Quality Assurance Notes

Patrick Adams

November 1, 2020

Note: This is a draft copy of notes generated by free code camp.
https://www.freecodecamp.org/

Contents

1 Quality Assurance And Testing With Chai

1.1 Learn How JavaScript Assertions Work

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.2 Test if a Variable or Function is Defined

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.3 Use Assert.isOK and Assert.isNotOK

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
isOk() will test for a truthy value and isNotOk() will test for a falsy value.
To learn more about truthy and falsy values, try our Falsy Bouncer challenge.

1.4 Test for Truthiness

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
isTrue() will test for the boolean value true and isNotTrue() will pass when given anything but the boolean value of true.
“‘js
assert.isTrue(true, ’this will pass with the boolean value true’);
assert.isTrue(’true’, ’this will NOT pass with the string value ’true’);
assert.isTrue(1, ’this will NOT pass with the number value 1’);
“‘
isFalse() and isNotFalse() also exist and behave similarly to their true counterparts except they look for the boolean value of false.

1.5 Use the Double Equals to Assert Equality

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
equal() compares objects using ==.

1.6 Use the Triple Equals to Assert Strict Equality

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
strictEqual() compares objects using ===.

1.7 Assert Deep Equality with .deepEqual and .notDeepEqual

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
deepEqual() asserts that two object are deep equal.

1.8 Compare the Properties of Two Elements

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.9 Test if One Value is Below or At Least as Large as Another

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.10 Test if a Value Falls within a Specific Range

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
.approximately(actual, expected, delta, [message])
Asserts that the actual is equal expected, to within a +/- delta range.

1.11 Test if a Value is an Array

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.12 Test if an Array Contains an Item

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.13 Test if a Value is a String

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
isString or isNotString asserts that the actual value is a string.

1.14 Test if a String Contains a Substring

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
include() and notInclude() work for strings too!
include() asserts that the actual string contains the expected substring.

1.15 Use Regular Expressions to Test a String

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
match() asserts that the actual value matches the second argument regular expression.

1.16 Test if an Object has a Property

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub. property asserts that the actual object has a given property.

1.17 Test if a Value is of a Specific Data Structure Type

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub. #typeOf asserts that value's type is the given string, as determined by Object.prototype.toString.

1.18 Test if an Object is an Instance of a Constructor

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub. #instanceOf asserts that an object is an instance of a constructor.

1.19 Run Functional Tests on API Endpoints using Chai-HTTP

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.20 Run Functional Tests on API Endpoints using Chai-HTTP II

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

1.21 Run Functional Tests on an API Response using Chai-HTTP III - PUT method

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
In the next example we'll see how to send data in a request payload (body).
We are going to test a PUT request. The '/travellers' endpoint accepts a JSON object taking the structure:
“‘json
{
"surname": [last name of a traveller of the past]
}
“‘
The route responds with :
“‘json
{
"name": [first name], "surname": [last name], "dates": [birth - death years]
}
“‘
See the server code for more details.

1.22 Run Functional Tests on an API Response using Chai-HTTP IV - PUT method

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
This exercise is similar to the preceding one. Look at it for the details.

1.23 Run Functional Tests using a Headless Browser

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.
In the next challenges we are going to simulate the human interaction with a page using a device called 'Headless Browser'.
A headless browser is a web browser without a graphical user interface. This kind of tool is particularly useful for testing web pages, as it is able to render and understand HTML, CSS, and JavaScript the same way a browser would.

1.24 Run Functional Tests using a Headless Browser II

As a reminder, this project is being built upon the following starter project on Repl.it, or cloned from GitHub.

