

(NEED TO VERIFY HOW TO ACCOMPLISH HIGHLIGHTED TEXT)  
  
From my understanding to achieve the goal of allowing users to create an account

on the app, being able to log in through their google or facebook account as well

and to use the models and save the results of their predictions in a mongodb

database can be best implemented by following the following steps:

STEP 1:

To set up your MongoDB Atlas database connection in your backend application,

follow these steps:

1. Create a new file, e.g., `config/db.js`, to store your database configuration.

2. Install the MongoDB driver for Node.js using a package manager like npm or yarn:

```

**npm install mongodb**

```

3. In the `**db.js`** file, require the MongoDB driver:

```javascript

**const { MongoClient } = require('mongodb');**

```

4. Define the connection string for your MongoDB Atlas database. You can obtain this

string from your MongoDB Atlas dashboard.

```javascript

**const uri = 'mongodb+srv://<username>:<password>@<cluster-url>/<database-name>?retryWrites=true&w=majority';**

```

Replace `<username>`, `<password>`, `<cluster-url>`, and `<database-name>` with

the appropriate values from your MongoDB Atlas account.

5. Create a connection function that establishes the connection to the database:

```javascript

**async function connectToDatabase() {**

**const client = new MongoClient(uri, { useUnifiedTopology: true });**

**try {**

**// Connect to the MongoDB Atlas database**

**await client.connect();**

**console.log('Connected to the database');**

**// Return the database instance for further use**

**return client.db();**

**} catch (error) {**

**console.error('Error connecting to the database:', error);**

**throw error;**

**}**

**}**

```

6. Export the `connectToDatabase` function so that it can be used in other files:

```javascript

**module.exports = connectToDatabase;**

```

7. In your main application file, e.g., `**app.js**` or `index.js`, **import the `connectToDatabase**` function:

```javascript

**const connectToDatabase = require('./config/db');**

```

8. Call the `connectToDatabase` function to establish the connection:

```javascript

**const db = await connectToDatabase();**

```

9. You now have access to the connected database instance (`db`) throughout your application. You can use it to perform database operations using the MongoDB driver.

Remember to replace `<username>`, `<password>`, `<cluster-url>`, and `<database-name>` in the connection string with your actual MongoDB Atlas credentials and database information.

By following these steps, you will be able to set up the connection to your MongoDB Atlas database in your backend application.

STEP 2:

To configure **Passport.js** and set up authentication strategies in your backend application, follow these steps:

1. Create a new file, e.g., `**passport.js**`, in your project's directory.

2. Install the required dependencies for Passport.js and the desired authentication strategies. For example, if you want to use Google and Facebook authentication, you can install the following packages:

```

**npm install passport passport-google-oauth20 passport-facebook**

```

3. In the `**passport.js**` file, import the required modules:

```javascript

**const passport = require('passport');**

**const GoogleStrategy = require('passport-google-oauth20').Strategy;**

**const FacebookStrategy = require('passport-facebook').Strategy;**

```

4. Configure the Google authentication strategy. Obtain the client ID and client secret by creating a new project in the Google Developers Console and enabling the Google+ API:

```javascript

passport.use(

new GoogleStrategy(

{

clientID: GOOGLE\_CLIENT\_ID,

clientSecret: GOOGLE\_CLIENT\_SECRET,

callbackURL: '/auth/google/callback',

},

(accessToken, refreshToken, profile, done) => {

// Add your custom logic to handle the Google authentication callback

// This function is executed when a user successfully authenticates with Google

// You can access the user's information from the `profile` object

// and store it in your database or perform any other required actions

}

)

);

```

5. Configure the Facebook authentication strategy. Obtain the app ID and app secret by creating a new app in the Facebook Developer Console:

```javascript

passport.use(

new FacebookStrategy(

{

clientID: FACEBOOK\_APP\_ID,

clientSecret: FACEBOOK\_APP\_SECRET,

callbackURL: '/auth/facebook/callback',

},

(accessToken, refreshToken, profile, done) => {

// Add your custom logic to handle the Facebook authentication callback

// This function is executed when a user successfully authenticates with Facebook

// You can access the user's information from the `profile` object

// and store it in your database or perform any other required actions

}

)

);

```

6. Export the `passport` object so that it can be used in other files:

```javascript

module.exports = passport;

```

7. In your main application file, e.g., `app.js` or `index.js`, import the `passport` object and initialize it:

```javascript

const passport = require('./passport');

app.use(passport.initialize());

```

8. Set up the routes for the authentication callbacks. This will handle the redirection after successful authentication:

```javascript

app.get('/auth/google/callback', passport.authenticate('google'), (req, res) => {

// Add your custom logic to handle the Google authentication callback

// This function is executed after successful authentication with Google

// You can redirect the user to the desired page or perform any other required actions

});

app.get('/auth/facebook/callback', passport.authenticate('facebook'), (req, res) => {

// Add your custom logic to handle the Facebook authentication callback

// This function is executed after successful authentication with Facebook

// You can redirect the user to the desired page or perform any other required actions

});

```

9. Customize the routes and callback functions according to your application's needs. You may need to define additional routes for authentication initiation and user registration.

