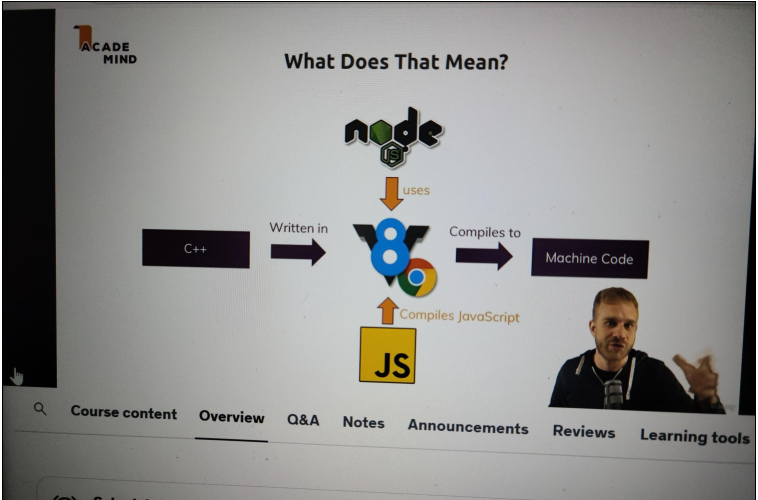
**What is Node js?**

****

So let's dive into the most important question, what is nodejs?

Nodejs is a javascript runtime

and now what does this mean?

You know javascript, it's a programming language you typically use in the browser to manipulate your dom,

to manipulate the page which was loaded in the browser,

for example to open a popup, a modal or add any kinds of effects because Javascript is a language that

runs in the browser that allows you to interact with the page after it was loaded and it therefore is a

crucial part when it comes to building interactive user interfaces in the browser,

so whatever your users see. However javascript is not limited to that.

Nodejs is a different version of javascript you could say, it is basically built on javascript, it adds

some features to it, is not capable of doing some other things you can do with javascript in the browser,

so it basically takes javascript and puts it into a different environment.

It allows you to run javascript code on the server

you could say, in theory not just on the server but on any machine though.

So it basically allows you to run javascript not just in the browser but anywhere else like a normal

programming language, like normal programs on your computer or some computer in the Internet effectively

making it a great choice for building web applications that run on servers which are just computers

running somewhere in the Internet.

So in detail, this means that we can use nodejs to run javascript outside of the browser,

that is the core takeaway,

now how does this work

technically? Well nodejs uses v8 and v8 simply is the name of the javascript engine built by Google

that runs javascript in the browser,

so back to the browser we now are. V8 is simply the name the creators gave their engine

and what does an engine mean?

Well it simply means that engine takes javascript code, the code running in your browser then or in node's

case which builds up on v8,

also the nodejs javascript code, it takes that javascript code and compiles it to machine code

and this is what your browser does too, what v8 does in your browser.

It does take your javascript code and compile it to machine code because that is the code that runs ultimately

on your computer

and that can be handled efficiently.

Now this is done by v8, v8 itself is written in C++ but that doesn't really matter too

much for you,

you don't need to write any C++ code to use javascript or nodejs

but nodejs

basically takes that v8 codebase which is written in C++ and adds certain features like for example

working with your local file system, opening files, reading files, deleting files,

these are all things which are not possible in the browser,

you can't access your local filesystem in the browser for security reasons, so this is not supported.

Nodejs adds these features to v8's engine you could say

so that you can suddenly do that. Now

nodejs does not run in the browser so these restrictions still apply,

there you use vanilla

v8, so v8 only without the nodejs extensions

but if you then install nodejs, you can use it to basically use that extended v8 version

to run javascript scripts on your computer which then suddenly can access these new features because they

don't run in the browser but are directly executed through that nodejs runtime you could say.

So this is how that works together and what nodejs does. It allows you to run javascript on your computer

and it adds useful functionalities to the javascript engine so that you can do more useful stuff there

than you can do with browser side javascript.

Now one important note maybe on this point

also is that of course some features are also taken away. In the browser you use javascript to interact

with the document object model,

so with the html elements on your page,

if you just execute a javascript file directly, you of course have no attached page and therefore these

features are missing. But this is a lot of theory,

why don't we just have a look at this and see how we can install and use nodejs and what it actually

does for us.

Let's do that in the next lecture.

**Installing Node.js and Creating our First App**

So we want to get started with Node.js and for that we first of all need to install it.

To do this, visit Node.js dot org and then there simply download this latest version.

In my case, that's 14.11.

It will change over time.

When you're watching this, you might see a higher version here.

It is important to note that these versions will change frequently, but they don't really bring many

new changes.

It's most of the time just behind the scenes optimizations and bug fixes.

I will keep this course updated though, as I have in the past, but with that, let's install this

latest version for this.

Simply download it onto your system and I'm doing this on a mac here on Windows.

It's exactly the same.

You can download the installer here and then you simply walk through that installer.

And again here that is the Mac installer on Windows.

You have a similar installer and you simply walk through all the steps you have there, leaving all

the default settings which are set up for you.

You don't need to tweak anything there.

Ultimately this will then install Node.js on your system.

This JavaScript runtime, this course is all about and we need Node.js to be installed in order to execute

JavaScript code with Node.js, which of course is our goal in this course.

So once this is installed, we can use it.

And the first and quickest way of getting started with it and of using it is that you open up your default

system, terminal or command prompt on windows.

So again, here I am on Mac OS.

So I'm using the built in terminal on windows.

You would simply open up the command prompt or PowerShell and you can check whether Node has been installed

successfully by running Node Dash V to check the version of Node JS that was installed.

And there you should see the version you just downloaded and installed.

And with that, if that succeeds, if you don't get an error, we know that it worked.

Now we can use Node and one way of using it is to enter an interactive mode.

Node JS offers you the so called rappel TA, which I will come back later.

You enter this interactive mode by simply executing Node as a command.

Now this enters a new mode in your terminal or command prompt, and here you can run certain node commands.

You can use it as a basic calculator to do basic math there, and you could do more than that.

You could run some JavaScript code in there.

Again, I will come back to this rappel later, but actually this interactive mode might be nice to

play around with Node.

You're not going to use it to write real Node programs for that.

Instead, you will use a code editor and store your code in files and then execute these files with

Node.

So let's do that.

Now for that, let's quit this interactive mode by pressing control C twice or control D or typing exit

as you see here.

And once you closed that mode, let's actually create a new project, a new folder in which we write

our first code.

And I did just that.

I created a new folder somewhere on my system and I opened it with my favorite code editor, Visual

Studio code, which you can find if you simply Google for VZ code or you visit code dot Visual Studio

dot com.

Visual Studio code is a free code editor, which is great for web development and it's available for

Mac OS, Windows and Linux so you can use it on any operating system.

Simply download it from code dot Visual Studio dot com and then walk through the installer that gives

you to install this editor on your system.

As a side note, you can use any IDE and code editor of your choice.

Of course, Visual Studio code is just what I will be using throughout this course and what I recommend

that you use.

If you don't have another favorite editor.

Of course, with it installed you can also open your project folder, which at the moment should be

empty with this tool simply by going to file and then open and then pick the folder in which you want

to write your code files.

Now, if you want to make sure that you got the same look and feel as I do, you can go to view appearance

and there you can control whether or not you see the sidebar, this activity bar on the left and so

on.

So that's how you can customize this.

And in addition, under preferences color theme, I am using the dark plus theme and you can switch

to this as well if you want to have the same colors.

Last but not least, under view extensions.

You can search for the material icon theme, which is 100% optional, but which will give you a specific

icon look, which you will see for all the discourse in my project.

So that's why I will enable this icon theme here.

For me, using it is optional though.

And with all of that we got the code editor set up, which we are going to use throughout this course

or which I am going to use.

And now we can write our first node code in a file for that.

I'll add a new file in there, which I'll name first dash app JS, dot JS because of course it will

hold some JavaScript code in there.

We can now write JavaScript code which can be executed by Node JS and a very simple first code snippet

we could write here is a simple console log statement which logs something to the console where we say

hello from Node JS for example.

This is code which would run in the browser.

It will also run if we execute it with Node.js.

Now to executed with Node.js, we need to execute this file here with Node.js.

And for that, the easiest way of doing that in Visual Studio code is that we go to terminal, new terminal.

This opens your default system, terminal or command prompt here, integrated in this IDE and already

navigate it into this project folder.

And then we can run this first app JS file by running node and then first the DASH app.

JS So simply add the file name after node and then you will not enter this interactive mode, but instead

execute this code file this JavaScript file with node JS and therefore you should see hello from Node

JS here.

Now that was a nice first example, but we can do more with Node and to right and at least a little

bit more realistic or more fancy application.

We can also try writing some output to a file instead of the console.

And for this we'll leverage one of the built in functionalities Node.js offers, and that would be the

file system functionality which enables us to work with the file system for this.

We first of all have to import it into that file to let Node know that we want to use this functionality.

And the syntax for that is that we call require a function made available by Node JS and we want to

require the FS module, which is one of node's core modules shipping together with Node JS.

And I will come back to this import syntax and to the node core modules in greater detail later.

For now, we just call this and we store the imported file system functionality in a simple constant

here.

And then we can use this file system to call right file sync, which is a method made available by this

file system object which we're importing.

And this method here will write a file to our harddrive.

And the argument it wants is the path to the file, including the file name.

And here we could name this hello dot text.

And then the second argument is the content of that file.

And here we could store hello from node js again.

So now this is our code here.

I am writing to a file by leveraging the file system module offered by Node.js.

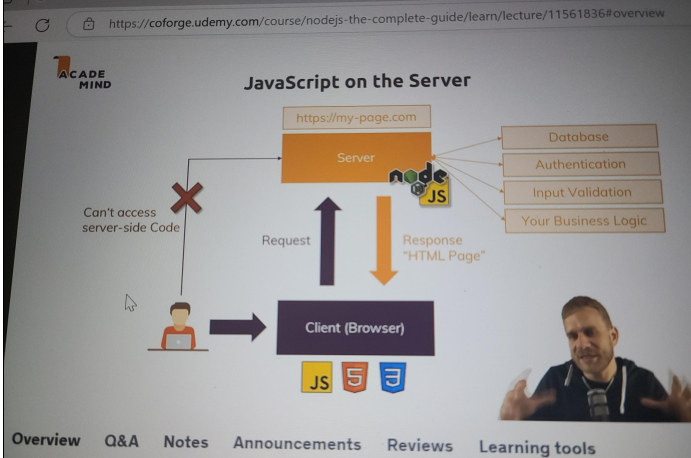
And if we now save this file and then run node first edges again, we should find a hello text file

next to our script file which contains this content, and that is how we can use Node.js.

Now, obviously, we're just scratching the surface at the moment.

We just learned about a brand new syntax, which you don't know from the browser, and therefore we

are going to dive way deeper into all of that over the next lectures and throughout this entire course.





**How The Web Works**

So let me quickly refresh our knowledge on how the web works.

If you are totally aware of all of this, you can of course skip this lecture.

Now the web works like this,

we get got a user, a client,

maybe you sitting in front of your browser, visiting a webpage or already being on a webpage and submitting

a form,

so you're interacting with webpages.

Let's say you are visiting it,

so you're entering some url into your browser

and what happens behind the scenes is actually that the browser reaches out to some domain name servers

to look that domain up because this domain is not really the address of your server,

it's basically an encoded human readable version of that address you could say, your server itself has just an

IP address but this is just some technical thing behind the scenes,

in the end you enter this url and it will lead to some server. You therefore or the browser therefore

sends a request to that server

with that given IP address I mentioned, so the IP address belonging to that domain.

Now thus far that's all interesting but now we reach the part where nodejs comes into play, where your

nodejs code matters.

You write the code that runs on that computer in the Internet which has that IP address,

you write the code that spins up that server which is able to handle the incoming request and do something

with it.

Now you don't need to use nodejs

for this, you could use PHP, asp.net, Ruby on Rails,

anything like that

but in this course, we'll obviously use nodejs because well it's a nodejs course.

Now in this code, you can do all kinds of things and I already mentioned this in the first course module,

user input validation,

communicating with the database maybe which runs on a separate database server but which you typically

reach out to from your backend, so your server side code

and once you're done with that, you do one important thing, you send back a response to the client.

This response can be some html text, some html code which is then handled by the client but it could

also be some other kind of data like a file, some json or xml

data.

The response is more than just the content by the way, a response and also a request also has headers,

this is some meta information which is attached to request and response describing what's inside it

for example

but we'll see this too.

So this is how the web generally works and nodejs is the part we will focus on, it is the code that makes

up that server in the end.

Now that request and response transmission is done through some protocol,

so basically a standardized way of communicating you could say because obviously, to correctly handle a

request and send back a response the browser can work with,

we have to follow some rules and these rules are defined by the protocol we use, http or https.

Http stands for hypertext transfer protocol

and there we simply define or it is defined how a valid request looks like and how the data should

be transferred from browser to server

and the other way around and https simply is the same with SSL encryption turned on where all the

data that is transmitted is actually encrypted so that if anyone is spoofing your connection, they can't

read your data.

Now towards the end of the course, I will show you how to enable https, for the majority we'll just

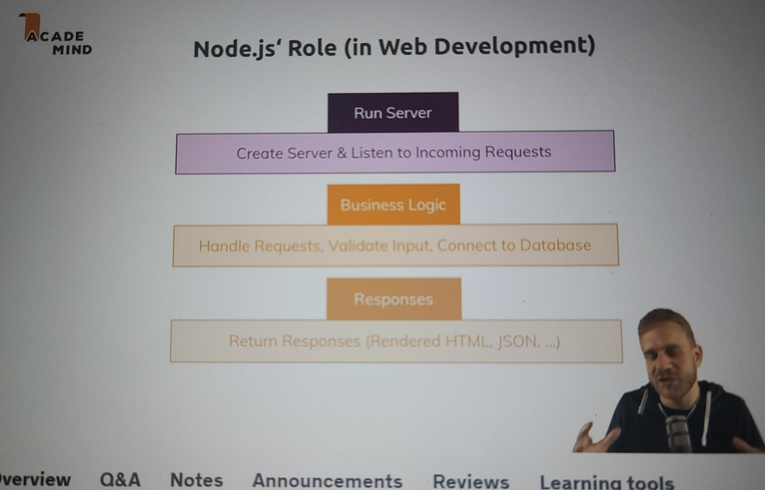
use http since we'll only be developing that code, we'll only work on it locally

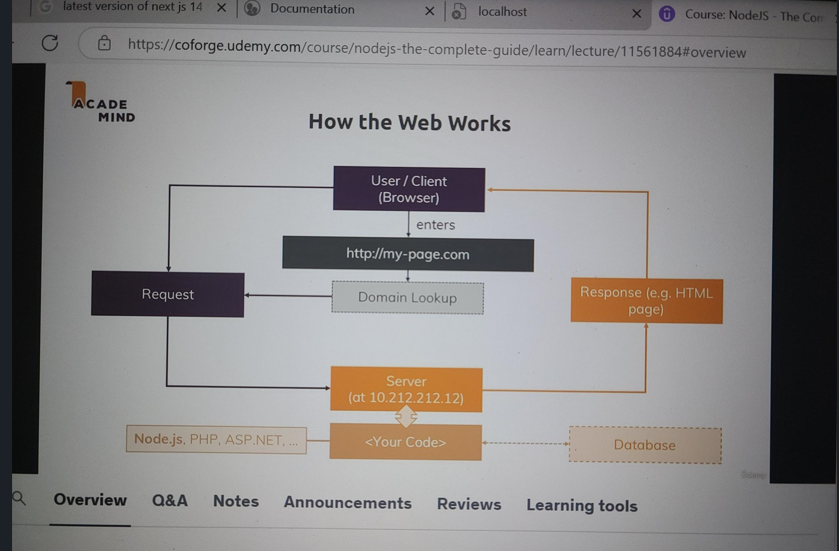
but once we put it into production, I will also show you how to turn on that SSL encryption.