2 Advanced Node And Express

2.1 Set up a Template Engine

As a reminder, this project is built upon the following starter project on Repl.it, or clone from GitHub.
A template engine enables you to use static template files (such as those written in _Pug_) in your app. At runtime, the template engine replaces variables in a template file with actual values which can be supplied by your server. Then it transforms the template into a static HTML file that is sent to the client. This approach makes it easier to design an HTML page and allows for displaying variables on the page without needing to make an API call from the client.
Add 'pug@~3.0.0' as a dependency in your 'package.json' file.
Express needs to know which template engine you are using. We will use the 'set' method to assign 'pug' as the 'view engine' property's value: 'app.set('view engine', 'pug')'
Your page will not load until you correctly render the index file in the 'views/pug' directory.
Change the argument of the 'res.render()' declaration in the '/' route to be the file path to the 'views/pug' directory. The path can be a relative path (relative to views), or an absolute path, and does not require a file extension.
If all went as planned, your app home page will stop showing the message "'Pug template is not defined.'" and will now display a message indicating you've successfully rendered the Pug template!
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.2 Use a Template Engine's Powers

One of the greatest features of using a template engine is being able to pass variables from the server to the template file before rendering it to HTML.
In your Pug file, you're able to use a variable by referencing the variable name as #{variable_name} inline with other text on an element or by using an equal sign on the element without a space such as p=variable_name which assigns the variable's value to the p element's text.
We strongly recommend looking at the syntax and structure of Pug here on GitHub's README. Pug is all about using whitespace and tabs to show nested elements and cutting down on the amount of code needed to make a beautiful site.
Looking at our pug file 'index.pug' included in your project, we used the variables title and message.
To pass those along from our server, you will need to add an object as a second argument to your res.render with the variables and their values. For example, pass this object along setting the variables for your index view: {title: 'Hello', message: 'Please login'}
It should look like: res.render(process.cwd() + '/views/pug/index', {title: 'Hello', message: 'Please login'});
Now refresh your page and you should see those values rendered in your view in the correct spot as laid out in your index.pug file!
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.3 Set up Passport

It's time to set up Passport so we can finally start allowing a user to register or login to an account! In addition to Passport, we will use Express-session to handle sessions. Using this middleware saves the session id as a cookie in the client and allows us to access the session data using that id on the server. This way we keep personal account information out of the cookie used by the client to verify to our server they are authenticated and just keep the key to access the data stored on the server.
To set up Passport for use in your project, you will need to add it as a dependency first in your package.json. "passport": "^0.3.2"
In addition, add Express-session as a dependency now as well. Express-session has a ton of advanced features you can use but for now we're just going to use the basics! "express-session": "^1.15.0"

You will need to set up the session settings now and initialize Passport. Be sure to first create the variables 'session' and 'passport' to require 'express-session' and 'passport' respectively.
To set up your express app to use the session we'll define just a few basic options. Be sure to add 'SESSION_SECRET' to your .env file and give it a random value. This is used to compute the hash used to encrypt your cookie!

```js
app.use(session({
secret: process.env.SESSION_SECRET,
resave: true,
saveUninitialized: true,
cookie: { secure: false }
}));
```

As well you can go ahead and tell your express app to use 'passport.initialize()' and 'passport.session()'. (For example, app.use(passport.initialize());)
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.4 Serialization of a User Object

Serialization and deserialization are important concepts in regards to authentication. To serialize an object means to convert its contents into a small key that can then be deserialized into the original object. This is what allows us to know who has communicated with the server without having to send the authentication data, like the username and password, at each request for a new page.
To set this up properly, we need to have a serialize function and a deserialize function. In Passport, we create these with passport.serializeUser(OURFUNCTION) and passport.deserializeUser(OURFUNCTION)
The serializeUser is called with 2 arguments, the full user object and a callback used by passport. A unique key to identify that user should be returned in the callback, the easiest one to use being the user's _id in the object. It should be unique as it generated by MongoDB. Similarly, deserializeUser is called with that key and a callback function for passport as well, but, this time, we have to take that key and return the full user object to the callback. To make a query search for a Mongo _id, you will have to create const ObjectID = require('mongodb').ObjectID;, and then to use it you call new ObjectID(THE_ID). Be sure to add MongoDB as a dependency. You can see this in the examples below:

```js
passport.serializeUser((user, done) => {
done(null, user._id);
});
passport.deserializeUser((id, done) => {
myDataBase.findOne({ _id: new ObjectID(id) }, (err, doc) => {
done(null, null);
});
});
```

