57)What is expressjs

In last section you saw that we have to write lot of code to handle routes and extracting data from body. That was just for geting string, we have to write different code for extracting files. Expressjs can help us with that. It does not have built in way of handling or parsing that data but it makes it easier to install another package that can can be hooked into our project that will then do the parsing for us.Now we can use a framework for all these nitty gritty task. So that we can focus on our business logic. Framework is set of helper functions but also a suite of tools and rules which we work. So basically we have a clearly defined way or atleast some outline on how we should structure our code and how we should work with that framework to write clean code.

There are many alternates of express. You can google them . but express is most popular. The great thing abut express is that it’s highly flexible and actuallt does’nt add to much functionalities out of the box but it gives you a certain way of building your application or of working with the incoming request that makes it highly extensible and therefore , there are dozens or hundreds and thousands of third party packages built for express js specifically that you can easily add to your node express node application without having to configure a lot. This is probably real strength of express and of course it also does ass some nice features out of the box.

58)Installing expressjs

Install express by-

**npm install –save express**

then we imported express in our app.js, then we created express application and stored in variable app by running express as function. so putting in other words express package seems to export function. you can see that by going to definition of express. It a typescript defination file not a js file. there we see it exports e, which is function. therefore it exports a function and we execute that function. this will initialize a new object , you could say where express js , the framework will store and manage a lot of things for us behind the scenes.so a lot of lgic is in this app constant. Now the app here actually also happens to be a valid request handler. So we pass app to create valid server. So if we do that and run **npm start**, you will actually have a running server which ofcourse will not handle any request though, because we have’nt defined any logic that should happen for incoming request. App will basically not do anything at this point, well almost. It does one thing for you. Well it sets up certain way of handling incoming request, that defines or is key characteristic of expressjs and we will have a look at that in next lecture.

App.js-

const http = require('http');

const express = require('express');

const app = express();

const { requestHandler } = require("./routes");

// const server = http.createServer(requestHandler);

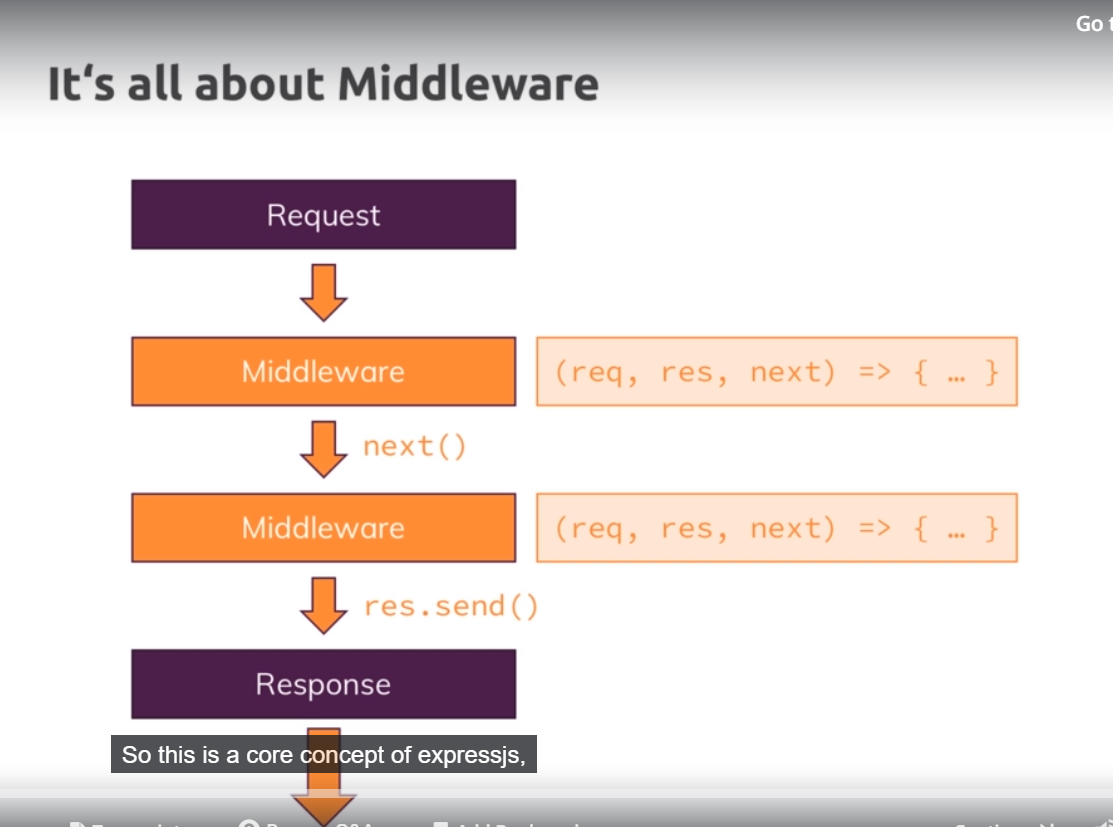
const server = http.createServer(app);

