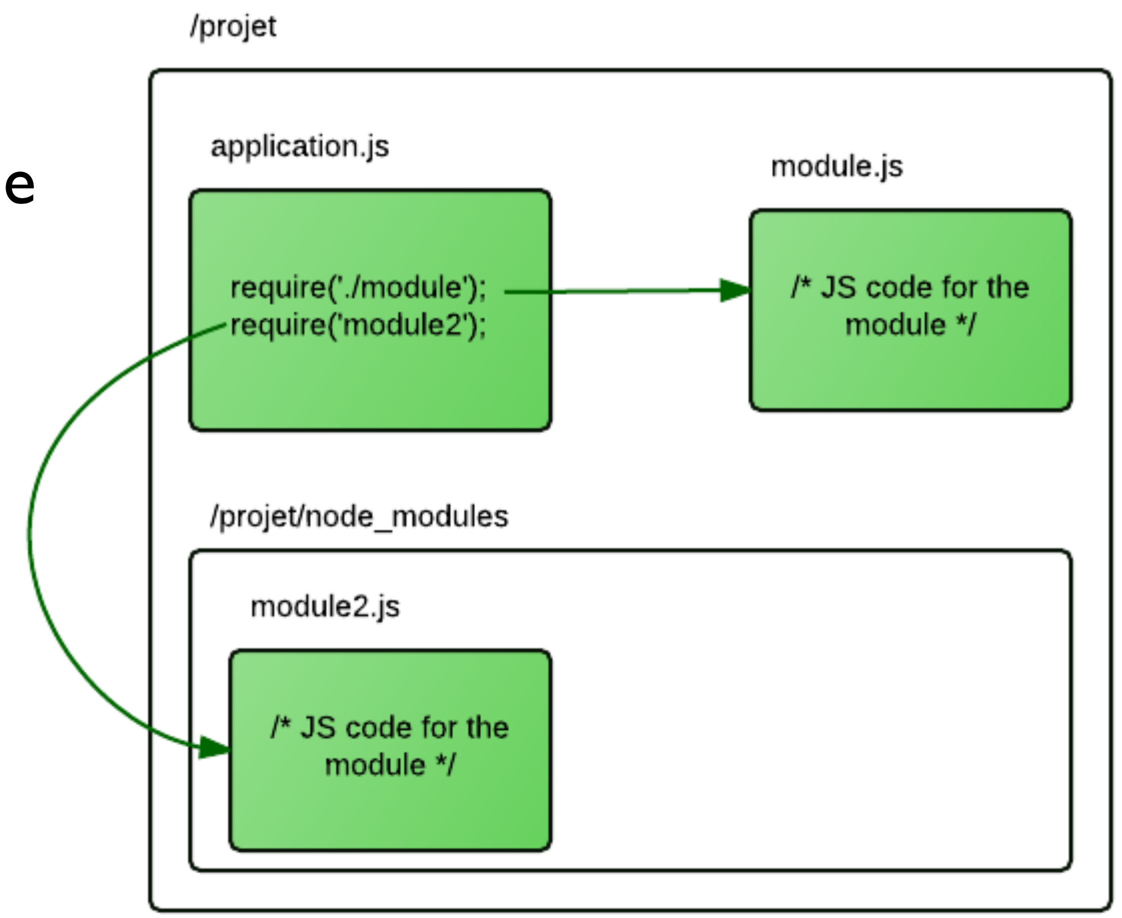
Modules in NodeJS

Modules in NodeJS

- Consider modules to be the same as JavaScript libraries.
- A set of functions you want to include in your application.

- Node implements **CommonJS** Modules specs.
  - CommonJS module are defined in normal .js files using module.exports
  - In Node.js, each file is treated as a separated module



https://openclassrooms.com

Type of Modules

- Built-in Modules
  - Node.js has a set of built-in modules which you can use without any further installation.
  - buffer, fs, http, path, etc.
  - Built-in Modules Reference
- 3rd party modules on www.npmjs.com
- Your own Modules
  - Simply create a normal JS file it will be a module *(without affecting the rest of other JS files and without messing with the global scope).*

Include Modules – `require()` function

- The basic functionality of `require` is that it reads a JavaScript file, executes the file, and then proceeds to return the `module.exports` object.
  - `const path = require('path’);`
  - `const config = require('./config');`

- Rules:
  - if the file doesn't start with "**./**" or "/", then it is either considered a **core module** (and the local Node path is checked), or a dependency in the local **node_modules** folder.
  - If the file starts with "**./**" it is considered a relative file to the file that called `require`.
  - If the file starts with "/", it is considered an absolute path.
  - If the filename passed to require is actually a directory, it will first look for package.json in the directory and load the file referenced in the main property. Otherwise, it will look for an index.js.
  - NOTE: you can omit ".js" and require will automatically append it if needed.

