**Lecture 77**

**Module Introduction**

Now that the core basics about nodejs and expressjs are set, it's time that we work more on what the user sees because ultimately, we obviously want to build an application which delivers some value to our users, right? Now thus far we're always returning static html pages and typically this is not what you do in real applications because you don't just have static html code most of the time, instead it is pretty common that you have some data managed on your server, later we will also do that in a database, that you have some data on your server which you want to dynamically output in the html code you send back to your users. An example would be a page with a list of products where these products of course are coming from your database, so from your server. Or for example, we already have a very simple form where people can submit a new product title, well right now we're not really doing anything with that, in a real application you would obviously want to store that and then kind of return it in some other page where you have that list of products I already mentioned. So in this module, we will have a first look at how we can start managing data on a node express backend, for now without a database, no worries there will be a very extensive module about the databases in this course so you will learn how to interact with a database too but for now, let's just manage data like this and let's focus on something else here, on rendering dynamic, for now dummy content in our views. So these html pages which we're returning should now become more dynamic and actually contain some content that is dynamically entered into them on the server, so that if we had additional data on the server, we would send back a different html page with different content. And for this, we'll use something called a templating engine and you will learn how such templating engines, there is more than one alternative and I will present some alternatives in this module, you'll learn how they work and how you can use them. So this is what's in this module and with that, let's dive right into it and let's have a look at how we could manage that data before we then have a look at how we could output it.

**Lecture 78**

**Sharing data across requests and users**

I'm back in the project and I actually changed the html code and the styles a little bit, you'll find all css files and html files attached and just make sure to move the html files into the views folder and the css files into the css folder in the public folder. I did this to well change the styling a little bit and also add a little bit of additional markup which we'll use in this module but no worries, I will walk you through all of that markup and regarding the styling, it's just a little bit nicer to look at right now and I also added some styling which we'll need in this module. So with that, I can of course visit the page on localhost 3000 and this is how it looks like right now, a little bit nicer than before in my opinion but of course you can always style this to your needs but let's now focus on the data. And right now, we don't really work with data in our app right in admin.js for example, we do get that data for a product here in a host route but we just log that to the console, we're not storing it, we're not working with it and working with it is kind of hard right now because we have no database where we could store it permanently but one thing we can of course do is we can store it in javascript variables and see how that works and if these are then shared across incoming requests from different users and that of course will hold some important learnings because you often well don't want to share such data. So let's see how that works before we then later in the course move towards a more permanent database driven solution. So let's say the incoming product title which we output here should be stored in a more permanent place and in general, I actually want to also add some fields to the form then later so that we can add more for a product than just the title but step by step. So let's start storing that title which we get here and just to bring that back to memory, right now we got this field here and if I submit this, well then I get this object with the key title and the value the user entered. So how can we store this? Well we could add a variable where we store it in and the first thing we could try is we could add a variable here in admin.js, let's say we create a new constant here which I'll name products which is an array and keep in mind even though it's constant, the array can receive new elements because the array itself is still the same object, we just add or remove elements to it but that doesn't affect the overall holding object. So now I got my products here and I actually want to export my products, so what I'll do down there is I will use a different syntax where I export my routes and export a router here and exports products and export my products constant. This has one important implication, since I changed the way I export my routes, I have to go to the app.js file and with that in the app.js file where I import my admin routes here, well actually this is the admin data and there, I want to access the routes object because there will be such a routes object because I'm creating it here. So admin data refers to all the exports you could say and there we now have routes and products and therefore when I do import my routes, I have to import them like this, of course admin data is then also something I have to change up here in the import. So now I got this adjusted and now I got my products exported too, products is an empty array. Now in here, in router post let's take the products and push a new element into this array, a new object let's say and that object will have a title which is the title I'm getting, so request body and keep in mind that also is an object with the title property so I will extract the title with the dot notation, I could of course just push the overall request body since that will be an object of the exact same structure but later I want to add more fields here and therefore, I will create a new object here, also to make it a bit clearer to see what's happening here. So now we're adding this to products, now in shop.js where we output all our products or where we want to do that at some point, we therefore need to get access to the products and for this, let's add an import up here, let's import admin data by requiring admin, so this admin.js file where we do export its routes, something we're not interested in in this file but also this products array, so now here let's console log admin data products so that should be the array. Now save everything and now let's simply see what we get. If I reload this page, the shop page, I get an empty array which makes sense because initially, this is an empty array, we export it, that makes sense. So let's go to add product now and let's add a book here and it clicked add product, we're back to the shop page and we see something interesting. We see the array with the book inside of it and that console log statement, where is this actually coming from? Well that is coming from the shop.js file here, we can also make this clearer by adding shop.js here and logging the products, so logging two things to make it clearer what is responsible for the output, so shop.js is logging the empty array and now let's try outputting that book here and now we got shop.js with the array with the book in it. Now this is interesting to see, so we can export something, some object or array, a reference type therefore and if we change that in the other file, it also gives us the update here. So this is interesting, this is one way of sharing data and to be honest we'll later use different ways because this has one disadvantage. Here if I reload shop, we still got that in there. Now let me open up a totally different browser, I'm in Firefox here and I also visited localhost 3000. So this is a totally different browser and this is kind of like a brand new user, it doesn't share any cookies with the other browser, nothing of that kind, it used the same IP address but that doesn't matter here. It's a brand new request as if it were made from a different machine and you will see, I still log this, so this is actually data which is inherent to our node server as it is running and therefore, it's shared across all users. Sometimes this is what you may want but very very rarely to be honest, actually you will probably never implement this, you always want to fetch data for a specific request and if that happens to be the same data you show for all users that send this request, this is fine but sharing this data across requests, across users is typically something you don't want to do because if you now edit this with user A, user B will see the updated version even though you might not want to show that. Maybe it's added that normally it wouldn't have been saved to the database yet, so you don't want to show that to the other users yet, maybe it's some personal data. So this is a pattern we can use for now here and it's fine for practicing what we want to practice here but later we'll learn about a technique to share data in memory here, in the node app across different requests but only for one and the same user and not across users because now we have shared data across requests and across users and we will later of course also learn how to use a database. But for now let's stick to this approach, let's use that for some dummy data sharing and let's see how we can get this data into our view now.

