User-Authentication in NodeJS

Agenda

- User Model
- PassportJS
- Signing Up
- Encrypting passwords
- Logging In

Setup

- Google and download "Postman". This is an application that allows you to quickly make http requests.
- We'll be using this to test our functionality as we build it.

Setup - Model

- We need to first create user model that has the fields to store the email and password for each user

```
//models/user.js

module.exports = function(sequelize, DataTypes) {
    var User = sequelize.define('users', {
        email: DataTypes.STRING,
        password: DataTypes.STRING,
        firstName: DataTypes.STRING,
        lastName: DataTypes.STRING
    }, {
        classMethods: {
            associate: function(models) {

            }
        }
    });

    return User;
};
```

Approach

- We're going to be building this out in pieces and connecting all the dots at the end.
- On one side, we're going to build the routes that will accept requests to 'Sign Up' and 'Log In'.
- On the other side, we're going to building the model and logic that fulfills these requests.
- The last step is to test that everything is working end-to-end.

Authentication Using PassportJS

- PassportJS is a node module whose sole purpose is to authenticate users and authorize requests, which it does through an extensible set of plugins known as strategies.
- You can use Passport to allow users to use their Facebook, GMail, LinkedIn and other accounts as authentication into your website.
- Since we are handling user accounts ourselves we will be using the Passport Local strategy.

Passport - Installation

- We'll need to install two node modules to accomplish this.

```
$ npm install passport --save
$ npm install passport-local --save
```

Sign Up - Routes

- Let's create a dummy route that accepts sign up requests.
- Create a new file in routes for users called 'user.js'.
- This file will simply export a function that accepts a passport object

```
//routes/user.js

var express = require('express');
var router = express.Router();

module.exports = function(passport) {

    // POST /user/signup
    router.post('/signup', function(req, res) {
        res.send('got the signup request')
    });

    return router;
};
```

Routes - Setup

- Next step is to incorporate this new route and passport into our application.
- After adding the below, test the route 'localhost:3000/api/user/signup' in Postman. You should get back a response

```
// add the following to server.babel.js

var passport = require('passport');

app.use(passport.initialize());

var userRoutes = require('./routes/user')(passport); // <-- pass the passport object into userRoutes

app.use('/api/user', userRoutes);
```

Passport - Setup

- We need to add the logic for our Local Strategy that will tell passport whether the information the user is sending over meets our requirements.
- Thought exercise: for signing up, what information do we want to validate?
- Let's get to work! We'll need to create the folder and files first:

```
$ mkdir strategies && cd strategies
$ touch passport-local.js
```

passport-local.js - Sign Up

- We are going to export a function that accepts a passport object.
- To define our local sign-up strategy, we'll call the 'use' function on the passport object and provide the following:
  - 'local-signup': the name we're calling this strategy.
  - usernameField: the field on the request that will contain the username
  - passwordField: the field on the request that will contain the password
  - passReqToCallback: whether to pass the incoming request to the callback that will handle the logic for signing up.
  - the callback: the function that will perform the logic and actually create the new user in our system

Let's see this in action...

```
var LocalStrategy = require('passport-local').Strategy;

module.exports = function(passport) {

    passport.use('local-signup', new LocalStrategy({
        usernameField: 'email',
        passwordField: 'password',
        passReqToCallback: true
    }, processSignupCallback));   // <<-- more on this to come

};
```

passport-local.js - Sign Up (cont)

- At this point we need to implement the 'processSignupCallback' function.
- This function is called with the following parameters:
  - request: because we passed 'passReqToCallback : true' from the step above, the first parameter to this function will be the incoming request.
  - email: the email of the user wanting to sign up.
  - password: the password of the user wanting to sign up.
  - done: this is called the 'verify callback' and it is a function we call when we've completed processing the sign up request.

PassportJS - the "done" callback

- The 'done' callback provides us a way of telling passport we're done executing the code we want to execute.
- There are three arguments that 'done' gets called with eg:
  `done (error, result, message)`:
  - error: if there was an error we can pass it to the done function and it will forward it along.
  - result: if the outcome was successful this is where we pass the result, if not, we pass the value 'false'.
  - message: this is an optional param that allows us to pass a message through the call chain.

```
function processSignupCallback(request, email, password, done) {
    // first search to see if a user exists in our system with that email
    UserModel.findOne({
        where: {
            'email' :  email
        },
        attributes: ['id']
    })
    .then(function(user) {









    });
}
```

```
function processSignupCallback(request, email, password, done) {
    // first search to see if a user exists in our system with that email
    UserModel.findOne({
        where: {
            'email' :  email
        },
        attributes: ['id']
    })
    .then(function(user) {
        if (user) {
            // user exists call done() passing null and false
            return done(null, false, 'That email is already taken.');
        } else {









        }
    });
}
```

```
function processSignupCallback(request, email, password, done) {
    // first search to see if a user exists in our system with that email
    UserModel.findOne({
        where: {
            'email' :  email
        },
        attributes: ['id']
    })
    .then(function(user) {
        if (user) {
            // user exists call done() passing null and false
            return done(null, false, 'That email is already taken.');
        } else {
            // create the new user
            var userToCreate = req.body; // make this more secure

            UserModel.create(userToCreate)
            .then(function(createdRecord) {
                //once user is created call done with the created user
                createdRecord.password = undefined;
                return done(null, createdRecord);
            });
        }
    });
}
```

Passport - Serialization

- We'll discuss this further, but for now we'll need to add the 'serializeUser' function to the passport object
- For our purposes this function will just do the following:

```
// inside of config/passport-local.js
module.exports = function(passport) {

    passport.serializeUser(function(user, done) {
      done(null, user.id);
    });

    passport.use('local-signup', new LocalStrategy({
        usernameField : 'email',
        passwordField : 'password',
        passReqToCallback : true
    }, processSignupCallback));
}
```

SignUp

- Let's hookup our sign up route to use passport.
- To do this we just need to call the authenticate function on passport and pass the name of the strategy we want to use.
- Once authenticated, Passport will add a "login" function to the request. We'll need to call this function with the authenticated user.

Sign Up - Logic

```
//routes/user.js

var express = require('express');
var router = express.Router();

module.exports = function(passport) {
    // POST /api/user/signup
    router.post('/signup', function(req, res, next) {
        passport.authenticate('local-signup', function(err, user, info) {
            if (err) {
                return next(err); // will generate a 500 error
            }
            if (!user) {
                return next({ error : true, message : info });
            }

            req.login(user, function(loginErr) {
                if (loginErr) {
                    return next(loginErr);
                }
                return res.json({
                    email: user.email,
                    id: user.id,
                });
            });
        })(req, res, next);
    });
    return router;
};
```

Local Strategy Initialization

- We need to add the following line to initialize or local strategy implementation

```
var passport = require('passport');

app.use(passport.initialize());

require('./strategies/passport-local')(passport);  // <-- add this line


var userRoutes = require('./routes/user')(passport);

app.use('/api/user', userRoutes);
```

Sign Up - Validation

- Test this out in Postman by making a POST request to localhost:3000/user/signup
- The request body needs to contain:

```
{
    "email": "test1@gmail.com",
    "password": "12345",
    "firstName": "myFirst",
    "lastName": "myLast",
}
```

- You should get back a response that has the newly created user.

Sign Up Considerations

- We probably don't want to issue the response with all the data that was passed in (especially the password)
- We can probably get away with just sending back the id, createdAt, and updatedAt.
- If we look at the record that was created in the database, we'll see the password is the same raw string that was passed in.
- We NEVER want to save passwords like this. It is considered extremely insecure.
- To secure our data, we will be encrypting the password.

bcrypt

- In order to encrypt the passwords we'll be using a node module called 'bcrypt'.
- At a high level, bcrypt is a function that accepts two arguments: a string and a random input and generates a 'hash' of that string (an encrypted string).
  - The first argument is the string we want to encrypt.
  - The second argument is what's called a 'saltRound'.
  - bcrypt uses the saltRound to generate a 'salt' which is the random input that is fed into the function that creates the hash.
  - The more rounds, the more random (secure) the data. However there is a time cost with having a lot of rounds, we will be using 10.

bcrpyt Analogy

```
// concat
var result = concat('Hello ', 10); // result = 'Hello 10'

var result = concat('My age is ', 10); // result = 'My age is 10'



// bcrypt
var result = bcrypt('myPassword', 10); // result = '&93h82khs20932jo'

var result = bcrypt('yourPassword', 10); // result = 'kjih&#Hj2n23928'
```

Password Encryption - Installation

- Let's start off by installing the bcrypt module

```
$ npm install bcrypt --save
```

bcrypt - Example

- We'll call the 'hash' function passing in the user's password and the saltRounds.
- Lastly, we'll pass in the callback function that will contain the hash. This is what we'll be saving in the database.
- You should save your saltRounds value in a configuration file
- Let's integrate this in our passport Sign Up process

```
const saltRounds = 10;

bcrypt.hash(user.password, saltRounds, function(err, hash) {
  // Store 'hash' in your password DB.
});
```

```
var bcrypt = require('bcrypt');

function processSignupCallback(req, email, password, done) {
    UserModel.findOne({
        where: {
            'email' :  email
        },
        attributes: ['id']
    })
    .then(function(user) {
        // check to see if theres already a user with that email
        if (user) {
            return done(null, false, 'That email is already taken.');
        } else {
            var userToCreate = req.body;

            bcrypt.hash(userToCreate.password, 10, function(err, hash) {
               userToCreate.password = hash;
               UserModel.create(userToCreate)
               .then(function(createdRecord) {
                    createdRecord.password = undefined;
                    return done(null, createdRecord);
               });
            });
        }
    });
```

Sign Up - Validation

- Test your Sign Up request is properly working.
- You should see a random string in the password column when you create a new user.
- Congratulations you've successfully allowed your users to sign up for your application.
- Next step: Logging In

Login - Setup

- Good news! We've done a lot of the work required with setting up passport.
- We just need to add the logic that determines whether a user can log in.
- Let's get to work!

Login - Local Strategy

- In the same module.export that we used for sign up, let's define a new login strategy.

```
// add to strageies/passport-local.js
module.exports = function(passport) {

    passport.serializeUser(function(user, done) {
      done(null, user.id);
    });

    passport.use('local-signup', new LocalStrategy({
        usernameField : 'email',
        passwordField : 'password',
        passReqToCallback : true
    }, processSignupCallback));

    passport.use('local-login', new LocalStrategy({
        usernameField : 'email',
        passwordField : 'password',
    }, processLoginCallback));
};
```

Login - processLoginCallback

- This is the callback where we'll be putting our logic that determines if the request is a valid login
- Thought exercise: what determines if a login attempt is valid?

```
function processLoginCallback(email, password, done) {
    // first let's find a user in our system with that email
    User.findOne({
        where: {
            'email' :  email
        }
    })
    .then(function(user) {
        if (!user) {
            return done(null, false, "No user name found with provided email")
        }

        // make sure the password they provided matches what we have
        // (think about this one, before moving forward)



    });
}
```

```
function processLoginCallback(email, password, done) {
    // first let's find a user in our system with that email
    User.findOne({
        where: {
            'email' :  email
        }
    })
    .then(function(user) {
        if (!user) {
            return done(null, false, "No user name found with provided email")
        }

        // make sure the password they provided matches what we have
        // (think about this one, before moving forward)
        bcrypt.compare(password, user.password, function(err, result) {
            if (!result) {
                return done(null, false, "Invalid Password for provided email")
            }
            user.password = undefined;
            return done(null, user);
        });
    });
}
```

Login - Routes

- Back in our routes/user.js file we'll need to add the route for logging in.
- We'll also need to denote which passport strategy to user for logging in.

```
// add this to routes/user.js

// POST /api/user/login
router.post('/login', function(req, res, next) {
    passport.authenticate('local-login', function(err, user, info) {
        if (err) {
            return next(err); // will generate a 500 error
        }
        if (! user) {
            return next({ error : true, message : info });
        }

        req.login(user, function(loginErr) {
            if (loginErr) {
                return next(loginErr);
            }
            return res.json({
                email: user.email,
                id: user.id
            });
        });
    })(req, res, next);
});
```

Login - Validation

- Test this new route in Postman. Send a request to 'localhost:3000/api/user/login'
- The request should be a POST request with the following in the body.
- You should get back the complete user object.

```
// POST localhost:3000/user/login with body:

{
    "email": "some email",
    "passwort": "myPassport",
}
```

Summary

- In order to enabled user sign up and login to our system we just performed the following steps.
  - Added our User table to the database.
  - Installed PassportJS.
  - Implemented the sign up functionality using passport-local.
  - Added bcrypt for encrypting passwords.
  - Implemented the login functionality using passport-local.
- What's next?
  - We want to keep track of who has logged in and who hasn't. Enter: The JSON Web Token

User Authorization

- Once a user has signed up or logged in, we want to give them access to certain information they didn't have before.
- We also want to prevent non-authenticated users from accessing this information as well.
- In order to accomplish this we'll be using a technology called a JSON Web Token (jwt).

JWT

- A jwt is essentially a long string that has certain information encoded in it.
- Your sever will be the only server that knows how to take the string, decode it and pull the information out of it.
- Some information we typically put in the jwt:
  - The user's id
  - An expiration date of the token
  - Anything else that makes sense for your application.

```
//example token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIsImlhd
CI6MTQ3ODgwNzMxMCwiZXhwIjoxNDc4ODA3MzQwfQ.VPvXni
WBG5O3UBWy8i5Ki79lY9W2Slyhfv5L8r8WzyA
```

JWT - Strategy

- The high level strategy is when the user signs up or logs in:
  - The server will generate the jwt.
  - It will return the token in the response of the sign up or log in requests
  - The client will then save the token in the browser's local storage.
  - From then on, each time the client wants to make a request it will need to pass the token to the server.
  - The server will decode the information
  - Then it will verify the token is valid/hasn't expired/belongs to that user etc
  - If all the checks pass, it will allow the request to go through.

JWT - Installation

- Let's start by installing the node module needed to generate these web tokens

```
$ npm install jsonwebtoken --save
```

jsonwebtoken usage

- Once we create a user we'll need to also create a token, encode the id into the token and save the user with the updated token value.
- The jwt module has a function call sign that takes the following parameters:
  - An object containing any additional information you want to encode.
  - The jwtSecret: this is what enables your server (and only your server) to decode any incoming tokens
  - An expiration time in seconds (eg 60 * 60 = 1 hour)
- Let's update our processSignupCallback so that we save the token in the User model.

```
var jwt = require('jsonwebtoken');

...

bcrypt.hash(userToCreate.password, 10, function(err, hash) {
  userToCreate.password = hash;

  User.create(userToCreate)
  .then(function(createdRecord) {
    jwt.sign({id: createdRecord.id}, 'MySuperDuperSecret', {expiresIn: 60 * 60}, function(err, token) {
      createdRecord.token = token;
      return done(null, savedUser);
    });
  });
});
```

Exercise - Logging In

- Add the needed code in your processLoginCallback so that when a user logs in, a jwt is generated.

Exercise - Solution

```
bcrypt.compare(password, user.password, function(err, result) {
  user.password = undefined;

  if (!result) {
    return done(null, false, "Invalid Password for provided email")
  } else {
    jwt.sign({id: user.id}, 'MySuperDuperSecret', {expiresIn: 60 * 60}, function(err, token) {
      user.token = token;
      return done(null, savedRecord);
    });
  }
});
```

JWT - Routes Update

- Now that we're properly creating the token, we need to properly return it in the response back to the client
- In your login and sign up routes, add an field for the token in the response

```
return res.json({
    email: user.email,
    id: user.id,
    token: user.token
});
```

JWT - Client Side

- Now that we're returning the token in the response, we'll need to save it on the browser's local storage.

```
fetch('/api/login', {
  method: 'POST',
  mode: 'same-origin',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(userAuthForm)
})
.then(response => response.json())
.then((response) => {
  if (response.error) {
    alert (response.message)
  } else {
    localStorage.setItem('token', response.token)
  }
})
```

JWT - Checkpoint

- We're now at a place where our server is generating the token
- It's returning it in the response with the login and sign up

Protected Routes

- We're at the point now where we can introduce protected routes that should be only accessible to logged in users.
- Let's start with creating a 'profile' route that displays a logged in user's information.

```
//routes/profile.js

var express = require('express');
var router = express.Router();
const models  = require('../db');

var User = require('../db').users;

router.get('/profile/:id', function(req, res, next) {
    User.findById(req.params.id)
    .then(function(user) {
        res.json({
            email: user.email,
            firstName: user.firstName,
            lastName: user.lastName,
            id: user.id
        })
    });
});

module.exports = router
```

Profile Routes

```
//server.babel.js
var profileRoutes = require('./routes/profile');
app.use('/api/protected', profileRoutes)
```

Sanity Check

- At this point we're not verifying any tokens, we just want to make sure our route is working.
- In postman, create a request for '/api/protected/profile/<someId>'
- You should see the response returning with the users information

Verify Tokens

- In order to verify a token is valid we'll be using another passport strategy

```
npm install passport-jwt --save
```

Passport-Jwt

- Now we'll need to write the code that does the actual verification and the code that tells passport when to do so.
- In the 'strategies' directory, create a file called 'passport-jwt.js'
- We're going to put code in the next slide that accomplishes this.

```
//strategies/passport-jwt.js

const JwtStrategy = require('passport-jwt').Strategy;
const ExtractJwt = require('passport-jwt').ExtractJwt;

const options = {
    jwtFromRequest: ExtractJwt.fromHeader('authorization'),
    secretOrKey: config.jwtSecret,
}

module.exports = function(passport) {
    passport.use(new JwtStrategy(options, function(jwt_payload, done) {
        User.findById(jwt_payload.id)
        .then(function(user) {
            if (user) {
                // user was found successfully
                done(null, user);
            } else {
                // no user was found for that id
                done(null, false, 'No user was found for the token provided');
            }
        });
    }));
};
```

Passport-Jwt - Explanation

- passport-jwt needs to know where the header is going to be located in each request.
  - There are a few different options here. We'll be using the ExtractJwt.fromHeader('authorization') option
  - So all incoming requests that require authorization will need to pass the token in the header under the 'authorization' key
- The function we pass in will contain the decoded jwt_payload. This will have the user id we set earlier in it.
- We need to go through and search for a user with that given id.
- Once we find the user we'll call the done function passing it through

Passport-Jwt - Routes

- Now that we've written the code that actually performs the verification we'll need to tell express which routes require the authorization.
- To do this we'll need to create an identifier in our routes to indicate that a route requires authorization.
- We're going to use 'api/protected' to denote this.

```
//server.babel.js

app.use('/api/protected', function(req, res, next) {
  passport.authenticate('jwt', {session:false}, function(err, user, jwtError) {
    if (user) {
      req.login(user, null, () => {})
      next()
    } else  {
      next(jwtError)
    }
  })(req, res, next)
});

app.use('/api/protected', profileRoutes)
```

Passport-Jwt - Routes Explanation

- For any incoming requests that are '/api/protected' we are going to call the authenticate function on passport and pass in the 'jwt' strategy to denote we want to verify the token
- When we do this the verification code we wrote above will execute and if successful, will pass us back a user
- if we don't get a user that means we had a jwtError occur
  - Either the token wasn't a real token
  - Or it was expired
  - Or our server didn't know how to read it
- If we get back a user then we'll log that user in and proceed to the next route handler
- If we got back an error we'll pass it on to the error handler

Passport-Jwt - Sanity Check

- Try to go back in postman and access the same 'api/protected/profile/<someId>' url
- You should get back an authentication error.
- Add the 'authorization' header to your request (you'll need an actual token for this)
- You should see a successful response with the user's information in it