

Homework W14: Passport

Reverse Engineering Code



Contents Page

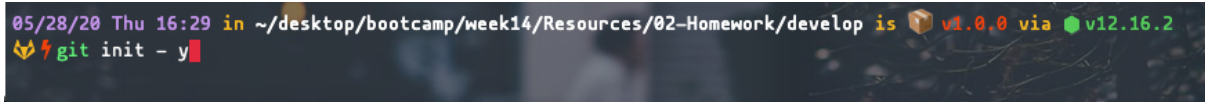
Dependencies and Server.js	p. 2 - 3
HTML and Routing	p. 4 - 8
Passport	p. 9 - 11
Sequelize	p. 12 - 15
Additions	p. 16
References	p. 17

>By Joshua K. Bader

Step One: Dependencies and Server.JS

Aim: To connect to localhost: 8080

Open terminal and inside your project file write `npm init -y` for default or `npm init` if you would like to make a custom Package.JSON. Your package.JSON will hold all the relevant metadata for your project including dependencies and start scripts. This can later be edited for include tests, watch etc.



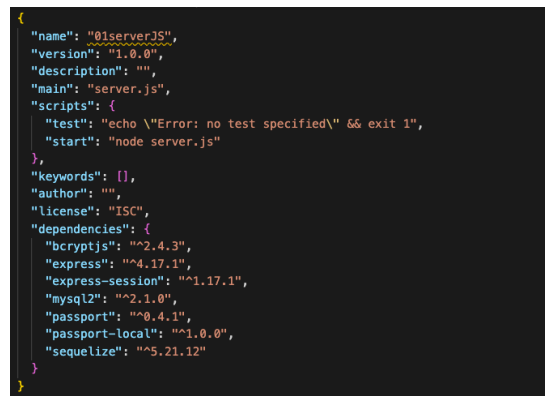
Next you will need to install your dependencies. For this project we are using express, passport and Sequelize.

For passport we are using `passport-local` which requires express-session, passport-local¹ and bcrypt for password hashing².

For Sequelize you will need to also install mySQL 2³.

In terminal type

1. npm install express passport sequelize
2. npm install bcryptjs express-session passport-local
3. npm install mySQL2 --save



Next we will need set up our server.js file and ensure that it makes a connection within our local host.

Open Visual Studio Code then locate your project folder and create a new file called `server.js`. This should be in the same folder as both your package.json and package-lock.json.

Inside `server.js` you will need to require express, set up your port number, initiate your middleware and finally add a console log to indicate express has connected with local host. You can add Heroku port now out of habit if you wish to avoid doing it in the future. Below is a snapshot of the initial setup.

¹ Jared Hanson, "Passport-Local", Passport.JS, <http://www.passportjs.org/packages/passport-local/>

² "Bcrypt", NPM, <https://www.npmjs.com/package/bcrypt>

³ "Sequelize", NPM, <https://www.npmjs.com/package/sequelize>

```

homeworkw14 > 01serverJS > js server.js > ...
1 // Requiring necessary npm packages
2 const express = require("express");
3
4 // Setting up port and requiring models for syncing
5 const PORT = process.env.PORT || 8080;
6
7 // Creating express app and configuring middleware needed for authentication
8 const app = express();
9 app.use(express.urlencoded({ extended: true }));
10 app.use(express.json());
11 app.use(express.static("public"));
12
13 app.listen(PORT, function() {
14   console.log("==> 🌐 Listening on port %s. Visit http://localhost:%s/ in your browser.", PORT, PORT);
15 }
16 );

```

Keynotes

- *Express.urlencoded*: will parse incoming requests and replaces the need to use the body-parser package with express. In this example, extended true means that the querystring library is passed but if you only want the querystring without the library then set extended to false.
- *Express.JSON()*: parses incoming requests as JSON and is required for express.urlencoded to work.
- *Express.static("public")*: allows express to access files within the "public" folder⁴.

Conclusion: Open up your terminal inside your project file and type, `Node server.js`. If everything works you should then get the reply below. You are now connected to localhost:8080

```

06/11/20 Thu 17:19 in homeworkw14/01serverJS on ʘ master [?] is 📦 v1.0.0 via
v12.16.2 took 7s
🐿️ ⚡ node server.js
==> 🌐 Listening on port 8080. Visit http://localhost:8080/ in your browser.

```

⁴ "Express()", Express, <https://expressjs.com/en/api.html#express>

Step Two: HTML and Routing

Aim: To get allow Member.html and Login.html to links to work. Write into the address bar /login, /members or /signup and go to the corresponding html page.

Before setting up your routes you will need to create and build your `public` folder. Inside your project create a new folder called `public` and then inside `public` make 3 html files called login.html, members.html and signup.html.

First build you initial document and attach the head tags. To speed up the process we are going to use bootstrap and jQuery so please include a link to both bootstrap in your head tag and a script tag for jQuery.

Login.html

Start with the `signup.html` page and when finished it should look something like the image below.

```
homework14 > 02-Html Files > public > login.html > html
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5    <title>Passport Authentication</title>
6    <meta charset="UTF-8">
7    <meta name="viewport" content="width=device-width, initial-scale=1">
8    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/lumen/bootstrap.min.css">
9    <link href="styleSheets/style.css" rel="stylesheet">
10 </head>
11
12 <body>
13
14   <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
15 </body>
16
17 </html>
18
```

Next you want to add in your form which should include 2 inputs labelled `Email` and `Password` as well as a submit button titled `Login`. Make sure to set your email input type to email. This will allow the email to be validated. Please give your form a H2 header with the title "Login Form" and include a link below your form that has a `href` tag targeting "/" inside a p tag that says 'or sign up here'

Add in a navbar element but do not add any additional buttons. For now this will be for consistency amongst all three pages as the members.html page will house a logout button in the navbar.

