Core Node JS course work

Advanced core Node 01.18.2018

L> Prereq - JS

Node != JavaScript !! We will focus on pure Node and node apis.

- Server side framework
- Why is Node so popular ~ full stack JavaScript
- None blocking.

The course will cover :

- The event loop.
- V8 and the call stack.
- Sockets.
- Event Emitters.
- Streams. and Clusters. {for scaling}

lets start by looking at these questions : 01.18.2018

1. What is the relationship between Node and V8 Can node work without V8?
2. How come when you declare a global variable in any nodejs file it is not really global to all modules?
3. When exporting the API of a node module, why can we sometimes use exports and then other times we have to use module.exports
4. Can we require local files without using relative paths?
5. Can different versions of the same package be used in the same application?
6. What is the event loop? is it part of V8?

7. What is the call back stack? 01.18.2018
8. What is the difference between { setImmiate() } and process.nextTick?
9. How do you make an async process return a value?
10. Can callbacks be used with promises or is it one way or the other?

01.18.2018

Course Overview

1. Node != JS, 2. Node's event-drive architecture

3. Node for web 4. Working with Streams

5. Concurrency Model and event loop 6. Node for Networking

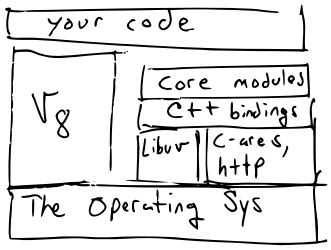
7. Node's Common Built-in Modules 8. Cluster's and Child processes

01.18.2018

1 Node Architecture V8 : Libuv

Node can run on V8 and Chakra

We will cover V8 feature groups and command-line options.



Examine node's relationship with the V8 engine and Libuv

We will also cover Nodes REPL 01.18.2018
i.e useful command line options

We will discuss the global object. The process object and the buffer module.

We will also discuss the ("require") module.

L> the require module entails

require("something")

Resolving, Loading, Wrapping
Evaluating, Caching

We will discuss C++ wrappers, and we will discuss how node wraps every module in a function. And the various ways a require module can be resolved ~ and how the require module can cache every module it uses.

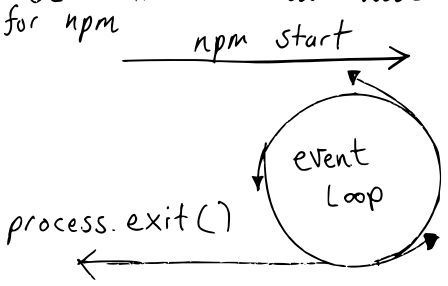
2nd part of the course 01.18.2018
We will discuss node's Concurrency Model and event Loop.

- We will define what IO really means... the different options for IO, e.g.

Synchronous, fork(),
Threads, Event Loop

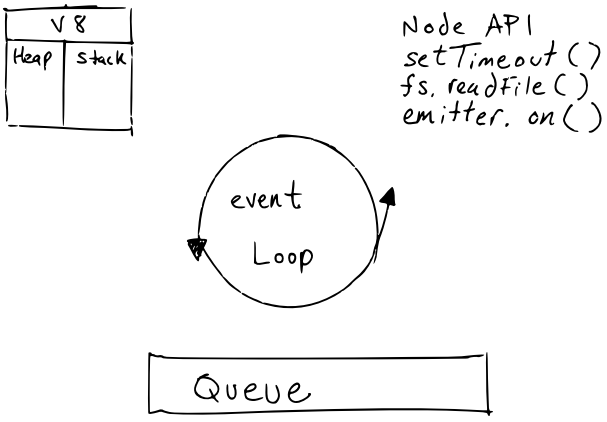
We will discuss the event model as it relates to Nodejs, Libuv, Ruby's Event Machine, Python's Twisted

We will see how Node uses the event loop for npm



01.18.2018

We will discuss the different players that need to be understood to understand the event loop.

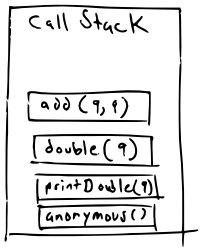


We will be exploring the V8's 01.18.2018

call stack → and see where in the code V8 is executing

e.g.

```
const add = (a, b) => a + b
const double = a =>
    add(a, a);

const printDouble = a => {
  const output = double(a);
  console.log(output)
};
printDouble(9);
```

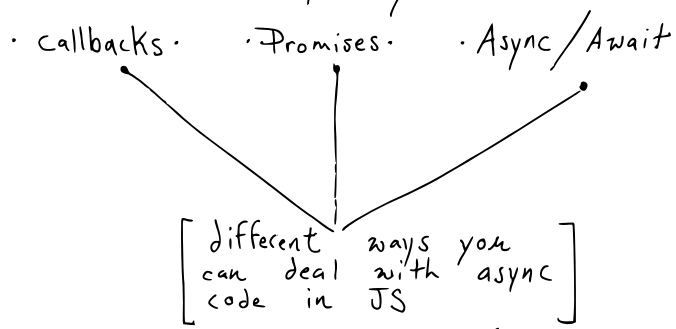


01.19.2018

We will also look at how slow operations affect the single threaded call stack.

