**Route Security and User/Admin Dashboard**

**Introduction**

**This document outlines the steps taken to implement route security and create user/admin dashboards in the project. It covers the changes made to existing files (authController.js, login.js, navbar.js, app.js, authRoutes.js) and elaborates on the newly created files (AdminDashboard.js and UserDashboard.js).**

**Changes in Existing Files**

**authController.js**

**The authController.js file was updated to include the user's role in the login response. This allows the frontend to differentiate between admin and normal users and redirect them accordingly.**

**Changes Made:**

* **Modified the login function to include the user's role in the response.**

exports.login = async (req, res) => {

  const { email, password } = req.body;

  try {

    const user = await User.findOne({ email });

    if (!user) {

      return res.status(400).json({ error: 'User not found' });

    }

    if (!user.isActive) {

      return res.status(400).json({ error: 'User is not active' });

    }

    const isMatch = await bcrypt.compare(password, user.password);

    if (!isMatch) {

      return res.status(400).json({ error: 'Invalid credentials' });

    }

    user.lastActive = Date.now();

    await user.save();

    const token = jwt.sign({ id: user.\_id, role: user.role }, process.env.JWT\_SECRET, { expiresIn: '1h' });

    res.json({ token, role: user.role });

  } catch (err) {

    res.status(500).json({ error: err.message });

  }

};

**breakdown of the changes and their implications:**

**Original Version (Similarities):**

* **Both versions use libraries like mongoose, bcrypt, and jsonwebtoken for interacting with the database, password hashing, and generating JWTs, respectively.**
* **Both versions define asynchronous functions for register and login that handle user requests.**
* **Both versions handle errors and send appropriate error messages in the response.**

**Registration Function (register):**

* **Original: Registers a user with the provided username, password (hashed), and role.**
* **New:**
  + **Checks for an existing user with the same email before registration.**
  + **Includes additional fields in the user object like name, isActive (set to true), and lastActive timestamp.**
  + **Uses email instead of username for checking existing users (potentially for better email uniqueness).**

**Login Function (login):**

* **Original: Uses username for login and sends a JWT token upon successful login.**
* **New:**
  + **Uses email instead of username for login (consistent with registration).**
  + **Checks if the user is active before allowing login (potential account management feature).**
  + **Updates the user's lastActive timestamp upon successful login.**
  + **Includes the user's role in the response along with the JWT token.**

**Improvements:**

**The new version of the controller functions incorporates additional security and user management features:**

* **Checking for existing users based on email during registration prevents duplicate email addresses.**
* **Verifying user activation status during login ensures only active accounts can access the application.**
* **Including the user's role in the login response allows for role-based authorization in subsequent API requests.**

**login.js**

**The login.js file was updated to handle the redirection of users based on their role after successful login.**

**Changes Made:**

* **Modified the handleSubmit function to check the user's role and navigate to the appropriate dashboard.**

import React, { useState } from 'react';

import axios from 'axios';

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

import './Login.css';

const Login = () => {

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

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

  const [message, setMessage] = useState('');

  const navigate = useNavigate();

  const handleSubmit = async (e) => {

    e.preventDefault();

    try {

      const res = await axios.post(`${process.env.REACT\_APP\_API\_URL}/api/auth/login`, {

        email,

        password

      });

      const { token, role } = res.data;

      localStorage.setItem('token', token);

      if (role === 'admin') {

        navigate('/admin-dashboard');

      } else {

        navigate('/user-dashboard');

      }

    } catch (err) {

      console.error(err);

      if (err.response && err.response.data) {

        setMessage('Error: ' + err.response.data.message);

      } else {

        setMessage('An error occurred');

      }

    }

  };

  return (

    <div className="login-container">

      <h2>Login</h2>

      <form className="login-form" onSubmit={handleSubmit}>

        <input

          type="email"

          placeholder="Email"

          value={email}

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

        />

        <input

          type="password"

          placeholder="Password"

          value={password}

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

        />

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

      </form>

      <p>{message}</p>

    </div>

  );

};

export default Login;