This is how the web works

in a nutshell and how nodejs is related to it and this is exactly where we will now continue working

with nodejs and where we will now finally create a server with nodejs.





**Creating a Node Server**

I'm back in Visual Studio Code, the IDE I'll use throughout this course and this is again a totally empty folder,

I only got my gitignore file in there because I will use version management here,

git

you don't need to use that at all, if it doesn't tell you anything you can just ignore that,

you don't need that gitignore file.

So I have an empty folder

and again I will now create a new file with command and or by pressing this icon here and this file

can have any name you want

but often you name it server.js or app.js

because it is the root file that makes up your nodejs application,

so the nodejs code you will execute on a computer in the cloud on a server in the end, so I'll name

it app.js here.

Now in this file, I want to create a server through nodejs and thus far in the first module, we only saw how

we can work with the file system,

so how can we now spin up such a server?

We again need to import some functionality because the way javascript works both for the browser and

nodejs,

there is a handful of functions and objects we can use globally without importing anything into the

file

but generally, most functionalities aren't available by default, to not pollute our global namespace

with all these reserved keywords and names basically

and also to make it very obvious in each file on which functionalities this file depends

and thus far, this file does not depend on anything.

Now there are a couple of core modules nodejs ships with and as you will learn throughout the course, you

can also install third party modules which do not ship with node but let's stick to the core modules

for now.

Here are a handful of them,

now not all core modules, just some

and as you can see there is fs which we already used thus far,

there also is path which helps us with constructing paths,

so paths to files on a file system that work on any operating system because Windows and Mac and Linux

use different path formats.

There is the OS package which helps us with operating system, relevant information and so on

and there are the two topmost packages, http and https

and as you might be able to guess, these two sound very helpful when it comes to creating a server and

working with http requests and http responses. And indeed, http helps us with launching a server or also

with other tasks like sending requests because a node app could also send a request to another server,

you can have multiple servers communicate with each other.

For example you could send a request to the Google Maps API to send some coordinates and get back

an address but that's just an example, let's keep it simple here and let's focus on the launch a server

aspect. Https would be helpful when we want to launch an ssl encoded server, so where all that data which

is transferred is encrypted

and as I mentioned earlier, this is something I'll come back to towards the end of the course.

Now with that. let's use that http module and to use it, we need to import it.

So we basically need to make sure that we can use features from that http module which nodejs ships

with but which still is not available globally by default,

we need to make sure that we can use these features in this file

and for this, we import that functionality.

We do this by creating a new constant

and you could create a var or use let too but since we'll have some value here which we will never

change, we can also just use a const to make this really clear that we will never touch this again,

we'll use it but we'll not overwrite it

and you can give this any name you want but typically, you keep the name of the module you're importing.

So I'll name this http

but again you could rename this to whatever you want.

Then you have an equal sign and now we need to assign a value

and now there's a special keyword, a special function

nodejs does expose globally,

so you can use it by default in any file you run via nodejs and that is the require keyword.

Now this is simply the way you import files in nodejs, require either takes a path to another file,

you can also import your own javascript files but we'll not do this for now,

we'll do this heavily throughout the course though

or if you don't have a path to one of your files, you can also import a core module, like http.

By the way, a path to one of your files always has to start with ./ or slash if it's an absolute

path, ./ would be a relative path,

so this would lead to the same folder and would now look for an http file.

By the way it automatically adds .js at the end,

you don't need to add that on your own but you can.

But this would now look for a local file named http,

if you omit ./ or slash at the beginning, it will not look for a local file,

so even if you had a file named http.js, it would not import this file,

let's get rid of it but instead it will always look for a global module named http and indeed, such a module

exists because nodejs ships with it.

So now we got this imported and now we can start using functionalities from that global module and

you can see that if you type http., this is how you access functions or so-called methods and properties

on objects in Javascript and as you can see, this http object which we import from the http module

has a bunch of fields and methods we can execute.

Most importantly, it has the create server method.

Now as the name suggests,

this is a crucial method when it comes to, well creating a server. And create server,

actually if we hover over it

we can see it, actually takes a so-called request listener as an argument. A request listener simply is

a function that will execute for every incoming request

so let's define such a function. For this I'll create a new function with the function keyword,

we can name it however you want,

rqListener or whatever you want, the name is totally up to you and this function has to receive two

arguments,

you can see that here if you hover over that. The request listener receives a request which is of type

incoming message and a response object,

so in short nodejs automatically gives us some object that represents the incoming request and allows

us to read data from that request and it gives us an object response which we can use to return a response

to whoever sent that request.

So now we have to accept these two arguments here and you can again name the arguments however you want,

you just have to keep in mind that the first one will contain data about the request and the second

one will help you send a response, so I'll name it req and res

and these are typical shortcuts you often see.

Now this is a function,

rqListener

and now we can pass this function reference so we don't execute it,

don't set these curly braces,

just pass the name to that function because this will simply tell create server

hey please look for this function with this name and execute it for every incoming request,

so this is now what we'll set up. This function will now run for every request that reaches our server

which will be started by calling create server or almost, one piece is missing,

I'll come back to that.

Now this is one way of doing it.

Now you don't have to explicitly create such a function though,

you can also use a so-called anonymous function.

So here, you can also type function req res, like this,

this is now a function without a name,

that is why it's called anonymous and it still achieves the same.

We pass that function to create server and therefore, node will execute this function whenever a request

reaches

our server. This is an event driven architecture

nodejs uses heavily.

You work a lot with such setups or such code snippets where you tell node if X happens, do Y,

so in this case if a request comes, please execute this function.

Now you can also use next-gen javascript syntax and use an arrow function where you omit the function

keyword and just have the two arguments followed by an equal sign and a greater sign

hence an arrow and then the function body. This is basically the equivalent to the function keyword approach.

Ok, so this is our create server callback function as it's called, it's called by nodejs whenever a request

reaches our server,

for now let's simply console log the request object to see what's inside.

Now if we execute this file, we can do this in the built-in terminal which is already navigated into

this project folder,

make sure you go into that project folder

if you are using the terminal outside of that IDE.

So once you are in a terminal, navigate it in that folder, you can run node and then app.js,

this will execute the app.js file and let's see what it does.

Hmm, nothing right?

We don't see console log and that makes sense because we didn't send a request to the server but we don't

even know where the server is,

how do we reach that server, which address does it have?

Well one important thing is missing,

this create server method actually returns a server.

So we have to store that in a new variable or constant and I'll use a constant because I'll never overwrite

it, I only create a server once.

So now the created server is stored here and now we can use that server and do something with it.

As you can see we get a bunch of methods we can call and one method is listen. Listen now actually starts

a process where nodejs will not immediately exit our script but where nodejs will instead keep this

running to listen,

that's why the method is named like this for incoming requests.

Now listen as you can see takes a couple of arguments, optional arguments,

the first one is the port on which you want to listen. Now in production you typically would not fill this

out and it would take the default of port 80

but here on local development, we want to use a different port and you can also define a hostname. Now by

default,

this will be the name of the machine this is running on,

so for our local machine, this is localhost by default.

So let's just pass a port, 3000 is a port you often use but you're relatively free to use any

port you want, the thousands port are typically pretty safe.

And now with that, if we re-execute this, you'll see one important thing. The cursor here in the terminal

doesn't go back in a new line because this process here is now still running,

it didn't finish, this file execution didn't finish because we now get an ongoing looping process where

this will keep on listening for requests

and this is obviously what you want, right?

You want to have a web server that keeps on listening for requests.

Now we can see that in action by opening a new browser window where we simply enter localhost 3000

and once you did that,

nothing happens because we haven't configured it to return any html page

but if you go back to your terminal, you'll see a lot of output there and that is this line, this is your

request being logged to the console.

Now let's analyze what happened here in detail and what's inside this request

over the next lectures but these few lines here already give you a fully functional or almost fully functional

web server

and this is how you create servers in nodejs and I know that this can be hard to wrap your head around

because it was for me when I started out with nodejs years ago, it was difficult to understand that

coming from a PHP background

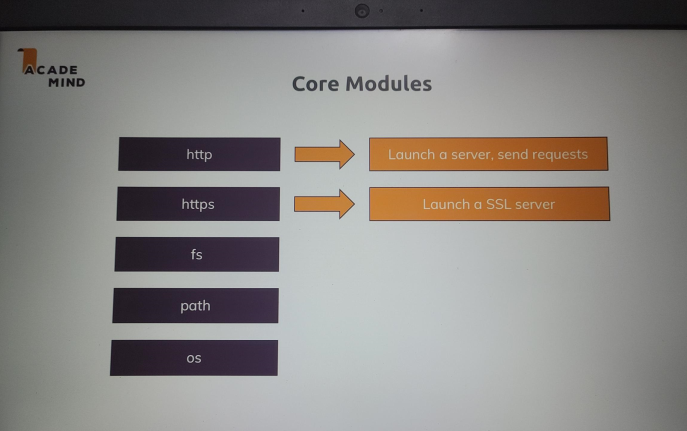
you suddenly write your own server, that sounded like something super complex.

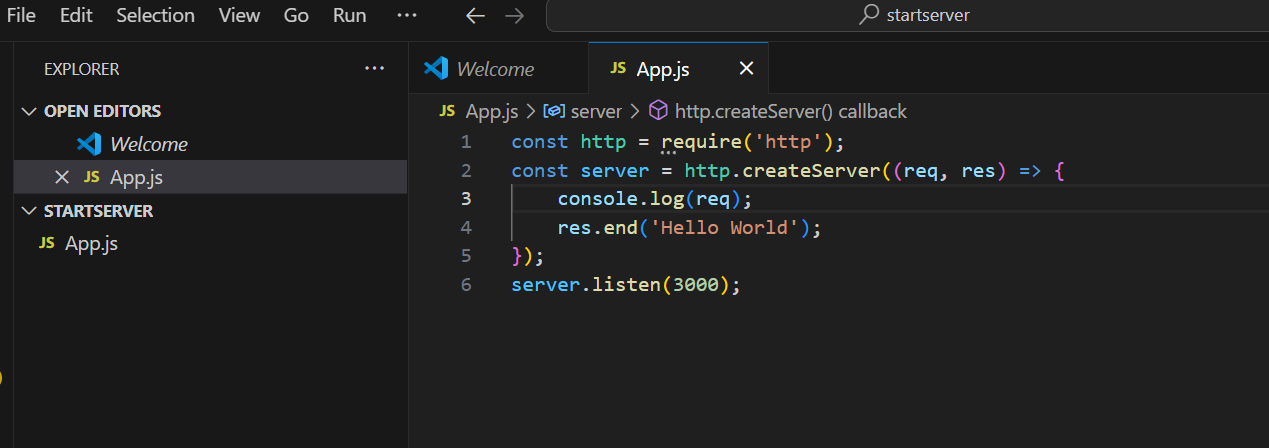
Well actually it's just these few lines

and from now on we'll just have to focus on doing something meaningful with the request and important, sending

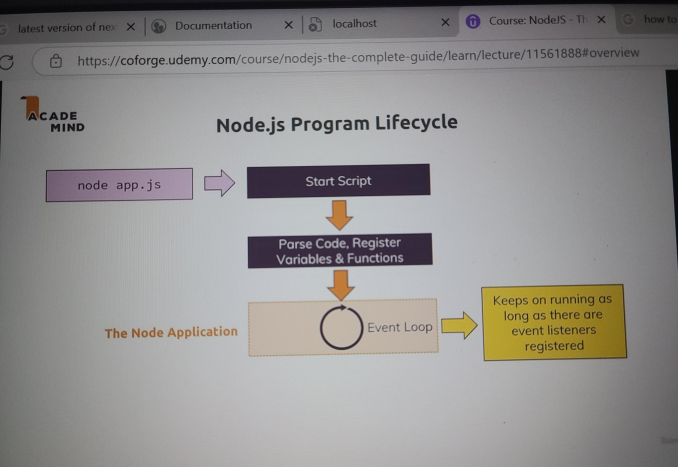
back a response.

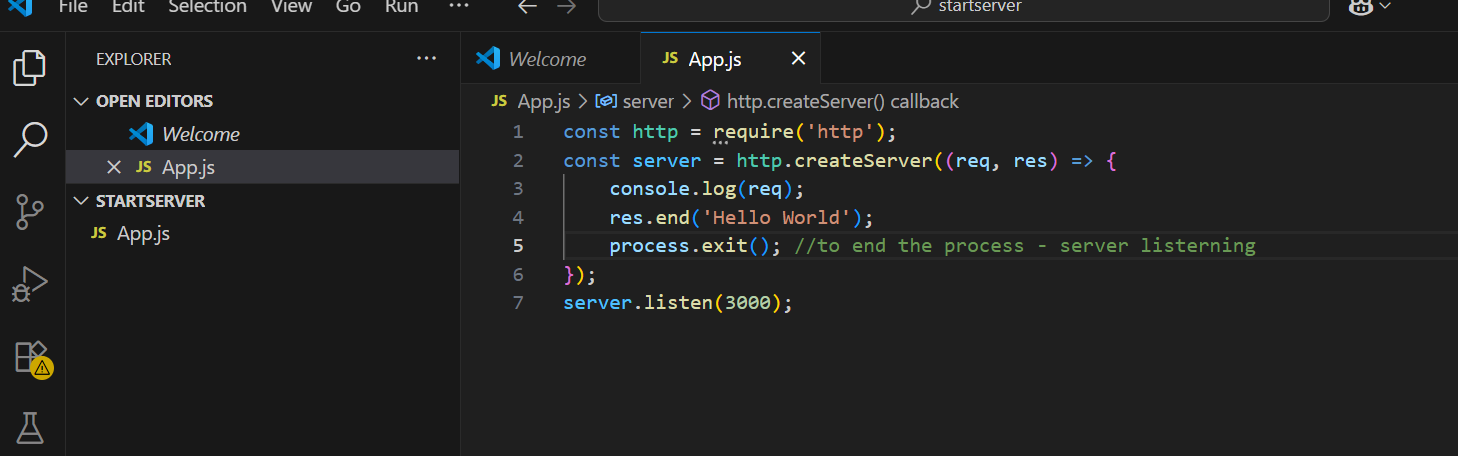
So time for detailed analysis in the next lectures.





