Design Document

AscentPay: Elevate Your Finances

Welcome to **AscentPay**, your new digital banking solution.

Sandhya koukuntla - 700769703

Contents

1. Introduction

1.1 Purpose

This document provides a comprehensive overview of the **Digital Banking Application**, outlining the technical design, system architecture, and functional specifications.

The purpose is to define a secure, scalable, and user-friendly banking system that enables customers to manage their finances digitally. It also ensures that administrative users can efficiently oversee operations, enforce security, and maintain data integrity.

1.2 Scope

The Digital Banking Application offers a modern solution for online banking by implementing the following key features:

- **User Management**: Registration, authentication, and role-based access for customers and administrators.
- **Transaction Management**: Enabling users to perform deposits, withdrawals, transfers, and view detailed transaction history.
- **Admin Monitoring**: Allowing administrators to manage user accounts, monitor transactions, approve new accounts, and maintain the security of the platform.

1.2.1 User Management

The system supports secure account creation, login, and profile management. Key highlights include:

- Encrypted password handling
- Role-based authorization (Customer, Admin)
- Ability for users to update personal information securely

1.2.2 Account Management

The application allows users to:

- View real-time account balances
- Deposit funds into their accounts
- Withdraw money
- Transfer money to other users within the platform

All financial operations are securely logged and validated to ensure data integrity and prevent unauthorized access.

1.2.3 Transaction Management

Users can:

- Access a comprehensive transaction history
- Generate account statements
- Search and filter past transactions

The transaction management system ensures transparency, quick access to financial data, and auditability for both users and administrators.

1.3 Definitions and Acronyms

- **AWS**: Amazon Web Services – A cloud computing platform offering hosting, storage, and database services.
- **EC2**: Elastic Compute Cloud – AWS service providing scalable virtual servers.
- **RDS**: Relational Database Service – AWS service for managed relational databases, used here for secure storage of user and transaction data.
- **JWT**: JSON Web Token – A secure token-based authentication method for user sessions.

1.4 System Overview

The **Digital Banking Application** is built using the following modern technologies:

- **Frontend**: Developed with **React.js**, providing a dynamic, responsive, and user-friendly interface for banking operations.
- **Backend**: Implemented with **Spring Boot**, a Java-based framework that manages API requests, business logic, transaction processing, and authentication services.
- **Database**: A **MySQL** database hosted on **AWS RDS** is used to store user information, transaction data, and account balances securely.
- **Cloud Hosting**: Deployed on **AWS EC2** instances to ensure scalability, reliability, and security.

  Additional AWS services like IAM (Identity and Access Management) and S3 (Simple Storage Service) are integrated for enhanced security and resource management.

2. User Interfaces

2.1 Authentication Screens

Login and Registration interfaces designed for user access and security.