How `require('/path/to/file')` works

- Node goes through the following sequence of steps:
  1. Resolve: to find the absolute path of the file
  2. Load: to determine the type of the file content
  3. Wrap: to give the file its private scope
  4. Evaluate: This is what the VM actually does with the loaded code
  5. Cache: when we require this file again, don't go over all the steps.

- **Note**: Node core modules return immediately (no resolve)

What's the wrapper?

- `node -p "require('module').wrapper"`

1. Node will wrap your code into:

```
(function (exports, require, module, __filename, __dirname){
        let exports = module.exports;
    // this is why can use exports and module objects.. etc in your code
    without any problem, because Node is going to initialize these and pass
    them as parameters through this wrapper function
});
```

2. Node will run the function using .apply()
3. Node will return the following:

```
return module.exports;
```

module.exports

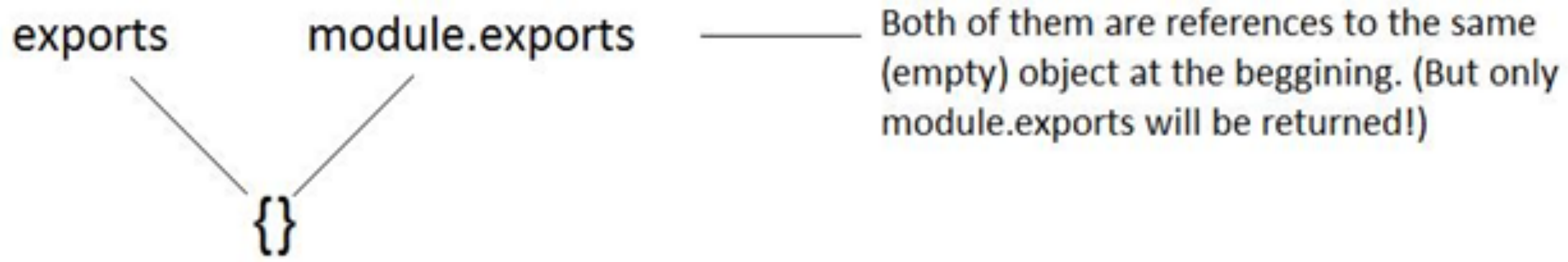
- Think about this object (`module.exports`) as return statement.

```
// helloModule.js
var sayHi = function(){
                console.log('hi');
             }

module.exports = sayHi;
```

```
// app.js
var hello = require('./helloModule');

hello();
```

exports vs module.exports

- `exports` **object is a reference to the** `module.exports`**, that is shorter to type**


exports
module.exports
{}
Both of them are references to the same (empty) object at the beggining. (But only module.exports will be returned!)

- **Be careful when using** `exports`**, a code like this will make it point to another object. At the end,** `module.exports` **will be returned.**

```
exports = function doSmething() {
    console.log('blah blah');
}
```

doSomething() isn't exported.

Create your own module

play folder

```
// play/violin.js
const play = function() { console.log("First Violin is playing!"); }
module.exports = play;
```

```
// play/clarinet.js
const play = function() { console.log("Clarinet is playing!"); }
module.exports = play;
```

```
// play/index.js
const violin = require('./violin');
const clarinet = require('./clarinet');
module.exports = { 'violin': violin, 'clarinet': clarinet };
```

```
// app.js
const play = require('./play');
play.violin();
play.clarinet();
```

Using Modules – Pattern 1

```
// Pattern1 - pattern1.js
module.exports = function () {
    console.log('Josh Edward');
};
```

```
// app.js
const getName = require('./pattern1');
getName(); // Josh Edward
```

Using Modules – Pattern 2

```
// Pattern2 - pattern2.js
module.exports.getName = function () {
    console.log('Josh Edward');
};

// OR
exports.getName = function () {
    console.log('Josh Edward');
};
```

```
// app.js
const getName = require('./pattern2').getName;
getName(); // Josh Edward

// OR
const person = require('./pattern2');
person.getName(); // Josh Edward
```