We will also explore the event driven nature of node using its core event emitter module

* Asychrony *



* Best of all : Node's error first callback !!! which is Node's standard

01.19.2018

· you can also combine callbacks with promises and then consume these promises using the asyn. wait feature.

· how to use arguments with emitted events.

```
import    ->  const EventEmmiter = require('events');
extend    ->  class Logger extends EventEmmiter {}
init      ->  const logger = new Logger();
emit      ->  logger.emit('event');
addListener -> logger.on('event', listenerFunc);
```

We will take a gander at the EventEmmiter module.

How to handle special error events

· and what happens when we register multiple listeners to the same event

In this course -> we will 01.19.2018

also see how node can work with TCP/UDP protocols.

We will create basic network examples, using the net module. How to communicate with TCP/IP sockets for read and write. Also look at how the sockets are event emitters. We will create a network server and then extend the server to a chat app. {server}

- We will also explore the dns module.

```
e.g.  const dns = require('dns')
      dns.lookup('ghrostfish.io', (err, address)
=>
    {
        console.log(address);
    });
```

01.19.2018

- We will also cover the udp sockets use, using the dgram module.

We will explore 'Node for the Web' Node's http, https support.

We will create a simple http server that is none-blocking. And streaming ready out of the box.

We will also create an https server with a self-signed-certificate will create using OpenSSL Kit.

We will also learn about the five major http module classes

[http Server] [1] — net.Server/EE

[http.IncommingMessage] [2] — ReadableStream/EE

01.19.2018

[http.Agent]3
globalAgent/new Agent()

[http.ClientRequest]5
Writable Stream/EE

[http.ServerResponse]4
Writable Stream/IEE

and identify these five modules in our code!

We will use node for running 01.22.2018
http(s) requests and basic web routing, handle HTML, JSON, and redirects (in the case of 401)

use node's native module to parse and format URLs and query strings.

* We will also cover - Node's common Built-in Modules.

Starting with the os module (one that you used before) e.g. os.freemem(); which can be used to access information directly from the operating system

@here : why not make a node app that detects if you're using win or Linux, 32, or 64 bit

We will explore (cover) the fs module by implementing three practical tasks

- We will create custom console objects i.e. using the util module, and use the util

module directly 01.22.2018

- We will also debug an issue within a script using a built-in debugger, create breakpoints define watchers and use reple to examine any point in the code!!!
- We will cover Working With Streams

★ Streams are Node's best and most misunderstood idea ★ -Dominic Tarr

We will use the Unix philosophy to build simple modular stream, which we can add to. We will do this by chaining them together.

★ Types of Streams ★

Readable	Writable
fs.createReadStream	fs.createWriteStream
Duplex	**Transform**
net.Socket	zlib.createGzip

★ Remember all streams are Event Emitters ★ 01.22.2018

we can consume them using either events or by piping just like in Linux i.e. the pipe.

in linux -> a | b | c | , a is piped to b -> c

in node -> a.pipe(b).pipe(c).pipe(d);

or

```
a.pipe(b);
b.pipe(c);
```

We will look at how to chain stream the pipe function

There are <u>two</u> basic ways of using Streams i.e the <u>implementing</u> and <u>Consuming</u>

There are also two different modes of readable streams

- Pause
  stream.read()
- Flowing
  Event Emitter

stream.resume() →
← stream.pause()

01.22.2018

For Implementing Streams → require('stream')

For Consuming Streams → piping/events

And Finally we will examine

★ Clusters and Child Processes ★

Scalability → in Node is something to think about early/ in the process.