**Here's a breakdown of the changes and their implications:**

**Original Version:**

* **Handles login form submission and sends an axios POST request to the login API endpoint.**
* **Upon successful login (assumed based on the code), sets a success message (Login successful!).**
* **Catches and displays any errors encountered during the login process.**

**New Version:**

* **Similar to the original version in handling form submission and API call.**
* **Key Difference:**
  + **Instead of setting a success message, it retrieves the token and user role from the API response (res.data).**
  + **Stores the token in localStorage for potential use throughout the application (authentication).**
  + **Uses useNavigate (from react-router-dom) to redirect the user based on their role:**
    - **Admins are redirected to the /admin-dashboard route.**
    - **Regular users are redirected to the /user-dashboard route.**

**Improvements:**

**The new version enhances the login functionality by:**

* **Implementing role-based routing: Users are directed to appropriate dashboards based on their role.**
* **Utilizing a token (JWT) for authentication: The retrieved token can be used to identify and authorize the user in subsequent requests.**

**navbar.js**

The Navbar.js file was updated to include a Logout button, which clears the session and redirects to the home page.

**Changes Made:**

* Added a Logout button with functionality to clear the token and redirect to the home page.

import React from 'react';

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

import LogoutButton from './LogoutButton';

import './Navbar.css';

const Navbar = () => {

  return (

    <nav className="navbar">

      <ul className="navbar-list">

        <li><Link to="/">Home</Link></li>

        <li><Link to="/register">Register</Link></li>

        <li><Link to="/login">Login</Link></li>

        <li><LogoutButton /></li>

      </ul>

    </nav>

  );

};

export default Navbar;

Breakdown of the changes:

**Similarities:**

* Both versions import React and Link from react-router-dom.
* Both versions render a navigation bar (nav) element containing an unordered list (ul) of navigation items (li).
* Both versions use Link components to define navigation links for different routes (/, /register, /login).
* Both versions import styles from Navbar.css (assumed).

**Changes:**

* **Original:** Uses additional classes for styling:
  + nav-list for the unordered list
  + nav-item for each list item
  + nav-link for each link component
* **New:** Removes the extra styling classes and relies solely on the base classes (navbar and navbar-list). This could indicate a cleaner or more centralized approach to styling.

**Additional Component:**

* The new version imports and renders a LogoutButton component (likely defined elsewhere). This suggests the addition of a logout functionality that might be triggered from the navbar.

**app.js**

The App.js file was updated to include routes for the user and admin dashboards and to protect these routes.

**Changes Made:**

* Added protected routes for user and admin dashboards using a custom ProtectedRoute component.

import React from 'react';

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

import Navbar from './components/Navbar';

import Home from './pages/Home';

import Register from './pages/Register';

import Login from './pages/Login';

import UserDashboard from './pages/UserDashboard';

import AdminDashboard from './pages/AdminDashboard';

import ProtectedRoute from './components/ProtectedRoute';

const App = () => {

  return (

    <Router>

      <Navbar />

      <Routes>

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

        <Route path="/register" element={<Register />} />

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

        <Route path="/user-dashboard" element={

          <ProtectedRoute roleRequired="user">

            <UserDashboard />

          </ProtectedRoute>

        } />

        <Route path="/admin-dashboard" element={

          <ProtectedRoute roleRequired="admin">

            <AdminDashboard />

          </ProtectedRoute>

        } />

      </Routes>

    </Router>

  );

};

export default App;

Breakdown of the changes and their implications:

**Similarities:**

* Both versions import React and routing components from react-router-dom.
* Both versions render the Navbar component throughout the application.
* Both versions define routes for the main pages (Home, Register, and Login) using Route components.

**Changes:**

* **Removed Container Elements (Original):**
  + The original version wraps the Router component with a div element containing the Navbar.
  + It uses another div with the class "content-container" to enclose the routes.
  + This structure might have been used for basic layout purposes.
* **Simplified Structure (New):**
  + The new version removes the extra container elements, resulting in a cleaner structure. The Navbar is rendered directly within the Router component.
* **Added Protected Routes (New):**
  + The new version introduces two routes for protected dashboards (user-dashboard and admin-dashboard).
  + These routes are wrapped with a ProtectedRoute component (likely defined elsewhere). This suggests implementing user roles and authorization to control access to specific dashboards.

