Full Stack Web Development Udemy course - App: Feedback collection application

2 Server side:

2-9: Differences between express and node:

Node: JS runtime used to execute code outside of the browser

Express: Library that runs in the Node runtime. Has helpers to make dealing with HTTP traffic easier

2-10:

commonJS modules syntax: const express = require(‘express’);

ES2015+ modules syntax: import express from ‘express’;

In front end (react), it is much easier to get access ES2015+ modules (included in creat-react-app)

In server side, we always just use commonJS modules. So use require instead of import.

2-11: Route handler:

Timeline

Description automatically generated

2-13 Deploy the app using Heroku:

2-13-1:

When Heroku runs our app, it can injest environment variables to process. (runtime configuration)

const PORT = process.env.PORT || 5000;

//5000 is the default one if in non-prod environment.

2-13-2: Using engines in package.json to specify the versions of the dependencies / packages when the code is running on prod servers.

"engines": {

"node": "14.9.0",

"npm": "6.14.8"

},

2-13-3: Heroku will find the script named as “start” in package.json to start execution.

2-13-4: .gitignore should be created so some changes commited will not trigger a new deployment. Also all the dependencies should not get commited.

2-14: We need to install Heroku CLI:

Using brew to install and please follow the command in the website.

After installation:

heroku login

heroku create

git remote add heroku [remoteGitAddressProvidedByCreateCommand]

git add .

git commit -m “initial code”

git push heroku master

heroku open // Then the app will start to run

//you can juse heroku logs –tail to see logs

3 Google OAuth:

3-18:

Diagram

Description automatically generated

3-19: PassportJS to create google OAuth flow, it is a library to finish authentication process.

Two libraries:

passport library: General helpers for handling auth in express apps.

passport strategy library: Helper for authenticating with one very specific method (email/password, Google, Facebook, etc.)

passport.use()

3-24: clientID and clientSecret

clientID: public token – we can share this with the public

clientSecret: private token – we do not want anyone to see it.

How to hide the clientSecret in code :

Diagram

Description automatically generated

3-25: Server side code please follow commonJS style:

module.exports = {

googleClientID: '458626602921-t2n3tb6gjp62ka47fmdehol7tgjeavsg.apps.googleusercontent.com',

googleClientSecret: 'PS\_ITcPRUT2tltZk46cp5qS5'

}

3-27: callbackURL (redirect URL) should be restricted in google cloud setting to avoid hackers.

Code up to now:

const express = require('express');

const passport = require('passport');

const GoogleStrategy = require('passport-google-oauth20').Strategy;

const keys = require('./config/keys.js');

const app = express();

passport.use(new GoogleStrategy({ //generic register

clientID: keys.googleClientID,

clientSecret: keys.googleClientSecret,

callbackURL: '/auth/google/callback'

}, (accessToken, refreshToken, profile, done) => {

console.log('accessToken: ', accessToken);

console.log('refreshToken: ', refreshToken);

console.log('profile: ', profile);

console.log('done: ', done);

}));

app.get('/', (req, res) => {

res.send({ hi: 'there' });

});

app.get('/auth/google', passport.authenticate('google', { //here google means google strategy

scope: ['profile', 'email']

}));

app.get('/auth/google/callback', passport.authenticate('google')); //here passport can see the query string code in URL so it knows what is the next step: turn the code to an user profile

const PORT = process.env.PORT || 5000;

app.listen(PORT);

3-28 As the code above show, passport is set to use google Stratety, in first around calling authenticate “google”, it askes google to grant permission. After google side is done and callback is happening, it will process the code to get accessToken and profile for the next step, which is returned in the callback function. In general, passport works as an agent for us to talk with google auth.

3-29 nodemon to auto-restart server after changes:

Change node 🡪 nodemon in a script and run script with npm run <scriptName>

4 Adding MongoDB

4-31 express refactor:

Sample express structure:

Diagram

Description automatically generated

Considering passport.js is an service (agent) for google auth, put it into services folder.

4-31-1: for passport module, after refractor, we only want it to run. We do not need it to return any class or function. So in the index.js we just need to write:

require("./services/passport"); //Just make sure it is running

However, in the authRoutes.js, we require the passport module again. It is the same passport module in the system so the original “use” generic register still works.

4-31-2: authRoute needs app from index.js, so we can return a function with an app argument expected from index.js. The return function can help us finish routes definition.

index.js:

require('./routes/authRoutes')(app);

authRoutes.js:

const passport = require('passport');

module.exports = app => {

app.get('/', (req, res) => {

res.send({ hi: 'there' });

});

app.get('/auth/google', passport.authenticate('google', { //here google means google strategy

scope: ['profile', 'email']

}));

app.get('/auth/google/callback', passport.authenticate('google')); //here passport can see the query string code in URL so it knows what is the next step: turn the code to an user profile

};

4-32: authentication

http is stateless. It needs cookie, token …

We are using cookie based authentication.