**The Node Lifecycle & Event Loop**

****

****

In the last lecture we wrote our first little nodejs server.

Now this server is not doing all it should do,

it's not returning a response for example but it's generally showing us how we do create server side

code with nodejs and how we do create our own servers.

Now let's understand what happened there,

we executed that file with node app.js

because our file was named app.js and this essentially started the script where nodejs went through

the entire file, parsed the code, registered the variables and functions and so on,

so it basically read our entire code and started executing it.

But then something important happened,

we never left that program, right.

The reason for this is an important concept in nodejs called the event loop, this is basically

a loop process which is managed by nodejs which keeps on running as long as there is work to do

you could say, it keeps on running as long as there are event listeners registered and one event listener

we did register and we never unregistered is that incoming request listener we passed or we set up with

the help of create server.

We passed a function to create server

and that is basically an ongoing event

listener, one we didn't unregister from and we shouldn't because our server should of course stay up

and running.

So our core node application basically is managed by this event loop, all our code is managed by this

and as I mentioned, nodejs uses such an event driven approach for all kinds of stuff, not just for managing

that server but that is of course a crucial part

but you will see it a lot throughout this course,

for example when we later access a database, you will see that there we also basically send that please

insert some data request and then we register some function that should be executed

once it's done. And nodejs uses this pattern because it actually executes single threaded javascript. So the

entire node process basically uses one thread on our computer it's running on.

Now as you might guess if we create a server with nodejs, it should of course be able to handle multiple,

thousands, tens of thousands or hundreds of thousands of incoming requests

and if it would always pause and then do something with that request, this would not be that great

hence it uses this event loop concept where in the end it always keeps on running and just executes

code when a certain event occurs

so that in general it's always available. And whilst this might still sound like ok but if I got two incoming

requests, it needs to handle two events,

well it is super fast in handling these requests and actually behind the scenes, it does some multi-threading

by leveraging the operating system. But this event loop is a core thing you have to keep in mind that

nodejs basically has an ongoing loop as long as there are listeners and create server creates a listener

which never stops

but if you eventually were to unregister and you can do this with process.exit, it would end and

we can see that too.

If we go back to our code and after this line, we type process.exit and execute this function and

we clear our console and we now repeat node

app.js, you see

it's now still running because this function never executed,

we had no incoming request yet. But if we now reload this page, localhost 3000, you see we still log

that request but then we're back in a new line in the terminal because now it did quit that process,

it didn't before but now it did.

It did quit that process with the help of process.exit.

Now typically you don't call that in your code because you don't want to quit your server, if it quits

people will not be able to reach your webpage anymore but this is important for understanding how this

works.

Process.exit basically hard exited our event loop and therefore the program shuts down because there

was no more work to do,

nodejs saw that there is no more work to do

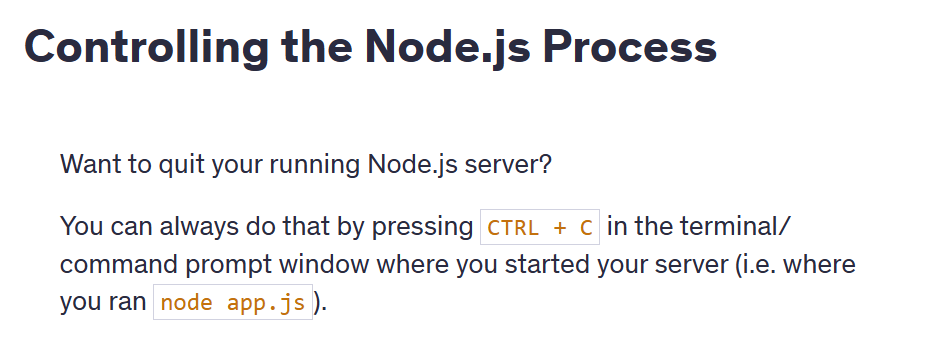
and it basically closed the program and gave control back to our terminal here.

So this is how we execute such a nodejs file and how we generally create such a server and what nodejs

does behind the scenes,

let's see how we can now also work with the request and with the response object we have in that anonymous

function here.



**Understanding Requests**

So let's go back to the request object we logged here.

Now important just to keep that in mind,

this request object is the object nodejs generated for us with all the data of the incoming request

when we visited localhost 3000 which we in turn can do because we listen to requests on that

port.

So this is the request object, if we have a look at it,

we see it's a very complex object.

There's lots of data in it,

it's not just data,

these are also partly functions we can call and so on,

so this is quite a complex object but we also see that for example we have some headers here. Headers

as I mentioned earlier are metadata, meta information added to a request and also to responses by the way

and there we see for example the host, this was sent too, the request was sent too.

We saw some headers attached by the browser

like for example how the response data should be cached and stuff like that, which browser we used

for that request, which kind of response we would accept, that we accept some html, xml and so on,

that we would accept encoded responses, so where the response is actually minified to well save

bandwidth and so on.

There also was a cookie attached

even, we haven't learned about cookies yet, we'll do so later

but this was attached at some Google Analytics cookie. We saw which http version was used and so on,

so there's a lot we can gain from that request but a bit too much.

Now there are only a few important fields you typically need.

The first important or interesting field is the url, now

let's output that and let's also output, you can output more than one value by separating them with

commas, request method and also request headers here.

Let's output these

three values and restart the server with node app.js,

so now it's again listening and let's reload that page on localhost 3000.

If we do so, now we see the output has changed,

we still have all the header stuff because we're outputting request headers but prior to that, we output

the method which you see here, it's get and you see the url and the url is just the slash here because

the url

is basically everything after our host and we just have localhost,

well nothing

and that basically translates to localhost slash.

If I had /test, now we see another output and there we see /test being logged here and then

also get for the method and our headers.

So this is basically how we can access some information about our request.

Now one crucial thing that is missing here is the response,

so let's have a look at sending responses in the next lecture.

**Sending Response**

In the last lecture you saw how to handle requests and how to read some data from the requests like

for example the url and which http method we used.

Now we'll learn about different http methods throughout the course,

by the way get is default method used if you just enter a url into your browser.

Let's now shrink this again,

let's quit the server,

you always need to quit and restart

if you edit it because otherwise your changes will not be reflected because the old process will still

be running

and let's now also use that response object.

Now we could log that with the console but actually this does not hold any interesting data,

instead we can use it to fill it with data

we want to send back.

We do this by calling res and now what?

There are a couple of methods we can use, for example set header,

this allows us to set a new header.

For example content-type and that is a default header which the browser knows and understands

and accepts and then as a second value here, as a second argument,

in set header, we set a value for this header key,

and we can send this to or set this to text.html

.

Now what this will do is it will attach a header to our response where we basically pass some meta information

saying that the type of the content which will also be part of the response is html.

Now there is of course only a certain set of supported headers

the browser understands and after this lecture,

you'll find another lecture with some link where you can learn which headers you can set.

Now you don't need to set that many,

let me say that

and later we will even learn about a package that does this for us so that we don't have to set the

content type on our own.

But here I will set it, now

one important thing is missing of course and that is the html code, right.

Thus far I'm saying we have html code but I'm not sending it.

Now we do this by setting response and now we can set write here,

write allows us to write some data to the response and this basically works in chunks you could say

or in multiple lines,

this would be a good picture to look at this,

you write multiple lines of response.

For example if we write html code like this, res write

and if this looks super strange now, it is, we'll learn about a way easier way of sending html later,

no worries.

So here I'm just sending html and you can also put longer html in there,

of course you could now have your head where you set the title, my first page like this and you could put

even more into that line but to make it easier to read

I'll write

a new line where I now have my body. So I'm basically writing a whole html document here in a very complex

way

and there I'll just add a h1 tag saying hello from my nodejs server.

Ok

so this is now some html code and it will be written to the response line by line.

We now also need to tell node once we're done with creating that response and we do this by calling

end, so after we set all the headers and wrote all the data to the response body, we call end and now

is the point where we must not write anymore.

We can still call write but this will result in an error because we must not change the response after

we ended it because this is basically the part where we will send it back to the client,

nodejs will send it back to the client.

So here it should send back a response with some html code inside of it where we also tell the browser

that it's html code,

the browser wouldn't know otherwise.

And with that if we save that file, make sure you never forget to save your changes and we re-execute

it,

we again have that running process

and now if I reload my localhost 3000 page here, we see hello from my nodejs server.

And if I open the chrome developer tools here which you can also do from the menu, I use the shortcut,

you can also use view developer, developer tools or that shortcut you see here.

Now if you do that, let we reload, in the network tab here you will see this request, this first request

and there you see headers, like for example in response headers,

there you see my content type which I set right, this is the header we set here

and if we click on response itself to see the response body, we see that html document code we

wrote with the head section, with my first page and so on.

So this is now how we can send a response,

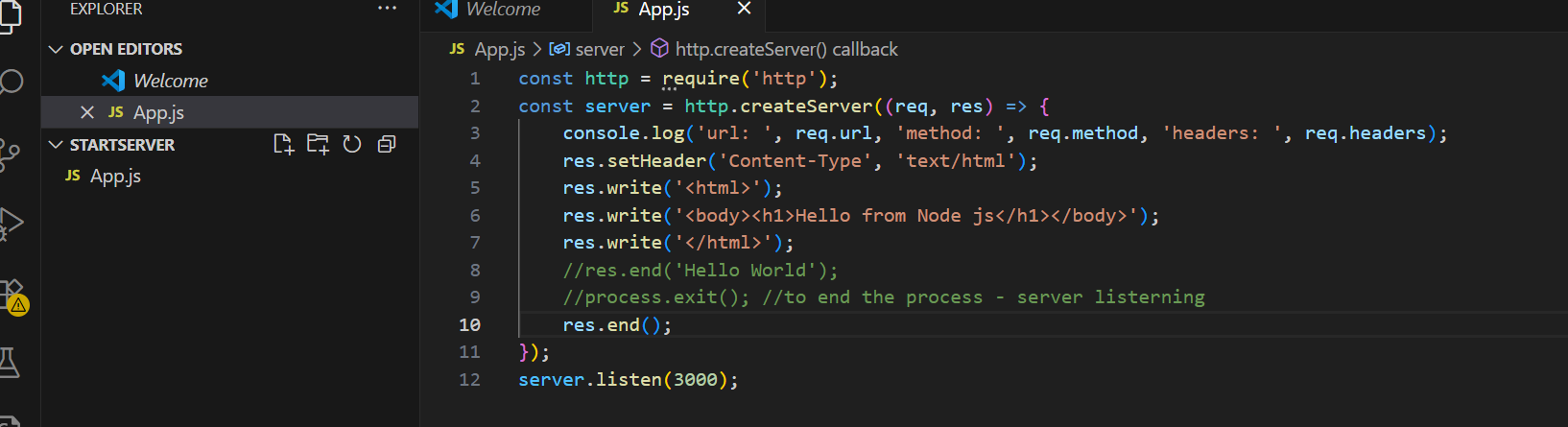
we'll later also learn about a way simpler way of doing that by using the expressjs framework

but it's super important that you understand all the nitty gritty details that go on behind the scenes

and in this case, we simply understand it by writing all the nitty gritty code on our own business. This is how

we can work with requests and send responses,

now let's connect both the request data we can get and the response data we can send.



**Request & Response Headers**

On both requests and responses, Http headers are added to transport metadata from A to B.

The following article provides a great overview of available headers and their role: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

Whilst this article is a great resource, especially to dive deeper, please **don't learn this list by heart** though! You'll encounter many of these headers throughout the course and I'll explain them when we need them.

**Routing Requests**

In the last lectures we learned how to spin up that server and that we get a request object with information

about the incoming requests and the response object that we can use to send back a response,

let's now connect both, requests and response.

Instead of printing some request data to the console which of course doesn't do much for us,

let's instead start writing a very simple web server that does different things depending on which route

we enter,

so depending on which slash whatever part we enter here.

So let's say for slash nothing,

we want to load a page where the user can enter some data which we then store in a file on the server

once it is sent. We can do this by first of all parsing the url. I'm storing it in a new constant and I

do this by accessing request

url, remember that was something like slash, /test, whatever we entered.

I will then add an if statement and check if url is equal to just slash and only this will match,

by the way the triple equal sign means that this will only be true if url is both a string and has

that value.

So now here if that is the case, I want to return a response which holds some html that gives the user an

input form and a button that will send a new request in return and that will not be a get request

by the way.

So let's do this step by step, let's copy this code here and put it into this if statement

and here I will write a html document with a head, maybe a different title, enter message

and the body will now not hold a h1 tag but instead a form,

this is a default html element of course with an input of type text

let's say and a button and this will be super ugly because we have no styling but it's about the

functionality for now,

the button tag must be closed by the way. The button will be of type submit so that it submits the form

and that will be some default html behavior we're using here

where a button with type submit in a form element will send a new request and we'll configure that request

in a second.

Let's first give the button a caption, send

and now on that form element here, we add an action which is basically the url this request which will

be generated automatically should be sent to and I will use /message here and this will automatically

target the host

it's running on, so localhost in our case here, localhost 3000

to be precise

and then we define the method,

the http method that should be used and there we previously saw if we expand this, that we get a get request

which is the default if we enter a url,

well here we are not entering a url instead we want to send a so-called post request. There is a limited

set of http words you can use, get and post are the two most important ones. A get request is automatically

sent

when you click a link or enter a url, a post request has to be set up by you by creating such a form,

there also are some other ways by using javascript but we'll ignore them for now.

So in html we create such a form and we defined that the method should be post and this will send a

post request to /message and the cool thing about form is it will not just send such a request,

it will also look into the form, detect any inputs or related elements like selects

we might have

and if we give that input a name which we should, message, it will also automatically put that message

into the request it sends to our server.

So now here when we visit just localhost 3000 slash nothing, we will return a response where

we render this html code.

Now let's also put a return statement in front of res end. This is not required to return a response

but to return from this anonymous function and to not continue this code because we return prior to

it

and this will quit the function execution. And we must do this because I told you that after res end,

we must not call any other res writes or res set headers

but this what happens if we not return because then it would just continue execution with these lines.

I don't want that,

if we make it into the if statement,

we should also quit here,

we should exit out of this function. With this let's restart the server by quitting it with control c and

restarting it with node app.js

and let's reload this page on localhost 3000 slash nothing

and we see my input and the send button here.

