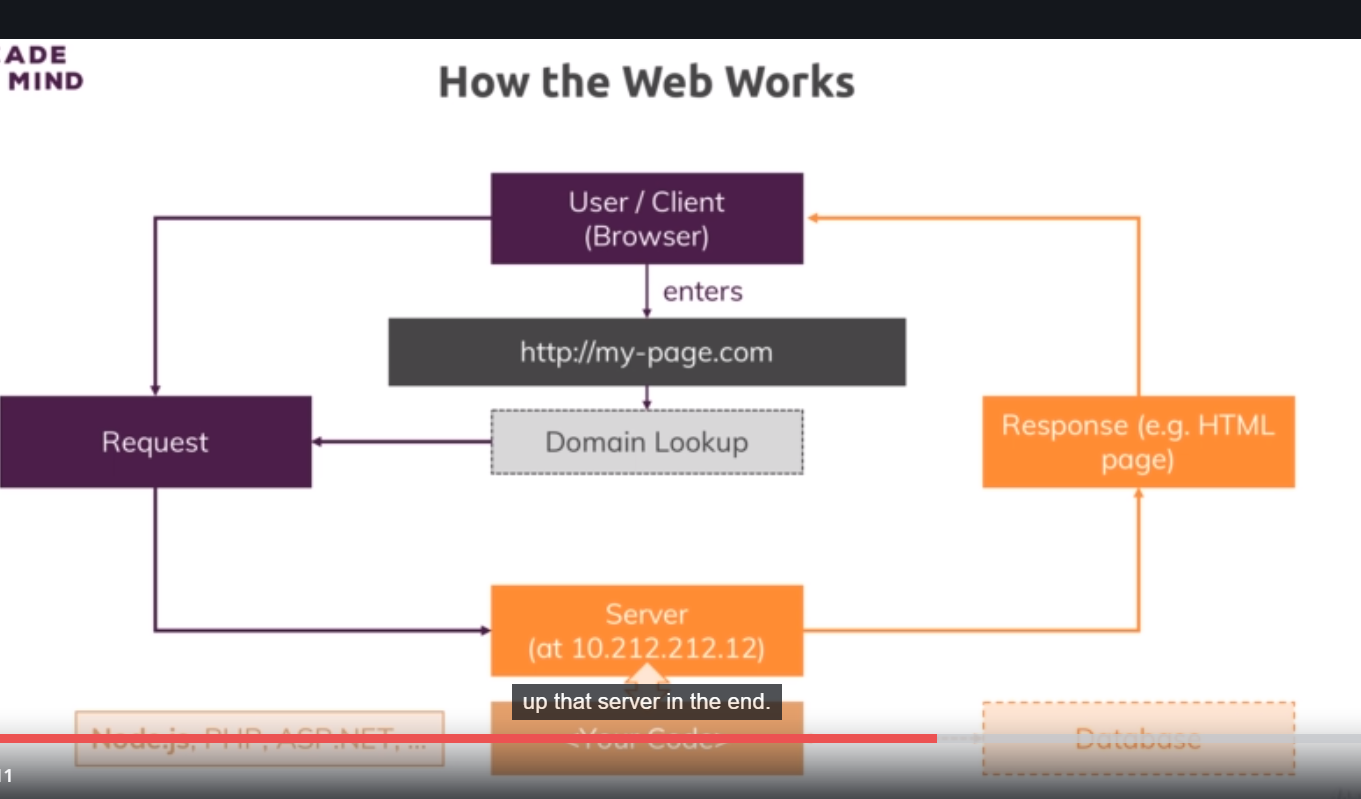
24)how web works

We know how this thing works-



Now this request and response transmission is done thorugh some protocol. So basically a standardized way of communicating you could say because obviously to correctly handle the request and send back a response browser can work with, we have to follow some rules and these rules are defined by protocol we use- http or https.

http – hyper text transfer protocol. Here we simply define, how a valid request should look like and how we data should be transferred from browser to server and other way around.

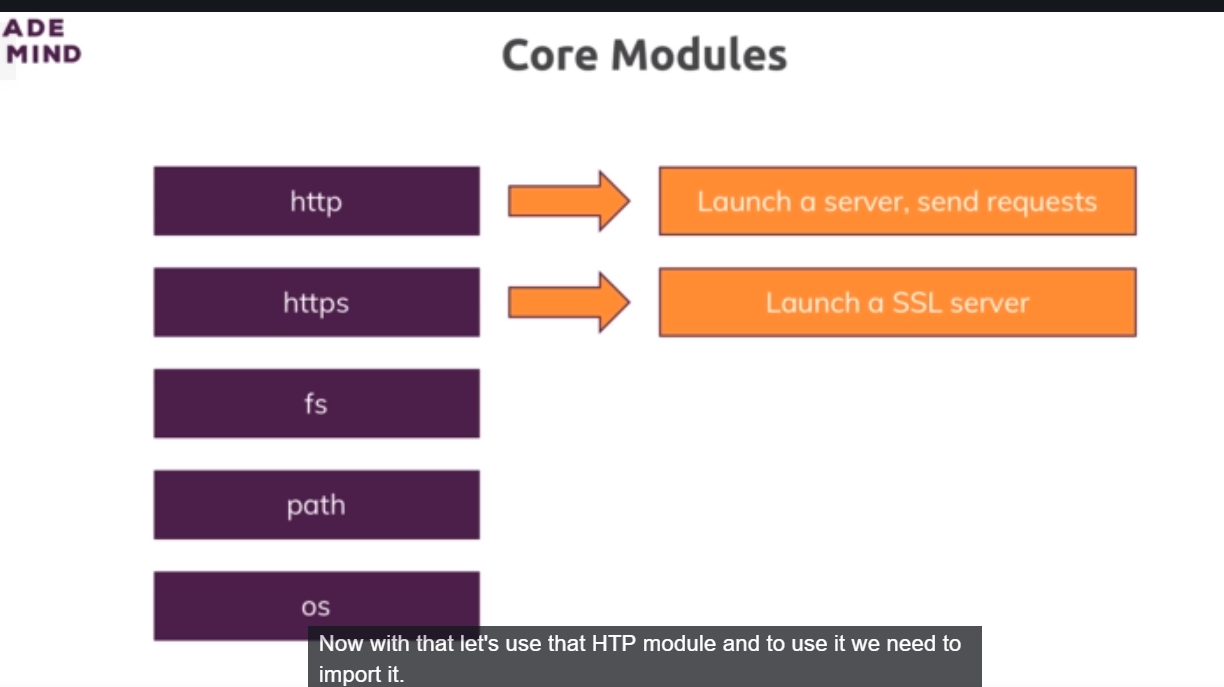
https is simply is same as http with ssl encryption turned on, where all the data that is transferred is actually encrypted so that if anyone is spoofing your connection they cannot read your data.