console.log('listening to 3000');

server.listen(3000);

59)Adding middleware

Express js is all about middleware. See figure-



In the end middle means that incoming request is automatically funnelled through a bunch of functions by express js. so instead of just having one request handler, you will actually have possibility of hooking in multiple functions which request will go through until you send a response. This allows you to split your code into multiple blocks or pieces instead of having one huge function that does everything and that is pluggable nature of expressjs, where you can easily ass other third party packages which simply happen to give you such middleware functions that you can plug into expressjs and add certain functionalities so this is core concept of expressjs.

We can use it our code. Before we pass app to createServer function. we can call method **use** on app, which is defined by express. Use allows us to add new middleware. Use is pretty flexible , it accepts an array of so called request handlers and it has some other use cases too. Easy way of using ot is to pass a function to it and function that you pass to use will be executed for every incoming request and this function will receive 3 arguments. Req and res-these you already know but now with some extra tricks and 3rd one is next. You can rename these. But do they do? Req and res are basically what we already know, they just a extra features now. Next is function passed by express js to this function. this function needs to be executed to allow the request to travel to new middleware.

App.js-

const http = require('http');

// const { requestHandler } = require("./routes");

const express = require('express');

const app = express();

app.use((req,res,next) => {

console.log('In the middleware');

next();

});

app.use((req,res,next) => {

console.log('In other middelware');

});

// const server = http.createServer(requestHandler);

const server = http.createServer(app);

console.log('listening to 3000');

server.listen(3000);

here w ehave 2 middlewares. We call next on first that is why our request reaches second middleware. If we dnt call next, then our request will die i.e it wnt reach next middleware. And here it wnt travel after our second middleware bcoz we are not calling next.

**So we should call next if we want function to go to next function(middleware).** If we are not calling next then we should send response from that function as request wnt travel further.

60)How middleware works

Here we will send response in second middleware. now sending response is easier, thanks to express js. we use res object to send response. We can using functions, setHeader and write(we can still use them, we used them in last section, code is in routes file). we can also use new utility function send, send allows us to send well a response and actually allows us to attach a body which is of type any. Lt me show what this could be. We can send html like this-

app.use((req,res,next) => {

console.log('In other middelware');

res.send('<h1>Heelo from express js</h1>');

});

Now if you open netweork tab of chrome, you can see that Content-Type of header is set to text/html. This is one of features of express js. since w esned some text here, express automatically sets header for us. We can overwrite by using setHeader on response object.

So now we are ending response here and this is ofcourse easier than using all these write chunks and it will be particularly easier once we start sending back files.

Question-

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5681365>

exact answer will be, after sending response, second middleware will be executed but if you try to do anything on response, you will get error.

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/t/lecture/11566260?start=0>

61)Express – Looking behind the scenes

Go to github repo of express.js. Go to lib folder. then go to reponse.js. in this file you will find lot of code. Search send and you will see how the send function(function that we are using in our code), how this is defined internally. This helps us understand it and this by the is always a great technique if you want to see what something does behind the scenes and if you need to do something yourself, for example set some header or if that is done for you . we had that default header of text/html, right. So lets see what send does internally.