Using Modules – Pattern 3

```
// Pattern3 - pattern3.js
class Person {
    constructor(name) {
        this.name = name;
    }

    getName() {
        console.log(this.name);
    }
}
module.exports = new Person('Josh Edward');
```

Warning: Not good practice

```
// app.js
const personObj = require('./pattern3');
personObj.getName(); // Josh Edward
personObj.name = 'Emma Smith';
personObj.getName(); //Emma Smith
const personObj2 = require('./pattern3');// cached in the same module
personObj2.getName(); //Emma Smith
```

Using Modules – Pattern 4

```
// Pattern - pattern4.js
class Person {
    constructor(name = 'Josh Edward') {
        this.name = name;
    }

    getName() {
        console.log(this.name);
    }
}

module.exports = Person;
```

```
// app.js
const Person = require('./pattern4');
const personObj1 = new Person();
personObj1.getName() // Josh Edward
personObj1.name = 'Emma Smith';
personObj1.getName(); //Emma Smith

const Person2 = require('./pattern4');
const personObj2 = new Person2();
personObj2.getName(); // Josh Edward
```

Using Modules – Pattern 5

```
// Pattern5 - pattern5.js
const name = 'Josh Edward';
function getName() {
    console.log(name);
}
module.exports = {
    getName: getName // closure
}
```

```
// app.js
const getName = require('./pattern5').getName;
getName(); // Josh Edward
```

Node core libraries

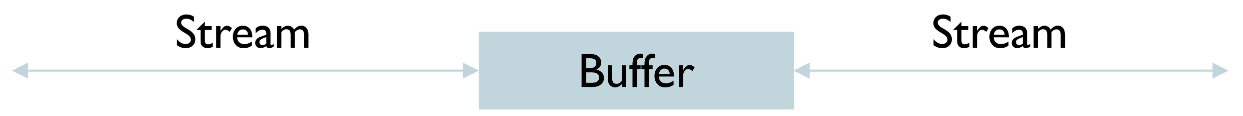
- Node provides many core libraries

```
const util = require('util'); // We do not use ./ before the filename
const sayHi = util.format("Hi, %s", 'Josh');
util.log(sayHi); //22 Feb 11:04:59 - Hi, Josh
```

- Read API documentation

Buffers and Streams

- **Buffer**: A chunk of memory allocated outside the V8 heap (intentionally limited in size, we cannot resize an allocated buffer).
- Buffer class are similar to arrays of integers but correspond to fixed-sized, raw memory allocations outside the V8 heap. The size of the Buffer is established when it is created and cannot be changed.
- **Stream**: a sequence of data made available over time. Pieces of data that eventually combine into a whole.



- *Example*: If you stream a movie over the internet, you are processing the data as being received. there will be a buffer inside your browser that receive the data from the server, process it when enough data is available (few seconds of pictures) , and stream it out to your monitor.

Character set vs. Encoding

- **Character set:** A representation of characters as numbers, each character gets a number. Unicode and ASCII are character sets. Where character get a number assigned to them.
- **Encoding:** How characters are stored in binary, the numbers (code points) are converted and stored in binary.

	h	**e**	**l**	**l**	**o**
Unicode character set	**104**	**101**	**108**	**108**	**111**
UTF-8 encoding	**01101000**	**01100101**	**01101100**	**01101100**	**01101111**

- Remember in HTML when a binary response comes from the server, it's mandatory in HTML5 to specify the encoding in the header `<meta charset='utf-8'>` Specify the character encoding for the HTML document.

Buffers Encoding

- Buffers data are saved in binary, so it can be interpreted in many ways depending on the length of characters. This is why we need to specify the char-encoding when we read data from Buffers. If we don't specify the encoding then we will receive a Buffer object with the data represented in a hex format in the console.

Buffer Examples

```
var buf = Buffer.alloc(8); // allocate a Buffer with 8 bytes
var buf = Buffer.allocUnsafe(8); //allocated 8 bytes Buffer with random data
var buf = Buffer.allocUnsafe(8).fill(); // fill buffer with 0
var buf = Buffer.from('Hello'); // create Buffer of 5 bytes and fill it
var buf = new Buffer('Hello', 'utf8'); //deprecated
console.log(buf);
// 48 65 6c 6c 6f stored internally in binary but displayed as hex in CLI

console.log(buf[2]); // 108 (Charset)

console.log(buf.toString()); // Hello

console.log(buf.toJSON());
// {type: 'Buffer', data = [72, 101, 108, 108, 111 ] } Unicode characterset

buf.write('wo'); // overwrite data in the buffer without changing its size
console.log(buf.toString()); // wollo
```

Slice a buffer

- A buffer can be sliced into a smaller buffer by using the slice() method:

```
var buffer = new Buffer('this is the string in my buffer');
var slice = buffer.slice(10, 20);
```

- Here we are slicing the original buffer that has 31 bytes into a new buffer that has 10 bytes equal to the 10th to 20th bytes on the original buffer.