Now at the end of the course I will tell you how to enable https ,for majority of our course, we will be working with http since we will only be developing , that code will work locally only. But once you put into production, I will also show you how to turn on ssl encryption.

25)Creating a nodejs server

We got empty folder, here we will create a file , we can name it anything you want, but often we name it as app.js or server.js because it is root file that makes up your nodejs application.

In this file we will create server. How can we do that? We will need to import some functionality. So the way js works for both browser and nodejs, there is a handful of functions and objects we can use globally without importing anything into that file, but generally most functionality are’nt avalible by default to not pollute global namespace with all these reserve keywords and names basically and also to make it very obvious in each file on which functionalities this file depends. Now there are couple of module that nodejs ships with and as you will learn from the course, you can also install third party modules that do not ship with nodejs. Lets stick to core modules for now. Here are handful of them(not all just some)-



There is fs which we already used. There is also paths which helps us in constructing paths, so paths to file on filesystem that work on any operating system because windows , mac and linux use different path formats. There is os ppackage that helps us with operating system relevant information and so on and there are 2 top most packages – http and https. As you might be able to guess,these 2 sounds very helpful when it comes to creating server and work with http request and responses and http helps us with launching http request and also with other tasks like sending request, because a node app could alo send a request to another server. You can have multiple servers communicating with each other. for example you could send that request to google api to send some coordinates and get back a address.

https helps with when we want to launch ssl encoded server, so where all the data that is sent is encrypted. And as I said earlier I will cover it at the end of the course.

with that lets use http module to create a server. To use it we need to import it. We basically need to make sure we can use features from http module which nodejs ships with but which is still not avalaible globally by default. We need to make sure that we can use this features in this file. and for this we import that functionality. We do this by creating a new const, you can also use let or var but since we have some value here which we will never change, we can use const to make it really clear that we will never touch this again. We will us it but we will never overwrite it. You can give it anyname you want but typically you keep the name of module you are importing. So we give it name http.

Now we have special keyword, special function, that nodejs does expose globally. So you can use it by default in any file you run by nodejs and that is **require** keyword. Now this is simply the way you import the files in nodejs. require either takes a path to other file,(you can also import your own js files),you can also import a core module like here we import http core module. By the way path to your files always has to start with ‘ **./**’ or with **‘/’** if it is absolute path, ./ will be relative path. **Like require(‘./http’)**, it will lead to same folder and will look for http file. it will automatically add .js at the end you dnt need to add it on your own but you can add it. But it would look for a local file named http.

But if you use **require(‘http’)**, it would not look for local file. so even if you had a file named http.js, it would not import this file but instead it would look for global module named http and indeed such modules exists because nodejs ships with it. So once we get this module imported. We can now start using functionalities from that global module and if you type http., ide will start giving you suggestions. Here we use createServer method. This function actually takes so called reuestListener as an argument.(we can see it by hovering over it). Request listener , simply is a function that will be executed for every incoming request.

So lets define a such a function. you can name it whatever you want. This function has to receive 2 arguments, there are request and response(as you can see by hovering over createServer method). So in short nodejs automatically gives us some object that represents the incoming request and allows us to read data from that object and it gives us object response which we can use to return a response to whoever sent that request. You can rename these arguments whatever you want. But you just have to keep in mind that first one will contain data about request and the second one will help you send the response.

Then we pass this function to createServer. Here we tell it that this is our request listener function , please execute it for all incoming requests. So this function will be executed for every request which reaches our server and server is started by calling cretaeServer method or almost, one piece is missing, I will come back to it.

App.js-

const http = require('http');

function rqListener(req, res) {

};

http.createServer(rqListener);

now this is one way of doing it. now you dnt have to explicitly cerate such a function though. You can also use so-called anonymous function. like this-

http.createServer(function(req, res) {

});

It is function without a name, that is why it is called anonymous function and it still achieves the same. We passed this function to createServer. So node will execute this function, whenever new request reaches our server. This is event driven architecture. Nodejs uses it heavily. You work a lot with such setups or such code snippets where you tell node, X happens do Y. so in this case if request comes then execute this function. you can also pass arrow function to createServer.

App.js-

const http = require('http');

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

console.log(req);

});

So this is our createServer callback as it is called. It is called by nodejs whenever requests reaches our server. Here we simply console.log req object.

If we execute this request then nothing happens, this makes sense as we are not sending any request. But we dnt even know where server is. How do we reach this server, what address does it have. Actually one thing is missing. createServer method returns a server(you can see by hovering over it).

