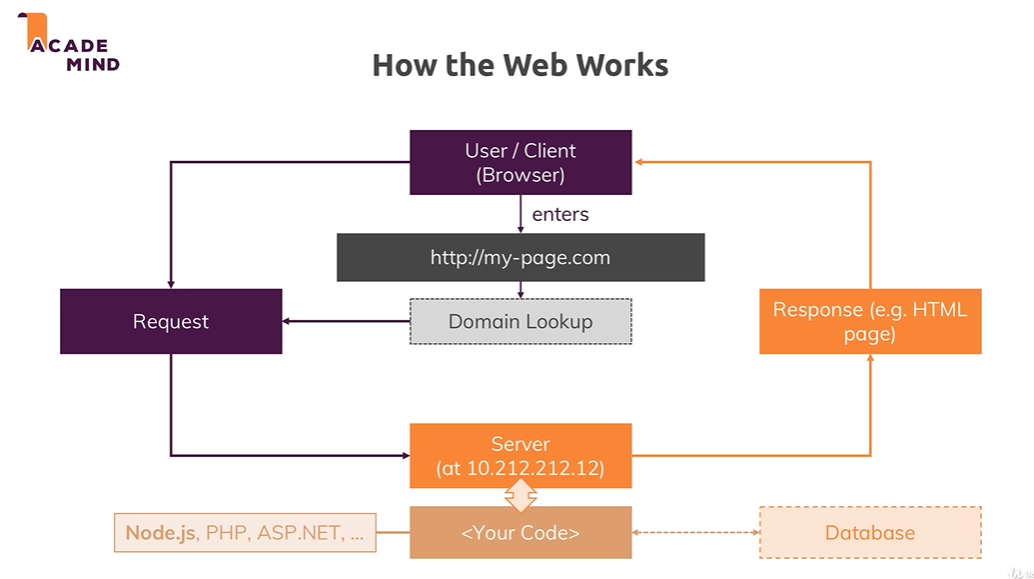
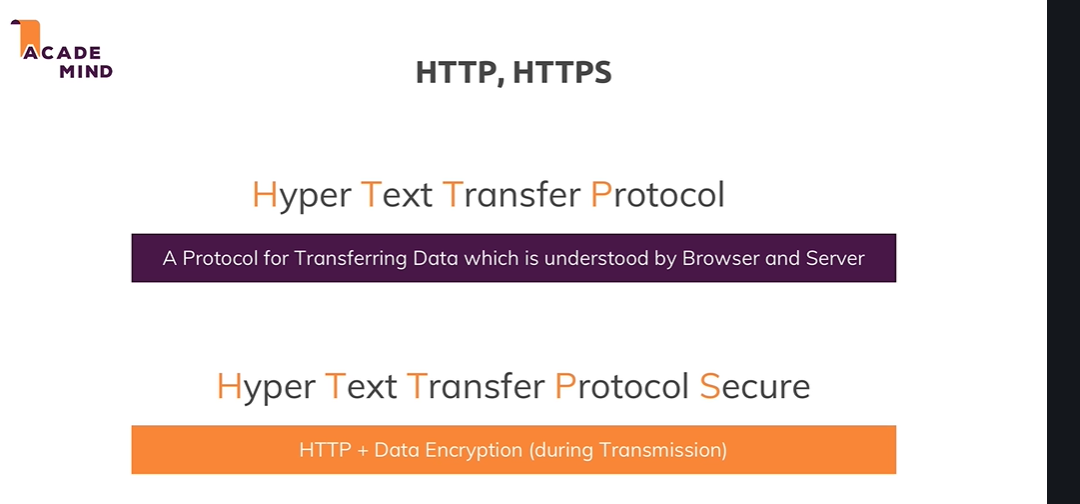
**Lecture 25**