Once server confirms the user is logged in, it will send back to browser with a header: set-cookie property with a value. Once the browser gets it, it will automatically get saved in browser and will be sent with following up requests.

4-33: Choose some consistant value in profile to identify user for our app (when they sign out and sign in using oauth again we can still identify whether it is the same person): using id in google profile: it is a unique identifying token.

Diagram

Description automatically generated

4-34: MongoDB:

In express, it uses mongoose.js library to talk with MongoDB. A modal class is used to get access to a single collection in mongoDB.

It has so many collections, each of which includes a lot of records. (The schema of each records can be totally different)

Diagram

Description automatically generated

4-38: How to manage your remote mongoDB: using atlas

npm install mongoose

const mongoose = require('mongoose');

mongoose.connect(keys.mongoURI, {

useNewUrlParser: true,

useCreateIndex: true,

useUnifiedTopology: true,

});

4-40: modal class in mongoose code maps to the “collections” in real DB. We can create a modal to represent to create a new collection and add all the properties. Finally it can be sent to mongoDB to execute based on the modal class.

4-40-1:

const mongoose = require('mongoose');

const { Schema } = mongoose;

const userSchema = new Schema({

googleId: String

});

mongoose.model('users', userSchema);

Although the records in the mongo DB can have different properties, mongoose wants to know all the different properties that our records have in the DB, and it requires us to define all those ahead of time with the schema object.

The modal statement above will not modify any existing collection if it already exists. Also, it does not run yet, it is just creating a modal class here.

4-40-2:

The model classes under the modals folder should not be multiple required by different other files. The reason is we do not want such statement: mongoose.model(‘users’, userSchema); //Create collection to execute so many times. Once there is a “require”, it will be executed for one time.

Remember all the module with same name under the project is the same module, so once you do some process for the module in js file A, and when you require the same module name in file B, it is the same one and A’s operation on this module will reflect on B as well.

4-41:

Load schema to a modal class (to create this modal class) and push to mongoose:

mongoose.model('users', userSchema);

pull the modal class out of mongoose:

const User = mongoose.modal('users');

Later on we can use modal class to create modal instances, then save them (persist them) to DB.

new User({googleId: profile.id}).save();

4-43: mongoose queries:

All the queries between mongoose and mongoDB are async actions, which returns promise.

passport.use(new GoogleStrategy({ //generic register

clientID: keys.googleClientID,

clientSecret: keys.googleClientSecret,

callbackURL: '/auth/google/callback'

}, (accessToken, refreshToken, profile, done) => {

User.findOne({ googleId: profile.id })

.then((existingUser) => {

if(!existingUser) {

new User({googleId: profile.id}).save();

}

})

}));

4-44:

done function in the callback function’s argument should be called once operations in callback is done. Thus it tells passport that we have finished creating a user and that it should now resume the auth process.

User.findOne({ googleId: profile.id })

.then((existingUser) => {

if(existingUser) {

done(null, existingUser);

}

else {

new User({googleId: profile.id}).save()

.then( user => {

done(null, user)

});

}

});

In the code above, new User is one modal instance, the user argument after then( is also a modal instance. These two modal instances are representing the same records in the collection. But for further processing we always use the one in the callback as it will have some changes while user is saved.

4-45-1: serialization:

passport.serializeUser((user, done) => {

done(null, user.id); //Here the user means the user in User collection and id is the db id for records.

});

user.id is a short cut for the \_id: $old” under the current collection

4-45-2: Why we use db ID in cookie:

Diagram

Description automatically generated

4-46: deserialization:

passport.deserializeUser((id, done) => {

User.findById(id)

.then(user => {

done(null, user);

});

});

Note: code above giving a function to serializeUser and deserializeUser is to define these two functions instead of executing it. Serialization is definitely one part of the passport. By giving the function to serializeUser and deserializeUser, you tell the passport “how to do it”.

4-47: cookie-session node module is to manage cookies in the application

In the cookieSession configuration, maxAge is counted by 1ms, keys is an array and you can have multiple keys and cookieSession will randomly use one of them.

4-48:

Diagram

Description automatically generated

So after you set the cookie-session and passport all configurations. Passport will help finish the process above automatically. So once a req comes from the route, it will go through the process above. After user signed in from google, cookie info will be addressed to all req process.

app.get('/api/current\_user', (req, res) => {

res.send(req.user);

});

4-49: Similarity to 4-8, after passport is properly configured, it will associated a (logout()) function to any request coming to trigger a logout function, which will kill the cookie .

app.get('/api/logout', (req, res) => {

req.logout();

});

4-50-1: middleware:

app.use means it is going through a middleware. Middleware is to do some pre-processing of the upcoming request before it is sent to different route handlers.