**Lecture 79**

**Templating engines – an overview**

I already mentioned at the beginning of this module that for putting dynamic content into our html pages, we would use something called templating engines. Templating engines work like this, we got a html-ish template and with that I mean that you typically write some code, some file that contains a lot of html, your html structure and markup, your style and javascript imports, all of that is typically included but you have some blanks in there, some placeholders. And then you have your node express content in your app, like our dummy array, our products array we're currently using and you've got a templating engine which understands a certain syntax for which it scans your html-ish template and where it then replaces placeholders or certain snippets depending on the engine you're using with real html content but that content, this html content it uses there is generated on the fly, on the server by the templating engine taking that dynamic content into account. So for example you could output anun ordered list with list items for the data you have in your node express app with the help of the templating engine and in the end, the result will be dynamically, on the fly generated html file which is then sent back to your users. So the users never see the template, they never see the placeholders, all that happens on the server, they just get a normal html page but it's not hardcoded by you as it currently is in our project but instead, it's generated on the fly. Now we got a couple of different available templating engines and actually, you get even more options than I'll show you here, at the end of this module, you'll find some useful links to dive deeper into them and learn about more alternatives. The three options I want to present to you are the three most popular ones, we've got ejs, pug formerly named jade and handlebars. Now these are free templating engines that use a different syntax and different set of features, different philosophies that you can use to well create these templates, inject your dynamic content and get html files out of them. Here's a brief, a very brief look at how their syntax looks like but I will also present all three of them in this module. Ejs looks something like this, you write normal html markup but then you also add like this smaller than percentage sign and then sometimes an equal sign, sometimes not depending on what you're doing, you will see that in this module and then the dynamic content you want to output. So if we had some name variable being injected into our template and you will learn how that injection works, if we got that then the value of our name variable would be output there and we would send back an html file with a paragraph tags and the value that was stored in name between them. Pug uses a different syntax, it doesn't use real html, it replaces this with a minimized version or a minimal version and then it also allows you to output dynamic content with this syntax, the hashtag curly brace syntax for example. Handlebars in turn uses html again but then you have the double curly brace placeholders for the dynamic content, so similar to ejs, actually handlebars has a bit less features available you could say or it follows a different philosophy but it's closer to ejs, pug is the well the outlier here which also uses a different html syntax. So a short summary would be ejs, normal html and then actually you got these placeholders which allow you to just use plain javascript in them actually, so you can also write if statements, for loops. Pug use as a minimal html version and a custom template language which is extensible but generally offers only a set of things, of operations you can do but if statements and lists would be included, iterations would be included. Handlebars uses normal html but also a custom template language with a limited set of features, again including common things like if statements or lists. So these are three popular templating engines, now let's briefly dive into them, how we install them and how we use them before we then pick the favorite for this course and stick to that favorite and again at the end of this module, you'll find more resources on all these engines.

**Lecture 80**

**Installing and implementing pug**

So back at the project, let me quit the development server because now I need to install another package and I will simply install these three templating engines so that we can work with them. So let's install them with npm install --save because all three engines are part of our node code and ship with the code we deploy on some computer in the end, so let's install them as production dependencies and there we need to install EJS, that is the package name and links to the websites of these packages with more documentation can also be found in the last lecture of this module. So EJS, then we also got pug and handlebars, there is a handlebars package but that's actually the wrong one, we need express-handlebars here because this has built-in integration with Express, EJS and pug already have that built into their core packages you could say. So let's now hit enter and now these three packages will be downloaded. Now what you can also see here by the way is that you can of course install multiple packages at once by simply repeating or by adding all the names after npm install, now the packages are installed and we'll not use them all at the same time but one after another so that we can have a close look at how they work and let's start with the most exotic package, pug. To use that, we have to go to the app.js file and now we have to let expressjs know and that is an express feature, not node by the way, another strong reason why we want to use express because with node, standalone node, this will be harder, you would have to do all that manually, here we can just tell express hey we got a templating engine that is express conforming and that is the case for all three we installed so please use it to render dynamic templates. We do that by going into the app.js and after we created our express app here and stored it in the app constant, we can set a global configuration value. Now what is that? App set allows us to set any values globally on our express application and this can actually also be keys or configuration items express doesn't understand, in that case it just ignores them but we could actually read them from the app object with app get and this would be another way of sharing data across our application but not really something I'm interested in here. What we can do is we can use a couple of reserved key names here, so configuration items we can set that do lead to expressjs behaving differently and you see a list of all these items here in this table. Now most of them don't really matter for us here but feel free to browse through that, interesting for us is the view engine and the views key. View engine allows us to tell express hey for any dynamic templates we're trying to render and there will be a special function for doing that, please use this engine we're registering here and views allows us to tell express where to find these dynamic views. So what we can do here is we can app set and set the view here, view engine to a string, pug. Now you can't enter anything here, we use pug here because we installed the pug templating engine and this engine actually ships with built in express support and auto registers itself with express so to say. So that is why this works, it doesn't work for all engines but you'll find more in the links in the last lecture, here it does work, pug is supported out of the box and with that, we're already set to go. We can set an additional configuration though, we can let express know where to find our views, however the default setting here in this last column, the default setting for views already is our, basically our main directory and then the views folder, still I'll send it explicitly here to show you how this would work if you would store your views in another folder which is not called views but maybe templates or whatever it is, that you have to set this configuration item here, whoops this one and here, I will set it too even though it wouldn't be needed because views is the default. So now we're telling express that we want to compile dynamic templates with the pug engine and where to find these templates. The last step of course is that we add templates, so let's go to views and let's add a shop.pug file here. So we now have a templating file and now pug actually works different to normal html, so I can't just copy over this whole html code. The good thing is however in my IDE here if I type html in the shop.pug file and then I use this html5 template, I get a pug structure of this and here we already see the minimal html syntax. We get no normal html tags but keep in mind that the pug templating engine will compile our code to normal html in the end. So this here is basically the equivalent to this part here you could say, except for the stylesheet imports. If we wanted to add these, we can do that here too and thankfully, the IDE helps me here, if I type link and hit tab, I get a pug conform implementation of this. So here I now can also add my links, the paths of course to the css files are the same as before. So this is now how we installed pug and how we added it, now let's quickly finish this file to look like this file before we then actually have a look at how we can add dynamic content. So let's replicate this link here and import the product.css file and now for the body, here I got a header and I got this main area, so let's replicate this in pug quickly. For this in the body, indentation matters in pug that's important, you basically structure your nesting of html with indentation levels here, so if head is nested in html then it's indented and if meta is nested in head, it's also indented. If they're on the same level, they are siblings, so this is how that works. So if you want to add something into body and not next to it, we have to indent here and then I add my header which has this main header class, so if I type again another useful feature of the IDE, of visual studio code, if I type dot and then the name like this and I hit tab, then this looks like it didn't do anything but actually this is the representation of a div with this class, now I want a header with this class so I just write header. So this strange syntax will create a header element with this css class and this is something you have to get used to and where I strongly recommend diving into the official pug docs if you want to stick to that, here by the way, I'll replace the title with my shop. So let's not just add the header but also nav, unordered list and so on. For this in the header I want to nest something, so let's add an indentation level and then unordered list dot and the class and now the same, whoops that was nav not unordered list so that should be a nav item. Now for the unordered list, let's take that class, let's go in there and it should be nested in the nav. So let's indent, unordered list dot class name and now we get the list items, so here let's again pick the list item here or the class on it, indent because it's nested in the unordered list, list item dot and then the class. And now last but not least, I have the anchor tag which also has the active class but which then also should have some text content in it and then also have a link. So let's now go in there, nest this anchor tag and the class, if you hit tab, you automatically, this is the attribute notation, so in parentheses after the anchor tag or after any element but always without any whitespace in between, this allows you to add attributes onto the HTML element, so here this should go to just slash and now for the text content, you could add this next to it, so shop like this. This will basically place this text between the opening and closing anchor tag. Now this is the setup I want to have here, of course we have two list items, we also got add product which leads to this link, so I have to replicate this. So next to the first list item, I add another one, main header item and then I have my anchor tag, this one without the active class, just with the link and there, I'll say add product. With this if I run npm start, we wouldn't see anything and I am aware of the fact that I didn't add the main content but let's ignore that for now. With this however, we wouldn't render this template because we're not telling express to do so. We're telling express that it should use this templating engine whenever we try to render a template but we don't try to do that. So in shop.js where we do define what should be our response, we have to change the response because right now, we're sending the html file, now we want to do something else. We can use response and then there is a special render method, this is provided by expressjs and it will use the default templating engine which is why we had to define it here, it will use that default templating engine and then return that template. And now we defined that all the views are in the views folder, we also don't have to construct a path to that folder instead we can just say shop. We also don't need shop.pug because we defined pug as the default templating engine so it will look for .pug files. With this if we save and we reload the shop page, we see the header, we don't see the main content because we didn't add this but the rest is working just fine. And if you inspect this or view the page source, you'll see this is normal html code, so it's not our minimal version which the browser wouldn't be able to read anyways but it is the html code pug generated for us based on that minimal version. Now one thing we're not doing here is we're not outputting anything dynamic but since that is the reason why we added templating engines, let's do that too in the next lecture.

