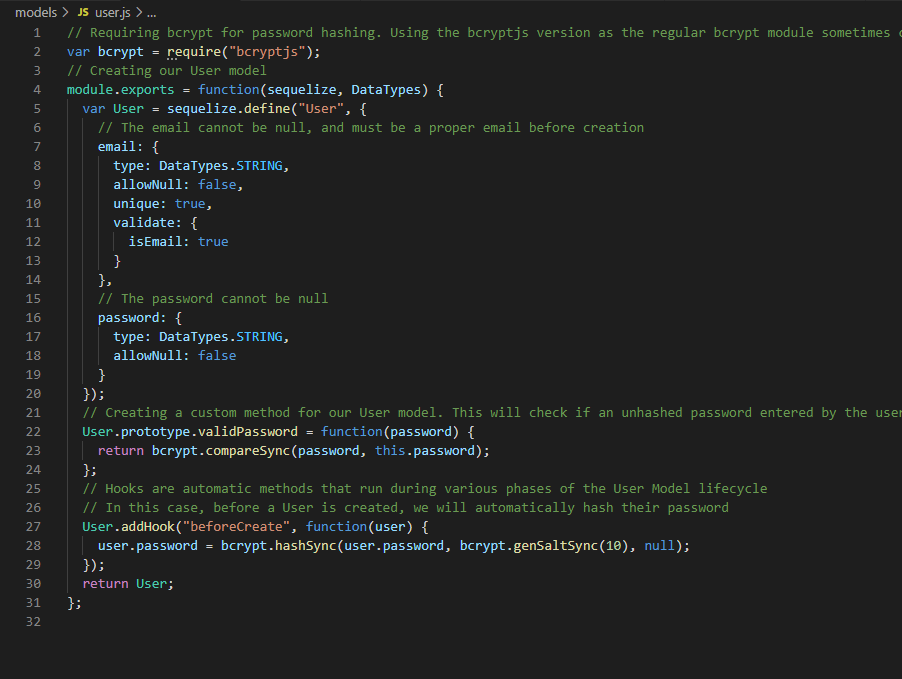
Bootcamp Homework Week 15 – Reverse Engineering Code

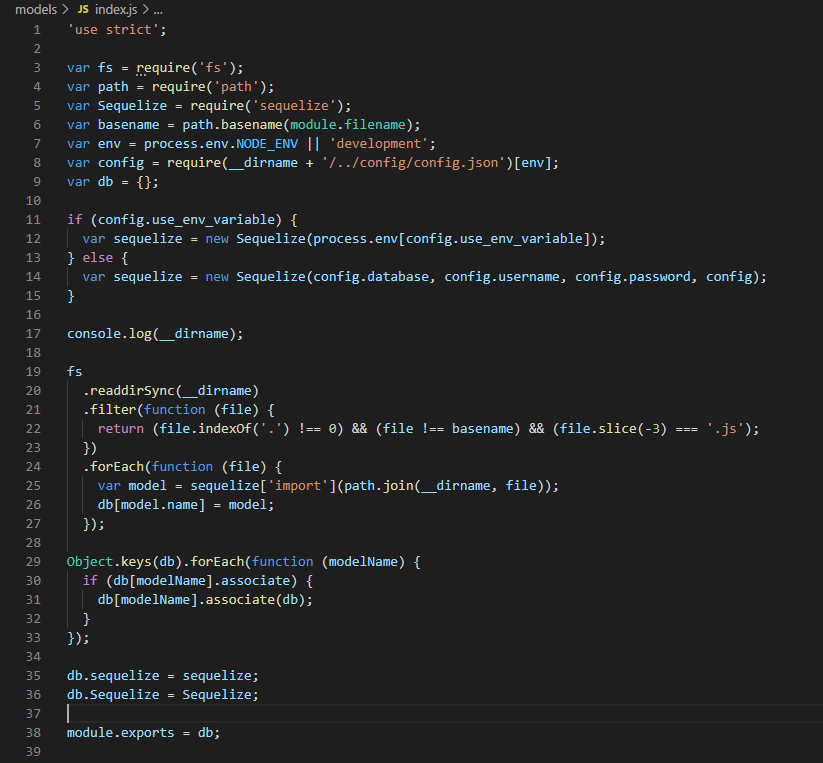
This application uses an MVC structure, with a server.js file to run the server.

**Model**

There are two models – index.js and user.js:



The user.js model creates the two database elements required for a user – the email and password. It uses sequelize to define the “user” model and defines the type of data that is being input, as well ensuring that the values are not null, are unique and are validated. The file also requires/imports a node module called “bcryptjs” which is used to both hash the user’s password and compare the inputted password to the stored hashed passwords in the database.