It does bunch of checks internally to see if we are using outdated versions of that function which we are not using in this project, so we can ignore it. At line 141, we can see it analyses what kind of data we are sending and if we it id of type string and if we have not set header(‘Content-Type’) before it sets this header. If we have other values that string like Boolean, number et then it sets this header as binary or json data. So this is some things it do internally. You do not need to go through entire code. It’s too much. But diving into it can help ometimes.

There is one other thing. W ecan shorten this code.

Instead of these 2 lines-

const server = http.createServer(app);

server.listen(3000);

we can just do this-

app.listen(3000);

this will do both these things for us. This is something we can see in official code too-

go to application.js file. serch listen. Here we can see that listen in the end just does 2 things we did before, it calls httpCreate and passes itself, so app object which we previously also passed to createServer and then second line makes sure that listen gets called on that server object

|  |
| --- |
| app.listen = function listen() { |
|  | var server = http.createServer(this); |
|  | return server.listen.apply(server, arguments); |
|  | }; |

First it pass app to createServer method, that we were doing manually. So it does same that we were doing internally. It just saves us some code. Now we can remove import of http module from app.js file.

So now we are using expressjs and you hopefully get a basic understanding of what it’ doing and why this helps you write cleaner code because now you have clearly defiend structure, use middleware funner and you save code.

62)Handling routes

Now we want a middlewre to be executed for request with path =’/’. I mentioned that use has different versions.If you hiver over use, you can see that it has 4 overloads. Sow e have 5 different ways of using that function. you can see explanation in docs-

Got to api refrence. Then click on application(on left), then see app.use. dnt be confused that there I only one definition instead of five, I have promised. The five basically just is made up of different combinations. So first argument is optional pat argument and that is what we are looking for. This allows us to filter out certain request.however it works a bit different than our if staements before but I will come back to that. Then as second argument we have callback. Then we can have more callbacks. We can also have multiple path filters(in first argument). Lets use it code-

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

console.log('In the middleware');

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

So this means that tis midldewre will be executed for all url whose path starts with ‘/’. So tis will be executed even for /sumit. This does not mean that full path, so path after the domain has to be ‘/’, but that it has to start with that. Now ofcourse every route starts with just a slash and then we have different other criteria.

Now lets say we want to handle other rote differently. So we will do this-

const http = require('http');

// const { requestHandler } = require("./routes");

const express = require('express');

const app = express();

app.use('/add-product', (req, res, next) => {

res.send('<h1>The add product page</h1>');

});

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

// const server = http.createServer(requestHandler);

// const server = http.createServer(app);

// server.listen(3000);

app.listen(3000);

console.log('listening to 3000');

here our request will trvel to second middleware because of reason explained above, but since we are not calling next on first middleware it wnt reach second middleware. Second middleware will receive request only which do not meet filter criteria of first middleware. **So order matter a lot.**  Now lets say w ewnt to have middlewre that runs for all request and then we want to handle different request differently-

it’s simple call next on first middleware-

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

console.log('this will be executed for all');

next();

});

app.use('/add-product', (req, res, next) => {

res.send('<h1>The add product page</h1>');

});

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

By the way of you sending reponse, this is good indication that you never want to call next too, because you dnt want to eecute any other response related code, just as before with vanilla nodejs, you dnt want to send more than one response, this wont work and will result in an error.

63)Parsing Incoming requests

Here we send user a form, ten we want to rint data entered by use on form and redirect user to ‘/’. In express we can do this by-

const http = require('http');

// const { requestHandler } = require("./routes");

const express = require('express');

const app = express();

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

console.log('this will be executed for all');

next();

});

app.use('/add-product', (req, res, next) => {

res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

});

app.use('/product', (req,res,next) => {

console.log(req.body);

res.redirect('/');

});

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

// const server = http.createServer(requestHandler);

// const server = http.createServer(app);

// server.listen(3000);

app.listen(3000);

console.log('listening to 3000');

express gives us function redirect. For redirection, it is easier to use than manually setting statusCode and then setting location header. But when we print req.body(this is provided by express js) we get undefined.we re almost there, request gives us this convenient body property. But By default express doesn’nt try to parse this request incoming body. For that we add a middleware , this should be excuted before all middleware in which you try to access body of request. We install body-parser package for that.