**Lecture 81**

**Outputting dynamic content**

We're rendering our shop template but we're not rendering any dynamic content, that however is the whole idea of this module. We get our admin data with the products, so let's actually take these products out of admin data products and now we want to pass that into our template, inject it into our template so that we can use it in this template file and somehow output it there. To do that, we can simply pass a second argument to the render method, the render method allows us to pass in data that should be added into our view. Here we can simply pass products, however not like this but as a javascript object where we map it to a key name which we then can use in the template to refer to the data we're passing in and we can pass in prods, simply to avoid naming confusion, you could use products but then I'll bind my products, so this constant to the prods key in this object. This is passed into this template and now in the template, we will just be able to access prods, this is now available, by the way we can also pass more than one field, we could pass in let's say a doc title which is shop and now we can use that too. And let's maybe start with the doc title so this additional field we're passing in this object. In shop.pug, let's say the title here should actually be that doc title. For this we can use the custom templating syntax pug gives us and if you just want to output some text, this is a hashtag followed by two curly braces and between these curly braces, you can put any value you are passing into your view, so any field you have in this object, like doc title, we can use that and simply refer to doc title here. If we now save that and we go back to our page, right now it's still in my shop right, if I now reload it's shop and it is shop because this is the title I'm storing in doc title here and doc title is what we're outputting here, so this flow is important to understand. Now of course we can also use that to output our products. For this next to the header, so on the same indentation level, this is important because it defines the nesting of the html, we can add the main element just as we have it here and now I don't want to have this h1 paragraph here instead now I want to output my product with this code. So I'll copy it over even though it is html code but we'll adjust this and transfer it to pug code. So the div here becomes just .grid, you don't need div because if you don't have anything, it's assumed to be a div, the article with this class becomes article.product item. However important, if you've got multiple classes, you need to merge them and simply concatenate them separated by dots and never forget the indentation, the article should be inside of this div with this class so let's indent it one level. Here we got an indented header because it's nested in the article and this header also has this class here, so let's add it like this and then in the header, we got a h1 tag so let's also indent this and here again this has a css class. Now it also has some text and you simply separate the text with a whitespace from the element and you never need closing tags here, that is handled for you by pug. So this is really a syntax that is very different to normal html and is really up to you if you like that that or not, I personally don't work too much with it but if you like it, this can of course allow you to write very lean html files. We don't need the closing header tag because again this is added automatically, so we can move on to the card image which should be next to the card header. So let's indent, remove that smaller than sign, add this class here without the class keyword and let's add the image inside of there. We get an image and again as before with the links here, if you have attributes, you wrap them in normal brackets, like this, so like the source and also the alt key. You can always by the way also use emett here, this is the plugin which helps you with auto-creating this, if you type the tag or the class name with a dot at the beginning and hit enter, it will auto-complete that for you and give you the respective pug representation and there we saw that we actually need a comma after source, right before alt. So now this will create an image nested in this card image, don't need the closing div tag. Now the card content is also a sibling to the card image, so let's indent this, remove that and add this dot here and let's do the same for the h2 tag, indent it, turn it into a pug conform setup here, we have some text next to it like this, for now that's static, later this will become dynamic and let's also do the same here for the paragraph, add the class here and have some static text and remove the closing tag. Now we're almost done, only a couple of elements to go. We get a div, now this should be also next to card content so let's indent it, a div as I mentioned doesn't have a tag you need to add, you can just have dot something, so dot and the class name. If you have no class, of course you would write that div otherwise there is nothing. The button is inside of card actions and this does have a class again and then it does have some text inside of it which we add next to the button element and all these closing tags can be removed. Now this outputs a grid of product cards however only with static content. Now to make this less static, we need to iterate through all the products and remember that we do pass the product into the view here on the prods key. So to iterate and repeat this article for all the products, we can simply add a special syntax provided by pug and you do create such a loop by adding each keyword then a value in which you want to store the value for the current iteration, so a single product in our case and then in and then the array through which you want to loop, so this would be prods in our case, again with prods referring to this key in this object we're passing to our view. So now we're looping through the prods, let's all indent this into the loop so that we repeat this entire block for each product in this prods array and now we can use the product variable which we're creating on the fly here to output the data, like the title. Here we could output hashtag and then again just product.title because remember, a single product we add to this array here is an object with a title key, we do create this here in admin.js, we push a new object with a title key, this is one single product as we'll have it when we loop through it, so this is what we add here or what we output here. With all these changes in place, if we now save this and we reload this page, we don't see anything because we have no products yet but if we add a first book here, we do actually see that here and we see that the image does not work anymore, so let me quickly add another free to use image, you can of course simply google for any image that you can use as a dummy image here for now. Once you got one, take that source whichever it is and simply replace it here, this is only for practice purposes here so now if we repeat this, this looks much better. By the way if you only change something in the template as I just did, you don't need to restart the server and node one will not do so because the templates are not part of your server side code, they are basically just templates which are picked up on the fly anyways so if you change them for the next request that's coming, they will automatically take that new version already. So with this, we get a basic introduction to pug, it's strange very minimal syntax and how you can output single values like some text or also loop through some items. Now of course we could have the case that we got no products at all, so one thing you also want to do is you want to ensure that you have a conditional check and render either content A or content B and we can do that too with pug. We can add an if statement and check if product.length, so if this array of products is greater than zero, if it is greater than zero then we'll output this, so let's indent it all into the if statement because then we do have some products to output otherwise I will go into the same level as grid here, as the if statement and add an else block because otherwise I will output h1 tag where I say no products. With this, now if I restart the server to clear my products array and I reload this page, I actually get an error because of course it's prods here, remember we're passing prods as a key not products, so prods is what I should check here. Now if I reload, I see no products but if I add one, first book like this, now we see that instead. So now we got the three most important parts, outputting simple text and so on, outputting a list and outputting conditional content. This is pug in a nutshell and this is how you use templating engines in expressjs in general. Now feel free as a practice to play around with that and also work on that add product html page and replace all these items or all the content here with pug templates. Keep the html code around, don't delete these files so that we can easily switch to other templating engines later but make sure you add pug templates for the other two pages and that you render them, also feel free to inject some dynamic code like the page title. In the next lecture, we'll do that together.