- **Note** that the slice function does not create new buffer memory: it uses the original untouched buffer underneath.
- **Tip:** If you are afraid you will be wasting precious memory by keeping the old buffer around when slicing it, you can copy it into another

Copy a buffer

- You can copy a part of a buffer into another pre-allocated buffer like this:

```
var buffer = new Buffer('this is the string in my buffer');
var targetBuffer = new Buffer(10);
var targetStart = 0,
    sourceStart = 10,
    sourceEnd = 20;
buffer.copy(targetBuffer, targetStart, sourceStart, sourceEnd);
```

- Here we are copying part of buffer (positions 10 through 20) into new buffer.

Files

```
const fs = require('fs');
const path = require('path');
console.log(__dirname); // returns absolute path of current file
const greet = fs.readFileSync(path.join(__dirname, './greet.txt'), 'utf8');
console.log(greet);

const greet2 = fs.readFile(path.join(__dirname, './greet.txt'), 'utf8',
    function (err, data) { console.log(data); });
console.log('Done!');
// Hello
// Done!
// Hello
```

- Notice the Node Applications Design: any async function accepts a **callback as a last parameter** and the **callback function accepts error as a first parameter** (null will be passed if there is no error).
- Notice: `data` here is a buffer object. We can convert it with toString or add the encoding to the command

Example Read/Write Files

```
var fs = require('fs');
var path = require('path');

// Reading from a file:
fs.readFile(path.join(__dirname, '/data/students.csv'),
               {encoding: 'utf-8'}, function (err, data) {
                                                      if (err) throw err;
                                                      console.log(data);
                                        });
// Writing to a file:
fs.writeFile('students.txt', 'Hello World!', function (err) {
                                                      if (err) throw err;
                                                      console.log('Done');
                                        });
```

What's the problem with the code above?

Streams

- Collection of data that might not be available all at once and don't have to fit in memory. Streaming the data means an application processes the data while it’s still receiving it. This is useful for extra large datasets, like video or database migrations.
- Stream types:
  - Readable (fs.createReadStream)
  - Writable (fs.createWriteStream)
  - Duplex (net.Socket)
  - Transform (zlib.createGzip)

Streams

Readable Streams

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

Writable Streams

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr

Streams example

```
const fs = require('fs');
const path = require('path');

// Stream will read the file in chunks
// if file size is bigger than the chunk then it will read a chunk and emit a 'data' e
vent.
// Use encoding to convert data to String of hex
// Use highWaterMark to set the size of the chunk

const readable = fs.createReadStream(path.join(__dirname, '/card.jpg'),
    { highWaterMark: 16 * 1024 });

const writable = fs.createWriteStream(path.join(__dirname, '/destinationFile.jpg'));

readable.on('data', function (chunk) {
    console.log(chunk.length);
    writable.write(chunk);
});
```

Pipes: src.pipe(dst);

- To connect two streams, Node provides a method called pipe() available on all readable streams. Pipes hide the complexity of listening to the stream events.

```
const fs = require('fs');

const readable = fs.createReadStream(__dirname + '/card.jpg');
const writable = fs.createWriteStream(__dirname + '/destinationImage.jpg');

readable.pipe(writable);

// note that pipe return the destination, this is why you can pipe it again
to another stream if the destination was readable in this case it has to be
DUPLEX because you are going to write to it first, then read it and pipe it
again to another writable stream.
```

Zip file using streams

```
const fs = require('fs');
const zlib = require('zlib');
const gzip = zlib.createGzip();
// this is a readable & writable stream and it returns a zipped stream

const readable = fs.createReadStream(__dirname + '/source.txt');
const compressed = fs.createWriteStream(__dirname + '/destination.txt.gz');

readable.pipe(gzip).pipe(compressed);
```

- A key goal of the stream API, is to limit the buffering of data to acceptable levels such that sources and destinations of differing speeds will not overwhelm the available memory.

Node as a Web Server

- Node started as a Web server and evolved into a much more generalized framework.
- Node `http` module is designed with streaming and low latency in mind.
- Node is very popular today to create and run Web servers.

