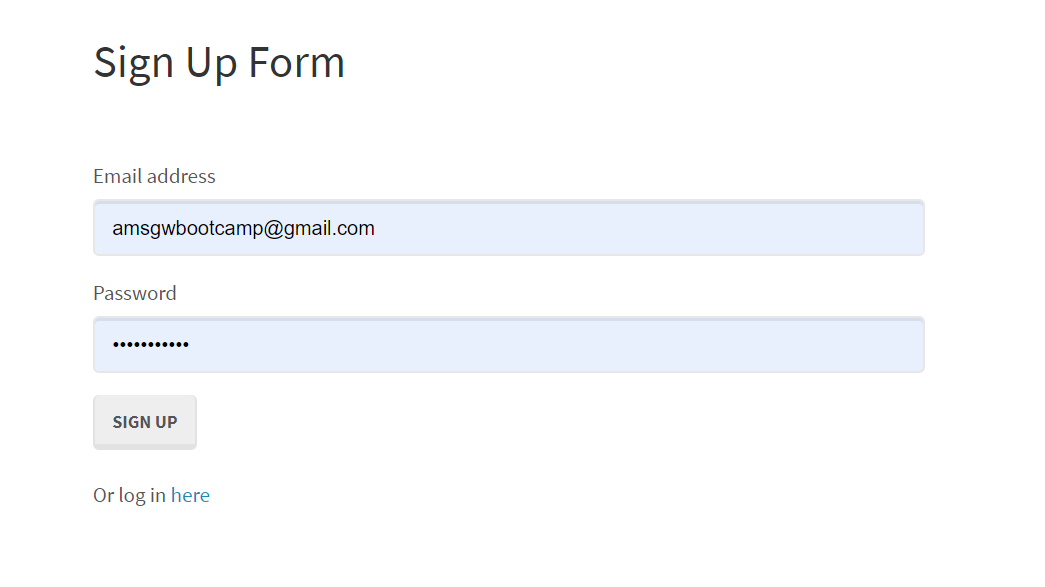
Homework 14:

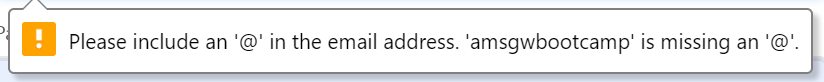
This project requires the following npm packages:

* express
* express-session
* passport
* passport-local
* path
* sequelize
* fs

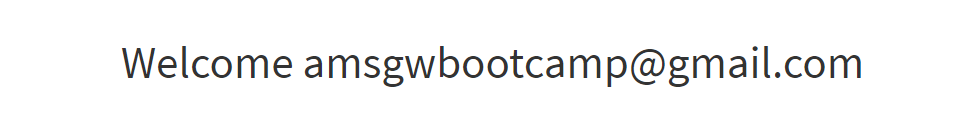
When the user enters the following in the browser: localhost:8080, they will be presented with the following screen:



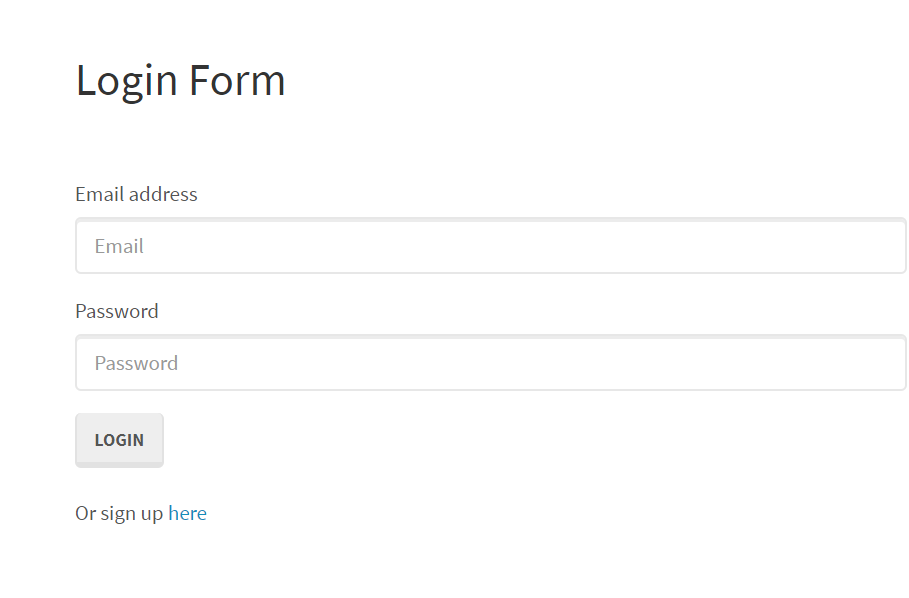
If the user does NOT enter a properly formatted email address, they will see the following error message appear:



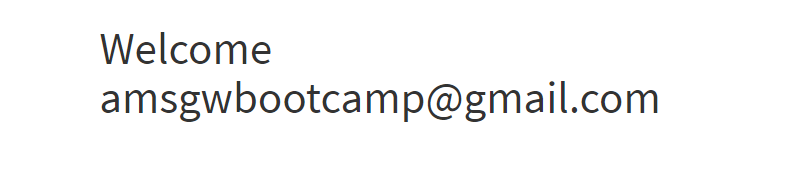
If the user does not enter an email address, no processing will continue until they enter all required information and press the SIGN UP button. If signup is successful, they will see the following screen:



Once the user has successfully signed up, they will be able to return by typing localhost:8080. At that point they can click on the log in here where they will be taken to the following screen:



Here they will be able to enter their email and password. They will then be taken to the following screen upon successful login:



The user can logout by clicking the Logout in the left-hand corner.

The directory structure of this project is as follows:

* Develop
  + **config**
    - **middleware**
      * [isAuthenticated.js](#isAuthenticated)
    - [config.json](#configJSON)

* + - [passport.js](#passport)
  + **models**
    - [index.js](#index)
    - [user.js](#user)
  + **public**
    - **js**
      * [login.js](#login)
      * [members.js](#members)
      * [signup.js](#signup)
    - **stylesheets**
      * [style.css](#style)
    - [login.html](#loginHTML)
    - [members.html](#membersHTML)
    - [signup.html](#signupHTML)
  + **routes**
    - [api-routes.js](#apiroutes)
    - [html-routes.js](#htmlRoutes)
  + [package.json](#packageJSON)
  + [server.js](#server)

***isAuthenticated.js*** (located in the config/middleware directory): this is a user defined function that will be used in the following javascript files:

* A check is made to see if the user’s email and password exists in the database.
* If they do and their password matches, they will be sent to the next page which is: members.html

***config.json*** (located in the config directory): this contains the connection information to connect to our database.

* This information can be used to create the remote connection setup in MySql Workbench.
* All databases contain the following information: the username, password, database name, IP address for the host and the version of sql we are using which is **MySql**.

***passport.js*** (located in the config directory): this contains the information for verifying if the user exists in the database or not. We have setup our passport use method to use the email address field as the username field.

files required:

* passport (npm package),
* passport-local (npm package),
* db (in the config/models directory).

Expects the following to be passed into the function

* email address
* password

A call will be made to the database looking for the user’s email address as the criteria for selecting a row in the database.

If the user is not in the database, the 401 status error is returned to the browser.

If the user is in the database, a check will be made comparing what the user entered for their password and the password from the database in the db object returned.

If they match, the information will be sent back to the browser to be used to populate members.html with a message that states Welcome <user’s email address>

If they don’t match, the 401 status error will be sent back to the browser but no message is displayed to the user.

Passport.serializeUser:  this saves the information to an object in memory so it can be retrieved later in the deserializeUser function.

Passport.deserializeUser: this passes the email address back to the calling function so it can be displayed on the browser to the user.

***index.js*** (located in the models directory):

files required:

* fs (npm package)
* path (npm package)
* sequelize (npm package)
* config.json

the following variable sets up which database we will be referencing in the config.json file: env

the following variable sets up which file to start from when importing models: basename

a check is made to see if config.use\_env\_variable is true or not. config.use\_env\_variable is used to identify which database configuration you want to use. Either from sequelize configuration file config.json or from .env variables.

This first call to .[readdirSync](https://nodejs.org/api/fs.html" \l "fs_fs_readdirsync_path) gets all the files in the directory of the currently executing script in a synchronous fashion (in this case /models/index.js is the currently executing script, so [\_\_dirname](https://nodejs.org/docs/latest/api/globals.html#globals_dirname) points to the models directory). The files in this directory are index.js and user.js.

Then it filters out the files that start with a . or are named index.js So after this function call, the list of files should be user.js.

Iterates over each file name and imports the file into the db.

Iterates over each model in the db (User) and invokes its associate function (if it has one), to setup any associations between models, foreign keys, cascades, etc.

This prevents the program from having to load up each model individually with separate lines of code (of which there could be many). Doing it this way allows you to continue adding new models without potentially forgetting to add the code to load it up in your index.js file.

The last thing we do before we exit is to set our db with all of the database configuration and model information.

Then we do a module.exports to be able to use this in our code in other files.

***user.js*** (located in the models directory):

files required:

* bcryptjs (npm package)

We are creating the User table that will contain the following fields:

* email – STRING, NOT NULL, UNIQUE VALUES
* password – STRING, NOT NULL

We have a method validPassword() which will be called to compare the password that the user entered against what is in the database. It will use another method called compareSync() which compares what the user entered in the form against the hashed value stored in the database.

We have another method called addHook() that will hash the entered password on the signup page before storing it in the database. We are ‘salting’ to make a password hash output unique to make pre-computation based attacks unhelpful.

***login.js*** (located in the public/js directory): this is the javascript file that will be used to process the login.html file.

The ready() method is used to make a function available after the document is loaded. Whatever code you write inside the $(document).ready() method will run once the page DOM is ready to execute JavaScript code.

First we assign variables for the fields on the login.html web page.

If the user clicks on the login button on the login.html webpage:

* We prevent the form being submitted to the server until we are ready to send it by using event.preventDefault() method.
* We create a new object called userData that contains 2 properties: email and password.
* We check to see if the user entered values for the email and password fields on the web page.
* If no values are entered for one or both, the screen will do nothing.
* If there are values present, we will call loginUser() sending the email address and password that the user entered in the form on the web page.
* We then clear out both the email and password fields on the web page.

loginUser():

* We will use the ajax method of “POST” sending /api/login to send the email and password entered on the form to be processed by the api-routes.js code.
* When the promise from the post method comes back, we will direct processing to the /members page. Which will display Welcome <email address> in the browser.
* If there is an error returned, we will console.log the error.

***members.js*** (located in the public/js directory): this is the javascript file that will be used to process the members.html file.

The ready() method is used to make a function available after the document is loaded. Whatever code you write inside the $(document).ready() method will run once the page DOM is ready to execute JavaScript code.

This will get the email address to be displayed to the user as **Welcome <email address>.**

***signup.js*** (located in the public/js directory): this is the javascript file that will be used to process the signup.html file.

The ready() method is used to make a function available after the document is loaded. Whatever code you write inside the $(document).ready() method will run once the page DOM is ready to execute JavaScript code.

First we assign variables for the fields on the signup.html web page.

If the user clicks on the sign up button on the signup.html webpage:

* We prevent the form being submitted to the server until we are ready to send it by using event.preventDefault() method.
* We create a new object called userData that contains 2 properties: email and password.
* We check to see if the user entered values for the email and password fields on the web page.
* If no values are entered for one or both, the screen will do nothing.
* If there are values present, we will call signUpUser() sending the email address and password that the user entered in the form on the web page.
* We then clear out both the email and password fields on the web page.

signUpUser():

* We will use the ajax method of “POST” sending /api/signup to send the email and password entered on the form to be processed by the api-routes.js code.
* When the promise from the post method comes back, we will direct processing to the /members page. Which will display Welcome <email address> in the browser.
* If there is an error returned, we will display a red box on the webpage. That is due to the fact that we are not pulling out the error message but we are trying to display an object. Which is why we see the red box with [Object].[Object] in it.

***style.css*** (located in the public/stylesheets directory): this file contains all of the styles that will be applied to all html pages.

There are only two forms that will be set with a margin of 50 pixels. This will be used in login.html and signup.html.

***login.html*** (located in the public directory):

reference the following files:

* [style.css](#style)
* [login.js](#login)

this html file uses a Bootstrap navigation bar and grid.

Within the grid there is a form that contains two input fields: email-input and password-input.

There is one submit button.

There is one href which will take you to the signup.html file

***members.html*** (located in the public directory):

reference the following files:

* [style.css](#style)
* [members.js](#members)

this html file uses a Bootstrap navigation bar and grid.

Within the grid there is a column which will display in the browser Welcome <email address>.

***signup.html*** (located in the public directory):

reference the following files:

* [style.css](#style)
* [signup.js](#signup)

this html file uses a Bootstrap navigation bar and grid.

Within the grid there is a form that contains two input fields: email-input and password-input.

There is also an alert that can be displayed. Currently it will only display [Object].[Object] in a red box because it has not been set to display the text. It has the object in there now.

There is one submit button.

There is one href which will take you to the login.html file.

api-routes.js (located in the routes directory):

files required:

* passport (npm package)
* db (models)

The following routes have been defined:

POST - /api/login – the [login.js](#login) file calls this to send the email and the password after being validated locally rather than by the domain (which in this case is the same) then redirects them to the members.html page.

POST - /api/signup – the [signup.js](#signup) file calls this to send the email and the password to create a new user in the database then redirects them to the members.html page or will display an error on the sign up page.

GET - /logout – the [members.html](#membersHTML) file calls this to log the user out of the system and returns them to the sign up form.

GET - /api/user\_data – the [members.js](#members) file calls this to get the user’s email and id from the server.

html-routes.js (located in the routes directory):

files required:

* path (npm package)
* [isAuthenticated](#isAuthenticated).js (config/middleware)

The following routes have been defined:

GET - / - will send the user to the members.html page if they have a login or the signup.html page if they do not.

GET - /login – will send the user to the members.html page if they have a login or the empty login.html page if they do not.

GET - /members – will send the user to the sign in form.

package.json (located in the develop directory): contains all of the information needed for this project.

{

  "name": "1-Passport-Example",

  "version": "1.0.0",

  "description": "",

  "main": "server.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "node server.js",

    "watch": "nodemon server.js"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "dependencies": {

    "bcryptjs": "2.4.3",

    "express": "^4.17.0",

    "express-session": "^1.16.1",

    "http": "0.0.1-security",

    "mysql2": "^1.6.5",

    "passport": "^0.4.0",

    "passport-local": "^1.0.0",

    "sequelize": "^5.8.6"

  }

}

server.js: (located in the Develop directory) -

files required:

* express (npm package)
* express-session (npm package)
* passport (npm package)
* db models (models directory)

var express = require("express"); -- so we can use npm express

var session = require("express-session"); -- so we can use npm express-session

var passport = require("./config/passport"); -- so we can use npm passport

var PORT = process.env.PORT || 8080; -- setting up so we can either use the PORT env variable or the 8080 if none is given.

var db = require("./models"); -- models for syncing with the database

var app = express(); -- setting up express

app.use(express.urlencoded({ extended: true })); -- this is to be able to parse the urls for our routes

app.use(express.json()); -- to send the data back in JSON format

app.use(express.static("public")); -- to be able to access the contents of the public folder

app.use(session({ secret: "keyboard cat", resave: true, saveUninitialized: true })); -- The session secret is a key used for signing and/or encrypting cookies set by the application to maintain session state. This is often what prevents users from pretending to be someone they're not -- ensuring that random person on the internet cannot access your application as an administrator. resave: 'Forces the session to be saved back to the session store, even if the session was never modified during the request. ' saveUninitialized: 'Forces a session that is "uninitialized" to be saved to the store.

app.use(passport.initialize()); -- basically adding passport instance to incoming requests so that authentication can be performed.

app.use(passport.session()); -- alters the request object and change the 'user' value that is currently the session id (from the client cookie) into the true deserialized user object.

require("./routes/html-routes.js")(app); -- letting this program know where our html routes file is located

require("./routes/api-routes.js")(app); -- letting this program know where our api routes file is located

This will sync our database with our javascript and log a message to the user upon success:

db.sequelize.sync().then(function() {

  app.listen(PORT, function() {

    console.log("==> 🌎  Listening on port %s. Visit http://localhost:%s/ in your browser.", PORT, PORT);

  });

});