Next create your links to your JavaScript and CSS files located inside your public folder. We will create these later. Refer to image.

Note: There should be an authenticator for password on the login page

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Passport Authentication</title>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/lumen/bootstrap.min.css">
  <link href="styleSheets/style.css" rel="stylesheet">
</head>

<body>
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <div class="navbar-header">
      </div>
    </div>
  </nav>
  <div class="container">
    <div class="row">
      <div class="col-md-6 col-md-offset-3">
        <h2>Login Form</h2>
        <form class="login">
          <div class="form-group">
            <label for="exampleInputEmail1">Email address</label>
            <input type="email" class="form-control" id="email-input" placeholder="Email">
          </div>
          <div class="form-group">
            <label for="exampleInputPassword1">Password</label>
            <input type="password" class="form-control" id="password-input" placeholder="Password">
          </div>
          <button type="submit" class="btn btn-default">Login</button>
        </form>
        <br />
        <p>Or sign up <a href="/">here</a></p>
      </div>
    </div>
  </div>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
  <script type="text/javascript" src="js/login.js"></script>
</body>

</html>
```

signup.html

Now that your login page is complete copy everything and paste your code inside signup.html. Both pages require the same information so there is no need to make anything drastically different.

Change the following:

- Form title is `Sign Up Form`
- Button label is `Sign up`

- Javascript src is `signup.js`
- `p` tag should say `or login` and the corresponding `a` tag should say `here`

For this page we will be using an authenticator so that if you sign up with an email already registered in the database an error will be thrown. For now copy the code from the above image and ensure you id="alert".

```
<div style="display: none" id="alert" class="alert alert-danger" role="alert">
  <span class="glyphicon glyphicon-exclamation-sign" aria-hidden="true"></span>
  <span class="sr-only">Error:</span> <span class="msg"></span>
</div>
```

Members.html

Finally with the members page you want to copy and paste everything from your `login.html`, delete everything within the `body` besides your script tags and the navbar. Refer to the adjacent image.

Now inside your navbar add an `a` tag which has a href link to `/logout` and add a container with a H2 element that contains a spam tag. Ensure inside your spam tag that you have a class with `member-name`.

```
Resources > 02-Homework > Develop > public > members.html > ...
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <title>Passport Authentication</title>
6   <meta charset="UTF-8">
7   <meta name="viewport" content="width=device-width, initial-scale=1">
8   <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/lumen/bootstrap.min.css">
9   <link href="stylesheets/style.css" rel="stylesheet">
10 </head>
11
12 <body>
13   <nav class="navbar navbar-default">
14     <div class="container-fluid">
15       <div class="navbar-header">
16         <div>
17         </div>
18       </div>
19     </nav>
20
21     <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
22     <script type="text/javascript" src="js/members.js"></script>
23
24   </body>
25 </html>
26
```

CSS and JavaScript files

Inside your public folder make 2 new folders called `js` and `stylesheets`. Inside you stylesheet folder create a new file called `style.css` and write the following;

```
form.signup, form.login { margin-top: 50px }
```

This will place a vertical gap between your form and the nav bar on both the login and signup pages.

Inside your `js` folder create three js files which correspond with your html files (login.js, members.js, signup.js).

login.js

Start your document by calling your jQuery function: `\$(document).ready(function(){ });` and start mapping out your functions:

1. Create 3 variables to grab the values from the email input, password input and the form itself. Let's call this emailInput, passwordInput and loginForm.
2. Create a function that listens for the submission of the `loginForm` and prevent default to control how the data is submitted.
3. Make an object inside the `loginForm` function that includes

```
homeworkid > 03-routes > public > js > logins > ...
1 $(document).ready(function() {
2
3   // Getting references to our form and inputs
4   var loginForm = $("form.login");
5   var emailInput = $("input#email-input");
6   var passwordInput = $("input#password-input");
7
8   // When the form is submitted, we validate there's an email and password entered
9   loginForm.on("submit", function(event) {
10     event.preventDefault();
11     var userData = {
12       email: emailInput.val().trim(),
13       password: passwordInput.val().trim()
14     };
15
16     if (!userData.email || !userData.password) {
17       return;
18     }
19
20     // If we have an email and password we run the loginUser function and clear the form
21     loginUser(email, password);
22     emailInput.val("");
23     passwordInput.val("");
24   });
25
26   // loginUser does a post to our "api/login" route and if successful, redirects us the the members page
27   function loginUser(email, password) {
28     $.post("/api/login", {
29       email: email,
30       password: password
31     }, function() {
32       window.location.replace("/members");
33       // If there's an error, log the error
34     }, "json")
35     .catch(function(err) {
36       console.log(err);
37     });
38   });
39 }
40
```

the value trimmed of both email and password input. Call it `userData`.

4. Make an if statement that says `if email or password has no value then return`.
5. Outside the `loginForm` function make a new function that takes 2 arguments `email and password`. This function post the email and password values to `/api/login`. In your `then` statement `window.location.replace` with "/members" page. Other words, if successful replace window with the members page. Remember to include an error catch that will console.log if there is are any errors. Call this function `loginUser`.
6. Finally call the `loginUser` function inside of `loginForm` function and make the 2 arguments `userData.email` and `userData.password`. Add `emailInput.val("")` and `passwordInput.val("")` at the bottom of the `loginForm` function.

note: To simplify you could place 1 argument within `loginUser` and use the `userData` object. In addition, `emailInput.val("")` and `passwordInput.val("")` serves no function as there is already a catch for blank inputs through the if statement.

If everything is correct, then you should finish with something reflecting the picture above.

Signup.js

Start by copying everything from `login.js` and pasting it inside of `signup.js`. Now you will need to make some quick modifications and add the validator for the password input.

1. Change the `loginform` to `signUpForm`. Ensure that you target the `signUp` form in your variable reference.
2. Change the `loginUser` function to `signUpUser`
3. Inside the `signUpUser` function change the `console.log` request in the error catch to call a function called `handleLoginErr`.
4. Make sure your post request goes to "/api/signup"
5. Make a new function called `handleLoginErr` which takes in 1 argument called `err`. You will then target the spam element text within the validator by first targeting the alert id. Once you written the target insert the argument `err.responseJSON`. As a result, when an error occurs within your form application an error will unhide and show the `err.responseJSON`.
6. Add a `fadeIn` animation to the alert id upon error 500.

Note: The error text given on `login.html` is [object Object]. To specify the error given by Sequelize please change `handleLoginErr` function in `signup.js`. Reference adjacent image.

