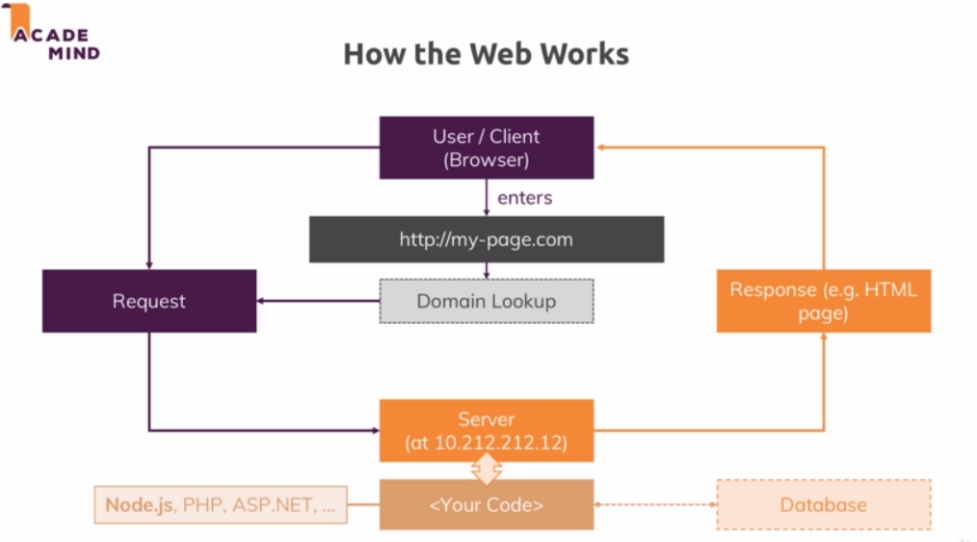
**How the Web Works:**



**HTTP/HTTPS**

The request and response is done using some protocol which is an standardized way of communicating. To correctly handle request and send back a response the client can work with - some rules have to be followed and these rules are defined by some protocol we use which is HTTP/HTTPS.

HTTP Hyper Text Transfer Protocol defines how a valid request look like and how the data should be transferred from client to the server and the other way round.

HTTPS Hyper Text Transfer Protocol Secure is the same as HTTP with SSL encryption turned on where all the data that is transmitted is actually encrypted.

**Creating a Node Server**

const http = require('http');

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

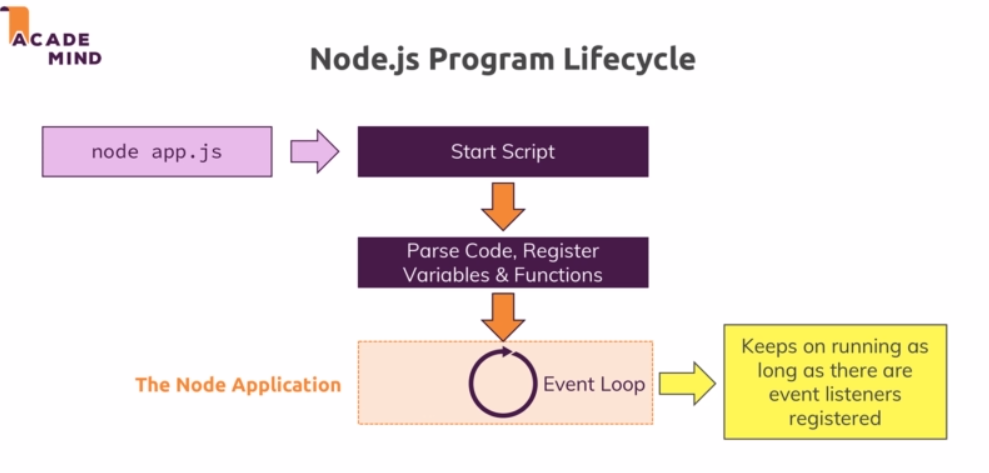
    console.log(req);

//process.exit();

});

server.listen(3000);

**Nodejs Program Life Cycle and Event Loop**

****

An event loop is a process which is managed by Nodejs which keeps on running as long as there is work to do - event listeners registered. One event listener we did registered here and never unregistered is that incoming listener inside createServer - which is on going event listener.

So all our core node application is managed by this event loop.

The entire node process basically uses a single thread.

