Implementation Guide

1) Introduction

This project implements a login system using OAuth 2.0 provided by Google and LinkedIn. Using OAuth 2.0, the login functionality allows users to authenticate via their own social media accounts. After login, sessions are managed via cookies, and eventually backend api calls will require the cookie authentication token to be reached.

2) Social Media Authentication Flow

The following section(s) will outline the exact steps taken by a user to login to the application using each provided social media platform.

2.1) Google Authentication Flow

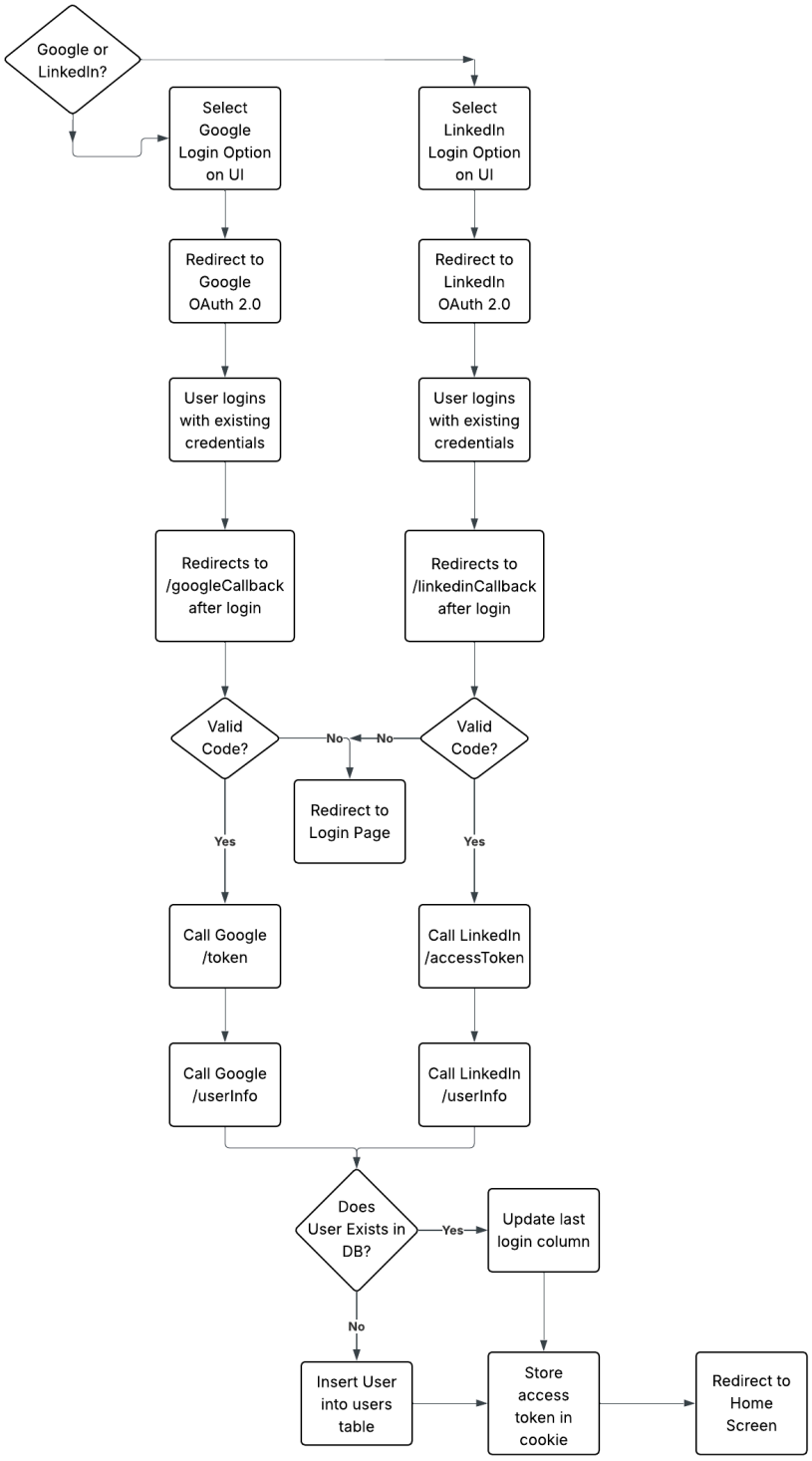
When a user wants to authenticate their session using Google OAuth 2.0, after visiting the landing page of my application, they must click the “Google” button, located on the left hand panel of the page. Upon clicking this button, the user is redirected to Google’s OAuth consent screen. From there, the user must login with their own existing accounts, and Google’s authentication functionality will confirm the credentials are valid. When this happens, the session is redirected to a backend API endpoint called ***“/googleCallback”*** . In the backend code of this endpoint, the server exchanges the code for an access token provided by Google’s **/token** endpoint. If the token is received from this endpoint, the authentication is complete, and the user's data is fetched from Google’s **/userInfo** endpoint. Once fetched, the application first validates whether the user already exists in the database table called **“users”**. If the user exists, the application updates the “updated_at” column, which tells us the last login timestamp of the current user. If the user does exist, the backend code inserts the following data: provider (either “google” or “linkedin”), provider id, name, email, created_at (when the user entry was created), and updated_at (last login date). Once the user data is updated in the database, the access token received from the OAuth 2.0 authentication process is stored in a cookie, and the user is redirected to the home screen.

