Let’s set up a React application with a Spring Boot backend where:

Login happens only once: The app stores access and refresh tokens, so the user stays logged in across the app.

Protected Routes: If a user tries to access any route that needs authentication without being logged in, they are redirected to the login page.

Public Routes: Routes that don’t need authentication (e.g., landing page) are accessible by anyone.

To achieve this, we’ll:

Implement a login form to get tokens.

Store tokens and manage authentication state globally.

Protect routes based on login status.

Use Axios to automatically attach tokens to requests when needed.

Step-by-Step Implementation

Step 1: Set Up Authentication Context

We’ll create an AuthContext to handle user login status, store access/refresh tokens, and provide login and logout functions.

Install required packages:

If you haven’t installed React Query or Axios:

bash

Copy code

npm install @tanstack/react-query axios

Create AuthContext.js: This file will set up global authentication state to manage tokens and login status.

javascript

Copy code

*// src/context/AuthContext.js*

import React, { createContext, useState, useEffect } from 'react';

export **const** AuthContext = createContext();

export **const** AuthProvider = ({ children }) **=>** {

**const** [isAuthenticated, setIsAuthenticated] = useState(false);

*// On initial load, check if there's an access token*

  useEffect(() **=>** {

**const** accessToken = localStorage.getItem('accessToken');

    setIsAuthenticated(!!accessToken); *// If token exists, set authenticated*

  }, []);

*// Login function to store tokens and set authentication*

**const** login = (accessToken, refreshToken) **=>** {

    localStorage.setItem('accessToken', accessToken);

    localStorage.setItem('refreshToken', refreshToken);

    setIsAuthenticated(true);

  };

*// Logout function to clear tokens and authentication*

**const** logout = () **=>** {

    localStorage.removeItem('accessToken');

    localStorage.removeItem('refreshToken');

    setIsAuthenticated(false);

  };

  return (

    <AuthContext.Provider value={{ isAuthenticated, login, logout }}>

      {children}

    </AuthContext.Provider>

  );

};

Explanation of AuthContext.js:

isAuthenticated: Tracks if the user is logged in.

login **function**: Stores tokens in localStorage and sets the app state to logged in.

logout function: Clears tokens and resets login state.

Step 2: Axios Instance with Interceptor

To make requests to protected endpoints simpler, we’ll use Axios to automatically attach the access token to each request.

Create axiosInstance.js: This instance includes an interceptor to add the access token and handles token refresh if the access token expires.

javascript

Copy code

*// src/axiosInstance.js*

import axios from 'axios';

**const** axiosInstance = axios.create({

  baseURL: 'http://localhost:8080/api', *// Adjust to your API base URL*

  headers: { 'Content-Type': 'application/json' },

});

*// Add access token to request headers*

axiosInstance.interceptors.request.use((config) **=>** {

**const** accessToken = localStorage.getItem('accessToken');

  if (accessToken) {

    config.headers.Authorization = `Bearer ${accessToken}`;

  }

  return config;

});

export default axiosInstance;

Explanation of axiosInstance.js:

Interceptor: Before each request, it checks for the access token and, if present, attaches it to the Authorization header.

Step 3: Login Component with Token Storage

Create a Login component where users input their email and password, which then requests tokens from the backend.

Create useLogin Hook: This hook sends a POST request to /auth/login with email and password.

javascript

Copy code

*// src/hooks/useLogin.js*

import { useMutation } from '@tanstack/react-query';

import axios from 'axios';

export **const** useLogin = () **=>** {

  return useMutation(**async** ({ email, password }) **=>** {

**const** response = await axios.post('http://localhost:8080/auth/login', { email, password });

    return response.data; *// Expecting { accessToken, refreshToken } in response*

  });

};

Create Login.js: This component will use the useLogin hook and store the tokens if login is successful.

javascript

Copy code

*// src/components/Login.js*

import React, { useState, useContext } from 'react';

import { useLogin } from '../hooks/useLogin';

import { AuthContext } from '../context/AuthContext';

import { useNavigate } from 'react-router-dom';

**const** Login = () **=>** {

**const** [email, setEmail] = useState('');

**const** [password, setPassword] = useState('');

**const** loginMutation = useLogin();

**const** { login } = useContext(AuthContext);

**const** navigate = useNavigate();

**const** handleLogin = (e) **=>** {

    e.preventDefault();

    loginMutation.mutate(

      { email, password },

      {

        onSuccess: (data) **=>** {

**const** { accessToken, refreshToken } = data;

          login(accessToken, refreshToken);  *// Store tokens and update login state*

          navigate('/protected'); *// Redirect to a protected page*

        },

        onError: (error) **=>** {

          console.error('Login failed:', error);

        },

      }

    );

  };

  return (

    <form onSubmit={handleLogin}>

      <input

        type="email"

        value={email}

        onChange={(e) **=>** setEmail(e.target.value)}

        placeholder="Email"

        required

      />

      <input

        type="password"

        value={password}

        onChange={(e) **=>** setPassword(e.target.value)}

        placeholder="Password"

        required

      />

      <button type="submit">Login</button>

    </form>

  );

};

export default Login;

Step 4: Route Protection for Public and Protected Routes

Define routes that only allow access if the user is logged in. If they aren’t, redirect them to the login page.

Set up ProtectedRoute Component:

javascript

Copy code

*// src/App.js*

import React, { useContext } from 'react';

import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

import { BrowserRouter as Router, Route, Routes, Navigate } from 'react-router-dom';

import Login from './components/Login';

import LandingPage from './components/LandingPage';

import Posts from './components/Posts';

import { AuthProvider, AuthContext } from './context/AuthContext';

**const** queryClient = new QueryClient();

**const** ProtectedRoute = ({ children }) **=>** {

**const** { isAuthenticated } = useContext(AuthContext);

  return isAuthenticated ? children : <Navigate to="/login" />;

};

**const** App = () **=>** {

  return (

    <QueryClientProvider client={queryClient}>

      <AuthProvider>

        <Router>

          <Routes>

            {*/\* Public routes \*/*}

            <Route path="/" element={<LandingPage />} />

            <Route path="/login" element={<Login />} />

            {*/\* Protected routes \*/*}

            <Route

              path="/protected"

              element={

                <ProtectedRoute>

                  <Posts />

                </ProtectedRoute>

              }

            />

          </Routes>

        </Router>

      </AuthProvider>

    </QueryClientProvider>

  );

};

export default App;

Explanation of Route Setup:

Public Routes: Landing page and login page don’t need authentication.

Protected Routes: Any route wrapped with <ProtectedRoute> will check if the user is logged in. If not, the user will be redirected to /login.

Step 5: Use Axios for Protected API Calls

Use axiosInstance to fetch data from protected endpoints; the token is automatically added to the request headers.

Example of Protected Data Fetching: Fetch posts with authentication.

javascript

Copy code

*// src/hooks/usePosts.js*

import { useQuery } from '@tanstack/react-query';

import axiosInstance from '../axiosInstance';

**const** fetchPosts = **async** () **=>** {

**const** response = await axiosInstance.get('/posts');

  return response.data;

};

export **const** usePosts = () **=>** {

  return useQuery(['posts'], fetchPosts);

};

With this setup:

Login once: Tokens are stored and reused automatically.

Protected routes: Redirect to login if the user is unauthenticated.

Authenticated requests: Axios interceptors manage tokens on each request.

You said:

I hope login page disappear Free login..yes or no

ChatGPT said:

ChatGPT

Yes, once the user successfully logs in, the login page will disappear. After login, they will be redirected to a protected page (or wherever you want), ensuring they won’t see the login page again unless they log out.

Here’s how it works in our setup:

After a successful login, we call navigate('/protected') (or any other protected route).

This navigation effectively "hides" or removes the login page, as the app will display the target route instead.

So, the login page will only appear when the user is unauthenticated and trying to access protected routes. Once logged in, they won't see the login page again.

You said:

I forgot to mention some APIs of Springboot application also need authentication..I hope thts included in the above code

ChatGPT said:

ChatGPT

Yes, the code above handles API authentication for the Spring Boot backend as well. Here’s how it’s managed:

Key Points Covering API Authentication

Axios Interceptor:

In axiosInstance.js, we add an interceptor that automatically attaches the access token to each request's Authorization header if a token is available. This is crucial for APIs that require authentication.

javascript

Copy code

*// src/axiosInstance.js*

import axios from 'axios';

**const** axiosInstance = axios.create({

  baseURL: 'http://localhost:8080/api', *// Adjust to your API base URL*

  headers: { 'Content-Type': 'application/json' },

});

*// Interceptor to add the access token*

axiosInstance.interceptors.request.use((config) **=>** {

**const** accessToken = localStorage.getItem('accessToken');

  if (accessToken) {

    config.headers.Authorization = `Bearer ${accessToken}`;

  }

  return config;

});

export default axiosInstance;

This interceptor ensures that any request made **using** axiosInstance includes the access token if the token is present in localStorage.

Using axiosInstance for Protected Endpoints:

When you use axiosInstance in any API call (e.g., fetching posts, updating user info, etc.), it will automatically add the Authorization header with the access token, making these requests authenticated.

Example:

javascript

Copy code

*// src/hooks/usePosts.js*

import { useQuery } from '@tanstack/react-query';

import axiosInstance from '../axiosInstance';

**const** fetchPosts = **async** () **=>** {

**const** response = await axiosInstance.get('/posts'); *// Requires authentication*

  return response.data;

};

export **const** usePosts = () **=>** {

  return useQuery(['posts'], fetchPosts);

};

API Requests Requiring Authentication:

Any Spring Boot API endpoint requiring authentication can be accessed this way. The backend will receive the Authorization header with the access token, allowing it to validate the token and authorize the request.

Summary

In summary, yes, this setup covers your Spring Boot APIs that need authentication:

The axiosInstance interceptor ensures the access token is attached to all requests.

Protected endpoints are called with this axiosInstance, automatically meeting the Spring Boot application's authentication requirements.

Let me know