Diagram

Description automatically generated

4-50-2: How cookie-session works:

After middleware, req.session will be attached to the req object. It is the process that cookie-session extract the cookie and attach to req.

Then, passport is looking at req.session and get what it needs to do deserialization.

4-50-3: differences between cookie-session and express-session

Diagram

Description automatically generated

Cookie-session stores all the data directly in the cookie.

Express-session stores all the data in a remote server-side data store (session store) and whenever request comes, it takes the id from that session and go to pull the data of the session store.

4-50-4: From the network call in the response header you can see

“Set-Cookie” when the cookie content is changed. (server want the browser to update its cookie).

Then under the request Header you will see “Cookie”, which is to send to server to tell server “who I am”.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Session=”XXX” is the encrypt session value.

Section 5: Environment

5-51: It is better the dev and prod environment have two sets of keys for mongoDB, google API, CookieKey .etc.

5-52: prod db connection:

mongodb+srv://dreamfancy1987:webstock1028@emaily-prod.2dqii.mongodb.net/myFirstDatabase?retryWrites=true&w=majority

5-54:

By convention, when you get properties from environment, the property in env should be defined with uppercase letter with underscore:

prod.js:

module.exports = {

googleClientID: process.env.GOOGLE\_CLIENT\_ID,

googleClientSecret: process.env.GOOGLE\_CLIENT\_SECRET;

mongoURI: process.env.MONGO\_URI,

cookieKey: process.env.COOKIE\_KEY

};

Then go to heroku.com you can add all the environment variables.

When you commit and push the code, heroku also shows some of the production env variables:

remote: -----> Creating runtime environment

remote:

remote: NPM\_CONFIG\_LOGLEVEL=error

remote: NODE\_VERBOSE=false

remote: NODE\_ENV=production

remote: NODE\_MODULES\_CACHE=true

5-57: fix proxy issue: http vs https

Two reason for the https request becomes http when callback happens

5-57-1: in the callbackURL when configuring GoogleStrategy in passport it is using relative path so passport can decide it is either http or https

5-57-2: After the code is launched at server, heroku proxy is responsible to receive request and distribute to your sever. After receiving it, heroku proxy thinks all the following request are inside the heroku internal network so they will assign it as http. (your server will also think heroku proxy is trustable so give a call back with http)

Solution:

5-57-3: Configure the GoogleStrategy to trust all the proxies between browser and server.

5-57-4: In the configuration of GoogleStrategy callbackURL, give absolute path.

passport.use(new GoogleStrategy({ //generic register

clientID: keys.googleClientID,

clientSecret: keys.googleClientSecret,

callbackURL: '/auth/google/callback',

proxy: true //GoogleStratey will trust all the proxies so http redirect will also be ok.

},

……

6-59:

While using create-react-app to generate react app, as this node module is only used for one time, so please use npx create-react-app client

6-60: Attention while generating the react project, it should be inside the server repo But this client side react project also has its own built in server.

Diagram

Description automatically generated

Two servers one for data only, one for front end using create-react-app (pre-build for babel and webpack)

6-61: Running sever and client side at the same time:

Server directory has one package.json and client directory inside server directory also has one package.json.

6-61-1: Add a client command under server to run client start. You need to add prefix to tell the command please run under the client directory:

"client": "npm run start --prefix client"

6-61-2: Then you can use concurrently command with multiple command included in “” to run both server side and client side concurrently.

npm install concurrently

"dev": "concurrently \"npm run server\" \"npm run client\""

6-62: Add proxy to the client-side package.json file:

create-react-app will do it automatically after we add a special file to hold the proxy code:

using [http-proxy-middleware@2.0.1](mailto:http-proxy-middleware@2.0.1)

6-63 proxy:

We use proxy to make client side can reach to server side (vice versa) easily we do not need to worry about base url changes:

Add “createProxyMiddleware” for the client side,

Add a file in src of client side calling: setupProxy.js (must be this name), then with the following code:

const { createProxyMiddleware } = require("http-proxy-middleware");

module.exports = function (app) {

app.use(

["/api", "/auth/google"],

createProxyMiddleware({

target: "http://localhost:5000",

})

);

};

6-63-1: By setting “setupProxy.js” in client side, server can be reached by “target” with some routes.

6-63-2: When changing package.json you need to manually kill the process and start again. The automatic reload will not happen if only package.json is changed.