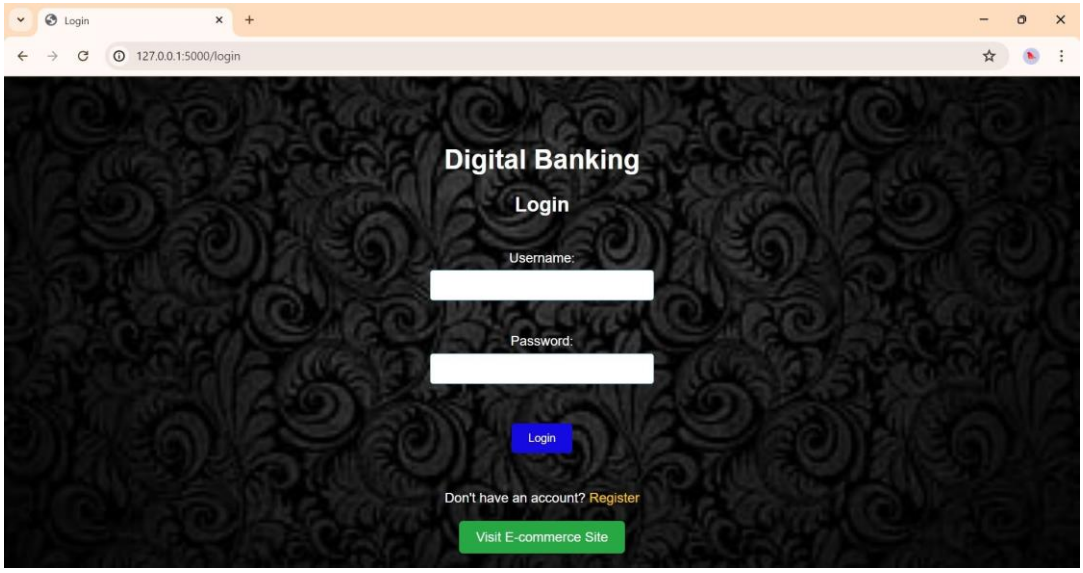

Figure 2.1.1 Login page

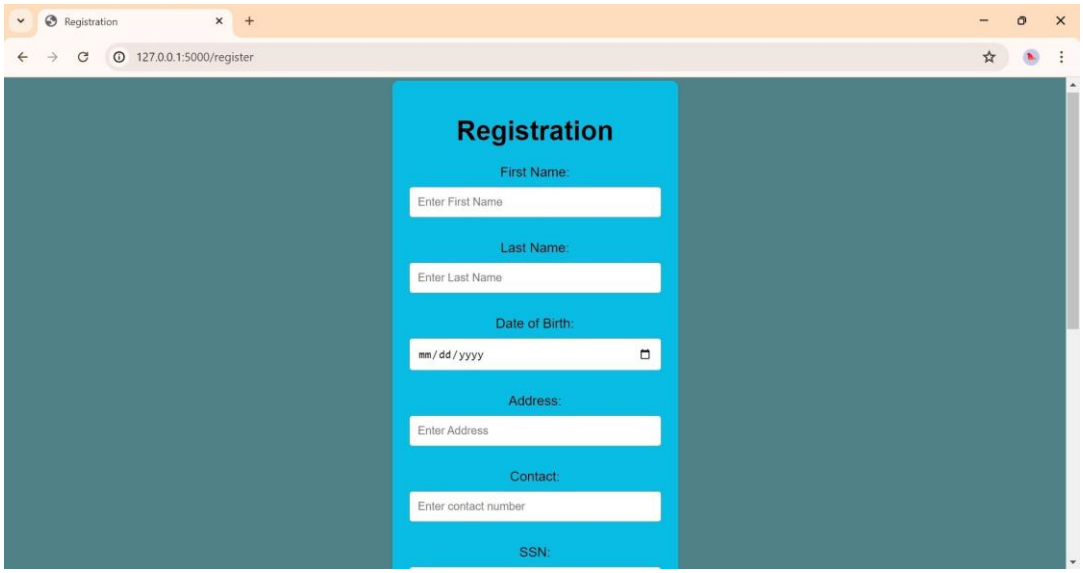

Figure 2.1.2 Registration page

2.2 Customer Dashboard

Customer dashboard provides transaction viewing, money transfer, and profile editing.