So we store the server in variable then we call listen method on it. 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 is why method is named like this) for incoming requests. Now listen takes couple of optional arguments. first one is port(if you dnt give port then by default it will be port 80). Then we define hostname, now by default this will be name of the machine , this is running on. So for our local machine this is local host by default.

App.js-

const http = require('http');

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

console.log(req);

});

server.listen(3000);

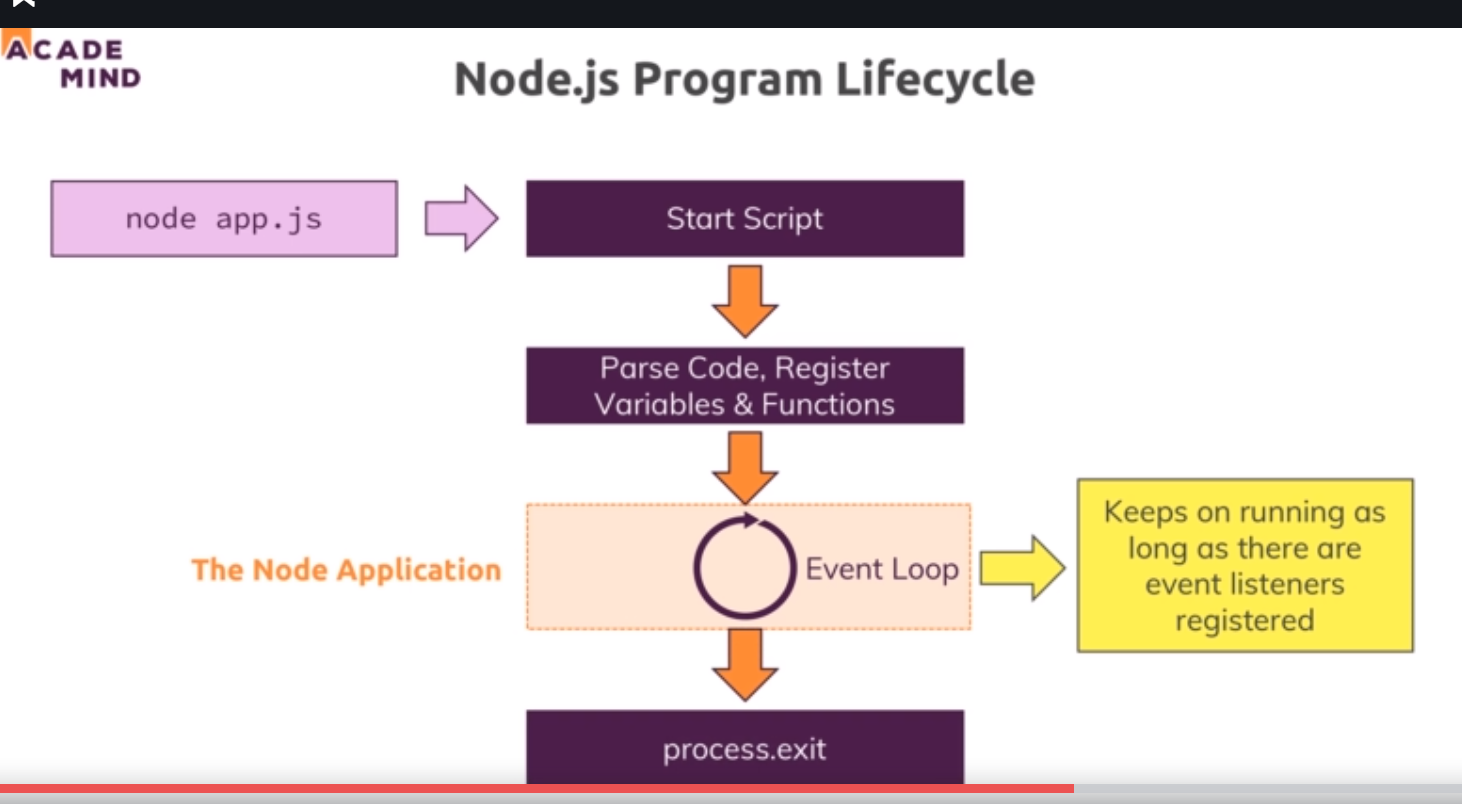
now if you run the code in console, the cursor will not get to next line because this process is still running. This file execution did’nt finish because now we got a ongoing looping process where it will keep on listening for requests. This is how you want it, right. You want web server that keeps on listening for requests.

Now if you open browser and hot localhost:3000, you can see your request is logged in console.

So above lines gives you fully functional or almost fully function web server. And this is how you create servers in nodejs.

26)The nodejs lifecycle and Event loop

Lets understand what happened in last lecture.



So we executed our file, app.js. and this essentially started a 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 important concept in nodejs called 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 registered and we never unregistered is that incoming request listener**, we passed or we setup with the help of createServer. We passed a function to createServer and that is ongoing event listener, one we did’nt unregister and we should’nt because our server of course stay up and running. So our core node application is managed by this event loop, all our code is managed by this . and as I mentioned node js uses such event driven approach for all kinds of stuff not just for managing that server but that ofcourse is crucial part. but you will see a lot throughout this course. For example when you 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 is done. And nodejs uses this pattern because it actually executes single threaded javacsript. So entire node process basically uses one single thread on our computer , it’s running on. Now as might have guessed if we crate server with nodejs, it should ofcourse be able to handle multiple, tens of thousnads 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 process where in the end it always keeps on running and just executes code when a certain event occurs. So in general it is always avalible and whilst this might sound like ok but if I got 2 incoming requests, it needs to handle 2 events, well it is superfast in handling these requests and actually behind the scenes it does some multithreading by leveraging the operating system but this event loop is core thing that you have to keep in mind that nodejs basically has an ongoing loop as long as there are listeniers and createServer creates a listerner which never stops. But if you eventually were to unregister and you can do this with process.exit, it would end .

