40)Section Intro

In this section we will introduce express.

41)Hello Expresss

In this lecture we will make web server using express. Express in one of reasons node go so popular. Because it is so easy to make rest api’s and sttic web servers. Express is a no non sense library. Now there are lot of different ways to configure it , so it can get pretty complex. That why we will use I next couple of sections.

Express is very vast library. Instead of having single doc on npm website, it has whole website dedicated to it. This is website-

<https://expressjs.com/>

you can use this link to explore things that we do not cover here. Now lets install express-

**npm install express@latest –save**

to run web server we need file, so lets create server,js n node-web-server folder. in this file we will configure various routes. Here we will start server, binding it to port on our machine. Now we will be deploying to real server latesr. For now our loca examples are going to happen in our local host. Code-

const express = require('express');

let app = express();

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

res.send('Hello Express');

});

app.listen(3000);

run this file, go to localhost:3000, you will see hello in browser.

Let set our first route. First argument is url pattern and second one is callback that will be called. Callback has 2 arguments req,res. What they do is clear from their name. req is request that comes in, response is response that we send back. We can set status code, you can customize data that you send back.

Then we use send method on res to send , it lets us respond to request, sending some data back. Now we are not done yet. Our route is set up but or app is not going to actually start listening.whta we need to do is to call app.listen, it is going to bind the application to a port on our machine. In local server we use 3000, later on will talk about how to customize this depending on whatever server you use to deploy your app to production.

Go to network tab in chrome dev tools. Here click on request and go to headers tab. In response headers you can see

Content/type: text/html

It tells browser that this is html, so please render it. It is sutomatically set by express. Express handles a lot of things automatically, that is why it is so popular. We can also do this-

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

now this is html, browser will interpret it.

We can also send json data. Code-

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

res.send({

name: "Sumeet Sood",

likes: ['Biking', 'cities']

});

});

When we pass object to res.send, express notices tht, it takes it converts it into json and send it back to browser. In browser we can see json data, it will be formatted by some extension. The reason it was picked up by browser was because content/type on response is set to application/json. It tells client that it is json data please parse it as such.

Lets set up secondroutr-

const express = require('express');

let app = express();

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

res.send({

name: "Sumeet Sood",

likes: ['Biking', 'cities']

});

});

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

res.send('About Page');

});

app.listen(3000);

42)Creating Web Server

Here we will learn how to setup static directory. So if you have website with html,js,images,css you can go ahead and serve that up, without needing to provide a custom route for every single file which would be a real burden. So lets create some static assets that we can serve up. We create a new folder everything inside this directory will be accessible via web server. Everything here should be intended to be viewable by anybody. this folder is public folder. it will have our all static assests. In this folder we created a file called help.html.

Now we have html page and goal is to be able to serve this page up in our express app without manually configure it.

Code-

const express = require('express');

let app = express();

app.use(express.static(\_\_dirname + '/public'));

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

res.send({

name: "Sumeet Sood",

likes: ['Biking', 'cities']

});

});

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

res.send('About Page');

});

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

res.send({

errorMesage: 'Unabale to handle request'

});

});

app.listen(3000);

we are going to do that using piece of express middleware, middle ware lest you configure how your express works and it is something we will be using extensively throughout the course. For now you can think of it kind of like a third party add on. You are saying express, hey you express you usually work like this, I would like to tweak alittle bit and work like this. In order to add some middle ware, we use **app.use**.it takes middleware function that you want to use. In our case we are using a built in piece of middle ware. Leter we will make our own middle wares.

Here we pass express.static function, it takesabsolute path to folder you want to serve up. We want to serve help.html we need to provide path to public folder. this means we need to specify the path from root of our hard drive. This can be tricky as our rojects move around. Luckily we have \_\_dirname variable, this is passed b wrapper function that we have explored earlier. \_\_dirname stores the path to your projects directory. In our case it holds the to node-web-server folder.

app.use(express.static(\_\_dirname + './public'));

with this our server is setup and we have nothing to do.

Now we can access http://localhost:3000/help.html

We will see help.html. we just open localhost:3000, we will get json data, but we change our url as shown above, we will get html page.

Being to able to set up static directory that easily has mode node the go to choice for simple projects that don’t really require a backend. If you want to create a node app for sole purpose of serving up directory, you can do it in about 4 lines of code. We can include jaavscript, css and images files in our directory.

app.listen also takes a second argument, we can specify function to be executed when our server is up because it can little bit of time to get server started.

app.listen(3000, () => {

console.log('Server is up on port 3000');

});

43)Rendering Templates with Data

Here we can see how we can serve templating engine. We will see handle bar view engine. Go to official website to learn more, something that we do not cover in this video. Now the thing we are going to be installing is actually a module that’s wrapper around handlebars. It’s gong to let us use it as express view engine. Goggle it-

npm/package/hbs

here you can see official docs. Now lets install this pacakage-

**npm install hbs –save**

actually configuring express to use this handlebars view engine is super simple. You just have to add one line. We call app.set, it allows us to set various express related configurations. There’s lot of built in ones , we will betalking about more of them later. For though we are going to pass key-value pair. Key is thing you want to set and value is value that we want to set. Here key is **view engine** , it tells express what view engine we want to use.

I order to make our first template,lets make a directory that will hold our templates. Name it views.views is default directory that express uses for templates. Inside this folder we will make template for our about page. Create file, about.hbs. file extension is important. If you want to name this folder something else, use this-

app.set('views', \_\_dirname + '/someFolder');

link-

<https://www.udemy.com/the-complete-nodejs-developer-course-2/learn/v4/questions/4548118>

Then in route handler we use res.render. render is going to let you render any of the templates you have set up with your current view engine.

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

// res.send('About Page');

res.render('about.hbs');

});

As I told you views is default directory, for templates. That is why we do not have to mention it here. About.hbs is in views folder. right now about. hbs is simple html. We will how we can inject data inside our templates.

Server,js-

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

// res.send('About Page');

res.render('about.hbs', {

pageTitle: 'About Page',

currentYear: new Date().getFullYear()

});

});

So we pass a object. we use data of this object in hbs like this-

About.hbs-

<body>

<h1>{{pageTitle}}</h1>

<p>Some Text Here</p>

<footer>

<p>Copyright {{currentYear}}</p>

</footer>

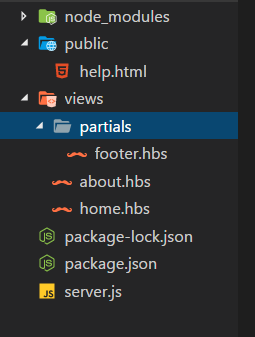
</body>

44)Advanced Templating

Now we have 2 hbs pages. Some html is repeated in both pages. To solve this problem we have use partial. Partial is a partial piece of your web site. Its something that you can reuse throughout your templates. Before using it we have to set up server.js to let handlebar know that we want to add support for partials. So we do this by-

hbs.registerPartials(\_\_dirname + '/views/partials');

here we have to give absolute path where our partials are placed. We create a folder in views called partials, here we will have other partial files. Structure-



Footer.js-

<footer>

<p>Copyright {{currentYear}}</p>

</footer>

Home.hbs(before using partials) –

<body>

<h1>{{pageTitle}}</h1>

<p>{{welcomeMessage}}</p>

<footer>

<p>Copyright {{currentYear}}</p>

</footer>

</body>

</html>

Home.hbs, after using partials-

<body>

<h1>{{pageTitle}}</h1>

<p>{{welcomeMessage}}</p>

{{> footer}}

</body>

</html>

Footer is name of our partial file. there is no change in how we pass data. Server,js-

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

res.render("home.hbs", {

pageTitle: "Home Page",

currentYear: new Date().getFullYear(),

welcomeMessage: 'Welocme to my website'

});

});

By default nodemon will not look into hbs files for changes. We want it to run this-

**nodemon server.js -e js,hbs**

here e means extension that we want to watch for changes.

This is one more thing I want to talk about. This is handlebar’s helper. Handlebar’s helper are going to be ways for you to register functions to run, dynamically create some output.

Right now we in server.js, we are currently inject currentyear inside both our templates and that’s not really necessary.

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

res.render("home.hbs", {

pageTitle: "Home Page",

currentYear: new Date().getFullYear(),

welcomeMessage: 'Welocme to my website'

});

});

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

// res.send('About Page');

res.render('about.hbs', {

pageTitle: 'About Page',

currentYear: new Date().getFullYear()

});

});

There is better way to pass this data in, this data should’t need to be provided because we are always going to be using exact same function. lets see how e can use helper. First we need to register this helper. In server.js-

hbs.registerHelper('getCurrentYear', () => {

return new Date().getFullYear();

});

Now where we call this helper function, callback that we specify here will be executed. Now we remove currentYear property from render calls-

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

res.render("home.hbs", {

pageTitle: "Home Page",

welcomeMessage: 'Welocme to my website'

});

});

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

// res.send('About Page');

res.render('about.hbs', {

pageTitle: 'About Page',

});

});

We used this currentYear , in footer. Like this-

<footer>

<p>Copyright {{currentYear}}</p>

</footer>

Now we replace it function

<footer>

<p>Copyright {{getCurrentYear}}</p>

</footer>

There is no need of any special syntax. If you use something inside {{}} that is not partial, handle bars is going to look for helper with that name, if there is no helper it will look for piece of data with that name. in this case it will find helper method with this name.