6-64: Why we have proxy setting in react:

Mostly it is for dev mode. Because in production, create-react-app will not exist. A bundle.js is the one which is to be deployed and it is on server directly. (by running npm run build under client you will get this file)

Diagram

Description automatically generated

6-65: reasons for hosting API and front end request to the same end point domain instead of different one:

6-65-1: cookie in browser is to the same domain and port. If different, a lot of extra settings are needed.

6-65-2: CORS issues:

Browser is on localhost: 3000, req to localhost:5000 is a CORS (cross origin resource sharing) request !

6-65-3: Digram on how google auth works while proxy is involved.

Diagram

Description automatically generated

Some points:

While req sits pending: it means the proxy will tell the browser to hold on. That is the reason browser is not going to time out.

Diagram

Description automatically generated

7 Client side

7-66 Async Await syntax:

7-66-1: When the return of the promise is a json, we need to call .json() to wait itself to be resolved. So the code will look as:

Graphical user interface, text, application

Description automatically generated

A picture containing table

Description automatically generated

7-66-2: For the code above, to write an ES2017 syntax, it will be like:

Text

Description automatically generated with medium confidence

7-66-3: We can use a return key word to avoid the else function in if statement. For example,

if(existingUser) {

done(null, existingUser);

}

else {

const user = await new User({googleId: profile.id}).save()

done(null, user);

}

Can be changed to

if(existingUser) {

return done(null, existingUser);

}

const user = await new User({googleId: profile.id}).save()

done(null, user);

7-72:

Sometimes when you make big changes on dependencies, you need to delete the package-lock.json and npm install again

7-73: How to create store in index.js:

import { Provider } from 'react-redux';

import { createStore, applyMiddleware } from 'redux';

const store = createStore(() => [], {}, applyMiddleware());

The first argument is a dummy reducer.

The second argument is related server-side-rendering.

7-75:

In the combineReducer function: The key of each reducer will be the state name.

import { combineReducers } from "redux";

import authReducer from './authReducer';

export default combineReducers({

auth: authReducer

});

7-78:

Attention <BrowserRouter> can only has one direct child.

7-81 use materializecss for CSS:

We should find a CSS library only specifying CSS using className. We should not use CSS library such as material-ui.com. Its styling is generated by javascript and it is hard to change.

So we decide to use materializecss.

7-82-1: webpack (included in create-react-app): module loader, finally helps several js file in the framework to a single file (bundle.js).

7-82-2: When we trying to import the materialize.min.css (a special version), we need to go to node modules and go to the dist folder to find the one you really want and specify it. (it is not a general import)

Webpack will assume you want to find an npm module if you do not use relative path:

//import materializeCSS from 'materialize/css/dist/css/materialize.min.css’;

Here we only want the contents of materialize.min.css but the general return is not useful, so we can write it as:

import ‘materialize-css/dist/css/materialize.min.css’;

7-84-1: Mostly the highest level of the root should have a container class, which can help define the boundary of the page.

7-84-2: How to decide if user is logged in. When react is booted in the browser, it will get the api /api/current\_user in the server to get the current user status.

7-85: proxy rule:

Add setup.Proxy.js with some path redirect. Then when react make get /post request using axios, all the paths will be examined by this file and do the redirect if necessary.

const { createProxyMiddleware } = require("http-proxy-middleware");

module.exports = function (app) {

app.use(

["/api", "/auth/google"],

createProxyMiddleware({

target: "http://localhost:5000",

})

);

};

7-86: How redux-thunk works:

While the action creator returns a function with argument dispatch inside, dispatch can happen after async function. We need to use this way because action creator can only return “plain object” back

7-88: While I am debugging it says port 5000 is already in use. SI need to find which process is using port 5000, then it

sudo lsof -i :3000

kill -9 {pid}

7-90-1: There can be more than two states (three states in this case) for whether you are logged in or not.

Null: still making request, do not know

User modal: with userID

False: user is not logged in.

7-90-2: connect function does not need to have mapStateToProps and actionCreator called at the same time. It can only have actionCreator: such as App.js: As long as it is loaded it will fetch the user although App.js does not need that state. However, Header.js needs the states, but it does not need to have actionCreator called, because when App.js is called, it is already loaded to redux.

7-91: To test what will happen when internet is slow, you can “throttling the internet connection”

Select network tab,then in the list below you can select whether to throttling and to what level you are throttling.

7-92: res.redirect can be called as a backend call response then the front end will also do the redirect by using the proxy set up.

app.get('/auth/google/callback',

passport.authenticate('google'),

(req, res) => {

res.redirect('/surveys');

}

);

7-94: How to log out:

Table

Description automatically generated

7-95: import statement should be on the top of the file. That is a mandatory for all JS files.