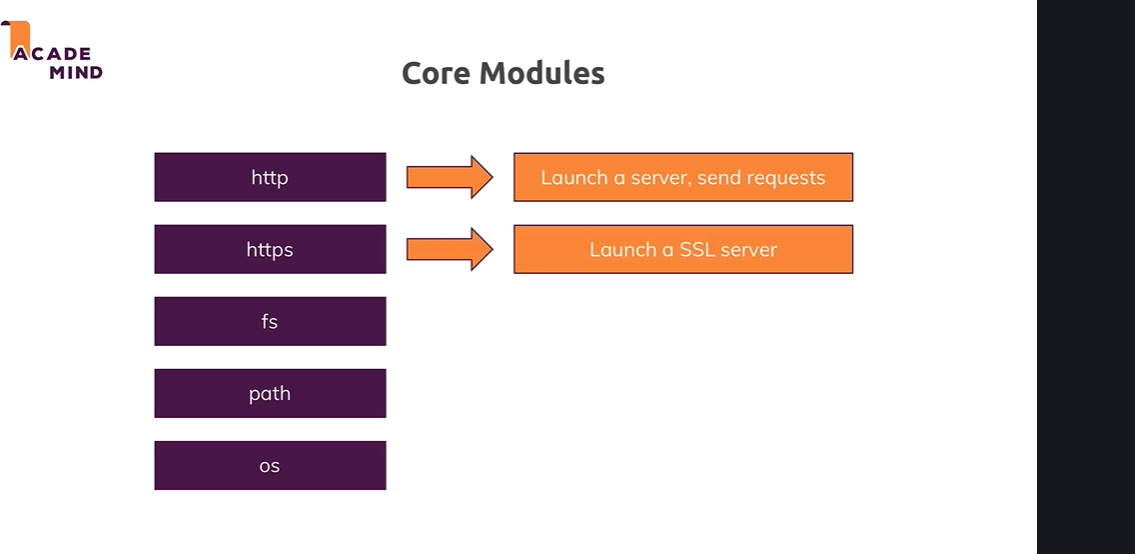
**How the Web Works**





**Lecture 26**

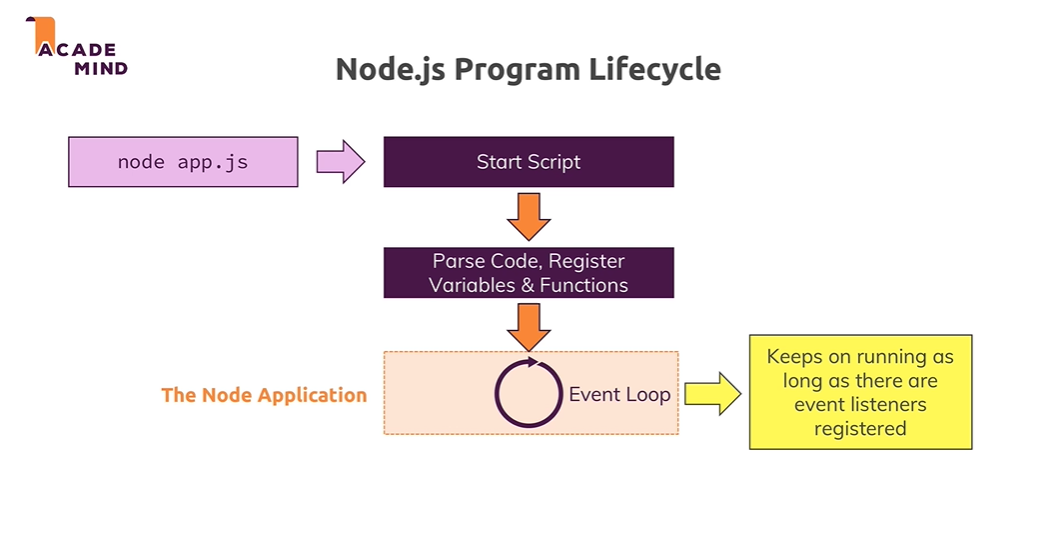
**Creating a Node Server**

* , 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
* 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
* Https would be helpful when we want to launch an ssl encoded server, so where all that data which is transferred is encrypted.
* http module and to use it, we need to import it.
* 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
* 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.
* When the path doesnot start with a ./ or / , it will look for a global module.
* http.createServer takes a request listener as argument.
* A request listener simply is a function that will execute for every incoming request
* Listen method on server l, actually starts a process where nodejs will not immediately exit our script but where nodejs will instead keep this running to listen.
* 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
* 

**Lecture 27**

**Node Lifecycle and Event Loop**

* 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.
* 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
* 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.
* 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



