Handling Authentication in NodeJS (Express) with Passport Part 3— Authentication and Authorization

Sijuade Ajagunna
May 29, 2020 · 2 min read

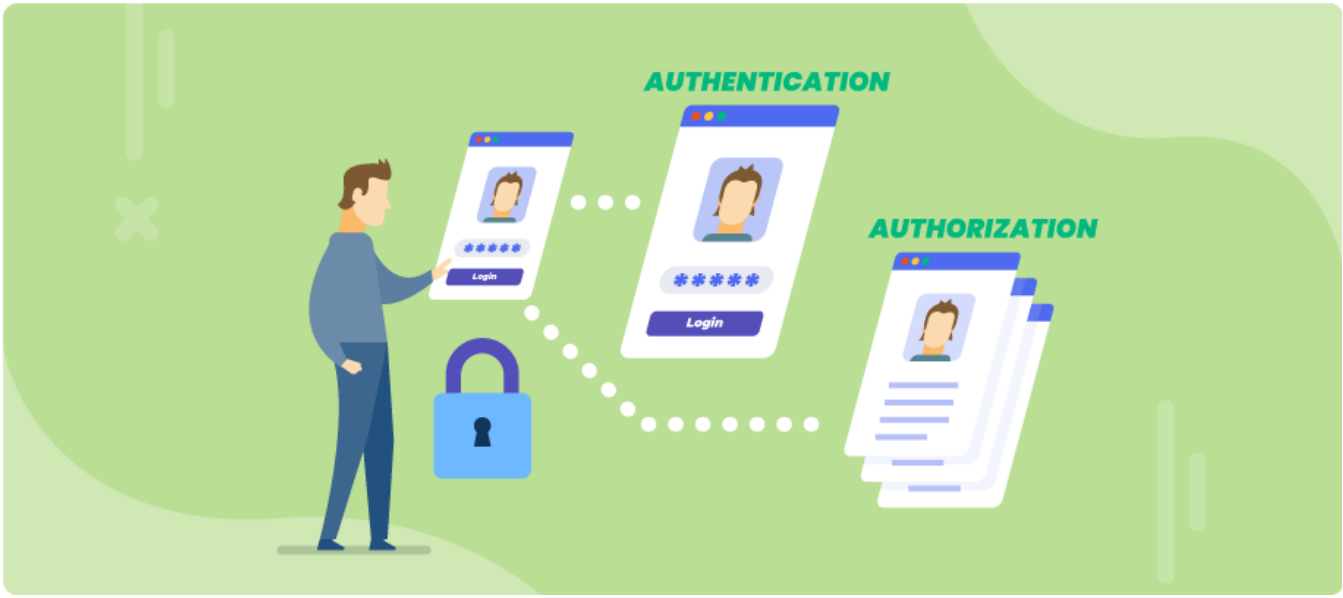

image credit: https://swoopnow.com/security-authentication-vs-authorization/

This is the third article in a series intended to help set up authentication and authorization in your server-side NodeJS applications using PassportJS.

Part 1

Part 2

You can get the full project from the GitHub Repository here.

So, our app *should be able to* authenticate users now, but we haven't tried that out yet as there's no way for a user to interact with it yet, so let's get around to fixing that.

Controllers

First, let's our controller file. In the controller folder, create an auth.controller.js file.

