Implementing JSON Web Tokens & Passport.js in a JavaScript Application with React

Paige Niedringhaus
Sep 13, 2018 · 13 min read

Before you say it, I know, I know. There's already a literal ton of MERN tutorials out there showing how to use JWT (JSON Web Tokens) and Passport.js with Express.

But here's what every one of those tutorials failed to mention, and what mine will cover:

> HOW and WHY to use the various authentication flavors Passport offers (including `passport-jwt`), and the gotchas that tripped me up

for hours on end as I put together a user registration application.

I don't know what your experience has been, but the majority of the projects I've worked on professionally have come with authorization already prebuilt by someone else. So in addition to improving my MERN skills, I viewed this as an opportunity to learn more about security and authentication and different ways to do it.

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

source files and download the full MERN project here. Once it's correctly hooked up on the backend, the front end is purely React.

Before I jump into all that though, let me give you a quick run down of JWT and Passport.js authentication.

What is a JSON Web Token?

A JSON Web Token, according to the site is:

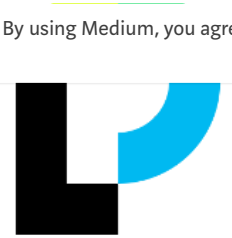
> *"…an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object." — JWT.io*

Essentially, JWTs are digitally signed tokens that can be verified and trusted, and they're becoming more and more popular for security and authorization between parties (like servers and clients) and for exchanging sensitive information, while verifying the token hasn't been tampered with or decoded.

I won't go into the details of how the tokens work, you can read all about that from a much more qualified source than I here, but suffice it to say, I chose to use JWT as part of my user verification strategy along with Passport.js to make my simple user registration application secure (and because I was curious to see if I could do it, since I didn't have a whole lot of hands on experience implementing authorization).

With that covered, the next piece of my authentication solution is Passport.js.

What is Passport.js & Why Should I Use it?



Passport is:

> *Simple, unobtrusive authentication for Node.js — Passport.js*

and it works exceedingly well with Express. Passport is authentication middleware for Node, which serves one purpose, to authenticate requests, in a modular way that leaves all other functionality to the application itself, making code cleaner, easier to maintain and provides a clear separation of concerns.

If you type the term 'JavaScript authentication middleware' or even just 'JavaScript authentication' into the Google search bar, Passport.js ranks within the top 5 search results. That's how ubiquitous this solution is in the JavaScript ecosystem.

Did I mention it boasts more than 500+ authentication strategies? It does. Whether you wish to log in with a simple username and password, with Github credentials, Facebook, oAuth, etc., there's probably a Passport.js strategy for it.

So it was a simple decision to choose Passport as part of my authorization strategy.

I'd like to add documentation on the site is pretty good, though some things like custom callbacks require much more careful reading (and trial and error, for me) to correctly set up. But I'm getting ahead of myself, I'll cover the gotchas in implementation.

How Do I Implement Them in Express.js (and a little React)?

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

Now on to the fun part, how do you implement both Passport.js and JWT into an Express/Node application? To be honest, it stumped me for a good while. But after numerous tutorials, re-readings of documentation and turning to Stack Overflow for help, I got to a place of understanding and satisfaction.

I should say, one of my goals for this app was to make it modular, make it super simple to add or remove functionality, so when you see my file structure, you'll notice all the routes for the CRUD functionality (create, read, update, delete) are built into separate files, and all the Passport authentication is handled in one, centralized file as well. Here's a look at the file structure of the API portion of my app.

The API File Tree

```
root/
├── api/
| ├── config
| | ├── passport.js
| | ├── jwtConfig.js
| ├── server.js
| ├── sequelize.js
| ├── package.json
| ├── models/
| | ├── user.js
| ├── routes/
| | ├── deleteUser.js
| | ├── findUsers.js
| | ├── loginUser.js
| | ├── registerUser.js
| | ├── updateUser.js
| ├── node-modules/
```

