Node (and React) Auth

Introduction

Here we will implement auth on the back end using JWTs (JSON Web Tokens). The implememtation is largely focused on the backend, and then we will build in some React structures to block certain routes in the future. For now our main game is to be able to identify a user, and also to block certain routes if the user is not authorised.

I'm trying to strip out the relevant auth here, rather than build the whole app. To get to a stage where this will be useful:

1. You need to be connected up to Mongo.
2. You need to be able to store users in the DB.
3. You should be hashing and salting the passwords before they enter the DB (BCrypt).
4. There should be a route to get a new user into the DB (a signup).
5. You should be using `dotenv` to store your secrets in a `.env` file.

Some Theory

We will need to understand a couple of key things to get our heads in the game here. There are also plenty of terrific articles out there for you to investigate further.

HTTP

HTTP is a stateless protocol, meaning that each request and response does not know anything about each other. This means that to have some persistence of the user, you need some way of confirming that the person at the computer is who they say they are, so that they can access the things on the site that are most relevant to them, but do so without logging in for every request.

There are several methods to do this, but a common one is to store a token in the browser that will be sent with each request, and validate the user. It will act as proof that this user has logged in, and that they are welcome to view this or that data.

This is stored in a session, or in a cookie. There are slight differences to each, but the main one is that a cookie is persistent beyond the broswer session (meaning that there is no need to login again after the broswer is shut, unless the cookie has since expired). Sessions will expire when the browser is shut.

A cookie is just a little bit of information that resides in the computer, and will be sent to the server with each request. One piece of information that we will store in the cookie to verify the user is a JWT.

JWTs

JWTs are the token that is going to be sent from the backend to the front, and here we will be storing it in a cookie. You can read more about JWTs easily on the web, but the important things to understand about them are:

1. They have three parts: a. The header: which tells you how the third part was hashed (very simply encoded (base-64)). b. The payload: some information that will be passed to the user, and held in the

token (again, just base-64 encoded). c. The signature: the hashed outcome of the header, the payload, and importantly, the secret. This is the element that ensures that the header and payload were not meddled with.
2. You should not store any sensitive information in them, as base-64 encoding is very simply decoded.
3. The can last forever, or expire, and we have to find ways to cope with these options.

Summary of Action

We are going to be passing a token back and forth.

1. We will have a user register, and create and send them a token (it will be stored in the header, and dealt with by the browser automatically).
2. Whenever the user makes a request to the server in any way, this token will be sent to the backend in the header (this will be done automatically).
3. When the user hits a route that has auth on it, we will check the token, and either grant access or not depending on the outcome.
4. If they login, a new token will be issued (in a cookie again).
5. When they log out, we will remove the cookie (and thus the token).