Next, let's write some methods to handle signup, login and one protected route.

```
import passport from "passport";
import debug from "debug";
import passportLocal from "../services/passport/passport-local";
import { ApplicationError, NotFoundError } from "../helpers/errors";

const DEBUG = debug("dev");

const createCookieFromToken = (user, statusCode, req, res) => {
  const token = user.generateVerificationToken();

  const cookieOptions = {
    expires: new Date(Date.now() + 10 * 24 * 60 * 60 * 1000),
    httpOnly: true,
    secure: req.secure || req.headers["x-forwarded-proto"] === "https",
  };

  res.cookie("jwt", token, cookieOptions);

  res.status(statusCode).json({
    status: "success",
    token,
    data: {
      user,
    },
  });
};

export default {
  signup: async (req, res, next) => {
    passport.authenticate(
      "signup",
      { session: false },
      async (err, user, info) => {
        try {
          if (err || !user) {
            const { statusCode = 400, message } = info;
            return res.status(statusCode).json({
              status: "error",
              error: {
                message,
              },
            });
          }
```

```
            createCookieFromToken(user, 201, req, res);
          } catch (error) {
            DEBUG(error);
            throw new ApplicationError(500, error);
          }
        }
      )(req, res, next);
    },

    login: (req, res, next) => {
      passport.authenticate("login", { session: false }, (err, user, info) => {
        if (err || !user) {
          let message = err;
          if (info) {
            message = info.message;
          }
          return res.status(401).json({
            status: "error",
            error: {
              message,
            },
          });
        }
        // generate a signed son web token with the contents of user object and return it in the
        createCookieFromToken(user, 200, req, res);
      })(req, res, next);
    },

    protectedRoute: async (req, res) => {
      res.status(200).json({
        status: "success",
        data: {
          message: "Yes you are. You are a Thor-n times developer",
        },
      });
    },
  };
```

We create a cookie on signup or log in which is sent over the browser or relevant REST client on subsequent requests.

Middleware

Next, we need to set up middleware for authorization. Users are barred or allowed entry into protected routes depending on their authorization status.

In the middleware folder, create an authenticate.js file and add the following to it:

```
import debug from 'debug';
import passportJWT from '../services/passport/config';
import { ApplicationError } from '../helpers/errors';

const DEBUG = debug('dev');
export default {
  authenticate: (req, res, next) => {
    passportJWT.authenticate('jwt', { session: false }, (err, user, info) => {
      if (err) {
        return next(err);
      }

      if (!user) {
        throw new ApplicationError(
          401,
          'invalid token, please log in or sign up',
        );
      }

      req.user = user;
      // DEBUG(user.userName);
      return next();
    })(req, res, next);
  },
};
```

authenticate.js hosted with ♥ by GitHub view raw

In simple terms, this function checks your request headers for your JWT, scans to see if there is a user associated with it and throws back an unauthorized error if there is none.

Routes

Finally, let's handle the routes. Create an auth.route.js file in the routes folder and add the following code to it:

```
import { Router } from "express";
import authController from "../controllers/auth.controller";
import catchAsync from "../middleware/catchAsync";
```

```
import authentication from '../middleware/authenticate';

const { signup, login, protectedRoute, logout } = authController;
const { authenticate } = authentication;

const authRouter = Router();

authRouter.post('/signup', catchAsync(signup));
authRouter.post('/login', catchAsync(login));
authRouter.get('/amiworthy', authenticate, catchAsync(protectedRoute));

export default authRouter;
```

auth.route.js hosted with ♥ by GitHub view raw

Import your ***auth.route.js*** file into your app.js and just after initializing Passport, mount the `authRouter` on a path. You can use this as a guide:

```
...
import authRouter from "./routes/auth.route";

...

passport.initialize()

app.use('/auth', authRouter);

...
```

app.js hosted with ♥ by GitHub view raw

And now, our small authentication app is good to go. Start your app by running `npm run dev` using Postman, or your favourite REST client, go to the start page. You can then try to sign up with the `/auth/signup` route and login with the `/auth/login` route.

You needn't bother with copying tokens on Postman as cookies are saved automatically for subsequent requests. With other clients, you can copy and paste the token in the authorization section of your request header as a Bearer token.

```
Authorization: Bearer <token>
```

You can try to access the protected `/auth/amiworthy` route, if it all went well, you should be granted access.

A common reason why authorization may fail is due to your JWT private and public keys. Ensure newline characters are completely replaced before adding them to your `.env` files or use an algorithm that doesn't require public and private keys if getting started with the RSA keys are proving difficult.

With that, we complete our work with the Passport Local Strategy. We'll try logging in with Google in the next article.

You can check out the project repo on GitHub.

Authorization Authentication Passportjs Security Token Mongodb