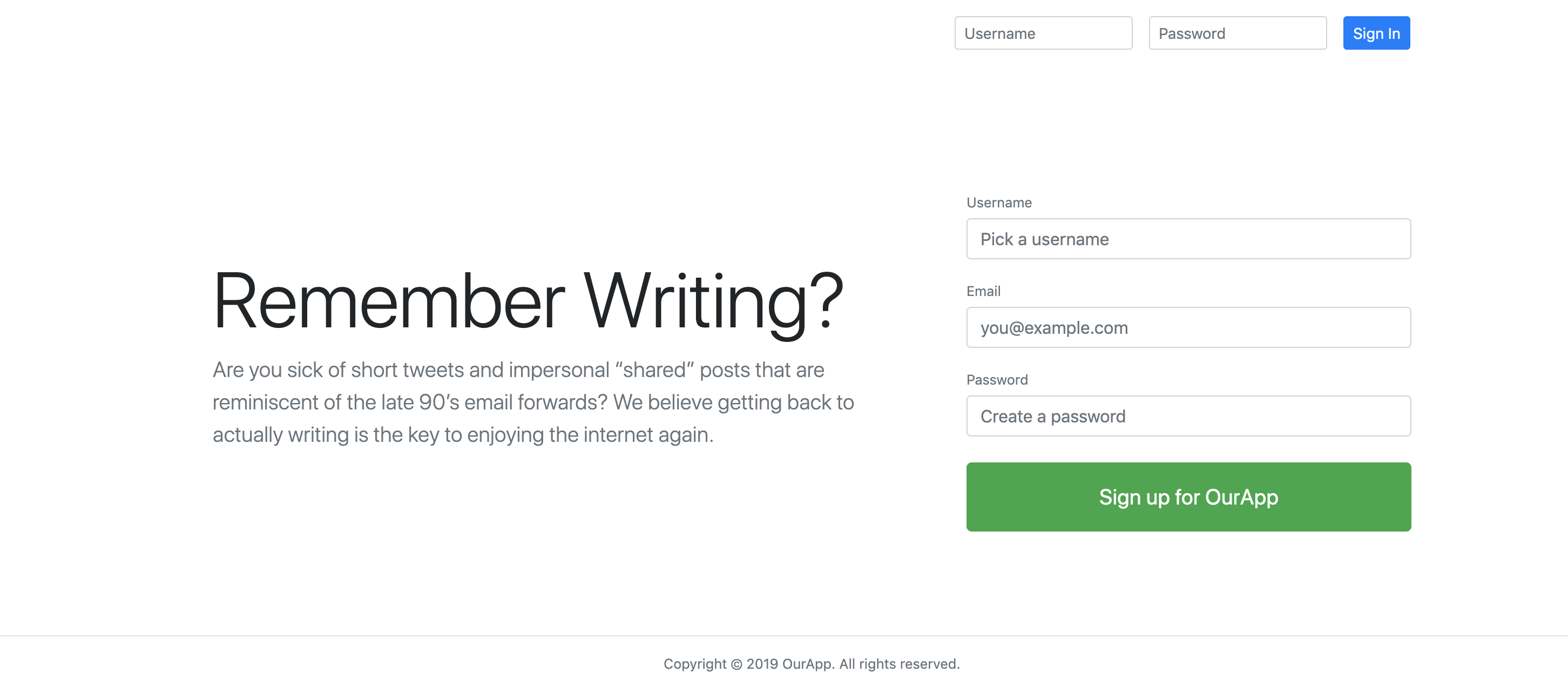
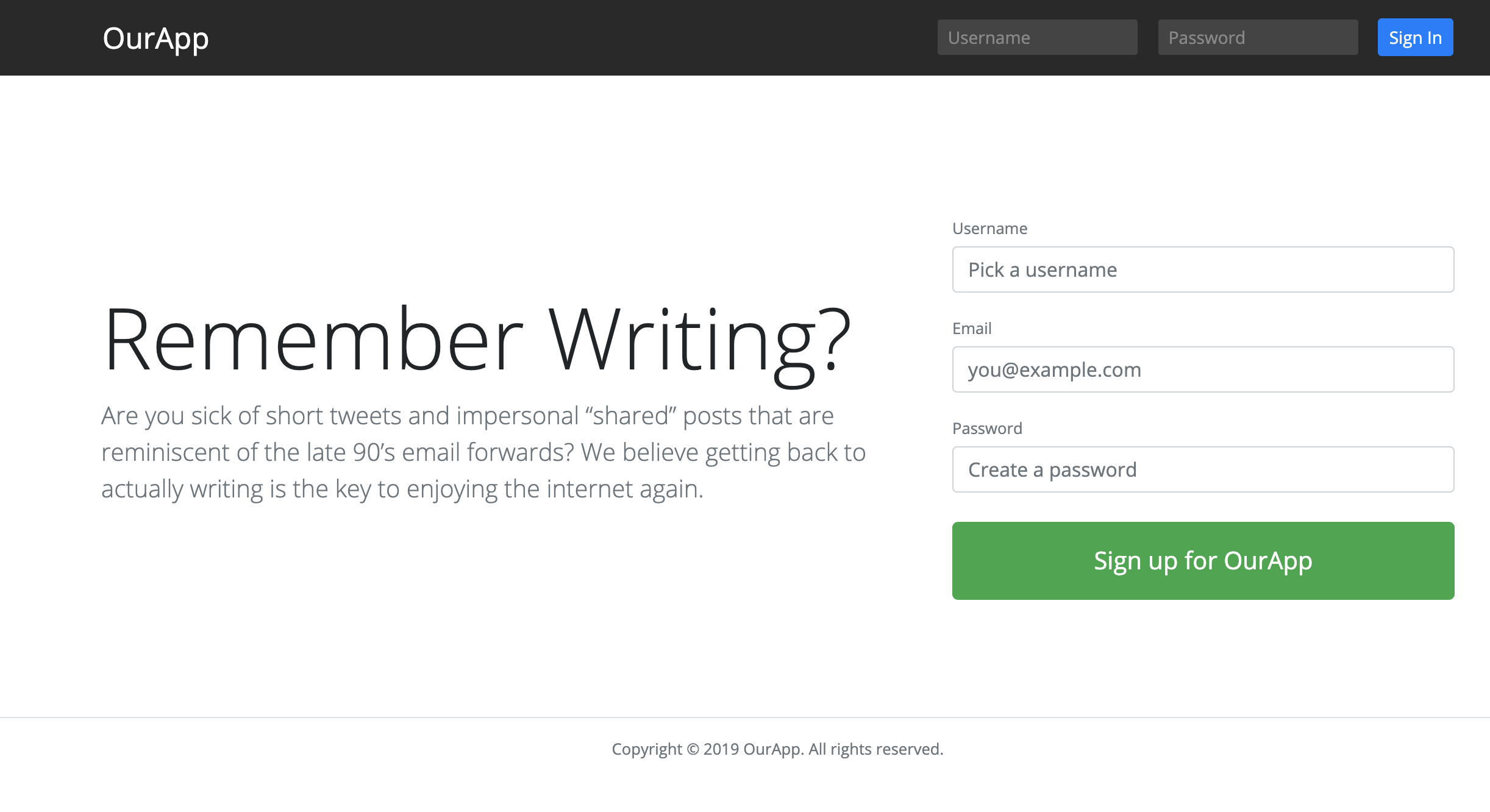
**Section 5: Starting our Complex App 2**

40. What’s next?

41. Let’s begin App 2

* Create a folder. This will be the main folder and the main file named app.js
* Use the express framework that will listen to incoming requests. Pick up express from npm. Create the json file that will serve as the ingredient list by npm init -y. json file will keep track of the packages needed.
* Install now the express, npm install express. We can now use the express framework in app.js file. const express = require(‘express’)
* Setup the express server, ‘node\_modules’ is now available. Create a variable calling the express. let app = express()
* Tell the app what should it do when it receives a get request to the base url. .get consists of 2 arguments. 1st is the base url (‘/’). 2nd is a function that uses (req, res) parameters in the function. Express will pass the objects into the function when it’s called. res.send is the respond send when function is called. include the html template in res.send (home-guest.html).
* Create a new folder named views, it will contain html files. Create the home-guest.ejs, paste the raw html code.
* Connect the home-guest.ejs to app.js. Configure app.js, app.set(‘views’, ‘views’). 1st ‘views’ is fixed, 2nd ‘views’ is the folder created to contain the html templates. Express now knows where to find the ‘views’ or the templates. Tell express what templating engine to be use, app.set(‘view engine’, ‘ejs’). Ejs is one of the engines, others are pug, handlebar. Each template has its own syntax and features. Install ejs, npm install ejs. Instead of res.send, render the hopme-guest, res.render(‘home-guest’)
* Tell the app to begin listening for in-coming request at port 3000.
* Other features are unavailable because of the absence of the css. 
* Create a new folder that will contain the css and js files. Make the folder accessible, app.use(express.static(‘public’)), setup the main.css inside public. Link the main.css to home-guest “/main.css”



* Setup the auto-restart. npm install nodemon. Setup the nodemon at package.json, under “scripts”, set “watch”: “nodemon app”, run auto restart by npm run watch.
* Notes:
* Express is the framework
* Node server is the web browser server
* ejs is the templating engine. HTMLs are .ejs
* This is the app.js code after 41.



42. Important Note about Package Versions to save frustrations

“watch”: “nodemon app”,

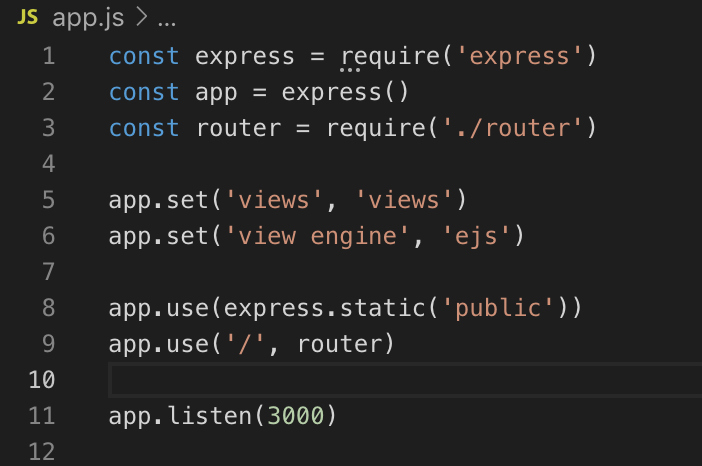
43. What is a router?

* Router has the responsibility of listing out URLs or route that wants to listen for and say what needs to happen in that route. Idea is to stay organize by organizing different files. Each file will then work with define responsibility and be route.
* Create a file in the root, router.js.
* Setup the router, create the mini app.js version in router.js
* Connect the router.js to app.js. in router.js; module.export = router in app.js; const router = require(‘./router’)
* Remove the are from app.get as it was defined in router.js. replace it with app.use(‘/’, router). The app should run as normal.

Notes:

* + require is use for packages not created and can pull in js files that were created. require function in node executes all the contents of the file and returns what that file exports and store in the variable.
  + module.exports = node.js will execute the “=” of module.export when its been called in. it can export numbers, strings, functions and objects.