Again not super pretty but it's working.

Now let's make sure that if we enter something and we hit send,

we see this, that we now not only see this but that something else happens.

By the way we do see this because now the url is /message and /message does not make

it into this if statement and therefore this code runs.

But we want to do something else

so let's do that in the next lectures.

const http = require('http');

const fs = require('fs');

const server = http.createServer((req, res) => {

  const url = req.url;

  const method = req.method;

  if (url === '/') {

    res.write('<html>');

    res.write('<head><title>Enter Message</title><head>');

    res.write('<body><form action="/message" method="POST"><input type="text" name="message"><button type="submit">Send</button></form></body>');

    res.write('</html>');

    return res.end();

  }

  if (url === '/message' && method === 'POST') {

    fs.writeFileSync('message.txt', 'DUMMY');

    res.statusCode = 302;

    res.setHeader('Location', '/'); //Redirecting the request to this location

    return res.end();

  }

  res.setHeader('Content-Type', 'text/html');

  res.write('<html>');

  res.write('<head><title>My First Page</title><head>');

  res.write('<body><h1>Hello from my Node.js Server!</h1></body>');

  res.write('</html>');

  res.end();

});

server.listen(3000);

**Parsing Request Bodies**

So time to parse the incoming requests and get the data that is part of the request because that data

should be whatever we entered here.

Now how do we get access to that?

Well we get requests url and request method,

you might think there is something like request data but there isn't.

Instead the incoming data is basically sent as a stream of data and that is a special construct javascript

in general knows

but nodejs uses a lot,

now what is such a stream of data though?

There is a connected concept, buffers and we'll have a look at both here.

So let's take our incoming request as an example,

there also are other streams like for example when working with files, we can also work with streams

but let's stick to requests here.

Our stream here is basically an ongoing process,

the request is simply read by node in chunks

you could say, in multiple parts and in the end at some point of time it's done

and this is done so that we theoretically can start working on this, on the individual chunks without

having to wait for the full request being read.

Now for a simple request like the one we're working with, this is not really required,

we only got one input field data,

it doesn't take so long to parse that. But consider a file being uploaded,

this will take considerably longer and therefore streaming that data could make sense because it could

allow you to start writing this to your disk,

so to your hard drive where your app runs, your node app runs on your server

whilst the data is coming in, so that you don't have to parse the entire file which is of course taking

some time and you have to wait for it being fully uploaded before you can do anything with it.

But this is how node handles all requests because it doesn't know in advance how complex and big they

are.

So you can start working on the data earlier,

the problem is with your code, you can't arbitrarily try to work with these chunks. Instead to organize

these incoming chunks,

you use a so-called buffer,

a buffer is like a bus stop.

If you consider buses, they're always driving

but for users or customers being able to work with them, to climb on the bus and leave the bus, you need

bus stops where you well you can track the bus basically and that is what a buffer is. A buffer is simply

a construct which allows you to hold multiple chunks and work with them before they are released

once you're done and you work with that buffer. Now that's pretty abstract but it's pretty easy to work

with fortunately

so let's see how that works in practice. When receiving a posted message before sending the response

and before writing to the file, we want to get our request data, right

and we do this by going to our request and registering an event listener.

We haven't done that thus far but as I mentioned, node uses these heavily. For create server,

it implicitly creates one for us,

now we do this on our own by using the on method. Now on allows us to listen to certain events and the event

I want to listen to here is the data event,

you see my IDE even gives me some help here and tells me which events I can listen to for a request.

So here I want to listen for the data event, the data event will be fired whenever a new chunk is ready

to be read,

you remember that buffer thing, this is basically helping us with that.

Now here we have to add a second argument which is that function that should be executed for every data

event, you remember create server,

it had a similar concept. There we defined a function that should be executed for every incoming request,

now we're defining a function to be executed for every incoming data piece.

So here I'll again using an ES6 arrow function,

you could also use the function keyword without that arrow then

and as you can also see on data, this listener receives a chunk of data.

So here we receive a chunk and this chunk is something we can work with here and now we have to do something

with this chunk to be able to interact with it.

For this I will create a new constant here and I'll name it body because I'll try to read the request

body,

you can name it however you want but it is the request body

we're working with. Now

the body should be an empty array

and now in that function here in the data event, I'll take my body and push a new element onto it.

By the way if you're wondering how we can edit a constant value, this only means that we can never

re-assign a new value, so we can never use body equals something again

but with push we're changing the object behind that body element, that body object, we're editing that

data in that object not the value itself, not the object itself.

It's a bit strange to wrap your head around but this is in the end how it works.

So we can now push a new element into this array to make it non-empty and we push our chunk here. Now nodejs

will execute this so often until it's done getting all the data out of our request,

that can be once, that can be multiple times and we can even log this to see how app, how often it does this

and what's inside of this

chunk. Now we need to register another event listener and that is the end listener, this will be fired

once it's done parsing the incoming requests data or the incoming requests

in general. Here it will again execute a function we define as a second argument and in this function, we

can now rely on all the chunks being read in and they're all stored in the body now.

Now to interact with this

and don't forget the comma after end, to interact with that, to work with all these chunks,

we now need to buffer them.

Remember that bus stop concept,

we get all these chunks we now need to do is something to be able to work with them, to basically have

one place where the bus stops and we can interact with it.

So here I'll now create a new constant, parsedBody and there I will use the buffer object which is available

globally, made available by nodejs

and I can concat my body.

So this will in the end create a new buffer and add all the chunks from inside my body to it.

And then on this buffer which I got here, parsed body is now a buffer, there I can call toString to convert

it to a string.

So this is a utility method offered by nodejs where we do something to our buffered chunks,

remember the bus is now waiting in the bus stop so to say, the buffer is our bus stop and now we do

something with it,

here we convert it to a string and this only works because I know that the incoming data will be text

because the body of that request will be text.

If it were a file, we would have to do something different

but it is no file and I know that it isn't because we're writing the code,

we know what we will receive,

right.

So this is the parsed body

and this is now finally what we can work with,

so let's also output the parsed body.

And this was a lot of talking so let's simply have a look with the server, with control c and restart it and

then send another request with some message here.

And now if you have a look at what's being logged, you see two elements.

The first one is coming from this console log

and you see that is a chunk, that is a chunk we can't work with

but now the parsed body receives or yields this line and that is something we can work with and it's

message equals something because we named our input here message

and as I said, that form will automatically send the request where it takes all the input data and

puts it into the request body as key value pairs where the names assigned to the inputs are the keys

and the values are what the user entered and that is what we have here,

a key value pair separating the key from the value with an equal sign.

Now and with that, we can now work with that and finally store the input in our file and we can do that

here in request on, request on

end to be precise by creating a new constant, message, taking the parsed body and splitting it on the equal

sign and then taking the element with the index one which is the second element in the resulting array

which is the element on the right of the equal sign.

And now we can move write file sync into the end function,

we don't want to execute it here because this will actually run before this function is called

because here we just register a function to be called in the future,

it's not executed immediately,

remember node doesn't wait and pause, it will not block the script execution,

it just registers this as a to-be-executed action

and then right away continues.

So if we have something that depends on the incoming data, we have to move it into the event listener

too so that it's also part of the to-be-executed code

sometimes in the future and doesn't run too early

and now we can write message to the message.txt file.

Let's now restart this file one more time

and enter

hello here and hit send

and now if you look into message text, we see hello, you see the exclamation mark was encoded.

Now that is something we can worry about later

but in general, this worked just fine.

And if you're now totally frightened by how complex nodejs is, this is the raw logic, we'll later

use expressjs which hides all that raw logic

but to understand why we use that, you first of all need to understand what is happening and why using

tools like expressjs which will make all of this much easier are great.

So with that, we've got our basic logic down,

let's now dive again into that event listener and writing files thing because there is something really

important you have to understand.

const http = require('http');

const fs = require('fs');

const server = http.createServer((req, res) => {

  const url = req.url;

  const method = req.method;

  if (url === '/') {

    res.write('<html>');

    res.write('<head><title>Enter Message</title><head>');

    res.write('<body><form action="/message" method="POST"><input type="text" name="message"><button type="submit">Send</button></form></body>');

    res.write('</html>');

    return res.end();

  }

  if (url === '/message' && method === 'POST') {

    //read the request data in chunks

    const body = [];

    req.on('data', (chunk) => {

      body.push(chunk);

    });

    req.on('end', () => {

      const parsedBody = Buffer.concat(body).toString();

      const message = parsedBody.split('=')[1];

      fs.writeFileSync('message.txt', message);

    });

    //fs.writeFileSync('message.txt', 'DUMMY');

    res.statusCode = 302;

    res.setHeader('Location', '/');

    return res.end();

  }

  res.setHeader('Content-Type', 'text/html');

  res.write('<html>');

  res.write('<head><title>My First Page</title><head>');

  res.write('<body><h1>Hello from my Node.js Server!</h1></body>');

  res.write('</html>');

  res.end();

});

server.listen(3000);

**Understanding Event Driven Code Execution:**

We already achieved a lot in this section, and I know that all this code looks kind of intimidating

and no worries.

It will become much easier, but I find it super important to learn it the hard way first so that you

never forget what's happening behind the scenes.

Now, one crucial thing which I know that people often struggle with is that the order of execution

of your code here is not necessarily the order in which you write it.

For example, this here will actually execute after this code, so it will even execute after we already

sent a response.

This has two important implications.

For one, sending the response does not mean that our event listeners here are dead.

They will still execute, even if the response is already gone.

But it also means that if we do something in the event listener that should influence in the response,

this is the wrong way of setting it up.

We should then also move the response code into the event listener if we had such a dependency.

But it also means that it's super important to understand that with request on or code like HTTP create

server.

These are some examples where no case uses a pattern where you pass a function to a function and node

will execute these past in functions at a later point of time, which is called asynchronously.

Now, it's not always the case that a past in function is necessarily executed at a later point of time.

But no case uses this pattern heavily and throughout the course.

I'll, of course, let you know when this is the case and when Node executes something asynchronously.

In such cases, no charges won't immediately run that function.

Instead, what it does when it first encounters this line is it will simply add a new event listener

internally.

It manages all these listeners internally.

In this case, for the end event on the request, which will be triggered automatically once no chars

is done parsing the request.

So this is something no trace does for you.

And it will then call that function for you once it is done.

So in the end, you can think of this like no charge as having some internal registry of events and

listeners to these events.

And a function like this is such a listener and when sort of something happens, so when Node.js is

done parsing your request, it will go through that registry and see I'm done with the request, so

I should now send the end event.

So let's see which listeners I have for that.

And it will then find this function and any other functions you might have registered for that and will

now call them.

But it will not pause the other code execution and that is so important to understand.

So for example, here now since I moved return response and into this function, the flow is like this.

It will now reach this if statement and if these conditions are met, it will go inside of it.

It will then register these two handlers and not immediately execute these two functions.

Instead, the functions are just registered internally in its event emitter registry and then it will

jump straight away to the next line.

And therefore right now if I would restart my server, save the code and restart my server here.

You will see that if I enter something here, I actually get redirected to this page or not even redirected.

As you can see, there is no 300 status code.

Instead, it just loads this page because it executes these lines because as a now mentioned multiple

times, it will not execute this right away and this return statement will therefore not quit this overarching

function here.

Instead, it just registers this callback and immediately moves on to the next lines and it would eventually

execute this line.

But that is already too late, which is also why we get the cannot set headers error here because it

already moved along and executed this code when all of a sudden the parsing of the request finished

and it executed this code and tried to again send a response which obviously is too late because it

already did here.

Now, I know that this is hard to wrap your head around, but it is a crucial concept that you can register

code functions which run sometime in the future, but not necessarily right now.

And therefore the next line of code, this code here can run or will run before this code, simply because

this is just a callback to be called sometime in the future.

And this setup is important because otherwise node would have to pause until it's done, pause until

it wrote the file, and if it does that, it will simply slow our server down and it's not able to handle

our incoming requests or do anything of that kind until it's done.

And that is not what we want.

We don't want to block our code execution.

We always want to be in that wait for new events, loop the event loop and then only execute code once

it's due to be executed and never block that event loop for too long of a time.

And this is why we have this setup and this has one implication for this line and for this line.

The implication for this line is that we reach it too early.

So to avoid this, we should actually return here.

We simply return requests on so that this gets executed, but the line thereafter doesn't.

And the important application about this line will be discussed in the next lecture.

const http = require('http');

const fs = require('fs');

const server = http.createServer((req, res) => {

  const url = req.url;

  const method = req.method;

  if (url === '/') {

    res.write('<html>');

    res.write('<head><title>Enter Message</title><head>');

    res.write('<body><form action="/message" method="POST"><input type="text" name="message"><button type="submit">Send</button></form></body>');

    res.write('</html>');

    return res.end();

  }

  if (url === '/message' && method === 'POST') {

    //read the request data in chunks

    const body = [];

    req.on('data', (chunk) => {

      body.push(chunk);

    });

    return req.on('end', () => {

      const parsedBody = Buffer.concat(body).toString();

      const message = parsedBody.split('=')[1];

      fs.writeFileSync('message.txt', message);

      res.statusCode = 302;

      res.setHeader('Location', '/');

      return res.end();

    });

    //fs.writeFileSync('message.txt', 'DUMMY')

  }

  res.setHeader('Content-Type', 'text/html');

  res.write('<html>');

  res.write('<head><title>My First Page</title><head>');

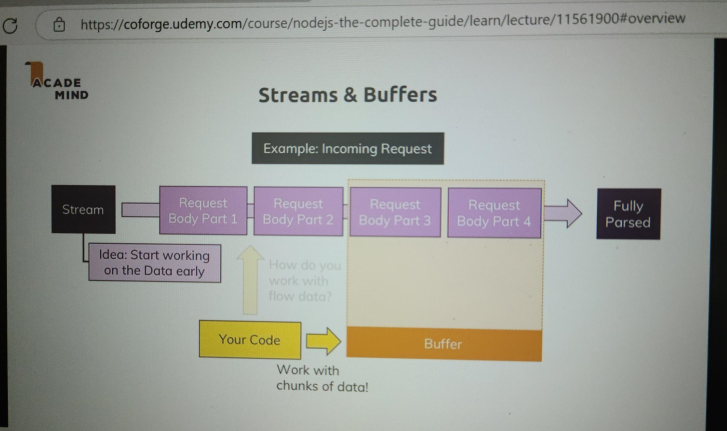
  res.write('<body><h1>Hello from my Node.js Server!</h1></body>');

  res.write('</html>');

  res.end();

});

server.listen(3000);

****

**Blocking and Non-Blocking Code**

So what is wrong with this write file sync line here?

Well it's the sync keyword.

I earlier mentioned that there also is a write file method but we use write file sync, the sync here

stands for synchronous and this is a special method which will actually block code execution until this

file is created.