NOTE: This deserializeUser will throw an error until we set up the DB in the next step, so for now comment out the whole block and just call done(null, null) in the function deserializeUser.
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.5 Implement the Serialization of a Passport User

Right now, we're not loading an actual user object since we haven't set up our database. This can be done many different ways, but for our project we will connect to the database once when we start the server and

keep a persistent connection for the full life-cycle of the app.
To do this, add your database's connection string (for example: mongodb+srv://:@cluster0-jvwxi.mongodb.net/?retryWrites= to the environment variable MONGO_URI. This is used in the connection.js file.
You can set up a free database on MongoDB Atlas.
Now we want to connect to our database then start listening for requests. The purpose of this is to not allow requests before our database is connected or if there is a database error. To accomplish this, you will want to encompass your serialization and your app routes in the following code:

```
“‘js
myDB(async client => {
const myDataBase = await client.db('database').collection('users');
// Be sure to change the title
app.route('/').get((req, res) => {
//Change the response to render the Pug template
res.render('pug', {
title: 'Connected to Database',
message: 'Please login'
});
});
// Serialization and deserialization here...
// Be sure to add this...
}).catch(e => {
app.route('/').get((req, res) => {
res.render('pug', { title: e, message: 'Unable to login' });
});
});
// app.listen out here...
“‘
```

Be sure to uncomment the myDataBase code in deserializeUser, and edit your done(null, null) to include the doc.
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.6 Authentication Strategies

A strategy is a way of authenticating a user. You can use a strategy for allowing users to authenticate based on locally saved information (if you have them register first) or from a variety of providers such as Google or GitHub. For this project, we will set up a local strategy. To see a list of the hundreds of strategies, visit Passport's site here.
Add passport-local as a dependency and add it to your server as follows: const LocalStrategy = require('passport-local');
Now you will have to tell passport to use an instantiated LocalStrategy object with a few settings defined. Make sure this (as well as everything from this point on) is encapsulated in the database connection since it relies on it!

```
“‘js
passport.use(new LocalStrategy(
function(username, password, done) {
myDataBase.findOne({ username: username }, function (err, user) {
console.log('User '+ username +' attempted to log in.');
if (err) { return done(err); }
if (!user) { return done(null, false); }
if (password !== user.password) { return done(null, false); }
return done(null, user);
});
```

}
));
‘‘‘
This is defining the process to use when we try to authenticate someone locally. First, it tries to find a user in our database with the username entered, then it checks for the password to match, then finally, if no errors have popped up that we checked for, like an incorrect password, the user's object is returned and they are authenticated.
Many strategies are set up using different settings, but generally it is easy to set it up based on the README in that strategy's repository. A good example of this is the GitHub strategy where we don't need to worry about a username or password because the user will be sent to GitHub's auth page to authenticate. As long as they are logged in and agree then GitHub returns their profile for us to use.
In the next step, we will set up how to actually call the authentication strategy to validate a user based on form data!
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.7 How to Use Passport Strategies

In the index.pug file supplied, there is actually a login form. It has previously been hidden because of the inline JavaScript if showLogin with the form indented after it. Before showLogin as a variable was never defined, so it never rendered the code block containing the form. Go ahead and on the res.render for that page add a new variable to the object showLogin: true. When you refresh your page, you should then see the form! This form is set up to POST on /login, so this is where we should set up to accept the POST and authenticate the user.
For this challenge you should add the route /login to accept a POST request. To authenticate on this route, you need to add a middleware to do so before then sending a response. This is done by just passing another argument with the middleware before your function(req,res) with your response! The middleware to use is passport.authenticate('local').
passport.authenticate can also take some options as an argument such as: { failureRedirect: '/' } which is incredibly useful, so be sure to add that in as well. The response after using the middleware (which will only be called if the authentication middleware passes) should be to redirect the user to /profile and that route should render the view profile.pug.
If the authentication was successful, the user object will be saved in req.user.
At this point, if you enter a username and password in the form, it should redirect to the home page /, and the console of your server should display 'User {USERNAME} attempted to log in.', since we currently cannot login a user who isn't registered.
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.8 Create New Middleware