```
function handleLoginErr(err) {  
  const errorMsg = err.responseJSON.errors[0].message  
  $("#alert .msg").text(errorMsg);  
  $("#alert").fadeIn(500);  
}
```

```

// If we have an email and password, run the signUpUser function
signUpUser(userData.email, userData.password);
emailInput.val("");
passwordInput.val("");
});

// Does a post to the signup route. If successful, we are redirected to the members page
// Otherwise we log any errors
function signUpUser(email, password) {
$.post("/api/signup", {
  email: email,
  password: password
})
.then(function(data) {
  window.location.replace("/members");
  // If there's an error, handle it by throwing up a bootstrap alert
})
.catch(handleLoginErr);
}

function handleLoginErr(err) {
$("#alert .msg").text(err.responseJSON);
$("#alert").fadeIn(500);
}
}
});

```

If everything goes to plan your signup.js file should look like the adjacent image.

Members.js

Call jQuery and then write a get request to "/api/user_data" which then changes the text of `member-name` class to the results of `data.email`. The end result should look like something below.

```

homework14 > 02Html Files > public > js > members.js > ...
1  $(document).ready(function() {
2    // This file just does a GET request to figure out which user is logged in
3    // and updates the HTML on the page
4    $.get("/api/user_data").then(function(data) {
5      $(".member-name").text(data.email);
6    });
7  });
8

```

Here we are saying that if passport validator works change member.name text to data.email for confirmation. We know this as the forms never target `/api/user_data`.

At this point we need to now connect all three pages through the routing.

Routing

Inside your main project folder create a new folder called `routes`. Inside routes make 2 `js` files named `api-routes.js` and `html-routes.js`. The html-routes page will be responsible for sending your HTML pages and your api-routes will deal with your form requests.

Html-routes.js

Inside your html-routes file you will need to require paths in order to join your html locations to the HTTP GET methods.

After you installed and required paths, create a `module.exports = function(app)` with 3 get responses inside. Each Get method should correspond with a html file.

The response should read like this:

```

app.get("/", function (req, res)
{ res.sendFile(path.join(__dirname,
".../public/signup.html")

```

```

homework14 > 03routes > routes > html-routes.js > ...
1  // Requiring path to so we can use relative routes to our HTML files
2  var path = require("path");
3
4
5  module.exports = function(app) {
6
7    app.get("/", function(req, res) {
8      res.sendFile(path.join(__dirname, "../public/signup.html"));
9    });
10
11    app.get("/login", function(req, res) {
12      res.sendFile(path.join(__dirname, "../public/login.html"));
13    });
14
15    app.get("/members", function(req, res) {
16      res.sendFile(path.join(__dirname, "../public/members.html"));
17    });
18
19  };

```

Note: `__dirname` means you are requesting from the html-routes location. Whilst the html files are inside the public directory express.static does not effect sendFile functions. For this reason, paths will be required. In addition, you can replace `app` with `router` via `express.Router()`.

End result should look like the image above.

Api-routes.js

Start by creating a `module.exports` function that takes in 1 argument called `app`. Inside your argument you want to create 2 post requests and 2 get requests. Each request will be as follows

- Post request for `/api/login`, which corresponds with `login.js`
- Post request for `/api/signup`, which corresponds with `signup.js`
- Get request for `/logout`, which corresponds with `members.js`
- Get request for `/api/user_data`, which corresponds with `members.js`

Unfortunately, until we setup passport and Sequelize the only route that can be complete is logout and user_data.

In the logout route simple write `req.logout()`, `res.redirect("/")` which means upon logout redirect to the sign up page. Your api-routes for now will look like the image below.

For user_data, state if not request.user then response.json an empty object, else response.json user email and id.

Your end result should look like the corresponding image.