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**

* 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
* Refer code

**Lecture 30**

**Sending Responses**

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

**Lecture 32**

**Routing Requests**

* Refer Code

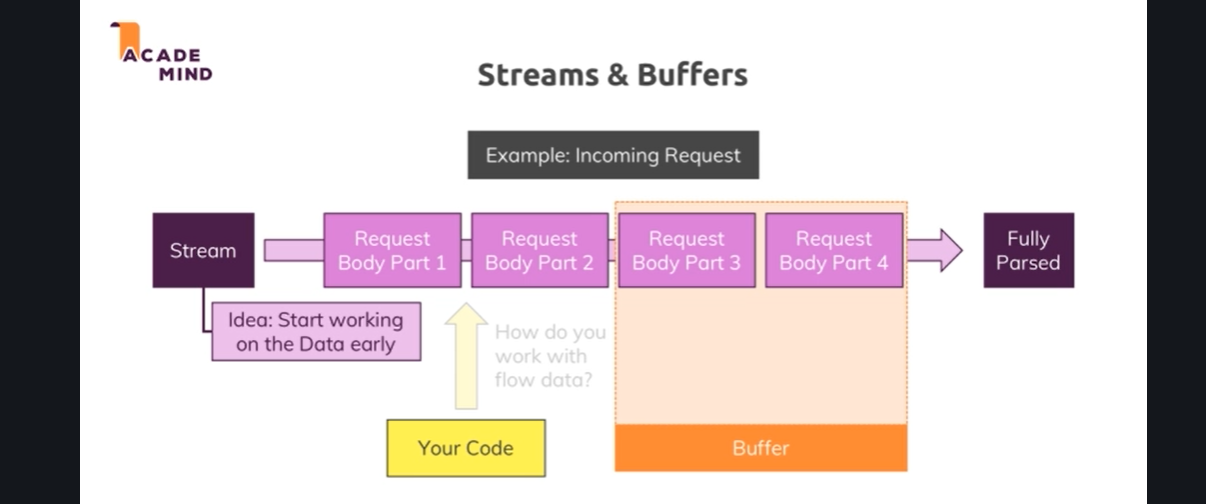
**Lecture 33**

**Redirecting Requests**

* Refer Code – 01-routing-requests

**Lecture 34**

**Parsing Request Bodies**

* Refer Code -- 02-parsing-request-bodies
* 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
* 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 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
* 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.
* 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.
* 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
* 

**Lecture 35**

**Understanding event driven code execution**

* Refer Code -- 02-parsing-request-bodies
* We cannot send another response when we have already sent a response using res.end().

**Lecture 36**

**Blocking and Nonblocking code**

* Refer Code -- 03-blocking-and-non-blocking-code.
* When 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
* . 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.
* 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

**Lecture 37**

**Nodejs – Looking behind the scenes**

* Refer Notes

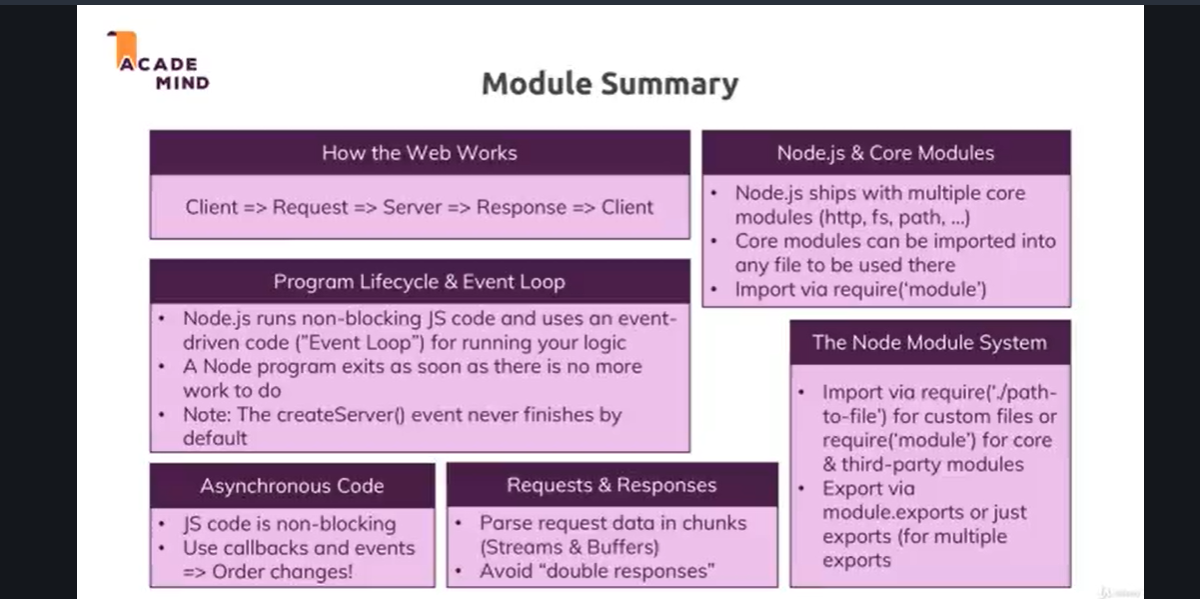
**Lecture 38**

**Using the node modules system**

* Refer Code 04-using-the-node-modules-system
* Refer Code 05-fixed-missing-head-tag

**Lecture 39**

**Wrapup**



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/>