10. Set up the authentication endpoints for Google and Facebook. These endpoints will initiate the authentication process:

```javascript

app.get('/auth/google', passport.authenticate('google', { scope: ['profile', 'email'] }));

app

.get('/auth/facebook', passport.authenticate('facebook'));

```

11. You can now use the configured Passport.js strategies to handle user authentication using Google and Facebook in your application.

Remember to replace `GOOGLE\_CLIENT\_ID`, `GOOGLE\_CLIENT\_SECRET`, `FACEBOOK\_APP\_ID`, and `FACEBOOK\_APP\_SECRET` with the actual credentials obtained from the respective developer platforms.

By following these steps, you will be able to configure Passport.js and set up authentication strategies for Google and Facebook in your backend application.

STEP3:

To create a user model and define the user schema using Mongoose, follow these steps:

1. Create a new file, e.g., `**User.js**`, in a `models` directory or any suitable location in your project's directory structure.

2. Install the Mongoose package if you haven't already:

```

**npm install mongoose**

```

3. In the `**User.js**` file, import the required modules:

```javascript

**const mongoose = require('mongoose');**

**const { Schema } = mongoose;**

```

4. Define the user schema using the `Schema` class from Mongoose:

```javascript

**const userSchema = new Schema({**

**name: {**

**type: String,**

**required: true,**

**},**

**email: {**

**type: String,**

**required: true,**

**unique: true,**

**},**

**password: {**

**type: String,**

**required: true,**

**},**

**// Add any other fields you want to store for the user**

**});**

```

Customize the schema fields according to your requirements. In the example above, we have included `name`, `email`, and `password` fields.

5. Create the user model by compiling the schema into a model using the `mongoose.model` method:

```javascript

**const User = mongoose.model('User', userSchema);**

```

6. Export the `User` model so that it can be used in other files:

```javascript

**module.exports = User;**

```

7. You can now use the `User` model to interact with the MongoDB database and perform operations such as creating new users, retrieving user information, updating user details, and more.

Remember to set up the connection to your MongoDB database before using the `User` model.

By following these steps, you will be able to create a user model and define the user schema using Mongoose in the `User.js` file.

STEP 4:

To implement the user registration route in the `**auth.js**` file, follow these steps:

1. Create a new file, e.g., `**auth.js**`, in a `routes` directory or any suitable location in your project's directory structure.

2. In the `**auth.js**` file, import the required modules and define the necessary routes:

```javascript

**const express = require('express');**

**const router = express.Router();**

**const bcrypt = require('bcrypt');**

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

```

3. **Implement the user registration route** by defining a route handler for the registration endpoint. This handler will receive the user's registration data, validate it, hash the password, create a new user document, and save it to the database:

```javascript

**router.post('/register', async (req, res) => {**

**try {**

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

**// Validate the user input (e.g., check for required fields, email format, etc.)**

**// ...**

**// Hash the password using bcrypt**

**const hashedPassword = await bcrypt.hash(password, 10);**

**// Create a new user document with the hashed password**

**const newUser = new User({**

**name,**

**email,**

**password: hashedPassword,**

**});**

**// Save the new user to the database**

**await newUser.save();**

**res.status(201).json({ message: 'User registered successfully' });**

**} catch (error) {**

**res.status(500).json({ error: 'An error occurred' });**

**}**

**});**

```

Customize the validation logic and error handling based on your requirements.

4. Export the `router` so that it can be used in other files:

```javascript

**module.exports = router;**

```

5. In your main application file (e.g., `**app.js**` or `index.js`), **import the `auth.js**` file and register the authentication routes:

```javascript

**const express = require('express');**

**const app = express();**

// ...

**const authRoutes = require('./routes/auth');**

**app.use('/auth', authRoutes);**

```

Adjust the path `/auth` according to your desired URL structure. By following these steps, you will be able to implement the user registration route in the `auth.js` file. This route will handle user registration by validating the input, hashing the password, creating a new user document, and saving it to the MongoDB database.

STEP 5:

To implement the user login route in the `**auth.js**` file, follow these steps:

1. Open the existing `**auth.js**` file where you implemented the user registration route.

2. Define a new route handler for the login endpoint, usually as a `POST` request:

```javascript

**router.post('/login', async (req, res) => {**

**try {**

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

**// Validate the user input (e.g., check for required fields, email format, etc.)**

**// ...**

**// Find the user in the database by their email**

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

**if (!user) {**

**return res.status(401).json({ error: 'Invalid credentials' });**

**}**

**// Compare the provided password with the hashed password stored in the database**

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

**if (!passwordMatch) {**

**return res.status(401).json({ error: 'Invalid credentials' });**

**}**

**// Generate a JSON Web Token (JWT) for the authenticated user**

**const token = generateToken(user);**

**// Send the JWT back to the client**

**res.json({ token });**

**} catch (error) {**

**res.status(500).json({ error: 'An error occurred' });**

**}**

**});**

```

In this example, `generateToken` is a placeholder function that you need to define to generate the JWT for the authenticated user. You can use a library like `jsonwebtoken` to handle JWT generation.