```
module.exports = function(app) {  
  
  app.post("/api/login", function(req, res) {  
  });  
  
  app.post("/api/signup", function(req, res) {  
  });  
  
  // Route for logging user out  
  app.get("/logout", function(req, res) {  
    req.logout();  
    res.redirect("/");  
  });  
  
  // Route for getting some data about our user to be used client side  
  app.get("/api/user_data", function(req, res) {  
    if (!req.user) {  
      // The user is not logged in, send back an empty object  
      res.json({});  
    } else {  
      // Otherwise send back the user's email and id  
      // Sending back a password, even a hashed password, isn't a good idea  
      res.json({  
        email: req.user.email,  
        id: req.user.id  
      });  
    }  
  });  
};
```

Server.js

Finally go to your server.js file and require both the api-routes and html-routes. Remember to pass through the argument of app from express. If everything is correct then you should still be able to load up your pages in your localhost and the redirect links should work.

```
// Requiring our routes  
require("./routes/html-routes.js")(app);  
require("./routes/api-routes.js")(app);
```

Conclusion: If everything goes right the `a` tag link on login.html and signup.html should work and typing localhost:8080/ login or members or signup in the address bar should go to the corresponding website.

Step Three: Passport

Aim: Use passport.js to create a cookie token

Return to Server.js and now you will need to require express-sessions and passport and include all the passport's middleware. Documentation on requirements can be found on passportjs's website under passport-local. It uses connect middleware which can be substituted with express⁵.

Information about connect can be found here: <https://github.com/senchalabs/connect#readme>

In order to substitute in express, the following will happen:

- Connect is replaced by express
- cookieSession is replaced by express-session
- bodyparser is replaced by express's urlencoded and json()
- app.use will be inside our api-routes and html-routes

For reference you can look at middleware on the Passport.js entry on npm <https://www.npmjs.com/package/passport>

After configuration your server.js file should look like the image below.

```
1 // Requiring necessary npm packages
2 const express = require("express");
3 const session = require("express-session");
4
5 // Requiring passport as we've configured it
6 const passport = require("../config/passport");
7
8 // Setting up port and requiring models for syncing
9 const PORT = process.env.PORT || 8080;
10
11 // Creating express app and configuring middleware needed for authentication
12 const app = express();
13 app.use(express.urlencoded({ extended: true }));
14 app.use(express.json());
15 app.use(express.static("public"));
16
17 // We need to use sessions to keep track of our user's login status
18 app.use(session({ secret: "keyboard cat", resave: true, saveUninitialized: true }));
19 app.use(passport.initialize());
20 app.use(passport.session());
21
22 // Requiring our routes
23 require("../routes/html-routes.js")(app);
24 require("../routes/api-routes.js")(app);
25
26 app.listen(PORT, function() {
27   console.log("=> 🚀 Listening on port %s. Visit http://localhost:%s/ in your browser.", PORT, PORT);
28 });
```

Next we need to create a new folder called config which houses another folder called middleware and file called passport.js. Inside of middleware make another file called isAuthenticated.js.

Passport.js

Inside of passport.js you want to copy and paste the strategy from passport-local and substitute username with email. In addition, you will need to require passport and localStrategy. An example of this can be found at <http://www.passportjs.org/docs/configure/>.

To get your session to work you will need to serialize and deserialize your user. An example of this can be found under the sub-heading of sessions in the passport documentation. For now both these functions will be callbacks and will not require the `User.findById` addition.

⁵ Jared Hanson, "Passport-Local", Passport.JS, <http://www.passportjs.org/packages/passport-local/>

Finally `module.exports` the passport functions as passport. Your page should now look like the image below. Please be aware we will be replacing much of the code on this page later to interact with sequelize.

```
homework14 > 03passport > config > passport.js > ...
1 const passport = require("passport");
2 const LocalStrategy = require("passport-local").Strategy;
3
4 // Telling passport we want to use a Local Strategy. In other words, we want login with a username/email and password
5 passport.use(new LocalStrategy(
6   // Our user will sign in using an email, rather than a "username"
7   function(email, password, done) {
8     User.findOne( email: email ), function (err, user) {
9       if (err) { return done(err); }
10      if (!user) { return done(null, false); }
11      if (!user.verifyPassword(password)) { return done(null, false); }
12      return done(null, user);
13    });
14  }
15 );
16
17 // In order to help keep authentication state across HTTP requests,
18 // Sequelize needs to serialize and deserialize the user
19 // Just consider this part boilerplate needed to make it all work
20 passport.serializeUser(function(user, cb) {
21   cb(null, user);
22 });
23
24 passport.deserializeUser(function(obj, cb) {
25   cb(null, obj);
26 });
27
28 // Exporting our configured passport
29 module.exports = passport;
30
```

Next open up your **isAuthenticated.js** file and reference Authenticate in the Passport.js documentation. As we will be passing this through html routes we can cut out some of the code. Start by making this file into a module via `module.exports = function (req, res, next)`. Next is used here to say that if the request passes then move to the next function within the request. This will make sense when used within html-routes.

Inside the function merely state if `request.user` return `next()` else return response redirect to `"/"`. What this does is that if user is passed through `isAuthenticated` then move to the next function in the request. If not, then redirect to the signup page. Your page should look like the image below.

```
Resources > 02-Homework > Develop > config > middleware > isAuthenticated.js > ...
1 // This is middleware for restricting routes a user is not allowed to visit if not logged in
2 module.exports = function(req, res, next) {
3   // If the user is logged in, continue with the request to the restricted route
4   if (req.user) {
5     return next();
6   }
7
8   // If the user isn't logged in, redirect them to the login page
9   return res.redirect("/");
10 };
11
```

Return to html-routes.js

Now that passport has been defined, we can start inserting our `req.user` and applying the `isAuthenticated` function to our members get method.

First require the `isAuthenticated` file and place your variable inside of your members get method between `"/members"`, and `function(req,res)`.

Next, inside of `"/"` and `"/login"` add a new condition saying if `request.user` then response.redirect to `"/members"` page. In other words, if the user is saved already in the system then redirect to the members page upon entering either the sign up or log in pages.

