Supplemental Document

Project Overview

This document outlines the technical tools and development methodology used in the implementation of the *PiggyBank* online banking platform. The project simulates banking functionalities such as account creation, user authentication, deposits, transfers, transaction history, and bill payments. It is designed as a full-stack web application, leveraging modern development frameworks, secure coding practices, and containerized deployment environments.

Tools and Technologies

Frontend

- **React (with Vite):** Utilized for constructing a modular and responsive single page application. Vite was selected to facilitate rapid development through optimized bundling and hot module replacement.
- **React Router DOM:** Provided client-side routing for navigation between key pages, including registration, login, dashboard, and account views.
- **Tailwind CSS:** Enabled scalable and consistent user interface styling using a utility first approach.
- **Lucide Icons:** Offered a clean and lightweight icon set for enhancing visual navigation elements.

Backend

- **Node.js with Express:** Served as the backend server and API layer, exposing RESTful endpoints for authentication, account operations, transaction processing, and bill payments.
- **Sequelize ORM:** Managed interactions with the PostgreSQL database using an object relational mapping approach, improving code maintainability and abstraction.
- **jsonwebtoken (JWT):** Used to implement secure session handling via HTTP-only cookies, mitigating common web vulnerabilities such as cross-site scripting (XSS).

- **bcryptjs:** Provided password hashing and verification for secure user credential management.
- **dotenv:** Managed environment specific configuration such as database credentials and secret keys.

Database

- **PostgreSQL:** Chosen as the relational data store for its robustness and compatibility with Sequelize. The database included normalized tables for users, accounts, and transactions, each supporting transactional operations.

Infrastructure

- **Docker Compose:** Employed to orchestrate the application's services, including frontend, backend, and database within isolated, reproducible containers. This approach streamlined local development and ensured consistency across environments.

Development Process

1. Initial Setup

A monolithic repository structure was established with distinct directories for client and server code. Docker Compose was configured to define and manage services, volumes, and network aliases for inter-container communication. Environment variables were set up using .env files to allow for secure and flexible configuration.

2. User Authentication

The system implemented secure registration and login mechanisms. Passwords were hashed before storage using bcryptjs, and authentication tokens were generated using JWT. A middleware layer validated the presence and validity of authentication tokens for protected routes.

3. Account and Transaction Management

Users were able to create and view checking accounts. Each account is assigned a unique number and associated with a specific user. Account specific endpoints enabled deposits, withdrawals, and internal transfers. A dedicated transaction history was maintained for each account, with timestamps and contextual information.

4. Bill Payment Feature

A bill payment module was integrated, allowing users to simulate payments to external payees. Bill payments were handled via a dedicated API endpoint that deducted funds and recorded the transaction as a BILL_PAY entry in the database. Sequelize transactions were utilized to ensure that balance adjustments and transaction entries were executed together as a single, consistent operation

5. Frontend Development

The frontend was developed using React components structured around views such as *Dashboard*, *Account*, *Login*, and *Register*. Protected routes were enforced using a custom <RequireAuth> component to ensure that only authenticated users could access sensitive pages. UI consistency and responsiveness were maintained through reusable component abstractions such as *Card*, *Input*, and *Button*, as well as layout containers.

6. Testing and Validation

Manual testing was conducted via Dockerized environments to simulate realistic use cases. Scenarios such as failed authentication, empty account states, incorrect inputs, and insufficient balance conditions were verified. API functionality was validated using browser developer tools and direct API calls.

Conclusion

The *PiggyBank* application demonstrates the integration of frontend and backend technologies to deliver a secure and functional banking simulation. Using React, Node.js, PostgreSQL, and Docker, the project reflects best practices in modern web development. Key features such as bill payment and internal fund transfers enhance the application's realism, while the use of authentication tokens and hashed credentials ensures data security. The project also highlights the value of containerization for scalable and portable development workflows.