Project Overview

The goal is to build a **Banking Dashboard** with core features that would be expected in a financial web application, demonstrating real-world functionality, security, and performance optimization. The project can simulate several modules typical to a banking platform, such as account summaries, transaction history, payment management, and settings, using a **React** front end and simulated APIs.

Key Project Components

1. **Architecture & Setup**:
    - **Structure the Application**: Using **React** with TypeScript for type safety and Next.js for optimized server-side rendering, which allows for better performance, especially in FinTech apps where security and responsiveness are critical.
    - **State Management**: Use **Redux** with middleware such as Redux-Saga to handle complex async actions and better control over side effects.
    - **Data Security**: Implement **JWT** tokens and **encryption** for simulating secure sessions.
    - **User Authentication & Authorization**: Implement a secure login page with multi-factor authentication (MFA) to mimic secure access.
2. **Dashboard Modules**:
    - **Account Overview**: Show the user's account details, balance, recent transactions, and an overview of assets.
    - **Transactions History**: List past transactions with filtering options (date, type, amount) and search functionality.
    - **Payment & Transfer Module**: Allow users to make payments and transfer funds, using mock data to simulate real transactions.
    - **Settings**: Include options for account settings, notifications, and security settings (e.g., password reset, MFA setup).
3. **Backend Simulation**:
    - **Mock API**: Set up a mock server using **JSON Server** or **Express** to simulate backend responses for account data, transaction records, and settings updates. This will also allow you to demonstrate how you'd handle API integration in production.
    - **API Security**: Secure endpoints with token-based authentication to showcase how the React app would handle token storage and secure data access.
4. **Performance Optimization**:
    - **Code Splitting and Lazy Loading**: Use React's lazy loading to defer loading of non-essential components.
    - **Error Boundaries**: Implement to catch and handle errors gracefully.
    - **Network Efficiency**: Apply caching and use `React.memo` and `useCallback` for optimization.
5. **Testing & QA**:

- **Unit Testing**: Write tests for critical components using **Jest** and **React Testing Library**.
- **End-to-End Testing**: Set up simple E2E tests with **Cypress** for login, data display, and key actions like transfers.
- **Code Review Best Practices**: Create a sample PR review process where you annotate areas of improvement or highlight performance concerns.

6. **Documentation**:
   - **Technical Documentation**: Outline the architecture, security considerations, and data flow.
   - **ReadMe**: A comprehensive README explaining the project purpose, features, installation, and setup.
7. **UI/UX & Security Enhancements**:
   - **Responsive Design**: Ensure the application is mobile-friendly with a responsive layout.
   - **Accessibility**: Implement ARIA labels and keyboard navigation for improved accessibility.
   - **Secure Coding Practices**: Use Helmet for secure headers, Content Security Policy (CSP) settings, and input validation on forms.
8. **Mentorship Simulation**:
   - **Code Review Checklist**: Demonstrate code review criteria to ensure best practices in React and JavaScript.
   - **Developer Notes**: Add inline comments on critical sections, explaining architectural decisions or security practices.