3. Customize the validation logic, error handling, and JWT generation based on your requirements.

4. Export the `router` so that it can be used in other files:

```javascript

**module.exports = router;**

```

5. Save the changes to the `auth.js` file.

By following these steps, you will have implemented the user login route in the `auth.js` file. This route will handle user authentication by validating the input, comparing the provided password with the hashed password in the database, and generating a JSON Web Token (JWT) if authentication is successful.

STEP 6:

Step 7: Protect routes with authentication middleware

1. Create a new file named `**auth.js**` in the `**middleware**` directory.

2. In the `**auth.js**` file, import the required dependencies:

```javascript

**const jwt = require('jsonwebtoken');**

**const config = require('config');**

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

```

3. Create an authentication middleware function that will be used to protect routes:

```javascript

**function authMiddleware(req, res, next) {**

**// Get the JWT from the request headers**

**const token = req.header('x-auth-token');**

**// Check if the token is missing**

**if (!token) {**

**return res.status(401).json({ error: 'Access denied. Token missing.' });**

**}**

**try {**

**// Verify and decode the JWT**

**const decoded = jwt.verify(token, config.get('jwtSecret'));**

**// Fetch the user from the database based on the decoded user ID**

**const user = await User.findById(decoded.user.id);**

**if (!user) {**

**return res.status(401).json({ error: 'Access denied. Invalid token.' });**

**}**

**// Attach the user object to the request for further use in protected routes**

**req.user = user;**

**// Proceed to the next middleware or route handler**

**next();**

**} catch (error) {**

**res.status(401).json({ error: 'Access denied. Invalid token.' });**

**}**

**}**

```

In this example, `config.get('jwtSecret')` refers to the JWT secret key stored in your configuration file.

4. Export the `authMiddleware` function so that it can be used in other files:

```javascript

**module.exports = authMiddleware;**

```

Step 8: Implement social login (optional)

To implement social login using Google or Facebook, you will need to integrate their respective OAuth strategies into your authentication flow. Here is a high-level overview of the steps involved:

1. Set up the necessary OAuth credentials and obtain client IDs and secrets from Google and/or Facebook.

2. Install the required passport strategies and dependencies, such as `passport-google-oauth` or `passport-facebook`.

3. In the `**passport.js**` file, configure the desired authentication strategies for Google and/or Facebook. For example, you can use the `passport-google-oauth` strategy like this:

```javascript

**const GoogleStrategy = require('passport-google-oauth').OAuth2Strategy;**

**passport.use(new GoogleStrategy({**

**clientID: GOOGLE\_CLIENT\_ID,**

**clientSecret: GOOGLE\_CLIENT\_SECRET,**

**callbackURL: GOOGLE\_CALLBACK\_URL**

**}, (accessToken, refreshToken, profile, done) => {**

**// Custom logic to handle the Google authentication callback**

**// Verify the user, create a new user if necessary, and invoke the `done` callback**

**}));**

```

Similarly, you can configure the `passport-facebook` strategy for Facebook authentication.

4. In the `**auth.js**` file, create additional routes to handle the OAuth authentication flow with Google or Facebook. These routes will use Passport.js to authenticate the user and generate a JWT. The exact implementation will depend on your chosen strategy and requirements. Here's an example for the Google OAuth login route:

```javascript

**router.get('/google', passport.authenticate('google', { scope: ['profile', 'email'] }));**

**router.get('/google/callback', passport.authenticate('google', { failureRedirect: '/login' }), (req, res) => {**

**// Generate a JWT and redirect or send the token back to the client**

**});**

```

Adjust the routes and options according to your specific OAuth strategy and requirements.

Step 9: Store user data and retrieve past predictions

To associate predictions with authenticated users and store them in your MongoDB database, you will need to modify the relevant files based on your application's structure. Here's a general approach:

1. In the `routes` file handling the prediction submission, ensure that the user is authenticated. You can use the `authMiddleware` created in Step 7 to protect the route.

2. When a user submits a prediction, retrieve their user ID from the authenticated request object (`req.user.id`).

3. Store the prediction in the MongoDB database, associating it with the user ID. This can be done in the same route handler using Mongoose or your chosen database library:

```javascript

const prediction = new Prediction({

user: req.user.id,

// Other prediction properties

});

await prediction.save();

```

Adjust the code according to your database schema and requirements.

4. To retrieve past predictions for a user, you can create a new route or modify an existing route to handle the request. Use the user ID to query the predictions associated with that user:

```javascript

router.get('/predictions', authMiddleware, async (req, res) => {

try {

const userPredictions = await Prediction.find({ user: req.user.id });

res.json(userPredictions);

} catch (error) {

res.status(500).json({ error: 'An error occurred' });

}

});

```

Adjust the code based on your database schema and the desired response format.

Remember to import the necessary models, dependencies, and middleware in your files for proper functionality.