App.js and router.js at the end of 43

44. What is controller?

* Controller is needed to simplify more the router.js, it separates the functions and logics. It will create different function under each controller files and router can easily executes those functions. Controller will organize by feature better and let the router just do the routing.
* Create a new subfolder “controllers”. Create the following files:
  + userController.js – goal is to export multiple functions that can be access from other js files. Configure as below:
    - Connect it to router.js const userController = require(‘’./controllers/userController’)
    - Modify the router.get to router.get(‘/’, userController.home) <<< this is for home page
    - This is for register page. Modify the home-guest.ejs’ <form> from “#” to “/register”

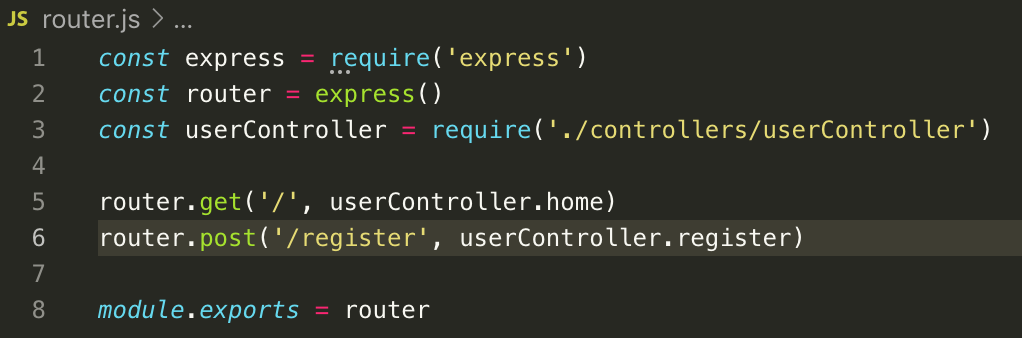


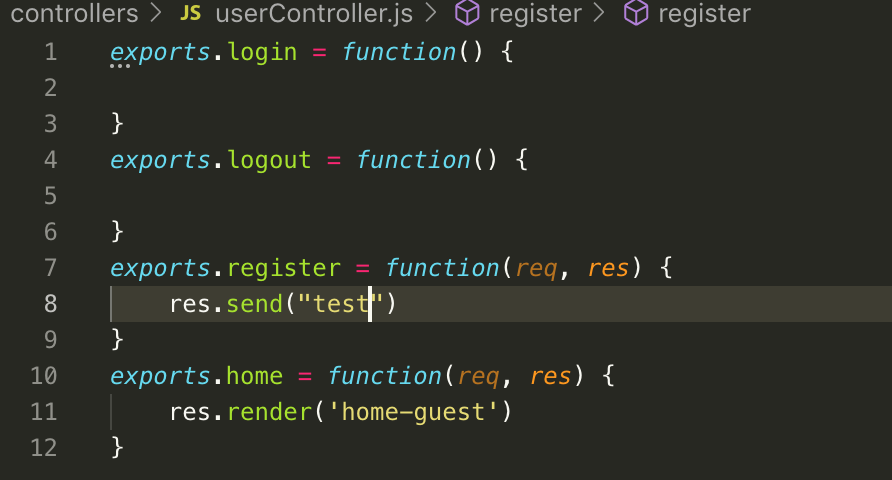
* + - In router.js, coinfigure the register = router.post(‘/register’, userController.register)
    - Assign the function of the exports.register pointing the render to
  + postControllers.js
  + followControllers.js
* Note:

MVC = model view controller; keep code organized.

* Files at the end of 44







45. Security note – temporarily do not use significant password.

46. What is a model? >>> define the structure of data. Nouns and verbs in English and properties and methods in objects.

* Configure the express app. Add app.use(express.urlencoded({extended: false})) --- this tells express to add the user submitted data on to request object and access from req.body. Add, app.use(express.json()). The application now accepts two most common ways of submitting data on the web. Traditional html form submit and sending a bit of json data.
* Create models subfolder, create User.js. create let User = function {} <<< this will serve as the constructor that will hold all the validations. Export this variable (module.exports = User. Import it at userController.js, const User = require(‘../models/User)
* At userController.js, Create let user = new User(req.body) under exports.register. req.body = data acquired from the form. “new” creates new object using User as the blueprint. User is defined at Users.js
* At User.js, create the variable constructor function(data), the data here is the recipient of the data coming from the req.body, the form submission.
* “this.data”, is the set property of the function. ‘data’ here is any name, this is supposed to be pointing at the let User as it is the one calling the function. However, because of “new” in the userController.js, it now points to the let user variable, and the new object is now assigned to let user variable.
* User.prototype.register = function() {} = this syntax will make javascript not to create a copy of the function for new object created. All object created using the let User constructor has the access to this function. ‘register’ is the method. This is very useful for thousands of new object created.
* Note:

Constructor function – reusable blueprint of an object.

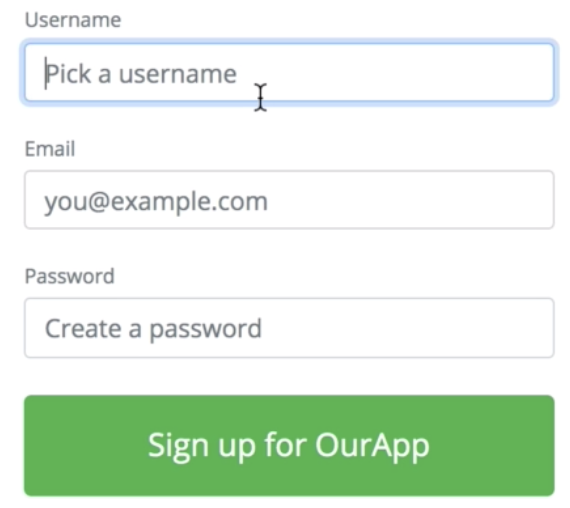
“this” keyword is use to point to what calls the function. This made the blueprint flexible.

After 46





47. Adding validation to our model

* Configure .register at userFunction.js to make it work. .register was derived. The objective is to
* Create a validation requiring the 3 input filled not to be blank and correct information is to be input upon pressing the “Sign up for our App” button.
* 
* At User.js, create the following criteria.
* Step 1. Validate user data
* Step 2. Only if there’s no validation error then save the user data into a database.
* Create User.prototype.register = function(){

this.validate() <<< see note 1

}

* Create an if statement to User.prototype.validate = function() {

If (this.data.username == “){

}

}

* Create this.errors = [] in User.js>>> this will push errors in the above if statement if statement is not met, this is same as the email and password. Complete the if statement and push this.error = [].
* Create an if statement in exports.register. this will validate if there’s no value input in the 3 fields. If (user.errors.length) = .errors is an array thus giving an access to the method .length. the statement will be true if the given .length value > 0.
* Additional validation:
* For security reason in password,

let the password should be 12 characters, if (this.data.password.length > 0 $$ this.data.password.lemgth <8) {this.errors.push(“Password must be at least 8 characters )}

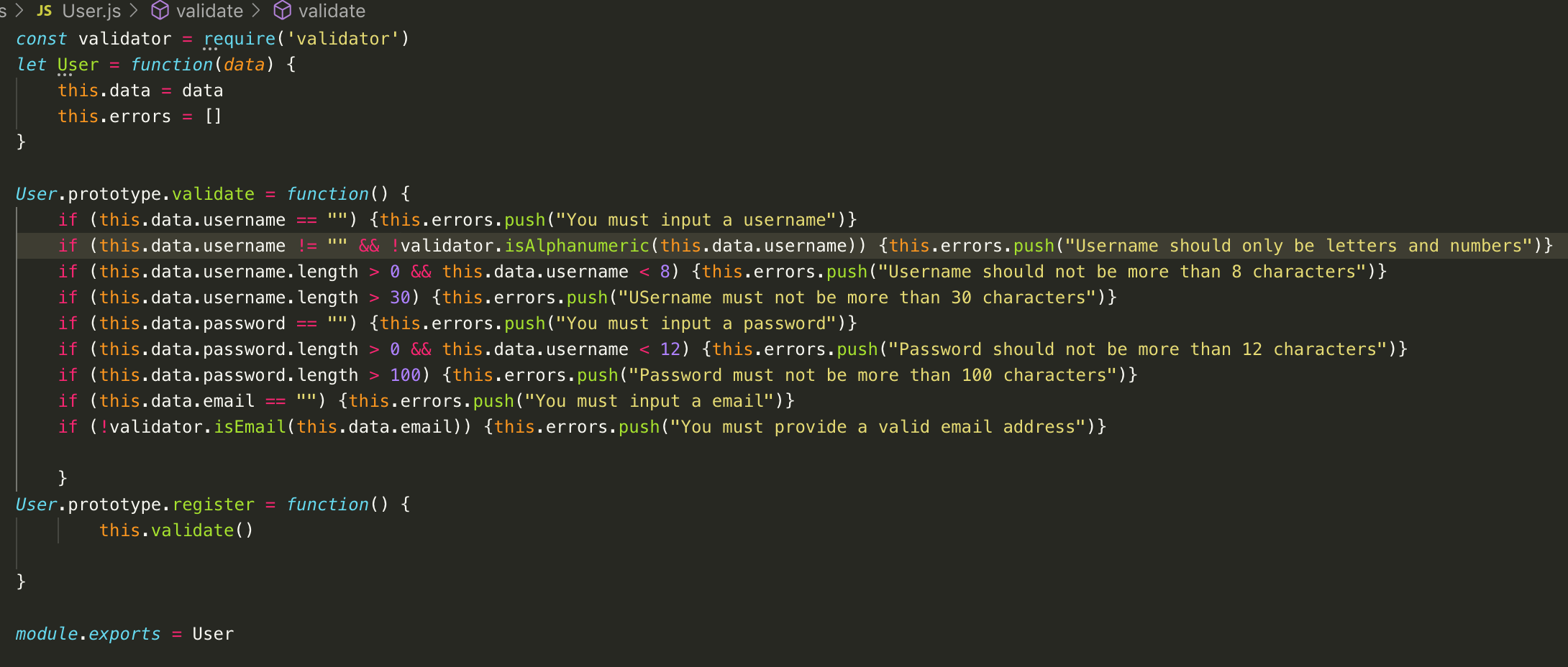
let password > 100 characters.

* For username set as well. Should use valid characters if(this.data.username != “” && validator.isAlphanumeric(this.data.username)) {“Username can only contain letters and numbers}
* For email, use npm package email validator package. Npm install validator. Import the package downloaded. Const validator = require (‘validator’).

If (validator.isEmail(this.data.email)){this.error.push(“You must provide a valid email)}. Use “!” as reverse application.

Note:

1. this.validate points to whoever calls the current function. In this case the current function is the .register and is being called at userController.js via user.register(). This made this ‘user’ be ‘this’, then it has an access to the prototype blueprint .validate. Thus this.validate also means user.validate.
2. other validation for username for incvalid characters, @ for email and number of characters in password are pending.





48. Quick Miscellaneous Clean Up

* Input field should accept only strings and numbers, no object or arrays.
* @ userController add, @ User.prototype.register = function(){

this.cleanup() <<< clean up can be other name

this.validate()

}

* @userController.js, create a blueprint constructor for cleanup; User.prototype.cleanUp = function(){
* If(tyoeof(this.data.username)) != “string) {this.data.username = “”}