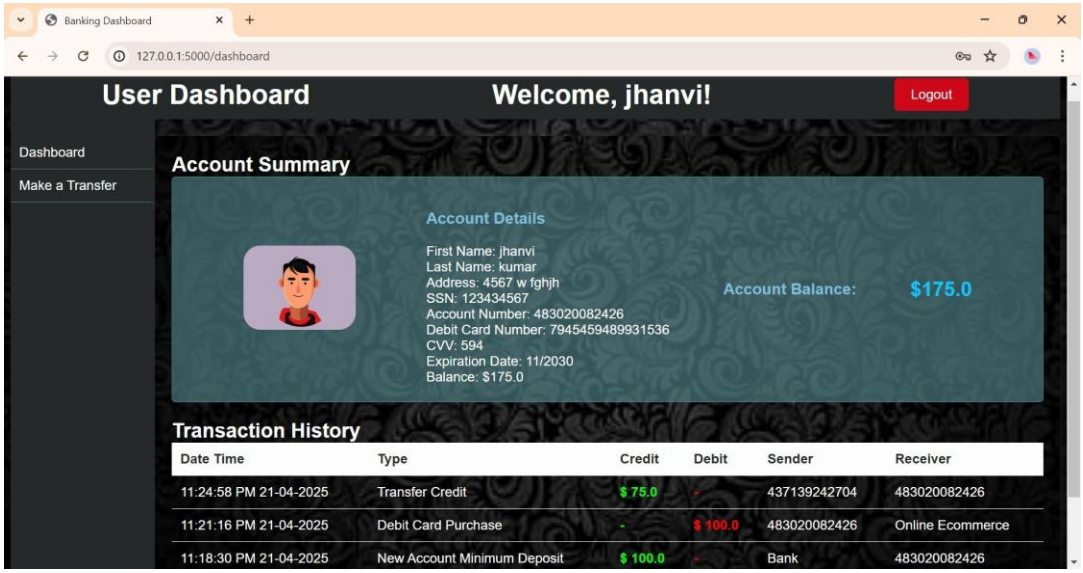

Figure 2.2.1 User dashboard

2.3 Admin Dashboard

Admin dashboard offers user management, account approvals, and transaction monitoring.