App.js-

const http = require('http');

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

console.log(req);

process.exit();

});

server.listen(3000);

now after we send requests, it is logged on console. We can see that in console, we are back in new line because now it did quite that process.

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 back control to our terminal here.

27)Controlling nodejs 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).

28)Understanding the requests

Lets see request object that we logged on console. Now this is very complex object with lot data as well as functions. So it is quite complex object but we also see that for example we have some headers. Headers as I have already mentioned earlier is a metadata, meta information added to a request and also too responses by the way. There are few important fields in request that you need. Lets print them-

const http = require('http');

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

console.log(req.url, req.method);

console.log(req.headers);

process.exit();

});

server.listen(3000);

**D:\Max's Courses\nodejs>node app.js**

**/ GET**

**{ host: 'localhost:3000',**

**connection: 'keep-alive',**

**'cache-control': 'max-age=0',**

**'upgrade-insecure-requests': '1',**

**'user-agent':**

**'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36',**

**accept:**

**'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8',**

**'accept-encoding': 'gzip, deflate, br',**

**'accept-language': 'en-US,en;q=0.9' }**

29)Sending Responses

We can log the response object just like request obect but it will not hold any interesting data. Instead we can use it to fill it will data, that we want to send back. We can use couple of methods on it.

App.js-

const http = require('http');

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

console.log(req.url, req.method);

console.log(req.headers);

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

res.write('<html>');

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

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

res.write("</html>");

res.end();

});

server.listen(3000);

Here we used setHeader. nowContent-Type is default header which browser knows and understands. Then we pass value of header. We know what this header does(it tells browser , whih type of content is in body). Now there is ofcourse only certain set of supported headers, which browser understands. In next lecture you will link where you can see which headers you can set. You dnt 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.we have set the content type but html is missing. We can add html by calling write. Write allows us to write some data to response and this basically works in chunks , you could say or in multiple lines. This would be good picture to look at this. You write multiples lines of response.

This is very complex way of writing html code. We will learn about a easier way later. So this will be written to response line by line. Then we tell node that we are done with creating response. We do this by calling end. Now we must not write any more. Now we should not call write on response , because it will cause error.because we must not change the response after we ended it, Because this is basically the part where we will send response to client. Nodejs will send it back to the client.

Now save and run your app. js and run it. In network tap of browser, you can see the request, it’s header and response. Well later we will learn about simpler way of doing that by using express framework. But its important that you understand all the nittiy gritty details that go behind the scenes. Here we understand it by writing all the nitty gritty details.

30)Request and response headers

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

The following article provides a great overview of available headers and their role:

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

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

31)Routing Requests

App.js-

const http = require('http');

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

const url = req.url;

if (url === '/') {

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

res.write('<html>');

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

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

res.write("</html>");

return res.end();

}

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

res.write('<html>');

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

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

res.write("</html>");

res.end();

});

server.listen(3000);

here if we enter localhost:3000, we send back a response. Now we used retrun statement before res.end(). Generally this is not required but we send it because we do not want next code to be executed. Because after calling res.end() we should not do anything with response.

In html we send a response, in response we have html. We have a form. Now in form we have we action and method attributes. Action specifies the url to which we want to send the response and method specifies the http method. We have button of type submit. When click on button, our form is submitted to url(that we specified in action) via http method (that se specify in method) . in form if we have any inputs and we have given name to them then along with form that input data will also be submitted. So here we have given name message to our input , so value that we fill in input is avalible in data submitted. It will be like that, we entered sumit I text box. Request body will be-

message=sumit

32)redirecting request

Here we want to handle get request with path ‘/message’. We want to save message in file and we want to redirect user to ‘/sood’.

Code-

App.js-

const http = require('http');

const fs = require('fs');

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

const url = req.url;

const method = req.method;

if (url === '/') {

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

res.write('<html>');

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

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

res.write("</html>");

return res.end();

}

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

console.log('got request');

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

res.statusCode = 302;

res.setHeader('LOCATION', '/sood');

return res.end();

}

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

res.write('<html>');

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

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

res.write("</html>");

res.end();

});

server.listen(3000);

We could have also used res.writeHead which allows us to write some meta information in one go.