We will discuss the reasons to scale an application

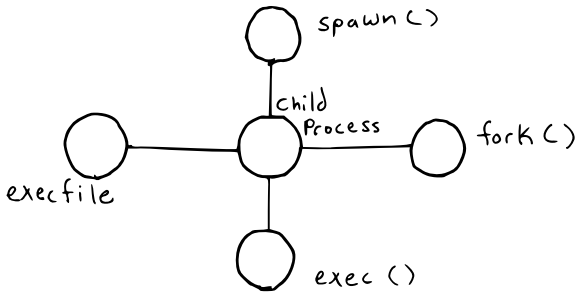
i.e for

Work Load ★ Availability ★ Fault Tolerance

and the 3 different Scalability Strategies

i.e. Cloning, Decomposing, Splitting

01.22.2018



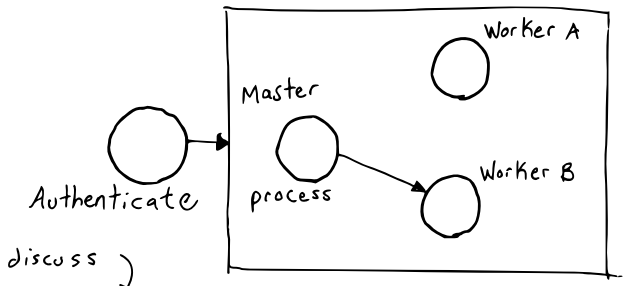
We will examine the different ways we use child processes - starting with the spawn() method. i.e. using the child process module

We will do this using the shell, customize IO objs detching the child process from parent and controlling environment variables.

We will discuss the differences between spawn, exec and execfile and we will fork a process so we don't cause issues with the main process

01.22.2018

We will scale a simple http server, how to broadcast a message from a master process to all forked processes, how to restart failed workers automatically and how to perform zero downtime restarts.



discuss

The problem of stateless communication when working with clusters and Load balancers and how to work around that.

04. Node's Architecture: V8 and libuv 01.22.2018

* Two most important Architecture V8 and libuv, Node uses V8 but will soon work on MS Chakra.

V8 Feature groups →

1. Shipping 2. Staged 3. In Progress

try this command at Repl $ node -p 'process.versions.v8'

these 3 features are included with V8.

1. Shipping comes with the standard node
2. Staging - still in test, use --harmony flag
3. In progress very unstable

01. 22. 2018

To see all the current in progress features at the Repl:

$ node --v8-options | grep "in progress"

To see all the V8 options at Repl

$ node --v8-options | less * Less is more in linux

You can now use 'strict' mode on all code run, instead of having to define strict manually.

You can also disable features that are enabled by default.

e.g. checking to see what options you can control for the garbage collector, use:

$ node --v8-options | grep gc

see the --trace_gc : To get a log line to see everytime the gc runs

01. 22. 2017

$ --expose_gc is useful when you're measuring memory use in your node app. The v8 module could be used for debugging etc.

Node uses V8 with a C++ binding(s)

Node itself has an Api that allows you to interact with Java Script, the Api eventually executes C++ code using function templates. It is not part of V8 itself. Node handles waiting (aysnc) for aysnc events using libuv.

When node is finished with waiting for I/O operations or timers it usually has callback functions, to invoke - Node will pass the control to V8. When V8 is done, it will pass the control back to Node.

Since V8 is single threaded, Node will wait until V8 is done processing its code. This does not depend on how many callbacks have been registered by Node.

This makes programming in Node 01.22.2018
easier. No race conditions, or locks.

★ libuv is a C library that has been developed by node.
It is used by Rust, and others

· libuv is used to abstract the none-blocking I/O operations to a consistent interface accross many operating systems. Libuv handles operations on the file system, TCP/UDP sockets child processes and others. libuv also provides a Thread pool to handle what cannot be done async at the OS level.

★ Most important thing about Libuv ★
Libuv provides Node with the event loop !!!

Node also has a few other dependencies :

These dependencies are 01.22.2018
http-parser, c-ares, OpenSSL, zlib
└> small c library for request/responses, small memory footprint

C-ares to perform async dns queries

OpenSSL - used mainly in tLS and encrypto modules provides many functions for encryption

zlib - fast async, streaming, compression/decompression interfaces.

5. Node's CLI and RepL (01.29.2018)

Very convenient -> you can tab for autocomp-lete
· The autocomplete is very special for global
★ memorize all global top level functions ★

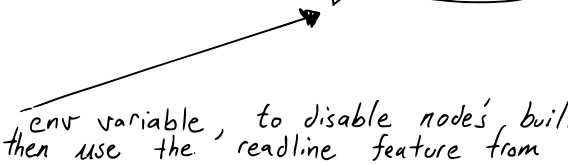
5. cont... 01. 29. 2018

node's global, at the RepL, global Top level functions are available without instantiation, e.g.

> console.log(global) ← will print the actual properties of Global.