As is, any user can just go to /profile whether they have authenticated or not, by typing in the url. We want to prevent this, by checking if the user is authenticated first before rendering the profile page. This is the perfect example of when to create a middleware.
The challenge here is creating the middleware function ensureAuthenticated(req, res, next), which will check if a user is authenticated by calling passport's isAuthenticated method on the request which, in turn, checks if req.user is defined. If it is, then next() should be called, otherwise, we can just respond to the request with a redirect to our homepage to login. An implementation of this middleware is:
‘‘‘js
function ensureAuthenticated(req, res, next) {
if (req.isAuthenticated()) {
return next();

}
res.redirect('/');
};
‘‘‘
Now add ensureAuthenticated as a middleware to the request for the profile page before the argument to the get request containing the function that renders the page.
‘‘‘js
app
.route('/profile')
.get(ensureAuthenticated, (req,res) => {
res.render(process.cwd() + '/views/pug/profile');
});
‘‘‘
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.9 How to Put a Profile Together

Now that we can ensure the user accessing the '/profile' is authenticated, we can use the information contained in 'req.user' on our page!
Pass an object containing the property 'username' and value of 'req.user.username' as the second argument for the render method of the profile view. Then, go to your 'profile.pug' view, and add the following line below the existing 'h1' element, and at the same level of indentation:
‘‘‘pug
h2.center#welcome Welcome, #{username}!
‘‘‘
This creates an 'h2' element with the class ''center'' and id ''welcome'' containing the text ''Welcome, '' followed by the username.
Also, in 'profile.pug', add a link referring to the '/logout' route, which will host the logic to unauthenticate a user.
‘‘‘pug
a(href='/logout') Logout
‘‘‘
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.10 Logging a User Out

Creating the logout logic is easy. The route should just unauthenticate the user and redirect to the home page instead of rendering any view.
In passport, unauthenticating a user is as easy as just calling req.logout(); before redirecting.
‘‘‘js
app.route('/logout')
.get((req, res) => {
req.logout();
res.redirect('/');
});
‘‘‘
You may have noticed that we're not handling missing pages (404). The common way to handle this in Node is with the following middleware. Go ahead and add this in after all your other routes:
‘‘‘js
app.use((req, res, next) => {

```
res.status(404)
.type('text')
.send('Not Found');
});
```

Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.11 Registration of New Users

Now we need to allow a new user on our site to register an account. On the res.render for the home page add a new variable to the object passed along--showRegistration: true. When you refresh your page, you should then see the registration form that was already created in your index.pug file! This form is set up to POST on /register, so this is where we should set up to accept the POST and create the user object in the database.
The logic of the registration route should be as follows: Register the new user > Authenticate the new user > Redirect to /profile
The logic of step 1, registering the new user, should be as follows: Query database with a findOne command > if user is returned then it exists and redirect back to home OR if user is undefined and no error occurs then 'insertOne' into the database with the username and password, and, as long as no errors occur, call next to go to step 2, authenticating the new user, which we've already written the logic for in our POST /login route.

```js
app.route('/register')
.post((req, res, next) => {
myDataBase.findOne({ username: req.body.username }, function(err, user) {
if (err) {
next(err);
} else if (user) {
res.redirect('/');
} else {
myDataBase.insertOne({
username: req.body.username,
password: req.body.password
},
(err, doc) => {
if (err) {
res.redirect('/');
} else {
// The inserted document is held within
// the ops property of the doc
next(null, doc.ops[0]);
}
}
)
}
})
},
passport.authenticate('local', { failureRedirect: '/' }),
(req, res, next) => {
res.redirect('/profile');
}
);
```

“‘
Submit your page when you think you’ve got it right. If you’re running into errors, you can check out the project completed up to this point here.
NOTE: From this point onwards, issues can arise relating to the use of the _picture-in-picture_ browser. If you are using an online IDE which offers a preview of the app within the editor, it is recommended to open this preview in a new tab.

2.12 Hashing Your Passwords

Going back to the information security section, you may remember that storing plaintext passwords is never okay. Now it is time to implement BCrypt to solve this issue.
Add BCrypt as a dependency, and require it in your server. You will need to handle hashing in 2 key areas: where you handle registering/saving a new account, and when you check to see that a password is correct on login.
Currently on our registration route, you insert a user’s password into the database like so: password: req.body.password. An easy way to implement saving a hash instead is to add the following before your database logic const hash = bcrypt.hashSync(req.body.password, 12);, and replacing the req.body.password in the database saving with just password: hash.
Finally, on our authentication strategy, we check for the following in our code before completing the process: if (password !== user.password) { return done(null, false); }. After making the previous changes, now user.password is a hash. Before making a change to the existing code, notice how the statement is checking if the password is **not** equal then return non-authenticated. With this in mind, your code could look as follows to properly check the password entered against the hash:
“‘js
if (!bcrypt.compareSync(password, user.password)) {
return done(null, false);
}
“‘
That is all it takes to implement one of the most important security features when you have to store passwords!
Submit your page when you think you’ve got it right. If you’re running into errors, you can check out the project completed up to this point here.

2.13 Clean Up Your Project with Modules

Right now, everything you have is in your server.js file. This can lead to hard to manage code that isn’t very expandable.
Create 2 new files: routes.js and auth.js
Both should start with the following code:
“‘js
module.exports = function (app, myDataBase) {
}
“‘
Now, in the top of your server file, require these files like so: const routes = require(’./routes.js’);
Right after you establish a successful connection with the database, instantiate each of them like so: routes(app, myDataBase)
Finally, take all of the routes in your server and paste them into your new files, and remove them from your server file. Also take the ensureAuthenticated function, since it was specifically created for routing. Now, you will have to correctly add the dependencies in which are used, such as const passport = require(’passport’);, at the very top, above the export line in your routes.js file.
Keep adding them until no more errors exist, and your server file no longer has any routing (**except for the route in the catch block**)!

Now do the same thing in your auth.js file with all of the things related to authentication such as the serialization and the setting up of the local strategy and erase them from your server file. Be sure to add the dependencies in and call auth(app, myDataBase) in the server in the same spot.
Submit your page when you think you've got it right. If you're running into errors, you can check out an example of the completed project here.

2.14 Implementation of Social Authentication

The basic path this kind of authentication will follow in your app is: User clicks a button or link sending them to our route to authenticate using a specific strategy (e.g. GitHub).Your route calls passport.authenticate('github') which redirects them to GitHub.The page the user lands on, on GitHub, allows them to login if they aren't already. It then asks them to approve access to their profile from our app.The user is then returned to our app at a specific callback url with their profile if they are approved.They are now authenticated, and your app should check if it is a returning profile, or save it in your database if it is not.
Strategies with OAuth require you to have at least a Client ID and a Client Secret which is a way for the service to verify who the authentication request is coming from and if it is valid. These are obtained from the site you are trying to implement authentication with, such as GitHub, and are unique to your app--THEY ARE NOT TO BE SHARED and should never be uploaded to a public repository or written directly in your code. A common practice is to put them in your .env file and reference them like so: process.env.GITHUB_CLIENT_ID. For this challenge we're going to use the GitHub strategy.
Obtaining your Client ID and Secret from GitHub is done in your account profile settings under 'developer settings', then 'OAuth applications'. Click 'Register a new application', name your app, paste in the url to your Repl.it homepage (Not the project code's url), and lastly, for the callback url, paste in the same url as the homepage but with /auth/github/callback added on. This is where users will be redirected for us to handle after authenticating on GitHub. Save the returned information as 'GITHUB_CLIENT_ID' and 'GITHUB_CLIENT_SECRET' in your .env file.
In your routes.js file, add showSocialAuth: true to the homepage route, after showRegistration: true. Now, create 2 routes accepting GET requests: /auth/github and /auth/github/callback. The first should only call passport to authenticate 'github'. The second should call passport to authenticate 'github' with a failure redirect to /, and then if that is successful redirect to /profile (similar to our last project).
An example of how /auth/github/callback should look is similar to how we handled a normal login:
“‘js
app.route('/login')
.post(passport.authenticate('local', { failureRedirect: '/' }), (req,res) => {
res.redirect('/profile');
});
“‘
Submit your page when you think you've got it right. If you're running into errors, you can check out the project up to this point here.