7-96: Differences between Link and a:

Link: Navigate to a different route rendered by React Router

a: Navigate to a completely different HTML document

8 Client Side Payment:

8-98: stripe for payment:

Two types of payment:

Pay for credit (simple, easy)

Sass Payment: Monthly Payment with migration in the middle ---really hard and expensive, do not use it. (recurly.com )

Publishable key: (used on front end)

pk\_test\_51JGXAECoOuahh7lf7jikGsFnB11wk0s9QUeUzSnTEIy0CiQ9tmi3MTJVNPCoY55Sn5kwHWlhjCWzXSkDbs92ZTeX009JPS5vRl

Secret Key: (used on backend)

sk\_test\_51JGXAECoOuahh7lfn1N4FMocBFcgV9vX3NhxBNMEJJLPack8VHjX5GtPjMPZGaEony1y8TXjK95JLrNrYW4L8kQf00UCGLFgiX

8-101: We want the react code importing files are all inside the client folder and we want to config for backend does not imported by react (keep separate), because --- as long as any files are imported by front end, it is in the browser and it is open for public to see. So we would rather have separate configurations for front end and back end keys.

Another reason is : front end we use ES2015+ modules and back end we use commonJS.

CommonJS can have some if else statement before it decides what to exports. But import does not allow that.

In other words, export or import with logic is only allowed in commonJS as below:

Graphical user interface, text

Description automatically generated

8-102: create-react-app adding custom environment variables

Approach 1: Temporary environment variables:

REACT\_APP\_SECRET\_CODE=abcdef npm start

Then in the code you can refer to it as process.env.REACT\_APP\_SECRET\_CODE

Approach 2: add them to .env in the root of the project (please note it is root of the project, not root of the source). React will go to root and find any files starting with .env to assign variables. We can have multiple .env files defined, for example, .env: default one; .env.local; .env.development .etc

Then put the file name on the right of the command, for example

npm start: .env.development

The naming of the file will decide when it is to be run.

* npm start: .env.development.local, .env.local, .env.development, .env
* npm run build: .env.production.local, .env.local, .env.production, .env
* npm test: .env.test.local, .env.test, .env (note .env.local is missing)

When create-react-app build the project, it will inject those env variables according to the table above.

While debugging and you can check environment by: process.env.NODE\_ENV

Note: For backend we are making environement files as js file as it can do the conditioning require (import). However, in front end as we cannot do conditional import, we just make them as plain file. For the plain file, no ‘’ nor space should be added. The plain format is A=B

8-103: react-stripe-checkout is a “button” component. Once it is loaded and user clicks it, a following form then come.

StripeCheckout component can have so many props and you need to see doc to identify them

amount: how much much you want to send by cents.

token: it is a callback function after an authorization token is retrieved from stripe API.

8-104-1: In the header, Under ui there is a function to return what should show under the ul. It can be one li or several li or none li. When return is multiple li, you should try to use the following format:

return [

<li><Payments /></li>,

<li><a href="/api/logout">Sign out</a></li>

];

8-104-2:

You can add debugger; in react code and when the code run to there in browser and it will stop.

8-105-1: the onToken will give an Id back representing the pending charge. Then our app will use it in the serverside to continue finishing the transaction. Also some card info is also back, except the card number. (but the last 4 digits can be coming back for us to do verification)

A picture containing text

Description automatically generated

So as long as we handle the token call back we are good on our side.

8-106-1: Every li under ul needs a key attribute

8-106-2: By adding a child <button> component for <StripeCheckout> It will take the place of the default button design for the button. So you can define the button yourself in the child component.

8-106-3: When you want to make a request to the backend server, it should be inside the action creator.

8-106-4: What is the action creator looks like when there is an argument: Put argument in the outer function and dispatch in the inner function. Here is the definition of the functions so the inner one can see the argument of the outer one as well.

8-106-5: It is possible two actionCreator dispatch the same action.type but payload may be different.

export const fetchUser = () => async (dispatch) => {

const res = await axios.get('/api/current\_user');

dispatch({ type: FETCH\_USER, payload: res});

}

export const handleToken = token => async dispatch => {

const res = await axios.post('/api/stripe', token);

console.log(res);

dispatch({ type: FETCH\_USER, payload:res.data });

}

But you cannot combine them because it may also have other actions added later.

8-107: A common mistake!

When you want to call an action creator in the component, you cannot call the imported one directly. Instead, you should call as this.props.actioncreatorname. Because only after it is using “connect” to the component, it can be attached with redux.

8-109-1: To find all the libraries to your project (stripe), you can go to npmjs.com to search with key word stripe, then all the possible libraries will show, even some of them are not on the stripe website.

When starting looking at the doc, you need to search the full doc. The only parts on npmjs does not have what we want. You need to click the link under documentation to the specific website to search all docs.

8-109-2: For this case, we see a “charges” section under the core resources of API. We can go through this part. Attention what is “what we need to send to stripe” and “what they will respond”.