Reverse Search is not available in RepL, but you can use = rlwrap (not on Windows)

you will have to install rlwrap {seperate} and run like so $ NODE_NO_READLINE=1 rlwrap node



env variable, to disable node's built in readline and then use the readline feature from rlwrap

The underscore is also very handy. 01. 29. 2018

> _ use it to access the last evaluated value

> Math.random()

> _

> let r = _

> console.log(r);

Node's RepL also has special commands that begin with a dot

> dot tab will print the list to console

e.g.

> .help , dot break will get you out of a multiline session

dot load can allow you to load another script.

01.29.2018
01.30.2018

You can use .save to save RepL session history to file.

There is also an editor with .editor

Node comes with its own RepL module you can use it to define custom sessions

To use the custom RepL function, just require it and invoke it with the repl.start()

You can even customize the RepL to stream with a socket.

You can also control the RepL's global object using context. e.g.

```
r.context.lodash = require('lodash');
```

now lodash will be available in your custom RepL.

To view the options available for node

```
$ node --help | less
```

01.30.2018

- use the -c on command line to evaluate syntax of your node application

-p, evaluates a string and prints the results

e.g. `$ node -p "process.arch"` {which architecture node is running on}

You can also check to see the number of processors on your machine, with

```
$ node -p "os.cpus()"
```

The -r option is used to preload one or modules before executing the script. Use this when loading custom modules without modifying the code itself.

with the -r option you 01.30.2018
can modify the any of the global features,
e.g. timers, loggers etc.

$ node -p "process.argv.slice(1)" test 42

• The list of arguments you supply after
the command line can be used
to supply the process with direct input.

• All arguments can be accessed using the
argv { argument vector }
all values in the argument vector are strings

05 ~fin~

Global Objects, Process, and Buffer

01.30.2018
Global Object, Process, and Buffer

★ The only true global object in node is
called, you guessed it : global.

• When you declare a top level variable, e.g.

var answer = 42; is defined local to the
util module and cannot be accessed even if
you require the util module

You can do this --> global.answer = 42;

But it is better that you do not declare at
the global scope.

★★★ To be a serious Node dev - you must
be able to recognize and use all the global
objects in REPL ★★★

This includes all the JavaScript, top level
functions -- all the functions.

01.30.2018

Some global objects are only available in REPL, like the underscore

Two of the most important objects available in global, is process and buffer

★ Node Process object ★

$ node -p "process" | less

The Node process object provides a "bridge" between a node application and its running environment.

- It has many useful properties -

$ node -p "process.versions" - to determine node version and version numbers of dependencies

01.30.2018

One of the most useful properties of process object is the .env, it provides the copy of the user environment, which is a list of strings you get with the env command in linux and set in Windows

- It is important to remember that this is a copy so if you are modifying any of the process.env variables, you are modifying an alternative form.

★ You should not read from process.env, directly. We usually use config stuff like api keys and passwords from the environment. And also from which ports to listen to.

"Connection Strings," database URI's to connect to, etc.

01.30.2018

You should (as a best practice)
put all of these into a config file or settings module, and always read from that module →

e.g.

```
const { config } = require ('./util');
config.port
```

a good example is to use "process.release.lts" to make sure that you're using a stable release of node like in prod.

Another thing you will be doing much of, is → to communicate with the environment.

And for this we use : stdin, and stdout ~

while stderror for writing to errors. These are pre-established streams, & which cannot be closed.

02.02.2018

The process object is an instance of the event emitter. Meaning we can emit events from process. We can also listen to certain events on the process

* on exit is emitted when node's event loop is done and has nothing else to do.

* or when a manual call to process.exit has been executed.

```
process.on('exit,' (code) => {
  // do one final syn operation, before process termination
});
```

here we can only do synchronous operations {inside this event handler} · you cannot use the event loop. ·

· The uncaught process event is highly misused · which is emitted whenever a js exception is not handled this uncaught process event propagates up to the event loop. { by default, node will print the stack trace and exit }

if we register a handler 02.02.2018

on the uncaught exception, node will not exit.

When the process exits the event handler will be invoked

The buffer class, which is available on the global object, is used heavily in node with binary streams of data. A buffer is essentially a chunk of memory allocated outside of the V8 heap. We can put some data in that memory, and that data can be interpreted in many ways. e.g. depending on a length of a character. Thats why when there is a buffer, there is a character encoding. * Because whatever we place in a buffer does not by default have any character encoding. So to read it we need to specify the encoding.

· Remember when this happened with the node test rig. ·

* When we read content from files or sockets and we don't specify an encoding – we will get back a buffer object.

a buffer is a low level data structure to represent a sequence of binary data. And unlike arrays, once a buffer is allocated it cannot be resized. 02.02.2018

· We can create a buffer in three major ways.

