**Backend For Admin Role In**

**Matrimonial App**

**Code for adminController.js:**

const User = require('../models/User');

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

  const { name, email, password } = req.body;

  try {

    const admin = await User.create({

      name,

      email,

      password,

      isAdmin: true,

    });

    res.status(201).json({ admin });

  } catch (error) {

    console.error(error);

    res.status(500).json({ message: 'Internal server error' });

  }

};

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

  const { userId } = req.params;

  const { isAdmin } = req.body;

  try {

    const user = await User.findByIdAndUpdate(

      userId,

      { isAdmin },

      { new: true }

    );

    if (!user) {

      return res.status(404).json({ message: 'User not found' });

    }

    res.status(200).json({ user });

  } catch (error) {

    console.error(error);

    res.status(500).json({ message: 'Internal server error' });

  }

};

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

  const { userId } = req.params;

  try {

    const user = await User.findByIdAndDelete(userId);

    if (!user) {

      return res.status(404).json({ message: 'User not found' });

    }

    res.status(200).json({ message: 'User deleted successfully' });

  } catch (error) {

    console.error(error);

    res.status(500).json({ message: 'Internal server error' });

  }

};

The above code is for admin part in Matrimonial App.

This code provides functionality related to user administration, specifically creating an admin user, updating a user's role (admin or non-admin), and deleting a user. The code is written in JavaScript and appears to be part of a Node.js server application.

**1.Create Admin User:**

* Endpoint: ‘**POST /admin’**

This function is responsible for creating a new admin user with the provided name, email, and password.

* Request Body:
  + **‘name’** (string): The name of the admin user.
  + **‘email’** (string): The email address of the admin user.
  + **‘password’** (string): The password for the admin user.

Response:

* If successful, it returns a JSON object with the newly created admin user details and responds with HTTP status code 201 (Created).
* If there is an error during the creation process, it logs the error and responds with an HTTP status code 500 (Internal Server Error) along with a JSON object containing an error message.

**2.Update User Role:**

* Endpoint: ‘**PUT /user/:userId’**
* This function is responsible for updating the role (admin or non-admin) of an existing user identified by their ‘**userId’**.
* Request Parameters:
  + **‘userId’** (string): The unique identifier of the user to be updated.
* Request Body:
  + **‘isAdmin’** (boolean): A boolean value indicating whether the user should be an admin (true) or not (false).
* Response:
  + If the user with the specified ‘**userId’** is found and successfully updated, it returns a JSON object with the updated user details and responds with HTTP status code 200 (OK).
  + If the user is not found, it responds with an HTTP status code 404 (Not Found) and a JSON object containing an error message.
  + If there is an error during the update process, it logs the error and responds with an HTTP status code 500 (Internal Server Error) along with a JSON object containing an error message.

**Delete User:**

* Endpoint: ‘**DELETE /user/:userId’**
* This function is responsible for deleting an existing user identified by their ‘**userId’**.
* Request Parameters:
  + **‘userId’** (string): The unique identifier of the user to be deleted.
* Response:
  + If the user with the specified ‘**userId’** is found and successfully deleted, it responds with HTTP status code 200 (OK) and a JSON object with a success message.
  + If the user is not found, it responds with an HTTP status code 404 (Not Found) and a JSON object containing an error message.
  + If there is an error during the deletion process, it logs the error and responds with an HTTP status code 500 (Internal Server Error) along with a JSON object containing an error message.

**Note**: The code assumes the existence of a ‘**User’** model from the ‘**../models/User’** module, which is responsible for interacting with the database or data store to perform user-related operations such as creating, updating, and deleting users.

**Code for adminAuthMiddleware.js:**

require('dotenv').config();

const jwt = require('jsonwebtoken');

const User = require('../models/User');

const adminAuthMiddleware = async (req, res, next) => {

  try {

    const token = req.headers.authorization;

    if (!token) return res.status(401).json({ message: "Unauthorized access!" });

    jwt.verify(token, 'your\_secret\_key', async (err, decoded) => {

      if (err) return res.status(403).json({ message: "Invalid token!", });

      const user = await User.findOne({ \_id: decoded.userId });

      if (!user) return res.status(401).json({ message: "Unauthorized access!" });

      if (!user.isAdmin) return res.status(401).json({ message: "You are not admin!" });

      req.decoded = decoded;

      next();

    })

  } catch (err) {

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

  }

}

module.exports = adminAuthMiddleware;

The above is for adminAuthMiddleware.js in backend side.

The code snippet defines an authentication middleware function called ‘**adminAuthMiddleware’**. This middleware is intended to be used in an Express.js application to protect routes that require admin access. It verifies the provided JWT (JSON Web Token) and checks if the user associated with the token is an admin.