```
homework14 > 03passport > routes > html-routes.js > ...
1 // Requiring path to so we can use relative routes to our HTML files
2 const path = require("path");
3
4 // Requiring our custom middleware for checking if a user is logged in
5 const isAuthenticated = require("../config/middleware/isAuthenticated");
6
7 module.exports = function(app) {
8
9   app.get("/", function(req, res) {
10     // If the user already has an account send them to the members page
11     if (req.user) {
12       res.redirect("/members");
13     }
14     res.sendFile(path.join(__dirname, "../public/signup.html"));
15   });
16
17   app.get("/login", function(req, res) {
18     // If the user already has an account send them to the members page
19     if (req.user) {
20       res.redirect("/members");
21     }
22     res.sendFile(path.join(__dirname, "../public/login.html"));
23   });
24
25   // Here we've add our isAuthenticated middleware to this route.
26   // If a user who is not logged in tries to access this route they will be redirected to the signup page
27   app.get("/members", isAuthenticated, function(req, res) {
28     res.sendFile(path.join(__dirname, "../public/members.html"));
29   });
30
31 };
32
```

This will not work yet as Sequelize has not been implemented however, to save time we will place the code here to save time. Your file should now look like the above image.

Return to api-routes.js

Now to add more functionality to our api-routes page. The pages will still not work as we need Sequelize to finally allow a transition between login to members through signup.

Start of by requiring passport and inside of your app.post api/login function add passport.authenticate("local") between the address and function. Passport.authenticate acts like a middleware and the final part of the passport authentication. References below:

Passport-local: <http://www.passportjs.org/packages/passport-local/>

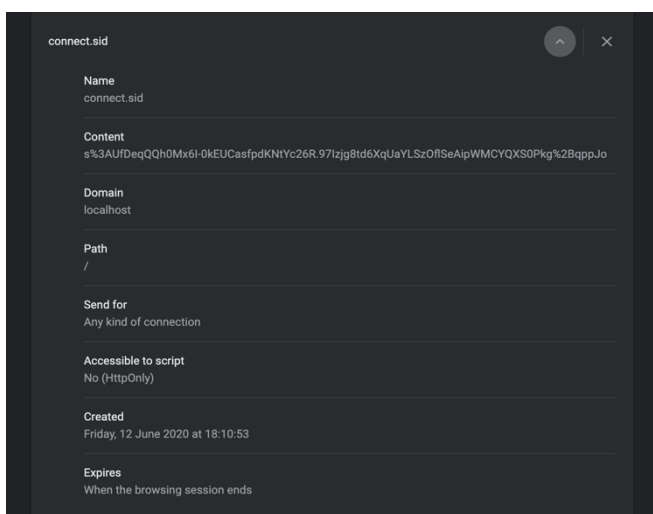
Passport-Authenticate: <http://www.passportjs.org/docs/authenticate/>

```
1 // Require passport
2 var passport = require("../config/passport");
3
4 module.exports = function(app) {
5
6   app.post("/api/login", passport.authenticate("local"), function(req, res) {
7     res.json(req.user);
8   });
9
10
11   app.post("/api/signup", function(req, res) {
12
13   });
14
15   app.get("/logout", function(req, res) {
16     req.logout();
17     res.redirect("/");
18   });
19
20   app.get("/api/user_data", function(req, res) {
21     if (!req.user) {
22       res.json({});
23     } else {
24       res.json({
25         email: req.user.email,
26         id: req.user.id
27       });
28     }
29   });
30
31 };
32
33
```

Warning! If you try use isAuthenticated here instead of passport.authenticate("local") then you will not be able to access the members.html page.

Api-routes should now look like the adjacent image

Conclusion: To check if passport is working you should see a cookie created when you enter details into the login page. Please reference the image below from Chrome as an example;



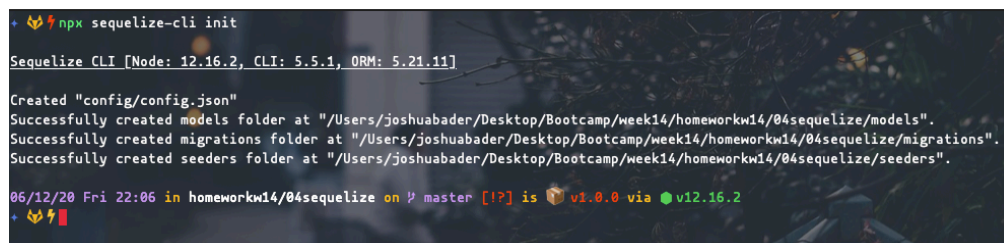
Step Four: Sequelize

Aim: Setup the database and get our page to finally log over to member.html

For this project we will be using Sequelize for the database end of passport.js. This will finally allow the login page to move through to members and the signup page to save the login details. Sequelize will require some models, a config.json file and some final stitching to bring everything together.

Setup

Lets begin with some magic. Open terminal in your root menu and type in `npx sequelize-cli init`. Sequelize will then automatically create your models folder, index.js, config.json, a migration folder and a seeders folder. Your terminal should look like the image below.



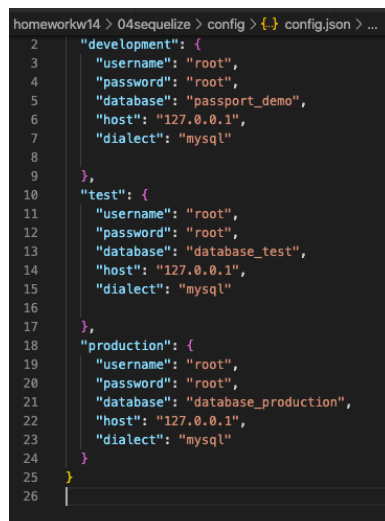
```
+ 🐉 npx sequelize-cli init
Sequelize CLI [Node: 12.16.2, CLI: 5.5.1, ORM: 5.21.11]