· Buffer.from()

· Buffer.alloc()

and Buffer.allocUnsafe()

e.g. Buffer.alloc(8) ~ will create a buffer of certain size (filled)

while Buffer.allocUnsafe(8) will not fill the created buffer.

To fill an unsafe buffer, use Buffer.allocUnsafe(8).fill()

* Fun with buffer alloc unsafe

> Buffer.allocUnsafe(800) ~ allocates an 800 byte buffer without initialization

To read its content using toString() _ 02. 02. 2018 _ 02. 03. 2018
we will actually see some data, you may even see things that you may recognize.

- you can also create a buffer using the from method

e.g. const string = 'touché';
const buffer = Buffer.from('touché');
console.log(string, string.length);
console.log(buffer, buffer.length);

Buffers are great for reading binary files, streaming images! An image file from a TCP stream.

★ Just like arrays and strings, you can use operations on buffers ~ with some differences like include, index of and slice.

e.g. slice operation on a buffer means that the same memory space is being shared.

it is recommended to use the StringDecoder on binary input. 02. 03. 2018

```
const { StringDecoder } = require('string_decoder');
const decoder = new StringDecoder('utf8');

process.stdn.on('readable', () => {
  const chunk = process.stdn.read();
  if (chunk != null ) {
    const buffer = Buffer.from([chunk]);
    console.log('with .toString function:', buffer.toString());
    console.log('With StringDecoder: function', decoder.write(buffer));
  }
});
```

Require() function and 'stuff' about modules

Modularity is a first class concept in node. You must fully understand how it works in node. 02.04.2018

- There are two core modules in node
- The require function - which is available at the global object. Each module gets its own require function.
- The module ~ module - which is also available on the global object. - And is used to manage all the functions we use with the require function

★ Requiring a module in node is a very simple concept. To execute a require call, Node goes through a few steps.

1. Resolving 2. Loading, 3. Wrapping 4. Evaluating 5. Caching

Resolving - to find the absolute path of a module. 02.04.2018

Loading - determines the content of the file at the resulting path.

Wrapping - gives every module its private scope. and what makes require local to every module.

Evaluating - is what the vm actually does with the code.

Caching - so when you 'require' again, you don't have to go through all the steps (again).

- The module object.

We require a module by loading the content of a file into memory

- Node modules have a one to one relationship with the file system.

```
$ node
> console.log(module);
{ id: 'repl',
  exports:
```

```
        { writer: { [Function: inspect] colors: [Object], styles: [Object] },
          _builtinLibs:
           [ 'assert',
             'buffer',
             'child_process',
             'cluster',
             'crypto',
             'dgram',
             'dns',
             'domain',
             'events',
             'fs',
             'http',
             'https',
             'net',
             'os',
             'path',
             'punycode',
             'querystring',
             'readline',
             'stream',
             'string_decoder',
             'tls',
             'tty',
             'url',
             'util',
             'vm',
             'zlib',
             'smalloc' ],
          REPLServer: { [Function: REPLServer] super_: [Object] },
          start: [Function],
          repl:
           { _domain: [Object],
             useGlobal: true,
             ignoreUndefined: false,
             rli: [Circular],
             eval: [Object],
             inputStream: [Object],
             outputStream: [Object],
             lines: [Object],
             context: [Object],
             _events: [Object],
             bufferedCommand: '',
             _sawReturn: true,
             domain: null,
             _maxListeners: undefined,
             output: [Object],
             input: [Object],
             completer: [Function: complete],
```

```
        _initialPrompt: '> ',
        _prompt: '> ',
        terminal: true,
        line: '',
        cursor: 0,
        history: [Object],
        historyIndex: -1,
        commands: [Object],
        writer: [Function],
        useColors: true,
        prevRows: 0 } },
  parent: undefined,
  filename: 'C:\\git\\NodeJS\\coreNode\\07\\repl',
  loaded: false,
  children: [],
  paths:
   [ 'C:\\git\\NodeJS\\coreNode\\07\\repl\\node_modules',
     'C:\\git\\NodeJS\\coreNode\\07\\node_modules',
     'C:\\git\\NodeJS\\coreNode\\node_modules',
     'C:\\git\\NodeJS\\node_modules',
     'C:\\git\\node_modules',
     'C:\\node_modules' ] }
undefined
```

02.04.2018

Before we can load the content of a file into memory - we need to find the location of the file.

Node searches the node modules directory in the current working path. Node will resolve the module to the local directory.

You can also require files in a directory.