2.15 Implementation of Social Authentication II

The last part of setting up your GitHub authentication is to create the strategy itself. For this, you will need to add the dependency of 'passport-github' to your project and require it in your auth.js as GithubStrategy like this: const GitHubStrategy = require('passport-github').Strategy;. Do not forget to require and configure dotenv to use your environment variables.
To set up the GitHub strategy, you have to tell Passport to use an instantiated GitHubStrategy, which accepts 2 arguments: an object (containing clientID, clientSecret, and callbackURL) and a function to be called when a user is successfully authenticated, which will determine if the user is new and what fields to save initially in the user's database object. This is common across many strategies, but some may require more information as outlined in that specific strategy's GitHub README. For example, Google requires a scope as well which determines what kind of information your request is asking to be returned and asks the

user to approve such access. The current strategy we are implementing has its usage outlined here, but we're going through it all right here on freeCodeCamp!
Here's how your new strategy should look at this point:
“‘js
passport.use(new GitHubStrategy({
clientID: process.env.GITHUB_CLIENT_ID,
clientSecret: process.env.GITHUB_CLIENT_SECRET,
callbackURL: /*INSERT CALLBACK URL ENTERED INTO GITHUB HERE*/
},
function(accessToken, refreshToken, profile, cb) {
console.log(profile);
//Database logic here with callback containing our user object
}
));
“‘
Your authentication won't be successful yet, and it will actually throw an error without the database logic and callback, but it should log your GitHub profile to your console if you try it!
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.16 Implementation of Social Authentication III

The final part of the strategy is handling the profile returned from GitHub. We need to load the user's database object if it exists, or create one if it doesn't, and populate the fields from the profile, then return the user's object. GitHub supplies us a unique id within each profile which we can use to search with to serialize the user with (already implemented). Below is an example implementation you can use in your project--it goes within the function that is the second argument for the new strategy, right below where console.log(profile); currently is:
“‘js
myDataBase.findOneAndUpdate(
{ id: profile.id },
{
$setOnInsert: {
id: profile.id,
name: profile.displayName || 'John Doe',
photo: profile.photos[0].value || '',
email: Array.isArray(profile.emails)
? profile.emails[0].value
: 'No public email',
created_on: new Date(),
provider: profile.provider || ''
},
$set: {
last_login: new Date()
},
$inc: {
login_count: 1
}
},
{ upsert: true, new: true },
(err, doc) => {
return cb(null, doc.value);
}

);
‘‘‘
‘findOneAndUpdate‘ allows you to search for an object and update it. If the object doesn’t exist, it will be inserted and made available to the callback function. In this example, we always set ‘last_login‘, increment the ‘login_count‘ by ‘1‘, and only populate the majority of the fields when a new object (new user) is inserted. Notice the use of default values. Sometimes a profile returned won’t have all the information filled out or the user will keep it private. In this case, you handle it to prevent an error.
You should be able to login to your app now--try it!
Submit your page when you think you’ve got it right. If you’re running into errors, you can check out the project completed up to this point here.

2.17 Set up the Environment

The following challenges will make use of the chat.pug file. So, in your routes.js file, add a GET route pointing to /chat which makes use of ensureAuthenticated, and renders chat.pug, with { user: req.user } passed as an argument to the response. Now, alter your existing /auth/github/callback route to set the req.session.user_id = req.user.id, and redirect to /chat.
Add http and socket.io as a dependency and require/instantiate them in your server defined as follows:
‘‘‘javascript
const http = require(’http’).createServer(app);
const io = require(’socket.io’)(http);
‘‘‘
Now that the _http_ server is mounted on the _express app_, you need to listen from the _http_ server. Change the line with app.listen to http.listen.
The first thing needing to be handled is listening for a new connection from the client. The on keyword does just that- listen for a specific event. It requires 2 arguments: a string containing the title of the event thats emitted, and a function with which the data is passed though. In the case of our connection listener, we use socket to define the data in the second argument. A socket is an individual client who is connected.
To listen for connections to your server, add the following within your database connection:
‘‘‘javascript
io.on(’connection’, socket => {
console.log(’A user has connected’);
});
‘‘‘
Now for the client to connect, you just need to add the following to your client.js which is loaded by the page after you’ve authenticated:
‘‘‘js
/*global io*/
let socket = io();
‘‘‘
The comment suppresses the error you would normally see since ’io’ is not defined in the file. We’ve already added a reliable CDN to the Socket.IO library on the page in chat.pug.
Now try loading up your app and authenticate and you should see in your server console ’A user has connected’!
Note:io() works only when connecting to a socket hosted on the same url/server. For connecting to an external socket hosted elsewhere, you would use io.connect(’URL’);.
Submit your page when you think you’ve got it right. If you’re running into errors, you can check out the project completed up to this point here.

2.18 Communicate by Emitting

Emit is the most common way of communicating you will use. When you emit something from the server to 'io', you send an event's name and data to all the connected sockets. A good example of this concept would be emitting the current count of connected users each time a new user connects!
Start by adding a variable to keep track of the users, just before where you are currently listening for connections.
“‘js
let currentUsers = 0;
“‘
Now, when someone connects, you should increment the count before emitting the count. So, you will want to add the incrementer within the connection listener.
“‘js
++currentUsers;
“‘
Finally, after incrementing the count, you should emit the event (still within the connection listener). The event should be named 'user count', and the data should just be the currentUsers.
“‘js
io.emit('user count', currentUsers);
“‘
Now, you can implement a way for your client to listen for this event! Similar to listening for a connection on the server, you will use the on keyword.
“‘js
socket.on('user count', function(data) {
console.log(data);
});
“‘
Now, try loading up your app, authenticate, and you should see in your client console '1' representing the current user count! Try loading more clients up, and authenticating to see the number go up.
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.19 Handle a Disconnect

You may notice that up to now you have only been increasing the user count. Handling a user disconnecting is just as easy as handling the initial connect, except you have to listen for it on each socket instead of on the whole server.
To do this, add another listener inside the existing 'connect' listener that listens for 'disconnect' on the socket with no data passed through. You can test this functionality by just logging that a user has disconnected to the console.
“‘js
socket.on('disconnect', () => {
/*anything you want to do on disconnect*/
});
“‘
To make sure clients continuously have the updated count of current users, you should decrease the currentUsers by 1 when the disconnect happens then emit the 'user count' event with the updated count!
Note: Just like 'disconnect', all other events that a socket can emit to the server should be handled within the connecting listener where we have 'socket' defined.
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.20 Authentication with Socket.IO

Currently, you cannot determine who is connected to your web socket. While req.user contains the user object, that's only when your user interacts with the web server, and with web sockets you have no req (request) and therefore no user data. One way to solve the problem of knowing who is connected to your web socket is by parsing and decoding the cookie that contains the passport session then deserializing it to obtain the user object. Luckily, there is a package on NPM just for this that turns a once complex task into something simple!
Add passport.socketio, connect-mongo, and cookie-parser as dependencies and require them as passport-SocketIo, MongoStore, and cookieParser respectively. Also, we need to initialize a new memory store, from express-session which we previously required. It should look like this:
“‘js
const MongoStore = require('connect-mongo')(session);
const URI = process.env.MONGO_URI;
const store = new MongoStore({ url: URI });
“‘
Now we just have to tell Socket.IO to use it and set the options. Be sure this is added before the existing socket code and not in the existing connection listener. For your server, it should look like this:
“‘js
io.use(
passportSocketIo.authorize({
cookieParser: cookieParser,
key: 'express.sid',
secret: process.env.SESSION_SECRET,
store: store,
success: onAuthorizeSuccess,
fail: onAuthorizeFail
})
);
“‘
Be sure to add the key and store to the session middleware mounted on the app. This is necessary to tell _SocketIO_ which session to relate to.
Now, define the success, and fail callback functions:
“‘js
function onAuthorizeSuccess(data, accept) {
console.log('successful connection to socket.io');
accept(null, true);
}
function onAuthorizeFail(data, message, error, accept) {
if (error) throw new Error(message);
console.log('failed connection to socket.io:', message);
accept(null, false);
}
“‘
The user object is now accessible on your socket object as socket.request.user. For example, now you can add the following:
“‘js
console.log('user ' + socket.request.user.name + ' connected');
“‘
It will log to the server console who has connected!
Submit your page when you think you've got it right. If you're running into errors, you can check out the project up to this point here.

2.21 Announce New Users

Many chat rooms are able to announce when a user connects or disconnects and then display that to all of the connected users in the chat. Seeing as though you already are emitting an event on connect and disconnect, you will just have to modify this event to support such a feature. The most logical way of doing so is sending 3 pieces of data with the event: the name of the user who connected/disconnected, the current user count, and if that name connected or disconnected.
Change the event name to 'user', and pass an object along containing the fields 'name', 'currentUsers', and 'connected' (to be true in case of connection, or false for disconnection of the user sent). Be sure to change both 'user count' events and set the disconnect one to send false for the field 'connected' instead of true like the event emitted on connect.
“‘js
io.emit('user', {
name: socket.request.user.name,
currentUsers,
connected: true
});
“‘
Now your client will have all the necessary information to correctly display the current user count and announce when a user connects or disconnects! To handle this event on the client side we should listen for 'user', then update the current user count by using jQuery to change the text of #num-users to '{NUMBER} users online', as well as append a <li> to the unordered list with id messages with '{NAME} has {joined/left} the chat.'.
An implementation of this could look like the following:
“‘js
socket.on('user', data => {
$('#num-users').text(data.currentUsers + ' users online');
let message =
data.name +
(data.connected ? ' has joined the chat.' : ' has left the chat.');
$('#messages').append($('').html('' + message + ''));
});
“‘
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

2.22 Send and Display Chat Messages

It's time you start allowing clients to send a chat message to the server to emit to all the clients! In your client.js file, you should see there is already a block of code handling when the message form is submitted.
“‘js
$('form').submit(function() {
/*logic*/
});
“‘
Within the form submit code, you should emit an event after you define messageToSend but before you clear the text box #m. The event should be named 'chat message' and the data should just be messageToSend.
“‘js
socket.emit('chat message', messageToSend);
“‘
Now, on your server, you should be listening to the socket for the event 'chat message' with the data being named message. Once the event is received, it should emit the event 'chat message' to all sockets io.emit with the data being an object containing name and message.
In client.js, you should now listen for event 'chat message' and, when received, append a list item to #mes-

sages with the name, a colon, and the message!
At this point, the chat should be fully functional and sending messages across all clients!
Submit your page when you think you've got it right. If you're running into errors, you can check out the project completed up to this point here.

3 Quality Assurance Projects

3.1 Metric-Imperial Converter

Build a full stack JavaScript app that is functionally similar to this: https://metric-imperial-converter.freecodecamp.rocks/. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.
Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

3.2 Issue Tracker

Build a full stack JavaScript app that is functionally similar to this: https://issue-tracker.freecodecamp.rocks/. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.
Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

3.3 Personal Library

Build a full stack JavaScript app that is functionally similar to this: https://personal-library.freecodecamp.rocks/. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but must be publicly visible for our testing.
Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

3.4 Sudoku Solver

Build a full stack JavaScript app that is functionally similar to this: https://sudoku-solver.freecodecamp.rocks/. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.
Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember to save the link to your project somewhere safe!

3.5 American British Translator

Build a full stack JavaScript app that is functionally similar to this: https://american-british-translator.freecodecamp.rocks/. Working on this project will involve you writing your code on Repl.it on our starter project. After completing this project you can copy your public Repl.it URL (to the homepage of your app) into this screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.
Start this project on Repl.it using this link or clone this repository on GitHub! If you use Repl.it, remember

to save the link to your project somewhere safe!