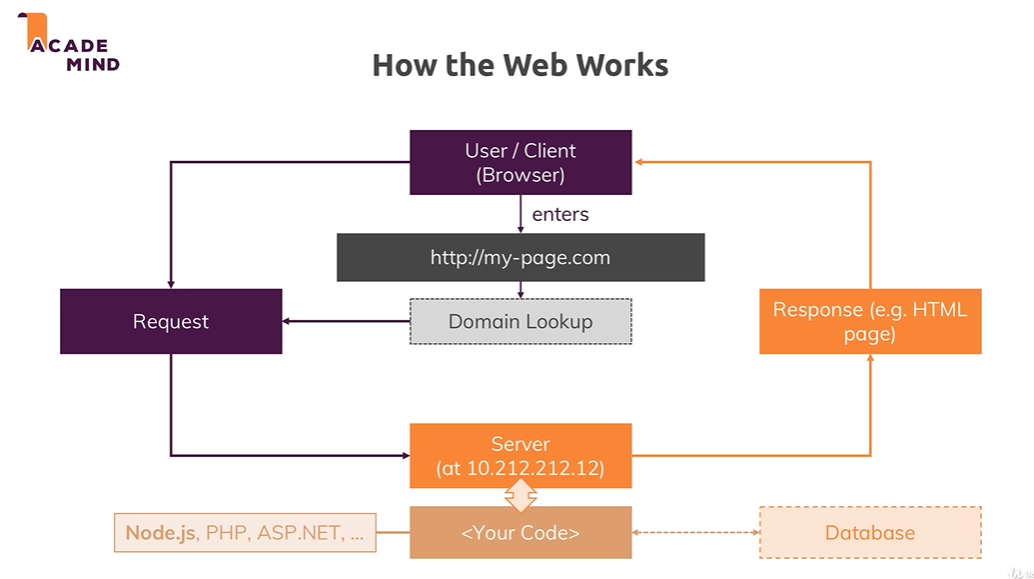
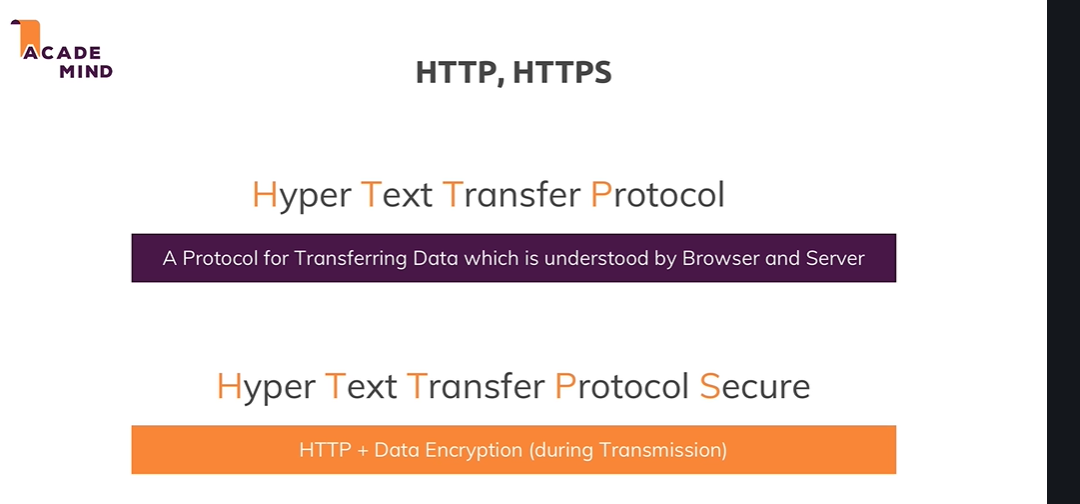
**Lecture 25**