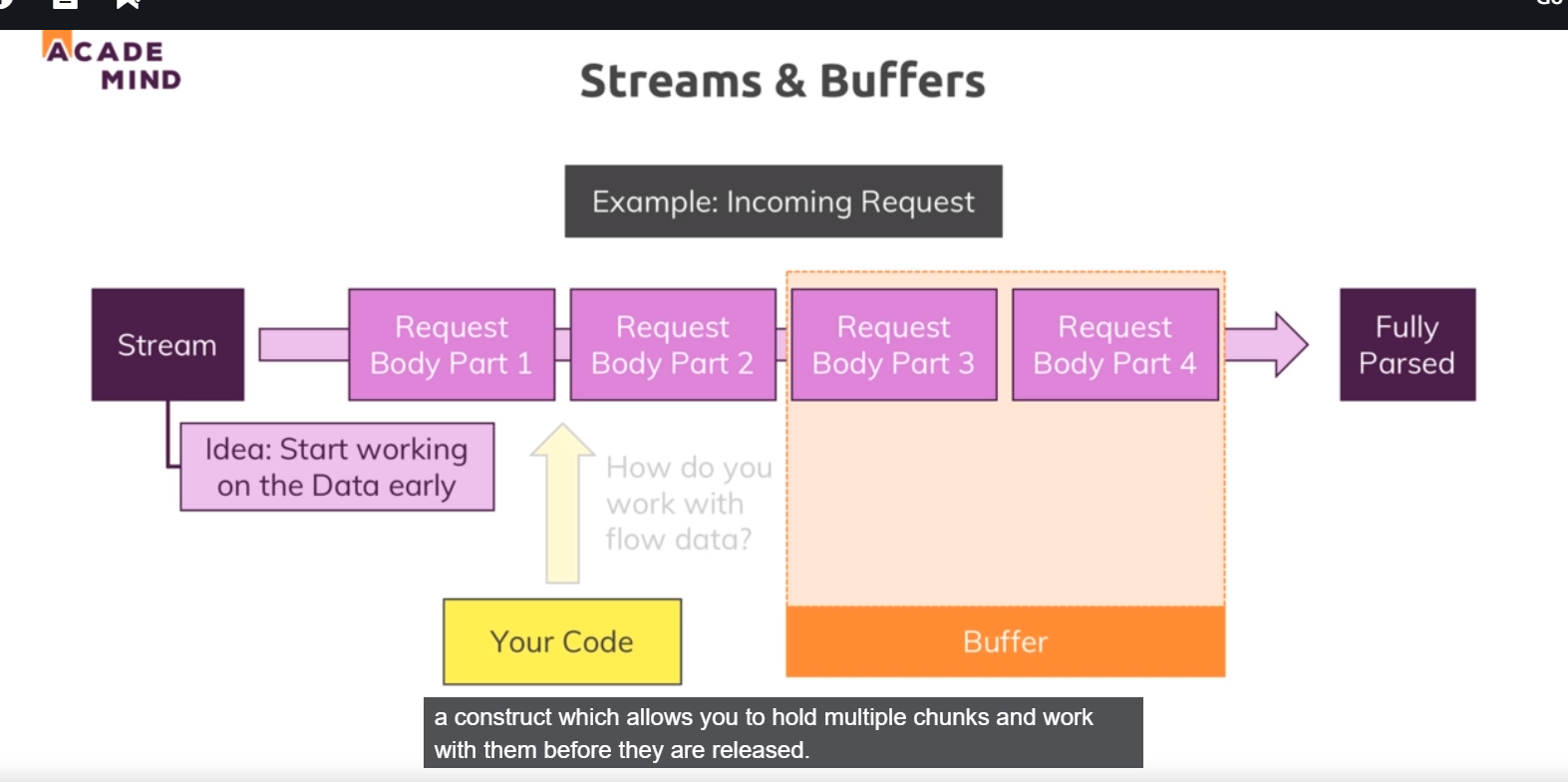
res.writeHead(302, {});

here we set status code of 302, which stands for redirection and then we pass js objects with some headers that we want to set. here we do it in 2 steps.

So when we set Header location and assign ‘/sood’, it will use the host we already are running on. Location is another header accepted by browser. So after we submit the form we are redirected to url that we specify in location header. Here we just saved the dummy data in file.

33)Parsing request bodies

Here we will parse he incoming request and get the data that is part of the request. Now how do we get access to that? We have url and method property on request object, you might think we have something like request.data that will give us data , but there is’nt. Instead incoming data is sent as stream of data and that is special construct javacsript in general knows but nodejs uses it a lot. Now what is stream of data though? There is a connected concept buffers and we will have look at both here.



So lets take incoming request as a example , there are other streams like for example when working with files, we can also work with steams. But lets stick to requests here. Our stream here is basically a 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 is done. And this is done so that we theoretically can start working on this individual chunks without having to wait for full request being read. Now for simple request like we are working with this is not required. We have only one input field data, it does;nt take so long to parse. But consider a big file being uploaded. This will take considerably longer and if we are streaming that data could make sense because it could allow you to start writing this to your disk on server whilst the data is still coming in. so that you dnt have to parse the entire file which ofcourse is 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 does’nt know in advance how complex and big they are. You can start working on data early.

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 busses, they are constantly driving. But for users or commuters being able to climb on the bus and leave the bus you need bus stops, where you intercat with the bus basivcally and that is what buffer is . a buffer is simply a construct which allows you to hold multiple chunks and work with them before they are realeased once you are done and you work with that buffer. Now that;s pretty abstract but it’s pretty easy to work with, fortunately. So lets see how it works in practice. Code-

App.js-

const http = require('http');

const fs = require('fs');

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

const url = req.url;

const method = req.method;

if (url === '/') {

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

res.write('<html>');

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

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

res.write("</html>");

return res.end();

}

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

const body = [];

req.on('data', (chunk) => {

console.log(chunk);

body.push(chunk);

});

req.on('end', () => {

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

console.log(parsedBody);

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

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

});

res.statusCode = 302;

res.setHeader('LOCATION', '/sood');

return res.end();

}

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

res.write('<html>');

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

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

res.write("</html>");

res.end();

});

server.listen(3000);

when receiving a posted message before sending the response, before writing to the file, we want to get our request data, right ? we do this by registerig a event listener on request. We have’nt don that thus far, but as I told you node uses it heavily. For createServer it implicitly created one for us. Now we do this on our own by using **on** method. Now on allows us to listen to certain events. Event that I want to listen here is data event. Our ide gives us some help, it tells which events can I listen to on request. I want to listen to data event.data event will be fired whenever a new chunk is ready to be read. You remember buffer thing, this is basically helping us with that. As a second argument to on we pass a function which should be executed for any data event. This function will receive a chunk a data, that is ready to be read. In function we push new chunk to body variable.

Then register new event listener, it is end. It will be fired when we are done parsing the incoming request data or incoming request. We pass it a callback that will be executed when this event happens, now in this function we can rely on all chunks being read and they are all stored in body variable now. Now in order to interact with them, to work with these,we need to buffer them. remember that bus that stop concept? we get all these chunks we noe need to do something to be able to work with them to basically have one place where the bus stops and we can interact with. Buffer object is made avalaible by nodejs gloabally. Then we use concat method on it. So this will in the end create new buffer and add all the chunks from inside my body to it. Then I can calltoString() methid on buffer to convert it into string. This works only because I know that incoming data will be text. If it would have been file, we have to do something different. here we just save it in a file.