Now working with files is available in two modes,

here is the synchronous mode and we block execution of the next line of code until this file is done.

Now for this short of a text we enter, this is super fast and we won't even notice it

but think about a huge file, a couple of hundreds of megabytes or even bigger, if you do something with

that,

read it, copy it, whatever it is and you block the code execution,

then the next line and all the other code will not continue to run until that file operation is done

and even new incoming requests of other users would not be handled until that file operation is done

and you don't want that, therefore you should not use that syntax here.

You can if you know you'll only have a very short file operation but even then, it's better to use this

write file method here which actually does not just accept the path and the data but also a third argument

and that is again such a callback,

so a function that should be executed when it's done.

So here again just as with create server, nodejs implicitly registers an event listener for us.

So here, I pass another function and this callback here actually receives an error object which will

be null if no error occurred

but if some error occurred, missing permissions or anything like that, you would get it here and you could

then handle it gracefully by returning a different kind of response, an error response showing to the user

that an error occurred

and otherwise return a normal response. Now here

I won't do error handling and there is really nothing that could go wrong here,

we'll of course dive into error handling a bit more later in the course

but I will move my normal response code in there because this response should only be sent if we're

done working with the file because that is ultimately the action

we want to do upon our request, right.

So now we actually have our event listener with some method or function that will be executed once we're

done parsing the request

and in that function that will be executed sometime in the future,

we have yet another event listener, this nested function here which will be executed once we're done

writing the file and this is actually pretty standard for nodejs.

You have this event driven architecture where you basically tell nodejs please do something and

it will then go ahead and offload that process to the operating system which does use multi-threading

and so on and will then continue its event loop to listen for event callbacks and always just dispatch

tiny actions like that to never block the code execution and then always just come back once an operation

is done by the operating system and so on.

So this is what nodejs does here and why it is high performant

because it never blocks your code,

it never blocks the server,

it just goes ahead and tells the operating system do that, do this and then eventually comes back and

does something in the callback, like send a response which is not a blocking operation because this

is super fast, just a couple of headers and off you go.

So I hope this is clear

and with that let's restart this by quitting the server, re-running node app and going back to just slash

nothing,

let's enter hello again here and send. And in message.text we see that, again concatenated with an extra

character, not something we have to worry about right now.

So you see write file also works

but this is the way you should use it due to the asynchronous nature of nodejs which I hope now

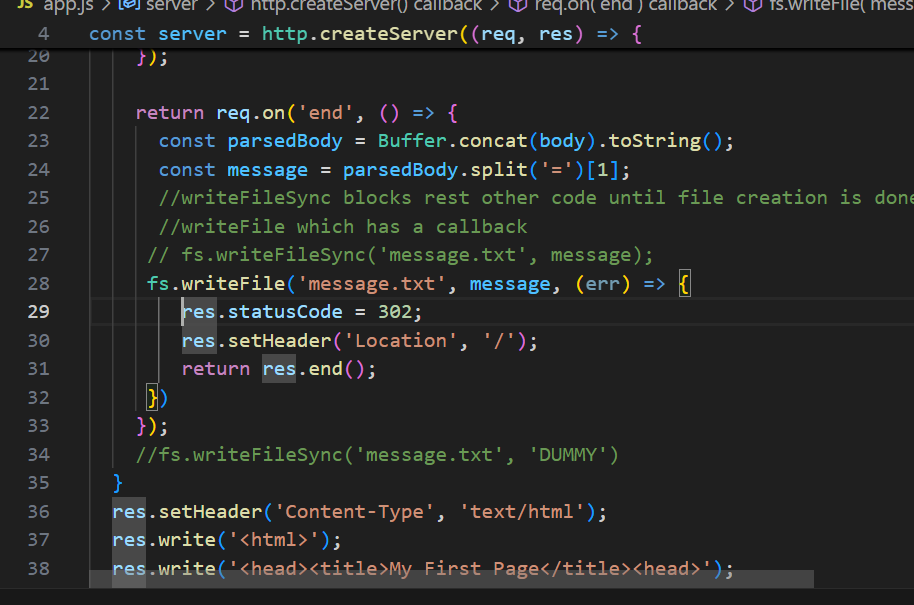
became clear because it is a crucial concept of nodejs and one you absolutely have to understand.

With that, let's wrap this module up with these core basics and let's from now on to move to a more easier

route where we will make our life simpler and not write

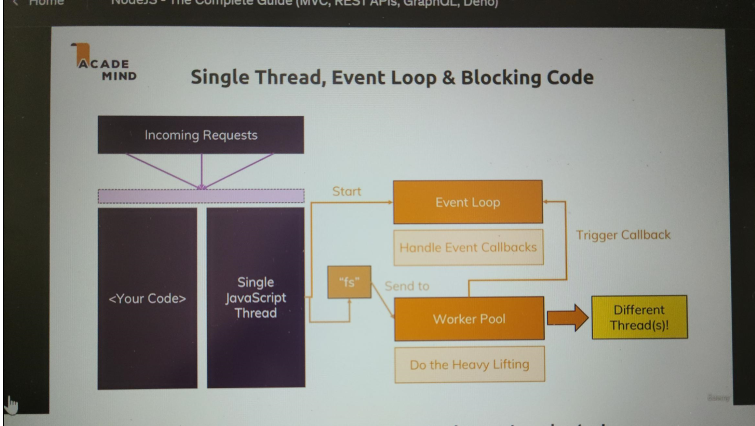
all this nitty gritty detail code on our own

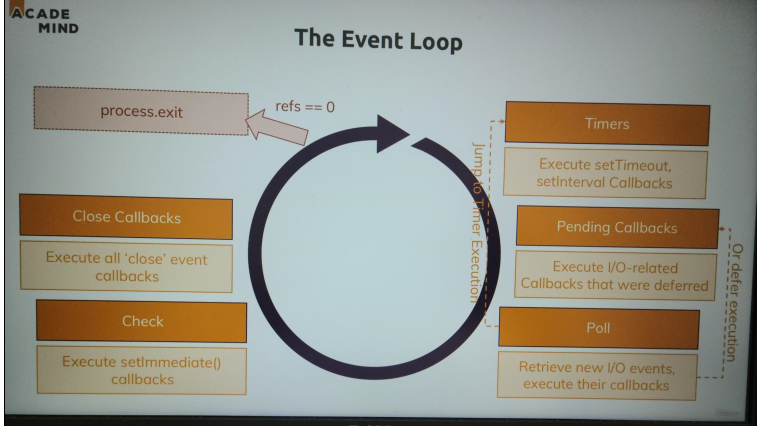
but it is super crucial that you understand what's going on here.

Top of Form

Bottom of Form

**Node.js - Looking Behind the Scenes**

****

****

Over the last lectures, I extensively mentioned how node works with asynchronous code

and i mentioned this event loop which is important.

Let me dive deeper into that and deeper into how

nodejs exactly executes your code to stay performant and still be able to handle long taking tasks

like working with files or things like that because there is an important concept which you have to

understand to understand, well what is happening behind the scenes.

So in our node application we have our code

and one important thing to understand and to really keep in mind is that nodejs uses only one single

javascript thread,

a thread is basically like a process in your operating system you could say.

So it's only using that one thread and the obvious question is how is it then able to handle multiple

requests because if we're not able to assign a new thread for each request, they ultimately end up all

running in one on the same thread

and this of course poses a security question,

can you then access data from request A, from request B and most importantly here at this point when

we talk about performance, there of course also is the question of does this not mean that if the

request A is still doing work, request B can't be handled? Well both is taken care of

nodejs

.

and at this point of time here, I want to focus on the performance question though I'll also briefly

touch on the security question.

Now let's start with the performance.

Let's say we have some code which accesses the file system as we already did in this course too. Now working

with files often is a task that takes longer because files can be very big and it doesn't necessarily

complete instantly,

therefore if we're doing this upon an incoming request, a second request might have to wait because we're

not able to handle it yet or it even gets declined,

so basically our webpage is down for that user.

Now one important construct I already mentioned is that event loop, the event loop is automatically started

by nodejs when your program starts,

you don't have to do that explicitly,

nodejs does that when well

it basically starts running code.

This is responsible for handling event callbacks though,

so all these nice functions we basically added thus far in create server for example, the event loop

is responsible for basically running that code when a certain event occurs you could say, it's aware

of all these callbacks and basically well, execute said code. That doesn't help us with our long taking

file operation though and it's important to understand that this operation is not handled by the event

loop, just the callback that we might have defined on write file

once it's done, that code will be handled in event loop but that code will finish fast,

so basically the event loop will only handle callbacks that contain fast finishing code.

Instead our file system operation and a couple of other long taking operations are sent to a worker

pool which is also spun up and managed by nodejs automatically.

This worker pool is responsible for all the heavy lifting,

this worker pool is kind of totally detached of your javascript code you could say and it runs on different

threads, it can spin up multiple threads, it's closely intervened with your operating system you're

running the app on,

so this is really detached from your code and this worker pool is therefore doing all the heavy lifting.

If you're doing something with a file, well a worker from that pool will take care and will do its job

totally detached from your code and from the request and from the event loop. The one connection to the event loop

we will have though is that

once the worker is done, so for example once we read a file, it will trigger the callback for that read

file operation and since the event loop is responsible for the events and the callbacks, this will in

the end end up in the event loop,

so there nodejs will then basically execute the appropriate callback.

Now this is a lot of behind the scenes stuff which is nice to know, you don't have to write any code

to make this work,

this is all built into nodejs

and if you write code as you learn it in this course, you automatically take advantage of this.

I still of course want you to understand how that works behind the scenes because I believe that helps

you understand

nodejs better.

Now let's look into that event loop

then. That event loop is in the end

a loop which is run or started by nodejs that keeps the nodejs process running

and as I just mentioned, that handles all the callbacks and it has a certain order in which it goes through

the callbacks.

So basically it's a loop that just well keeps on looping

unsurprisingly, at the beginning of each new iteration it checks if there are any timer callbacks it

should execute. We haven't set up any timers yet

but basically there is set timeout and set interval,

you might know this from frontend javascript too, there these methods also exist. Now in nodejs you can

also set a timer and basically you set a timer and always pass a method, a function that should be executed

once that timer completes and nodejs is aware of this and at the beginning of each new loop iteration,

it executes any due timer callbacks,

so any callbacks that have to be executed because a timer completes. Then as a next step, it checks other

callbacks,

for example if we had write or read file, we might have a callback because that operation finished and

it will then also execute these callbacks.

Now be aware that with IO here, I mean generally any input output operations that typically is file

operations but can also be network operations and in general, I'm talking about blocking long taking

operations.

Now it's important to understand that nodejs will leave that phase at a certain point of time and that

can also mean that if there are too many outstanding callbacks, it will continue its loop iteration and

postpone these callbacks to the next iteration to execute them. After working on these open callbacks

and hopefully finishing them all,

it will enter a pull phase. The pull phase is basically a phase where nodejs will look for new IO events

and basically do its best to execute their callbacks immediately if possible.

Now if that's not possible, it will defer the execution and basically register this as a pending callback,

so this is how that works.

Important, it also will check if there are any timer callbacks due to be executed

and if that is the case, it will jump to that timer phase and execute them right away,

so it can actually jump back there and not finish the iteration

otherwise it will continue

and next set immediate callbacks will be executed in a so-called check phase. Set immediate is a bit like

set timeout or set interval, just that it will execute immediately but always after any open callbacks

have been executed,

so typically faster than set timeout with one millisecond of open duration, let's say

but after the current cycle

well finished or at least finished open callbacks that were due to be handled in that current iteration.

And now we're entering a highly theoretical terrain

and I don't want to dive too deep into that, though you will find a couple of resources that do dive

deeper at the end of this module in case you want to join the nodejs team and really dive super

hardcore into this.

Now we're nearing the end of each iteration cycle and now nodejs will execute all close event callbacks,

so if you registered any close events and in our code, we haven't

but if you had any close events, this would be the point of time where nodejs executes their appropriate

callbacks.

So roughly spoken, we have timer callbacks, we then have any IO related callbacks and other event callbacks

and set immediate followed by close event callbacks,

so close events are basically handled separately or their callbacks are handled separately we should

say. Well and then we might exit the whole nodejs program but only if there are no remaining event

handlers which are registered and that is what I mean with this refs equal null thing here. Internally

nodejs keeps track of its open event listeners and it basically has a counter, references or refs

which it increments by 1 for every new callback that is registered,

every new event listener that is registered

so every new future work that it has to do

you could say and it reduces that counter by 1 for every event listener that it doesn't need anymore,

every callback it finished and since in a server environment

we create a server with create server and then listen to incoming requests with listen,

this is an event which never is finished by default and therefore, we always have at least one reference

and therefore we don't exit in a normal node web server program. We can call the exit function as you

already saw

and if we do anything else like at the beginning of the course when we just used node to execute a file

that did not listen to a web server or on a web server, then it also finishes eventually once its done

with its work.

So this is how the event loop works

and this can be very theoretical, all these different phases typically don't matter to you, certainly

not when getting started with nodejs.

Now consult these advanced resources if you want to learn more,

for now just be aware of that first picture I showed you, of how nodejs handles complex work, that

there is this worker pool that takes care that your things are not getting blocked and that it therefore

stays performant.

Now I also touched on that security thing with that one javascript thread

and regarding that, we also got no problems by default,

though I will come back later to how we can manage global data and how we should manage it to ensure

that we do not spoil data from request A to request B. By default

we have some separation because remember that callback method in create server gets executed for every

new incoming request and therefore this function only runs for that incoming request and anything we do

to the request or response object there will not be exposed to our request or response objects because

each function is only scoped to itself and not accessible by the other functions,

so by default we have that separation due to how javascript works.

Now that was a big chunk of theory,

again not something you have to learn by heart but something where you should at least keep that big

picture in mind of how nodejs manages its work.

**Using the Node Modules System**

So let's wrap this module up

and right before we finish, let's actually start wrapping it up by improving our code a little bit.

We've got all this code in this file and actually typically you write multiples or work with multiple

files

and why don't we create a new file that actually contains our routing logic,

so the logic where we check the url and do different things.

So I'll create a new file here, routes.js,

the name is up to you and I get this special icon by my IDE,

it is a normal javascript file,

don't be confused.

And in that file, I basically want to have my if statements here and my default response code,

so I'll cut all of that out of here so that this is a pretty lean file and move it into routes.js.

Now this wouldn't work like that,

let me say that, we'll have to tweak that

but this is now the code moved over.

I don't need the file system in

app.js anymore so we can remove that import,

we do need http because we still use that there and we don't need the url and method here so let's

remove that, go over routes.js,

add this fs imported on top

and now start working with that.

Now what do we need to do in this file?