The `package.json` File

And here is the `package.json` and its dependencies so you can see exactly what I'm using.

```
{
  "name": "mysqlregistration-api",
  "version": "1.0.0",
```

```
    "start": "nodemon server.js --exec babel-node --presets es2015"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/paigen11/mysqlRegistration.git"
  },
  "author": "Paige Niedringhaus (paigen11@gmail.com)",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/paigen11/mysqlRegistration/issues"
  },
  "homepage": "https://github.com/paigen11/mysqlRegistration#readme",
  "dependencies": {
    "babel-cli": "^6.26.0",
    "babel-preset-es2015": "^6.24.1",
    "babel-preset-stage-0": "^6.24.1",
    "bcrypt": "^3.0.0",
    "body-parser": "^1.18.3",
    "cors": "^2.8.4",
    "express": "^4.16.3",
    "jsonwebtoken": "^8.3.0",
    "morgan": "^1.9.0",
    "mysql2": "^1.5.3",
    "nodemon": "^1.18.3",
    "passport": "^0.4.0",
    "passport-jwt": "^4.0.0",
    "passport-local": "^1.0.0",
    "sequelize": "^4.38.0",
    "sequelize-cli": "^4.0.0"
  }
}
```

Nothing super fancy here: just jsonwebtoken, passport, passport-local and passport-jwt.

The dependencies include a few extras like `babel` so I can use ES6 syntax in my Node.js app, `bcrypt` for password hashing and `sequelize` as my MySQL ORM, but the things you need to focus on are `jsonwebtoken`, `passport`, `passport-local` and `passport-jwt`. These are the necessities for this blog.

The `server.js` File

I'll start with the `server.js` file first, as it requires the least explanation. This file is purely for starting the server, initializing the use of Passport in the app and setting up the routes and parsing of requests from the client side.

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

```
import Cors from 'cors';
import bodyParser from 'body-parser';
import logger from 'morgan';
import passport from 'passport';
const app = express();

const API_PORT = process.env.API_PORT || 3000;

require('./config/passport');

app.use(Cors());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.use(logger('dev'));
app.use(passport.initialize());

require('./routes/loginUser')(app);
require('./routes/registerUser')(app);
require('./routes/findUsers')(app);
require('./routes/deleteUser')(app);
require('./routes/updateUser')(app);

app.listen(API_PORT, () => console.log(`Listening on port ${API_PORT}`));

module.exports = app;
```

A little ES6 in the imports and consts, but what you should focus on is the passport.initialize() and the routes.

Pretty straightforward right? Great. Moving on.

You might also have noticed the `require('./config/passport');`, this is the next step I'll go over. The JWT and Passport configuration, inside of the folder named `config`.

The `jwtConfig.js` File

The JWT config is very simple, it's the secret required by JWT to encode and decode the tokens. Typically, this would be stored as an environmental variable in a file that's not checked in to Github, but to show how this works, I've set it in here.

```
module.exports = {
  secret: 'jwt-secret',
};
```

This is the jwtConfig.js file in the config folder.

The `passport.js` File

The real setup for all my Passport authentication comes in the `passport.js` file. Here it is.

```
import jwtSecret from './jwtConfig';
import bcrypt from 'bcrypt';

const BCRYPT_SALT_ROUNDS = 12;

const passport = require('passport'),
  localStrategy = require('passport-local').Strategy,
  User = require('../sequelize'),
  JWTstrategy = require('passport-jwt').Strategy,
  ExtractJWT = require('passport-jwt').ExtractJwt;

passport.use(
  'register',
  new localStrategy(
    {
      usernameField: 'username',
      passwordField: 'password',
      session: false,
    },
    (username, password, done) => {
      try {
        User.findOne({
          where: {
            username: username,
          },
        }).then(user => {
          if (user != null) {
            console.log('username already taken');
            return done(null, false, { message: 'username already taken' });
          } else {
            bcrypt.hash(password, BCRYPT_SALT_ROUNDS).then(hashedPassword => {
              User.create({ username, password: hashedPassword }).then(user => {
                console.log('user created');
                return done(null, user);
              });
            });
          }
        });
      } catch (err) {
        done(err);
      }
    },
  ),
);

passport.use(
  'login',
  new localStrategy(
    {
      usernameField: 'username',
      passwordField: 'password',
      session: false,
    },
    (username, password, done) => {
      try {
        User.findOne({
          where: {
```



```
          } else {
            bcrypt.compare(password, user.password).then(response => {
              if (response != true) {
                console.log('passwords do not match');
                return done(null, false, { message: 'passwords do not match' });
              }
              console.log('user found & authenticated');
              return done(null, user);
            });
          }
        });
      } catch (err) {
        done(err);
      }
    },
  ),
);

const opts = {
  jwtFromRequest: ExtractJWT.fromAuthHeaderWithScheme('JWT'),
  secretOrKey: jwtSecret.secret,
};

passport.use(
  'jwt',
  new JWTstrategy(opts, (jwt_payload, done) => {
    try {
      User.findOne({
        where: {
          username: jwt_payload.id,
        },
      }).then(user => {
        if (user) {
          console.log('user found in db in passport');
          done(null, user);
        } else {
          console.log('user not found in db');
          done(null, false);
        }
      });
    } catch (err) {
      done(err);
    }
  }),
);
```