**Lecture 83**

**Converting HTML Files to Pug**

So let's convert all these html files to pug templates now and for this, I'll start with add product and create an add-product.pug file in the views folder. Now again I will create a html5 skeleton with the help of emmet here by typing html, selecting that html5 template here and hitting tab, so this already gives us a nice start. Now I'll set the title to add product and now I want to convert this form to pug and of course the header too. Now the header is something we already got in the shop so we can just copy that here, move it over to add product in the body like this, just make sure that you move the active class from the first anchor tag to the second one because now this is the active route and of course I also want to add my main section with the form now. I'll again reverse engineer it by copying the main html code, going in there and next to the header, I want to add it. Main is now just main, the form then is nested in that so let's indent it, the form has this class so let's chain this and the form also has attributes so let's add them here as a comma separated list in brackets. Now we can indent the div here because that is of course nested in the form, the div has this class and you learned that a div can also be omitted then so you can just have dot and the class to render a div. And then in there we have the label, the label here has an attribute, so let's add this and it has some text which is separated with a whitespace as you learned. Now we also have an input here, input of type text so let's add these brackets set here to render some attributes separated with commas and the id can actually be rendered separately with a hashtag in front of it, again using its css selector, something you already solve for the class, this also is the css selector for this class, pug uses these selectors, so now you have #title. Now let's remove the closing tag of the div and let's also add the button, it's indented because it's part of the form so let's make sure it's in the form, let's give that the button class don't forget this and let's add the attribute list to make this a submit button, like that and then a whitespace to separate the add product text. Let's remove the closing tags and that should be it. Now let's also render some dynamic content here, for example add product here, this title. I'll again output this dynamically, page title for example, now we have to make sure that we pass this key into this template and we do render this template in admin.js. So here where I do send this file, I don't send this file instead I now render, I render my admin file here, the admin excuse me not the admin, the add-product.pug file, this is automatically picked then and I pass in an object which holds the key value pairs I want to be able to access in the template and there, I got my page title key, so that key I'm trying to access here, I got that and I will assign a title of add product, just like that. With that if we reload, this is gone because the server restarted due to our server side changes. Now if I click add product, this is the html code by the looks of it, I forgot to add the imports, yeah I need to of course also import my stylesheets, so let's copy the links from shop.pug and add them here in the header. Let's also check the add product html file, I also need to import the forms.css file as you can see. So let's duplicate this here, this line and also add this import. Now no server restart is required because we just changed the template and therefore nodemon doesn't restart and now the reload, this looks much better and let's give this a try, it also still works. So that's it, now let's also work on the 404 page. For this I'll add a 404.pug file here and I will again create a normal html5 skeleton, I will say page not found here and I will already copy the, what do you need, the main css file, I will copy that import here from the add-product.pug file, this link, let's copy it over to the 404.pug file so that we got that. And then in the 404.html file, I get the header and I got Page Not Found, well we got the header already in add product so let's copy the header here, move that 404.pug into the body by indenting there and next to the header inside the body, I'll add a h1 tag where I say page not found. And with that we should have it, now let's just make sure that this gets rendered and for that, we have to go to the app.js file because that is where we have our catch all middleware and there instead of sending a file, I will now also render the 404 file and as before, it will automatically look in the views folder due to our setup and it will look for .pug files. So with that, if I now enter any random path which doesn't exist, we indeed get page not found. So this is now working and this is now using pug as a templating engine. Now there are a couple of other nice things that pug can do that I want to show you.

**Lecture 84**

**Adding a Layout**

You probably do notice that we do repeat a base structure in all our files here and of course it's a bit annoying that we manually have to do that, for example this base setup and also the import of main css as well as the header is part of every single page and therefore it's a bit cumbersome if we manually have to repeat this setup or this import all the time, so what can we do regarding that? We can create a so-called layout. For this I'll create a new subfolder in views which I'll name layouts in case we might have multiple ones and I'll add a main layout.pug file. Now I'll take my add or my shop.pug file or maybe 404 which is the most minimal one and copy that file content into main layout. Now of course we get some elements in here which are dynamic now, which depend on the page I want to use and the question is how can we reuse this skeleton anyways. Well we can extend this layout from inside our other pug views and we can actually define some placeholders, some hooks in this layout where other views can then enter their content. For example here in the links, we get a base layout which looks like this but in other views that should extend this layout, they might use this layout and add more links in this place. We can define such a hook by adding the block keyword which pug understands and then defining any name of our choice, styles for example. Now we will be able to add more styles from inside other files here and I will show you how to add this in a second. I also want to remove that active link here, by the way also in the 404.pug file because I have no assumption about what's active and I will show you how to dynamically set this in a second too. Now the content below the header also is dynamic and here I again will reserve a block, so a hook which I'll name content and the name again is up to you. So now we get this basic layout with two hooks we can dynamically enter content into from inside our other files. Now let's start with the 404.pug page here, there I will just copy that content that matters to me and remove the entire rest and we can now extend the layout by adding the extends keyword which pug understands and now we just need to point at the layout. For that we have the layouts folder and we want to use the main layout file in there, we have to add .pug here too. So now we're telling pug that we want to extend this and now we just have to tell it what to render in that content and maybe also in that styles hook we defined. For styles I don't need any special setting here but for the content block, I want to enter my own custom content and I do this by again typing block content but now since I extend a layout, this will not define a hook but actually allow me to add content in that layout and then indent it here, I define what should be injected into the content block in the layout, so in this place. With that if I save that and I enter some random path which doesn't exist, I still get page not found but now behind the scenes, this actually uses my layout and we can of course do the same on the other pages, like add product. There let's also for now keep the other content so that we can copy and paste content over into our new layout, so we extend main layout pointing at the layout file and here I will add something to the content but also into my styles block and we do that by simply typing block styles here and then indent it, we add what we want to add and these are these two lines because these are the two style imports which are not part of the default layout. So let's add them here in the styles block and then we have the other block, the content block, this one where I want to enter my own content. So for that, I will copy the entire part in the main section here or the entire main section actually, like this, cut it and move it over and replace this h1 tag here and make sure the indentation is correct and now we can remove the entire rest here and we should now have a file that does extend our layout and therefore still looks the way it look before but now it uses this way more concise approach which also ensures that if we ever change something to the layout, we don't have to change it in all the files but only in that main layout file and the other files since they extend it will automatically get it. So let's save this, let's go to add product and this looks good. Styling is good so all these styles are imported as you can see and it looks like before almost. We got this active class missing, we don't see that the add product link is the active one and that makes sense because in the main layout, we make no assumption what is active. So the question is, how can we dynamically set this?