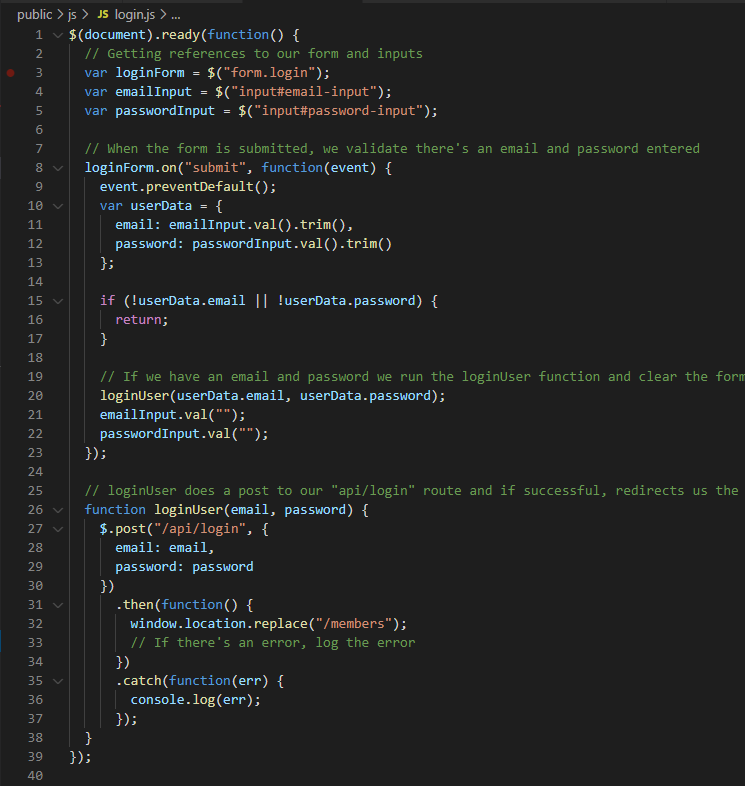


index.js – this file performs a coordination role by establishing the configuration using either the native configuration (process.env.NODE\_ENV) or the development configuration (sequelizing the elements of the config.json file in order to achieve that). The file starts by calling ‘use strict’, which is a pre ES6 method used to ensure that the programming conforms to a certain standard/code. It is generally not necessary for post 2015, ES6 scripts. After setting up the configuration environment, the NODE FS package is used read the files in the directory (in which the function/file is contained) and filters through until it finds and returns files which have a suffix of “.js”, is not equal to 0, and the file is not equal to basename (which means not the index.js file) – those filenames are then extracted into an array and sequalized for the database. The process, in short, pulls in the models that are defined in the models directory. Each model is associated (according to the associations set up in the model) and then keyed into the exports object. The main library and the instance are then exported as “db” for use in the server.js file.