Now this wuld actually be included in express by default because community wanted that again, it was in past , then it was removed, then it was re added. I will use that third party package which is recommened way of using it, because they can remove it again.

code-

const http = require('http');

// const { requestHandler } = require("./routes");

const express = require('express');

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

const app = express();

app.use(bodyParser.urlencoded({extended: false}));

app.use('/add-product', (req, res, next) => {

res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

});

app.use('/product', (req,res,next) => {

console.log(req.body);

res.redirect('/');

});

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

// const server = http.createServer(requestHandler);

// const server = http.createServer(app);

// server.listen(3000);

app.listen(3000);

console.log('listening to 3000');

We execute bodyParser.urlEncoded in middleware. You can pass options to configure it , but here we do not need it.it registers a middleware, so urlEncoded function yields us such a middleware function(like we pass to middleware) and this package will cll next in this function so that our request reaches next middleware but before it do that, it do the whole request body parsing that we have to do manually in prevous sections. Now this will not parse all kinds of possible bodies,files,json and so on. But this will parse bodies like we are getting here, send through form. If we have other bodes like files , we will use different prsers and this makes express js so extensible. If we need something we can just plug it it. You can see how easy it is. Lets run it.

So I am getting warning, so I should pass config object to urlEnoded. There I set extended to false. This is if it should be able to parse non-default feature you could say. Whenever you install new package, manually restart your server dnt rely on nodemon to do it.

Now we can see that we are geeting js object , with key as name of input and value us text enterted by user. This is way easier to work with.

64)Limiting middleware execution to POST requests

We want this middleware only to be executed for post request-

app.use('/product', (req,res,next) => {

console.log(req.body);

res.redirect('/');

});

To do that we can use post method instead of use.syntax and all other things remain same. Similarly we have other methods like get,put,patch ,delete. Now w eare filtering request based on http method and url as well.

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

console.log(req.body);

res.redirect('/');

});

65)Express router

We are putting all logic in app.js. which is not ggod. We ant to split our routing logic in different files and import it into app.js. we could do this, we could create files where we export these functions(function we pass to middleware) but express js actually gives nice way of outsourcing routing into other files.

So we create routes folder. in that folder we have 2 files, admin.js and shop.js. we want to move some routes to these files. Now one convenient feature offered by express to achieve this is to go into these files and import express there again, you can and you typically do this in multiple files and then we can use a feature on it called router. And we create Router by calling function Router on express, we execute this function. this router is like mini express app tied to other express app or pluggable into other express app, I will say, which we can export here.so we export router at the end of the file. but before that we use router to register things . so now router here can be used to again define a use function for all request , a get function for get and so.

Then we cut paste 2 routes from app.js to admin.js and replace app with router, now router that we export from admin.js has these 2 routes registered. Other code can stay as it is because router functions(use,get,post) here basically work in exactly the same way as the app functions(use,get,post) does or app.

admin.js-

const express = require('express');

const router = express.Router();

router.get('/add-product', (req, res, next) => {

res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

});

router.post('/product', (req, res, next) => {

console.log(req.body);

res.redirect('/');

});

module.exports = router;

then we import this router in app.js. we give relative path and again we can emit .js at the end.