**authRoutes.js**

The authRoutes.js file was updated to ensure the correct import path for the authController and to define routes for login and registration.

**Changes Made:**

* Corrected the import path and added the necessary routes.

const express = require('express');

const router = express.Router();

const authController = require('../contollers/authController');

const verifyToken = require('../middleware/verifyToken');

router.post('/register', authController.register);

router.post('/login', authController.login);

router.get('/user-dashboard', verifyToken, (req, res) => {

  if (req.userRole !== 'user') {

    return res.status(403).json({ message: 'Access denied' });

  }

  res.status(200).json({ message: 'Welcome to the user dashboard' });

});

router.get('/admin-dashboard', verifyToken, (req, res) => {

  if (req.userRole !== 'admin') {

    return res.status(403).json({ message: 'Access denied' });

  }

  res.status(200).json({ message: 'Welcome to the admin dashboard' });

});

module.exports = router;

Breakdown of the changes and their implications:

**Original Version (Basic Authentication):**

* Defines routes for user registration (/register) and login (/login).
* Uses the corresponding functions (register and login) from the authController.

**New Version (Enhanced Authentication with Authorization):**

* Similar route definitions for registration and login.
* **Key Difference:** Introduces two additional protected routes:
  + /user-dashboard: Accessible only to users with the "user" role.
  + /admin-dashboard: Accessible only to users with the "admin" role.
* Uses a middleware function verifyToken (likely defined elsewhere) before accessing the protected routes. This middleware probably verifies the presence and validity of a JWT token in the request.
* Checks the user's role (req.userRole) extracted from the token within each protected route.
* Returns a 403 Forbidden response if the user's role doesn't match the required role for the dashboard.
* Returns a success message (200 OK) if the user is authorized to access the dashboard.

**Improvements:**

The new version implements role-based authorization on top of basic authentication:

* It restricts access to specific dashboards based on the user's role retrieved from the JWT token.
* This enhances security by ensuring only authorized users can access sensitive data or functionalities.

**New Files**

**AdminDashboard.js**

This file defines the admin dashboard component, which displays the interface for admin users.

**File Location:**

src/pages/AdminDashboard.js

**UserDashboard.js**

This file defines the user dashboard component, which displays the interface for regular users.

**File Location:**

src/pages/UserDashboard.js

**ProtectedRoute.js**

This component is responsible for protecting routes by checking if the user is authenticated.

**File Location:**

src/components/ProtectedRoute.js

import React from 'react';

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

const ProtectedRoute = ({ children, roleRequired }) => {

  const token = localStorage.getItem('token');

  const userRole = localStorage.getItem('role');

  if (!token) {

    // If no token, redirect to login

    return <Navigate to="/login" />;

  }

  if (roleRequired && userRole !== roleRequired) {

    // If role doesn't match, redirect to home or a 403 page

    return <Navigate to="/" />;

  }

  return children;

};

export default ProtectedRoute;

Breakdown of its functionality:

1. **Props:**
   * children: The component or element that will be rendered if the user is authorized to access the protected route.
   * roleRequired (optional): The specific user role required to access the route. If not provided, any authenticated user can access it.
2. **Authorization Check:**
   * Retrieves the token and user role stored in localStorage.
   * Checks if a token exists. If not, it redirects the user to the login page (/login) using Navigate from react-router-dom.
3. **Role Check (if applicable):**
   * If roleRequired is provided, it compares the retrieved user role with the required role.
   * If the roles don't match, it redirects the user to the home page (/) using Navigate. This behavior can be customized to redirect to an error page (e.g., 403 Forbidden) if needed.
4. **Rendering:**
   * If the user is authorized (has a token and the role matches if required), it renders the children component or element that is wrapped by this ProtectedRoute.

**In essence, this component acts as a gatekeeper for routes that require authentication and potentially specific user roles.** It ensures only authorized users can access specific parts of your application.