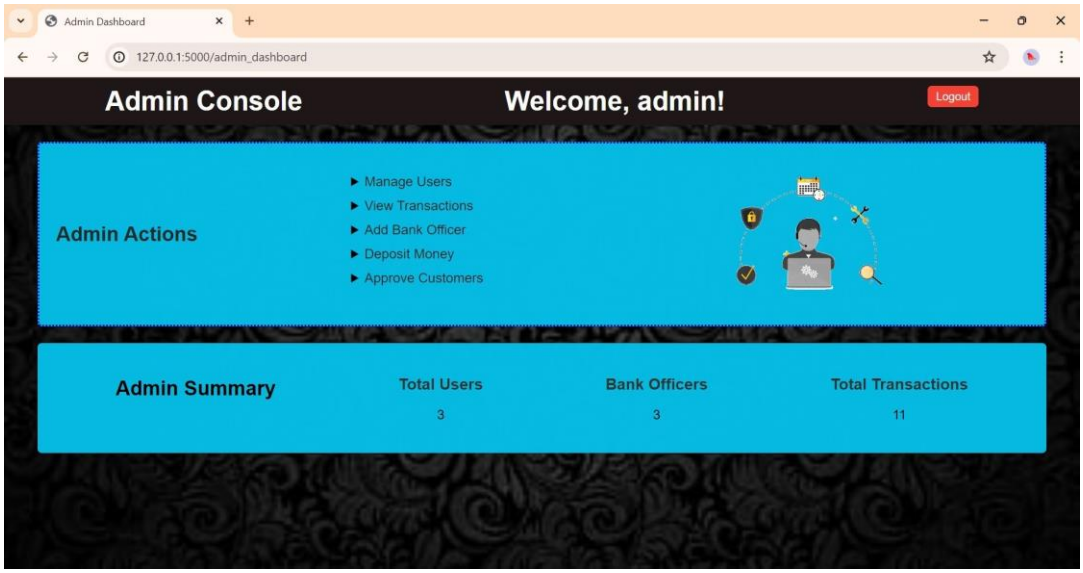

Figure 2.3.1 Admin console

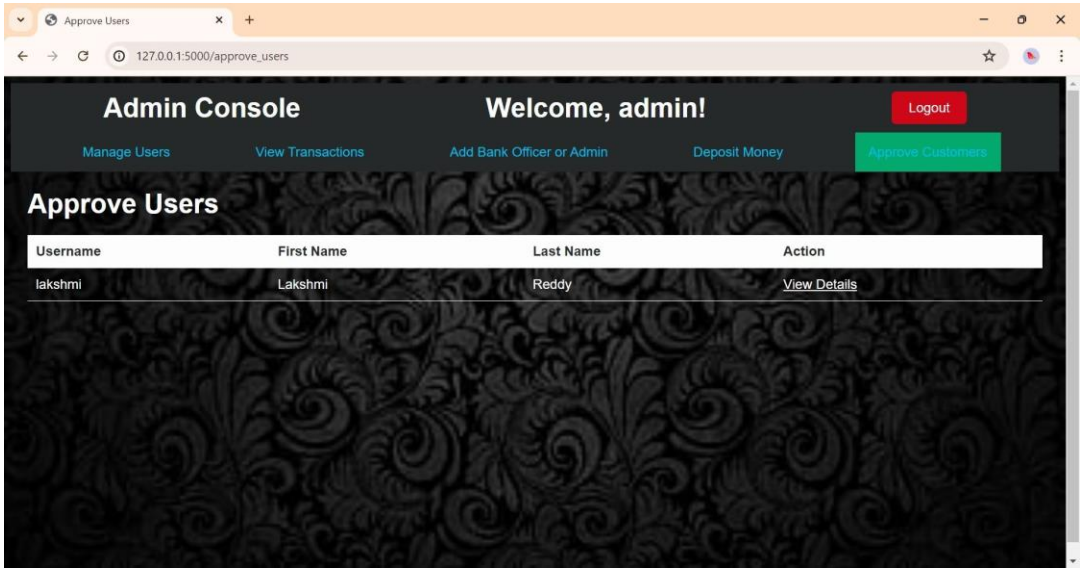

Fig 2.3.2 Approve users

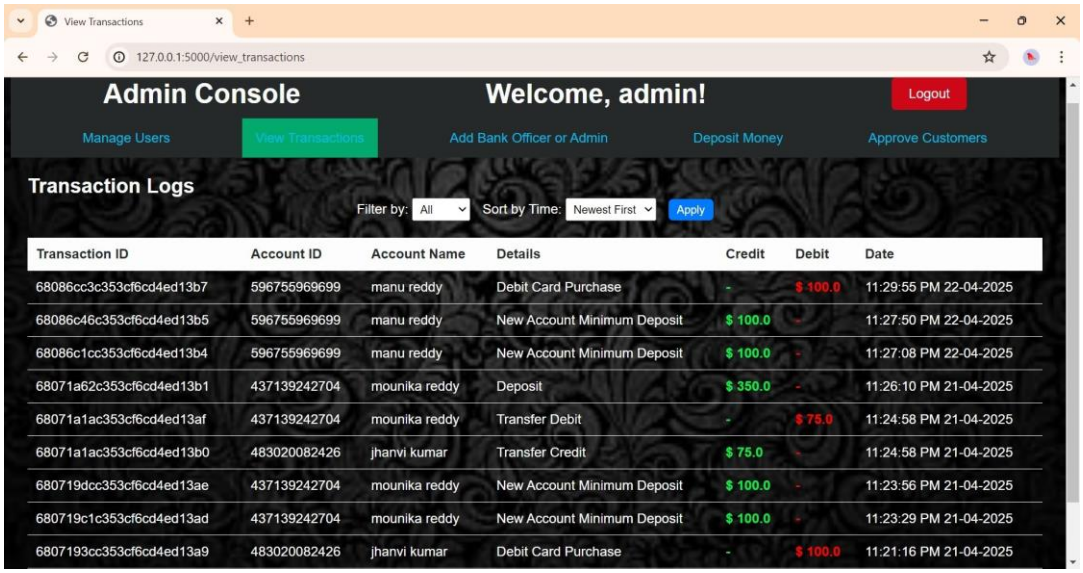

Figure 2.3.3 View transactions

3. System Architecture

3.1 Technology Stack

The Digital Banking Application is designed using a modern and scalable technology stack to ensure high availability, security, and performance:

- **Frontend**:

  Built using **React.js**, a popular JavaScript library for creating responsive, component-based user interfaces. It ensures smooth user interactions, fast rendering using the virtual DOM, and seamless navigation across different banking functionalities.

- **Backend**:

  Powered by **Spring Boot**, a lightweight and robust Java framework that simplifies the development of secure RESTful APIs. It manages business logic, user authentication, transaction processing, and communication with the database layer.

- **Database**:

  MySQL is used as the primary relational database, providing structured storage for user profiles, account details, and transaction histories. Hosted on AWS RDS for reliability, automatic backups, and failover support.