**Lecture 85**

**Finishing the pug template**

Let's make sure we do set this active class on the right link depending on which page we are. Now for that, I will go to my routes and there since I'm working on the add product to the admin.js file where I do have that add product route, this one. Now I will return a new field here, the path field and I will set this to admin add product. Now you can set this to whatever you want, it doesn't have to match the path this route has, just a pattern I like. So now I pass this path into my view so that the view can find out what is the path for which this was loaded. Now in main layout, we will get this path and now I can do something interesting. In the main layout here, I know that I want to add the active class to this add product if the path and the path now simply is a variable i get passed into the page, if the path simply is this one and that is something we define and therefore you could pick any path you want. So now I will add a check here. Now a class can be added, a css class as an attribute here too, you don't have to use that dot notation, so class is equal to and now I'll have some javascript code which we can enclose with brackets here and now here, I simply check if path and path is simply just what I pass in here, so this key, if path is equal to this value and that is why I meant you can use whatever path you want, you are defining the condition here. So if path is equal to this, then I know that add product is the page I'm on because only for the add product route I do set this path. So then if that is the case, I want to render active as a class here otherwise an empty string, so otherwise I basically set no class. With this set up, now if I reload this page, you'll see add product is yellow because now it is marked as active, if I go to shop that's not the case but on shop, we also don't use the layout yet. So let's continue now. For the shop I also want to use the layout, so I will go to add-product.pug, copy my code here and move it up over to shop.pug, let's keep the original code for now, extend the main layout therefore. Now I do need the product.css file here but I don't need the forms so let's remove that from the styles block and for the content I will need main but we don't render a form here, instead we render this if loop, so let's or this if block, so let's copy that entire code and put it in here, make sure indentation is correct that you don't mess this up and now we can get rid of all the other code here. Now again for the styling of the header, we would have a problem if I now reload because it's not marked as active because I need to use that same approach I did for the add product route. I pass in an additional key path and there, I pass slash like this or /shop, again you don't have to use the real path that led to this route. I'll use slash though to keep this matched and now in main layout, I copy my code from down there where I dynamically add this class and add it here for the shop too but the part I'm checking for is just slash here and now I do add active here if the path variable which gets injected into the template matches slash which is only the case for this route. So now if I save this, I got an error here yeah this should not be an equal sign here, this should be a colon in shop.js . because we're creating a javascript object and now if I reload here, this is active, this is active and if we try adding a book, this also works. So this is now working. Now there is one thing I saw that I forgot and that is the title for the page, it's always page not found and that should of course not be the case. Now you could turn this title into a block which you then set from inside the template or you do add a dynamic output here with hashtag curly braces and give this a name, page title maybe and now you just need to make sure that every render function passes in a page title. So for the 404 page, for the add product page and the shop page, we have to pass in that page title key. So starting in admin.js, here we already got page title so we're fine, in shop.js here it's doc title so let's rename this to page title to be in line with the other file and with our layout and in the app.js file where we have our 404 route, I'm not passing anything so there let's also send some data into the view and let's also set page title here, page not found. If we now go to our page and reload it, we see shop here as a title, for add product we see add product and if we enter something random here, we see Page Not Found. So now this is working and now we're using pug. Now you can do way more with pug and you can combine all these features and do crazy stuff, I recommend diving into their official docs if you plan on using that, I instead want to show you the alternatives too now that we learned the basics of how templating engines work and how you can use them to output dynamic content in your views.

**Lecture 87**

**Working with Handlebars**

Now we had a very close look at pug as a templating engine but it's only one of three options. Now this was a quite extensive look because I also explained what templating engines are and what their core logic is, it'll be a bit faster for the two other languages therefore or the other two solutions so let's continue with handlebars which does not follow such a minimal html approach but uses normal html mixed with some templating logic. So let's use handlebars and for this, let's go to app.js . and in there, we now need to change our view engine. Now we did install express handlebars but this actually is a package that is not auto-installed by express, so instead we manually have to tell express that there is such an express handlebars engine available and for this, we first of all import it and to find out whether you do need to do this for your favorite engine or not, you'll simply check their docs because there, this is mentioned. So for express handlebars, we add express.hbs, the name is up to you by requiring express handlebars. So now this is imported and now we have to tell express that this exists, that this is an engine it can use. We do this by calling app and now there is an engine method and this registers a new templating engine in case we're using one which is not built-in, pug was built-in kind of, express handlebars is not. So to register this, we have to define a name for our engine and you can use any name you want though of course you should try to not clash with the built-in engines to which you also find a link at the end of this module by the way. Handlebars is a name we can use for example, so let's give this engine this name and now we just have to tell express this is the name, now what is the actual tool I should use? And that is express.hbs, so that object we just imported, that just turns out to be a function we can call and we have to call that basically initialises this engine you could say, so this function returns the initialised view engine which we can assign to engine here. So express handlebars gives us this engine and now we have to switch the view engine here to handlebars and obviously this name here has to match the name you set up here. With this you're good to go, now you're ready to use handlebars in your code. Now how do you use it? Now you do create new files for this and let's start with the 404.html page. I'll create a 404.handlebars page, now we have to name it handlebars here as this is the default by express handlebars and we defined this as an engine name here. You can also change the name for example to hbs, like this and now you're able to use .hbs as an extension, so this is how that works and how you register handlebars as the view engine. Now with that registered, let's use it and let's take the 404.html file and copy its content into the 404.hbs file because handlebars uses normal html with some custom syntax therefore there is no minimal html version as it was with pug. Now in this file let's remove that active class on that link in the navigation because that's still wrong, in that file I now want to change that here, I want to output that title dynamically so that you can see how that works with handlebars. Keep in mind that in app.js where we load that 404 page, we are passing in this data and the way you pass data into templates doesn't change with the engine, this is always the same type of flow. You pass in an object with key value pairs where the keys and therefore indirectly also the values are available in the template, just the way you use it in a template differs from engine to engine. And in handlebars, we output this value here by adding double curly braces, opening and closing and between these we add page title, so that key name for which we want to output the value. And with that if we save that and we go back and reload some page which does not exist, we should still get page not found, this time handled through handlebars and just as before with pug if we inspect the source, we of course see no handlebars code, the double curly braces was simply replaced with the content that should be rendered. So this is how we use handlebars and now if you're feeling confident with the help of the official docs where you learn how to use if statements and loops, feel free to go ahead and also replace the add product pages with the handlebar equivalents otherwise we'll of course do that together in the next lecture.