Now if you are frightened by, how complex is nodejs, this is raw logic, later we will use express js which will hide all this raw logic. But we need to understand what actually is happening and why we are using tools like expressjs.

See some questions here-

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5461316>

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5473098>

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5555620>

34)Understanding Event Driven Code execution

Here we saw that when pass callbacks to function, they will be executed later point of time. like to setTimeout we pass a function which will be executed later, after time lapses/

35)Blocking and Non-Blocking code

We have 2 methods writeFile and writeFileSync. Later one is blocking code. Working for file is avalibel in 2 modes, synchronous and asynchronous. In synchronous mode we do not move to next line until that operation is finished(in this case when file is created). Even new incoming requests of other users will not be handled until that file operation is done. We do not want that. Therefore you should not use writeFileSync.

writeFileSync accepts 2 argumnts, name of file and data to be written.

But writeFile accepts 3 argumnts, first 2 are same, third one is callback which should be excuted when operation is completed. This callback gets err object as a argument. This will be null if no error occurs. But if there is a error like permission issue, you can handle it here in this function, gracefully by returning some other response, i.e by showing to user that error occurred. We will see error handling later in this course. We move response code in callback, because response should be sent only when we are done with writing file.

Code-

const http = require('http');

const fs = require('fs');

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

const url = req.url;

const method = req.method;

if (url === '/') {

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

res.write('<html>');

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

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

res.write("</html>");

return res.end();

}

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

const body = [];

req.on('data', (chunk) => {

body.push(chunk);

});

return req.on('end', () => {

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

console.log(parsedBody);

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

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

res.statusCode = 302;

res.setHeader('LOCATION', '/sood');

return res.end();

});

});

}

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

res.write('<html>');

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

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

res.write("</html>");

res.end();

});

server.listen(3000);

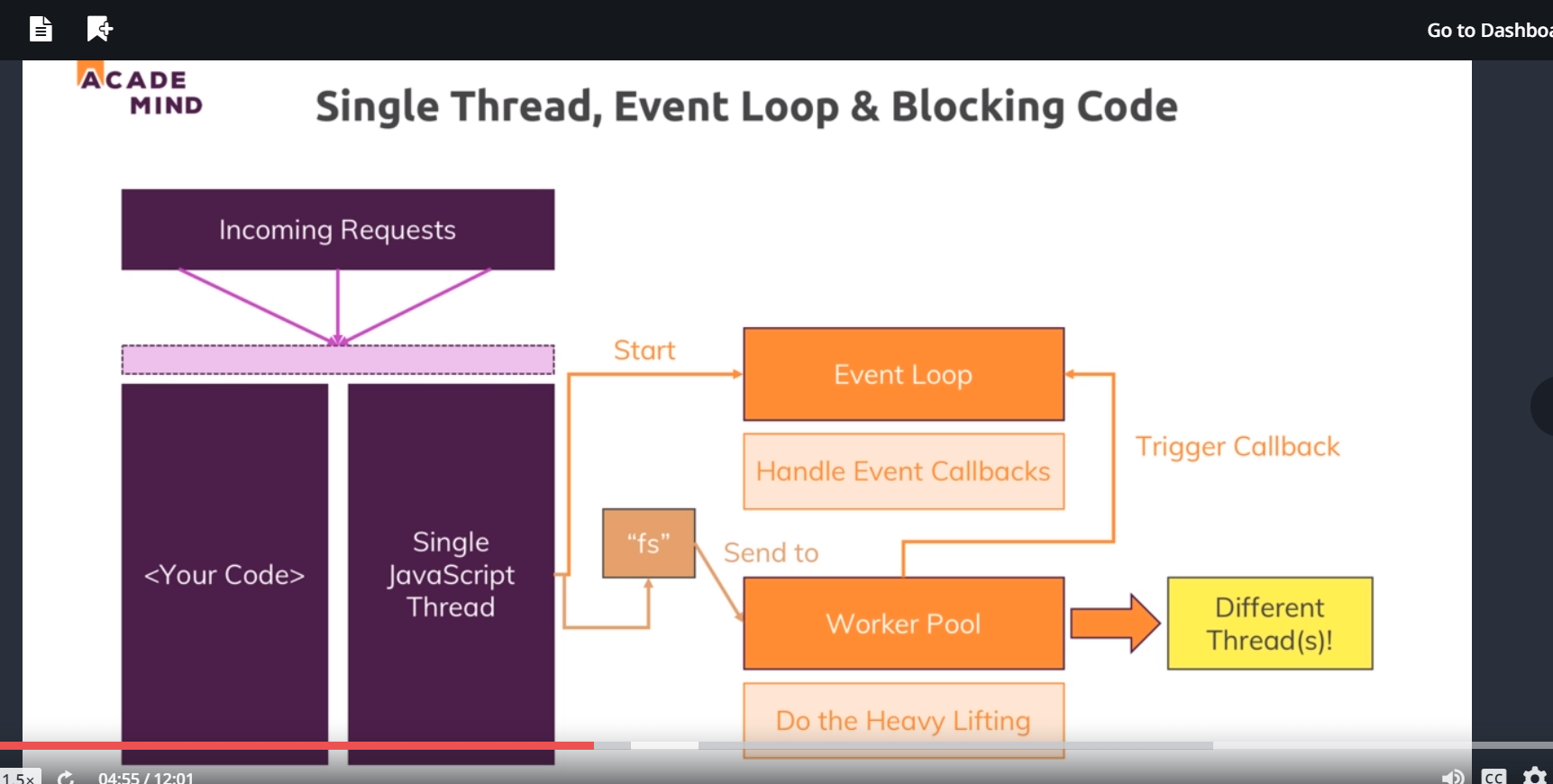
so we will start writing file when we are done with parsing the request body, then when we are done with writing file, then we will send response. So both of these are event listeners. This is actually pretty standard for nodejs. You have this event driven architecture, where you tell nodejs please do something and then it will go ahead and offload that process to operating system which does use multi-threading and so on and will then continue its event loop to listen for an event callbacks and always just dispatch tiny actions like that to never block code execution and then always just come back once an operation is done by operating system and so on. So this is what nodejs does here and why it is highly performant. Because it never blocks your code, it never blocks your server. It just goes ahead and tells operating system do that, do this and then eventually comes back and does something in callback like send a response which is not a blocking because it is super fast.