}

* To get rid of bogus properties; @ user.js. this is to purify data property

this.data = {

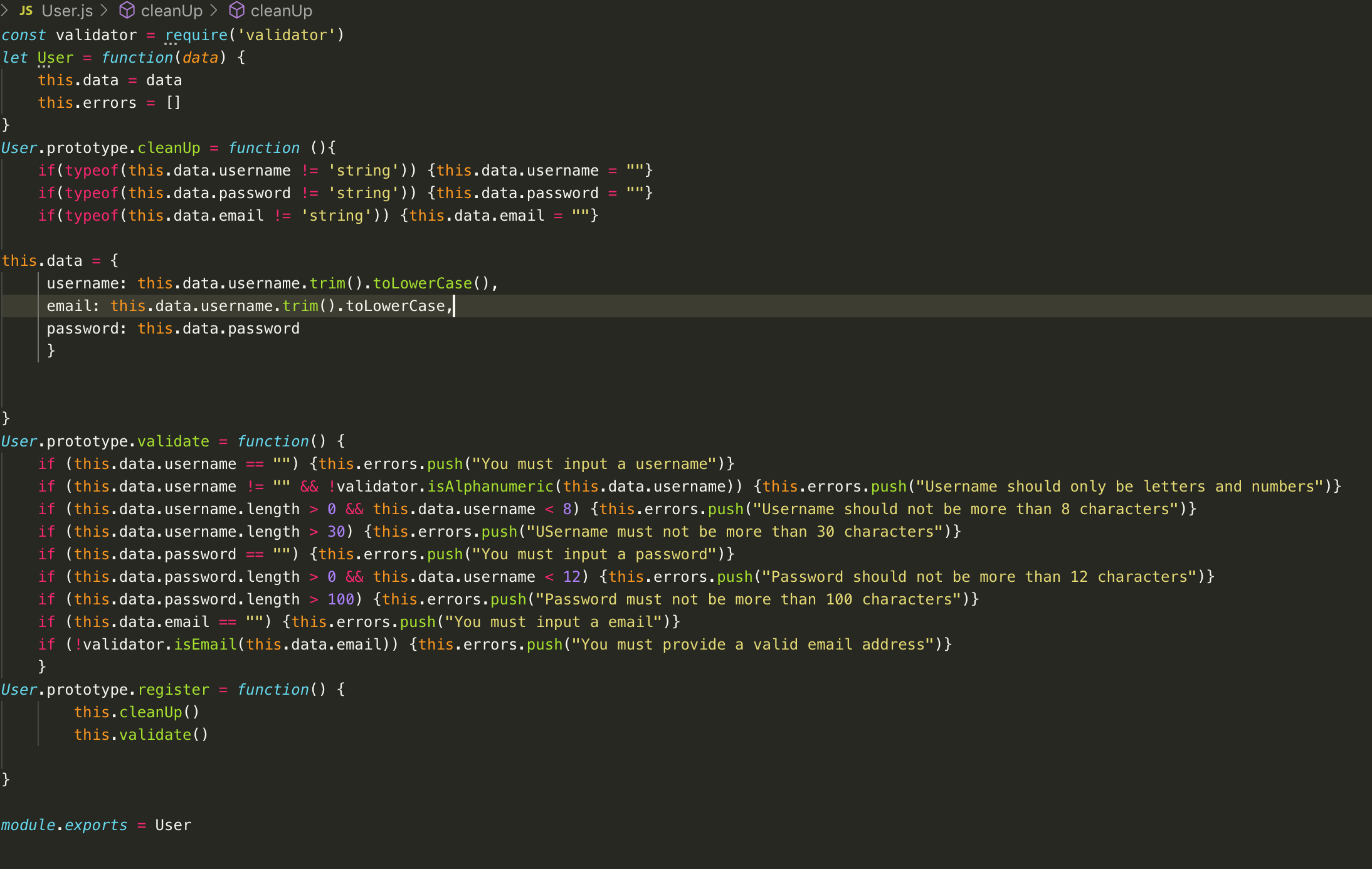
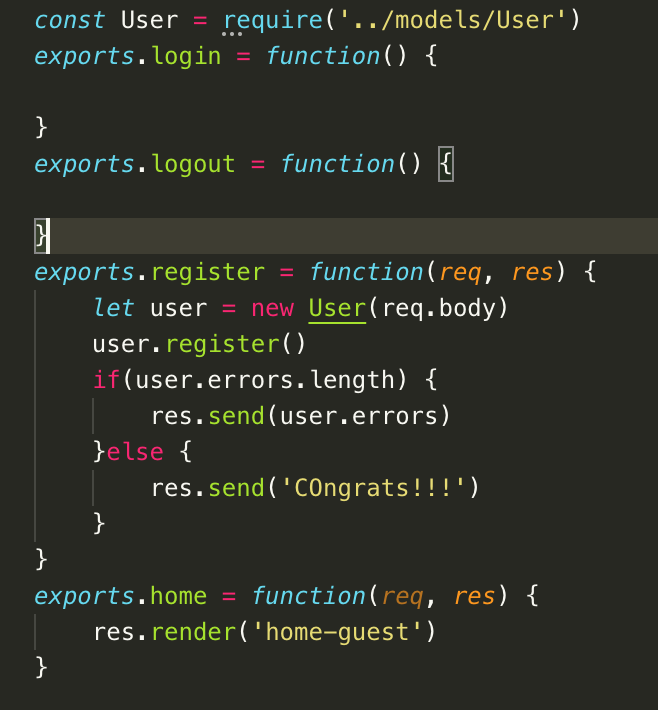
username: this.data.username.trim().toLowerCase()

email: this.data.usernam.trim().toLowerCase()

password: this.data.password

}

* .trim = For further cleanup

49. Connecting to Database in a Re-usable Fashion

* Create a mongoDB
* Connect the app to new DB once in a separate file who’s entire job is to connect to DB and use in each model.
* Create ‘db.js’ file and create a connection to mongoDB. Npm install mongodb.
* @ db.js, const mongodb = require(‘mongodb’)
* Mongodb(a, b, c)

a = connectionString, link from mongoDB

b = {useNewURLParser: true, useUnifiedTopology: true}

c = function(error, client){

module.exports = client.db()

const app = require(‘./app’)

app.listen(3000)

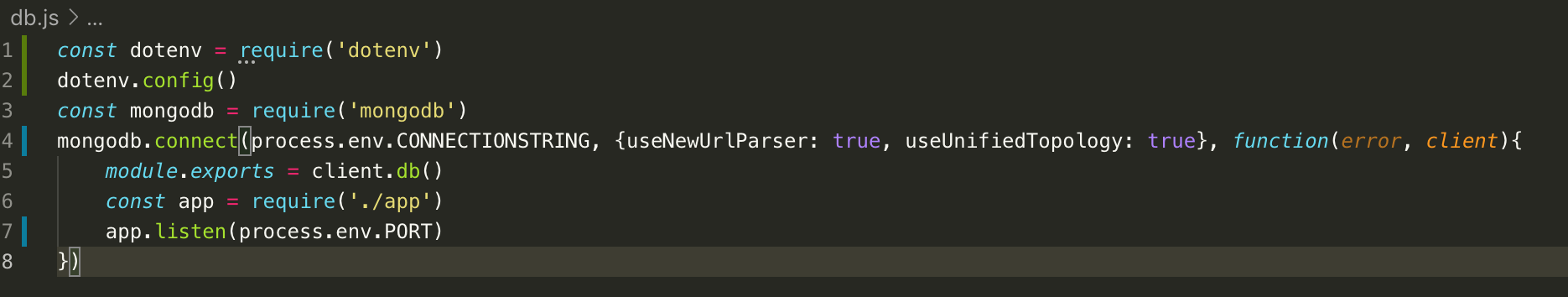
}

* **Above will make the app to display after a successful connection to db. The setup is perfect now for MVC setup.**
* @ app.js; remove app.listen(3000) then replace with module.exports = app. This just transfers the connection to db to db.js from app.js file.
* Then revise “watch”: “nodemon app” to “nodemon db”
* Connect Users.js; const usersCollection = require(‘../db’).collection(“users”); “users” is from mongoDB collection. db, because it was module.exports = object client.db().
* This can now do CRUD
* In Users.prototype.register = function…. + if statement

If(!this.errors.length) {usersCollection.insertOne(this.data)}

50. Best Practive Time out: Environment Variables

* Npm install dotenv
* .env is important so that the username and password are not hard coded at d.js.
* Create .env file; CONNECTIONSTRING= the connection string from mongoDB.
* Remove the connectionStrings in db.js
* Add const dotenv = require(‘dotenv’)
* Dotenv.config() <<< config will run all the codes inside the created .env file.
* Can use now the process.env.CONNECTIONSTRING in replace of connectionStrings in mongodb.connect(….)
* At .env; create the variable PORT=3000. Replace now the 3000 in app.listen() in db.js.





51. Quick Note

52. Letting users Log in

* Goal is to let user submit the main form and determine if the username and password is correct base from existing database.
* Replace # in the home-guest.ejs. to /login; this is for header. The user will send a POST request top that /login url.
* Create new route to the app. Configure router.js = router.post(‘/login’, userController.login). method .login is from the userController.js.
* Define the method .login in userController.js. supply req, res parameter. In the function, create a new object from the blueprint using = new User(req.body)
* Create the method login in the model as the model file should handle all the logic and data managing.
* Open the User.js and create the blueprint for .login; User.prototype.login = function(){

this.cleanUp()

usersCollections.findOne({username: this.data.username}, function(err, attemptedUser){

if(attemptedUser && attemptedUser.password == this.data.password){

}else {

Console.log(‘sorry’)

}

})

}

If object this.data.username find a matching data in the DB, it will then pass that data to attemptedUser (it can be any name).

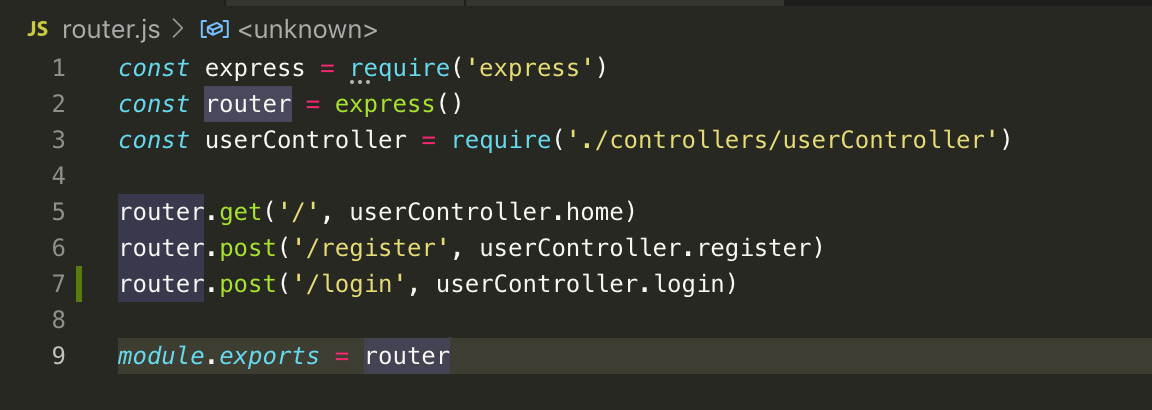
Console.log is only for testing

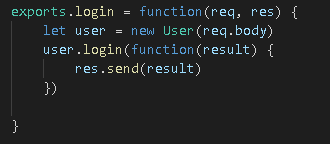
Function(err, attemptedUser) is originally a function. It was then later changed to an arrow function so that ‘this’ will point to .findOne. if anonymous function is use the ‘this’ will point to a global variable as .findOne is not an object. it’s a function of mongoDB.

At User.js, assign (callback) as the parameter of the method .login function.

Note:

It’s not the job of the model to send route, it’s the job of the controller.





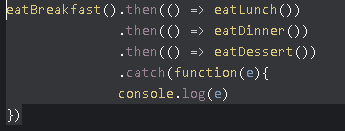
53. What is a promise Part 1

* More logical and modern way of handling timing situation of the login function
* Goal is to convert the traditional callback method to a Promise.
* At User.prototype.login, cut and replace the content of the function with return new Promises(function(resolve, reject){

})

Then, paste the cut syntax inside the function of the Promise.

* The first callback will now then the resolve, second callback is the reject
* Promise is a constructor or a blueprint that can be use to create a new Promises
* Change the anonymous function to arrow function so that it will not manipulate the this.data.username
* At userController.js,remove the function in user.logi() <<< we will use now the Promise blueprint here. To use a Promise add .then().catch().
* .then() is for resolve and .catch() is for the reject
* Create a function with a parameter in .then(function(){}) so that if ever the Promise is resolve it can pass it to the parameter of the function and then to res.send()
* .catch(function(){}), happens if the Promise don’t happen. The .catch will send the reject or the error to the parameter of the function, and the function will send the error message via the parameter of the function.
* At User.js, convert the traditional callback to Promise. Cut the if statement
* Remove the 2nd argument of the .findOne method.
* Add the .then().catch() to the findOne method. Paste the if statement inside the arrow function. Assign attemptedUser as the parameter of the .then() so that it will match the if statement.  
  .catch statement will send a response if there’s a database breakdown.
* At this point Promise was created and leverage it, and leverage the pre-existing Promise from mongoDB. Succeeding will make clear what is Promise.
* See the exercise, review Promise further.
* Async await >>> await let the function to finish before it executes the next function. However it only runs for Async function. Do the exercise to visualize the theory. Add Async before the function is the magic + the await in the succeeding functions.
* Below are the clean way to write a Promise:

54. What is a promise Part 2

55. Running Multiple Promises Efficiently when order doesn’t matter

56. Hashing user passwords

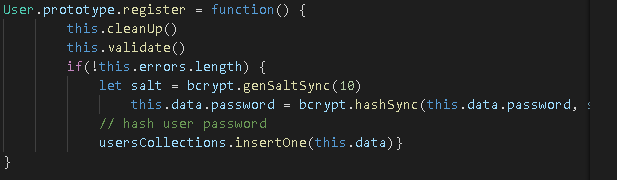
* Hashing is converting to a new value. This will protect the password of the user.
* Hashing is different from encrypting.
* Hashing is one way street
* Npm install bcryptjs
* At userController.js, call the bcrypt
* At User.js, go to .register model

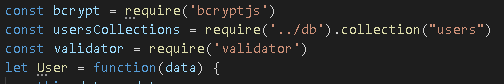


* Update the login function to update and match the hash password. Update after && portion. Change the number of characters to 50 characters



After 56





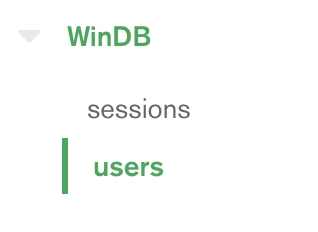
57. How can we identify or trust a request?

