Detailed Implementation Report

This report provides a detailed explanation of the React frontend implementation, focusing on UI design, component architecture, social media login integration, error handling, security measures, and testing approaches. The document is limited to approximately 5 pages in scope.

1. UI Design and Component Architecture

UI Design

The user interface was designed with a **clean and minimalistic aesthetic**, following modern web app best practices:

- **Authentication Pages**: Simple login and signup screens, with Google login integrated as a primary option.
- **Dashboard**: Displays user-specific data after successful login.
- **Responsive Design**: Layout adapts to different screen sizes, ensuring accessibility across devices.

Component Architecture

We structured the frontend with **React functional components** and hooks:

- **App.jsx**: Root component handling routing.
- **AuthWrapper.jsx**: Higher-Order Component (HOC) to protect routes and ensure only authenticated users can access certain pages.
- **Login.jsx**: Implements Google OAuth popup logic and form-based login if needed.
- **Dashboard.jsx**: Displays user data after login.
- **UserContext (deprecated)**: Initially considered for global user state, later replaced by `AuthWrapper` for clarity and simplicity.

This modular design promotes maintainability and separation of concerns.

2. Integration with Social Media Login and API Functionality

Google OAuth Popup

We implemented Google login using Spring Security's default `/oauth2/authorization/google` endpoint. The React frontend:

1. Opens a centered popup window to start Google OAuth.
2. Backend handles Google authentication and generates a token.
3. Backend sends the token to the parent window using:

```
window.opener.postMessage({ token: token }, 'http://localhost:5173');
window.close();
```

4. React listens for the message:

```
useEffect(() => {
  function handleMessage(event) {
    if (event.origin !== "http://localhost:8080") return;
    if (event.data?.token) {
      sessionStorage.setItem("token", event.data.token);
      navigate("/dashboard");
    }
  }
  window.addEventListener("message", handleMessage);
  return () => window.removeEventListener("message", handleMessage);
}, [navigate]);
```

API Functionality

- **Token Storage**: Tokens are stored in `sessionStorage` to persist across tabs but clear on browser close.
- **Authenticated Requests**: Each API call includes `Authorization: Bearer <token>` in headers.
- **CORS Handling**: Configured in Spring Boot (`CorsConfig`) to allow requests from the frontend domain.

3. Error Handling and Security Measures

Error Handling

- **Login Errors**: If OAuth fails, the user is redirected back to the login screen with an error message.
- **Network Errors**: API failures trigger user-friendly messages like “Unable to connect, please try again.”
- **Fallback UI**: If session token is missing or invalid, users are redirected to `/login`.

Security Measures

- **CORS Configuration**: Restricted to `http://localhost:5173` during development to prevent cross-site request forgery.
- **Token Management**: Tokens are never stored in `localStorage` (reduces risk of XSS attacks).
- **PostMessage Security**: Verified `event.origin` to ensure tokens are only accepted from trusted sources.
- **Backend Validation**: The backend enforces role-based access control and verifies tokens before serving protected resources.

4. Testing Approach

Unit Testing

- **React Testing Library**: Used for testing individual components (e.g., Login form renders, buttons trigger handlers).
- **Mocking API Calls**: Axios and fetch requests mocked to test components without real backend.

Integration Testing

- **End-to-End Flow**: Tested OAuth login by simulating the popup flow and verifying token persistence.
- **Protected Routes**: Ensured users without tokens are redirected to `/login`.

Security Testing

- **Token Tampering**: Verified backend rejects malformed or expired tokens.
- **CORS Verification**: Confirmed backend rejects requests from unapproved origins.

5. Conclusion

The implementation successfully integrates:

- A clean, modular UI with reusable components.
- Secure Google OAuth authentication.
- Strong error handling and CORS protections.
- Testing strategies that validate both functional and security aspects.

This ensures a robust, maintainable, and user-friendly frontend aligned with industry best practices.