8-110: body-parser middleware:

Express does not parse the payload of the request by default, so you need a middleware. After body-parser is used, then the body is available under req.body. It will parse the stringified request to json style.

// Express v4.16.0 and higher

// --------------------------

const express = require('express');

app.use(express.json());

app.use(express.urlencoded({

extended: true

}));

// For Express version less than 4.16.0

// ------------------------------------

const bodyParser = require('body-parser');

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({

extended: true

}));

8-111-1: When debugging you may see a request to backend status showing pending in the network tab, it means the response is not handled yet.

Table

Description automatically generated

8-111-2: In the User modal we define the user Schema with credit as type number and default value 0.

const userSchema = new Schema({

googleId: String,

name: String,

credits: { type: Number, default: 0 }

});

Then we need to see when the user is initialized in server side / react side:

8-111-3: When user try to log in, passport will do the authentication and the user info will be retrieved (or created if a new user) with the user object, including all the schemas. So default value of credit 0 is also created.

Note: User means the user modal mapping DB. Whenever do User.find() is to query the DB. Whenever new User().save is to create a new user in DB. The return value user is an user modal object presenting in express side.

Whenever a request comes, passport middleware will check the cookie then decide if to attach .user to the req based on cookie value (if the same user)

const User = mongoose.model('users'); //pull the modal class from mongoose

…

async (accessToken, refreshToken, profile, done) => {

const existingUser = await User.findOne({ googleId: profile.id });

if(existingUser) {

return done(null, existingUser);

}

const user = await new User({googleId: profile.id}).save()

done(null, user);

}));

…

app.post('/api/stripe', async (req, res) => {

console.log(req.body);

const charge = await stripe.charges.create({

amount: 500, //This is not getting from front end, but a confirming

currency: 'usd',

source: req.body.id,

});

req.user.credit += 5;

const user = await req.user.save(); //updated userModal

res.send(user);

attention to see the res.send above you still need to see from network tab as it will not show on the browser or console automatically without processing.

8-114:

How to send error with status back:

return res.status(401).send({ error: 'You must log in !' }); //Unauthorized. Here return is ok just to end the execution

8-115: How to define middleware yourself: Example: make the route specific middlewares:

One example middleware:

module.exports = (req, res, next) => {

if(!req.user) {

return res.status(401).send({ error: "You must log in"}); //No next because res sent error

}

next();

};

Wire the middleware to where you want only: you can put to specific request handler: Add the middleware as the second argument of the handler

const requireLogin = require('../middlewares/requireLogin');

module.exports = app => {

app.post('/api/stripe', requireLogin, async (req, res) => {

…

Actually there can be as many as middlewares in the route handler function. Even for (req, res) => {} is also one but it is just the last one so no next function as the third argument of that function.

9 How production look like for routing

9-118: In local, create-react-app is to “build” the app running and serve the local server. In production, we do not have create-react-app. So by running npm run build, all the files which are going to serve production are under the build folder. Under the build/static you can see compiled js files under the js folder and compiled css file under css folder. The entry point is still the index.html

9-119: How routes should be defined in express index.js:

require('./routes/authRoutes')(app);

require('./routes/billingRoutes')(app);

if(process.env.NODE\_ENV === 'production') {

//Express will serve up production assets, including main.js, main.css

app.use(express.static('client/build'));

//Express will serve up index.html if it does not recognize any routes above, including express route and public assets above. Giving index.html back is to assume react router knows it

const path = require('path');

app.get('\*', (req, res) => {

res.send(path.resolve(\_\_dirname, 'client', 'build', 'index.html'));

});

}

First, it will go through express defined routes to see if any match.

If not, check the production assets, go to the public assets.

If still not, then serve the html back, assuming the react router will know it.

9-120: How to do deployment:

Build folder is not intentionally to be commited to github.

Diagram

Description automatically generated

The problem of option #2 is you need to install a lot of developer dependencies on production, such as webpack .etc.

CI: CI server is to run test and install the client dependencies, build the project, commit the changes to another branch (a deployment branch on git), push that build to heroku.

Go to circleCI.com to check more.

We can set a heroku-postbuild job (postbuild here means post server build) to finish the process above.

Diagram

Description automatically generated

9-121: Make a script named as postinstall in the package.json. Then heroku (deployment provider) will run it after installing all the server side modules (deps). You can also use heroku-prebuild, or heroku-afterbuild as well

You can run some commands such as bower or grunt (bower: client js management system; grunt: task management)

Differences of the scripts:

postinstall: Will run after the dependencies installed, even on your local

heroku-prebuild: Run before heroku install the dependencies, only on prod.

heroku-postbuild: Run after heroku install the dependencies, only on prod.