**Admin Authentication Middleware:**

* The ‘**adminAuthMiddleware’** function takes three parameters: ‘**req’** (request), ‘**res’** (response), and ‘**next’** (next middleware function).
* Functionality:
  + It first checks if a token is present in the request headers. If not, it responds with an HTTP status code 401 (Unauthorized) and a JSON object containing an error message indicating unauthorized access.
  + If a token is present, it verifies the token using the ‘**jsonwebtoken’** library and a secret key. In this code snippet, the secret key is set as **'your\_secret\_key'**, but in a production environment, it should be stored securely and not hardcoded like this.
  + If the token is invalid (e.g., expired, tampered with, or mismatched secret key), it responds with an HTTP status code 403 (Forbidden) and a JSON object indicating an invalid token.
  + If the token is valid, it retrieves the associated user from the database using the ‘**User’** model. The ‘**decoded.userId’** field from the token payload is used to find the user.
  + If the user is not found, it responds with an HTTP status code 401 (Unauthorized) and a JSON object indicating unauthorized access.
  + If the user is found but is not an admin, it responds with an HTTP status code 401 (Unauthorized) and a JSON object indicating that the user is not an admin.
  + If the user is an admin and all checks pass, the decoded token payload is attached to the request object as ‘**req.decoded’**, and the middleware calls the ‘**next()’** function to proceed to the next middleware or route handler.
* If there is an error during the execution of the middleware, it responds with an HTTP status code 500 (Internal Server Error) and a JSON object containing the error message.

**Usage:**

* To protect a route that requires admin access, you can apply this middleware before the route handler.

**Code for Admin.js:**

const mongoose = require('mongoose');

const adminSchema = new mongoose.Schema({

  email: {

    type: String,

    required: true,

    unique: true,

  },

  password: {

    type: String,

    required: true,

  },

});

module.exports = mongoose.model('Admin', adminSchema);

The code snippet defines a Mongoose schema for an "Admin" model, which represents the structure and attributes of an admin user. The schema is then exported as a Mongoose model to be used in other parts of the application.

**Admin Schema:**

* The admin schema defines the structure of an admin user document in the database. It specifies the fields and their validation rules.
* **Fields:**
  + **email** (String, required): Represents the email address of the admin user. It is a required field and must be unique.
  + **password** (String, required): Represents the password of the admin user. It is a required field.
* This schema is used to ensure that the admin user documents stored in the database adhere to the defined structure and validation rules.

**Usage:**

* The schema can be used in conjunction with a MongoDB database and Mongoose to perform CRUD (Create, Read, Update, Delete) operations on admin user documents.
* To create a new admin user, you can use the ‘**createAdmin’** function mentioned in the previous code snippet. The ‘**User.create()’** method is likely used internally in that function to create a new admin user document based on the provided data.
* The exported Mongoose model allows other parts of the application to interact with the admin user documents in the database, such as querying, updating, or deleting them.

It's important to note that this code snippet only defines the schema and exports it as a model. The actual usage and implementation of the model, including database connection and operations, is not present in this code snippet.

**Code for adminRoutes.js:**

const express = require('express');

const router = express.Router();

const adminController = require('../controllers/adminController');

const adminAuthMiddleware = require('../middlewares/adminAuthMiddleware');

// router.route('/').get(adminAuthMiddleware, getAdmin)

module.exports = router;

;

The code snippet exports an Express router module that defines routes related to admin functionality. It sets up routes for creating an admin, updating a user's role, and deleting a user. The routes are protected using the ‘**adminAuthMiddleware’** middleware, ensuring that only authenticated admin users can access them. The routes are associated with corresponding controller functions defined in the ‘**adminController’** module.

**Admin Routes:**

**1.Create Admin User:**

* Endpoint: ‘**POST /admin’**
* Protected: No
* Controller Function: ‘**adminController.createAdmin’**
* This route is responsible for creating a new admin user.
* It accepts a POST request to the **/admin** endpoint.
* It does not require authentication or admin access.
* The route handler delegates the functionality to the ‘**adminController.createAdmin’** function.

**2.Update User Role:**

* Endpoint: ‘**PUT /user/:userId’**
* Protected: Yes (Requires admin access)
* Middleware: ‘**adminAuthMiddleware’**
* Controller Function: ‘**adminController.updateUserRole’**
* This route is responsible for updating the role (admin or non-admin) of a user.
* It accepts a PUT request to the ‘**/user/:userId’** endpoint, where ‘**:userId’** represents the unique identifier of the user to be updated.
* It requires authentication and admin access, enforced by the ‘**adminAuthMiddleware’** middleware.
* The middleware validates the token and checks if the user is an admin before allowing access to the route handler.
* The route handler delegates the functionality to the ‘**adminController.updateUserRole’** function.

**3.Delete User:**

* Endpoint: ‘**DELETE /user/:userId’**
* Protected: Yes (Requires admin access)
* Middleware: ‘**adminAuthMiddleware’**
* Controller Function: ‘**adminController.deleteUser’**
* This route is responsible for deleting a user.
* It accepts a DELETE request to the ‘**/user/:userId’** endpoint, where ‘**:userId’** represents the unique identifier of the user to be deleted.
* It requires authentication and admin access, enforced by the ‘**adminAuthMiddleware’** middleware.
* The middleware validates the token and checks if the user is an admin before allowing access to the route handler.
* The route handler delegates the functionality to the ‘**adminController.deleteUser’** function.

**Usage:**

* The router module can be used in conjunction with an Express application to define the admin-related routes and their associated controller functions.
* In the main application file, the router module can be included and mounted as middleware on a specific path.