**Lecture 88**

**Converting our project to handlebars**

So let's work on handlebars now, for now without the layout. So let's go to the add-product.html page and copy that over into a add-product.hbs file, like this. Now this is now our add product page, this is all looking good, now let's output the page title dynamically and we're not using a layout for now, I will show you how this would work in a second too, so let's add the page title, active class should here be on add product, no need to assign this conditionally because we're in the add product.hbs file anyways here, this is not in the layout. And now let's also go down there, this should all of course still work, so this was pretty straightforward I guess. So if we now save this and click on add product, this should work and this is now rendered by handlebars because that is our view engine. Now let's go to the shop.html file and let's copy all that code here and add a shop.hbs file here, add that code here, the title becomes shop or better dynamically output page title. And now the interesting part of course is here in the main section, there we can comment in this code, so let's remove the comment tags here and now I want to loop through all the articles and then of course also change the image here, this is still the old not working image, let's take that new image link I added in the pug template, so let's move that into here too, like that. And with it added, let's now repeat this article for all the products we have and show no products found text if we got no products and for this we need to use a new handlebars index which you obviously don't know yet. Maybe you explored it in the official docs, otherwise let's add it together here. First of all there is an if helper, you still add your double curly braces here but then you add a hashtag for special block statements, block statements simply are statements which are not just outputting some text but which actually wraps some content that should be output conditionally or in a loop, here we can then add the if keyword and check if prods.length is greater than zero. So it's the same condition as in the pug template, just the syntax is a bit different. Now we want to output this article or actually not the article, the entire grid, so let's actually switch that, the entire grid here, let's indent this, indentation here doesn't matter but makes it easier to read. Here we want to wrap this and now we want to close that block at the end and we do that with a closing statement where we also have if here, by the way if here also has to sit next to the hashtag. So this opens an if block with the condition, then this content is only rendered if that condition is met and then we close it. Now of course we also want to have an else block so let's add this here too, hashtag else is then the key and there we simply put the content we want output if the condition is not met, no products found. With that let's save that and let's go back to the shop and we get an error. The problem we have here is that handlebars doesn't actually support statements like this, it just supports the output of keys that yield true or false. Now this means that we have to move that logic from the template into our node express code and pass the result of this check into the template. So we would go to shop.js and there, we actually add a new key value pair we pass into the template, has products maybe and this simply holds a value which is the result of our check here, products, so this products array here length greater 0. So now we just pass this key in there which is true or false, the result of this check and this is a core difference to pug already besides that html thing. Here in handlebars, we can't run any logic in our handlebars template, we just can output single property, single variables and their value and we can only use these in if blocks too. So here I can now check if has product is true because this is now the data I pass into this template and this actually also has a positive side, it might sound very complex but it forces us to put all our logic into the node express code where our logic typically should live and keep our templates lean because if you put too much logic in your templates, it can be hard to understand your code because you always have to check both, your express code and your templates otherwise you know the template is really just about displaying stuff, the logic always happens in node express. So these are two kinds of philosophies, it's up to you to choose your favorite but this is how express handlebars or how handlebars in general handles this. So now with this change, we can reload this page and now we just have a problem with else here but that is just that it's not #else, just else, just the else keyword my mistake, so now if I reload, we have no products found and if I do add my first book here, we do actually see that. However this of course is not the values we have in our products array, we have some hardcoded data here because I still have to add a block where we loop through all products. And this is done with an each block statement with #each and then you pass in an array, prods in our case. So now prods is passed into this loop and now in there, this element is repeated for each element and we also have to close this with /each after we're done repeating our code. So now this block of html code is repeated for every product. The question now just is how can we access this product and here handlebars also gives us only one way, it gives us the this keyword which refers to the element in the array for the current iteration, so to each product. So therefore we can output this title here for example because this will always refer to one product and one product still is a javascript object as stored in the array here in the admin.js file, an object with a title key. So now if we save this and we reload this page, now we see first book here because this is the title we assigned. So this is now how handlebars works and it's really important that you understand this different philosophy of having less logic in the template, more logic in the node express code, so you have to prepare everything there so that in the template, you don't have to write any javascript expressions. Therefore this is now the project converted to handlebars except for one thing and that is of course the layout.