"heroku-postbuild": "NPM\_CONFIG\_PRODUCTION=false npm install --prefix client && npm run build --prefix client"

Please remember the setting above is to have react running inside the node server in production !!!

Explanation on the command above:

After server side dependencies are automatically added by heroku, heroku does not know there is a client folder and there is another package.json inside the client folder. And the build for the client needs some client side dev dependencies, so first please add that variable at the top to declare please install all client side dependencies then run the build.

Different commands are separated by &&. You need to specify to run the command in sub client folder by adding –prefix client.

9-121: The heroku-postbuild will run after the project build (majorly server side) done

remote: -----> Build

remote: Running heroku-postbuild

remote:

remote: > server@1.0.0 heroku-postbuild /tmp/build\_2a63499d

remote: > NPM\_CONFIG\_PRODUCTION=false npm install --prefix client && npm run build --prefix client

Some key points during the production deployment debug:

9-121-1: If it is working in local but not prod (everything compiled well), check environment variables setting for production only (on heroku website). Also because of this you cannot remove the repository randomly as these settings can be missed.

9-121-2: When you try to change URL, please update the whitelist on some authentication service such as google oath.

9-122-1: Whenever you see build issues and you want to reinstall all the dependencies to avoid there is a mismatch, you need to delete the package-lock.json file on both client and server

9-122-2:

remote: SyntaxError: Unexpected token ...

or something related to the @hapi module, this typically means that there is a mismatch between the Node version used locally and the version Heroku is attempting to use

9-123 Debugging experience!!!

9-123-1:

When there is argument --- in the command, you need to use “run” key word. Instead of writing npm start –prefix client, you need to write as npm run start –prefix client.

9-123-2:

About the code below:

if(process.env.NODE\_ENV === 'production') {

const path = require('path');

//Express will serve up production assets, including main.js, main.css

app.use(express.static(path.resolve(\_\_dirname, 'client', 'build')));

//Express will serve up index.html if it does not recognize any routes above, including express route and static react routes

app.get('\*', (req, res) => {

console.log(path.resolve(\_\_dirname,'client', 'build', 'index.html'));

res.sendFile(path.resolve(\_\_dirname,'client', 'build', 'index.html'));

});

}

9-123-2-1: express.static() returns a function instead of a path. So you cannot use path to take place of it.

9-123-2-2: res.sendFile() is not res.send() please attention on the differences.

9-123-2-3: To know the absolute path in production, we can use path.resolve(\_\_dirname, ‘client’, ‘build’, ‘index.html’) to allocate the file. Note dirname will bring you to the root of the project.

10-127 & 10-128: For the schema of recipients of the email, it is considered as an array of Strings:

Using survey as a (document) collection:

Diagram

Description automatically generated

Use survey as a subdocument collection:

Diagram

Description automatically generated

Why: because physical limitation of MongoDB:

Mongo size limit for a single record = 4MB. So Survey can only be an independent document. If one user has so many surveys as subdocument collection and each survey is associated a lot of email address, it may reach the limit of the document.

To check how much space a String (email) occupies, you can go to <https://mothereff.in/byte-counter>. In the video it counted, if a survey takes 200000 emails it is already reaching the limit so we want user to be independent document.

10-129-1: When defining the schema, you can define not only the type but also the default value. For this case, you just need to add an object with type and default as key in it.

const recipientSchema = new Schema({

email: String,

responded: { type: Boolean, default: false }

});

10-129-2: How to create subdoc collection:

The subdoc collection definition will not be declared as mongoose.model but it will be exported and used by the (parent) document collection. Please compare:

Surveys.js: (document collection)

const mongoose = require('mongoose');

const { Schema } = mongoose;

const RecipientSchema = require('./Recipient');

const surveySchema = new Schema({

title: String,

body: String,

subject: String,

recipients:[RecipientSchema],

yes: { type: Number, default: 0 },

no: { type: Number, default: 0 },

\_user: { type: Schema.Types.ObjectId, ref: 'User' },

dateSent: Date,

lastResponded: Date

});

mongoose.model('surveys', surveySchema);

Recipient.js: (subdoc collection)

const mongoose = require('mongoose');

const { Schema } = mongoose;

const recipientSchema = new Schema({

email: String,

responded: { type: Boolean, default: false }

});

module.exports = recipientSchema;

Explanation on the line referencing \_user:

In the ref: User collection, find by its ObjectId (ObjectId is one type auto-defined by schema)

Underline before \_user is just a convention when writing code to remind the programmer this is a referencing field.

10-131-1: In the index.js of the server(express) side, we are requiring all the document models. But we do not need to require any sub-document models as they have already got required in their parent document models.