**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.

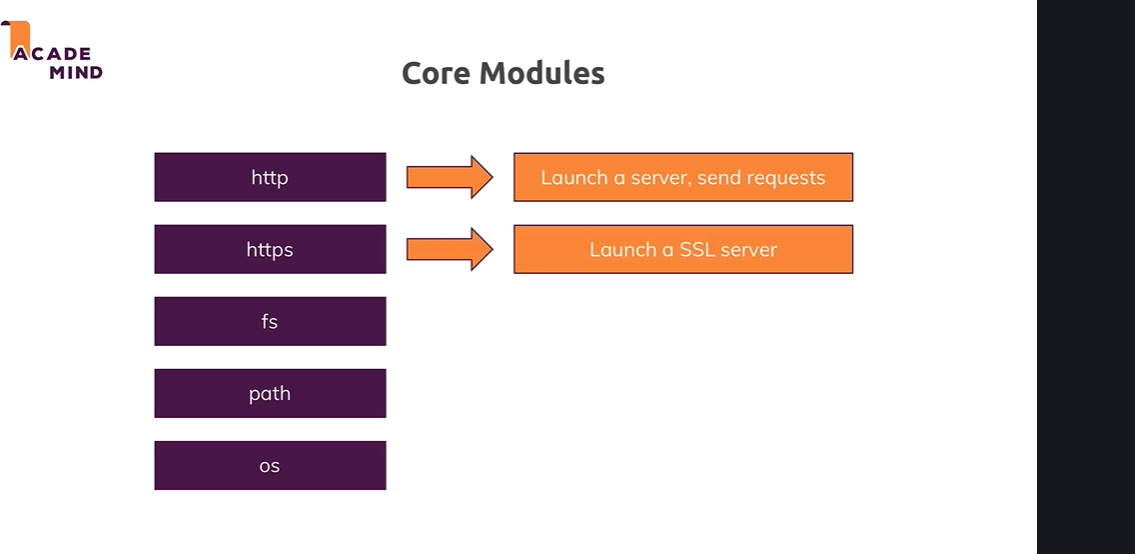




**Lecture 26**

**Creating a Node Server**

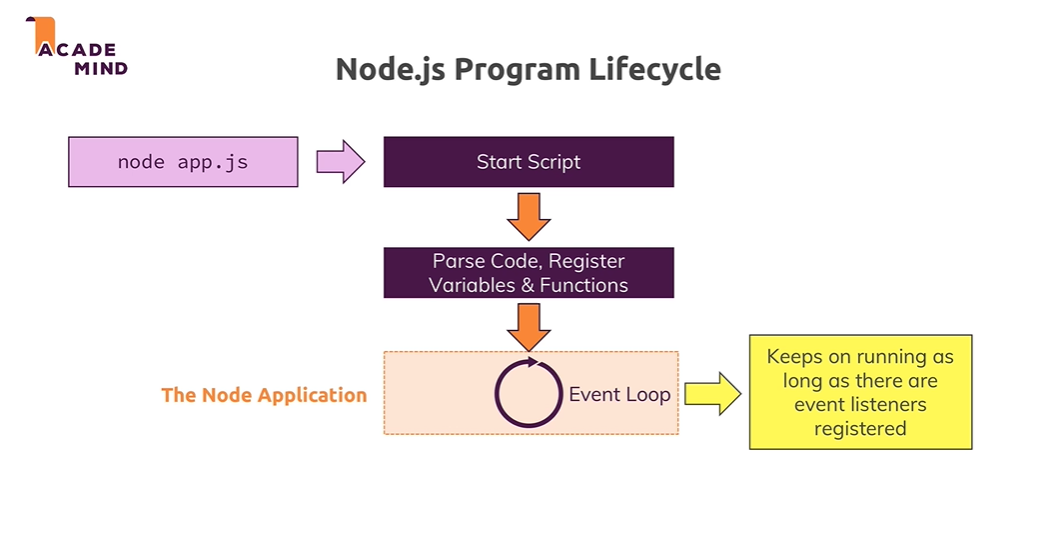
I'm back in Visual Studio Code, the IDE I'll use throughout this course and this is again a totally empty folder, I only got my gitignore file in there because I will use version management here, git you don't need to use that at all, if it doesn't tell you anything you can just ignore that, you don't 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.



**Lecture 27**

**Node Lifecycle and 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.



Controlling the Node.js Process

Want to quit your running Node.js server?

You can always do that by pressing CTRL + C in the terminal/ command prompt window where you started your server (i.e. where you ran node app.js).