We somehow need to be able to connect app.js to routes.js,

right

because we need to be able to send our incoming request to that file

so to say. And for that, let's create a new function, I'll name it

request handler and we can either create a function like this,

it should receive request and response as arguments, just as this function does because we'll effectively

replace that function or we use an ES6 function,

storing it in a constant, request handler which looks like this. Bit of a strange syntax if you've never

seen it before but we're essentially creating an anonymous arrow function which we store in a constant

and this therefore is the function name.

Now here we again receive request and response and in that function, we now move all that code because

that code obviously uses the request and response object,

so we need to have them available as local variables and we do now because these arguments are now named

this way.

We also use url and method,

so we need to re-add these constants by getting that data out from the request, request url and request

method and now we just need to export this handler. We're importing with this require syntax

but how are we exporting in nodejs?

There are two ways of exporting,

the first one is to go at the bottom and add module.exports,

this is another keyword or object which is exposed globally to you by nodejs which has an exports

property

and we can assign a value to this, like our request handler,

so this constant which holds this function, it's now stored in module exports.

And since this is a global object exposed by node, node is actually able to work with this

and we can now import from that routes.js file by requiring it and node will look for module exports

and see if something was registered for this file here

and we do register something in module exports, the request handler and you can register anything here.

You can add a new javascript object with multiple key value pairs, whatever you need, here

I'll just register my function.

So now I can go back to app.js and import my routes, the name of that constant is up to you,

by requiring it

and since this is now not a global module, we don't just type routes,

instead we want to add a local path to it with ./ and you can omit .js because nodejs

will automatically attach this at the end.

You can add it though

but I'll just type

./routes separated from the core modules to make it really clear that this is a custom

file

and now node will go ahead and look for a routes.js

file in the same folder as app.js which it will find and in that file, it will look for module

exports and see what's registered in there

and now we export that request handler method and now we can use that,

it will be stored in that routes because we assign whatever is exported from that file in that routes

constant,

so this routes constant will ultimately hold this function

and now we can use that here as a handler,

routes. Don't execute it,

so no parentheses, just pass the name telling node hey please execute the function that's stored in

routes for incoming requests.

And now if we save that and we restart the server and we reload this page, this is looking good

and this is also looking good,

we should have tests stored in message.text and we do.

So now we simply split our code over two files, having one file which is very lean that just spins

up the server

but and that's important,

that also creates a connection to another file through that import and through that export where we

export our request handler function here.

This is how that works,

now one important note about nodes module system, the file content here is actually cached by node and

we can't edit it externally,

so if we somehow would define routes as an object and we tried to add a new property on the fly here, this

would not manipulate the original file,

so this is basically logged, not accessible from outside,

we can only export stuff that we can now read from outside. Though

you could have functions which you export that start changing stuff inside of that file

but let's not make that too complicated for now,

we'll see all of that throughout the course obviously. For now we have that connection,

there's one other syntax you could use, instead of module exports,

sometimes you export many things and you could do that by having an object which has like the handler

key and that is my request handler function

and then also it has some text key which is some hardcoded text

in this case, now we would export two things and that is how you can group that or separate these two

things and still have one export being managed here which is the most you can have

and now in app.js, routes would be that object and not that function.

So here we would have to access the handler property,

so this property which holds the function reference we want to use and we could also simply output console

log routes some text here.

So this is how we can have multiple exports in one file,

now you see some hardcoded text here from this console log and we still have the same functionality

as before.

Now there also is a different way of exporting multiple things,

besides this code which you can of course use, you can also have module.exports.handler request

handler and then also module.exports. some text, some hardcoded text.

Now it might look different but we still only have one export,

we still have module exports which bundles all the exports but we explicitly assigned the different properties

like this,

so this is basically equivalent to this code.

If I now save this and restart my server, we see some hardcoded text and if I submit this, it also

still works.

Last but not least, there is a shortcut for this syntax,

you can now omit module and just write exports,

this is simply a shortcut supported by nodejs, not some general javascript magic,

it's just an explicit shortcut supported by nodejs where now we also have multiple exports being merged

together into one export,

so therefore again when I execute this,

we see some hardcoded text and some last value being submitted here still works and still ends up in

message.text.

So this is how we can connect multiple files by exporting either one element, one function as we had

it initially with module.exports equals request handler,

right

this is what we had initially,

let me comment this out with two slashes in front of it or module exports pointing at an object to combine

multiple things or using module.exports.handler equals request handler and module.exports.

some text equal some text or again,

this shortcut which is the same as this, just with this shortcut offered by nodejs.

So this is how imports and exports work, how the module system works and we'll work with a lot of files

throughout the course

so this is important to understand.

With that out of the way, let's now finally wrap this module up.

**Routes.js**

const fs = require('fs');

const requestHandler = (req, res) => {

    const url = req.url;

    const method = req.method;

  if (url === '/') {

    res.write('<html>');

    res.write('<head><title>Enter Message</title><head>');

    res.write('<body><form action="/message" method="POST"><input type="text" name="message"><button type="submit">Send</button></form></body>');

    res.write('</html>');

    return res.end();

  }

  if (url === '/message' && method === 'POST') {

    //read the request data in chunks

    const body = [];

    req.on('data', (chunk) => {

      body.push(chunk);

    });

    return req.on('end', () => {

      const parsedBody = Buffer.concat(body).toString();

      const message = parsedBody.split('=')[1];

      //writeFileSync blocks rest other code until file creation is done. so we should use

      //writeFile which has a callback

     // fs.writeFileSync('message.txt', message);

     fs.writeFile('message.txt', message, (err) => {

        res.statusCode = 302;

        res.setHeader('Location', '/');

        return res.end();

     })

    });

    //fs.writeFileSync('message.txt', 'DUMMY')

  }

  res.setHeader('Content-Type', 'text/html');

  res.write('<html>');

  res.write('<head><title>My First Page</title><head>');

  res.write('<body><h1>Hello from my Node.js Server!</h1></body>');

  res.write('</html>');

  res.end();

}

//module.exports = requestHandler;

//while exporting more than 1 item

module.exports = {

    handler: requestHandler,

    someText: 'Some hard coded text'

}

**App.js**

const http = require('http');

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

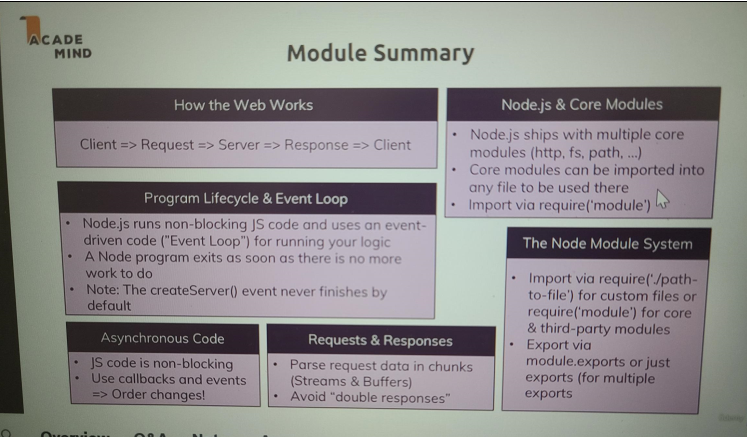
//const server = http.createServer(routes);

//multiple exports extraction

const server = http.createServer(routes.handler);

server.listen(3000);

**Wrap Up**

****

**Understanding NPM Scripts (npm init – to create as a project)**

For this module I'm back in the application or the project we worked on in the last core section and you

can find my state attached to this video so that you can start right at the same point as I do.

So it is just the code we wrote over the last lectures

and one thing we constantly had to do in the last section or throughout this course was that we always

had to run node app.js to start our well app or our node application with that file.

Now this is certainly ok and not too much work but it actually is possible to define some scripts in

a nodejs project that can help us with tasks like this,

we can also use them for other tasks but especially for this, they can be useful right now

and for that we have to use a feature we didn't use thus far.

We have to use npm, npm stands for node package manager and it is installed together with nodejs, so

you don't need to do anything,

you got this installed already.

Now we will also use npm to install some additional third party packages to our project soon, so packages

that are not included in nodes core,

so not part of nodes core modules but we can also use npm to initialize a so-called node project or

to add some extra features to it

to be precise because we obviously already got a node project here but now in this project, in a terminal

navigated into this project, you can run npm init.

Now this will not delete your code or anything like that,

no worries,

simply hit enter and you'll be prompted with a couple of questions here.

It'll ask you first of all for a name of your package

and for now you can simply translate this with project name.

Now you can pick any name you want, the part in the parentheses here

always is the suggestion, the default

it will pick if you don't choose your own name,

so if I just hit enter here, it will take that default.

So you can basically do that for all these questions,

for the description, you can leave that empty

then it will have no default text either but you can also enter some text like complete nodejs

guide, this is what this course is and then define an entry point with fs for example. You can leave

the task command empty, leave that empty, keywords you can choose some if you want,

you can put your name into the offer field though this is also not required and you can always choose

a license

but if you don't plan on sharing this project publicly,

this also doesn't really matter.

So with this what you get is this package.json file and there you also see all these settings

or configurations you just set up and you can of course edit them there too,

so if you had a typo in your description, you can just edit it here.

Now this is using the json format which is a special kind of data format which basically looks a

lot like javascript objects and it pretty much is based on that,

there is one important thing to understand though,

the keys are always put between double quotation marks and so are the values, except for numbers or arrays

or true or false which are not put between these

but that's too much

for now, we can ignore that for now just so that you understand what you got there,

it's basically a configuration file for your project.

Now what does this configuration file give you?

Well let's clear the console here,

with this configuration file, you'll see that we got a scripts section there which has one default script

that won't do anything for now,

you can add your own scripts here and I will tell you how to execute them too of course.

For that, let's add a comma after this test script and add a new script name which has to be put between

double quotation marks and there, let's name it start. Now

start is actually a special script name as you will see in a second,

so make sure to type this correctly and then between the quotation marks,

you type a command that should be executed.

So this is a command which you could also type down there

and there we always have to type

node app.js,

so let's now put that between these double quotation marks here

node app.js,

like this.

With that save that file

and then you can run npm start. Start is a reserved name and this will always look for such a start

script here.

And if you do that, it will as you can see here just execute node app.js.

So it does the same you had before but now you can always just well run this command instead of running

node

app.js. Not that much of a saved characters but a few at least

and it's also a good practice because if you ever share this project,

it's pretty common that people just have to run npm start and that they don't have to guess which of your

javascript files is the entry file.

So you can quit this with control c of course as always and I mentioned that this would be a special

script name.

You can add more scripts, also without using a special name,

you can indeed choose any name you want,

just make sure to always wrap the name in double quotation marks and that it does not contain any blanks

or whitespaces.

So for example we could have start-server.

Now this can also run node app.js,

so it will do the exact same as this script and therefore it's of course redundant

but I want to show you something

and now if you try running npm start-server, you'll get an error.

You basically get an error that is not a known command and indeed it isn't because just typing

the script name here will not work,

start

just was the special case as I mentioned.

Indeed for normal scripts with their own custom names, you have to run npm run and then your script name,

so npm run start-server will now also start the server.

Now as I mentioned, this is of course redundant, npm start is way shorter but I want you to understand

how you can add your own scripts.

Now if you worked with something like angular or react or vue or any modern frontend development workflow,

you will have seen that you use such scripts a lot to trigger build workflows for your projects for

example and indeed you can use that for all kinds of tasks to want to execute

but for now we'll not dive deeper into that and if you haven't worked with angular or react, it's

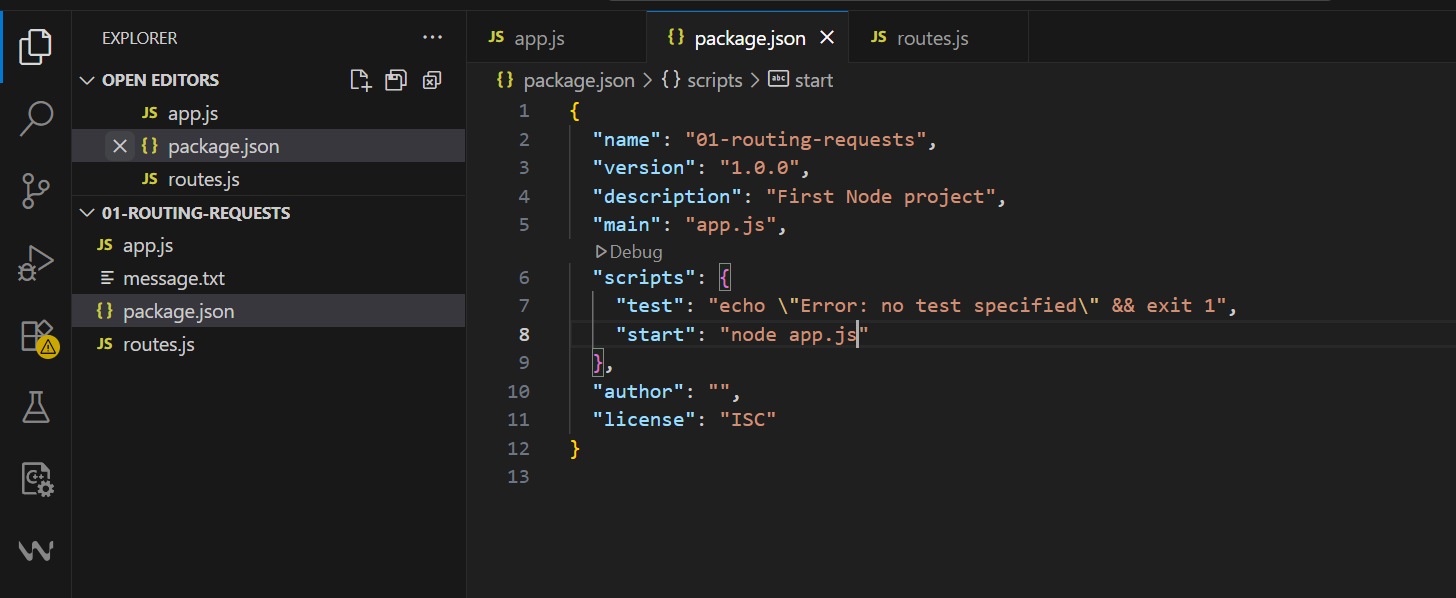
also

no problem,

you will see what I mean

later in this course when I explore node's functionality as a build tool a bit more.

For now let's just use that npm start script to start our application conveniently.



**Installing 3rd Party Packages**

npm install nodemon --sav-dev <- for dev

npm install nodemon –sav <- for PROD too

So in the last lecture, we added a script to start our application and this is nice to know that we

have this scripting functionality. Now with such a package.json file available,

so therefore with this being a managed node project you could say, we can also do one other very important thing,

we can install third party packages because a typical node project looks like that. You have your local

project with your code obviously and you use a lot of core node packages like the fs package or the http

package we're already using

but often this does not suffice

and in the next core section, we will install one big third package actually because you typically have

such dependencies third party packages.