Web Server Example

```
const http = require('http');
const server = http.createServer();
```

`http.IncomingMessage`
Implements `ReadableStream` Interface

`http.ServerResponse`
Implements `WritableStream` Interface

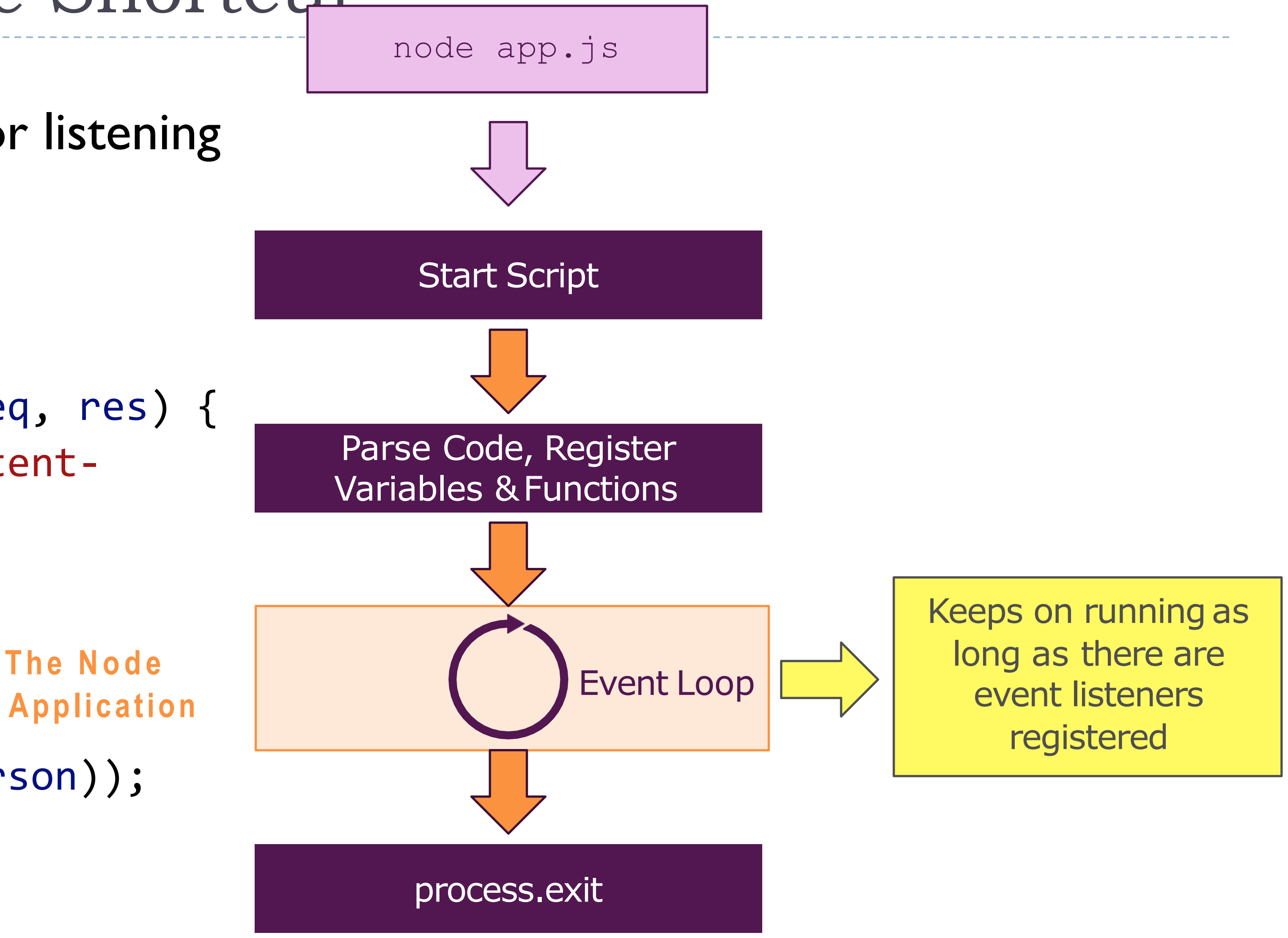
```
server.on('request', function(req, res) {
                          res.writeHead(200, {'Content-Type': 'text/plain'});
                          res.write('Hello World!');
                          res.end();
                    });
server.listen(3000);
```

After we run this code. The node program doesn’t stop.. it keeps waiting for request