**Lecture 29**

**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.

**Lecture 30**

**Sending Responses**

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.

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>

**Lecture 32**

**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.

**Lecture 33**

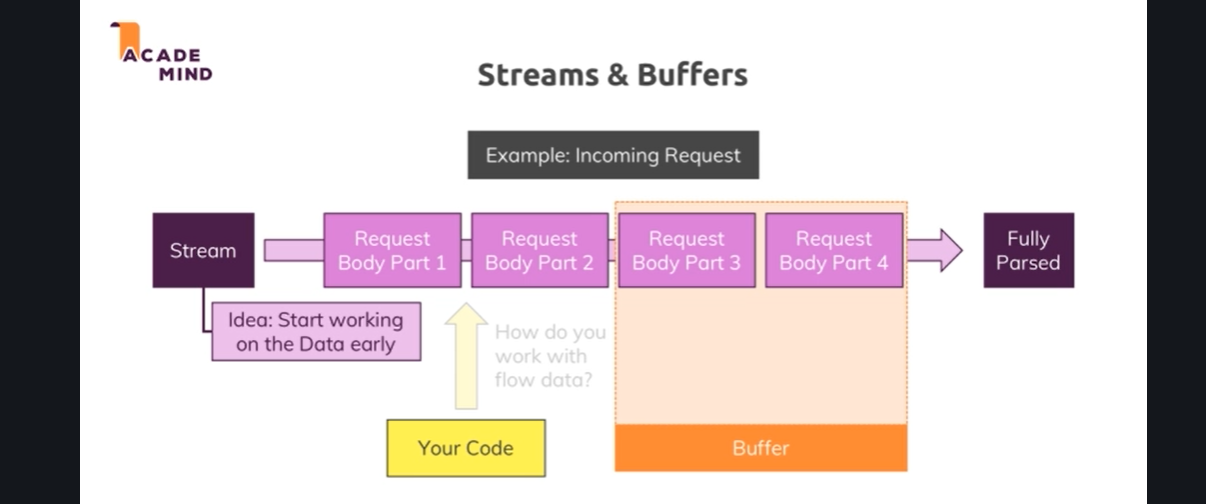
**Redirecting Requests**

So we made sure that we listen to requests to just slash nothing and that we return some html code with our input field on it. Now when we click that send button, we send a post request to /message but we're not doing anything with that, time to change that. Let's add another if statement here and let's check if the url is equal to message and let's add another condition and that condition is that I want to be sure that we're not handling a get request but a post request here, so let's also parse the method from request method and make sure that method is equal to post. Now we'll only enter this if statement if we have a post request to /message which happens to be exactly what we create with this form. In this case, I want to do two things, I want to redirect the user back to slash nothing, so not leave him on /message and I want to create a new file and store the message the user entered in it. Now this involves a couple of things, let's start with redirecting and creating that file. We actually already worked with a file in the first core section and do you remember how that worked? Feel free to go along on your own if you know that. We need another package, another core module and that was the file system core module. So let's import it by storing the functionality in a constant, you can name it however you want, I'll name it fs because the package is named fs but you can also name this differently. This, not the package, just the constant. So fs allows us to work with the file system and here I now want to write a new file, so here let's execute write file and write file takes a path to the file and I'll just use the file name to create it in the same folder as app.js and I'll simply name it message.text and in there, I obviously want to store what the user sent. Now this is a little bit more work, so for now let's just put some dummy text in there and let's redirect the user. Now important, we should actually use write file sync for now and I'll explain what the difference to write file which also exists is in a while, so let's use write file sync for now. Now write head which basically allows us to write some meta information in one go and then we set a status code of 302 which stands for redirection and then we pass a javascript object with some headers we want to set and you could also do this in two steps by the way, you can also set the status code to 302 and then simply have, whoops set header And there we set the location, this is also a default header accepted by the browser and we set that location to just slash and I will automatically use the host we're already running on and then we have to call res end. And important, as before return this so that we don't execute these lines otherwise we will get an error. Now let's restart our file here and make sure to go back to slash nothing and reload that page with the input field and send any value and you should simply reload that in the end because you get redirected here but you can see that redirect here in the network tab of the developer tools, here 302 indicates we send a request to message and we were redirected to localhost. So this is working and we also got the message text with dummy inside of it. Now let's also get rid of that though and let's work on actually parsing the data the user sent us and writing that data into that file.

**Lecture 34**

**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.



**Lecture 35**

**Understanding event driven code execution**

We already achieved a lot in this section. And I noted 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'll never forget what's happening behind the scenes. Now one crucial thing which I note 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 year will actually execute after this code so it will even execute after we already sent the response. This has two important implications for one sending the response does not mean that our event listeners here our debt. 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 the response this is a 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 we request on or code like HTP create server. These are some examples where no chase uses a pattern where you pass a function to a function and node will execute DS passed in functions at a later point of time which is called asynchronously. Now it's not always the case that a passed in function is necessarily executed at a later point of time but no J has used this pattern heavily and for the course of course lets you know when this is to case and when node executes something asynchronously in such cases. No chance won't immediately run that function. Instead what it does when it first encounters this line is it will simply add a new eventless 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 change is done. Parsing the request so does this something no chase 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 chase having some internal registry of events and listeners to these events and a function like this is such a listener and when served something happens so when no chance is done parsing your request it will go through the registry and see. I'm done with the request so I should now send the end events. Let's see which listeners I have for that and will then find this function and any other functions you might have registered for that and will now call them but it will not posti average code execution and that is so important to understand. So for example here now since I moved the return response and into this function the flow is like this. It will now reached as if statement and if these conditions are met it will go inside of it. It will then register DS to handlers and not immediately execute these two functions. Instead the functions are just registered internally in it's event emitter registry and then it will jump straight away to the next line and therefore right now if I would restart my server and save to 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 redirect it as you can see. There is no 300 status code instead of just Lote's this page because it executes these lines because as I'm now mentioned multiple times it will not execute just right away. And this return statement will therefore not quit this overarching function here. Instead it just registers this callback and immediately moves onto 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 hetero's error here because it already moved the long 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 it is just a callback to be called sometime in the future. And this set up 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 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 application for design is that we reach a too early. So to a white this we should actually return here. We simply return request 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.

**Lecture 36**

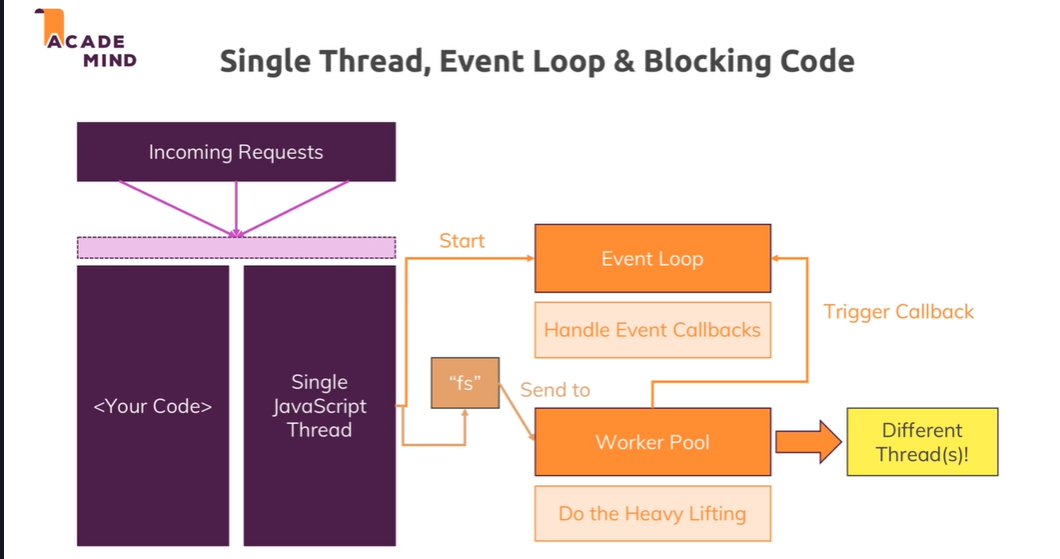
**Blocking and Nonblocking code**

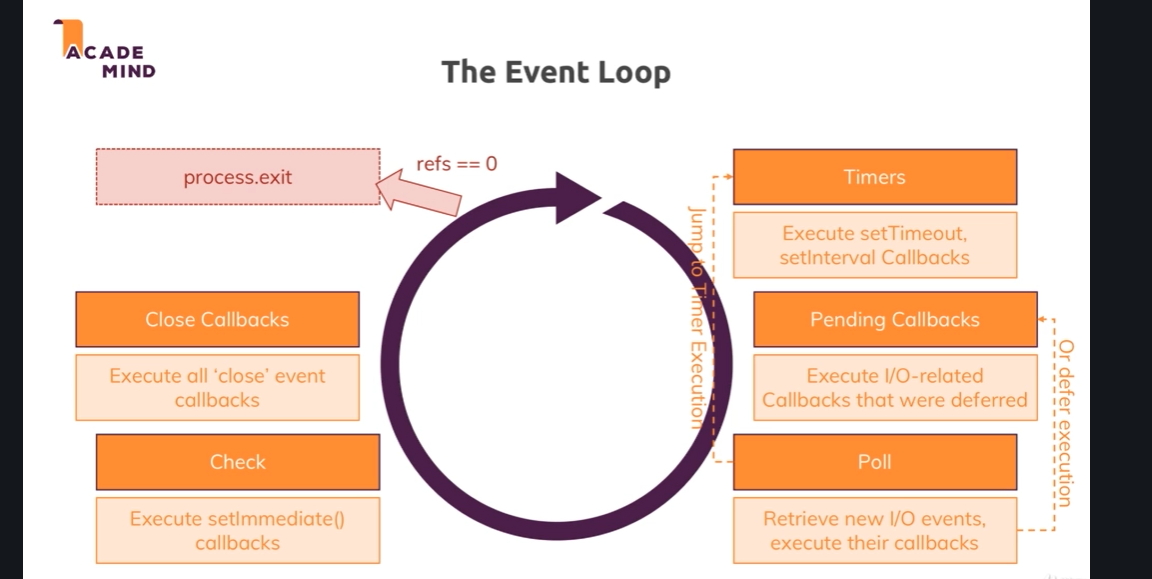
So what is wrong with this write file sync line here? Well it's the sync keyword. I earlier mentioned that there also is a write file method but we use write file sync, the sync here stands for synchronous and this is a special method which will actually block code execution until this file is created. 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.

**Lecture 37**

**Nodejs – 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.





**Lecture 38**

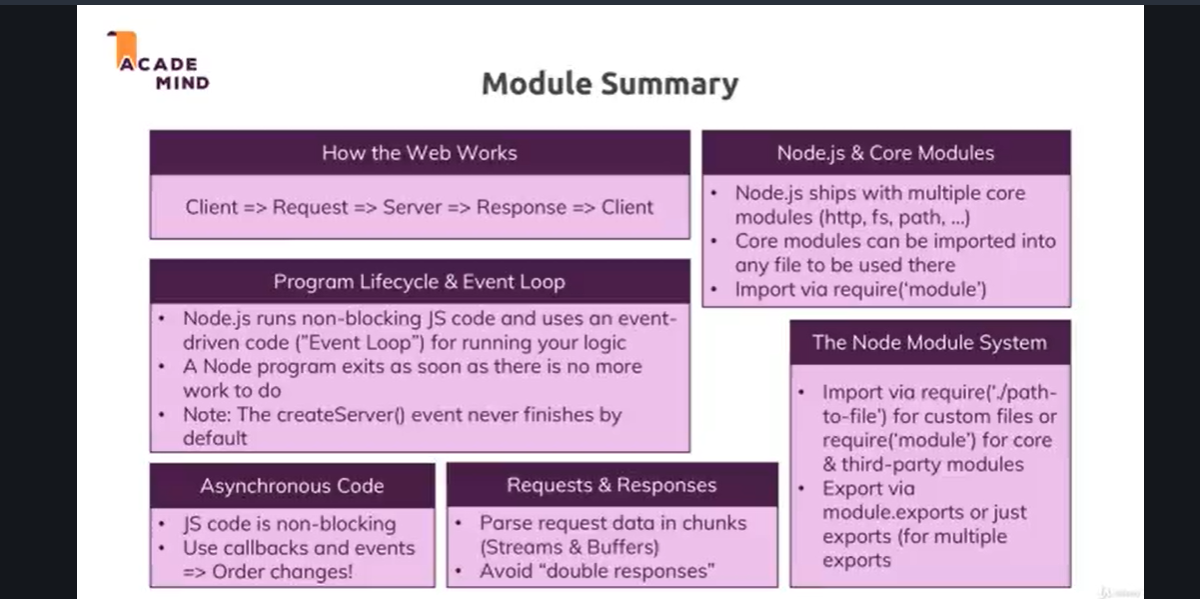
**Using the node modules system**

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.

**Lecture 39**

**Wrapup**

So let's sum this module up now. First of all we had a brief refresher on how the web works and in general, it looks like this, client so the browser sends a request to the server, the server does some magic, reaches out to a database, works with files, sends back a response, can be html, can be something different and sends it back to the client, the browser which then can display that and that is the entire flow we also saw in this module. Now nodejs is the part that runs on the server and regarding its lifecycle, there is one important concept and that is that so-called event loop. Nodejs code runs in a non-blocking way which means we only register a bunch of callbacks and events and nodejs will eventually trigger that code once a certain task is done, so that the javascript threat is always free to handle new events, new incoming requests and stuff like that. And node program can exit if there is no more work to do but on a server, this well should at least never happen because create server registers an event listener for an event which is never done, if a new event is triggered, so if a new request is received, this does not mean that node unregisters the event listener instead we keep on listening for more requests and that is of course how a server should behave. So it's this cycle that's important to understand and that we have this loop which always keeps on going, keeps on waiting for new events and which does something when some event happens and then basically dispatches some actions to the operating system you could say for example and then again frees up the threat. Now this also involves a lot of asynchronous code which we saw with all the callbacks. The javascript code should be non-blocking, so we have this callback and event driven approach where we are able to register some code to be executed in the future instead of running right away and blocking the main threat because this has to be avoided under all circumstances and nodejs is built around that concept of avoiding this issue. We also saw how to work with requests and responses the nodejs way. We saw that we have to parse the requests data which arrives in chunks and that we can use this concept of streams and buffers which I explained and that we should avoid sending double responses so that after res end, you must not send another response and this can happen easily if you forget about that asynchronous nature and that a line of code you write might not execute immediately. So depending on where you write it, if it's in an event listener, it will not execute immediately. That is what I mean and that is what is important to keep in mind here. We also learn that nodejs is all about using it's built in functionalities and whilst there are some global variables or objects or functions we can use, this also means that we should use its core modules. Core modules are things like the http, the fs or the path module, there are more and you can learn all about them in the official nodejs docs of course, we'll also use quite a lot of them throughout this course and these core modules give us a couple of different functionalities that allow us to basically do whatever we ever could want to do on a server like create a new server with the help of the http module. They're imported via the require syntax and we can only use them in the file into which we import them and if we want to use them in two different files, we have to import them in both files separately. Now that leads us to the node module system and this basically works with the help of this require keyword which pulls some functionality from one of our files if we start with slash or ./ or from a core or third party module, we haven't used any third party modules thus far but we'll also do that soon and it basically pulls in whatever we export there and stores it in a new variable or constant as we did it in this module. And export is an important keyword here, we do export with the help of module.exports or for multiple exports with the export shortcut I showed you in the last lectures. So this is what we learn in this module and I know that this was a lot of theory or nitty gritty stuff about nodejs. Doesn't look too easy and beautiful but it'll get way more beautiful from now on, it is super important to never forget what nodejs is and does it for you though because many courses right away start with expressjs which we'll also use starting soon and therefore you never really learn what's happening under the hood which is sad because this is important and makes you a better node developer. But with that, let's move on.



Attached, you find the source code for this section.

Useful resources:

* Official Node.js Docs: <https://nodejs.org/en/docs/guides/>
* Full Node.js Reference (for all core modules): <https://nodejs.org/dist/latest/docs/api/>
* More about the Node.js Event Loop: <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>
* Blocking and Non-Blocking Code: <https://nodejs.org/en/docs/guides/dont-block-the-event-loop/>