If we were to unregister the listener then we can simply do it using:

**process.exit();**

**Understanding requests**

req.url - / (everything after host:port)

req.method - Http method

req.headers

**Sending response**

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

res.write('<html>')

res.write('<head><title>Learning Node</title></head>')

res.write('<body><h1>Hello from my first node server</h1></body>')

res.write('<html>')

res.end()

**Routing Requests**

const url = req.url;

if(url==='/') {

res.write('<html>');

    res.write('<head><title>Learning Node</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();

}

**Redirecting requests:**

const method = req.method;

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

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

    res.statusCode = 302;

    res.setHeader('Location', '/');

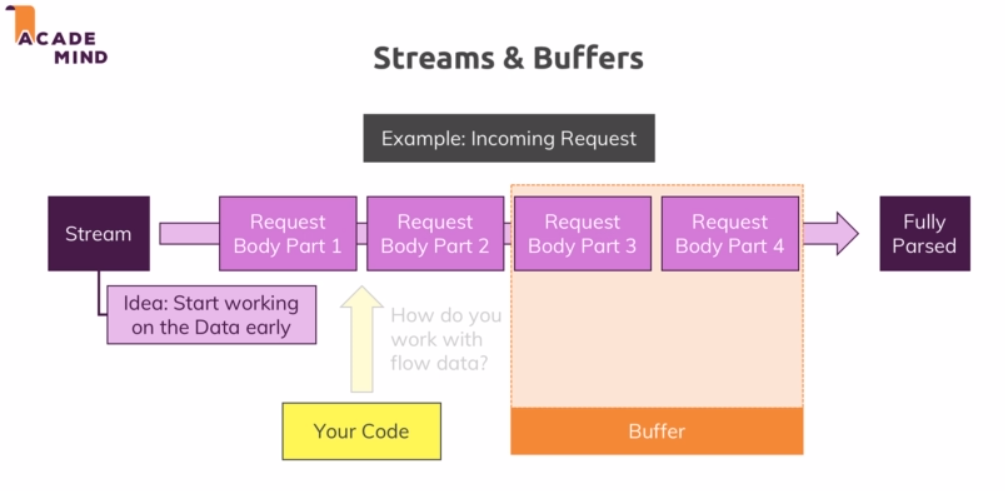
    return res.end();

}

**Parsing Requests:**

The incoming data is send back as a stream of data.The request is simply read by node in chunks and this is done theoretically so that we can start working on chunks without having to wait for the full request being read. Consider a file being uploaded which will considerably take longer and therefore streaming the data could make sense because it could allow to to writing this to your disk where your app runs on your server while the data is coming in. So we don’t have to parse the entire file which is of course taking time and we have to wait for it to fully been uploaded before we can do anything with it. So this is how node handles all requests.

W we can not arbitrarily start working with chunks, instead to organize using chunks we use so called buffer.



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

    const body = [];

    req.on('data', (chunk) => {

        body.push(chunk);

        console.log(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', '/');

    return res.end();

}

**Understanding Event Driven Code Execution**

The order of execution of the code is not necessarily the order in which it is written. The Nodejs uses a pattern where we pass a callback function and Nodejs will execute them at a later point of time which is called asynchronously. In these cases, Nodejs won’t immediately execute that function instead what it does when it first encounter that function it will attach a new event listener internally and it will then call that callback once it is done. So we can think of this like -

Nodejs has its internal registry of events and listeners to these events and when something happens, it will go through that registry and invoke the corresponding listener to that event. It will never pause the code execution which is the most important thing to understand.

**Blocking and Non-Blocking Code**

writeFileSync - means write file synchronous which is a special method which will actually block code execution until the file is written.

Working with files is available in two modes:

* Synchronous:Block execution of the next line of code until the operation is done. writeFileSync(filepath, data)
* Asynchronous: writeFile(filepath, data, (err) = > {})

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', '/');

       return res.end();

  });

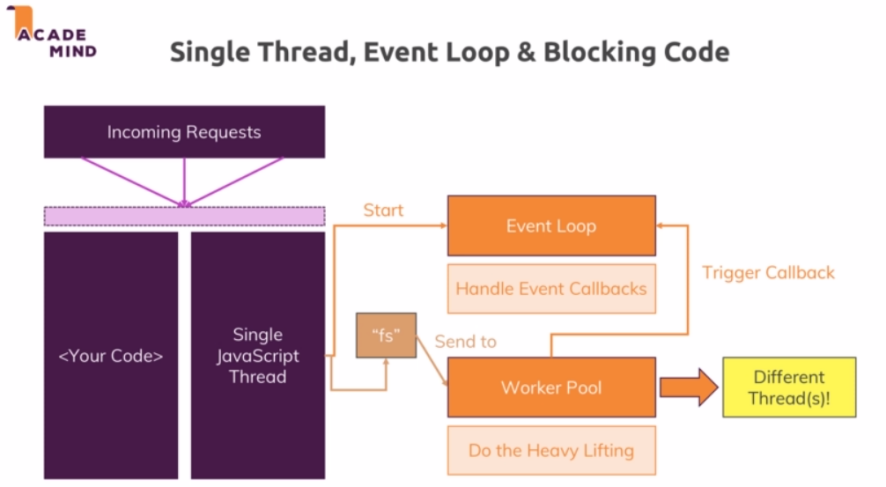
});

**Nodejs - Looking behind the Scenes \*\*(Watch this again)**

Nodejs uses single Javascript Thread. A thread is basically like a process in OS. The obvious question then is how it is able to handle multiple requests because if we are not able to assign a new thread for each request, they ultimately end up running in one and the same thread and this of course poses a security question - can you then access data from one request A from another request B and most importantly the other implication is if the request A is being processed, request B can’t be handled?