**Lecture 89**

**Adding the layout to handlebars**

Handlebars does support layouts but it works a bit differently than it did with pug. First of all we have to configure something, in app.js where we register our handlebars engine, we have to pass in some options. Now my IDE helps me here, if I hit control space it shows me which options I can set and you can of course explore the official docs to learn all about them. Now what it allows me to do here is it allows you to set up where my layouts live, so which folder I can find my layouts. Let's set this here to a string which by default is views layout so you don't need to set this but you can set it kind of to reconfirm this but this is redundant, you only need to set it if you would store it somewhere else, like in lays or if you had a different default views folder. I'll set it to this even though that is the default so that you see that setting, now you can also define a default layout that should be used for all files and here I'll just name it main and that means you will have to have a main or main layout in my case, you will have to have a main-layout.hbs file in your layouts folder. So let's add a main-layout.hbs file in the layouts folder and let's set it up similarly to the pug layout and that was of course influenced by the 404 page. So let's copy over the 404 html code into the main-layout.hbs code and first of all page not found, of course we want output this dynamically here, we'll output the page title here with the handlebars handlers. Now regarding the styling, in pug we defined a block where we could dynamically add the content that should be rendered from inside the view that extended this layout. Unfortunately hbs doesn't have such a feature so we can't do that, we can't define a block here, instead the only thing we can do there is we can define a placeholder with three curly braces, opening and closing, so not normal double curly braces but three curly braces and then adding body here and you have to exactly use this placeholder. This is understood by handlebars and you will then be able to target this in your views that extend the layout automatically because you set it as the default layout. However if you have some part like this where you need to add some styling depending on the page you are on, you will have to solve this differently, in a kind of a similar approach as we solved the active class here in pug, you will have to add an if statement here in your main layout and this is of course a bit of a more cumbersome or complex way, though you can do way more with handlebars, you could define custom helpers that help you with that but this is beyond the scope of this course where I just want to give you a brief introduction, so let me show you the easiest way of solving this and this is that we add an if block here and we check if product.css and if this is the case, let's also close the if statement here, then I will add a link to my product.css file and I will repeat that same logic of course for forms and again as I said, this is the easy solution but it will do for now. So forms and here we have forms.css. So this is working now, now what regarding the active class, we also want to add that dynamically, right? Well again we can simply add the class here and I add my handlebars handlers here and I add an if statement and check if active shop is true and this is a variable we'll have to pass into this view, if that is true then I'll add the active class and of course also close may have statement so he can also write this inline, that is an important take away and I'll do the same for the add product route here where I check if active add-product is true. Of course we'll have to make sure that these variables are also passed into the view at least when they're needed, if you don't pass them, they're always treated as false, so you only need to pass them if you want to have them treated as true. With that, our layout is prepared, now let's go back to the shop.js file to start with that and as I just mentioned, we now have to pass in active shop, this is this variable or this property we're checking for here in the if statement and we should set this to true for this route to ensure that for this route, the active class is added. So let's set this to true and therefore in the main layout, this will be true and therefore active will be rendered, active add product would be false because we don't pass it at all and therefore this would not be rendered. So that is that, now of course we also need to make sure that the product.css gets rendered, so we need to pass in this too. So back in shop.js, let's also pass in product.css and set this to true, so you can already tell, you configure a lot from inside node express when using handlebars and that is its philosophy. So now with that set up, we can go back to the layout, have that finished and move over to shop.hbs. Now this will use the layout by default, you could disable this by going to the render function for a given page and setting a special layout key and setting it to false, this is a special key that is understood by handlebars and it would not use the default layout, otherwise it will. So since it will do that by default, in shop.hbs we can get rid of this entire code here, including the header and just have in this file what should be injected into our triple curly brace body tag in the main layout, so here, whatever should get entered in this place should be added to shop.hbs and nothing else. So with that if we save this and we go to the shop, you'll see we get an error and the error is a bit annoying to be honest with express handlebars, it's looking for the main layout.handlebars file. So it does not take .hbs here and for whatever reason, you explicitly have to tell handlebars to do that differently. You have to go to the options and set the extension name which only applies to the layout and not to all files, just as this applies to all files but the layout, you have to set the extname here to hbs. This is really strange but it is how express handlebars works, so now we're setting the extension to hbs for layouts too and now if we reload this page, we see our normal shop page. And with that we can of course also go to add product, here add-product.hbs, remove everything but the part which should be rendered in the place of our body, triple curly brace body tags. And if we reload, something's missing right, do you know what went wrong? Well we're missing some styles and the reason for that is that we need the forms and for this however, we need to ensure that forms css is set to true, we will also need product css by the way. So in admin.js, we should ensure that when we rendered this route, we set forms css to true and product.css, this one, product.css to true and that we also to get that active link in the navigation, that we also set active add product to true because active add product is the key we're looking for here. With that, if we save all files and we reload here, this is looking better and now it's all working again with handlebars. So as you can see, handlebars uses a different philosophy, definitely something you have to get used to, therefore your templates are leaner. And there is no better or worse way of doing this, it really is up to you which one you prefer. Let's now have a look at a third option, ejs which is my personal favorite.

**Lecture 90**

**Working with EJS**

So now we had a look at pug and at handlebars, two possible templating engines you can use and if you plan on using them, definitely dive deeper into their docs. Now I want to show you my favorite and the templating engine we'll use for the rest of this course and that is ejs. Now ejs is a templating engine which is just like pug, supported out of the box so we don't need to register the engine as we did it with handlebars, we can remove that, we can therefore also get rid of that handlebars import and here we just set the view engine to ejs now. Now ejs has a very nice syntax in my opinion and has a nice mixture of the extended functionalities of pug, so not regarding the html but regarding the javascript code you can use in your templates, that you can do things like that simple comparison here in shop.pug file, this comparison here with the if statement which we couldn't do in handlebars, that is again possible with ejs and this is the nice thing I like about it and still, it uses normal html just like handlebars which I personally also like but of course you can follow along in this course with any solution, I will stick to ejs though and all source code will therefore be provided with that templating engine. So let's use it then and let's start with the good old 404 page again. I'll add my 404.ejs file and now in there, let's start with the html code again and let's copy that into that file here, so all the 404 html code, let's move it in there, get rid of that active class which still incorrectly is there and now we got this set up. Now ejs does not support layouts but we will find a solution to at least have some kind of reusability of certain building blocks but for now let's just create the whole document in that file. So this is now our page and let's start with the title again and let's output that dynamically. Now in handlebars, we used double curly braces, in pug we used hashtag and then single curly braces, in ejs we use a smaller than sign and then a percentage sign and then if we just want to output a value in the place we're placing this placeholder, we add an equal sign right after the percentage sign and then we close this with just a percentage sign and a greater than sign. So this is the syntax and now here we can output the data which our template receives and just as before, the method of how we receive that hasn't changed, so we can still output page title here, just as before. So just to bring this back in memory, in app.js we're passing in that page title here and therefore we can reference page title in our template. So this will still work and with that we can already test this. If I now save this and I reload here, I get an error for add product because we've got no ejs template for that yet but if I enter anything else which is not found, we do indeed see our not found page rendered through ejs. So this is now working and with that let's move on to add product, let's add the add-product.ejs file here and in this file, again I'll just copy in my add product html code, here I also want to output the title dynamically though, so again our syntax here which we used just a second ago, let's output the page title here and then the rest can stay as it is for now. Now the more interesting part is the shop file because there, we also need an if statement and a loop. So let's add our shop.ejs file and in that shop.ejs file here, let's copy in the html code we got here for the shop and let's make sure we also take that new image here, so let's copy that from one of the other files then move over to shop.ejs and replace that here. And with that, now let's remove these comments again so that this is part of our page, let's remove that part up there and now in this file, we can of course again simply output our title equals page title as we did it before but the more interesting part are obviously the if statement and our loop, so we want to output this grid here only if we do have some products. Now an if statement is placed a bit differently than with handlebars, we still use our smaller than percentage sign syntax but not an equal sign because we don't directly output a value here in this place, instead we want to enclose, we want to wrap a certain block of code and we do this by adding our opening and closing ejs tags you could say like this and now in there and that's the cool thing, you can write vanilla Javascript code. Now we know that we will get in our shop route here, we will get our prods key which holds our products array and this will be an array and therefore here what we can do is we can write a normal javascript if statement, so normal javascript code and that is really important and simply say prods.length greater 0 and if that is the case, I want to render this, so thereafter after this grid, I also close this and now here important, you simply close the curly brace which you also have to open here at the beginning. So in the end what you do is you write a normal if statement as you would write it in a javascript file, just that the part inside of that statement is not javascript code but this html code and I find this to be very straightforward and easy to understand. So this is our if statement here with the opening curly brace and once we're done, we close that curly brace. Now we can of course then also combine this and add an else block here just as we would do it in normal javascript again and eventually once we're done, also close that and now here we can output our h1 tag where we say no products found. Now that is nice, let's see that in action by saving and by now going to the shop page and indeed, we get no products found. If I now do add a product on add-product here, first book, we do see it here so the if statement is working. Now again of course we want to loop through all the products and output our data and for that again, we use the same logic as with the if statement. We create simply the normal javascript code we would use for looping and there are multiple ways of doing that, we could use prods forEach to use this forEach function vanilla javascript supports or we use a normal for loop, we could say for let product of prods and then we simply open a curly brace, wrap all the content that should be repeated and once we're done, we add another ejs tag where we close that curly brace, again using normal javascript and then we have our product in there and now we can again output it as before with this ejs equal syntax as I like to call it here, basically this inline output syntax. And now with that if I reload this page here, we see object object, that is my mistake, obviously we want to access product title because product indeed is an object, so now we see the title. This is ejs and I prefer this syntax because I like having that flexibility of being able to write some javascript code in the template whilst having a clean syntax that nicely mixes with html because I personally like using html instead of the minimal version pug offers us. But as I mentioned, you can go with any approach you like.