* Sessions – is a tool to medicate the amnesia between the browser and the server upon successful switching.
* Npm install express-session
* At App.js, configure the session
* At userController.js – add ‘req.session.user = {}. User can be of anyname. This enable the express to use session.
* Cookie – right clock the browser and go to Application tab, then cookies. Cookies are evidence of a user visiting the webpage.
* Tokens

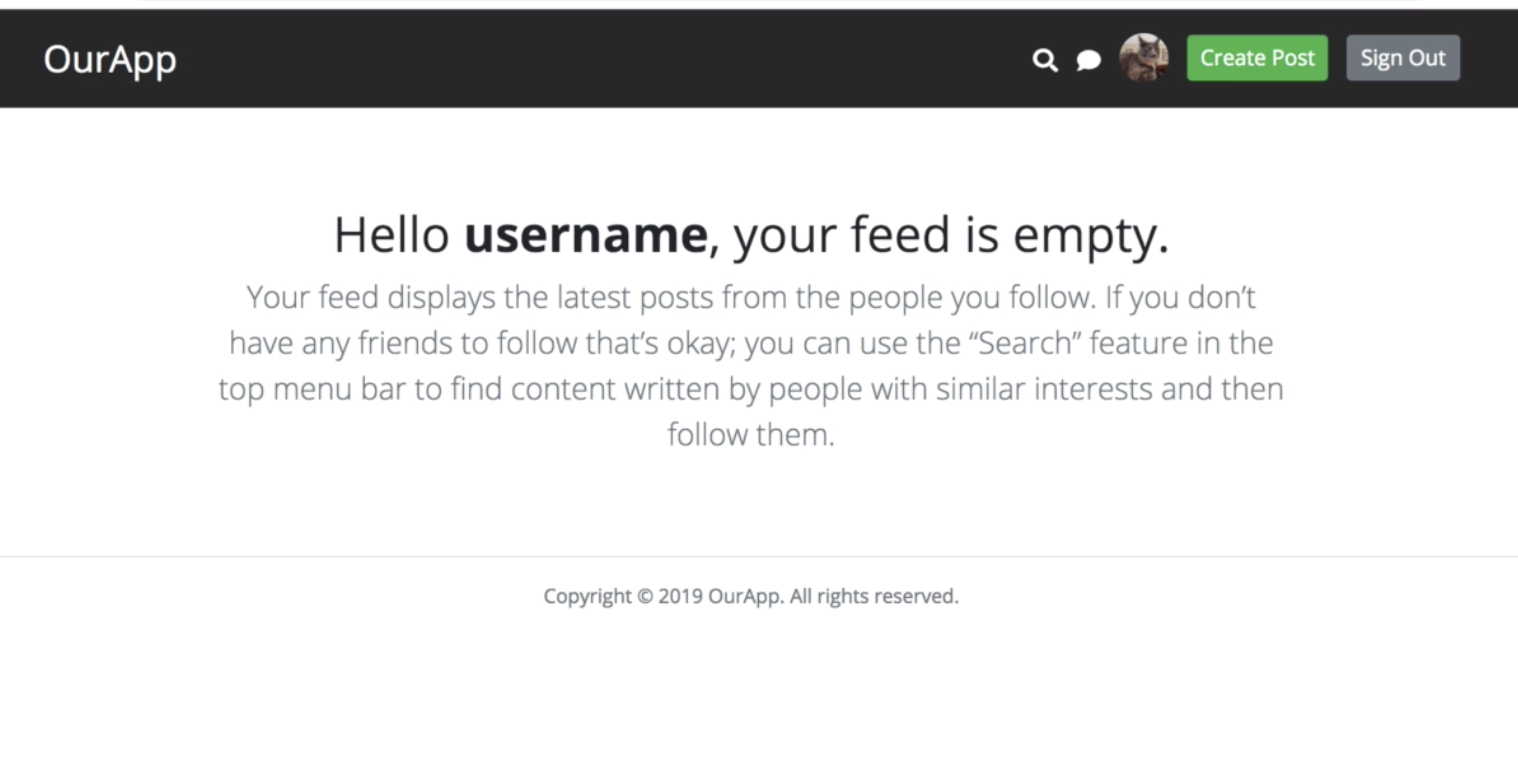
58. For those who are in a hurry

59. Understanding Sessions

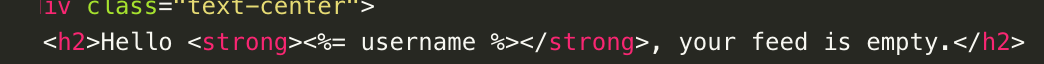
* Npm install connect-mongo
* Modify the sessions options at app.js. add const MongoStore = require(‘connect-mongo’)(session)
* Below secret: “Javascript is Nice”…., add:
* Store: new MongoStore({client: require(‘./db’)}).
* At db.js file, go to module.exports = client.db and remove .db
* At User.js, add.db() before .collection.
* 
* To:
* 
* Database now has a session collection and the cookie is now stored in database and not in the browser.



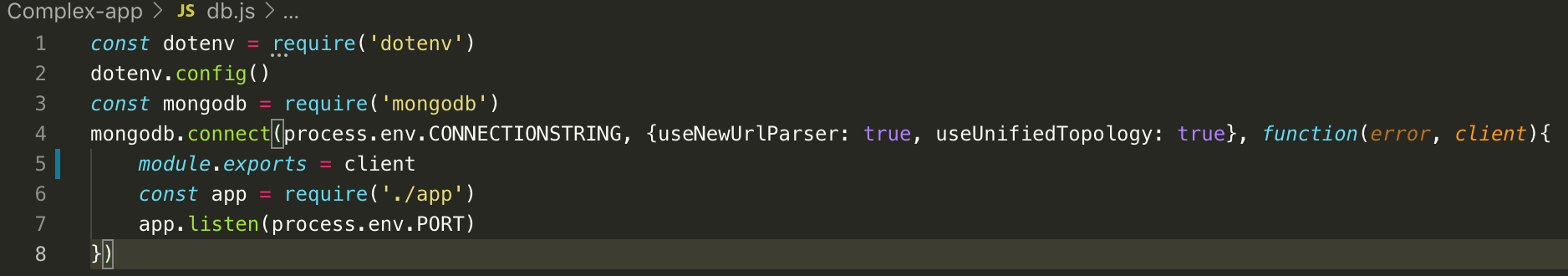
* Session has the capability to store the user information like the username or id.
* Goal, display below HTML upon successful log in and make it work.

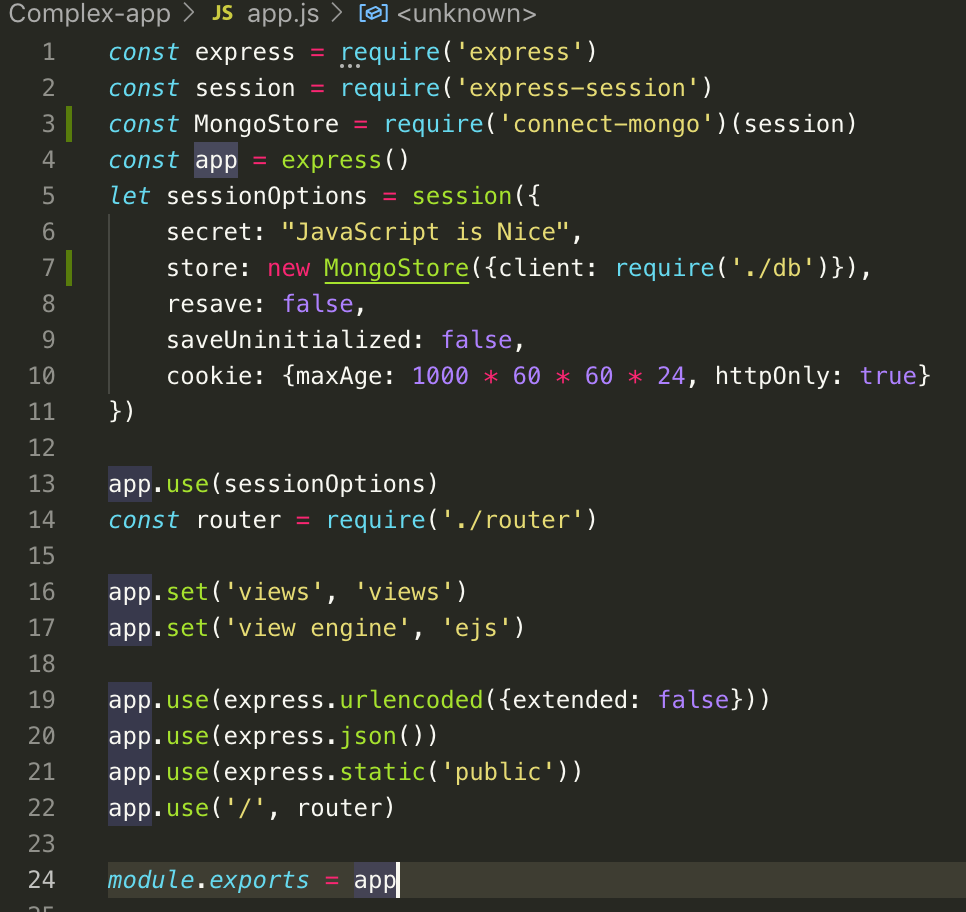


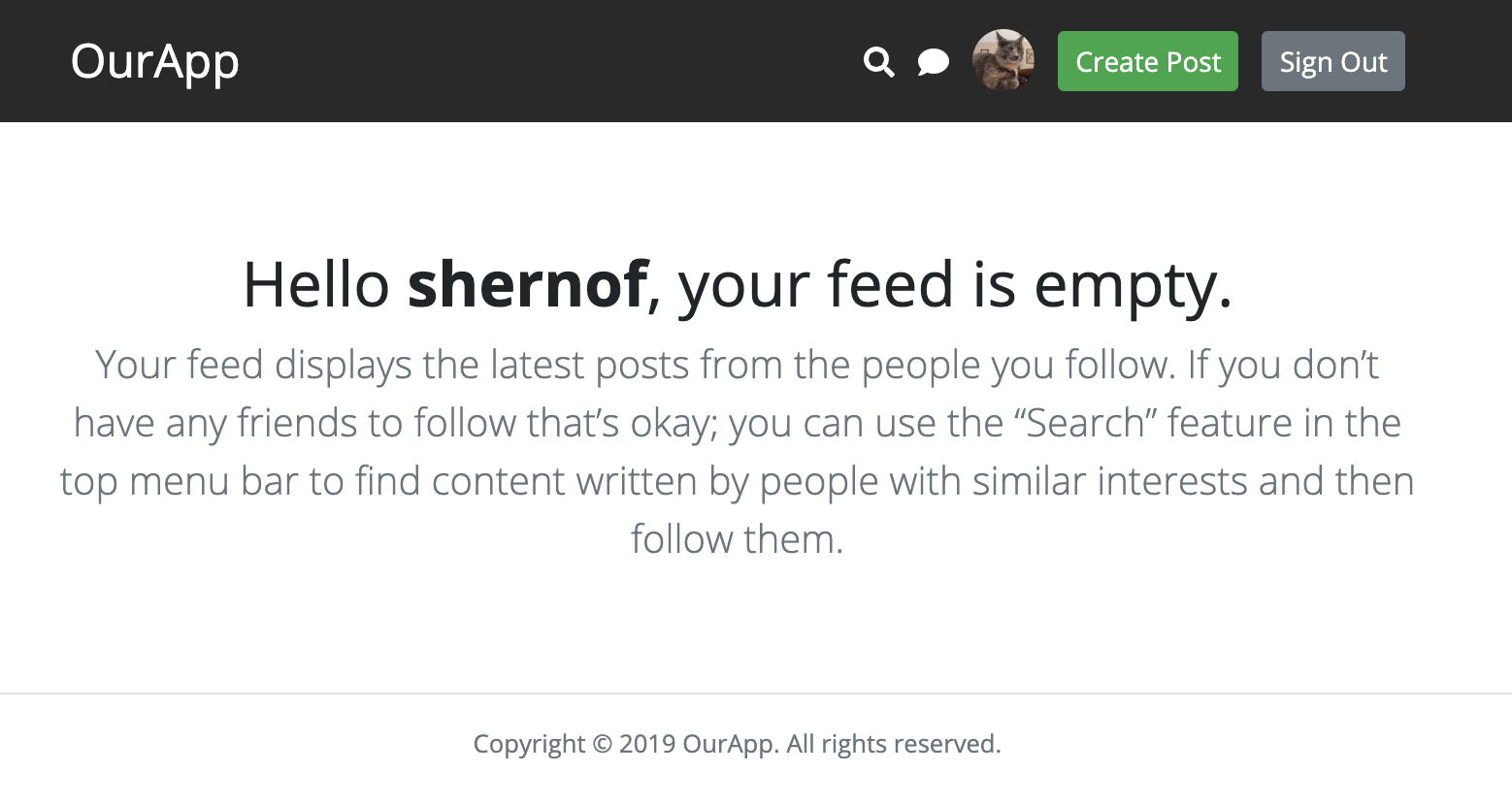
* Copy the raw html of home-logged-in-no-results to a new file inside the views folder.
* Then render (‘home-dashboard’, {username: req.session.user.username } at userController.js method .home. this will display the html above and replace **username** to user id that just logged in together with the modification of <strong>username</strong> of the home-dashboard.ejs. change the username to a dynamic type by changing it to <%= username %