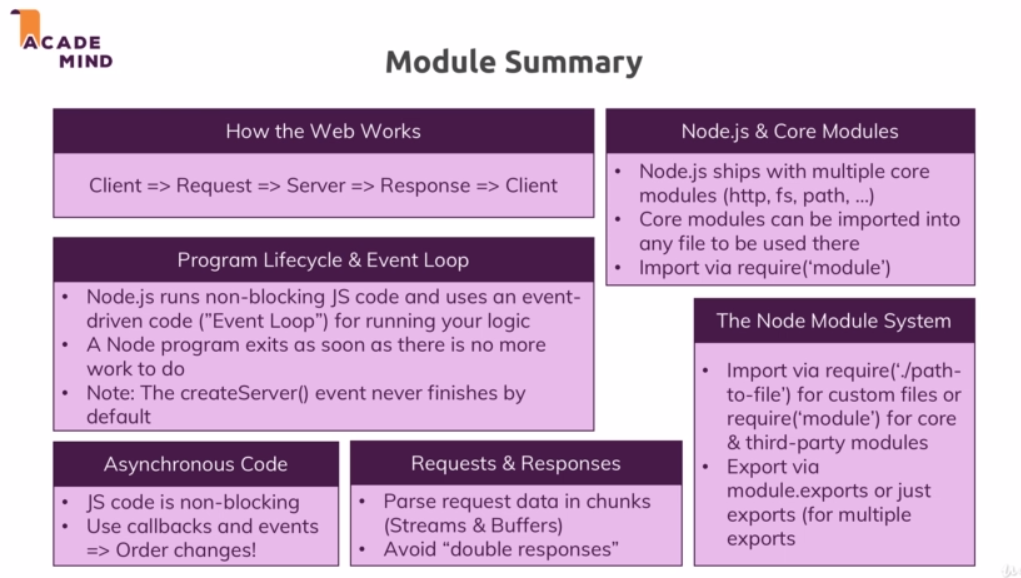
Let’s start the answer with performance - Let say we have some code which accesses the file system. Now working with files takes longer time to process. So if we are doing this on an incoming request, a second request might have to be await because we are not able to handle it yet or even it gets declined - basically our web site is down for that user. Now one important construct is the event loop - which is automatically started by Nodejs when your program starts which is responsible for handling event callbacks. So the event loop is basically responsible for running the code when a certain event occurs. It’s aware of all these callbacks.

So the long taking time tasks are not handled by the event loop. The callback that we might have defined on file write - that code will be handled in the event loop. The long time taking operations are sent to a worker pool which is also spun up and managed by the Nodejs automatically. This worker pool is responsible for all the heavy lifting. This worker pool is totally detached from the Javascript code and it runs on different threads - it can spin up multiple threads. So this worker pool is therefore doing all the heavy lifting.The one connection to the event loop we will have though is that once the worker is done - it will trigger the callback for the read file operation and since the event loop is responsible for the events and the callbacks, this will in the end will end this event. So there Nodejs will basically executes the appropriate callback.This is all built into Nodejs.



Let’s get to know the event loop now - It is a loop which is run or started by the Nodejs that keeps the Nodejs process running and handles all the callbacks. It has a certain order in which goes through the callbacks.

* At the beginning of the iteration it checks if there are any timer callbacks it should execute. setTimeout() and setInterval() are those timer function
* Then as the next step it checks for the other callbacks. For example I/O (like file operation or network operation).Now its 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 pending callbacks it will continue its loop iterations and postpone these callbacks for the next iteration to execute them.
* Then it will enter the Poll phase which is basically a phase where Nodejs will look for I/O events and do its best to invoke their callbacks if possible. If it is not possible it will defer the execution and register it as the pending callback.



**Useful resources:**

Official Node.js Docs: [https://nodejs.org/en/docs/guides/](https://nodejs.org/en/docs/guides/" \t "https://www.udemy.com/course/nodejs-the-complete-guide/learn/lecture/_blank)

Full Node.js Reference (for all core modules): [https://nodejs.org/dist/latest/docs/api/](https://nodejs.org/dist/latest/docs/api/" \t "https://www.udemy.com/course/nodejs-the-complete-guide/learn/lecture/_blank)

More about the Node.js Event Loop: [https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/](https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/" \t "https://www.udemy.com/course/nodejs-the-complete-guide/learn/lecture/_blank)

Blocking and Non-Blocking Code:

[https://nodejs.org/en/docs/guides/dont-block-the-event-loop/](https://nodejs.org/en/docs/guides/dont-block-the-event-loop/" \t "https://www.udemy.com/course/nodejs-the-complete-guide/learn/lecture/_blank)