Yes, all this is part of the passport.js file.

I realize this is a lot of code to take in. If you'd like to see just the code for Passport and the routes with Passport authentication, I created a few gists here with the code.

I'll walk through the two types of Passport implementation happening here: passport-local for the `register` and `login` methods and passport-jwt for the `jwt` method.

Passport-local uses a username and password, and passport-jwt uses a JWT payload to verify the user is legit.

from the React client on the front end.

Once the passport-local strategy has been passed a username and password (which I verify that both inputs are at least filled before ever calling back to the server), the first check I do is with Sequelize, my SQL ORM (like Mongoose for MongoDB), to determine if that username exists in the database. If it returns `null`, the authentication fails (no user in the db matches), and typically, a `401 Unauthorized` would be thrown back from the server to the client with no further information.

Passport Gotcha #1: Info in Error Handling

This type of error handling, to me, was the most frustrating thing about Passport. The lack of information to let a user (or myself) know what the actual error was beyond `401 Unathorized`.

But there's a better solution that requires a little extra work, with **custom callbacks**. I'll go into more detail about the callback soon, but for now, you can see if there's some sort of error, instead of returning a `return done(null, user);`, passport can pass back `return done(null, false, { message: 'bad username or passwords don't match' });`. This message can then be passed back from the server to the client, actually telling the user what the problem is (which is what I wanted for this app). If you prefer just to tell the user their authentication failed, but not disclose why, that's cool too. Your choice, but I want to show how error messaging can be done.

The rest of the passport-local strategy is pretty self-explanatory, on `register` the user's password is hashed and salted with the encryption package `bcrypt`, and then when the `login` method is invoked, it hashes the newly entered password and checks the passwords with bcrypt's `compare` function before returning either a positive or negative on the verification.

Now, that brings me to the passport-jwt strategy, which is named `jwt`. This is the authentication that's called on the protected routes in the application: `findUsers`, `updateUser`, `deleteUser`. For the jwt strategy, the JWT is passed back from the client with each call to the server (I passed mine in an authorization header with the key: `JWT`), which is extracted and then decoded using the secret (which is stored in another file, but should really be an environment variable known only to the system).

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

returned with `done(null, user);`, if the user is found or with `done(null, false);` if it's not (which should almost never happen because the JWT includes the username in its encrypted form, so unless that's somehow been tampered with or the db has been, it should be able to find the user).

Passport Gotcha #2: Passport-Local Wants Return, Passport-JWT Does Not

This brings me to my second gotcha which tripped me up for a good bit of time; not all passport strategies require the same resolution. If you're looking closely, you'll see the passport-local strategies both have `return done(null, user);` but the passport-jwt strategy has `done(null, user);`. See the difference? It's minuscule, but having (or removing that `return`), is the difference between that user data being passed back from the middleware to the server or not.

And it halted my progress for a good few hours, before the kindness of Stack Overflow helped me work through the issue (it's the first time I've actually had to ask SO for an answer I couldn't find already, but it was well worth it). So be aware of when to `return` or not.

Ok, so I've walked through the two Passport strategies and the three methods I'm using for middleware authentication in my program, now on to after the authentication on the server.