3.2 System Components

The application architecture is composed of three main components working together:

- **React Single Page Application (SPA)**:
    - Provides a responsive client-side user experience.
    - Manages routing, form submissions, and API interactions via Axios or Fetch.
    - Supports token-based (JWT) session management for security.
- **Spring Boot API Server**:
    - Exposes secure REST endpoints for account operations, user management, and transaction processing.
    - Handles business validation, security policies, and error management.
    - Implements service layers for scalable code management.
- **AWS Backend Infrastructure**:
    - Ensures secure cloud hosting, data storage, and scalable service deployments using EC2 and RDS.
    - IAM roles manage permissions and protect resources.

3.3 Database Design

The Digital Banking application leverages a **relational database model** to securely manage user identities, bank account data, and financial transactions. The design is structured to uphold **data consistency**, **referential integrity**, **security**, and **efficient performance**, which are critical for any financial platform.

The system maintains logical relationships between different entities, ensuring that user actions like deposits, withdrawals, and fund transfers are accurately recorded and traceable.

3.3.1 Key Entities:

- **User**:
  Represents an individual who has registered on the platform.
  Each user is uniquely identified and maintains attributes such as:
  - **user_id** (Primary Key)
  - First Name
  - Last Name
  - Email Address (Unique)
  - Encrypted Password
  - Phone Number
  - Role (e.g., Customer, Admin)
  - Residential Address
- **One User can be linked to one or more Accounts**, allowing users to maintain multiple bank accounts under the same identity.
- **Account**:
  Represents a user's active bank account. Each account has:
  - **account_id** (Primary Key)
  - **user_id** (Foreign Key referencing User)
  - Account Type (e.g., Savings, Checking)
  - Current Balance
  - Account Creation Date
- An **Account is always associated with one User** but a User may hold multiple Accounts.

Accounts serve as the source or destination for Transactions.

- **Transaction**:
  Represents the financial operations carried out between accounts. Each transaction record includes:
  - **transaction_id** (Primary Key)
  - **sender_account_id** (Foreign Key referencing Account)
  - **receiver_account_id** (Foreign Key referencing Account)
  - Transaction Amount
  - Transaction Type (Deposit, Withdrawal, Transfer)
  - Timestamp
  - Transaction Status (Pending, Completed, Failed)
- A **Transaction is linked to two accounts** (sender and receiver) and captures complete audit trails for accountability and user transparency.

3.3.2 Entity Relationships:

- **User → Account**:
  A **one-to-many** relationship where each User can have multiple Accounts.
- **Account → Transaction**:
  A **one-to-many** relationship where each Account can initiate or receive multiple Transactions.
- **Transaction → Account**:
  Each Transaction references the involved Accounts — Sender and Receiver — using foreign keys to maintain transactional integrity.

These relationships ensure that the banking application can efficiently track financial flows, manage account balances in real-time, and safeguard against anomalies like double-spending or unauthorized transfers.

The structure and relationships are summarized visually in the following Entity Relationship Diagram:

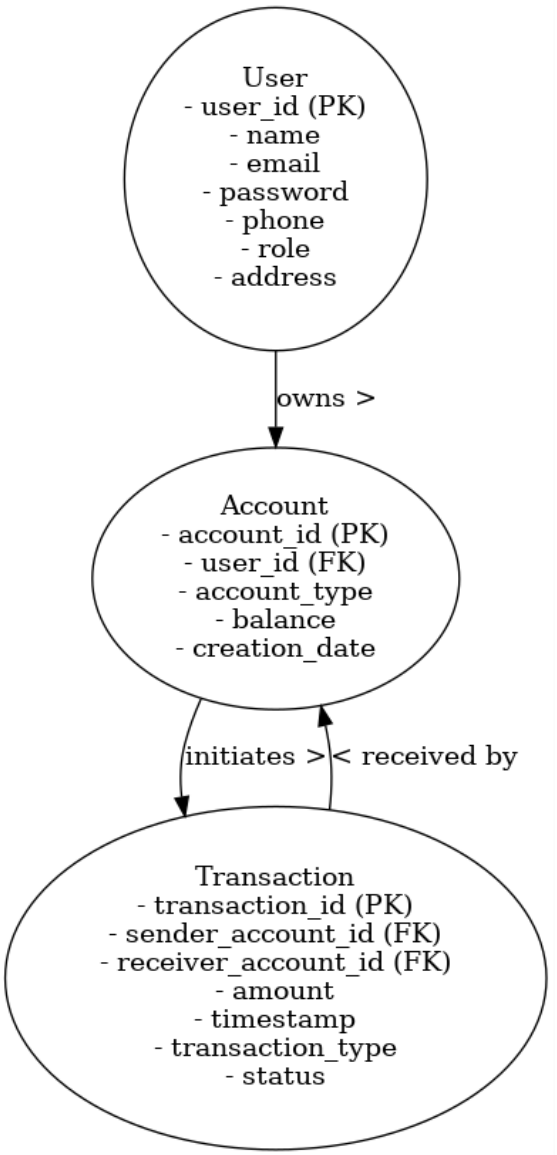

Figure 3.3.1 ER Diagram for Digital Banking Application

3.4 AWS Services Integration

The application infrastructure leverages several AWS services to enhance security, scalability, and operational efficiency:

- **Amazon EC2 (Elastic Compute Cloud)**:

- Hosts the backend Spring Boot application and the frontend React build files.
- Provides auto-scaling and instance monitoring features.

- **Amazon RDS (Relational Database Service)**:
  - Hosts the MySQL database with automated backup, snapshot creation, and multi-AZ deployment for high availability.
- **IAM (Identity and Access Management)**:
  - Manages secure access policies and roles for EC2, RDS, and other AWS resources, ensuring minimal exposure.
- **S3 (Simple Storage Service) (Optional)**:
  - Can be used to store static frontend assets or user-uploaded documents securely.

This cloud-based deployment guarantees robust disaster recovery, scalability for future expansion, and compliance with best security practices.

4. Functional Requirements

This section outlines the major functional modules and operations required for the **Digital Banking Application**. These functionalities are essential to ensure secure user management, efficient transaction processing, effective administrative control, and detailed reporting for users and administrators.

4.1 User Management

The application must provide secure and intuitive user management features, including:

- **User Registration**:
  New customers must be able to create an account by providing necessary information like name, email, password, phone number, and address.

Passwords should be securely hashed before storage.

- **User Authentication**:
  Existing users must log in using their registered email and password. JWT (JSON Web Tokens) should be used for managing session authentication securely.

- **Role-Based Access Control**:
  Different access permissions must be provided based on the user's role:

  - **Customers**: Access to their personal accounts and transactions only.
  - **Admins**: Access to view/manage all users, approve accounts, and monitor system activities.

- **Profile Management**:
  Users should have the ability to view and update their personal information, such as address, phone number, and password reset.

- **Account Status Management**:
  Admins should be able to deactivate or reactivate user accounts when necessary, especially for security or compliance purposes.

4.2 Transaction Management

The system must support robust and traceable transaction capabilities:

- **Deposit Funds**:
  Users should be able to deposit money into their account by specifying the amount. The balance must update immediately upon successful deposit.

- **Withdraw Funds**:
  Users can withdraw money from their account, provided sufficient funds exist. Appropriate checks must be in place to prevent overdrawing.

- **Fund Transfers**:

Users should be able to transfer money from their account to another account registered on the platform.

- Double-entry records must be created (debit sender, credit receiver).
- Real-time balance updates must occur for both accounts.

- **Transaction History**:
  Users must be able to view a detailed list of all their transactions with filters such as date range, transaction type (Deposit, Withdrawal, Transfer).

- **Validation & Error Handling**:
  The system should validate input amounts, check sufficient balance, and handle transaction failures gracefully with clear error messages.

4.3 Admin Operations

Administrative users must have specialized tools to manage the overall system:

- **User Account Management**:
  - View list of all users.
  - Approve pending account registrations.
  - Deactivate/reactivate user accounts.
  - Reset user passwords if requested.