**Lecture 91**

**Working on the layout with partials**

Now we added some ejs magic or some ejs templates here. One thing that is missing here is the layout functionality pug or handlebars gave us and indeed ejs doesn't have layouts but we can use so-called partials or includes, by the way a feature that pug and handlebars also know. The idea here is that you have some code blocks which you reuse in different parts of your templates and you can therefore just share them across your templates, so it's a bit like the opposite of a layout, instead of having one master layout where you put your individual view parts into, you have a couple of separated shared view parts which you can merge into the views you're creating and for that I'll create a new subfolder in the views folder which I call includes and that name is up to you. There I will create a couple of shared files or shared code blocks and which code do we share across all our views? Well we certainly share this part here where we have like the beginning of the html document including the title, including the main css even and then we share the body of each document so we also can outsource that even though it's not that much code right now but that might change if we have a common script that we import everywhere, we also share the navigation. So there are three include files I want to create here, head.ejs for the start of the document, end.ejs and of course you can name these files however you want and navigation.ejs. And now I'll just go to the 404.ejs file, grab the beginning of the page, everything that is shared, cut that out of here and move it into my head.ejs file. Now I just need to import this and this can be imported into the 404.ejs file by adding our ejs syntax, now with a minus and you use that if you want to output unescaped html code, that by the way means that by default if you have this syntax with the equal sign and you would render some variable that holds a string that holds html code, it would not render that html code but render it as text to avoid cross-site scripting attacks, with a minus you can avoid this and really render the html code. We can do this combined with a keyword offered by ejs, the include keyword which allows us to include a certain element into this page and then you close that with the normal ejs tag, without the minus. Now here in include, you simply add a string which holds the path to the file you want to include and you have to enter this path as it's seen from the file you're in, so the 404.ejs file is in the views folder so the file we want to include is in a subfolder. So we just have includes and then head.ejs and this will now take the html code in here and as I mentioned, then also render it here. Now let me save that and let's go to some route that doesn't exist and we still have a valid document here as we can prove if we open the page source, so this all works, now we're using an include. By the way if you use an equal sign here, you see now it gets rendered as text and that is what I meant, this is escaping the values so it's not rendering it, so if you had some script tag or anything fishy in there, it would not render it, it would just display it as text and therefore protect you but if you know what you're doing like we do here because we wrote the code we're including, we can and we should of course include it as html. So this is now using an include and we can now also use that for navigation, so let's take all that navigation code here, put it into the navigation.ejs file and in there, we'll also have to manage the active link again, I'll come back to that in a second, for now let's just go back to 404.ejs and here again, I will just include includes navigation.ejs and close that. And last but not least, the end of the document is also shared even though it's very short, let's move that closing body and html tag into end ejs and let's share this too, so here at the very end I will have include includes end.ejs, like this. Now with all that out of the way, I can still load this page just fine and it works. Now let's also implement this and this is of course a great practice for you too, so feel free to pause the video, let's also implement this for the other two templates we have. Were you successful? Let's try it for add-product.ejs, there we also want to take our common code here out, so the start of the document not the two links which are exclusive to this page here though, so which we really only need there but instead of the other part, we can now include our includes folder and there, head.ejs to bring back that start of the document here. In add-product, we obviously need to replace more like the header, so there we also should use include and then go to the includes folder, navigation, close that and close the tag and also the same for the end of the document since we did that before include, go to includes and then here includes and then there we have end.ejs. So now we get this in there too but of course one problem we face if we reload this page is that our active class is gone here, so we need to bring that back and mark this navigation item as active if we are on the add product page. Now for this, let's go back to the admin.js file real quick where we do render this route and there let's remember that we already added this path for pug and we can reuse that functionality. We can go to our navigation.ejs file and then add a class here and there simply output something rendered through ejs, we can do this inline with an equal sign so we don't need to enclose a block of html code and simply check if path is equal to and now the path here is /admin/addproduct, so basically the path which we define here and if this is equal, I'll use a ternary expression here which is an inline if statement in Javascript, so here we have the condition and if that is mapped, we now do whatever comes after the question mark. So then I want to render active here and since I'm inside the class text here, I will basically just add the active class, this is how you can read this, always keep in mind that this simply gets replaced with text when the template is rendered, so this will just become class equals active and then I add the else condition with a colon and else is just that I rendered nothing. And now I can copy that here, that class assignment and do the same for add product where this should belong, there where I did it originally, I should check for just slash because that is actually the path we set for the shop here in the shop.js, this path. So now I got this in place, I'm checking for the two different paths here and I render the active class based on that and therefore if we save all of that and we reload the page, we got add product added here. Now let's go to the shop.ejs file and there let's do the same. Keep the link which we need in that file, which is the product.css file and replace the other part with our general include, so in the includes folder, there we have the head.ejs file, replace the navigation as we did it before, so there we have include includes navigation.ejs and of course also replace the body of the document. So here we have include includes and ejs, like this. And now with that, if we save that too and we go to the shop page, it looks like it did before and we should still be able to use our application as we did before but now we're using includes which is kind of giving us the same experience as with layouts, we have to repeat all this include code all the time but that is okay and we still benefit from having shared code. So with that out of the way, we got ejs added and we'll continue to work with that throughout this course and I hope you got a solid understanding of why we use templating engines, which options you have and how they roughly differ and with that, you should of course be able to choose your favorite.