Code-

App.js-

const http = require('http');

// const { requestHandler } = require("./routes");

const express = require('express');

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

const app = express();

const adminRoutes = require('./routes/admin');

app.use(bodyParser.urlencoded({extended: false}));

app.use(adminRoutes);

// app.use('/add-product', (req, res, next) => {

// res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

// });

// app.post('/product', (req,res,next) => {

// console.log(req.body);

// res.redirect('/');

// });

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

// const server = http.createServer(requestHandler);

// const server = http.createServer(app);

// server.listen(3000);

app.listen(3000);

console.log('listening to 3000');

Now we are importing router object here and this router object in turn has routes registered(in admin.js) and the nice thing about this router is that it is a valid middleware function. so we call app.use and put our adminRoutes there. Now it will automatically consider our routes in admin.js when funnelling the request through this middleware in app.js. so jus like before order matters. So if we reverse their order-

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

app.use(adminRoutes);

we will never reach admin routes. Now lets move rest of routes to shop.js. code-

app.js-

// const http = require('http');

// const { requestHandler } = require("./routes");

const express = require('express');

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

const app = express();

const adminRoutes = require('./routes/admin');

const shopRoutes = require('./routes/shop');

app.use(bodyParser.urlencoded({extended: false}));

app.use(adminRoutes);

// app.use('/add-product', (req, res, next) => {

// res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

// });

// app.post('/product', (req,res,next) => {

// console.log(req.body);

// res.redirect('/');

// });

app.use(shopRoutes);

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

// res.send('<h1>Hello from Sumeet</h1>');

// next(); // this allows request to continue to next middleware

// });

// const server = http.createServer(requestHandler);

// const server = http.createServer(app);

// server.listen(3000);

app.listen(3000);

console.log('listening to 3000');

shop.js-

const express = require('express');

const router = express.Router();

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

res.send('<h1>Hello from Sumeet</h1>');

next(); // this allows request to continue to next middleware

});

module.exports = router;

now on important thing, if I switch the position of routes here that is if ido this-

app.use(shopRoutes);

app.use(adminRoutes);

our app will work fine. You will say that admin routes will never be reached because **get(‘/’)** in shproutes will match all routes, but this only happens because we have **get** in shoprroutes and not **use.** Get,post and so will actually do an exact match here. if we replace get with use in shoproutes, then it will executed for all incoming request and adminRoutes will never be reached.

So this exact matching is not achieved because we use router but recause we use get here and that would have been same if we stick to app way of doing this in app.js file, we had previously.

66)adding 404 error page

Right now we get error if we visit a route that we have not defined in our app. we would like to show a error page. So we add a third middeware, after our 2 middleware that handles our routes. This middleware will only we reached for reqest for whichw e do not have matching route-

App.js-

app.use(shopRoutes);

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

// res.send('<h1>Hello from Sumeet</h1>');

// next(); // this allows request to continue to next middleware

// });

app.use((req,res,next) => {

res.status(404).send('<h1>Page Not Found</h1>');

});

// const server = http.createServer(requestHandler);

// const server = http.createServer(app);

// server.listen(3000);

app.listen(3000);

here we also set status by chainng method status before e=send. You can also chain setHeader method to set headers. But remember send method needs to be last in chain. And ofcourse this chaining works no matter you used router approach or app approach.

67)Filtering Paths

Lets say all routes in admin ].js start with ‘/admin’. Now either we can handle t in this way-

App.js-

app.use(adminRoutes);

admin.js-

router.get('admin//add-product', (req, res, next) => {

res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

});

router.post('admin//product', (req, res, next) => {

console.log(req.body);

res.redirect('/');

});

Second approach would be-

app.use(‘/admin’,adminRoutes);

in admin.js-

router.get('/add-product', (req, res, next) => {

res.send('<form action="/product" method="POST"><input type ="text" name ="title"><button type="submit">Add Product</button></form>');

});

router.post('/product', (req, res, next) => {

console.log(req.body);

res.redirect('/');

});

Now we will reach admin routes only if request url start with ‘/admin’. /amdin part will be removed from url when matching routes in admin.js.

So this filtering mechanaims allow us to put a common starting segment for our path, (which all routes in a given file use) to outsource that into this app.js file, so we dnt have to repeat that commn purl part in al routes in that routes file.