Created "config/config.json"
Successfully created models folder at "/Users/joshuabader/Desktop/Bootcamp/week14/homework14/04sequelize/models".
Successfully created migrations folder at "/Users/joshuabader/Desktop/Bootcamp/week14/homework14/04sequelize/migrations".
Successfully created seeders folder at "/Users/joshuabader/Desktop/Bootcamp/week14/homework14/04sequelize/seeders".

06/12/20 Fri 22:06 in homework14/04sequelize on  master [!?] is 🍷 v1.0.0 via 🟢 v12.16.2
+ 🐉
```

Inside your VS code check the `Config.JSON` file and make sure that the username and password are the same as your mySQL login details. Next change the development database to `passport_demo` and remove operatorsAliases alongside the `,` next to "mysql". Your config file should now look like the image below.

Inside the root folder delete both migrations and seeders as this will not be needed for this project. Next you want to check your index.js file is in order.



```
homework14 > 04sequelize > config > {} config.json > ...
2   "development": {
3     "username": "root",
4     "password": "root",
5     "database": "passport_demo",
6     "host": "127.0.0.1",
7     "dialect": "mysql"
8   },
9
10  "test": {
11    "username": "root",
12    "password": "root",
13    "database": "database_test",
14    "host": "127.0.0.1",
15    "dialect": "mysql"
16  },
17
18  "production": {
19    "username": "root",
20    "password": "root",
21    "database": "database_production",
22    "host": "127.0.0.1",
23    "dialect": "mysql"
24  }
25 }
26
```

Index.js

The concept behind this file is as follows

- Line 11 is saying that if the connection is within heroku's database then set Sequelize to process.env. If not then set Sequelize connection to config.json.
- Line 18, the model directory is read and pulled into Sequelize excluding index.js. These models will then become exports Objects.
- Line 27, all model's associate functions are actioned if they exist.
- Line 34, your Sequelize database (Sequelize) and an instance of the Sequelize library are then exported and allow access in other folders⁶.

User.js

Inside your models folder create a new file called 'Users'. This is where we will define how our sign up information is stored within mySQL via Sequelize.

⁶ Vapurmaid, "StackOverFlow", <https://stackoverflow.com/questions/49480021/sequelize-model-loading-in-nodejs>

First Sequelize documents tell us to define our model we should use the `sequelize.define()` function⁷. Create a class variable named `User` which equals `sequelize.define()` with 2 arguments. This first argument is the model name, “User”, next is a object. Inside your object we are going to define email and password which are the two pieces of information we receive from the signup form.

Email is an object and inside that object will be a number of properties. The properties include:

- `type: DataType.STRING`
DataType is data provided by the form and that information is a String.
- `allowNull: false`
As this is essential with identifying a user, email cannot be null.
- `unique: true`
As users are defined by an email address, this needs to be unique to avoid multiple accounts for the same user email
- `validate: { isEmail: true }`
another validator

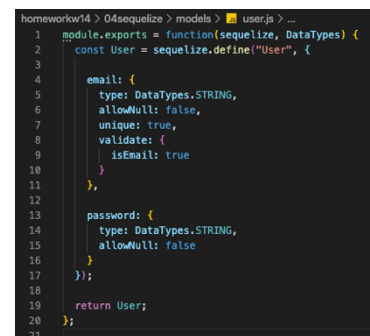
Next will be password and it has only 2 properties.

- `type: DataType.String`
DataType is data provided by the form and that information is a String.
- `allowNull: false`
For security this should be set to false.

Now below your `User` class write ``return User`` and then wrap everything inside a `module.exports` with a function that takes in 2 arguments of `sequelize` and `DataTypes`. You should have something that resembles the adjacent image.

bcrypt

Next we need to add in `bcrypt` to hash the input password by initially requiring `bcrypt`. Next under the `User` model and above ``return User`` we want to insert a function to hash the password. According to `bcrypt` documentation a password can be hashed by using;



```
1 module.exports = function(sequelize, DataTypes) {  
2   const User = sequelize.define("User", {  
3  
4     email: {  
5       type: DataTypes.STRING,  
6       allowNull: false,  
7       unique: true,  
8       validate: {  
9         isEmail: true  
10      }  
11    },  
12  
13    password: {  
14      type: DataTypes.STRING,  
15      allowNull: false  
16    }  
17  });  
18  
19  return User;  
20 };  
21
```

```
`bcrypt.hashSync(userPasswordInput, bcrypt.genSaltSync(10), callback)`
```

For this situation we will be using 10 rounds which is about ~10 hashes/sec. Your function should look like the following;

```
`user.password = bcrypt.hashSync(user.password, bcrypt.genSaltSync(10), null)`8
```

Here the `user.password` will be changed into a hashed password and there will be no callback function. This function will then be wrapped in a hook that needs to occur before being added to the database.

According to `sequelize` documentation, to declare a hook you need to use the `Model.addHook("Time")` function. In this situation the model is `User` and the time is `beforeCreate` which is referenced under 'Hooks firing order'⁹. Your function should then look something like this;

⁷ "Model Definition", Sequelize, <https://sequelize.org/master/manual/model-basics.html>

⁸ Bcrypt, "To Hash a Password", NPM, <https://www.npmjs.com/package/bcrypt>

⁹ "Available hooks", Sequelize, <https://sequelize.org/master/manual/hooks.html>

```
`User.addhook("beforeCreate", function(user) {
  user.password = bcrypt.hashSync(user.password, bcrypt.genSaltSync(10), null)
});`
```

Finally you want to compare the input password and hashed password saved inside our database ensure that the correct password has been entered. Above your initial User.addhook function() add a new function ;

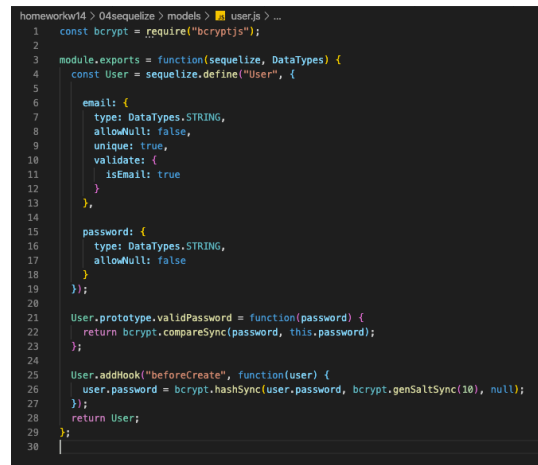
```
`bcrypt.compareSync(password, this.password)`
```

Now wrap this inside a prototype.validatePassword function and place return on the compareSync. Your function should look something like this;

```
`User.prototype.validPassword = function
(password) {
  Return bcrypt.compareSync(password,
  this.password)`
}
```

Note: compare password will be used inside passport.js to validate the password.

Your User Model is now complete and reflect the following image. Next lets fix passport.js



```
homework14 > 04sequelize > models > user.js > ...
1  const bcrypt = require("bcryptjs");
2
3  module.exports = function(sequelize, DataTypes) {
4    const User = sequelize.define("User", {
5
6      email: {
7        type: DataTypes.STRING,
8        allowNull: false,
9        unique: true,
10       validate: {
11         isEmail: true
12       }
13     },
14
15     password: {
16       type: DataTypes.STRING,
17       allowNull: false
18     }
19   });
20
21   User.prototype.validPassword = function(password) {
22     return bcrypt.compareSync(password, this.password);
23   };
24
25   User.addHook("beforeCreate", function(user) {
26     user.password = bcrypt.hashSync(user.password, bcrypt.genSaltSync(10), null);
27   });
28   return User;
29 };
30
```

Return to Passport.js

Now that we have defined our model we can now correctly use sequelize inside of passport.js. First require the model's folder and call the variable db.

Next inside the LocalStrategy we need to indicate our parameter as we are using email as our username. According to passport.Js documentation we can use the `usernameField` to indicate that we will be using email as a means of signing in¹⁰. Change your LocalStrategy to look like this;

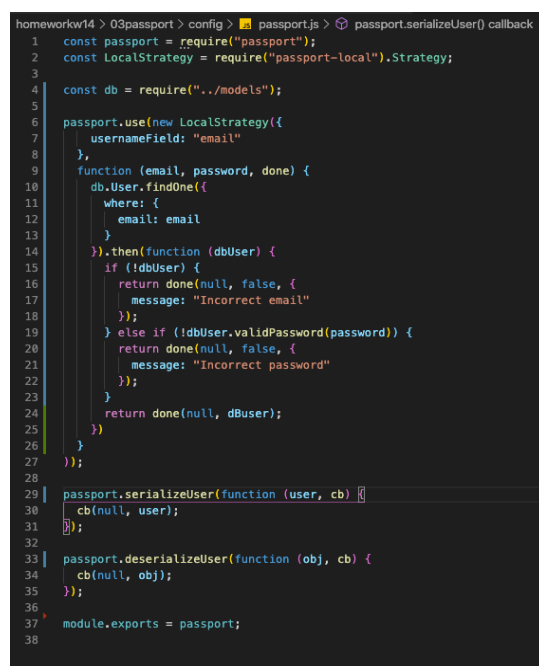
```
passport.use(new LocalStrategy({
  username: "email"
}),
Function (email, password, done){
```

Next we need to specify to Sequelize where to compare the form input data (email) with the database. Next within the User.findOne function make the following changes and include a promise;

```
db.User.findOne({
  where: { email: email}
}).then(function(dbUser){
```

In your then statement, take the previously inserted err, user function and place it inside the curly brackets. Remove the function wrap and start defining your if statements; not dbUser, incorrect email or return done.

Password.js is now complete.