- **Transaction Oversight**:
  - View transactions across all accounts.
  - Monitor large or suspicious transactions for fraud detection.

- **Service Monitoring**:
  - View system metrics such as active users, total transactions, total balances held.

- Receive alerts in case of unusual activities (optional future enhancement).

- **Role Management**:
  - Assign, update, or revoke admin privileges when necessary.

4.4 Reporting

The platform must enable users and admins to generate and access essential financial and operational reports:

- **User Transaction Statements**:
  - Users should be able to download monthly or custom-period statements showing all their transactions.
- **Admin Financial Summaries**:
  - Admins should have access to reports summarizing platform-wide transaction volumes, total deposits, withdrawals, and active account balances.
- **Export Capabilities**:
  - Reports should be exportable in standard formats such as PDF or CSV for record-keeping or tax filing purposes.
- **Audit Trails**:
  - For compliance and security, an audit log must record critical operations such as account creation, deletions, and financial activities.

5. Deployment

This section outlines the deployment strategy for the Digital Banking Application, detailing the necessary steps for setting up frontend, backend, database, environment configurations, and cloud infrastructure on AWS.

The deployment ensures that the system is scalable, secure, and accessible to endusers with high availability.

5.1 Frontend and Backend Deployment

Frontend Deployment (React.js):

- **Build Process**:
  - Use `npm run build` to generate optimized static assets for production (HTML, CSS, JavaScript).
- **Hosting**:
  - The built frontend files are hosted on an **AWS EC2 instance** using a web server like **NGINX**.
  - Alternatively, frontend can be hosted using **AWS S3 static website hosting** if required.
- **Security Configuration**:
  - Configure HTTPS using SSL/TLS certificates (e.g., Let's Encrypt) for secure client-server communication.
- **Domain Mapping**:
  - If a custom domain is used, map it to the EC2 public IP or set up Route 53 for DNS management.

Backend Deployment (Spring Boot):

- **Server Setup**:
    - Deploy the Spring Boot JAR file on an **AWS EC2** instance (Ubuntu or Amazon Linux).
    - Manage the application with **PM2** (Process Manager) or run it as a Linux service (systemd) for automatic startup on boot.
- **API Accessibility**:
    - Expose backend APIs on standard ports (e.g., port 8080 for HTTP).
    - Configure security groups to allow only necessary inbound traffic.
- **Backend Environment**:
    - Install Java (OpenJDK 17 or 21) on the EC2 instance.
    - Setup and configure environment variables for database connection, AWS credentials, etc.

5.2 Database Setup

The application's relational data is stored securely in an AWS-hosted database:

- **AWS RDS (Relational Database Service)**:
    - Launch an RDS instance with **MySQL 8.x**.
    - Choose `db.t3.micro` or similar instance type (if using Free Tier).
    - Configure multi-AZ deployments for high availability (optional enhancement).
- **Database Configuration**:
    - Create tables for Users, Accounts, and Transactions.

- Apply necessary constraints (Primary Keys, Foreign Keys) and indexing.
- Enable automated daily backups and performance monitoring.

- **Security Settings**:
  - Place the RDS instance inside a **private subnet** to restrict direct access from the internet.
  - Use Security Groups to allow traffic only from the EC2 instances hosting the backend.

5.3 Environment Variables

To ensure flexible and secure deployment, sensitive information and configuration details must be managed using environment variables:

Variable Name	Description
`DATABASE_URL`	Connection string to RDS MySQL database
`AWS_ACCESS_KEY_ID`	AWS programmatic access key for S3/other services
`AWS_SECRET_ACCESS_KEY`	Secret key paired with access key
`AWS_REGION`	Region where EC2 and RDS are deployed
`JWT_SECRET`	Secret key for token encryption and validation
`SPRING_DATASOURCE_URL`	Spring Boot datasource URL for DB connection

Best Practice:
Environment variables should never be hardcoded into the codebase.
Use `.env` files (secured) or AWS Systems Manager (SSM Parameter Store) for managing sensitive configurations.