The `registerUser.js` File

The next step is how to implement these newly minted methods inside of the various routes. Here's the `registerUser` route.

```
import User from '../sequelize';
import passport from 'passport';

module.exports = app => {
  app.post('/registerUser', (req, res, next) => {
    passport.authenticate('register', (err, user, info) => {
      if (err) {
        console.log(err);
      }
      if (info != undefined) {
```

```
        req.logIn(user, err => {
          const data = {
            first_name: req.body.first_name,
            last_name: req.body.last_name,
            email: req.body.email,
            username: user.username,
          };
          User.findOne({
            where: {
              username: data.username,
            },
          }).then(user => {
            user
              .update({
                first_name: data.first_name,
                last_name: data.last_name,
                email: data.email,
              })
              .then(() => {
                console.log('user created in db');
                res.status(200).send({ message: 'user created' });
              });
          });
        });
      }
    })(req, res, next);
  });
};
```

Notice the closure style invocation of this route — that's what makes messages from the authentication layer to the server and client possible.

As you can see, this implementation of passport looks a little different than most examples out there, and it's because I'm using the custom callback version of Passport, which requires a closure call. I'm using this so that the error message that's passed back if Passport's authentication fails can be sent from the server to the client instead of the obscure `401 Unauthorized`.

Because this is being implemented as a callback (which is why you see `(req, res, next)` not once, but twice in these scripts; it gives me access to the `(err, user, info)` from the Passport middleware. The `info` is what I'm interested in — that's the message being sent back, so if the info is anything besides null, that means the authentication failed and I can then send the message to the client to let them know why.

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

callbacks work with passport-local, the little method `req.logIn()` must be called before the `user` data is handled if it comes back successfully from the middleware.

This is documented in the Passport documentation, but it's importance is not stressed as much as I would like (and I missed it the first few times trying to get custom callbacks to work), which is why I highlight it now.

It must happen, and once that's done, I'm able to take the extra inputs from the client side for registering a user, find that same user created in the database during the passport-local `register` call, and update the user file with the extra info. I could have passed this extra data through to the middleware as well, but I want Passport to only handle authentication, not user creation as well. Modularization, remember.

Plus, if the authentication were to be split out into a separate service with a separate database of just encrypted usernames and passwords, this would make it easier to do so, then use the username or ID to find and update the corresponding user record in this registration service.

Once all that's been successfully taken care of, a 200 HTTP status and success message are sent back to the client.

The `loginUser.js` File

Seeing the `loginUser` route now should make more sense.

```
import User from '../sequelize';
import jwtSecret from '../config/jwtConfig';
import jwt from 'jsonwebtoken';
import passport from 'passport';

module.exports = app => {
  app.get('/loginUser', (req, res, next) => {
    passport.authenticate('login', (err, user, info) => {
      if (err) {
        console.log(err);
      }
      if (info != undefined) {
        console.log(info.message);
        res.send(info.message);
      } else {
        req.logIn(user, err => {
          User.findOne({
            where: {
```

```
            const token = jwt.sign({ id: user.username }, jwtSecret.secret);
            res.status(200).send({
              auth: true,
              token: token,
              message: 'user found & logged in',
            });
          });
        });
      }
    })(req, res, next);
  });
};
```

Same style of closure for the login route.

The same style of custom callbacks and closures are used, the biggest difference is that once the user is successfully verified and located in the database, the JWT token is generated using the `jwt.sign();` function, which sets the username as the ID passed in the JWT, and encrypted by the secret I set earlier.

Once again, if all this works successfully, a 200 HTTP status is sent with a boolean I named `auth` set to true for the client side, the newly generated token and a short login success message.

The `findUsers.js` File

And last, but not least, here's the `findUsers` route. I used the same passport-jwt authentication for the `updateUser` and `deleteUser` routes, so I'll just show this one as an example.

```
import passport from 'passport';

module.exports = app => {
  app.get('/findUser', (req, res, next) => {
    passport.authenticate('jwt', { session: false }, (err, user, info) => {
      if (err) {
        console.log(err);
      }
      if (info !== undefined) {
        console.log(info.message);
        res.send(info.message);
      } else {
        console.log('user found in db from route');
        res.status(200).send({
```

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

```
            email: user.email,
            username: user.username,
            password: user.password,
            message: 'user found in db',
          });
        }
      })(req, res, next);
    });
};
```

I used custom callbacks for all my Passport authenticated routes (I appreciate good, clear error handling), so this code style should look routine by now.

This time, when `passport.authenticate()` is called, I implement the JWT strategy defined in the `passport.js` file. The same `(err, user, info)` gets passed back from the middleware, but there's no call from `req.logIn()` this time. Instead, I just return the `user` object found during the authentication and pass all the required fields to the client along with the `auth` boolean and success message. The JWT token is still stored in local storage on the client side so there's no need to regenerate or pass it to the client again as well. Which brings me to my final passport gotcha.

Passport Gotcha #4: Passing Authorization Headers Correctly

This one's not exactly a Passport-specific gotcha, but it is yet another thing that tripped me up.

As I said, I pieced together my (now much more whole) understanding of JWT and Passport from a bunch of other tutorials and documentation, and one of those tutorials led me astray. It had me passing the JWT back to the client in something besides true authorization headers, which is the way that I chose to have my passport-jwt strategy extract the JWT payload from the client request. For this reason, I wanted to touch briefly on one piece of the front end code, where I pass back the token in the proper format, so you don't have to waste the time debugging like I did.

```
async componentDidMount() {
    let accessString = localStorage.getItem('JWT');
    if (accessString === null) {
      this.setState({
        isLoading: false,
        error: true
```

```
    await axios
      .get('http://localhost:3003/findUser', {
        params: {
          username: this.props.match.params.username,
        },
        headers: { Authorization: `JWT ${accessString}` },
      })
      .then(response => {
        this.setState({
          first_name: response.data.first_name,
          last_name: response.data.last_name,
          email: response.data.email,
          username: response.data.username,
          password: response.data.password,
          isLoading: false,
          error: false,
        });
      })
      .catch(error => {
        console.log(error.data);
      });
  }
```

The client side call that loads user data after passing back the JWT stored in the user's local storage.

I'm using the popular promise-based HTTP client Axios to make my server calls, but what you really need to notice is the `headers` section. Before the call to the server happens, I extract the JWT from local storage using `localStorage.getItem('JWT')` (the key I set with the JWT value when I passed the token forward from the `loginUser()` route on the server side), and then I set it using `headers: { Authorization: `JWT ${accessString}` }` and a little ES6 string interpolation. That way, if my authorization headers had more than one value to parse out, it would still be easy for passport-jwt to extract the correct JWT payload info by finding the string associated with 'JWT'.

And that's it, I just wanted to cover it because it blocked me for a while and only through API testing with Insomnia, could I confirm my JWT token was actually working, and suss out that I was passing authorization headers from the client side incorrectly.

Conclusion

I know I wrote a lot (and provided a bunch of code snippets and gists), but authentication is not a simple concept. Nor is it truly possible to really make an app so

To make Medium work, we log user data. By using Medium, you agree to our Privacy Policy, including cookie policy.

to withstand people with malicious intent.

Just be sure to look out for the gotchas I highlighted around Passport — and you'll be on your way to a more secure site.

Thanks for reading, I hope this proves helpful and gives you a better understanding of implementing Passport authentication and using JSON web tokens. Claps and shares are very much appreciated!

If you enjoyed reading this, you may also enjoy some of my other blogs:

- Postman vs. Insomnia: Comparing the API Testing Tools
- Using Docker & Docker Compose To Improve Your Full Stack Application Development
- Sequelize: The ORM For SQL Databases with NodeJS

. . .

References and Further Resources:

- Passport.js: http://www.passportjs.org/
- JSON Web Tokens: https://jwt.io/
- MERN App with Passport & JWT Repo: https://github.com/paigen11/mysql-registration-passport
- Gists of Passport implementation with routes: https://gist.github.com/paigen11/c72c8c20da9cd440f45025a1b05e5e58
- Axios: https://www.npmjs.com/package/axios