10-131-2: When handling route requests, please attention on conditions, such as if user has logged in, also if user has logged in, she/he has enough credits to create the survey .etc. These conditions can be added as middleware and add to the argument of the routes.

Also, middleware can be shared by multiple routes and it should have good reusuability.

10-131-3: for the status 400-499, it means the req has something wrong with it. (not correct request).

400 bad request (cannot understand request)

401 unauthorized

403 forbidden: (the server understand the request but is refusing to fulfill it)

10-133-1: When defining the route, we are actually designing the API and interface with front end. In practice, we always start from backend then go back to front end.

10-133-2: When using mongoose model, please remind that do not require a model multiple times in different files (the initialization and DB operation inside will also get executed whenever you do any require)

Mostly we will require the models in index.js. Then for all the following files executed, you can do the following instead of directly require the (survey) mongoose model:

const mongoose = require(‘mongoose’);

const Survey = mongoose.model(‘survey’);

10-134: If any property it just has a default value and you just want to use it, you do not need to mention in the instance definition. Also, Here is how to define sub-document instance.

const survey = new Survey({

title,

subject,

body,

recipient: req.recipient.split(',').map(email => ({ email: email }))

});

10-135: When splitting and processing Strings, please remember to trim the spaces:

**recipients: recipients.split(',').map(email => ({ email: email.trim() })),**

10-136-1: system design: Batch operations, reduce number of requests.

Diagram

Description automatically generated

If we use the same template for the email, the link cannot specify who is responding the email because every one gets the same link in the email. That is by default. The email provider sendgrid can help customize the link to resolve this problem.

10-137:

Diagram

Description automatically generated

When sendgrid send a message to our server ---using webhook:

A webhook (also called a web callback or HTTP push API) is a way for an app to provide other applications with real-time information. A webhook delivers data to other applications as it happens, meaning you get data immediately. ... This makes webhooks much more efficient for both provider and consumer

10-138: Sendgrid setup:

Password: As it requires 16 charactors, I repeat wXXX8 two times.

sendgrid API key:

SG.1\_AjCcc9QFy\_tJEIJCwFHg.W0UGkBX-P3nP79EAJe5iDCYykWOHWJ2YFN76CSbrohI

When you go to the API code base on github, check if there is a README.md or usecase folders. Sometimes the document is written in github folders. For sendgrid-nodejs project, all documents are under docs/use-cases folder.

10-139-1: When working on server side code, please always try to see the server log to see any bugs and you can fix them earlier. The earlier, the better.

10-139-2: Debugging tip:

When it is saying “unexpected token” or require(‘abc’) has issue, pay attention on the ‘’ single quote you use. It should not be the Chinese style single quote but it should be English style single quote !!!

10-140-1:

https://javascript.info/json

JSON related methods:

JSON.stringify --- convert objects into JSON

JSON.parse --- convert JSON back into object

Please note that a JSON-encoded object has several important differences from the object literal:

Strings use double quotes. No single quotes or backticks in JSON. So 'John' becomes "John".

Object property names are double-quoted also. That’s obligatory. So age:30 becomes "age":30.

10-140-2: We define Mailer.js as a service. Note the M is in capital because it will export a class. (However, passport.js will not export anything so it is in lower-case.

10-144-1: The mailer class is to receive the email information to construct an object to send to sendGrid. Note that the helper function for Email and helper function for content is to make some formatting for user input thus it can be accepted by sendGrid well.

class Mailer extends helper.Mail {

constructor({ subject, recipients }, content) {

super();

this.from\_email = new helper.Email("chaoyangfan@gmail.com");

this.subject = subject;

this.body = new helper.Content('text/html', content);

this.recipients = this.formatAddresses(recipients);

}

}

module.exports = Mailer;

10-144-2: email’s click tracking:

Diagram

Description automatically generated

10-144-3: What does @ mean in npm modules:

This is a new feature of NPM called 'scoped packages', which effectively allow NPM packages to be namespaced. Every user and organization on NPM has their own scope, and they are the only people who can add packages to it.

This is useful for several reasons:

It allows organizations to make it clear which packages are 'official' and which ones are not.

For example, if a package has the scope @angular, you know it was published by the Angular core team.

The package name only has to be unique to the scope it is published in, not the entire registry.

For example, the package name http is already taken in the main repository, but Angular is able to have @angular/http as well.

10-144-4: Debugging tip!!!

The tutorial asked you to npm install a module, which succeeds. But you cannot find the module under npmjs.com. A high priority is that it is deprecated. For this course, it asked me to install sendGrid. But you can only find @findgrid/mail and other package which name is different. To confirm it is deprecated, use the direct URL: https://www.npmjs.com/package/sendgrid

Even it is deprecated, it will show the information here.

10-144-5: