**Lecture 58**

**What is Express.js?**

So what is express and why do we use it? Well I briefly mentioned it, writing all that server side logic is pretty complex, just remember what we had to do to parse an incoming request. For extracting the body, we manually had to listen to the data event, to the end event and then create a buffer which we in the end converted to a string and this was just one type of data we could get. If we get other kinds of data, like for example we get a file or differently structured data, then we would have to write new logic. Now expressjs helps us with that, it actually doesn't have a built-in way of handling or parsing that data but it makes it easy to install another package that can easily be hooked into our project that will then do the parsing for us and you will see what I mean in a second. We in general don't want to care about all these nitty gritty details, we want to focus on our code that defines our application, so the thing that really sets our application apart from other applications, our unique selling point you could say and we do use a framework for this, for all the heavy lifting. A framework is basically a set of helper functions but also a suite of tools and rules with which we work, so basically we have a clearly defined way or at least some outline on how we should structure our application, our code and how we should work with that framework to write clean code and of course, I will teach you all of that for expressjs in this module. So expressjs helps us with that and this is why we will dive into it here. Now of course expressjs is not the only package or framework you can use for nodejs that will help you write better nodejs code and focus on your business logic. Now from one you could of course stick to vanilla nodejs, we only use that thus far and of course that works and depending on the complexity of your application or the level of challenges you are seeking, you can absolutely stick to vanilla nodejs, you can theoretically write everything on your own just with that. There also are other frameworks you could use, for example there is adonis.js. Now if you ever used laravel for php, this is basically a laravel inspired framework for nodejs but not from the same creators. There also is koa or a sailsjs and there are many more, you can basically Google for expressjs alternatives and you will find plenty of blogposts diving into the different alternatives and what their strengths and weaknesses are. But expressjs is by far the most popular and most often used one which is why I will also teach it here. . So why don't we just install it and take a closer look ourselves.

**Lecture 59**

**Installing Express.js**

Now back in our project, let's add expressjs and you could also simply create a new project, make sure to install nodemon though, I will keep using that but I will get rid of the routes.js file because I will basically create a brand new application you could say. So therefore of course in the app.js file, I have to get rid of that import and that of course means I'm now missing my route handler or my request handler here but we'll work on this later. So let's now install expressjs by running npm install --save, now why not --save dev but --save? Because this will be a production dependency. We don't just use that as a tool during development, it will be an integral part of the application we ship and therefore, it definitely also has to be installed on any server or any computer where we run our application once we deploy it, it is a major piece of our application. So let's install it as a production dependency with this command and the name of the package just is express, so npm install --save express will install expressjs into our project. Now once it is installed, you see an entry was added to the dependencies in our package.json file here and now we can start using it and to use it, I'll go to my app.js file. and here I will import express, you can name that constant however you want of course but the package is just called express. Now you can also of course group that together with the core module import, I like to separate my node specific modules and the third party packages and also my own imports if I have them with empty lines so that you can clearly see what's what but this is not required. So now express is imported and now let's also get rid of that console log statement here, you can create an express application and store it in a constant named app, though that name of course is always up to you by running express as a function. So put in other words, the express package seems to export a function in the end and actually you can see this if you hold command or control in Windows and you click on the express here, you're taken to the source code and there, you can actually see that in the end at the bottom of the file, it exports E and don't worry about that syntax here, this is not javascript file, it's a definition typescript file but still it exports E which is this function in the end. So it exports a function here and therefore we execute it as a function and this will initialize a new object, you could say where expressjs, the framework will store and manage a lot of things for us behind the scenes, so a lot of logic is in this app constant here. Now the app here actually also happens to be a valid request handler, so you can pass app here to create server and if you do that and you run npm start, you will actually have a running server which of course will not handle any requests though because we haven't defined any logic that should happen for incoming requests, app will basically not do anything at this point. Well almost, it does one thing for you and that is it sets up a certain way of handling incoming requests that defines or that is a key characteristic of expressjs and we'll have a look at that in the next lecture.

**Lecture 60**

**Adding Middleware**

Expressjs is all about middleware and you see a diagram here, in the end middleware means that an incoming request is automatically funneled through a bunch of functions by expressjs, so instead of just having one request handler, you will actually have a possibility of hooking in multiple functions which the 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 this is the pluggable nature of expressjs, where you can easily add other third party packages which simply happen to give you such middleware functions that you can plug into expressjs and add certain functionalities but more on that later. So this is a core concept of expressjs, the middleware and we can use that by going here after we created the app object but before we passed it to create server and then we can use the app and call a method which is defined by the express framework, use. Use allows us to add a new middleware function, now the use method is pretty flexible, it accepts an array of so-called request handlers here and it has some other use cases too. Now one easy way of using it is that you simply pass a function to it and this function here, this function you pass to app use will be executed for every incoming request and this function will receive three arguments, the request and the response object as you already know it basically with some extra tricks learned though and a third argument which is the next argument. Now you can rename any of these arguments but what do they do? Request and response as I just mentioned are basically what you know with some extra features. Next is actually a function, a function that will be passed to this function by expressjs and this can be confusing because you are passing a function as an argument to the use method and this function you're passing is receiving yet another function here on the next argument and this next argument, basically this function you're receiving here has to be executed to allow the request to travel on to the next middleware. Now let me show you what I mean. We can simply console log in the middleware here, like this, now since I have nodemon this automatically restarts the server and let's now go to the browser and reload localhost 3000. Now actually this will keep on spinning, you see, so we don't get a response which makes sense because we've got no logic where we would send one, in the console here at the bottom, you see in the middleware though, so this did execute, this is what I meant, this allows us to hook into this funnel through which the request is sent. If I duplicate this and I add another use statement here in another middleware and I save this and let it restart therefore and I now reload this page here on localhost 3000, then I see in the middleware here again and I see it twice because I pressed reload twice in my case here but I don't see in another middleware. Now the reason for that is that we have to call next here to allow the request to travel on to the next middleware in line. So it basically goes from top to bottom through that file you could say, through all the middleware functions but only if we call next, if we don't call next it just dies, so if we don't call next, we should actually send back a response because otherwise the request can't continue its journey, so it will never reach a place where we might send a response but if we also don't send one here, well then we never send one. So now with this next call added, we actually make it into this middleware and we should therefore see this console log and here we could then even send a response. We'll do this as a next step because sending responses also changed a bit. So restarted the server, reload that page and now you see in the middleware and in another middleware thanks to next. So this allows the request to continue to the next middleware in, whoops middleware in line which is the middleware below this one. So this is a crucial concept, this ideas of middleware and you can use any function that has this format, so that receives request, response and next. And you should call next if you want to allow the request to go to the next function, you should send a response if you got other plans, so let's send a response in this middleware in the next lecture.

**Lecture 61**

**How Middleware works**

We added middlewares and I mentioned that this is a crucial concept of the express framework and that we should call next if we don't send a response because otherwise, the request will just die and will not continue to the next middleware. Now in this middleware, we're not calling next and there also wouldn't be a next middleware in line, this is our entire script right, there is nothing to come. Expressjs and that's important doesn't send a default response or anything like that, so instead we should send a response here. So we can use the response object and now sending responses actually gets easier, thanks to expressjs . Instead of setting a header which we still can do and writing which we also still can do, so we can still send responses as before but instead of doing this, there is a new utility function we can use, send. Send allows us to send well a response and actually this allows us to attach a body which is of type any, now let me show you what this could be. We could send good old html code here, just h1 tag, hello from express, like this. If we do that and we now reload this page here, we see hello from express, by the way one thing you'll notice is that if you open your network tab here and you inspect that request you got, you will see that under headers, the content type is automatically set to text html here. So this is done for you, this is another feature provided by express here. The send method by default here since we send some text here simply sets an html content type, you can still set one manually with set header of course, so you can always override this expressjs default but you can also rely on the default where the default response header is text html. And now with that, you see that we also get no dying request anymore because even though we're not calling next here and we shouldn't, we're doing the alternative, we're sending a response with send and this is of course easier than using all these write chunks and it will be particularly easier once we start sending back real files or the content of files, something we haven't done at all thus far. Now this is something we'll also do in this module but for now, make sure you understand this basic middleware concept, that you add functions that are hooked into this funnel through which the request goes and you either have next to reach the next middleware or you send a response to, well not do anything else. And of course if we would send a response here instead of calling next, we would never reach that middleware, so you really just travel from middleware to middleware, from top to bottom by calling next.

**Lecture 62**

**Express.js – Looking behind the scenes.**

So we now had some basic work with expressjs . now let's also dig into the internals of expressjs, at least a little bit. Here is the expressjs code repository on github, it's open source so you can dive into the code and no worries, we'll not do a deep dive analysis but it's interesting to see how some things work. So in there on that github repo, click on lib and then you'll find a response.js file, now click on it, in that file you will find a lot of code. Now let's simply search for send and then an opening bracket and you will see how the send function, so basically a function we are calling here, how this is defined internally and this helps us understand it and this by the way 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 and we had that default header of text html right, so let's see what send does internally. It does a bunch of checks to see if we're using outdated versions of that function which we didn't in this course so let's ignore that and then down there, it basically analyzes what kind of data you are sending and you see that if it's a string data, so some text as we are doing it here, that in this case it sets the content type to html but only if we haven't set it yet. So it checks if the content type header is not present yet in which case it sets it, otherwise it would leave our default. If we have other values like a number, a boolean and so on, it would actually set it to binary or json data. So this is just some of the internal things it does and you don't need to go through the entire code here, it's a bit much and you're using express so that you don't have to do everything on your own but diving into this can help sometimes. Now one other interesting thing to see is that we can actually also shorten this code here where we set up the server. We can pass app to that create server method but instead we can also just use app and call listen and this will do both these things for us, something we can see in the official code if you go into the application.js file. In there what we can see is that if we search for listen here, the listen function in the end just does the two things we did before, it calls http create server and passes itself, so the app object which we previously also passed to that passes it to create server and then this in the end just make sure that listen gets called on that server object. So it internally does the same we did here and this of course save some code and now we can also remove that http import up there. So now our code looks like this and we'll still work fine, it restarts here and if I reload this page, this is looking good to me. So now we're using expressjs and you hopefully get a basic understanding of what it's doing and why this helps you write cleaner code because now you have a clearly fine structure, use this middleware funnel and you save code. The question of course is how can we now handle different routes as we previously did where we had slash message and slash nothing and so on and of course it would be nice if we now could also read incoming requests in an easier way. Well we'll do both and see both over the next lectures.

**Lecture 63**

**Handling different routes**

So let's see what else expressjs can do for us and let's also start building a more exciting application then. So for this, we want to handle different routes, different urls. To do that I'll first of all remove our dummy middleware which is not doing anything and this second one should only trigger for requests that go to just slash nothing. Now how can we filter for such requests? Well I mentioned that the use function here has multiple versions, you can see that here for overloads, so we basically have four different or five different ways of using that function. You can see a great explanation in the official docs in the end, on expressjs.com there I'm in the API reference for the version we're using here, the latest version and there you find app.use where you find the explanation of how to use it. Now don't be confused that this is only one definition instead of the five I've promised, the five basically just is made up of different combinations, so in the end, this is how you can use app.use. You got an optional first argument which is some path and that already is what we're looking for, this allows us to filter out certain requests, however this works a bit different than our if statements did before but I'll come back to that. Then we have the callback so basically the function that should be executed and we can have more than one of that callback, we can have as many as we want, we can also have multiple path filters here. Now you can obviously read more in the docs here but let's just use it and learn during the course. So we can add a path at the beginning, for example just slash, this however is the default by the way and now we would handle this for just visiting slash right. If I reload, we still see hello from express, Now what happens if I for example enter /add-product? We still see hello from express and we still see I'm in another middleware, so this middleware gets executed for both slash and add product because this does not mean that the full path, so the part after the domain has to be a slash but that it has to start with that. Now of course every route starts with just a slash and then we have different other criteria. So what we can do is we can simply duplicate this and add it before this middleware and add /add-product. Now why before this middleware and not after it? Because remember, the request goes through the file from top to bottom and if we don't call next, it's not going to the next middleware. Well I am not calling next here, so in the end if we have /add-product, this middleware will be reached first because top to bottom, add product will match this middleware and since I don't call next, this middleware will never get a chance of handling that request even though the filter here would have well, matched that request too. So here if I just add the add product page like this and I save this, you will see that on /add-product, we see the add product page and on any other path including random stuff or just slash nothing, you see hello from express and this is how we can use that middleware approach to control what is getting shown and the order here as well as the fact whether we are calling next or not matters a lot. By the way if you are sending a response, this is a good indication that you never want to call next too because you don't want to execute any other response related code just as before with vanilla nodejs, you don't want to send more than one response, this won't work and will result in an error. So this is the code we can use here, this is the code that allows us to route our requests into different middleware and if we have a middleware that should be applied to all requests, we would simply add it on top of all the other middlewares and then add it like this. If we don't add a filter or a filter that matches all requests it should match, then this middleware will always run first and if we call the next function, well then of course the request will also be able to continue. So if we have this middleware here which allows the request to continue and we have console log, this always runs well with this code. What we will get is that if I save this, if I reload here and I also go to add product here, we have this always run twice because well, this always runs, that's just how it works. So this is how the middleware works and how you can work with it to funnel your request into the right place.

**Lecture 64**

**Parsing Incoming requests**

Now we had a very close look at the whole middleware thing, let's now understand how we can actually work with incoming requests and how we can extract data and for that I again want to be able to handle a post request. So let's say on add product here, I want to actually return a html page with a form. For that I'll again return a form and just as a side note, this of course is a bit of an incomplete html document, we should also wrap this in html and body and so on tags. I'm keeping this shorter here to make it easier to read but later we will also write proper html code, no worries. So I have my form here and in there, I'll have my input of type title, of type text here with a name of title let's say and I'll add a button again and that button will be of type submit because it should submit this form and send a post request therefore and I'll simply give it a caption of add product. So let's simulate that this is a form that allows us to add a product to our own online shop or something like that. Now this is our form here and the form needs an action, so the path, the url to which the requests should be sent and let's name this product and the method should be post let's say, can also be written like this. So this will send a html code back which holds a form and now we need a route or a middleware that handles requests to product. So we can add app use/product, now the important part here is we can place that prior or after this middleware, they won't clash because they have nothing in common regarding the path, they have product in common but /add-product is different to /product, it just has to come before this one because otherwise this would execute prior to that. So this position here looks all right and then we again have our function which receives these three arguments, as a side note, you can of course omit an argument you are not planning to use, at least the third one you can't omit to the first one if you want the response because the order does matter but if you never use the third one, you can omit it but I always add it here to make it clear that it exists. So now with that, we have this function which we'll execute for product and in there I want to redirect and for now I want to log the incoming data to the console. Now what we can do here is for redirecting, I can use response redirect which certainly is easier than manually setting the status code and setting the location header. So redirect is another convenience function added by express and here I can redirect to let's say just slash, so it will automatically redirect me to the slash route. But of course this is not the only thing, I'm also interested in getting the body of my incoming requests, so extracting what the user has sent me and for this, expressjs now has a convenience feature for us. If I console log request body here, this is a new field added by express and let's see what's in there. So if I now save this, we should be able to go back to /add-product and hopefully see an input field and let's add a book here and hit add product, we're redirected to slash, this is working and in the console, we see undefined. Now let's get rid of the other console logs so that this is less clouded with logs, we can also remove that. So we see undefined and the reason is that we're almost there, request gives us this body convenience property here but by default, request doesn't try to parse the incoming request body. To do that, we need to register a parser and we do that by adding another middleware. and you typically do that before your route handling middlewares because the parsing of the body should be done no matter where your request ends up and there, I want to parse the incoming request body. Now for that we can install a third party package and we do that by running npm install --save because this will also be a package that is used in our code here, that does matter for production. So just save not save dev and the name is body parser. Now this would actually be included in express by default because the community wanted that again, it was in the past, then it was removed, then it was re-added, I will use that third party package which is the recommended way of using it because if they ever decide to pull it out of express again, this code I'm teaching you will still work. So now we installed a new package, the body parser and we can import that here, I'll store it in a body parser constant, the name as always is up to you and the package is named body-parser and now we can use that here by calling body parser, so using that object and then .urlEncoded. This is a function you have to execute and you can pass options to configure it but you don't have to here and now what this does is it registers a middleware, so this function in the end just yields us such a middleware function, so this parses such a function here in the end even though we can't see it and this package will in the end, in this middleware function call next in the end, so that the request also reaches our middleware but before it does that, it will do that whole request body parsing we had to do manually in the previous core sections. Now this will not parse all kinds of possible bodies, files, json and so on but this will parse bodies like the one we're getting here, sent through a form. If we have other bodies like files and we'll do that also in this course, we'll use different parsers and this makes expressjs so extensible. If we need something, we can just plug it in, you see how easy that is, it's one line of code, well two if you count the import then. Now with that, we should actually get an output for this console log statement. So now let's restart the server, by the way if you install a new package, you need to restart, you can't rely on the auto-restart from nodemon and we should configure one thing as I'm getting warned here, you should pass the config options here and set extended to false, this is if it should be able to parse non-default features you could say, so let's add this to comply with what we should use here and with that, we get the body parser enabled. Now let's try this again and let's go back to add product and let's add our book again, add product, we're redirected and now we see this is what we get, a javascript object with a key value pair which also makes extracting that value easier than we had to do before with the split function where we manually had to create that array and so on. Now we get an object where we simply get the key we defined in our input here, so this name and then the value the user entered and this is definitely simpler than our custom approach we used before and now we can work with all the data our users yield us, store them in the database, something we'll do later, show them in the response, whatever we need to do. Now one thing of course is missing. This right now would also execute for incoming data request, well we only want to listen to a post request, so what can we do regarding that?

**Lecture 65**

**Limiting middleware execution to post requests**

We're able to parse incoming request bodies with the help of the body parser package which is pretty neat but right now as I mentioned, this middleware always executes, not just for post requests but also for get requests, what can we do regarding that? Well instead of app use, we can also use app get. This is basically app use, it has the same syntax as app use, we can use a path or don't use a path but it only will fire for incoming get requests, so this is another form of filtering besides filtering for the path, app get allows us to filter for get requests and on the same page, we also got app post to filter for incoming post requests and just by changing this word, this middleware will now only trigger for incoming post requests with this path and not for get requests. So if I save this and I go to /product, you see I get hello from express, so I don't end up here even though I entered /product but it was a get request but if I send a post request through that form I have on add product, if I do this here, book too, you see we get this output, so we clearly made it into this middleware due to our filtering. So this is another way of using that middleware function, instead of use which will work with all http methods, we can also use get or post to filter for these. And additionally you also have delete, patch and put which are other http works which we'll use later in the course because we can't really use them from a normal html document.

**Lecture 66**

**Using Express Router**

We're nearing the end of this module because we already learned a lot about the core concepts of expressjs and this therefore is a crucial module because all the rest of the course will basically build up on this and this knowledge of how expressjs works. Now even though our dummy app here is really simple thus far, we're already putting all our code into the single app.js file which is therefore getting bigger. Now obviously for an app of this size, it's not a problem at all, it's still pretty small but typically we want to split our routing code over multiple files, we want to basically export our logic in different files and import it into this file. We could do this, we could create files where we export these functions but expressjs actually gives us a pretty nice way of outsourcing routing into other files. And for this I'll change our folder structure a bit, I'll add a new folder which I'll name routes. Now you don't have to name this, you could name it differently too but it's a convention you often see, that you put your routing related code, so your code that should execute four different paths and http methods, that you put that into files which you store in the routes folder and there since we're building or we're slowly building towards an online shop here, I'll have a route which I'll name admin.js because this should be the route that handles the creation of products which the admin of the shop can do. I'll also add another file and that will be shop.js, so basically what the users see let's say. Now I'll not build the full shop here, we'll slowly develop it over to the next lectures and modules because it uses a lot of cool features like databases and so on but we can at least start putting our code into these files here. The add product route and this product post request should certainly go into our admin.js file because these are routes that are reached by the admin and the general route here should go into our shop.js file, so that users that are visiting our front page see this route. Now one convenient feature offered by expressjs to achieve this is to go into these files and import express again, you can and you typically do import this into multiple files and then we can use a feature of it called the router. Now you can also create this with a lowercase r at the beginning, the name is totally up to you and I do create that router by calling express.router and this is a function I execute. This router is like a mini express app tied to the other express app or pluggable into the other express app I'll say which we can export here, so here I can use module exports and set it equal to the router. Now of course this doesn't do much, we have to use that router to now register things and actually I'll name this here with a lowercase r to be in line with my other names, this however has to have a capital case r. So now the router here can be used to again define a use function for all requests, a get function for get, post for post and so on So basically we can go back to the app.js file, cut these two admin routes from there, put them in here in the admin.js file and simply replace app with router here. Now the router gets exported, so the router now has these two routes registered because we exported here and this is the object on which we registered these routes, the other code can stay as it is because the router functions here basically work in exactly the same way as the app use function does or the app get and so on function does, I'll rename this to get though because I only want to handle get requests to add product and return this form and with that, with this exported here, we can now import that into the app.js file. Now for this, I'll add an import at the top separated from express to make sure or to make clear that this is my own file and I'll name it admin routes, the name is totally up to you but I do require a relative path to the routes folder and that in there, the admin file and you can omit the .js as I already explained, this will be added automatically. So now this is importing this router object here and this router object in turn has these routes registered and the nice thing is about this router that it is actually a valid middleware function. So we can take admin routes and just call app use and put our admin routes in there, just like this, not calling it like a function, so without parentheses but simply just the object itself, the router object we're exporting in this file. We can use this here and now this will automatically consider our routes in the admin.js file when filing the request through this middleware here. Now just before, the order matters so if we put this after this middleware, we will never reach that so this hasn't changed. Now we can do the same for our front facing route here, let's go to the shop.js file and there again, feel free to pause the video and try this on your own, try to implement this with the express router as we just did in the admin.js file. Were you successful? Let's import express first of all by requiring express, then let's create that router object by calling express.router as a function, let's export the router here and let's then use app use or paste in what I copied but replace app with router and maybe use with get. You don't have to do that, the use method would exist too but now we only handle get requests here. Now we can go back to admin.js, excuse me, to app.js and import all routes there too, the order of the imports doesn't matter, so my shop routes I require them from the routes folder and there from the shop.js file and now here again, the order matters, we should register this second. Now if I save this and I reload add routes, add product, this works. Now actually here's one important thing to understand, even if I would switch the position here and have shop routes first and I reload, it would work and we would not end up in this route but this only happens because I have get here. Get, post and so on will actually do an exact match here. If I would use use here as I did before to handle any incoming http method, then if I reload here we see hello from express again. So this exact matching is not achieved by using the router but because we use get here and that would have been the same if we stick to the app way of doing this in the app.js file we had previously. So get, also make sure that it's not just a get method but this exact path and therefore now if I enter some random stuff, I actually get an error because now I got no single middleware that would handle this but I do have my route set up here now and split up and then registered here and as I mentioned, it's not the worst practice to still care about the order here even though at the moment, it'll work fine no matter what the order is but if you ever change something back to use, it would matter and therefore why don't we just care about it right from the start.

**Lecture 67**

**Adding a 404 Error page**

Now one thing we did in the last lectures is we used the express router and right now, we have a set up where we also have some unhandled routes. If we enter some random string here, we get this error. Now typically, we would want to see a 404 error page and we can do that. Back in the app.js file, we can take advantage of the middlewares or the way express uses the middlewares and funnels the request through them. Remember that the request goes from top to bottom so if it finds some middleware that handles it, it will go in there and then for example here for slash with that get method on the router, we would actually end here, we also get no next call here so no other middleware would be executed. But if we got no fitting middleware and we don't have one here, then we actually make it all the way to the bottom and eventually we don't handle that request. So to send a 404 error page, we simply have to add a catch all middleware at the bottom where we don't need a path filter but we could add slash but that's the default anyways and then simply handle request response next since I use use here, this will also handle all http methods, not just get requests and there, I can then also send some code like Page Not Found. So little dummy html document and maybe we also want to set the 404 status code and you can do that by chaining another method prior to send and that is the status method and of course you cannot just use that here, you could have used it here or in the admin.js file too. So always before sending, you can call status to or set header, you can actually chain all these method calls, send just has to be the last one. So here I'm calling status to set my status code, another convenience method added by expressjs and I'll set it to 404 which is the common code for a page was not found. With this added, if I save this and then I now reload my dummy path here, I get page not found whereas for add product, I still get that form, so this is still working, the rest however changed.

**Lecture 68**

**Filtering paths**

There is one other notable feature about the express router that I want to show you, now we have a catch all route and we got our outsourced routes here. Now sometimes these outsourced routes have a common starting path, so let's say all the admin routes actually are triggered with admin/add-product and admin, maybe also add product, we can repeat the path here because we got different methods, get and post, so these will be two different routes too. So that is one important take away already, the same path can be used if the methods differ. Of course here in the form, I should also point at add product then but there is another important take away. If we have such a setup where our paths in such a router file start with the same part or with the same segment here, we can take that segment out of this route here and then go to the app.js file and add it here, so add that segment as a filter. Now only routes starting with /admin will go into the admin routes file so to say and not only that, it also will or expressjs will also omit or ignore this /admin part in the url when it tries to match these routes, so now /add-product will match the /admin/add-product route because /admin was already stripped out here you could say, let me show this to you in practice. If I reload my add product route, we get page not found because this does not exist anymore, this is now /admin/add-product and indeed, here is the form and if I now add my book again here and hit add product, I get page not found and the reason for that is that of course here in form action, I'm leading to /add-product but this should be /admin/add-product too because we want to reach that route which is the admin.js file which is only reachable through requests that have /admin at the beginning. So let's give this another try, let's go to /admin/add-product and let's try adding that book again, now we are redirected and now we can also see that we are logging this here. So this filtering mechanism here in app.js allows 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 that we don't have to repeat it for all the routes here. Implicitly, this route is reached under admin add product and so is this route here, this one with a post request and this one with a get request. Now this can be a bit challenging to wrap your head around but this is another core thing you have to understand, how these requests are funneled through and how they may reach this file if they start with /admin because we are filtering here and how in that file, the /admin part is then not checked again but it only checks the second part and therefore reaches this or these requests route here.

**Lecture 69**

**Creating HTML Pages**

So now that we had a very close look at that whole express middleware thing and how routing works in express and how we can use that to our advantage, how we can filter routes and so on, let's actually start working on what we serve to the user. Thus far it has always been some dummy html content but you're probably not in the course to just learn how to build dummy html content, so let's return some real html files to the user that also don't look like crap. For this I'll create a new folder in my project and I'll name it views, now that name is up to you, you can also name that folder whatever you want but in that course, we will slowly go towards a MVC, a model view controller structure and I will explain what this is a little bit later. One part of it is that we manage our views, so what we serve to the user in one place of our application in the views folder here. Now the views will just be a bunch of html files here though. So I'll create a new file and that will be my shop.html file, it's a file I want to serve for users visiting just slash and I also want to have my add product file here. So I'll add add-product.html, later by the way in case you already know the concept of templating engines, we'll use these two so that we can dynamically add content into the html files but for now, let's just start with these files. So let's start here in add-product.html and now here's one important note, if you're not interested in writing that html code, you can skip this lecture now and find the finished html code attached to this and the next lecture. So if you want to skip, you can do that and just follow along in the next lecture where I will provide that finished html code otherwise let's now create it together. So in this add-product.html file, I'll now create a new html5 skeleton and visual studio code helps you with that, as you saw if you just type html5, it should open this pop-up, if it doesn't you can force it by typing or by hitting control and space and then navigate to html5 with the arrow keys and hit enter and it gives you this nice skeleton which basically defines a basic, well html skeleton. Now here I'll change the title to add product and in the body, there I now want to have my form. I don't just want to have the form in there though, I also want to have some navigation bar that allows me to switch to my shop.html page, to the slash route and the other way around. So here I will first of all add a header and in that header, I'll add a nav bar and in that nav bar, I'll add an unordered list with list items which are links where I go to slash, so this is just my shop and then another list item, add product which will be /add product. This is the page we're on here but I always want to show both options, obviously you can write the html code that fits your needs. So this is the header, it will be pretty unstyled for now, we'll add styling too and now let's add a main section too, so this is also a normal html element which holds or which should hold the content of our page here. And there I want to have my form with the action that is also add product, remember here we're then targeting this post route and in there or there, I will add my post method and I will now also add my input here, the input of type text with a name of title and I will add my button of type submit here which I'll label add product. Now we will add more to this form later because a product is obviously not just made up of a title but for now, this will do. We got our form in here, let's now copy that entire html code and paste it into the shop.html file and there of course in the main section, I don't want to have a form instead a h1 tag my products and below that later in the course, we will render all the products. So here we will have a list of all the products and right now we don't have that but we'll get there once we learn how to manage data on the server and so on. Now the rest of the page can stay the same for both pages and again as I mentioned, styling is missing. Let's now move onto the next lecture where the goal will be to serve these html pages before we then work on the styling.

**Lecture 70**

**Serving HTML Pages**

So welcome back, in case you skipped you'll find these two html files attached to this lecture, so make sure to download them and enter them, insert them into your views folder and now the goal is to serve them. We didn't work on the js files in the last lecture, we just added the html files and make sure to explore them to understand what they do, in the end we got the same content as before, we just added an extra header and then in add product, we still have the form, in shop.html we got some dummy code since we have no products to serve yet and now I want to serve these html files in my routes. Now how does that work? Let's start in the shop.js file, instead of sending some text or this html text here in this case, let's instead send a file with send file and send file well allows us to do just that, send back a file to the user and as you see here in description, it automatically sets the content type response header field and we'll see if that works for us or not. So send file is what I execute and now we just need to point at that file we want to send. Now here, the question is how does the path look like? The file is in the views folder but how should this path now look like? Well we could try using slash and assume that we see all of that from the view of the app.js file which is in the end the file which starts our entire server, the fact that shop.js in in a subdirectory doesn't really matter because we export its functionality and import it into the app.js file which is in the root folder. So we could try using slash for the root path, an absolute path seen from the root folder and then views and then shop.html, like this. Let's give this a try, let's save this, go back to the page and reload localhost 3000 slash nothing and I don't see that. Well the reason for this is that this path is incorrect, let's try ./ here, if we now reload, path must be absolute is the error we get. So whatever we tried, this doesn't seem to work, the reason for this is that an absolute path would be correct but slash like this actually refers to our root folder on our operating system not to this project 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 can import the path core module by requiring path like this and then here, we send a file where we create a path with the help of this module by calling the join method, join yields us a path at the end, it returns a path but it constructs this path by concatenating the different segments. Now the first segment we should pass here is then actually a global variable made available by nodejs and that is the underscore underscore and that's important, these are two underscores dir name. This is a global variable which simply holds the absolute path on our operating system to this project folder and now we can add a comma and simply add views here because the first segment is basically the path to this whole project folder, the next segment is that we want to go into the views folder and then the third segment will be our file, so here shop.html and don't add slashes here because and that's important, we use path join not because of the absolute path, we could build this with dir name and then concatenating this manually too but we're using path join because this will automatically build the path in a way that works on both Linux systems and Windows systems because as you might know, on Linux systems you have paths like this and I'm not talking about paths in the url but on your file system now but on Windows, you use backslashes for your paths and therefore if you manually construct this with slashes, it would not run on Windows and the other way around. Path join basically detects the operating system you're running on and then automatically builds a correct path. Now with that, we could expect that it works but actually dir name here will point in this routes folder, right. Dir name gives us the path to a file in which we use it and we're using it in the shop.js file in the routes folder, so this will point to the routes folder but views is actually located in a sibling folder to routes. So what can we do regarding that? Now the solution is that we add one more segment in there and that is ../ and this simply means go up one level, so this will now build a path where it first goes into the folder of these files, so into routes, then it goes up one level then into views, so if it's up one level it's in the root folder then into views and then it serves this and with that if we now load localhost 3000/ again, we see that html file being served. And now is a great time for you to pause the video and ensure that you serve add product when this route gets loaded. Were you successful? Let's do it together. For this, let's first of all import the path module again, the core module, so const path require path to pull that in. We don't need to install that because it is a core nodejs module and then here, we don't use send but send file and we will then use path join, the dir name variable to get the path to these files folder and then we can go up one level and then into views, whoops, should be a string, views and then we want to serve the add-product.html file. With this if we save that and we head over to add-product, whoops that should be admin/ add-product, we see this page too. Now the styling is missing because we don't have any but this works and we can also check on add-product that the correct content type was assigned by express. So this also works and now we see how we can serve simple html files for the different routes we have. Now here's one bonus task for you which I want you to solve and which we'll solve together in the next lecture, add a new html file which is your page not found page which you then serve if we ever reach this middleware function.

**Lecture 71**

**Returning a 404 page**

Were you successful? Let's add it together. I'll add a new html file in views and I'll name it 404.html but the name is totally up to you, well it has to be a html file of course. I'll then create a new html5 skeleton with the help of visual studio code here and then here I'll name this Page Not Found, that sounds like a fitting title and now you can get of course really creative here, I will in the end just add an h1 tag where I say Page Not Found, obviously you can add more content here if you want. Now I want to return this html file whenever we make it into this middleware and therefore again, I should send a file here but obviously I want to construct the path with the help of the path module to make it work on all operating systems, so require the path here and then go down there, send and don't send but send a file and that file will use a path which we construct here with their name and now here's a gotcha, we already are in the project folder here because we're in app.js, so we don't need to go up a level here because we already are in the root folder. Instead here we can go right away into the views folder and then serve the 404.html file, like this, we also still want to set this status code because we still have a 404 error. If you now save this and you enter any random route that doesn't exist, you should see that page not found being served and you see that 404 error being sent back too. So this is now working too and and this is now a way better, well way of serving this all but styling would also be nice, wouldn't it? So let's work on this over the next lectures.

**Lecture 72**

**Using a helper function for navigation**

Before we work on the styling, let me add one note on how we navigate it to the root folder in shop.js and admin.js. For one you could just use dot dot here instead of dot dot slash and this would be preferable even though both should work on Windows and Mac because now we make no assumption about the separator we're using when constructing a path. So with this, if I go back and I now go to just admin add product here, it still works but now we get a cleaner way of doing this but there is an even nicer way we could implement this. We could also get the parent directory with the help of a little helper function. For this I'll create a new folder here, helpers or util, I'll go with util, you can name it however you want and there I'll add my path.js file and it doesn't matter that this clashes with the global module because we'll import it differently anyways. Now there, I'll add an export with module exports and I want to export a little function that helps me construct a path to the parent directory or not a real function, instead a variable I should say. First of all here, I'll import the path with require path as we did it before and then I will use a different function here, not join but dir name. Dir name as you can see in their quick help on the right here returns the directory name of a path, so this sounds pretty useful, if we use that we just have to find out which directory or for which file we want to get the directory name. Well there we can use the global process variable, that is also a variable that is available in all files, you don't need to import it and there you will have a main module property. This will refer to the main, well module that started your application, so basically to the module we created here in app.js and now we can call file name to find out in which file this module was spun up. So put in other words, this gives us the path to the file that is responsible for the fact that our application is running and this file name is what we put into dir name to get a path to that directory. With this we can import from this file, here I'll add my own import separated from the other ones simply to make it easier to identify and I'll name this root dir, the name is totally up to you but I will require this from going up one level into util and then path and this root directory is in the end what I want to inject here. So root directory is what I'll start with when creating this path. And now let's try reloading this page here and it still works and it should still work because now we're in the end having a pretty neat way of constructing a path to our root directory. I'll do the same in shop.js, import root dir from the util folder and the path file in there and replace dir name dot dot with root dir. Again you could have totally sticked to the old approach but this one is a even cleaner one and one that should be pretty straightforward to use and that should work on all operating systems and it always gives you well the path to the root file. With that, let's move on to the styling.

**Lecture 73**

**Styling our pages**

Our html pages look pretty boring and obviously this is no css course, I got one if you want to learn more about css though but that's just a little side note but this is no css course but still, it's important to understand how you can serve css in your node apps too because typically, you have css and also javascript code in your apps. Now let's start simple, I'm on the slash page here, so on shop.html and let's add some styling here with good old style tags in the head section and this is not the way we'll keep it, this is just so that we can work on it and auto-update this because importing an external style file wouldn't work at the moment to spoil the fun already. Now as before, you can skip this lecture if you don't want to write the style together with me and you'll find it attached to the next lecture, otherwise let's do this together. So let's start styling this and let's start with the header. Now I won't target the element directly instead this will be my main navigation or my main header, let's give it a class, main header header, this will be the main header nav and if you're wondering about the strange css class name, I'm using a styling system named bem, more on that can be learned in that css course I mentioned or of course you can google css bem, it's a certain way of naming your classes to avoid double class names basically, here I'll have my main header, whoops, header item list and here I'll have my main header items like this and you can of course name this however you want. So these are a bunch of css classes that should help us with styling, now let's start with the header. The header should have a width of 100% to take the full width of the page and let's say it should have a height of 56 pixels or if you want to work in rem which has certain advantages, we could also go with 3.5rem. Now let's also give it a nice background color and here you're really free, I'll not stick to red but now I can hover over this and use this nice color picker and pick any color you want, now I'll try to pick a nice yellow golden-ish orange color, something like this maybe, if you click on rgb it will also convert this to a hex code so let's use that and of course the class is not header but main header. And now if I save this and go back and reload, we got that, looks better. Now there is some padding in the body so let's actually define that we want to have no padding and no margin, I think it's a margin, that we don't want to have that, let's add this, looks almost better. We still got some margin here at the top, this is coming from our list in here though, that some margin call isn't doing its job here, now let's also get rid of that. By now also styling our main header item list. So let's copy that, add a class here, set the list style to none and remove any margin and padding we might have here, that looks better. Now obviously the items are not looking nice, for this I will actually change the display here to flex to use flexbox for the list and then style our items here a bit, on these items for now I'll remove all margin and padding they might have, reload. Now thanks to flexbox they're sitting next to each other, now let's also style our links in there. So main header item, any nested anchor tag should have no text decoration and a color of white maybe, looks a bit better. Now vertical centering would also be nice and for that I'll go back to the main header, excuse me, back to the main header nav here, let's add a class for that too. And here I'll first of all use a height of 100% because this is a child element of the main header and therefore it should take all its height and this will also be display flex now and we can center items vertically with align items centered now. With this if I save that and reload this page, now this is centered, now let's add some padding to our main header. For this let's go here, add a padding, top and bottom I don't need padding but left and right maybe 1.5rem, something like this, on mobile phone also still ok I'd say. Let's also change the global font by the way so in the body I'll set the font family to sans-serif for now, looks a bit nicer than the serif font I think and now I want to have some space in-between my navigation items there so let's actually go to the main header item and let's add a margin to the left and right of 1rem, like that and for now, this will do. Now little hovering effect for the links maybe, so let's also go here, main header item hover and main header item active and let's also define a rule which would apply if the anchor has an active class attached to it and there I want to set the color to well we have this nice little yellow touch here, so we could go into the blue-ish direction here, maybe like this purple-ish color, let's let's see how that looks like. Not too shabby, so let's go with that and now since we have that active class rule here on shop.html where this link is active, we can actually add the active class here and you will learn how to set that dynamically later in the course too of course. So now this is active, now for the main section let's also style this. Here I will define a general rule for main that I want to have a padding of let's say 1rem in all directions and now this is the style I want to go with. Now let's also of course work on the admin add product styling here and we can simply copy the entire style object from the shop.html file in the add-product.html file and this will already get us pretty far. Now one thing we should change in add product, we should of course add all these classes, so we can basically copy that header and later we will also learn how to reuse it across files by the way. So let's copy that header, by the way that auto-formatting I'm doing occasionally can be set up in the preferences by going to the keyboard shortcuts and searching for format document, this is the shortcut I am using here but that's just a side note. So here we now also have to change the active class, it's not on the shop link here but of course it should go onto the add product link and with that if I now reload this page, looks way better. Now let's work on the form here too and for this, I want to go to my form, give it a class of product form maybe, the name is up to you, add a little div with which I give a class of form control and in that div, I'll insert this input here and I'll also add a label for the title, put title as a label on it, give that input the ID title so that it matches the label for accessibility reasons simply and now let's style this too. For this I'll use my form control, so this class I added to this rounding div and here, I want to make sure that a label as well as an input which is nested in a form control uses a display of type block so that it takes the full available width and I also want to style the input then, so the input here should have a border of one pixel solid and then I will reuse that yellow color we used for the header. The font should be inherited so that it uses the default font family and so on and I'll give it a little border radius of like two pixels maybe, let's see how that looks like or the radius. If I now reload this, doesn't look too bad, let's now also work on the button. For this I will set up a generic button style here for now, inherit the default font, give it a border of one pixel solid and this light gray a background of white maybe, let's maybe give it a black border here simply and a border radius of three pixels. Now let's see how that looks like, it's ok but let's reuse the yellow color here actually and let's also give the text that yellow color. Additionally I want to make sure that we assign the pointer cursor to get this hand cursor when hovering over it and last but not least, I want to have some spacing but with my form control div and the button. So on the form control itself, I'll add some margin to top and bottom of let's say 1rem and 0 left and right so that this looks like that and I think for the button, maybe you want to go with the purple color since that should be easier to read and let's also assign a hover and an active state for that button and when it is active, I just want to set the background color to purple and the text color to white then. So now it looks like this and this doesn't look too shabby, we can go with that I guess, we can always work on this and we will continue to work on this throughout the course but for now this is good. Let's now center the entire form and for that I assigned the product form class to the form. So here I want to give this a width of let's say 30rem, a max width of 100% or of 90% in case we're on a mobile view, give it a margin of auto, this will automatically center it horizontally and now just make sure that the input and so on take the available space so let's reduce this 20 actually, I think 30 is a bit too much and on the form control for laying an input I'll give it a width of 100%. and now this looks much better. Now I'm not happy with the white text here now that I think about it so let's actually go to the header and change the white link text to black, let's see how that looks like, yeah I think I prefer that so let's actually take that and also add it to shop.html so that we have the black color there too and I think the purple could be a little bit darker. So let's now also go to the hovering and active style here and make this a darker purple and replace the purple in all occurrences with that color style. I'm using a multi-select feature of the IDE here to do that quickly, I can of course also search and replace to do that, reload, yeah I think that is a bit better. Now you can always tweak this to your needs, it's not a css course here. I just want to have some basic styling that we can use. So this is the style with which we'll go from now on and which we'll of course continue to tweak. Now one thing we should also do now, let's copy that style from the shop.html file and put it into the 404.html file here because there, I also want to take the header and copy that into there because I don't just want to have the page not found thing, I also want to show the header. So that now if I do enter some route which does not exist, like this, we also got the header there and we can at least leave that route, though we can't because I obviously got an error in the url, that should be admin flash add product. Now let's fix that everywhere here including our form of course, like that and now if I reload, we can navigate around just fine here and try adding a book, that seems to work. And with that, we get the styling in there, we get the basic html markup but of course regarding the styling, the issue is that it's all in our html files. It would be nice to have some external files for that, wouldn't it?

**Lecture 74**

**Serving files statically**

In the last lectures I added some styles and please just take these three html files which you find attached to this lecture and replace your existing views with them because I don't just, I didn't just add some styling, I also added some css classes. Now the page will look something like this, you can tweak the styles if this is not your style, obviously this course is about the logic, the nodejs logic and we will continue working on this project, the html and the styling too. One issue we have though is right now all the styles are defined in the html files and I want to use external css files. Now the problem is right now we can't easily import them but let's see how we at least theoretically would want to do that. Now typically, you would have some css files somewhere and would point at them when your app gets served, now you can create a new subfolder and you can name it whatever you want but the convention is to call it public because you want to indicate that this is a folder that holds content which are always exposed to the public crowd or which is always exposed to the public, so where you don't need any permissions to access it and that's important. All your files here are not accessible by your users, if you ever tried to enter localhost and then something like views, shop.html, that will not work because this is simply accepted by express and it tries to find a route that matches this. It tries to find it here in app.js basically and also of course in shop routes and so on. It doesn't find that route and therefore it doesn't give you access, you can't access the file system here and that is of course good and what you want. But now I actually want to make an exception, I want that some requests can just access the file system because ultimately let's say in shop.html, I want to have something like a link in here where I simply point at something like css, main.css, anything like that and my imagination would be that in public, I have a css folder with a main.css file in there and that is the file I want to serve with this link. Now right now, this wouldn't work but let's take that code already, that style code, cut it out of shop.html and move it in there because that pretty much is the main styling we use in all our pages, the header and the body. Let's remove the style tags here and obviously if I now save and reload my main page, all the styling is gone now because it can't find the main css file as far as you can see here in the developer tools because we can't access the file system. Now you could say yeah the path is incorrect right, it's public css but even if I change it to this path, you will see that if I reload this file, it will never work and now it does look in the public folder. For this we need a feature expressjs offers us, we need to be able to serve files statically and statically simply means not handled by the express router or other middleware but instead directly forwarded to the file system. And for this, we register a new middleware with app use and this this one expressjs ships with, therefore we use the express object itself, so this object we're importing there and this does have a static method and this is a built-in middleware as you can read on the right, it serves static files. So we can execute this function. and now we just have to pass in a path to the folder which we want to serve statically, so basically a folder which we want to grant read access to, it's only read access but that's still more than what we normally have. And here again we can construct this path with path join and then simply our dir name, so our root folder and then public because I want to grant access to the public folder in our current folder here, so in the dir name, so in the root folder. With this, user should be able to access the public path and now if I save this and I reload shop.html, still doesn't work. The reason for that is that the path with public at the beginning here is wrong. Instead here, we should omit this and directly act as if we are in the public folder already because this is basically what express will do here, it will take any request that tries to find some file and that's important, it looks at the extension, so anything that tries to find a .css or a .javascript files, if we have such a request, it automatically forwards it to the public folder and therefore then the remaining path has to be everything but that public, so therefore we strip the public out of this path and just act as if we already are in the public folder because this is where file requests will be forwarded to. So now if I reload here, this looks much better right, now we find that main.css file because now this path can be resolved because we request a file here and if I omit .css, it therefore will fail but if I add it again, this is handled by the static middleware and forwards the request to the public folder. And by the way, you could register multiple static folders and it will funnel the request through all of them until it has a first hit for the file it's looking for But here I'll just go with the public folder. Now with that, we can of course take that link and also add it in add product in our head section, now there we just have to watch out in the style text, I got more than what I set up in main.css so I'll also add a product.css file and move the extra styling code which begins here with product form, we'll move all that into product.css and the other part in the style tags here can now simply be removed and therefore I now also have to add an import here to css, product.css. And in the 404 page, I of course also want to import my main css code and there I got no extra code, so we can remove the style tags and simply import the link here. Now with that if we save that and we reload this page, it works, add product is looking good. Let's add a book here, looking good and let's visit a route that does not exist, also looking good regarding the styling at least. So this is now how we can serve files statically and you're not just limited to css and javascript files, you can also serve images and so on. So this is the last important piece for now, how you can serve content statically, in the next modules we'll then dive deeper into actually doing something with that user data we can submit there and we'll dive into a very interesting concept, the concept of templates because that will allow us to turn our static html code, so basically the code which is hardcoded which doesn't have any dynamic element into more dynamic elements where we can inject data we have in our javascript code in the html templates we return to the user. Pretty interesting, we'll dive into all of that in the next modules.

**Lecture 75**

**Wrapup**

That's it for this module. It was quite a lot, a lot about middleware I guess but it is the crucial basis for the rest of the course where we will of course continue with nodejs but by using expressjs, a powerful framework and that is what expressjs is. It's a nodejs framework, so you still use nodejs, we for example still use the path core module in this module but you build up on it and get a bunch of utility functions and a clear set of rules on how you should structure your app and it's all about middleware and understanding that flow of requests through all the middleware functions. Expressjs is very popular because it's highly extensible and as you already saw with the body parser in this module, you can easily plug some packages, a lot of packages actually into an express app because they just expose such middleware functions and you can therefore easily add them and the request gets funneled through them. Now that concept of middleware is really important, middleware functions are these functions that took the request object, a response object that helped you with sending a response and that next argument which turns out to be a function you should call to forward a request to the next middleware in line and in line means from top to bottom in your root file. This is a crucial point you have to understand, that you should always call next unless you are sending a response in which case you should never call next and that you can therefore cleverly structure your middleware to transform a request, read something from it and send different responses depending on the route you're accessing, so depending on the path and method you're sending. You learned that you can filter request by path and method easily with app use by adding a path or app get, app post and that if you filter by method, like if you had app get, that the paths would then be matched exactly otherwise with app use, the path you passed would only be matched with the beginning of the url, the part after localhost. You also can use the express router package instead of app use, app get because this allows you to elegantly split your routes across multiple files since the router you export there can be added as a middleware function into app use in your root file. Last but not least, we also served some files and it's important to know that you're not limited to serving dummy text or anything like that, you can send files for example as we did it with html files and if a request is directly made for a file like a css or javascript file but also images even though we didn't see that in this module, you can enable static serving for such files with the help of express static which is a crucial part of any web application you're building because you typically have such files that are dependencies of your html files for example and this is the core basic knowledge you should have about expressjs, this is what we'll now build up on and this is where we will now dive deeper into to learn how to render dynamic content, how to access databases, enable authentication, manage data on the server and so much more. So let's continue.