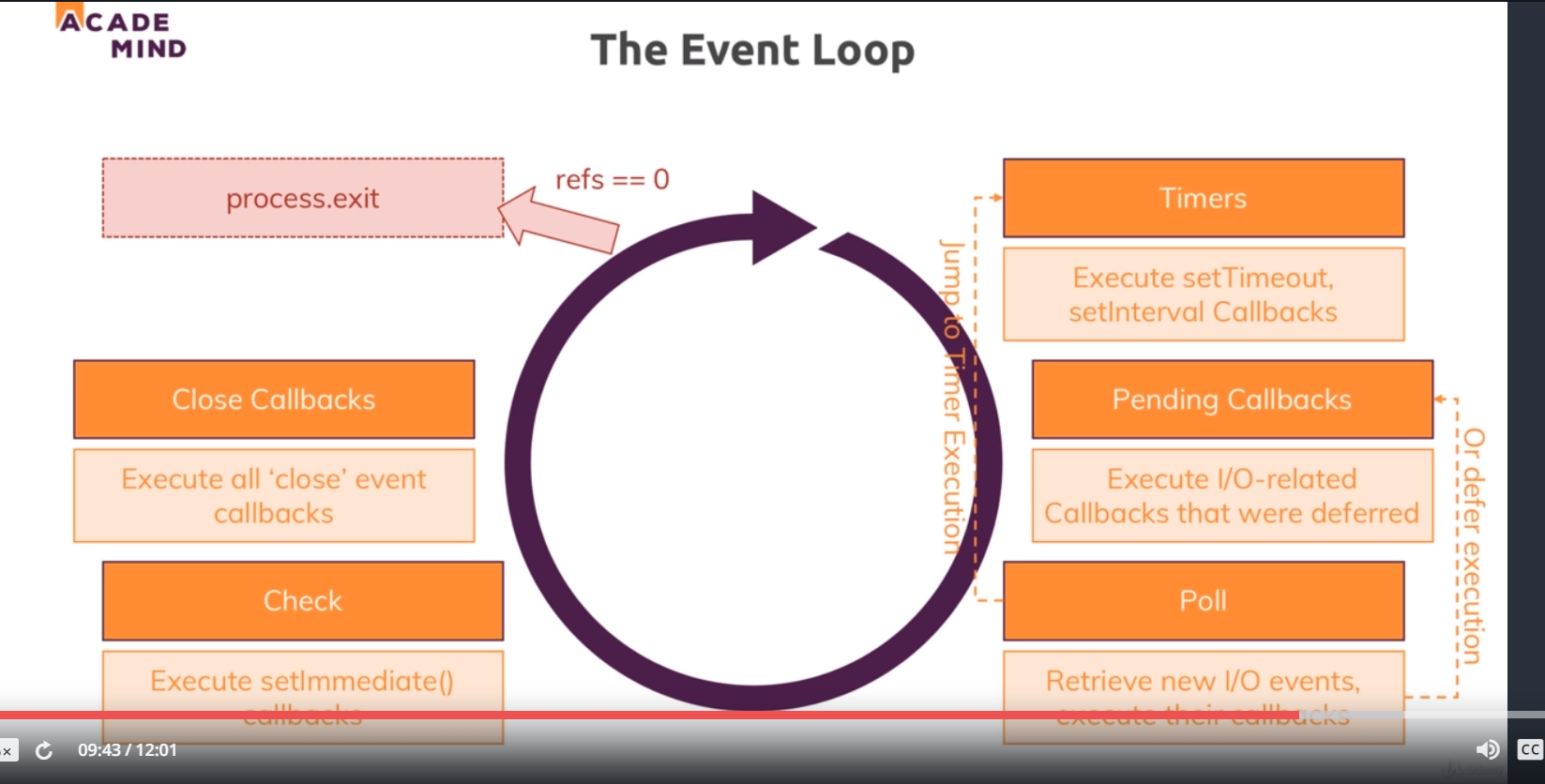
So we should use fileSynce because of asynchronous nature of nodejs.

Now we will move on and make our life easier. We wnt write this nitty gritty code by ourself. But is it super crucial that you understand what is going on in here.

36)nodejs- Looking behind the Scenes

Here event loop was explained. See video.





My questions answered-

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5480144>

Additional link-

<https://www.youtube.com/watch?v=zphcsoSJMvM>

<http://voidcanvas.com/nodejs-event-loop/> (this is important)

37)Using the node Modules System

We got all our code in one file and typically we work with multiple files. Here we will create a different file for routing logic.