So you want to use some functionalities some code which you didn't write on your own but which is also

not included into nodejs. Packages could help you with parsing incoming requests, validating user input,

anything of that kind.

Now we will use express in the next lectures or in the next module to be precise, body parser is another

package we'll use throughout this course and there are thousands of packages available that offer all

kinds of utilities you can add to your projects

so that you don't have to reinvent the wheel.

These packages are available through the npm repository,

that is a cloud package repository where all these packages live

and you can conveniently install and manage them via npm,

remember that tool that shipped with node. And this is exactly what we will do now to install a first

little utility package that will speed up our development workflow.

Because right now what we have to do is whenever we change our code, we have to quit the development

server with Control-C and restart it,

right,

so if we have it running with npm start now, remember this is our new command to start the server

and I would change something here, like for example here I fixed that /head right,

whenever I do this I have to save this and for my change to have an effect, I have to quit the server

with Control-C and restart it.

Now this is a bit cumbersome because we just want to be able to type and then hit save

and it should automatically restart and right, that would be a great workflow during development and

to achieve this, we have to install a third party package that gives us just this functionality.

Now how do we add such a third party package?

We do that with the help of npm and there, we get the install command,

so just as we had run to run one of our scripts, install installs a third party package.

Now how do we install it?

First of all we have to know the package name

and if you're wondering well how do I know the package name,

well that comes with courses like this, experience or by simply googling for certain problems which

you want to have solved and all of a sudden you find a thread where some package is mentioned.

Now I can tell you for this auto-restart mechanism,

there is a package called nodemon, written like this.

By the way you can always search for npm and then the package name if you know that name and you will

find an entry on npmjs.com.

Now this is the package page basically where you find a description, installation instructions, usage

instructions, how to configure it because most packages give you an easy way of using it and then always

give you configuration possibilities.

You'll see how popular the package is,

what's version it's using, if there is an exernal little home page, where the source code can be found if

it's open source.

So you find a bunch of stuff here, pretty useful,

you also see how many versions are existing

and by default, you will always install the latest version by the way

but let's go back to installing it

before we dive deeper into this whole npm thing.

So we want to install it and this command would install it

but don't hit enter yet,

you can define how this should be installed because packages which you install can be divided into development

packages, so packages which mostly help you during development and production dependencies,

so packages that helps you for the app as it's running on a server, for example nodemon would be a

development dependency because we only use it during the development process,

once we install our app on a real server we don't need it there.

The real server which is running somewhere in the Internet of course shouldn't restart

and it also doesn't have to because we'll not change its code dynamically.

And you can basically tell npm which kind of dependency this is, this does not make a huge difference

and you can omit the setting but it helps you understand which package is used for what. Now you do add this

by adding --save-dev,

if you had just save like this, this would install it as a production dependency,

so a package which we really use and use in our code and work with and with this we're indicating that

this only adds something we used during development.

There also is a third option by the way,

-g, we'll not install it in this project but globally on your machine

so that you can use it anywhere.

Now let's first of all install it with save dev, like this --save-dev.

Now what this will do is it will download it from the npm repository and install it into this project,

so not globally on your machine but into this project.

So now you see you get a report here that it finished successfully, what it did and it gives you a couple

of new things in your project.

It gives you that node modules folder, the package log json file and it updated the package.json

file.

There we see that the new dev dependency section was added and that stands for development dependencies,

as I said you can differentiate between different dependencies,

we'll see production dependencies later too

and there you see that nodemon was installed and which version was picked.

Now regarding that character here,

well this basically defines how this package will be updated

if you rerun just npm install, without defining an extra package name because this command standalone

will simply go through all your packages mentioned in package.json and install them and it would automatically

pick a later version if available

but more on npm and packages can also be found in a separate module

later in the course.

So this is basically how we now install this

and the question is where is it installed?

Well that is the node modules folder

and actually that is a huge folder as you can tell. The reason for this is that

for one we got nodemon in there,

if we look for n, we see it here.

Now this is basically the source code of the package or the build version of the package we installed

and this package simply happens to have a couple of peer dependencies,

you can see them here and here.

So we got a bunch of dependencies in there and these and their dependencies are also installed,

that is why you could end up with quite a big node modules folder but you can always delete that node

modules folder if you need to free up space.

Now you can't use that package but you can then rerun npm install if you start working on that project

again and it will re-install this package and all its pure dependencies and therefore recreate the node

modules folder,

this is how packages work in node projects.

So you need that node modules folder while still using the packages but if you're not working on the

project, you can delete it if you want, if you need the free space and then just remember to rerun npm

install once you are working on the project again.

The package log json file by the way just stores the exact versions I installed today so that if you

share your project with others, they can actually get these exact versions too instead of the latest versions

but again, more on npm in a separate module.

**Global Features vs Core Modules vs Third-Party Modules**

The last lectures contained important concepts about available Node.js features and how to unlock them.

You can basically differentiate between:

* **Global features**: Keywords like const or function but also some global objects like process
* **Core Node.js Modules**: Examples would be the file-system module ("fs"), the path module ("path") or the Http module ("http")
* **Third-party Modules**: Installed via npm install - you can add any kind of feature to your app via this way

**Global features** are **always available**, you don't need to import them into the files where you want to use them.

**Core Node.js Modules** don't need to be installed (**NO npm install** is required) but you **need to import them** when you want to use features exposed by them.

Example:

const fs = require('fs');

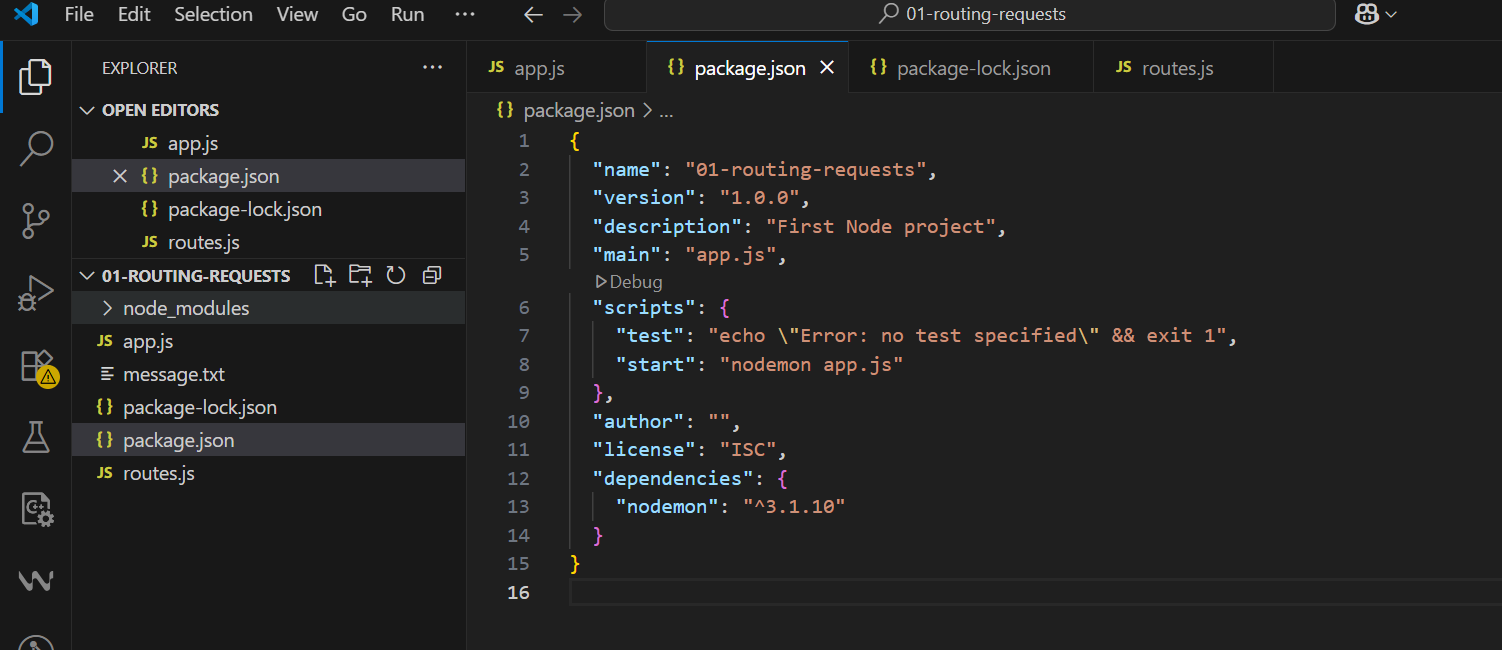
You can now use the fs object exported by the "fs" module.

**Third-party Modules** **need to be installed**(via npm install in the project folder) **AND imported**.

Example (which you don't need to understand yet - we'll cover this later in the course):

1. // In terminal/ command prompt
2. npm install --save express-session
3. // In code file (e.g. app.js)
4. const sessions = require('express-session');

**Using Nodemon for Autorestarts**



So for now we got that nodemon package installed and it will just work because it automatically installed

all the dependencies

this package needs in turn.

So how can we now use it?

Nodemon is a utility tool and it allows us to run our application, our node application through this

package here which will run node apps.js

in the end but which will also watch our files for changes and restart the process for us if

we do change something. So we can simply change start here,

so the node app.js command to nodemon app.js

and this will look for a nodemon tool which it will find in this project because we installed it

here. As a side note

if you were to run nodemon app.js down there, you would get an error that this command is not found

because it's only installed in this project and not globally on your machine but the terminal will try

to find this globally. Here

it will work because this will look globally.

So if you now run npm start, this will simply start the node server and output some extra information

and if you now go to your routes.js file and edit something,

let's just add an extra line and you save that file, you see it's restarting.

And this of course is very convenient because now, we dont have to spend time on manually exiting and

restarting,

now is this done for us.

**Global & Local npm Packages**

In the last lecture, we added nodemon as a local dependency to our project.

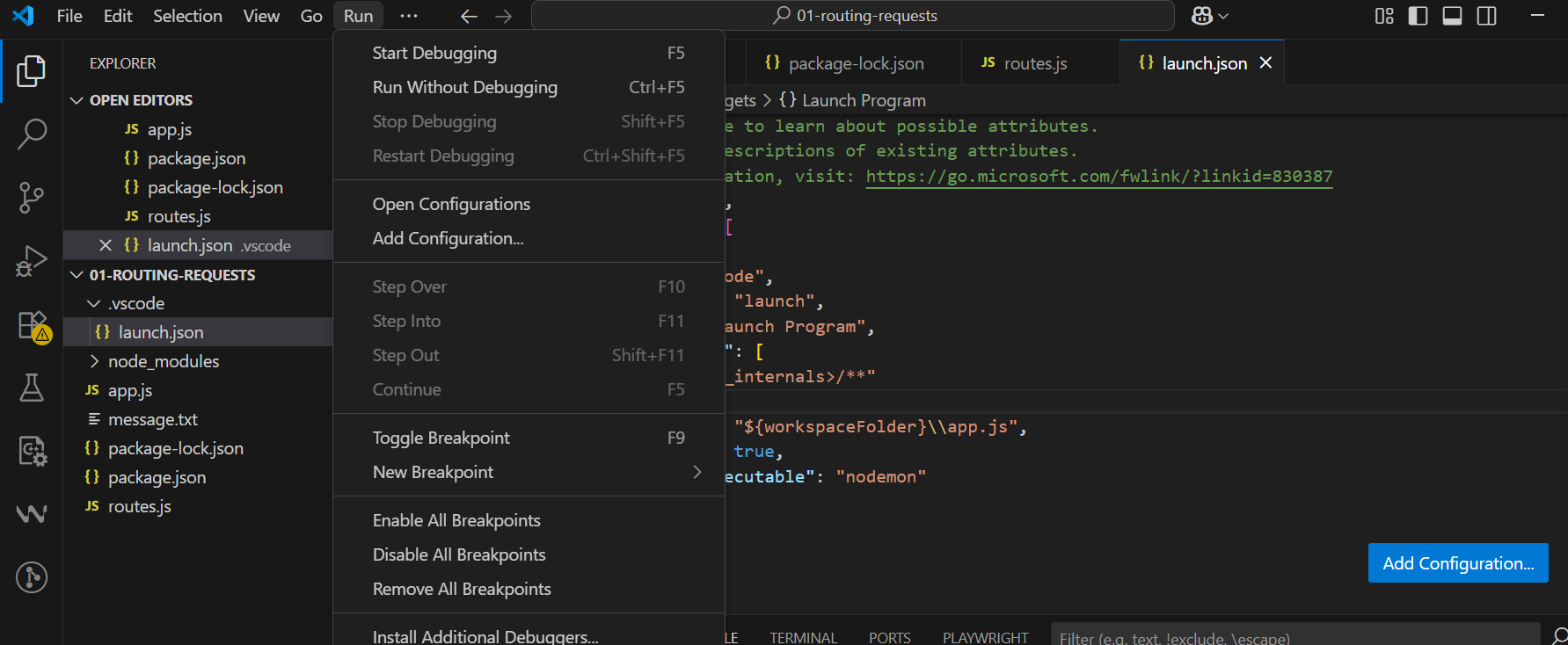
The good thing about local dependencies is that you can share projects **without the node\_modules** folder (where they are stored) and you can run npm install in a project to then re-create that node\_modules folder. This allows you to share only your source code, hence reducing the size of the shared project vastly.

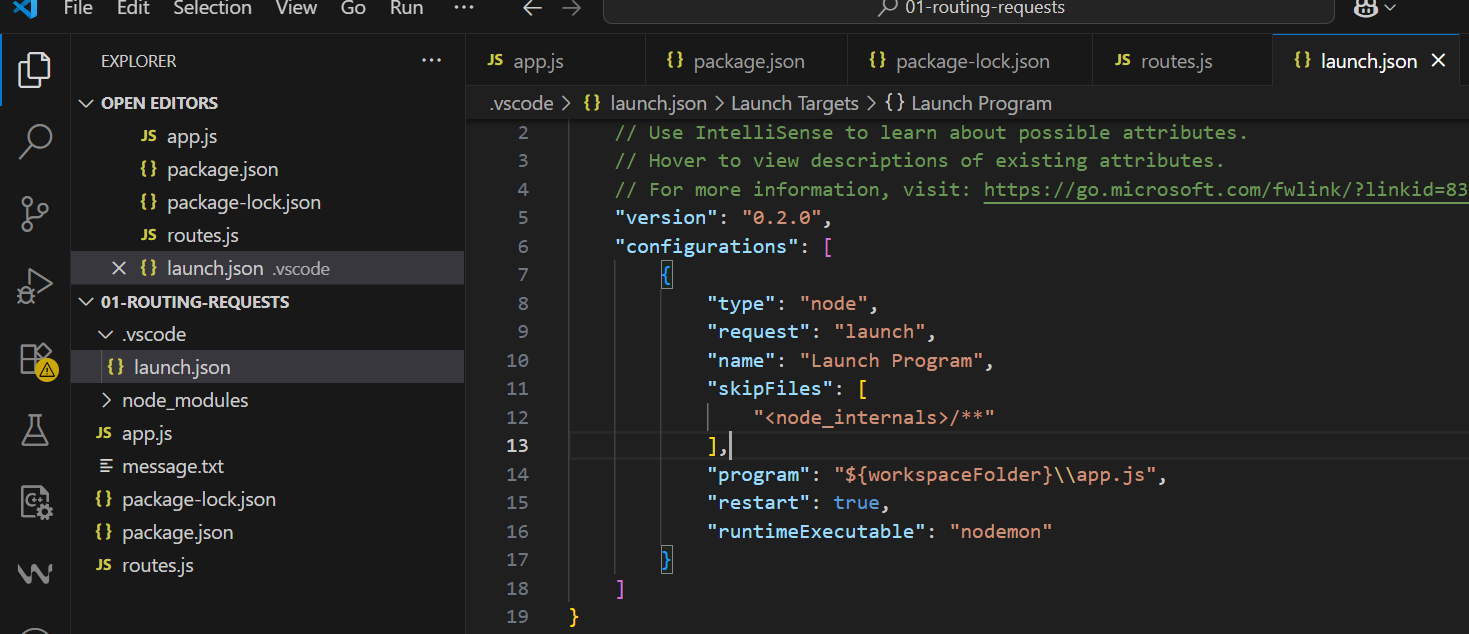
The attached course code snippets also are shared in that way, hence you need to run npm install in the extracted packages to be able to run my code!

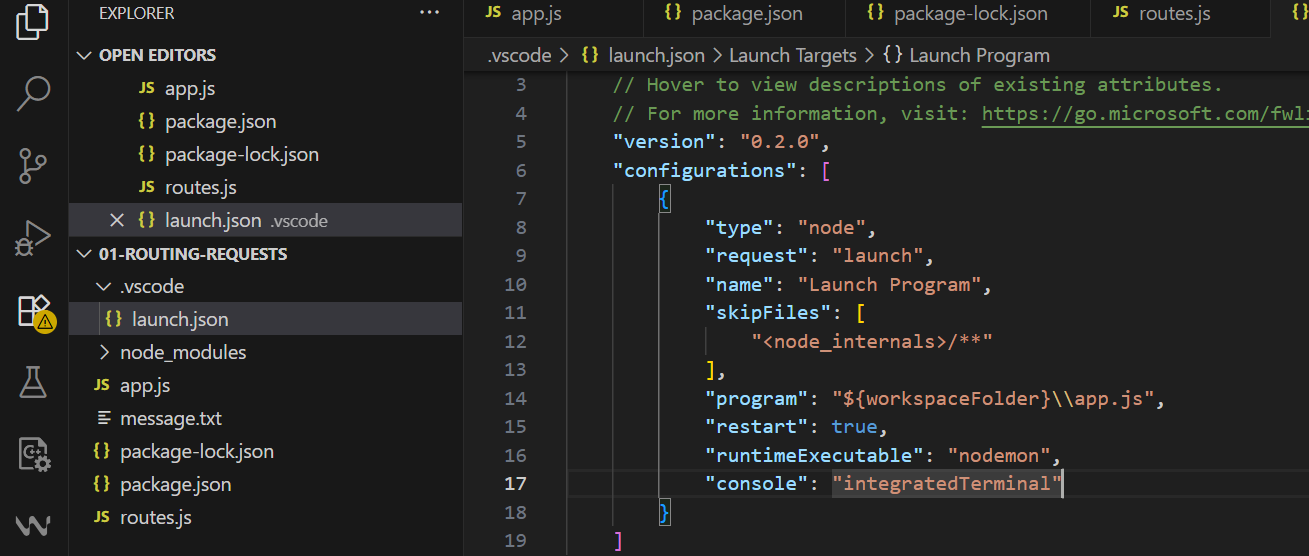
I showed that nodemon app.js would **not work** in the terminal or command line because we don't use local dependencies there but global packages.

You could install nodemon globally if you wanted (this is NOT required though - because we can just run it locally): npm install -g nodemon would do the trick. Specifically the -g flag ensures that the package gets added as a global package which you now can use anywhere on your machine, directly from inside the terminal or command prompt.

**How to use Debuggers:**

****

****

****

**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.

The great thing about express is that it's highly flexible and actually doesn't add too much functionalities

out of the box but it gives you a certain way of building your application or of working with the

incoming requests that make it highly extensible and therefore, there are dozens or hundreds and thousands

of third party packages built for express specifically that you can then easily add to your node express

application without having to configure a lot and this is

probably the real strength of express and of course it also does add some nice features out of the box.

So why don't we just install it and take a closer look ourselves.

**Installing Express.js**

npm install --save express <- as it needed in production too so just –save

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.

**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.

**App.js**

const http = require('http');

const express = require('express')

//express returns a method which is also like a

//request handler

const app = express();

//middleware1

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

    console.log('This is middleware');

    next();

});

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

    console.log('This is middleware2');

    //next();

});

const server = http.createServer(app);

server.listen(3000);

**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.

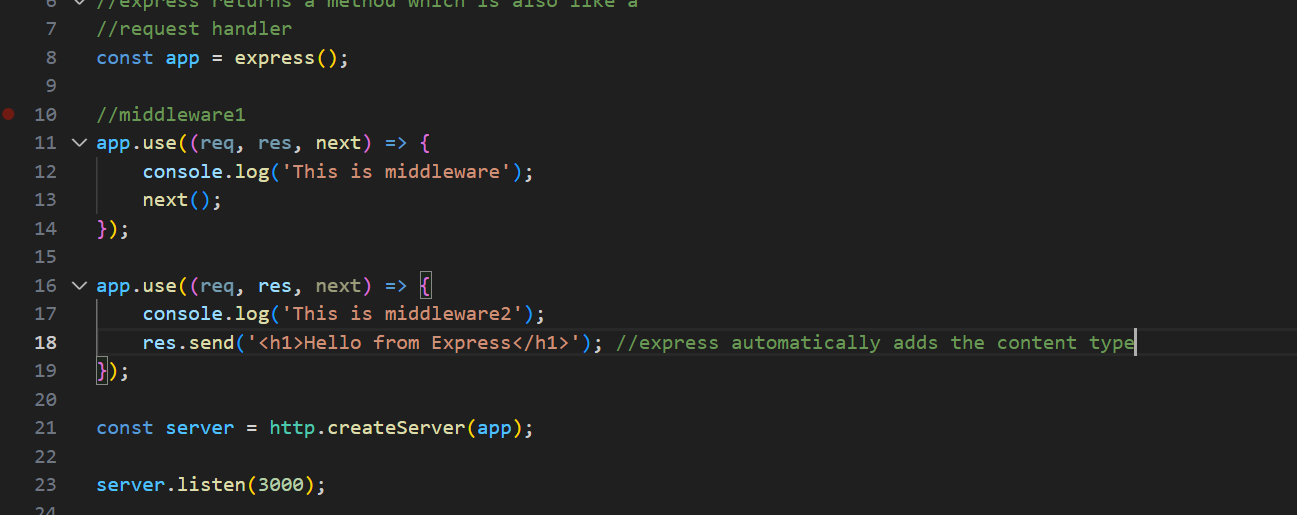
Now this is something we'll also do in this module but for now, make sure you understand this basic middleware

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.

****

**Express js takes care of server creation and listening – app.listen(port)**

const http = require('http');

const express = require('express')

//express returns a method which is also like a

//request handler

const app = express();

//middleware1

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

    console.log('This is middleware');

    next();

});

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

    console.log('This is middleware2');

    res.send('<h1>Hello from Express</h1>'); //express automatically adds the content type

});

//the below two actions are handled by express - app.listen(3000)

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

// server.listen(3000);

app.listen(3000);

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.

Top of Form



**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.

const express = require('express')

const app = express();

//middleware1

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

    //console.log('This is middleware');

    res.send('<h1>This is the product page</h1>');

   // next(); not required when using express send

});

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

    //console.log('This is middleware');

    res.send('<h1>This is the home page</h1>');

   // next();

})

app.listen(3000);

**Parsing Incoming Requests**

Install the package -> npm install --save body-parser

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?

const express = require('express')

const bodyParser = require('body-parser')

const app = express();

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

//middleware to parse FormData

app.use(bodyParser.urlencoded());

//middleware for product-add

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

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

})

//middleware1

app.use('/product', (req, res, next) => { //This works for both GET and POST

//app.post('/product', (req, res, next) => { //THis for post alone, for get - app.get('/product', (req, res, next) => {

    console.log('request body:', req.body);

    res.send('<h1>This is the product page</h1>');

})

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

    //console.log('This is middleware');

    res.send('<h1>This is the home page</h1>');

   // next();

})

app.listen(3000);

**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.

const express = require('express')

const bodyParser = require('body-parser')

const app = express();

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

//middleware to parse FormData

app.use(bodyParser.urlencoded());

//middleware for product-add

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

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

})

//middleware1

// app.use('/product', (req, res, next) => { //This works for both GET and POST

app.post('/product', (req, res, next) => { //THis for post alone, for get - app.get('/product', (req, res, next) => {

    console.log('request body:', req.body);

    res.send('<h1>This is the product page</h1>');

})

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

    //console.log('This is middleware');

    res.send('<h1>This is the home page</h1>');

   // next();

})

app.listen(3000);

**Bottom of Form**

**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.

**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.

const express = require('express')

const bodyParser = require('body-parser')

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

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

const app = express();

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

//middleware to parse FormData

app.use(bodyParser.urlencoded());

app.use(adminRoutes);

app.use(shopRoutes);

//middleware to bring error page - 404

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

    res.status(404).send('<h1>Page not found</h1>');

})

app.listen(3000);

**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.

**Admin.js**

const express = require('express');

const router = express.Router();

//middleware for product-add

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

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

})

//middleware1

//app.use('/product', (req, res, next) => { //This works for both GET and POST

router.post('/product-add', (req, res, next) => { //THis for post alone, for get - app.get('/product', (req, res, next) => {

    console.log('request body:', req.body);

    res.send('<h1>This is the product page</h1>');

})

module.exports = router;

**App.js**

const express = require('express')

const bodyParser = require('body-parser')

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

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

const app = express();

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

//middleware to parse FormData

app.use(bodyParser.urlencoded());

app.use('/admin',adminRoutes); // setting common base path

app.use(shopRoutes);

//middleware to bring error page - 404

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

    res.status(404).send('<h1>Page not found</h1>');

})

app.listen(3000);

**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.

**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.

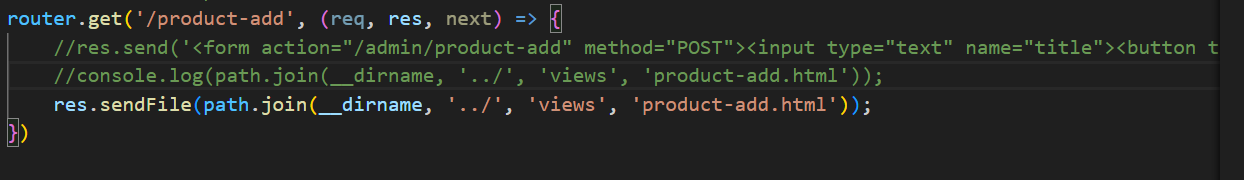
Now here's one bonus task for you which I want you to solve and which we'll solve together in the next

lecture, add a new html file which is your page not found page which you then serve

if we ever reach this middleware function.

Top of Form



Bottom of Form

**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.

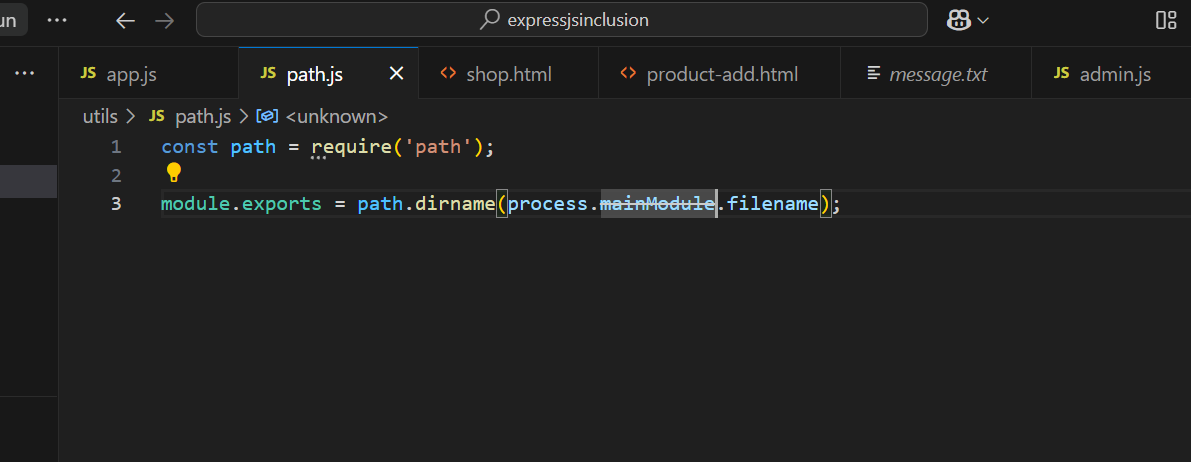
Again

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.

****

const express = require('express')

const bodyParser = require('body-parser')

const path = require('path');

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

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

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

const app = express();

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

//middleware to parse FormData

app.use(bodyParser.urlencoded());

app.use('/admin',adminRoutes); // setting common base path

app.use(shopRoutes);

//middleware to bring error page - 404

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

    //res.status(404).send('<h1>Page not found</h1>');

    console.log('Page:', path.join(\_\_dirname, 'views', 'page-not-found.html'), 'pat:', \_\_dirname);

    //res.status(404).sendFile(path.join(\_\_dirname, 'views', 'page-not-found.html'));

    res.status(404).sendFile(path.join(rootDir, 'views', 'page-not-found.html'));

})

app.listen(3000);

**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.

****

****

**Sharing Data Across Requests & 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.

**Admin.js**

const path = require('path');

const express = require('express');

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

const router = express.Router();

const products = [];

// /admin/add-product => GET

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

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

});

// /admin/add-product => POST

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

  products.push({ title: req.body.title });

  res.redirect('/');

});

exports.routes = router;

exports.products = products;

**Shop.js**

const path = require('path');

const express = require('express');

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

const adminData = require('./admin');

const router = express.Router();

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

  console.log('shop.js', adminData.products);

  res.sendFile(path.join(rootDir, 'views', 'shop.html'));

});

module.exports = router;