In our server.js

Let's look at some code:

```
const mongoose = require('mongoose')
const express = require('express')

// These are the two new requires we need to get things going with auth.
`cookie-parser` just helps us to read the cookie in the request.
const cookieParser = require('cookie-parser')
const dotenv = require('dotenv')
const jwt = require('jsonwebtoken')

const cors = require('cors')

// This just gives us access to the file with the secrets via dotenv.
dotenv.config()

// Models
const User = require('./models/User.js')
const Task = require('./models/Task.js')

// Our custom middleware.
const withAuth = require('./middleware')

// Using dotenv to get the secrets from the .env file. We get our secret
for our JWT creation, as well as the Mongo connection URL.
const mongoUri = process.env.MONGO_URL
const secret = process.env.SECRET

const port = process.env.PORT || 4090
const app = express()
app.use(cors())

// We will use this middleware to be able to read the cookie's info from
the incoming request. (And the other two do similar jobs for URL encoding,
and JSON data, from the body.)
app.use(cookieParser())
app.use(express.json())
app.use(express.urlencoded())

mongoose.connect(mongoUri, { useNewUrlParser: true, useUnifiedTopology:
true },
  (err, database) => {
      if (err) {
        throw err;
      } else {
        console.log(`Successfully connected to ${mongoUri}`);
      }
    }
);

// This is just a regular route, set up for testing, and with no auth
running on it.
app.get('/home', (req, res) => {
  console.log('hit the home route')
  res.send('Welcome!');
});

// This is a little route that will help us to see if the auth is running.
app.get('/protected', withAuth, (req, res) => {
  console.log('hit the protected route')
  res.send('The password is potato');
})

// Could be a handy route to return the login status of the user.
app.get('/checkToken', withAuth, (req, res) => {
    console.log('Token is was checked out')
    res.sendStatus(200);
})

// POST route to register a user. We will be putting the user into the DB
(with salted and hashed passwords), and also giving the user a token at
the same time, so that they are logged in from the point of registration.
app.post('/register', (req, res) => {

    console.log('Hitting the register route.')
    console.log(req.body)

    // The request body should contain the email and password of the user,
whcih we extract.
    const { email, password } = req.body

    // Creating a new Mongoose User object
    const user = new User({ email, password });
    // Saving the user
    user.save((err) => {
        // If it didn't work, let the front-end know
        if (err) {
            console.log(err)
            res.status(500)
                .send(`Error registering new user. Please try again. Err:
${err}`);

        } else {
            // Here we are setting the JWT payload to be just the user
email (and this is shorthand for `payload = { email: email }` )
            const payload = { email }

            // Using the package to create the JWT, with the payload, our
secret, and an expiry time
            const token = jwt.sign(payload, secret, {
                expiresIn: '1h'
            })

            // This is Express helping us to set the cookie field on the
response. We set the cookie section as a token, say what that token is
(the one we prepared earlier), and also add the httOnly flag, meaning that
it is inaccessible to the JavaScript Document.cookie API in the browser.
            res.cookie('token', token, { httpOnly: true })
                // We set the status to 200, all ok.
                .status(200)
                // and send a little JSON that might be useful on the
front-end.
                .send({
                    message: "You have signed up",
                    email: email
                })
        }
    })
})

// Here is our route to authenticate a user, which is really a login.
app.post('/authenticate', (req, res) => {
    console.log('authenticating')
    const { email, password } = req.body
    // checking if the user exists, based on the req.body
    User.findOne({ email }, (err, user) => {
        // Error route - something went wrong checking the DB
        if (err) {
            console.error(err);
            res.status(500)
                .json({
                error: 'Internal error please try again'
            });
        // We checked the DB, but didn't find such a user with that email
        } else if (!user) {
            console.log('no such user')
            res.status(401)
                .json({
                    error: 'Incorrect email or password'
                });
        // Found a user with that email
        } else {
            // Here we are checking the password that has been hashed wih
BCrypt (so the package with salt and hash the password again, and check it
against the hash in the DB).
            user.isCorrectPassword(password, (err, same) => {
                // Error path
                if (err) {
                    console.log('mixup')
                    res.status(500)
                        .json({
                            error: 'Internal error please try again'
                        });
                // Password doesn't match route
                } else if (!same) {
                    console.log("details didn't match somehow")
                    res.status(401)
                        .json({
                        error: 'Incorrect email or password'
                    })
                // All is ok and matches, so we will send the token
                } else {
                    console.log('Sending the token after authentication.')
                    // Setting what we want to be the payload (the email)
                    const payload = { email }
                    // Using the package to sign the JWT
                    const token = jwt.sign(payload, secret, {
                        expiresIn: '1h'
                    });
                    console.log(token)
                    // Setting the cookie on the response. We are calling
it 'token', it's value is the token that we created, and we are setting
`httpOnly`
                    res.cookie('token', token, { httpOnly: true })
                    // sending the response with the payload as JSON
                    res.json(payload)
                }
            });
        }
    });
});

// Simple route to tell the browser to clear out the cookie (called token)
app.get('/logout', (req, res) => {
  console.log('Logout route hit')
  res.clearCookie('token');
  res.send({ success: true });
})
```

```
app.listen(port, () => console.log(`Server listening at
http://localhost:${port}`))
```

Now we need the middleware

This will be the function that does the checking of the token. We can run this function on the whole app, but as you can see in the implementation above, we are just going to be running this on particular routes.

Let's look at the code:

```
// middleware.js
// We need a package to help us make and verify JWTs. There is plenty of
good information out there on how JWTs are formed. The main thing we need
to understand here is that we need a package to help us to verify and sign
JWTs, which will be the token passed back and forward between the front
and backend to have persistence of user.
const jwt = require('jsonwebtoken')
// This will help us by retrieving important information from our .env
file, and keeping it out of GitHub.
const dotenv = require('dotenv')

// Sets up dotenv to work its magic.
dotenv.config();

// Here we are simply drawing out the secret from the .env file. It is
vital to keep the secret from being discovered, and out of our code
(obviously). It is the part of the JWT that helps to form the signed part
of the JWT. While the first two parts of a JWT are encoded (meaning they
can be decoded), the last part of a hash formed of the first two parts and
the secret. The JWT is compromised if the secret is known.
const secret = process.env.SECRET

// `withAuth` is middleware. In Express, middleware is a function that
performs some processing on the incoming request before handing it through
to the route (or not passing on to the route in some instances). They take
three arguments, the request object, the response object, and a `next()`
function. The next function is called when you are ready to move to the
next middleware.
const withAuth = (req, res, next) => {

    // Here we are extracting the token from the cookie. It is called
token.
    const token = req.cookies.token;

    // If we don't find a token, we send back an 'unauthorised' status
(401), and a message to that effect.
    if (!token) {
        console.log('No dice mate, get a token.')
        res.status(401).send('Unauthorised: No token provided');
    // If the token is there..
    } else {
        // we need to verify it (whether it's still valid, or been
tampered with).
        jwt.verify(token, secret, (err, decoded) => {
            // If the token is there and invalid, we let them know.
            if (err) {
                console.log('Invalid token route. Broken token.')
                res.status(401).send('Unauthorized: Invalid token');
            // Otherwise we will let them through (using the next()
function), and we can also set whatever else we like on the request
object. Here we just set the key email to be the users email, in case we
want to access that (and it will be useful in many cases, as we will often
need to find various resources related to the user). The id would also be
very handy.
            } else {
                console.log('Break on through.')
                req.email = decoded.email;
                next();
            }
        });
    }
}

module.exports = withAuth;
```

More code?

This isn't all you need for the full API, but from this you can get an idea of how to put together enough code to have some routes protected. The pattern is the same for any route that requires auth.

Adding React?

React Routes and Auth