69)Serving HTML files

Here we want to send html files. We use sendFile method on response. As you can see in ide, this method automatically sets content-type header, we will see if that works for us or not. Now we need to give path to file that we want to serve. This path needs to be absolute. Here we pass absolute path send from app.js because all functionality in the end is imported into app.js file. shop.js-

const express = require('express');

const router = express.Router();

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

res.sendFile('/views/shop.html');

});

module.exports = router;

we get error –

Error: ENOENT: no such file or directory, stat 'D:\views\shop.html'

If we use relative path –

res.sendFile('./views/shop.html');

we get-

TypeError: path must be absolute or specify root to res.sendFile

So w eneed to specify absolute path. But / like this(in our first absolute path) refers to root of our os not to this folder. so in order to construct the path to this directory and this file here ultimately , we can use a feature provided by nodejs, another core module. We import path module. Thenw e send file wherew e create a path with the help of this module by calling join method. Join yields us path in the end, it returns a path but it construts this path by concatenating the different segments. First segment that we pass here is global variable variable made avalaible by nodejs, it is **\_\_dirname.**  This variable holds the absolute path on our operating system. To this project folder. the we pass other 2 segments. Do not pass / here because and that’s important , we use joinPath not because of absolute path, we could build this with \_\_dirname and concatenating other 2 segments(views and shop.html), but w eare using pathjoin because it will automatically build the path in way that works on both linux and windows system.

In linux we have path like this-

‘/user/products’

On windows-

‘\user\products\’

Therefore if you manually construct this with slashes, it would works on windows but will fail on linux. Path.join basically detects the o system you are running on and automatically builds the correct path.

Now \_\_dirname points to routes folder.\_\_dirname gives us path to a file in which we use it and we are using it in shop.js file in the roites folder. so it will point to the routes folder. but views is located in sibling folder to routes. Solution to this is we add one more segment to our path i.e ‘../’ it means take one step back and then take add rest of path.

Shop.js-

const path = require('path');

const express = require('express');

const router = express.Router();

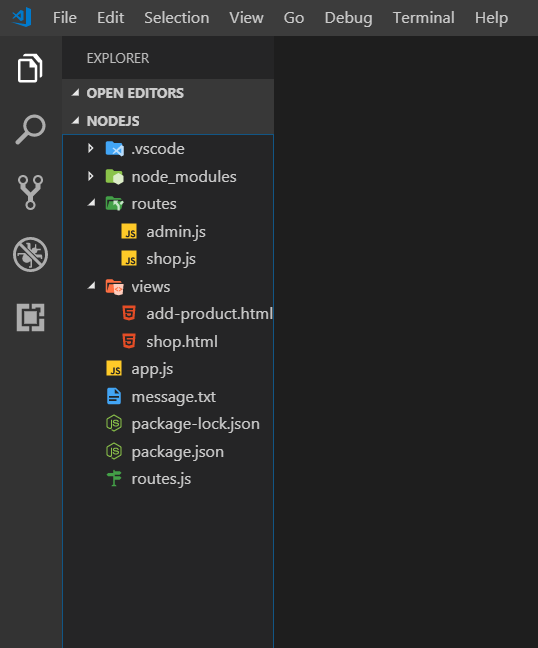
router.get('/', (req, res, next) => {

res.sendFile(path.join(\_\_dirname, '../', 'views', 'shop.html'));

});

module.exports = router;

filesystem-



Similarly serve other html file in admin.js.

71)Using a helper function for navigation

Before we move forward, let me add one note on how we navigate to root folder in shop.js and admin.js. first we can use .. instead of ../. so this will work fine-

res.sendFile(path.join(\_\_dirname, '..', 'views', 'shop.html'));

this should be preferable event through both should work on windows and mac because now we make no assumption about separator, we are using while constructing a path.

But there is even a nicer way we can implement it. We could also get the parent directory with the help of little helper function.

