Authentication/Authorization

Hunter Jorgensen

Authentication and Authorization

In the last assignment, we've started seeing authentication and authorization.

Authentication is the process of confirming who you are. It usually involves providing a username and password, and more recently other steps.

Authorization is figuring out what to provide for you. Once you are *authenticated*, how do you determine what information to provide to the user? There are complex ways to manage this, but our system will be pretty simple.

Authentication: SFA, 2FA, RedFA...

A modern piece of authentication is often composed of confirming your identity through multiple ways. In addition to providing a password, you may also need to input a code that was texted to you, click on a link that is emailed to you, etc.

Single-Factor Authentication (SFA/1FA) - requires a combination of password and username to login.

Two-Factor Authentication (2FA) - in addition to a username/password, it requires an additional piece of informational (a unique key sent to your phone, etc.)

Multi-Factor Authentication (MFA) - in addition to 2FA, requires another independent means of authentication (finger print, YubiKey, etc.)

Cookies

Remember in your React apps how if you refresh the page, you lose all current progress and data? However, major websites like Amazon, Google, etc. are able to recall that you're logged in/logged out even if you refresh the page. How do they do this?

The most common answer is cookies.

A cookie is a piece of data that is sent back and forth with each HTTP/S request that allows. There, we can store (encrypted) user information on each request to ensure that the correct user is logged, has permissions to the data, etc.

Authentication Options

Once a user has confirmed their identify (by logging in), it is incumbent that the app will continue to have the user to be securely logged in: it would be frustrating for a user to keep logging in every time they clicked a button on your website!

Fortunately, there are 3 common ways to maintain authentication for a user: session, JWT and OAuth/OpenID.

JWT

JWT, or **JSON Web Token** or token-based authentication, stores all the relevant user information (userID, username, etc.) in a token that is sent as part of every request/response to the server. This token is stored in a cookie that is included with every response. Typically the information on the cookie is encrypted and decrypted only with a secret key that exists on the server.

Implementing JWT In Node

First, install some libraries that will help us handle a JWT:

```
npm install --save jsonwebtoken cookie-parser express-session
```

Then you need to add a UNIQUE SECRET to you .env file (and then remember to add to Heroku) that you can create at the root of your Node app

```
SUPER_SECRET=“my_secret_ssshhhhh_1234567890”
```

Initialize cookie-parse

Add the following lines to server.js:

```
const cookieParser = require('cookie-parser');

...

app.use(cookieParser());
```

Cookie-parser is Express middleware (i.e. code that modifies the request/response structure) that lets directly intercept and modify a request coming in to your Express/Node app.

Goal: Register/Login, LoggedIn and Logout

The goal here is to create 4 new APIs:

Register: an API that accepts a username/password in the body, created the user and attaches the user info to the cookie

Login: an API that accepts a username/password in the body, and if those match an existing

Authenticate/LoggedIn: check the current user is logged in

Logout: this removes the cookie

Set the Cookie When User Register/Logs In

When the user registers or logs in, we'll create a cookie that tracks whatever payload information we're interested in. No information is being saved on the server side: instead, we'll decode the token information once the next request comes back, and do logic accordingly.

Demo

Note that we if look at the Network tab and this request and then the cookie, we'll see a 'token': we added that! Also, this will match the response setCookie

ALSO, you will want to add this logic to your registration section

Creating our own Middleware!

We have already learned about Express routes (app.get(“/..”,...), app.post…, etc.), but sometimes we want to interact with a request BEFORE it reaches out route. This is what middleware is for!

When a new request comes in, we’ll want to decode the token and include it in our request object for our logic to use or to protect resources from getting accessed. If the token is missing or invalid, we can send an error back so the frontend can redirect or show an error.

Demo

Refactor Route

Now that we have this middleware, we can make our routes that are based on getting information from the user more robust. For instance, we can update our pokemon.controller get logic (and pretty all other similar API's.)

Demo

Update React to Drop Username Logic

Since the user is getting tracked via JWT now, we no longer need to keep do a lot of the username logic we had on the frontend (since we can move that to the backend).

Demo

Notice how this cleans up our logic quite a bit and makes our URL safer and easier to use?

Reroute on Missing JWT

There is the case where a user is logged out or their JWT token has expired. In that situation, we would want to reroute the user to the login page so they can decide to login or register.

We'll want to add an API that returns true/false if the user is logged in, and do a redirect if they're not. For this we'll use a Higher Order Component (i.e., a React Component that manages other Components) to check if the user is logged in or not!

Demo

Session Based Authentication

In the **session** based authentication, the server will create a session for the user after the user logs in on the server side, with a session ID stored on a cookie. While the user stays logged in, the cookie would be sent along with every subsequent request. The server can then compare the session id stored on the cookie against the session information stored in the memory to verify user's identity and sends response with the corresponding state.

Note: there are some greats libraries (particularly Passport's Local Strategy) that handle this logic quite well, but I find that the syntax is hard understand for newer web developers. You are welcome to use it in your assignment or project, but I am not using it to make it more clear what is going on.

Setting Up A Session

Going from JWT to Session based authentication is pretty simple. Express-session will abstract a lot of the logic out, but the changes below are pretty minor (we're just writing to a session object on the request):

Demo

What is happening under the hood? Express-session attaches a unique session ID to the cookie of every request. When that session ID is found in the cookie, it pulls from memory the associated session and attaches it to the request object.

You can test this by logging in and checking out the network tab

More Permanent Session Storage

If you restart the server, you will find that your session is lost! This is because session data is stored in memory on the server, and when the server is reset, all session data is lost. In order to handle this, we can add session data to our Mongo instance.

```
npm install -s connect-mongo
```

Demo

Auth Component-Based Logic

Just like we can redirect a page when a user is logged in or logged out, we can make modify certain components based on whether a user is logged in or logged out:

Demo

Difference between JWT and Session

Hopefully by now the difference is clear, but if not

JWT (JSON Web Token) are cookies that are passed back and forth between the server. The server encrypts/decrypts the token to manage the state, but the server itself doesn't store anything related to the user. This is usually easier to maintain (no storage!) but has higher security concerns.

Session (different from browser sessions) use a cookie to track a session on the backend, but the session itself exists on the backend. It usually needs to be store in some database, but protects the user information more.

Other Concepts: OAuth

OAuth is an open-standard authorization protocol/framework that describes how one service can allow an authenticated user to access a resource on another service service. OAuth was first designed in 2010, and OAuth2.0 released in 2012.

An example of this is logging on to a website and being able to login with another companies credentials like Facebook, Google, etc. OAuth is simply a set of rules and practices services must follow to be able to authenticate a user.

OAuth is about authorization: it allows a service to access your data, information, etc. on your behalf.

Other Concepts: OpenID

OpenID is often compared to OAuth, but is different in that it focuses on the authentication part of the process: i.e., the step of proving who you are. Logging in with services like Amazon, Facebook, Apple, etc. may incorporate OpenID into their OAuth framework and can allow you to access multiple sites with a single user instance or password.

Example: when you go to Airbnb and sign up with Google. Google asks for your login details (OpenID) so that you can Airbnb your contact/personal information (OAuth).

Other Concepts: SSO

SSO, or Single sign-on, is the process of logging in once but having access to different software products. The common example of this is logging in with Google, but gaining access to websites such as YouTube, Gmail, etc.

Lab: Improving our Authentication

Follow the steps to implement JWT- or session-based authentication.

- Add this code to your user.controller.js and the logout logic to your React App. Make it so that when logout is called, the user is successfully logged out (2 points)
- Add a role field to the user schema so that users can be a ‘TRAINER’ or ‘MASTER’. Update the loggedin, or add a new API, so that there is a component or page that can only be seen if the user is a ‘MASTER’. (4 points)