App.js-

const http = require('http');

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

const server = http.createServer(routes);

server.listen(3000);

routes.js-

const fs = require("fs");

const requestHandler = (req,res) => {

const url = req.url;

const method= req.method;

if (url === '/') {

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

res.write('<html>');

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

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

res.write("</html>");

return res.end();

}

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

const body = [];

req.on('data', (chunk) => {

body.push(chunk);

});

return req.on('end', () => {

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

console.log(parsedBody);

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

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

res.statusCode = 302;

res.setHeader('LOCATION', '/sood');

return res.end();

});

});

}

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

res.write('<html>');

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

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

res.write("</html>");

res.end();

};

module.exports = requestHandler;

// module.exports = {

// handler: requestHandler,

// someText: 'somestring'

// };

// module.exports.handler = requestHandler;

// module.exports.someText = 'SomeString';

// exports.handler = requestHandler;

// exports.someText = 'SomeString';

here we export function in one routes.js and import it in app.js.

now one important note about node\_modules system, the file content is actually cached by node and we cannot edit it externally. So if we somehow would define routes as an object and we tried to add new property on the fly here, this would not manipulate original file. so this (rotes.js) is basically locked and not accessible from outside. We can only export stuff that we can read from outside. Although you can have functions exported, that can change stuff inside of the file. we will see that throughout the course obviously.

There is one other syntax that you could use for exporting. Sometimes you export many things-

In routes.js-

module.exports = {

handler: requestHandler,

someText: 'somestring'

};

Or you can also do it in this way-

module.exports.handler = requestHandler;

module.exports.someText = 'SomeString';

there is a shortcut, you can omit module. This is simply a shortcut provided by nodejs, not some general javascript magic-

exports.handler = requestHandler;

exports.someText = 'SomeString';

In app.js-

const http = require('http');

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

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

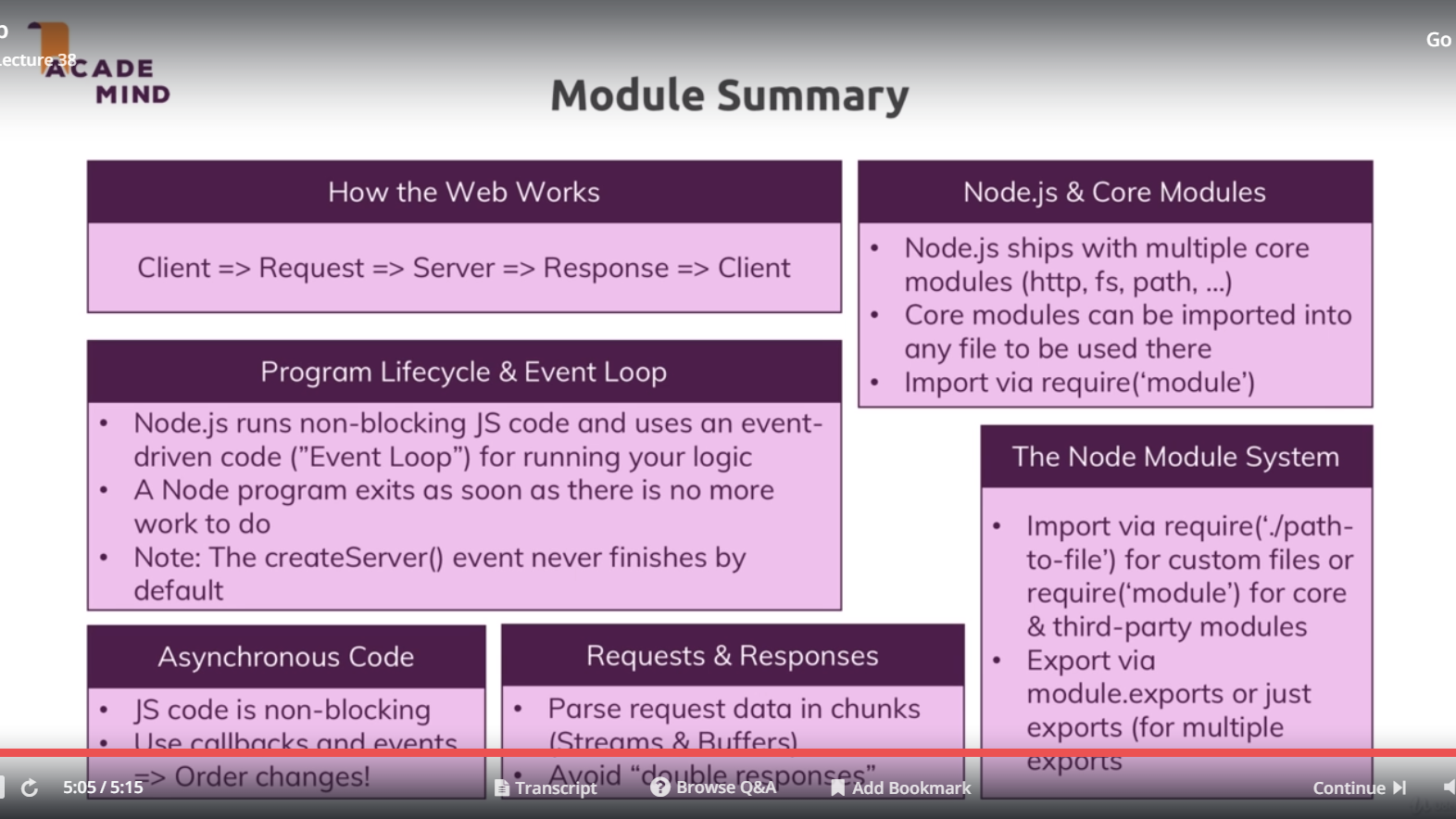
server.listen(3000);

question-

<https://www.udemy.com/nodejs-the-complete-guide/learn/v4/questions/5661398>

here i want to go to definations of routes in routes.js from app.js. it was nor working fine so adam insisted this approach.

38)Wraps Up



39)Useful resources and links

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/>