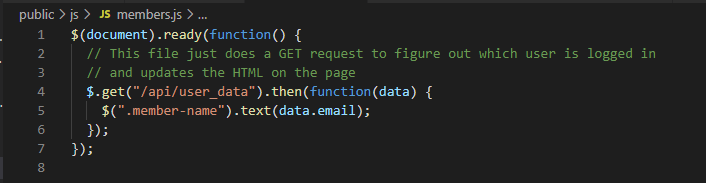
**Views (Front-End)**

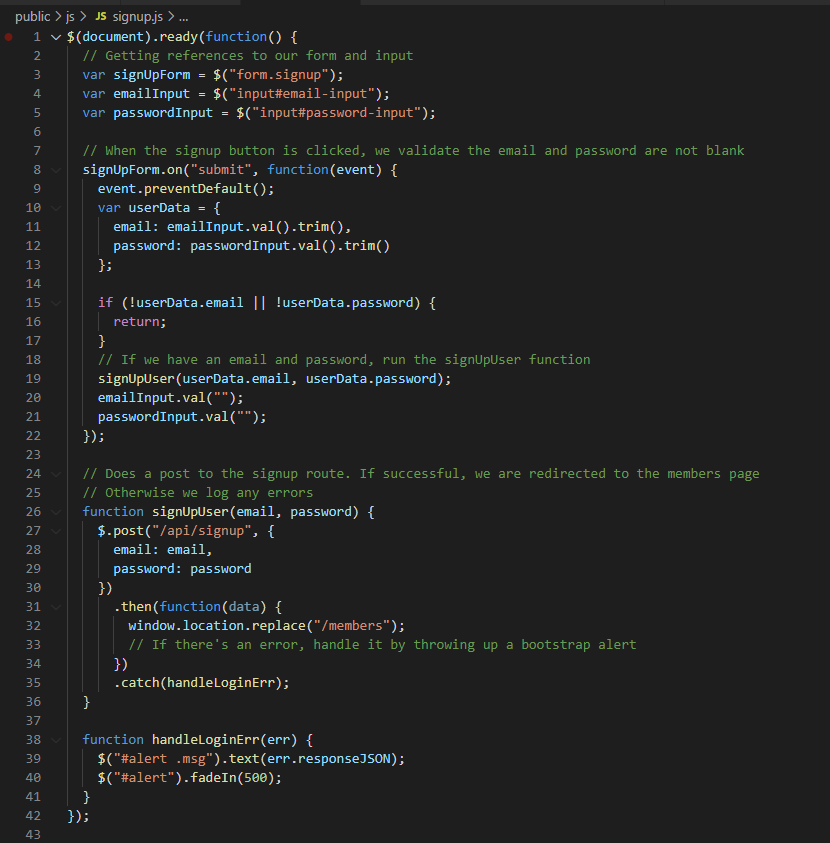
The “views” (front-end) are held in the Public folder, with the three js files (login, member and signup), which provide functionality for the three html files (login, member and signup) and are styled by the style.css file.

Login.js – this file links the login html file form and button, and contains the function that fires when the submit button is clicked (checking that an email and password have been entered, storing the values in an object (userData) and running the loginUser function. That function posts to the api/login route and if successful (ie they are valid credentials) the user is redirected to /members. An error is logged if there is an error:



Member.js - this is a fairly straightforward file that prints the users email to the html page after running a get request:



Signup.js – this file takes the user’s input from the signup form (email + password) and when the signup button is clicked, it checks the validity of the input and then the function posts that data to the signup route. If there is an error, that is handled by another function (handleLoginErr) which provides the nature of the error (the text thrown by that error) and a code 500 alert:

Login.html – uses bootstrap/bootswatch for styling, along with the style.css file held within the stylesheets folder. This file contains a login form with input for email address and password and a login button, as well as a redirection to the signup page if the user is not already a member.

Members.html - uses bootstrap/bootswatch for styling, along with the style.css file held within the stylesheets folder. This file contains a welcome header, and navbar link to the logout pag.

Signup.html - uses bootstrap/bootswatch for styling, along with the style.css file held within the stylesheets folder. This file contains a form that allows the user to input their email address and password, along with a sign-up button that the user clicks when they have completed their input. It also contains a link to the login page in case the user is already a member.

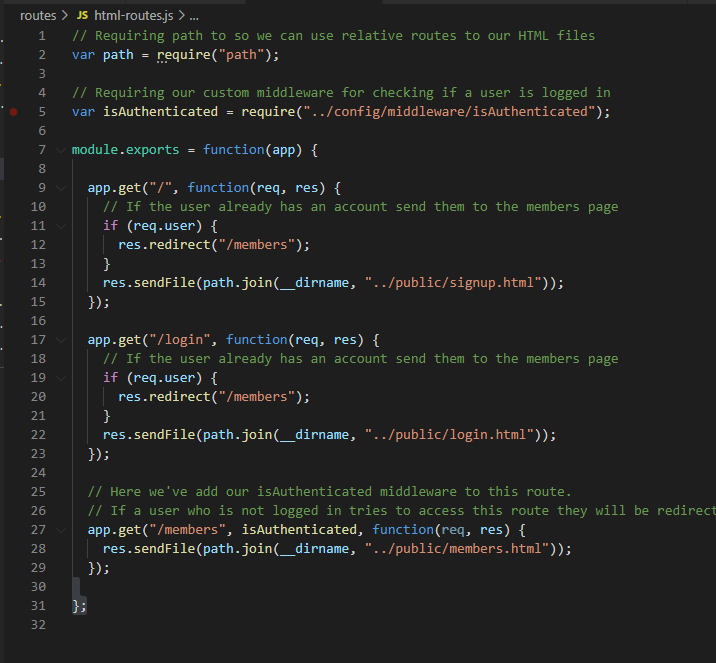
**Controller (Routes)**

There are two files for the routes – one for the API (api-routes.js) and one for the HTML (html-routes.js)

The api-routes file requires the models and the passport configuration file and provides the following routes:

* A “post” route for authenticating the user (using the passport.authenticate middleware) – at api/login
* A “post” route for signing up the user (the “User” model is used to take in the user’s email and password) – at api/signup. The user is redirected back to the login page if successful or shows a 401 error message if not successful.
* A “get” route (/logout) for logging out the user
* A “get” route for showing the user their email and id if they are successfully logged in (or returning an empty object if they are not logged in).
* 

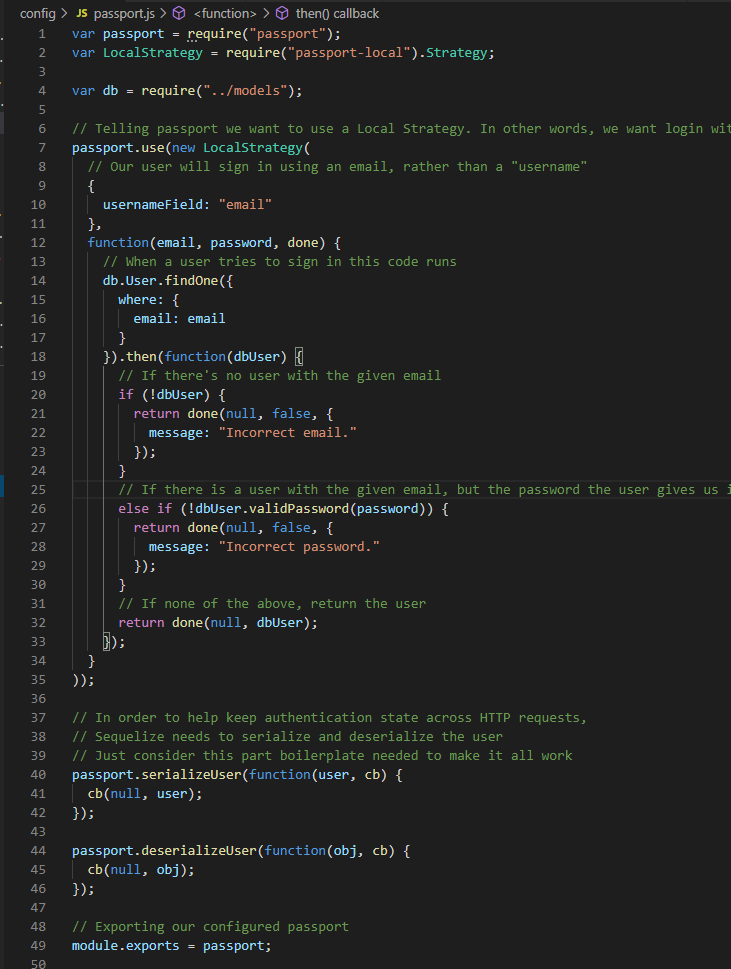
The html-routes file requires the “path” module, so the app can use relative routes for the html files. Is also requires the middleware file “isAuthenticated.js” (see below) which is used at the /members html page to check if users are authenticated. The three html routes use .get functions to provide routes for the login, signup and members pages. Further express functions are used within the get calls (redirect, for example, is used to send users who are already signed up to the /members page; res.sendfile transfers the file at the given path):



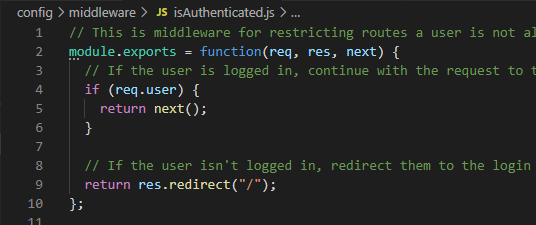
**Configuration**

The app has a config.json file which provides all of the details required to connect into the MySQL database (including setup for development, test and production environments).

Passport.js is responsible for creating a local strategy for the passport node module (which is authentication middleware for node.js – it allows a user to authenticate using a username, password, facebook, twitter and other apps for identification). The file requires the passport package and then creates a local instance of the package where the users email address is used instead of a username. When a user tries to sign in, the email address that is entered is checked against the db to see if it matches. If the email address doesn’t match, it provides the prompt: “incorrect email” – similarly, if the password entered doesn’t match, it provides the prompt: “Incorrect Password”. The also serialized and deserializes the user (necessary to keep the authentication state across HTTP request; code required to allow the package to function effectively):



Additionally, there is an “isAuthenticated.js” file used to restrict the routes the user is not allowed to visit if they are not logged in (essentially redirecting them back to the login page):



**Node Modules**

The app also contains the node modules which are downloaded as part of the NPM install process.