```
homework14 > 03passport > config > passport.js > passport.serializeUser() callback
1  const passport = require("passport");
2  const LocalStrategy = require("passport-local").Strategy;
3
4  const db = require("../models");
5
6  passport.use(new LocalStrategy({
7    usernameField: "email"
8  },
9  function (email, password, done) {
10     db.User.findOne({
11       where: {
12         email: email
13       }
14     }).then(function (dbUser) {
15       if (!dbUser) {
16         return done(null, false, {
17           message: "Incorrect email"
18         });
19       } else if (!dbUser.validPassword(password)) {
20         return done(null, false, {
21           message: "Incorrect password"
22         });
23       }
24       return done(null, dbUser);
25     });
26   });
27
28   passport.serializeUser(function (user, cb) {
29     cb(null, user);
30   });
31
32   passport.deserializeUser(function (obj, cb) {
33     cb(null, obj);
34   });
35
36   module.exports = passport;
37
38
```

¹⁰ "Parameters", passport.js, <http://www.passportjs.org/docs/username-password/>

Return to api-route.js


First require your models into api-route.js and call the variable db. Next inside your `api/signup` post method you want to create a Sequelize call that captures your email and password inputs from the signup form. As signup.js has already declared your email and password as a part of the request.body write our sequelize function as follows;

```
`db.User.create({ email: req.body.email, password: req.body.password})`
```

As this is sequelize you will need to attach a promise via then and inside your then function you want to redirect to "api/login". Remember to add your HTTP response status codes here. 307 is the response status code for 'Temporary Redirect'. Follow that up by also including a catch statement which responds with a 401 error (Unauthorized) as a JSON¹¹.

```
` .then(function() {  
  Res.redirect(307, "/api/login")  
})  
 .catch(function(err){  
  Res.status(401).json(err)  
})`
```

Your api-routes.js should now look like the following image file.



```
homework14 > 03passport > routes > api-routes.js > ...  
1 // Require passport  
2 const passport = require("../config/passport");  
3  
4 module.exports = function(app) {  
5  
6   app.post("/api/login", passport.authenticate("local"), function(req, res) {  
7     res.json(req.user);  
8   });  
9  
10  app.post("/api/signup", function(req, res) {  
11    db.User.create({  
12      email: req.body.email,  
13      password: req.body.password  
14    })  
15    .then(function() {  
16      res.redirect(307, "/api/login")  
17    })  
18    .catch(function(err){  
19      res.status(401).json(err)  
20    })  
21  });  
22  
23  
24  app.get("/logout", function(req, res) {  
25    req.logout();  
26    res.redirect("/");  
27  });  
28  
29  app.get("/api/user_data", function(req, res) {  
30    if (!req.user) {  
31      res.json({});  
32    } else {  
33      res.json({  
34        email: req.user.email,  
35        id: req.user.id  
36      });  
37    }  
38  });  
39  
40 }
```

Return to server.js

Last thing we need to do now is sync up our database with server.js via sequelize. According to the documentation this is achieved via db.sequelize.sync() function¹². Now wrap the app.listen function within a db.sequelize.sync().then(function(){ and your finished. Your code should look like this;

```
`db.sequelize.sync().then(function(){ app.listen(PORT, function() { console.log ("You are listening to  
port %s. " PORT) }) });`
```

Finally at the top of your page require the models folder and call the variable db.

Open up terminal, type "node server.js".

Open your browser and test if you can now go to the member.html page.

Note: Make sure that mySql.server start is running

Conclusion: You should now be able to get to the member.html page and log out. To check that you have logged out correctly, once on the signup page type into the address bar "localhost:8080/members". You should be redirected back to the signup page

¹¹ "HTTP response status codes", Mozilla, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>

¹² "Implementing a todo app", Sequelize.Readthedocs.io, <https://sequelize.readthedocs.io/en/rtd/articles/express/>

Step Five: Additions

Up until this point we have been aiming to get our passport file up and running. Now that it is working we need to evaluate how to improve these files for the future. This document already covers a number of issues in the '**note / warning**' tags but I will re-illiterate those points here.

1. The warning on sign up for using an already added email needs to be fixed. The fix already exists inside the notes
2. The login page should also have a similar warning. Currently you are unsure if the email or password entered is incorrect and instead your returned to the sign up page.
3. Login.js and signup.js can be greatly reduces in lines and function changed to an AJAX call
4. Is Authenticated could just exist inside the htmlroutes.js file.
5. All vars can be replaced with consts
6. You can now implement Google or Facebook into your passport.js

References

- "Authenticate", Passport.JS, <http://www.passportjs.org/docs/authenticate/>
- "Available hooks", Sequelize, <https://sequelize.org/master/manual/hooks.html>
- "Bcrypt", NPM, <https://www.npmjs.com/package/bcrypt>
- "Bcrypt : To Hash a Password", NPM, <https://www.npmjs.com/package/bcrypt>
- "Configure", Passport.Js, <http://www.passportjs.org/docs/configure/>
- "Connect", Senchalabs, <https://github.com/senchalabs/connect#readme>
- "Express()", Express, <https://expressjs.com/en/api.html#express>
- "HTTP response status codes", Mozilla, <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>
- "Implementing a todo app", Sequelize.Readthedocs.io, <https://sequelize.readthedocs.io/en/rtd/articles/express/>
- Jared Hanson, "Passport-Local", Passport.JS, <http://www.passportjs.org/packages/passport-local/>
- "Model Definition", Sequelize, <https://sequelize.org/master/manual/model-basics.html>
- "Parameters", passport.js, <http://www.passportjs.org/docs/username-password/>
- "Passport", NPM, <https://www.npmjs.com/package/passport>
- "Passport-Local", Passport.JS, <http://www.passportjs.org/packages/passport-local/>
- "Sequelize", NPM, <https://www.npmjs.com/package/sequelize>
- Vapurmaid, "StackOverFlow", <https://stackoverflow.com/questions/49480021/sequelize-model-loading-in-nodejs>