interface communicates with the Node.js backend through structured API requests, allowing operations such as withdrawals, deposits, account creation, loan approvals, and card processing to occur dynamically. All actions are executed without requiring page reloads, providing a seamless Single Page Application (SPA)-like experience. This results in a smooth, fast, and user-friendly banking workflow.

7.1 Objective of the Front-End Design

The primary objectives of the front-end component are:

1. To present a streamlined and responsive user interface for both customers and banking administrators.
2. To support efficient navigation across modules such as Dashboard, Accounts, Transactions, and Services.
3. To interact dynamically with backend APIs for real-time banking operations including deposits, withdrawals, and administrative approvals.
4. To ensure clear separation of functionality based on user roles (customer view vs. admin panel).
5. To enhance usability through consistent design, clear feedback messages, and organized visual components.

7.2 Technology Stack Used

The front-end of the SecureBank system is developed using a lightweight and efficient technology stack focused on clarity, responsiveness, and smooth interaction with the backend services.

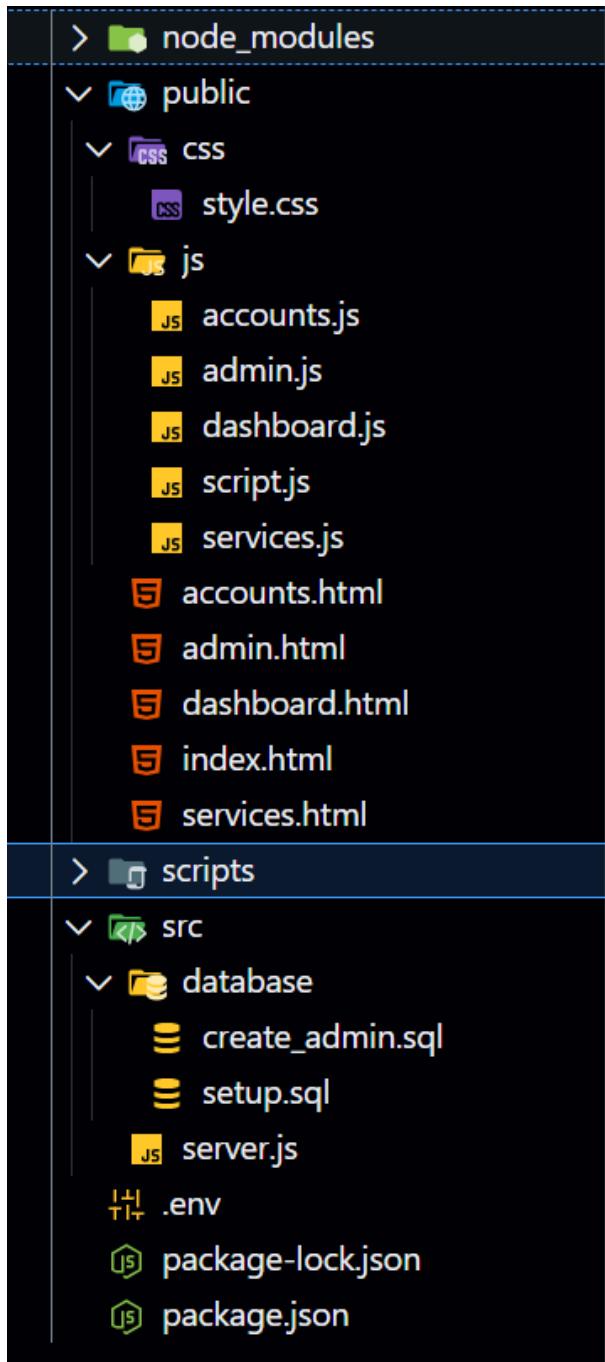
Technology	Purpose
HTML5	Provides the structural layout of all pages and components in the system.
CSS3	Defines styling, design consistency, responsiveness, and overall visual presentation.
JavaScript (ES6+)	Implements client-side logic, manages user interactions, handles API requests, and updates UI dynamically.
Express.js (Backend Routing Layer)	Used to serve the front-end pages and manage communication between client actions and server APIs.
JSON Web Tokens (JWT)	Facilitates secure authentication and ensures protected access to user-specific and admin-specific features.

7.3 Application Structure

The front-end of the SecureBank system is organized using a clear and structured file hierarchy to maintain readability and scalability. All interface pages, along with their supporting JavaScript logic and styling, are stored inside the **public/** directory, which is served directly by the Express.js backend.

Each page (Dashboard, Accounts, Services, Admin Panel, etc.) is implemented as an individual HTML file within the **public** folder, and every page's functionality is handled by a corresponding JavaScript file located under **public/js/**. Styling is managed through a centralized CSS file stored inside **public/css/**.

This organization ensures that every feature—whether it is viewing accounts, performing transactions, or processing admin approvals—is neatly maintained within its own dedicated HTML–CSS–JS set.



This structure ensures that each functionality of the system—such as managing user accounts, handling transactions, viewing dashboards, processing service requests, and administering approvals—is implemented within a dedicated and well-organized front-end module.

7.4 Core Components and Their Roles

Below are the main UI components/pages of the SecureBank front-end, written in the same detailed style as your friend's document but fully adapted to your actual project.

1. index.html (Landing Page & Authentication Interface)

Serves as the entry point of the system.

It hosts the login form and the registration modal.

Key elements include:

- User login panel
- Registration form popup
- Basic layout introducing SecureBank

JavaScript handled by: **script.js**

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Modern Bank Management System</title>
7     <link rel="stylesheet" href="css/style.css">
8     <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css">
9 </head>
10 <body>
11     <div class="container">
12         <nav class="navbar">
13             <div class="logo">
14                 <i class="fas fa-university"></i>
15                 <span>SecureBank</span>
16             </div>
17             <div class="nav-links">
18                 <a href="#" class="active">Home</a>
19                 <a href="#accounts">Accounts</a>
20                 <a href="#transactions">Transactions</a>
21                 <a href="#services">Services</a>
22             </div>
23             <div class="auth-buttons">
24                 <button id="loginBtn" class="btn">Login</button>
25                 <button id="registerBtn" class="btn btn-primary">Register</button>
26             </div>
27         </nav>
28
29         <main>
30             <section class="hero">
31                 <div class="hero-content">
32                     <h1>Welcome to Modern Banking</h1>
33                     <p>Experience secure and seamless banking operations</p>
34                     <button class="btn btn-primary">Get Started</button>
35                 </div>
36                 <div class="hero-image">
37                     <div class="floating-card"></div>
38                     <div class="floating-card"></div>
39                     <div class="floating-card"></div>
40                 </div>
41             </section>
42
43             <section id="features" class="features">
44                 <h2>Our Features</h2>
45                 <div class="features-grid">
46                     <div class="feature-card">
47                         <i class="fas fa-shield-alt"></i>
48                         <h3>Secure Banking</h3>
49                         <p>State-of-the-art security for your transactions</p>
50                     </div>
51                     <div class="feature-card">
52                         <i class="fas fa-mobile-alt"></i>
53                         <h3>Mobile Banking</h3>
54                         <p>Bank on the go with our mobile solutions</p>
55                     </div>
56                 </div>
57             </section>
58
59         </main>
60     </div>
61 </body>
62 </html>
```

```
56             <div class="feature-card">
57                 <i class="fas fa-exchange-alt"></i>
58                 <h3>Quick Transfers</h3>
59                 <p>Instant money transfers to any account</p>
60             </div>
61         </div>
62     </section>
63 </main>
64
65     <!-- Login Modal -->
66     <div id="loginModal" class="modal">
67         <div class="modal-content">
68             <span class="close">&times;</span>
69             <h2>Login</h2>
70             <form id="loginForm">
71                 <div class="form-group">
72                     <label for="username">Username</label>
73                     <input type="text" id="username" required>
74                 </div>
75                 <div class="form-group">
76                     <label for="password">Password</label>
77                     <input type="password" id="password" required>
78                 </div>
79                 <button type="submit" class="btn btn-primary">Login</button>
80             </form>
81         </div>
82     </div>
83
84     <!-- Register Modal -->
85     <div id="registerModal" class="modal">
86         <div class="modal-content">
87             <span class="close-register">&times;</span>
88             <h2>Register</h2>
89             <form id="registerForm">
90                 <div class="form-group">
91                     <label for="reg-username">Username</label>
92                     <input type="text" id="reg-username" required>
93                 </div>
94                 <div class="form-group">
95                     <label for="reg-password">Password</label>
96                     <input type="password" id="reg-password" required>
97                 </div>
98                 <div class="form-group">
99                     <label for="full-name">Full Name</label>
100                    <input type="text" id="full-name" required>
101                </div>
102                <div class="form-group">
103                    <label for="email">Email</label>
104                    <input type="email" id="email" required>
105                </div>
```

```

82      </div>
83
84      <!-- Register Modal -->
85      <div id="registerModal" class="modal">
86          <div class="modal-content">
87              <span class="close-register">&times;</span>
88              <h2>Register</h2>
89              <form id="registerForm">
90                  <div class="form-group">
91                      <label for="reg-username">Username</label>
92                      <input type="text" id="reg-username" required>
93                  </div>
94                  <div class="form-group">
95                      <label for="reg-password">Password</label>
96                      <input type="password" id="reg-password" required>
97                  </div>
98                  <div class="form-group">
99                      <label for="full-name">Full Name</label>
100                     <input type="text" id="full-name" required>
101                 </div>
102                 <div class="form-group">
103                     <label for="email">Email</label>
104                     <input type="email" id="email" required>
105                 </div>
106                 <div class="form-group">
107                     <label for="phone">Phone</label>
108                     <input type="tel" id="phone" required>
109                 </div>
110                 <button type="submit" class="btn btn-primary">Register</button>
111             </form>
112         </div>
113     </div>
114
115
116     <script src="js/script.js"></script>
117 </body>
118 </html>

```

2. dashboard.html (Customer Dashboard Page)

Displays a personalized welcome message and navigation options.

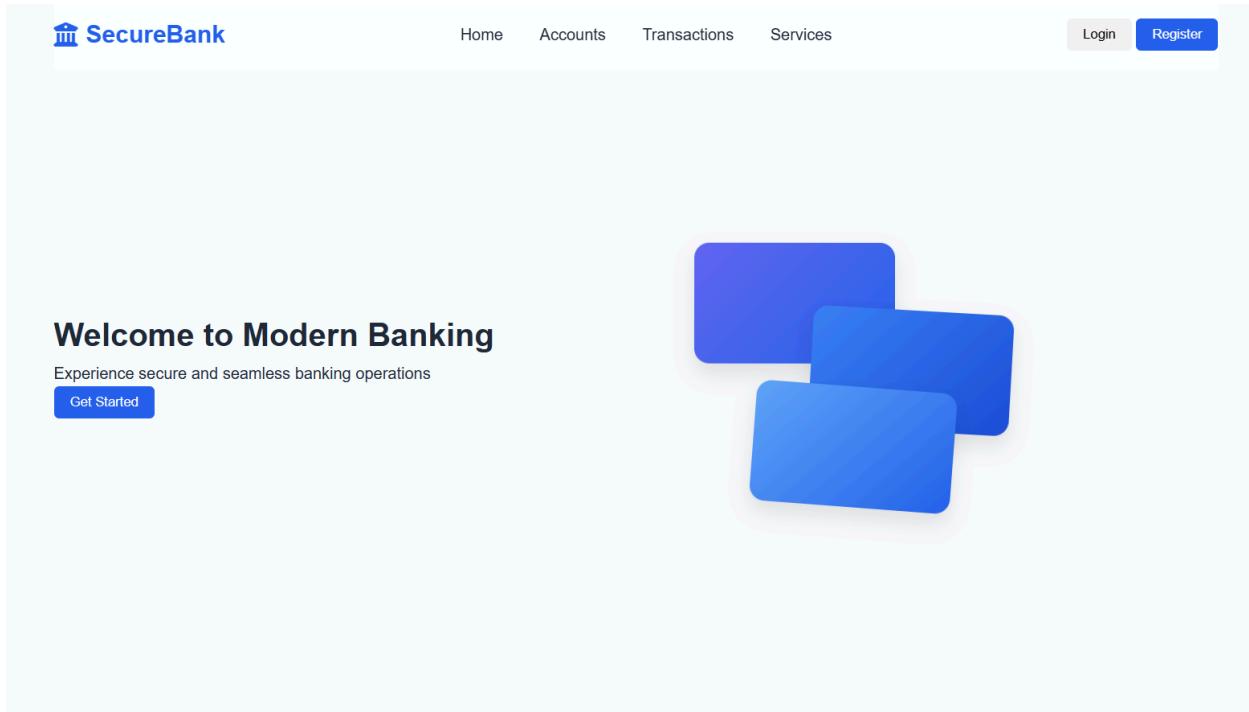
Acts as the starting point after successful login.

Key features:

- Links to Accounts, Transactions, and Services

- Clean dashboard layout
- Fetches and displays user information

JavaScript handled by: **dashboard.js**

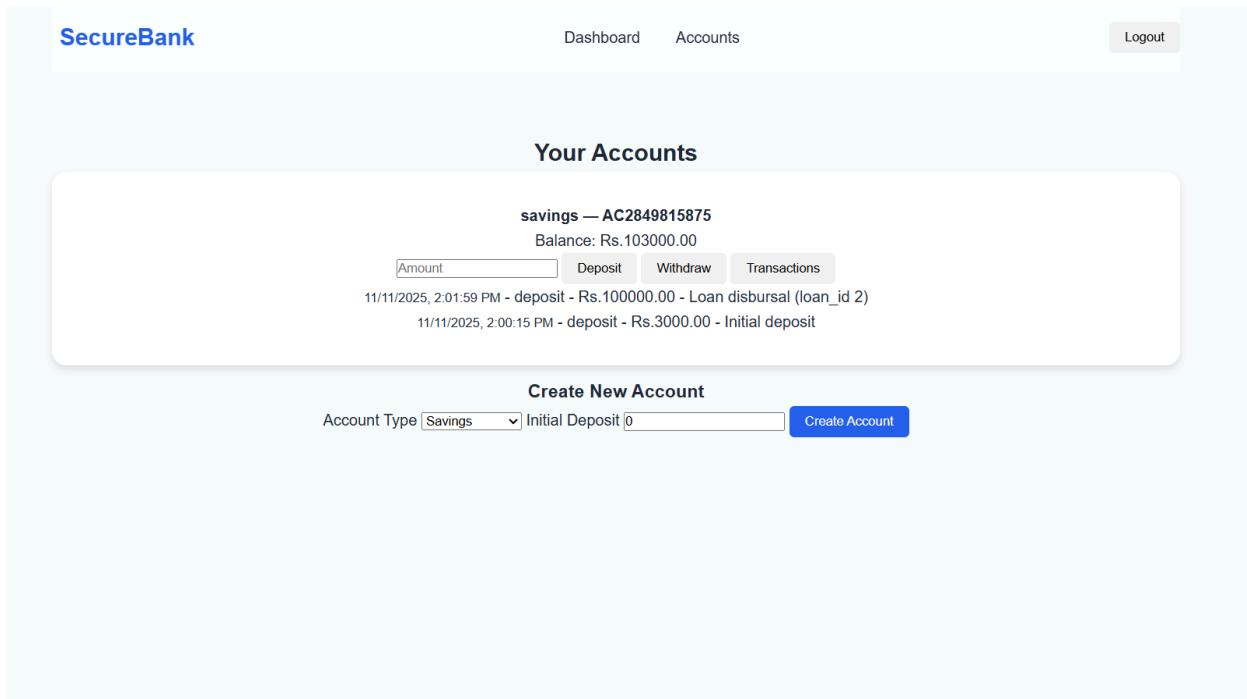


3. accounts.html (Account Management Page)

Allows users to view account details, balances, and recent transactions.
It provides interfaces for deposits, withdrawals, and opening new accounts.
Key features:

- Shows account balance
- Buttons for Deposit, Withdraw, and Transactions
- Displays transaction history
- Form for creating a new account

JavaScript handled by: **accounts.js**



4. services.html (Banking Services Page)

Provides access to loan applications, card requests, and related features.

Key features:

- Form to apply for loans
- Form to request debit/credit cards
- Status messages after submission

JavaScript handled by: [services.js](#)

Banking Services

Cards & Payments

Your Cards

DEBIT CARD

Card: X1762849905045
Expires: 11/11/2030

[Block Card](#)

CREDIT CARD

Status: requested
Requested: 11/26/2025

Request New Card

Card Type

Debit Card

[Request Card](#)

Loans & Financing

Your Loans

\$120000.00

Term: 12 months
Status: pending
Rate: 10.00%
Estimated EMI: \$10549.91
Applied: 11/26/2025

\$100000.00

Term: 24 months
Status: approved
Rate: 10.00%
Estimated EMI: \$4614.49
Applied: 11/11/2025

Apply for Loan

Loan Amount (Rs.)

Term (months)

Interest Rate (%) (optional)

[Preview Loan](#)[Apply Now](#)

5. admin.html (Admin Dashboard for Bank Staff)

Dedicated to administrators for managing customer requests.

Key features:

- List of pending loan applications
- List of pending card requests
- Approve/Reject workflows

JavaScript handled by: [admin.js](#)

6. style.css (Global Styling Component)

Ensures consistent color themes, button styles, form layouts, spacing, and responsive behavior across all pages.

It also manages:

- Navigation bar styling
- Card-style panels for admin approvals
- Modal and input field styling

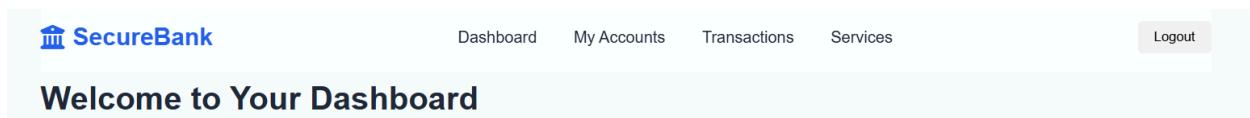
7. Navigation Bar Component (Shared UI Header)

Appears across major pages such as Dashboard, Accounts, Transactions, and Services.

Key features:

- SecureBank logo
- Navigation menu
- Logout button
- Responsive alignment for clean user experience

Implemented within individual HTML files and styled via **style.css**.



8. Registration Modal (UI Popup Component)

Displayed on index.html when the user opts to create a new account.

Key features:

- Inputs for username, password, full name, email, and phone
- Clean form layout
- JavaScript validation and API submission

Managed by: [script.js](#)

The image shows a registration form titled "Register" with a light gray background and rounded corners. At the top right is a small "X" icon. The form fields are labeled "Username", "Password", "Full Name", "Email", and "Phone", each with a corresponding input field below it. A blue "Register" button is located at the bottom left of the form area. The entire form is set against a dark gray background.

Register

Username

Password

Full Name

Email

Phone

Register

9. Transaction Panels (Deposit / Withdraw Components)

Used within accounts.html.

Key features:

- Input for transaction amount
- Dedicated buttons for deposit and withdrawal
- Real-time update of balance after API response

Managed by: [**accounts.js**](#)

The screenshot shows the 'SecureBank' application interface. At the top, there is a navigation bar with links for 'Dashboard', 'Accounts', and 'Logout'. Below the navigation bar, the main content area is titled 'Your Accounts'. It displays a card for an account named 'savings — AC2849815875' with a balance of 'Rs.103000.00'. The card includes tabs for 'Amount', 'Deposit', 'Withdraw', and 'Transactions'. Underneath the card, there are two transaction logs: '11/11/2025, 2:01:59 PM - deposit - Rs.100000.00 - Loan disbursal (loan_id 2)' and '11/11/2025, 2:00:15 PM - deposit - Rs.3000.00 - Initial deposit'. At the bottom of the page, there is a 'Create New Account' form with fields for 'Account Type' (set to 'Savings') and 'Initial Deposit' (set to '0'), followed by a 'Create Account' button.

10. Admin Approval Cards (Loan & Card Request Components)

Appearing on admin.html, these components present each pending request in a separate card.
Key features:

- Applicant name and details

- Inputs for credit limit or interest rate
- Approve/Reject buttons
- Clean and readable layout for bank staff

Managed by: [admin.js](#)

Bank Admin

[Logout](#)

Pending Card Requests

Soham Jagushte
@SohamJ

Card Type: CREDIT
Requested: 11/26/2025
Credit Limit (min: 1,000)

✓ Approve ✗ Reject

Pending Loan Applications

Soham Jagushte
@SohamJ

Amount: Rs.120000.00
Term: 12 months
Applied: 11/26/2025

Interest Rate Approve Reject

7.5 Styling with CSS

The visual design and layout of the SecureBank system are defined through `style.css`, which applies a consistent and professional theme throughout all pages.

Key styling aspects include:

- **Color Scheme:** A clean and modern palette with blue accents to reflect trust, security, and banking identity.
- **Layout System:** Flexbox and responsive container designs are used to maintain alignment and flexibility across different screen sizes.
- **Form Design:** Inputs, buttons, and modals are styled with rounded corners, spacing, hover effects, and error highlighting for improved usability.
- **Navigation:** The top navigation bar maintains consistent spacing and typography for clear accessibility.
- **Responsiveness:** Media queries ensure full compatibility with mobile phones, tablets, and desktops, maintaining smooth layout transitions across resolutions.

Each page maintains a structured and user-friendly design, ensuring coherence, accessibility, and an intuitive interface for both customers and administrators.

Github Link: https://github.com/Psj1234/dbms_exp

Conclusion

The SecureBank Online Banking System has been successfully developed as a comprehensive, database-driven web application that facilitates seamless interaction between customers, administrators, and essential banking services. The system demonstrates the practical application of Database Management System (DBMS) principles within a modern full-stack environment, integrating structured data storage, secure transaction handling, and a user-centric interface.

The database has been carefully designed using MySQL, incorporating well-defined tables such as **users**, **accounts**, **transactions**, **audit_log**, **bank_staff**, **mutual_funds**, **fixed_deposits**, and others. These tables maintain strong relational integrity through the use of primary and foreign keys, ensuring consistency across all stored information. Standard DML operations—including **INSERT**, **SELECT**, **UPDATE**, and **DELETE**—are utilized to perform core functionalities such as account creation, user authentication, deposits, withdrawals, card requests, loan applications, and administrative approvals.

Transaction Control Language (TCL) concepts are implemented effectively in multi-step financial operations. Features such as withdrawals use **row-level locking (FOR UPDATE)**, along with explicit **COMMIT** and **ROLLBACK** commands, guaranteeing atomicity and preventing inconsistencies during critical operations. The inclusion of an **audit_log** table further strengthens data accountability by recording every significant system action, enhancing transparency and security.

The backend, developed using **Node.js** and **Express.js**, establishes efficient and secure communication with the database through a **MySQL connection pool**. Environment-based configurations protect sensitive credentials, while modular API routing ensures scalability, maintainability, and separation of concerns. Robust error-handling mechanisms and token-based authentication (JWT) contribute to a secure and stable operational environment.

On the front-end, the system employs **HTML**, **CSS**, and **JavaScript** to deliver an intuitive and interactive user experience. Through real-time API communication, users can manage accounts, view balances, initiate transfers, request cards, apply for loans, and monitor financial activities without page reloads. The role-based interface ensures that administrative functions—such as user management, approvals, dashboard monitoring, and backend oversight—are accessible only to authorized personnel. Consistent styling, form feedback, and responsive layouts enhance usability across devices.

In summary, the SecureBank Online Banking System successfully integrates foundational DBMS concepts with modern web development technologies to produce a secure, scalable, and user-oriented banking platform. The project not only highlights database connectivity, data integrity, and transactional reliability but also demonstrates how full-stack development can be aligned with real-world financial operations to improve accuracy, efficiency, and user satisfaction.