* Circular module dependency is allowed in node. *

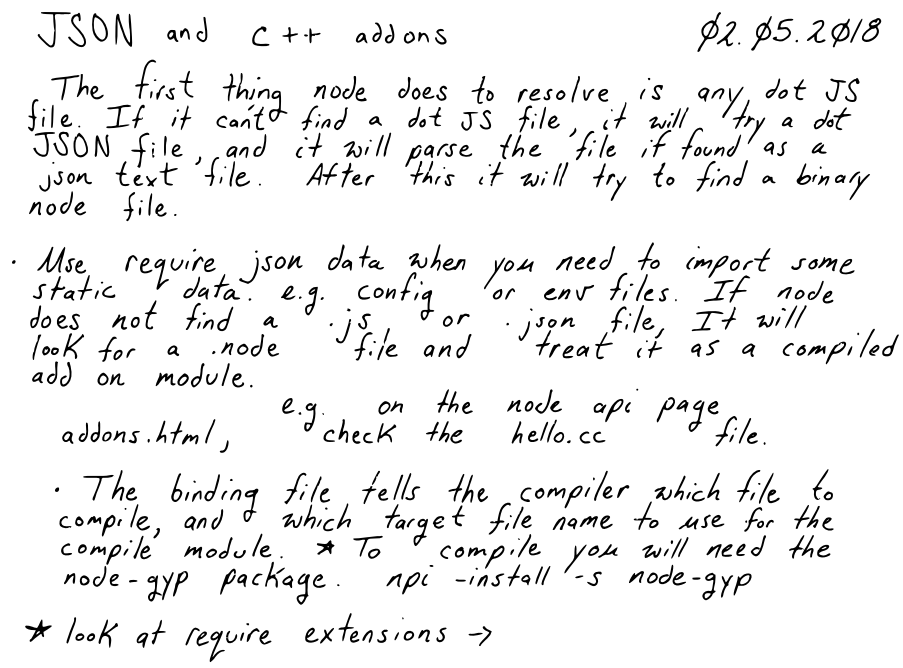
In any module exports is a special object.

Also examine the loaded module ~ which node keeps false.

* you cannot use export objects inside timers.

So what happens when module one requires module two, and module two requires module one?

Addons




```
require.extensions
{ '.js': [Function],
  '.json': [Function],
  '.node': [Function: dlopen] }
```

```
// hello.cc edited by Stan Alam taken from https://nodejs.org/api/addons.html
#include <node.h>

namespace demo {

using v8::FunctionCallbackInfo;
using v8::Isolate;
using v8::Local;
using v8::Object;
using v8::String;
using v8::Value;

void Method(const FunctionCallbackInfo<Value>& args) {
  Isolate* isolate = args.GetIsolate();
  args.GetReturnValue().Set(String::NewFromUtf8(isolate, "Multiverse(s)"));
}

void init(Local<Object> exports) {
  NODE_SET_METHOD(exports, "hello", Method);
```

```
}

NODE_MODULE(NODE_GYP_MODULE_NAME, init)

}  // namespace demo
```

try this at the node REPL 02.05.2018

1. > require.extensions['.js'].toString()
2. > require.extensions['.json'].toString()
3. > require.extensions['.node'].toString()

for 1. dot JS files, node just compiles the content

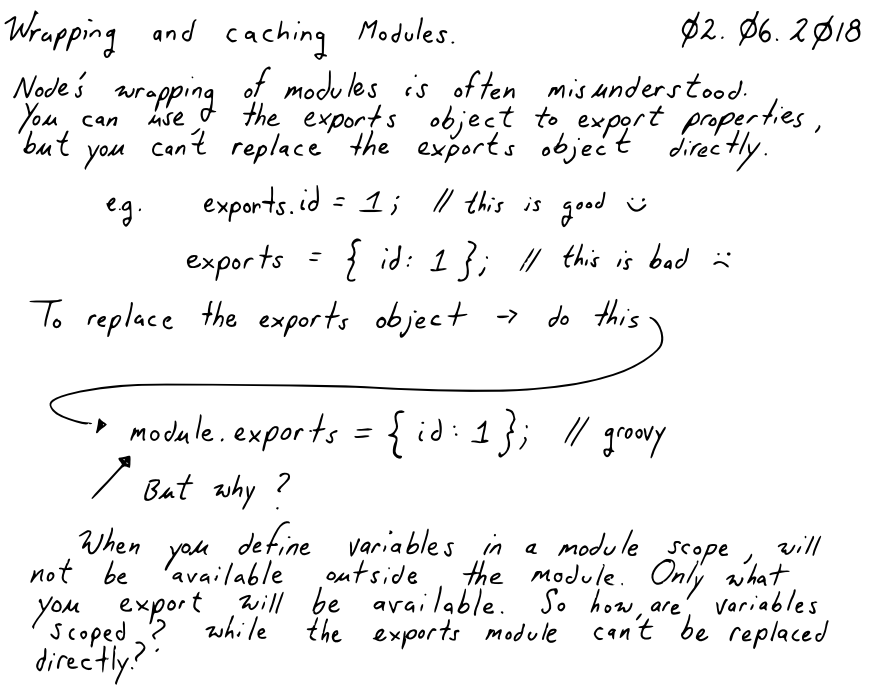
for 2. dot json, node uses json parse (JSON.parse)

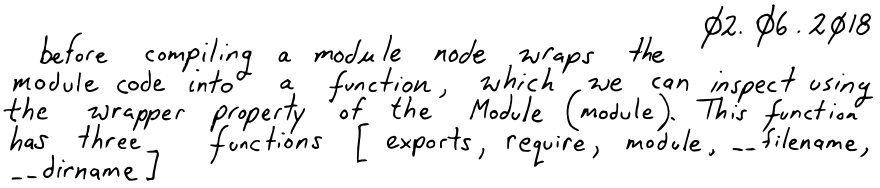
for 3. or for a node file, node uses process.dlopen

~fin~

```
> require.extensions['.json'].toString()
'function (module, filename) {\n  var content = fs.readFileSync(filename,
\'utf8
\');\n  try {\n    module.exports = JSON.parse(stripBOM(content));\n  }
catch (e
rr) {\n    err.message = filename + \': \' + err.message;\n    throw err;\n
}\n
}'
> require.extensions['.node'].toString()
'function dlopen() { [native code] }'
>
```

Wrapping, Caching modules





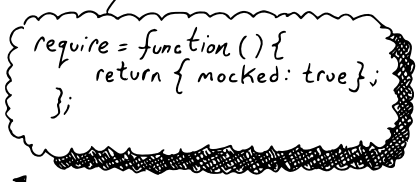

```
> require('module').wrapper
[ '(function (exports, require, module, __filename, __dirname) { ',
  '\n});' ]
>
```

The five arguments of this function * make sure to understand how this function wrapping process works. It is what keeps the top level variables in any module scoped to that module. This is to why the exports require variables appear to look global, when in fact they are specific to each module. This also pertains to the __filename and __dirname variables { which contains the variables absolute filename and directory path. * All these variables are, simply function arguments whose values are provided to the wrap function by node.

* exports is simply a variable reference to module dot exports. It is equivalent to doing this at the top of the module --> let exports = module.exports;

* Make sure to understand how JS object reference works!

There is really nothing special about the require module, it is a function that takes a module or path and returns the exports object. You can override the require function to do your own logic if you wish to. e.g. for testing purposes, we want every require line to be mocked by default. 02.07.2018



this reassignment will suffice
do use this : const fs = require('fs');
console.log(fs);

you can test this by requiring any module. And then console logging that module. The output will print out the mock object by default.

```
const print = (stars, header) => {
  console.log('*'.repeat(stars));
  console.log(header);
  console.log('*'.repeat(stars));
};
// node >= 8
// or use harmony flag
```

02.07.2018

Lets also try using the module with the require. Which can be done by requiring the file and use whatever the file exports as a function and call that function.

e.g.
```
const starChild = require('./starChild');
starChild('42', 'hola!');
```

Now determine if the file is being run as a script or if the file is being required.

```
// use this if condition
if (require.main == module) {
}
```

* When this if statement is reached, i.e when we run this as a script, the condition will be true.

02.07.2018

you can also add an else to see if the script is being required by other files.

In such a case, change the exports object to be the print function.

e.g. ... `print(process.argv[2], process.argv[3]);`

```
} else {
    // being, here required
    module.exports = print;
}
```

This should now work directly from the command line, and when you run a node script that requires your custom function.

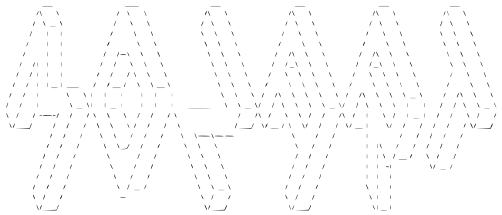
Caching : is very important to understand. 02. 08. 2018

e.g. a simple demonstration > require('./ascii-art');
Let's say you require this ascii art module, which prints an ascii banner. e.g. "Mozart"

If you require the ascii module twice, it will still print once, i.e. the header "Mozart." You can print out the full path of the cache, like so → console.log(require.cache); and you can also delete the whole path. e.g.

delete require.cache['/Users/stan/ascii-art.js']

then node will reload the module twice.
It is more efficient to wrap the log line in a function - and export that function. This way when we require the function, the function will execute the console.log statement.