5.4 Hosting on AWS EC2 and RDS

AWS EC2 Setup:

- Launch EC2 instances (Ubuntu 22.04 or Amazon Linux 2023 AMI).
- Configure instance types based on expected load (e.g., `t2.micro` for testing, scalable upwards for production).
- Install necessary software packages: Java, Node.js, NGINX, PM2.
- Configure security groups:
  - Open only necessary ports: 22 (SSH), 80 (HTTP), 443 (HTTPS), 8080 (backend API).
- Use Elastic IPs for a static public IP address (optional).

AWS RDS Setup:

- Launch RDS MySQL instance.
- Enable automated backups, monitoring, and encryption.
- Restrict database access to backend EC2 IP addresses only.

Load Balancer (Optional for scaling):

- AWS Application Load Balancer (ALB) can be set up in front of multiple EC2 instances for load distribution and failover handling.

6. Monitoring and Maintenance

Ongoing monitoring and maintenance of the Digital Banking Application are critical to ensure system stability, security, and high availability.
Proactive monitoring helps detect issues early, optimize performance, and maintain regulatory compliance in a financial environment.

This section outlines the essential strategies and tools used for continuous system health tracking, data protection, and operational integrity.

6.1 System Monitoring (AWS CloudWatch)

- **AWS CloudWatch Integration**:
  The EC2 instances hosting the frontend and backend services, as well as the RDS database, are monitored using **AWS CloudWatch**.
- **Key Metrics Tracked**:
  - **EC2 Monitoring**:
    - CPU Utilization
    - Network In/Out Traffic
    - Disk I/O Operations
  - **RDS Monitoring**:
    - Database Connections
    - Read/Write Latency
    - Free Storage Space
    - Query Performance
- **Alarms and Notifications**:
  - CloudWatch Alarms are set up to notify administrators when critical thresholds are crossed (e.g., CPU > 80%, low disk space).
  - Notifications are sent through **Amazon SNS (Simple Notification Service)** via email or SMS.
- **Dashboards**:
  - Custom CloudWatch Dashboards visualize real-time system performance for easy management and proactive scaling if needed.

6.2 Regular Backups

- **Database Backups**:
  - AWS RDS automatically creates daily snapshots of the database.
  - Retention policies are configured (e.g., retain last 7 days of backups).
  - Manual snapshots are also created before major deployments or system updates.
- **Frontend/Backend Code Backups**:
  - Source code is version-controlled via Git repositories (e.g., GitHub or AWS CodeCommit).
  - Server backups are taken periodically if major configuration changes occur.
- **Disaster Recovery Plan**:
  - In case of infrastructure failure, instances and databases can be restored from the latest backup with minimal downtime.

6.3 Security Updates and Patching

- **Operating System Updates**:
  - EC2 instances are regularly patched with the latest security updates for the underlying OS (Ubuntu or Amazon Linux).
- **Application Updates**:
  - Frontend and Backend dependencies (Node.js libraries, Java libraries) are reviewed for vulnerabilities (using tools like `npm audit`, `OWASP Dependency-Check`).
- **Database Patching**:

- AWS RDS provides automatic minor version upgrades to ensure the database is secure and stable.

- **Firewall and Security Groups**:
  - Periodic audits ensure only necessary ports are open.
  - IAM roles and policies are reviewed quarterly to maintain leastprivilege access.

6.4 Error Tracking and Alerts

- **Application-Level Monitoring**:
  - Spring Boot backend uses built-in Actuator endpoints to expose health checks, metrics, and error states.
  - Custom error logs are generated and rotated automatically.
- **Centralized Logging**:
  - CloudWatch Logs service aggregates application logs from EC2 instances for real-time debugging.
- **Alert Systems**:
  - Alerts are configured for:
    - Application errors (HTTP 5xx responses)
    - Unauthorized access attempts
    - System resource exhaustion
  - Admins receive immediate email/SMS alerts for critical events.
- **Incident Response Plan**:
  - Clear escalation procedures are defined.

- Critical errors trigger automatic ticket creation in project management tools (e.g., Jira, ServiceNow) if integrated.