Now helpers can also take arguments. Lets take a helper. It will take a string and it will return that string in upper case.

Server.js-

hbs.registerHelper('screamIt', (text) => {

return text.toUpperCase();

});

Lets sue this in home.html-

<body>

{{> header}}

<p>{{screamIt welcomeMessage}}</p>

{{> footer}}

</body>

We are need to pass 2 aguments then-

<p>{{screamIt welcomeMessage secondArgument}}</p>

So using handle bar’s helpesr we can create both functions that don’t take arguments and functions that do take arguments.

45)Express Midlleware

Express middleware allows you to add on to existing functionality that express has. So if express does’nt do something you would like it to do, you can add some middleware and teach it how to do that. We already have added some middle ware.

app.use(express.static(\_\_dirname + '/public'));

We use middleware and we teach express how to read from a static directory .

We call **app.use** to register middleware, then we provide middle ware function to use.

Middleware can do anything you can execute some code like logging into the screen, you could make a change to request or response object, we will be doing just in authentication section, we will check request that it had valid token before it reaches any request handler.

We can also use middleware to respond to a request, we could send something back from the middle ware just like we would anywhere else using **response.render** or **response.send.**

To explore middlewre we are going to create some basic middleware right now. Just below where we call app.use registering our express static middleware, we again call app.use, it takes a function. app.use just take one argument which is this functiom. Now this function has 3 arguments req,res and next. You know about req and res. Next is when we want req to travel ahead, like reaching other middle wares or request handlers.

So if we do something asynchronous the middle ware is not going to move on until we call next. If we do not call next request will not go to next middleware or request handler. Code-

const express = require('express');

const hbs = require('hbs');

const fs = require('fs');

let app = express();

hbs.registerPartials(\_\_dirname + '/views/partials');

app.set('view engine', 'hbs');

app.use(express.static(\_\_dirname + '/public'));

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

let now = new Date().toString();

let log = `${now}: ${req.method} ${req.url}`;

console.log(log);

fs.appendFile('server.log',log + '\n', (err) => {

if (err) {

console.log('Error has occured');

} else {

console.log('File Written');

}

});

// fs.appendFileSync('server.log', log + '\n');

next();

});

hbs.registerHelper('getCurrentYear', () => {

return new Date().getFullYear();

});

hbs.registerHelper('screamIt', (text) => {

return text.toUpperCase();

});

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

res.render("home.hbs", {

pageTitle: "Home Page",

welcomeMessage: 'Welocme to my website'

});

});

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

// res.send('About Page');

res.render('about.hbs', {

pageTitle: 'About Page',

});

});

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

res.send({

errorMesage: 'Unabale to handle request'

});

});

app.listen(3000, () => {

console.log('Server is up on port 3000');

});

Here we save the request information in server-log file. ig we do not call next function in middleware, the request is not going to move forward. The more thing you can also send response back from middleware. There is no difference between res object that you get in middle ware and what you get in request hander.

Code-

const express = require('express');

const hbs = require('hbs');

const fs = require('fs');

let app = express();

hbs.registerPartials(\_\_dirname + '/views/partials');

app.set('view engine', 'hbs');

app.use(express.static(\_\_dirname + '/public'));

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

let now = new Date().toString();

let log = `${now}: ${req.method} ${req.url}`;

console.log(log);

fs.appendFile('server.log',log + '\n', (err) => {

if (err) {

console.log('Error has occured');

} else {

console.log('File Written');

}

});

// fs.appendFileSync('server.log', log + '\n');

next();

});

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

res.render("maintainance.hbs");

next();

});

hbs.registerHelper('getCurrentYear', () => {

return new Date().getFullYear();

});

hbs.registerHelper('screamIt', (text) => {

return text.toUpperCase();

});

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

res.render("home.hbs", {

pageTitle: "Home Page",

welcomeMessage: 'Welocme to my website'

});

});

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

// res.send('About Page');

res.render('about.hbs', {

pageTitle: 'About Page',

});

});

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

res.send({

errorMesage: 'Unabale to handle request'

});

});

app.listen(3000, () => {

console.log('Server is up on port 3000');

});

Here we created another middle ware. Now we send the response from middleware back. So our request do not travel further.This middle ware we stop anything else from executing. We do not call next, so request will not love further.

Here one thing is important. We access static files in browsers, they wll be accessed and we will not see maintainance page. This is because middleware that send static files is configured before mainatince page middleware, in case of static files, response is sent back from that middleware. Request does not reach maintainance middle ware. So the order of middlewares matter.

We can solve this problem by moving static middleware after maintainance middleware.

One more thing , if you want to know methods tat exist on request object, go to express official website, click on api refrence then on right hand side click on request. Similarly you can see all methods here.