There we call dirname on path. Dirname as you can seend in quick help on right, returns the directory of the path. So this sounds pretty useful, if we use that we just have to find out which directoy or which file we want to get the directory name. there we use global process variable that is avalaible in all files, on it we have main module property. This will refer to main module that started your application. So basically the module, we created in app.js, then we can call filename to find out in which file this module was spun up.so in other words-

process.mainModule.filename gives us the path to the file that is responsible for the fact that our application is running. Then we put this file name into path.dirname to get a path to that directory.

So

process.mainModule.filename

gives –

**D:\Max's Courses\nodejs\app.js**

And

console.log(path.dirname(process.mainModule.filename));

gives-

D:\Max's Courses\nodejs

Lets use it in our routes file

Path.js-

const path = require('path');

module.exports = path.dirname(process.mainModule.filename);

admin.js-

const rootDir = require('../util/path');

router.get('/add-product', (req, res, next) => {

// res.sendFile(path.join(\_\_dirname, '../', 'views','add-product.html'));

// res.sendFile(path.join(\_\_dirname, '../', 'views','add-product.html'));

res.sendFile(path.join(rootDir, 'views', 'add-product.html'));

});

72)styling css pages

It is simportant to learn how you can serve css in node app too, because typically , you have css and javascript code in your apps. Lets start with css in your html with style tag. We wnt keep it in this way. Exporting a style sheet wnt work at this moment. Code-

If you are wondering about strange class name, I ma using a styling system named bem, you can google – css bem to learn more. It is certain way of naming your classes to avoid double class names. Then we added hmt and css here.

73)Serving files statically

All css is in html file. we want to have separate css file.Now the problem is right now we can’t easily import them. But lets see how we atleast theoretically would want to do that. Now typically , you would have some css files somewhere and point at the and app would get served. Now you can create a new folder and can make it whatever you want. The convention is to call it public because you want to indeicate that this folder that holds content which are always exposed to public , so where you dnt need permissions to access it, all you files like html are not accessible by our users. If you enter this in browser-

Localhost:3000/views/shop.html

That will not work, becaue this is accepted by express and it tries to find route that matches it. So you cnt access the file system. That is good and what you want. But now I want to make a exception, I want some files to access filesystem. bcoz in html, we will point to css files. So cut all css from html and paste it in main.css file in css folder in public folder. in html have this-

<link rel="stylesheet" href="public/css/main.css">

But we see in browser that these files are not served. You can see in network tab you can see that we see 404 status for main.css. it makes get request at this url-

<http://localhost:3000/public/css/main.css>

now to make this work we need a feature express js offers us . we need to serve the files statically. Statically means not handled by express router or other middleware but directly forwarded to file system. To do this we register a new middlesware. It is the one that express js ships with. We use static method on express and pass it path to folder that we want to serve statically. So basically the folder which we want to grant read access to. It’s only read access but that is still more that what we normally have.

const express = require('express');

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

const path = require('path');

const app = express();

const adminRoutes = require('./routes/admin');

const shopRoutes = require('./routes/shop');

app.use(bodyParser.urlencoded({extended: false}));

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use(adminRoutes);

app.use(shopRoutes);

app.listen(3000);

now if we save and rerun, still we cnt acesscss file. this is because we are using wrong path. We dnt need to use public, we need to assume that we are in public folder.

so correct path would be-

<link rel="stylesheet" href="/css/main.css">

So this is what express will do here, it will take any request that tries to find some file and that’s important , it looks at extension. So if we have such a request it automatically forwards it to public folder and therefore in html path dnt need to start from public folder. so we need to give path assuming that we are in public folder because this is where file requests will be forwarded to.

You can also register multiple static folders and it will funnel the request through all of them until it has first hit for file. like that-

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use(express.static(path.join(\_\_dirname, 'public2')));

you can serve any files like that css, js images etc.

question-

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5705629>

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5705299>

74)Wrapup

Express js is all bout middleware and unsderstanding that flow of request through all the middleware fucntions. Expressjs is very popular bcoz its highly extensible and as you already saw with body parser in this module, you can easily plug some pacakges , a lot of packages actually, into express app because they just expose such middleware functions and you can add them and request gets funnelled through them.