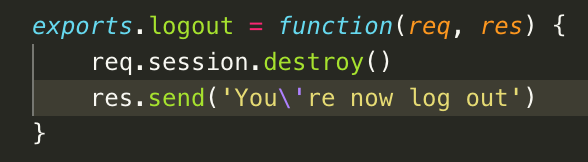
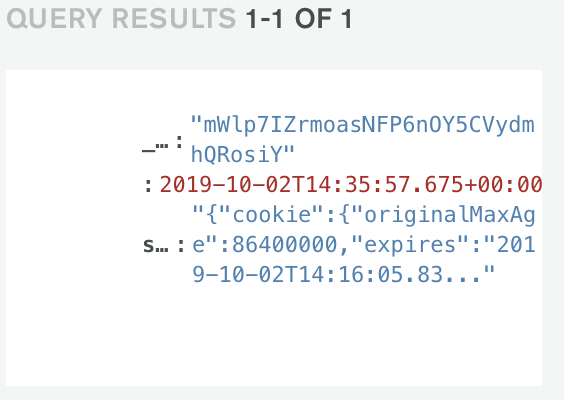






* 
* Above is the outcome after 60.

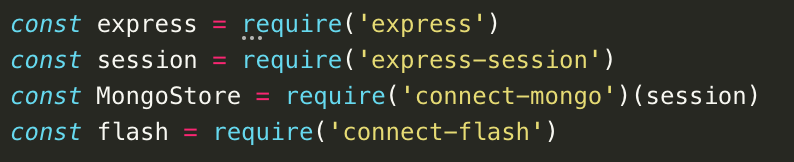
60. Letting users logout

* At home-dashboard go to Sign Out button, replace the place holder to /logout, then configure the router.js, router.post(‘/logout’, userController.logout), then build the object.logout at userController.js @ .logout.
*  .destroy will look at the database for
* matching session Id and destroy that upon incoming request. Below shows the logout and destroys one cookie.
*  
* Redirect now to home-guest page by creating a callback function @ .destroy in userController.js;
* Req.session.destroy(function() {
* Res.redirect(‘/’)
* })
* Try to sign out, it will now go to log in page again.
* Remove the intermediate “congrats” screen after login. At userController; remove res.send() and replace with req.session.save(function() {
* Res.redirect(‘/’)
* })
* 
* Redirect to login page if wrong username or password or “Flash Message”

61. Adding flash messages – show red message when failed login.

Npm install connect-flash

Add new line const flash = require(‘connect-flash’) at app.js; then app.use(flash)

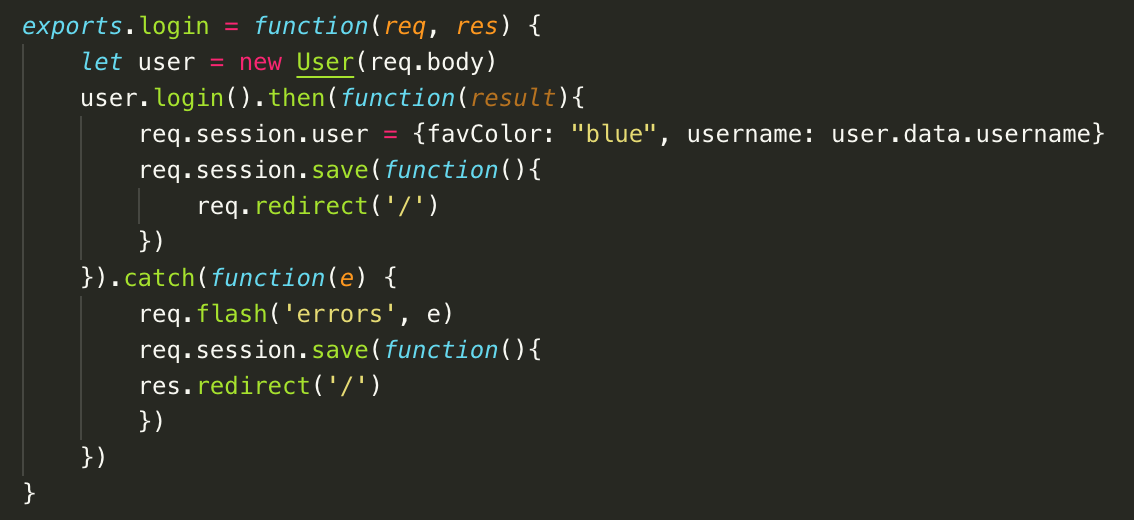




At userController, reconfigure .login and repla e .catch (e) with res.redirect(‘/’)

Add a new line above res.redirect(‘/’) and use the req.flash(‘errors’, e). e is from the function.

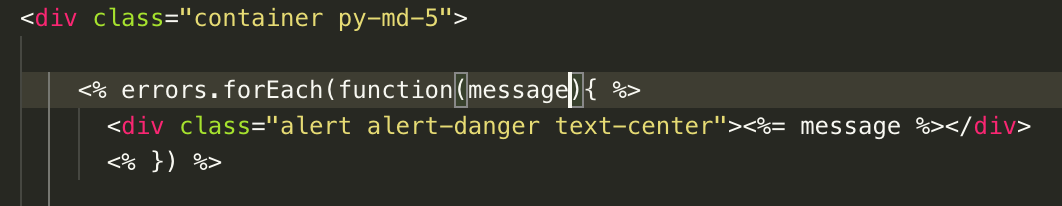
Manually tell session to save. Req.session.save(function()), then res.redirect(‘/’)



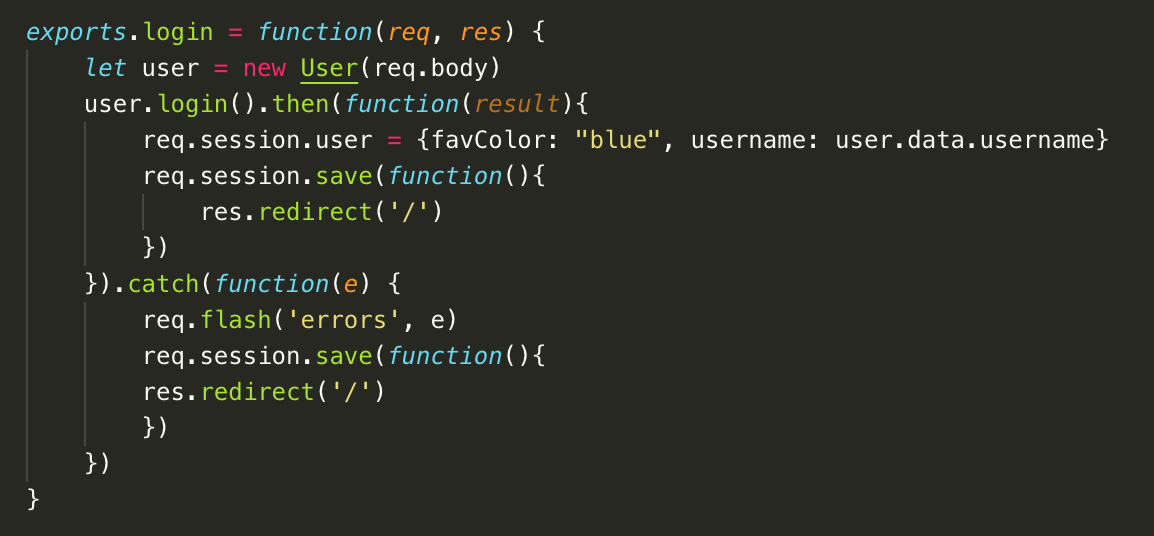




At home-guest.ejs

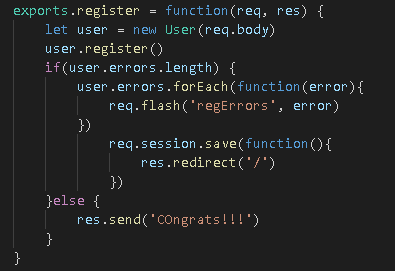






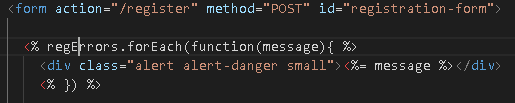
62. User registration improvements (Part 1)

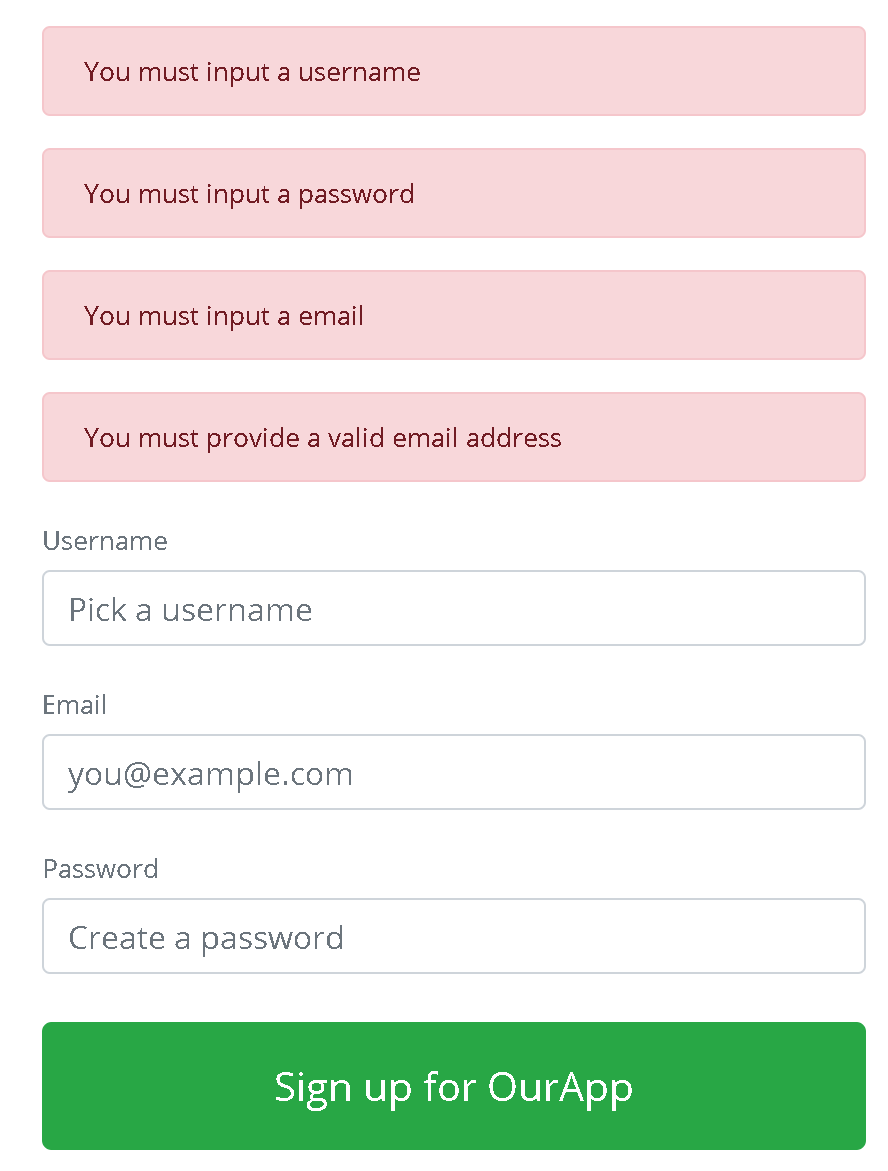
At userController.js, exports.register.





At home-guest, at <form action=”/register”

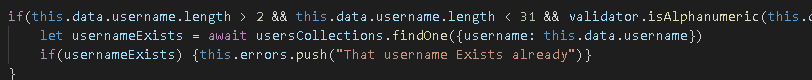
. 3 errors will be display now on top of the input fields.



Server-side validation is for security

Browser-side validation is for user experience.

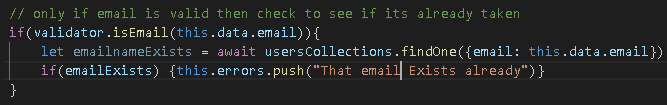
At User.js at User.prototype.validate



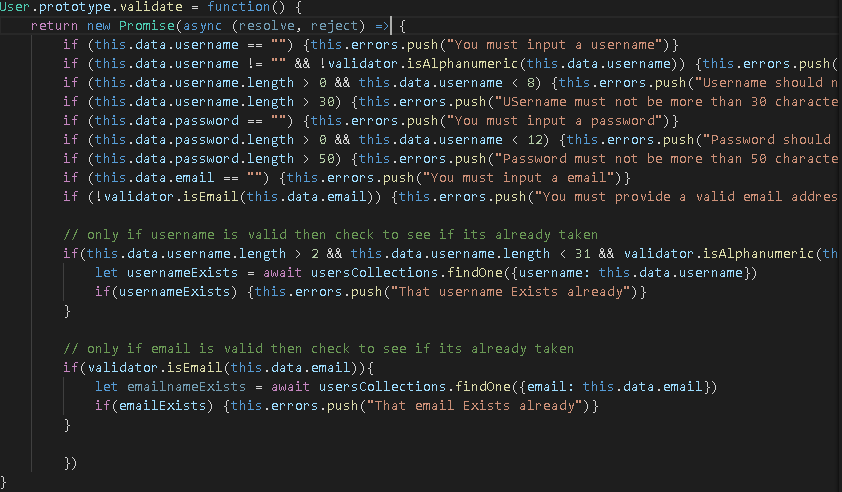
Because of the await add async to function(){}…

.

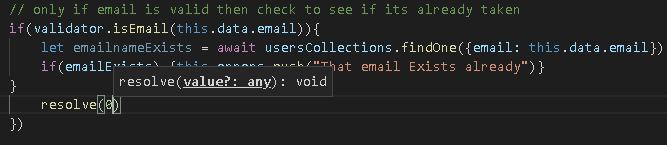
Add also for the email



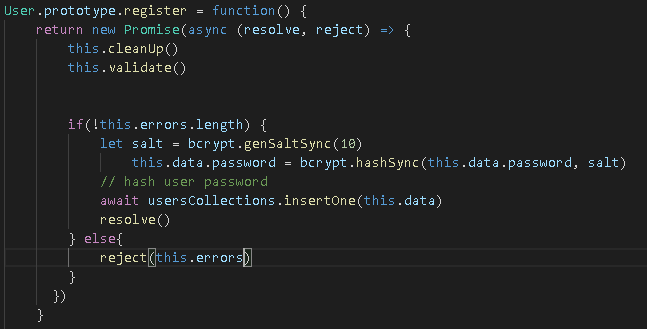
At User.prototype.register. Ensure do all the validation at this.validate. starting from async, select it til the end of the function.then paste it inside a new function. Change the async function to arrow function.



Update by adding resolve() in the .isEmail



Setup the Promise at User.prototype.register

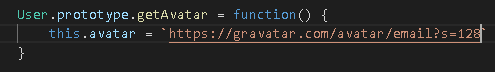


At userController.js

63. User Registration Improvements Part 2

64. Adding user profile photos

* Avatar should be per user and will appear everytime the user post.
* Gravatar – globally recognize avatar. = gravatar.com
* The avatar is base from email address but will not reveal the email.
* At User.js,



Npm install md5.

Replace the email above with ${md5(this.data.email)}.

Add this.getAvatar(0 in User.prototype.login

At userController.js, replace the favColor to {avatar: user.avatar…

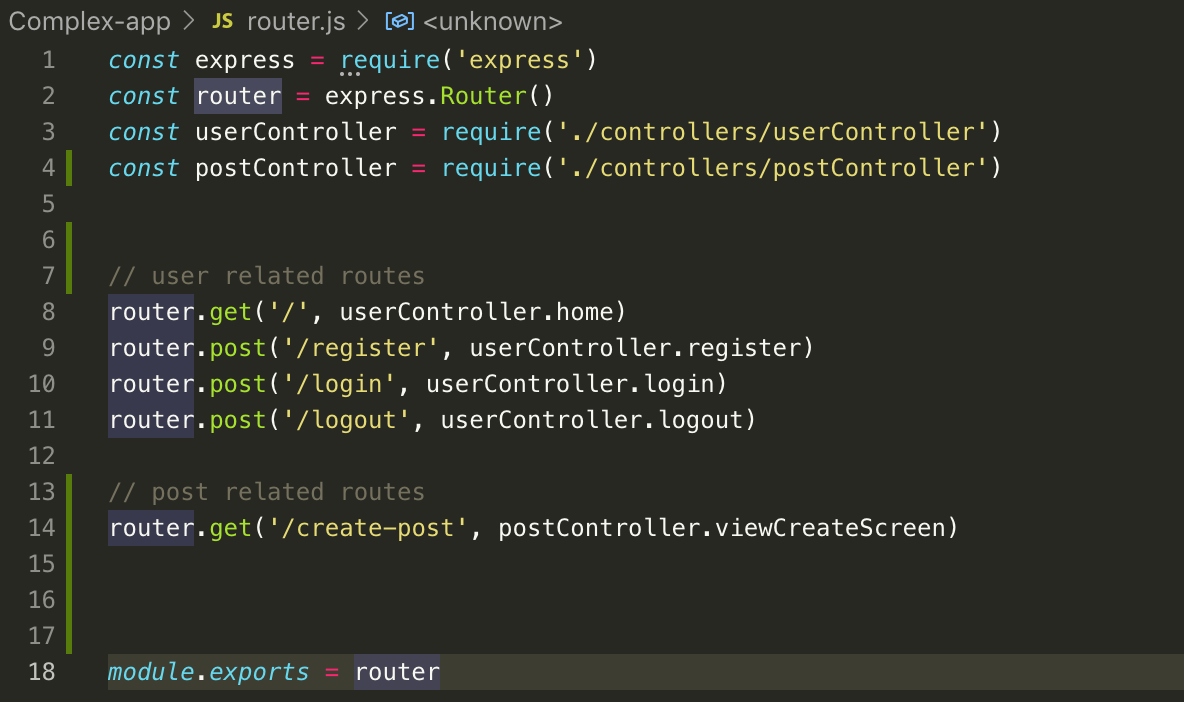
At exports.register, and exports..home, add avatar: user.avatar

At home-dashboard, at line 20, make the src = dynamic src=”<%= avatar %>

**Section 6 User Created Posts**

65. Letting users create Posts 1

Objective is to have the logged in user be able to create post by utilizing the postController.

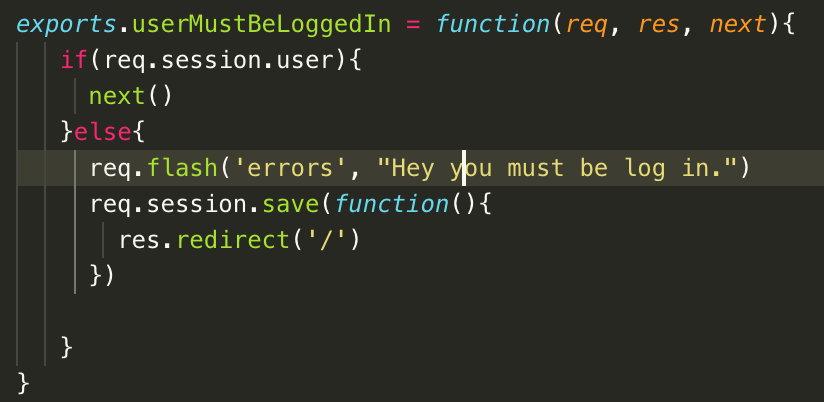
* Create the create-post.html.
* Adjust the Create Post button to send it to /create-post url. In home-dashboard.ejs, adjust it in the element btn >>> Create post and change the # to /create-post. configure the router.js
* 
* Setup the postController.js
* Export the functions .viewCreateScreen = function(req, res){
* Res.render(‘create-post’)

}

* Ensure that only logged in and legitimate user can only post.
* Create a re-usable function that will enable the valid user to create post when logged in. this will be coming from userController.

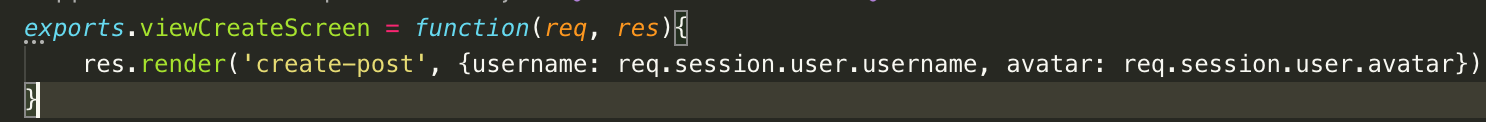
Go to router.js and include userController.usermustBeLoggedIn

}



Fix the header in create-post.

At postController, add another parameter in res.render(‘/create-post’)

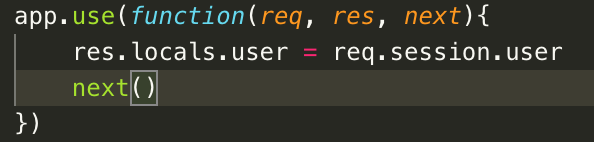


At views, create a subfolder named includes and create the header.js. copy the header of home-dashboard.ejs and paste in header.ejs. delete that header in home-dashboard and replace with <%- include(‘includes/header’) %>. Do it as well for create-post.ejs.

This header is now reusable. Include() is an ejs function. - is print or output

66. Letting users create Posts 2

**This section is more on removing the duplications in every html template.**

at app.js, 

this will utilize .locals that will remove the duplication and repetitive use of ‘{username: req.session.user.username, avatar: req.session.user.avatar}’. this code can now be removed from post and userControlloer. .user can be of any name.

 >>>>

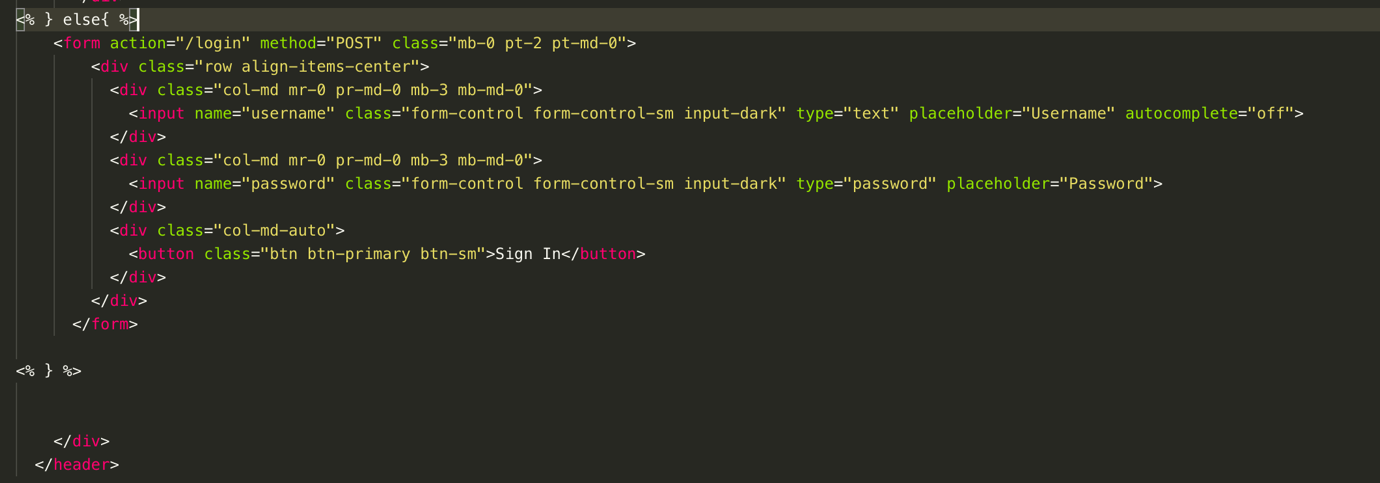


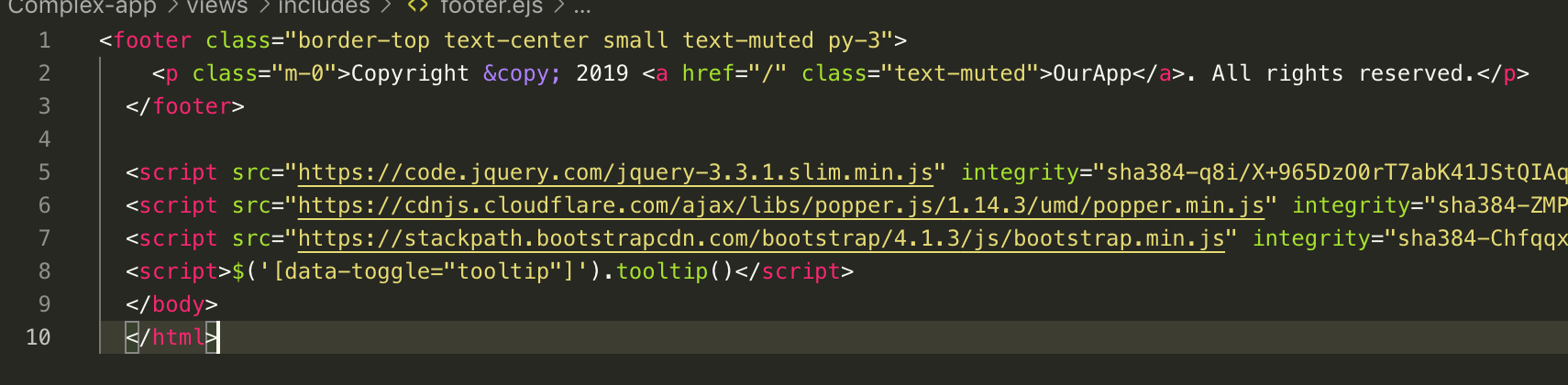
At home-dashboard and use user.username and at header.ejs user.avatar. it doesn’t need to repetitively pass the req.session.user data for every template

Remove more duplication, home-guest.ejs, copy the <form> and remove the header and use the default <%- include()’includes/header’) %>

<% %> is an ejs in and out dynamic mode

At header.js. this will make the log in form always available if not log in and make the options available for users that are logged in.

  
for footer, create the footer.ejs



And call it in every html page.



67. Post model 1

Create the title and body of post in db.

Form should submit to

At create-post, replace the # to create-post

At router.js, in post related routes.



Create the object .create in postController.js

Create the model of the .create. create new Post.js in models directory.

Require the model in postController const Post = require(‘../models/Post’)

Let post create a new object from the model let post = new Post()

68. Post Model 2

Addition of author in the mongoDB

69. Viewing a post 1

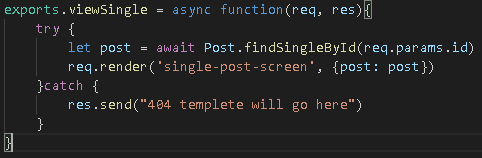
Create single-post-screen.ejs and add the code. Add the header and footer from includes

At router.js 

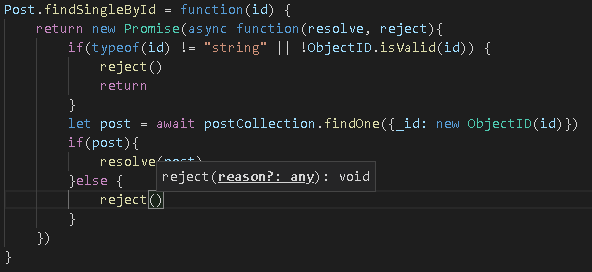
Then at header.js, add the style by putting “/” in main.css to fix the formatting.

70. Viewing a post 2

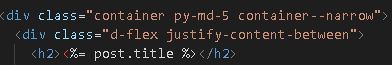
At postController, .findSingleById = any name

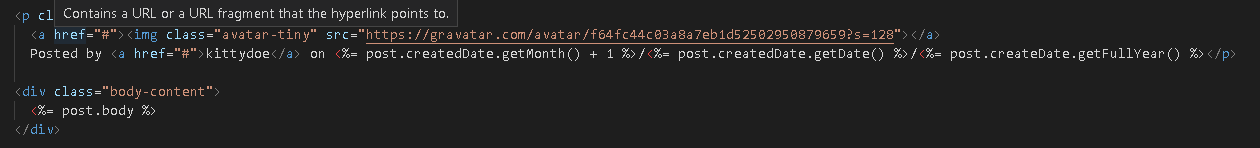


At Post.js



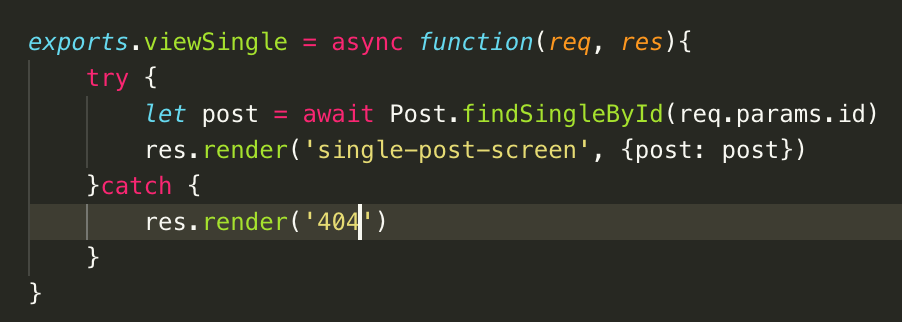
At single-post-screen.ejs = <%- post.title %>





71. Performing a lookup in MongoDB Part 1

Setup the 404 template. Get the 404 template. Html. Create the 404.ejs. Linked in postController.



Extract the username and avatar from mongoDB using lookup.

At post.js,

.aggregate is use for performing complex or multiple operations.

$match and $lookup are the operations of the multi operation function of aggregate.

$match – will look first for the match with reference to \_id

$lookup – define the items to look up

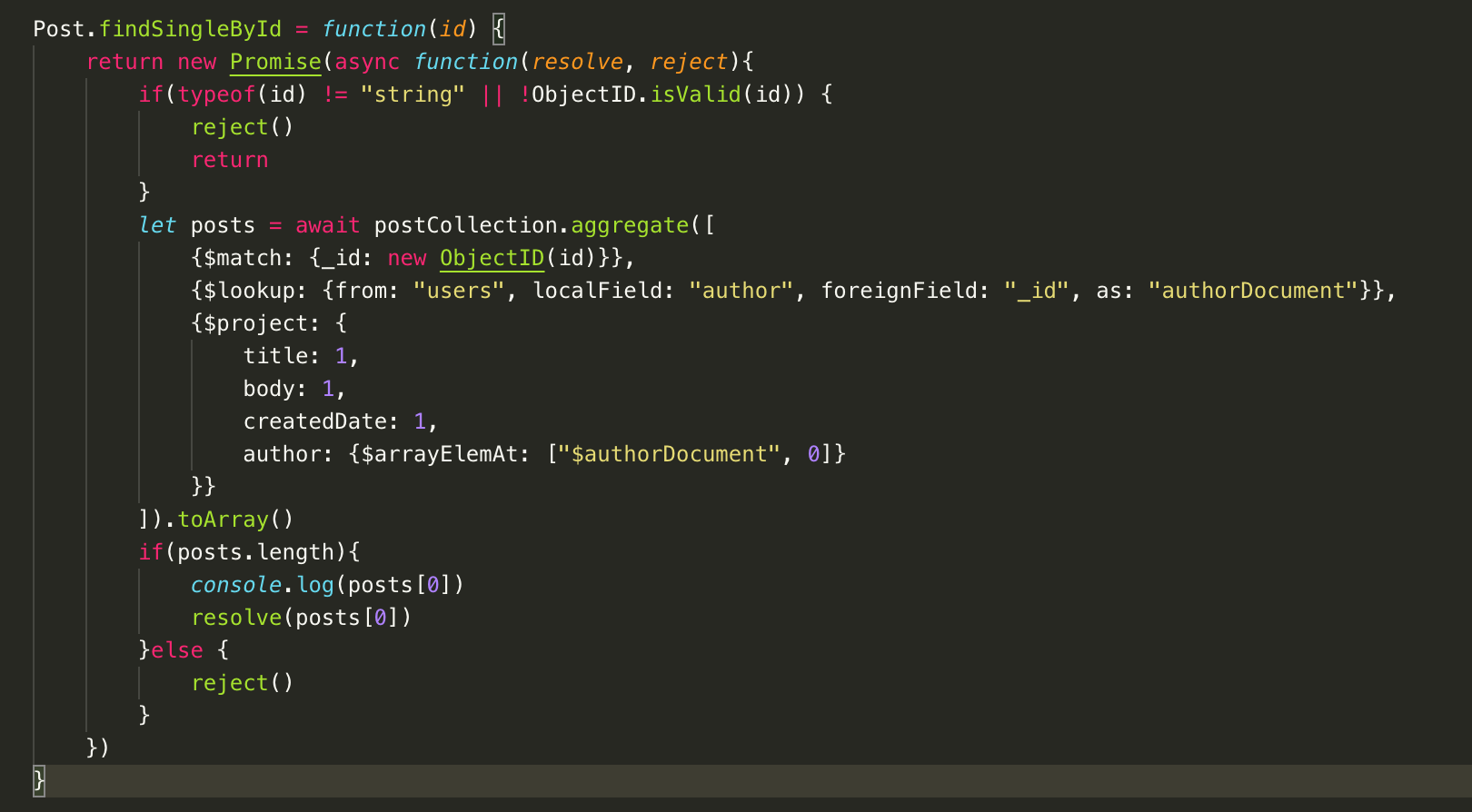
posts[0] - used to return the first value of the array

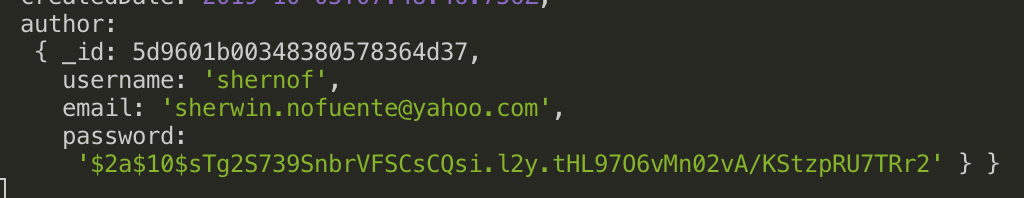


From above it can now extract multiple information from mongoDB as below. That can be use to display in html template.



Use another aggregate operator named project. Project defines what we want to display out from the information extracted above.





‘author’ is now an object that can be use to extract the name and email for avatar.

.map creates new array

72. Performing a lookup in mongoDB Part 2



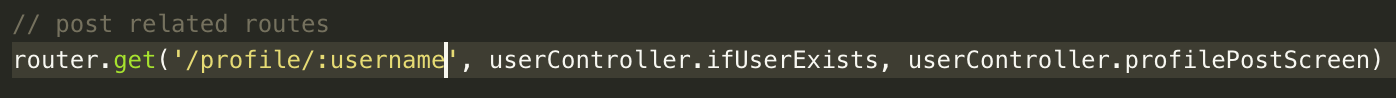
Here we can get now the avatar and username:

At single-post-screen.ejs

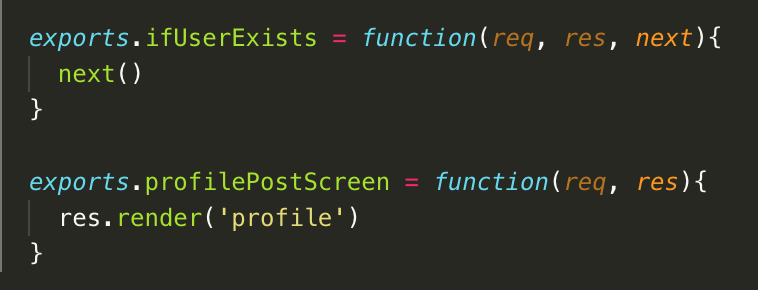
73. User Profile Screen

Create profile.ejs and paste the raw html file.

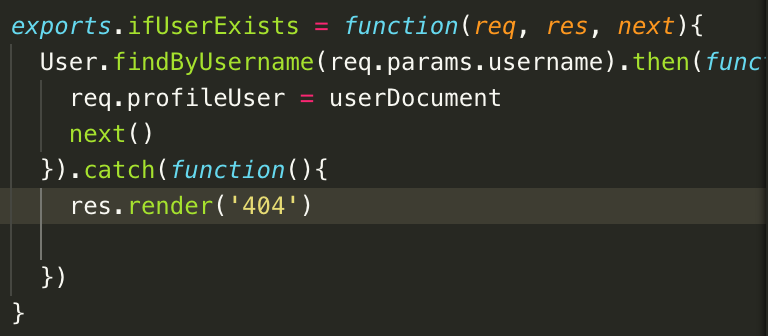
Configure the router.js to route the profile.ejs. create the .ifUserExists and .profilePostScreen.



At userController.js, create the 2 functions above. This will display the user profile.



.ifUserExists:



At User.js, create .findUsername



For exports.profilePostScreen;



Make the username and avatar in the profile.ejs dynamic:



74. View posts by author

Continue to work on the exports.profilePostScreen…

At Post.js, work on the Post.findbyAuthorId,

75. Is the current visitor the owner of the post?

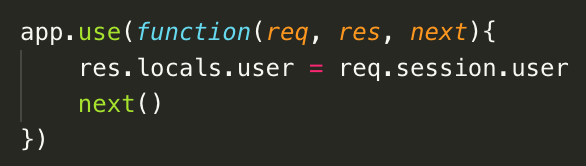
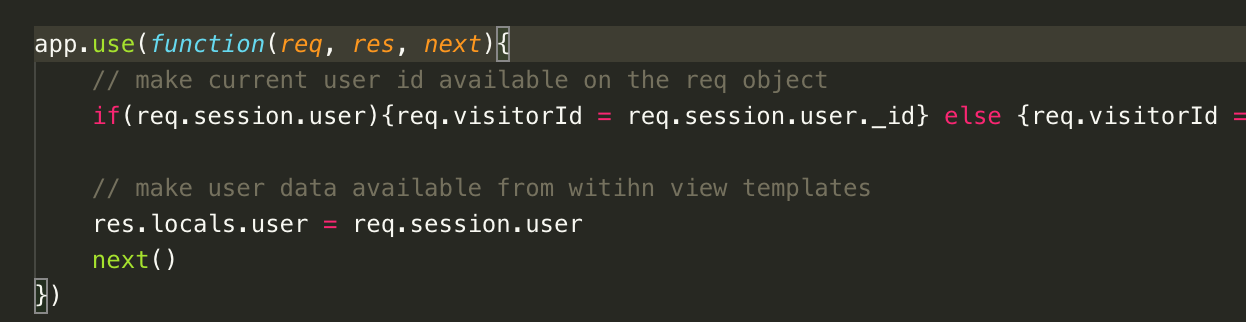
At header.ejs, at “My Profile” >>



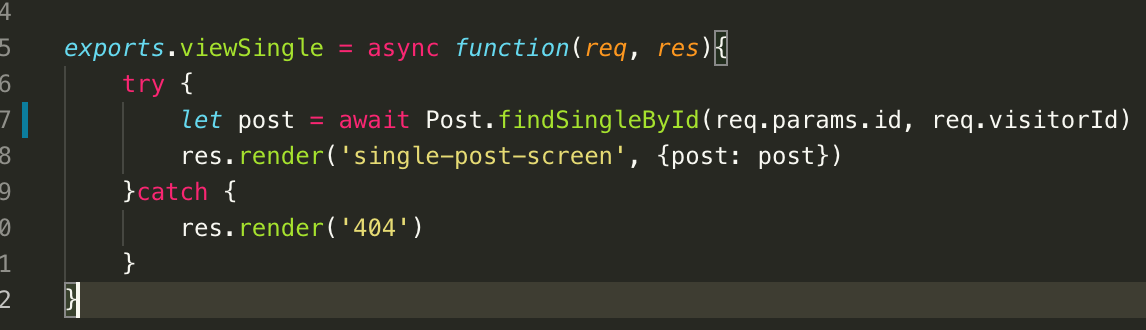
This will set the profile page when the profile picture is clicked.

Next step is to set the author only be able to delete and edit the post.

At app.js,

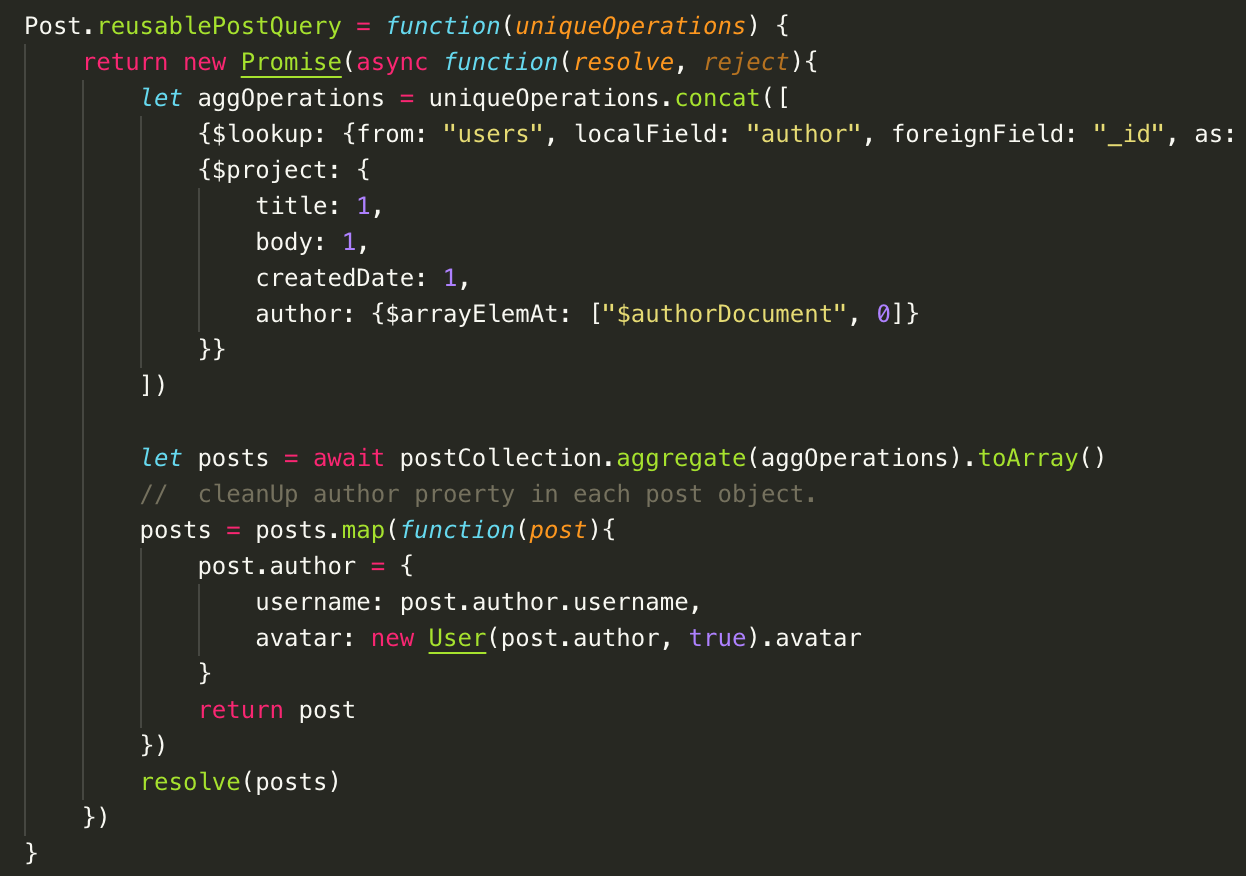
At postController.js,

At Post.js,

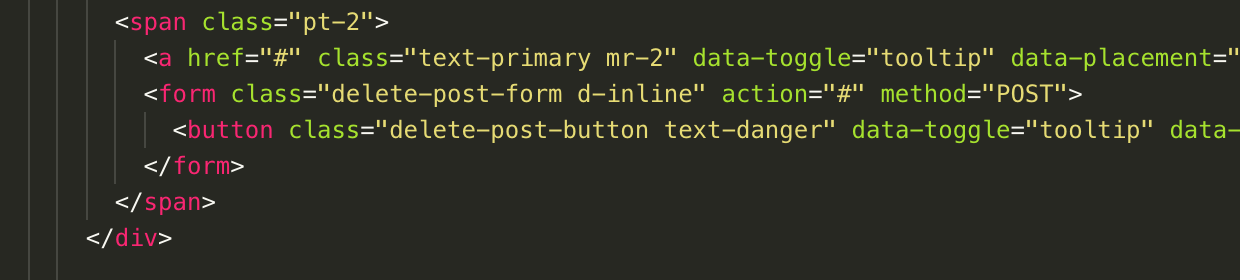


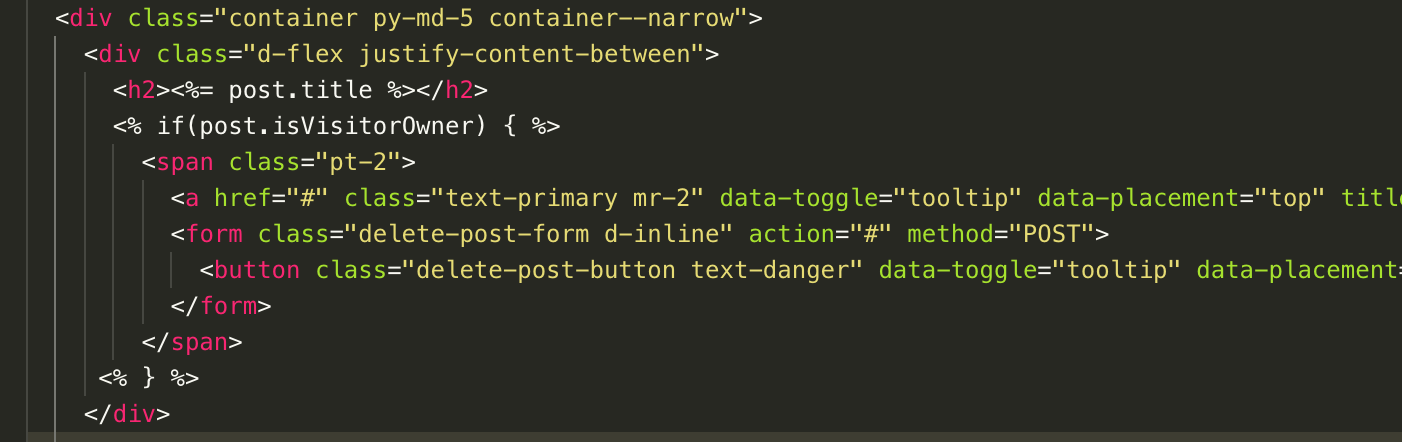






At single-post-screen.ejs, cut the span tag





76. The edit screen for the post

77. Updating posts in database 1

78. Updating posts in database 2

79. Miscellaneous improvements

80. Markdown: Safe User Generated HTML

81. Make this quick edit to your code

82. Let users delete a post