Authentication and Authorisation using passport

Let's break down the code step-by-step to understand how `passport`, `passport-local`, and `passport-local-mongoose` are used for user authentication in a Node.js application with Express and MongoDB.

1. Project Setup and Installation

First, we set up our project and install the necessary packages:

```bash
npm init -y
npm install express mongoose passport passport-local passport-local-mongoose
express-session
```

- **`express`**: A web framework for Node.js to create a server and handle routes.
- **`mongoose`**: An ODM (Object Data Modeling) library for MongoDB and Node.js.
- **`passport`**: A middleware for authentication in Node.js.
- **`passport-local`**: A strategy for Passport to authenticate users using a username and password.
- **`passport-local-mongoose`**: A Mongoose plugin that simplifies setting up a username and password in Mongoose.
- **`express-session`**: Middleware to manage user sessions.

2. Create a Mongoose User Model

Create a user model that uses `passport-local-mongoose` to handle password hashing and salting:

```
const mongoose = require('mongoose');
const passportLocalMongoose = require('passport-local-mongoose');

const userSchema = new mongoose.Schema({
  username: String,
  password: String,
});

userSchema.plugin(passportLocalMongoose);

module.exports = mongoose.model('User', userSchema);
```

- **`mongoose.Schema`**: Defines the structure of the documents in a collection.
- **`username` and `password`**: Fields in the schema for user authentication. Although we define `password` here, it will be managed and hashed by `passport-local-mongoose`.
- **`passportLocalMongoose`**: This plugin adds fields and methods to the schema to simplify handling user authentication. It provides methods like `register`, `authenticate`, `serializeUser`, and `deserializeUser`.

3. Set Up Express and Passport

In the main application file (`app.js`), set up the server, connect to MongoDB, and configure Passport for authentication:

```
const express = require('express');
const mongoose = require('mongoose');
const passport = require('passport');
const LocalStrategy = require('passport-local');
const session = require('express-session');
const User = require('./models/user'); // assuming your model is in a models
folder

const app = express();

mongoose.connect('mongodb://localhost/yourDatabaseName', { useNewUrlParser:
true, useUnifiedTopology: true });

app.use(express.urlencoded({ extended: true }));
```

- **Connecting to MongoDB**: Using Mongoose to connect to a MongoDB database.
- **`express.urlencoded`**: Middleware to parse incoming request bodies.

4. Session Configuration

Set up session management for persistent login sessions:

```
app.use(session({
  secret: 'yourSecret',
  resave: false,
  saveUninitialized: false,
}));
```

- **`express-session`**: Manages user sessions. `secret` is used to sign the session ID cookie. `resave` and `saveUninitialized` manage session behavior.

5. Passport Configuration

Configure Passport to use the `LocalStrategy` provided by `passport-local-mongoose`:

```
app.use(passport.initialize());
app.use(passport.session());

passport.use(new LocalStrategy(User.authenticate()));

passport.serializeUser(User.serializeUser());
passport.deserializeUser(User.deserializeUser());
```

- **`passport.initialize`**: Initializes Passport for authentication.
- **`passport.session`**: Middleware to manage persistent login sessions.
- **`LocalStrategy`**: Uses the `authenticate` method provided by `passport-local-mongoose` to handle username and password authentication.
- **`serializeUser` and `deserializeUser`**: Methods for encoding and decoding user data to and from the session.

6. Register Users

Create a route to handle user registration:

```
app.post('/register', async (req, res) => {
  try {
    const newUser = new User({ username: req.body.username });
    await User.register(newUser, req.body.password);
    res.redirect('/login');
  } catch (err) {
    res.send(err);
  }
});
```

- **`User.register`**: A method from `passport-local-mongoose` that creates a new user and hashes the password.
- **Error Handling**: Sends an error if registration fails.

7. Login Users

Handle user login using Passport's `authenticate` method:

```
app.post('/login', passport.authenticate('local', {
  successRedirect: '/dashboard',
  failureRedirect: '/login',
}));
```

- **`passport.authenticate('local')`**: Authenticates the user using the local strategy. If successful, redirects to `/dashboard`; otherwise, redirects to `/login`.

8. Logout Users

Create a route for logging out users:

```
app.get('/logout', (req, res) => {
  req.logout(function(err) {
    if (err) { return next(err); }
    res.redirect('/');
  });
});
```

- **`req.logout`**: Passport method to log out the user and destroy the session.
- **Redirect to Home**: After logout, the user is redirected to the home page.

9. Protect Routes

Middleware to protect routes so only authenticated users can access them:

```
function isLoggedIn(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  }
  res.redirect('/login');
}

app.get('/dashboard', isLoggedIn, (req, res) => {
  res.send('This is the dashboard page.');
});
```

- **`req.isAuthenticated()`**: Passport method to check if the user is authenticated.
- **`isLoggedIn` Middleware**: If the user is authenticated, proceed to the next function; otherwise, redirect to `/login`.

10. Starting the Server

Basic server setup:

```
// Route for home page
app.get('/', (req, res) => {
  res.send('Home page');
});

// Route for login page
app.get('/login', (req, res) => {
  res.send('Login page');
});

// Start the server
app.listen(3000, () => {
  console.log('Server started on port 3000');
});
```

- **Routes**: Define routes for the home and login pages.
- **`app.listen`**: Starts the server on port 3000.

Summary

This setup uses `passport-local-mongoose` to simplify user authentication with Passport.js. It handles the creation of user records, password hashing, and provides utilities to authenticate and manage user sessions. With this structure, you can register, log in, and protect routes easily in a Node.js application.