2.2) LinkedIn Authentication Flow

When a user wants to authenticate their session using LinkedIn OAuth 2.0, after visiting the landing page of my application, they must click the “LinkedIn” button, located on the left hand panel of the page. Upon clicking this button, the user is redirected to LinkedIn’s OAuth consent screen. From there, the user must login with their own existing accounts, and LinkedIn’s authentication functionality will confirm the credentials are valid. When this happens, the session is redirected to a backend API endpoint called ***“/linkedinCallback”*** . In the backend code of this endpoint, the server exchanges the code for an access token provided by LinkedIn’s **/accessToken** endpoint. If the token is received from this endpoint, the authentication is complete, and the user's data is fetched from LinkedIn’s **/userInfo** endpoint. Once fetched, the application first validates whether the user already exists in the database table called “users”. If the user exists, the application updates the “updated_at” column, which tells us the last login timestamp of the current user. If the user does exist, the backend code inserts the following data: provider (either “google” or “linkedin”), provider id, name, email,

created_at (when the user entry was created), and updated_at (last login date). Once the user data is updated in the database, the access token received from the OAuth 2.0 authentication process is stored in a cookie, and the user is redirected to the home screen.

2.3) The following flow chart outlines the processes described above:



3) Database Management System
To store user data from each authentication session, the application uses a postgreSQL database hosted using supabase. The database has one table, thus far, called users, which stores the following data:

- provider = a string either value “google” or “linkedin”, indicating which platform the user is authenticated with
- provider_user_id = the returned id given by Google or LinkedIn
- name = the name designated with the user account
- email = the email designated with the user account
- created_at = the timestamp at which the entry in this table was created
- updated_at = the last time the user login via this platform and email

Each entry in this users table has a unique pair of (provider, email), making it so no user can have the same email for the same provider chosen.

Every time the user logs in, the application checks if that user exists. If the user does exist, the user’s updated_at column is updated to the current timestamp. If the user does not exist, the user's data is added to the table.

4) Session Management
For this application, I have chosen to use a token based session management system. When the user logs in via the social media accounts they already obtain, the OAuth 2.0 process provides us with an access token. This access token is then stored in an HttpOnly cookie, prior to the redirection of the user to the home screen.

In future implementation, the user will be logged out of the current session if the token expires, or if any api endpoint is hit and the token does not exist or has expired.

5) Error Handling, Logging, and Security Measures

5.1) As mentioned in earlier sections, when the user has logged in to their respective platforms, and the application is redirected to the appropriate backend endpoint, the application checks if the code is valid. If the code is not valid, the user is redirected back to the login screen.

If the token is valid, the application proceeds to exchange the code for the access token, and uses that token to get the user’s data. The application then checks to see if the data is invalid or missing before inserting into the table (or updating if the data already exists in the table). If any of this is incorrect, the database will throw an error, and the application will redirect to the login screen.

5.2) At any stage where an error is encountered, the application console logs the error for record. It is ensured additionally that no sensitive data is ever written to the console.

5.3) As mentioned in earlier sections, the access token is treated as a piece of sensitive data, and is stored in an HttpOnly cookie prior to being redirected to the login screen. This access token is never stored anywhere insecure such as the users database table, or localStorage. With the use of an HttpOnly cookie, only the backend code can access the token, the risk of XSS attacks is mitigated.

In addition to the cookie being HttpOnly, the cookie has SameSite Strict tags set on it, mitigating the risk of CSRF attacks.

6) Conclusion

With each of these sections taking into consideration the best technologies available and secure implementations, the application provides a secure and convenient method of authentication. Using OAuth 2.0, the users can login using their own social media accounts and maintain their sessions using the access tokens provided.