Web Server Example Shortcut

- Passing a callback function to createServer() is a shortcut for listening to "request" event.

```
const http = require('http');

http.createServer(function (req, res) {
    res.writeHead(200, { 'Content-
Type': 'application/json' });
    const person = {
        firstname: 'Josh',
        lastname: 'Edward'
    };
    res.end(JSON.stringify(person));
}).listen(3000, '127.0.0.1');
```


node app.js
Start Script
Parse Code, Register Variables & Functions
The Node Application
Event Loop
Keeps on running as long as there are event listeners registered
process.exit

Send out an HTML file

- What's the problem with the code below?

```
index.html
<html>
        <head></head>
        <body>
                <h1>{Message}</h1>
        </body>
</html>
```

```
const http = require('http');
const fs = require('fs');
http.createServer(function (req, res) {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    var html = fs.readFileSync(__dirname + '/index.html', 'utf8');
    var message = 'Hello from Node.js!';
    html = html.replace('{Message}', message);
    res.end(html);
}).listen(1337, '127.0.0.1');
```

Let's create a big file!

```
const fs = require('fs');

const file = fs.createWriteStream('./big.file');

for(let i=0; i<= 1e6; i++) {
  file.write('Lorem ipsum dolor sit amet, consectetur adipisicing elit,
  sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
  enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
  aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit
  in voluptate velit esse cillum dolore eu fugiat nulla pariatur.
  Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
  officia deserunt mollit anim id est laborum.\n');
}

file.end();
```

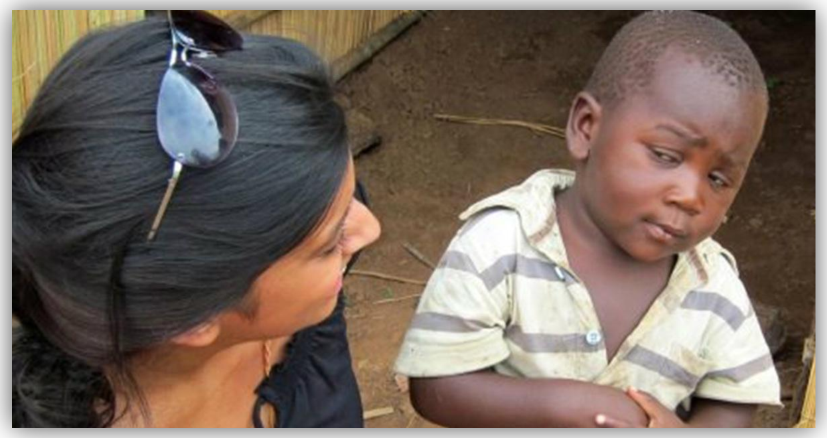
Reading the file

- What's going to happened to the Node process in memory? Will this code still work with 2 GB file or more?

```
const fs = require('fs');

const server = require('http').createServer();

server.on('request', (req, res) => {
      fs.readFile('./big.file', (err, data) => {
                  if (err) throw err;
                  res.end(data);
      });
});

server.listen(8000);
```

What can we do?

```
require('http').createServer(function(req, res) {
	var src = fs.createReadStream('/path/to/big/file');
	src.on('data', function(data) {
		if (!res.write(data)) {
			src.pause();
		}
	});
	res.on('drain', function() {
		src.resume();
	});
	src.on('end', function() {
		res.end();
	});
});
```



A Simpler solution – Use Stream

- We can simply use stream.pipe(), which does exactly what we described.

```
const fs = require('fs');

const server = require('http').createServer();

server.on('request', (req, res) => {
      const src = fs.createReadStream('./big.file');
      src.pipe(res);
});

server.listen(8000);
```

Resources

- Node Resources
  - Node Modules
  - Node HTTP
  - Anatomy of an HTTP Transaction
  - fs Module
  - path Module

- Other Resources
  - CommonJS
  - CommonJS Module Format
  - RequireJS
  - Hypertext Transfer Protocol
  - List of HTTP Status Codes
  - Postman

Exercise

1. Create a simple Node script that converts 'www.mim.edu' domain name to the equivalent IP address. (Search and learn 'dns' module, resolve4)
2. Create a web server that's going to send a response of big image (bigger then 3MB) to any client that sends a request to your specified server:port. Use the best way for performance. (Try to solve this in many different ways and inspect the loading time in the browser and send many requests to see the performance differences)
3. Using Node Stream API, create a script to unzip a file (after you zip it). (Use zlib.createGunzip() stream)