NPM 02. 08. 2018

"know your npm"

There are a lot of great things you can do with npm.
* first of all npm is not really part of node. It just comes packaged with node. "It's quite popular among the node community. Then there is drum roll... (Yarn) by facebook. facebook claims to be a lot faster than npm. "which is probably true."

* Two important things to consider with npm, i.e. npm cli, and the npm registry.

* The npm cli can work with different registries. npm cli can also be used with git repos directly.

e.g. lets say you wish to install the express module directly. ↳ npm i expressjs/express
↑ the name of the repo

You can few the head of the latest commit using, (npm ls)

If you want to check what package 02.12.2018
npm will install, without actually
installing, use the dry-run command param,

$ npm -i --dry-run ★ This will only report what
will be installed.

$ npm ls -g ★ Will list all top level packages
and their dependencies, which will be a very big tree
to view.

$ npm ls -g --depth=0 ★ This shows only the
first level of the tree.
* you can also use the ll command instead of ls

$ npm ll -g --depth=0 ★ This shows the
description and more
$ npm ls -g --depth=0 --json information about the
package.

★ This will parse the output in JSON
in the case you wish to run from a program.

02.12.2018
If you wish to install all your dependencies locally
you can create your own node_modules folder.
However, just use the package JSON. Which will
instruct npm to manage dependencies in a local node
modules folder. The bare-min that should be included
in package JSON → should be name and version number
using the standard semantic versioning.

★ If you are installing a dependency, you should
document it in package JSON. ★ you can document
it in three main categories

If you use $ npm -i -S { The capital "S" will
be considered a production
dependency. The -D flag will be considered as
a dev dependency. If you use the -O, it will
be considered an optional dependency.
* For optional dependencies, the code will check for
their existence, and only use them if they are installed

You can see how all these "commands" are documented
in the package JSON

use the npm update to update a single package or all packages, if the package name is not provided. There is a version range specified in package JSON * The update command will work according to that. The version range is specified by an operator and version, the operator will be greater than or lesser than. The "=" operator is the default operator if no operator is specified. You can use a star or "x" to represent a whole range for that level. e.g. "request": "4.2Ø.*" Using the tilda works like an x does at the last level, but only when the x is greater than what is specified. * e.g. "request": "~1.2.3..." is the same as 1.2.3x But as long as x is greater than or equal to (3)

ØZ.12.2Ø18

So for the range ~1.2 ... 1.2.x for all x greater than or equal to Ø

• There is also the ^ operator which works well with packages less than 1.Ø. Because it allows changes that do not modify the left most non-Zero digit in the center array?

ØZ.14.2Ø18

To update the npm cli itself, using npm itself -> do this $ npm i npm -g, which will update npm itself.

* To check for outdated packages do this -> $ npm outdated -g

npm has a lot of configurations we can control -> $ npm config list -l

* $ npm config set save true ; is also very important * This will make install always use the "--flag"

Searching the npm registry, use the $ npm search [int]

* npm shrinkwrap => locks down all dependency versions. This way anyone who installs using your package.json -> she/he will be getting all the package where version numbers are frozen.

Some groovy npm tricks ϕ2.14.2ϕ18

↳ like npm home command, $ npm home lodash will open the web page for lodash, there is also npm repo lodash, which will open the repository page of the package

★ If you have packages installed in node modules but are not saved to package.json, you manually installed without an entry to package.json, you can clean it up using the $ npm prune { command }

Also some easter eggs: like $ npm visnup, or xmas

Concurrency Model and Event Loop

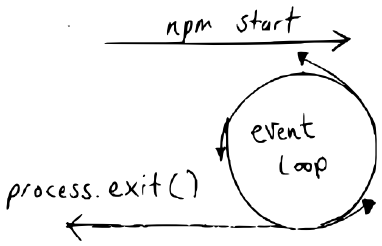
What is I/O, every one knows it ϕ2.14.2ϕ18 means Input, output ... but what does it really mean?

I/O is used to label a communication between a process in a computer CPU and anything external to that CPU, this will include memory, disk, network and even other processes. These processes communicate with signals or messages. When signals are received by the process ~ they are input, outbound messages / signals are output. Almost all operations that occur in a computer are I/O. However, we should be explicit -> when discussing Node's Architecture, in this context, Node use of I/O refers to access of disk and network resources. This is the most time expensive part of all operations. ★ Node's event loop comes from the understanding of this dilemma. Requests can be handled by Sync, or fork() ← but fork does not "scale." ★ Threads, single threaded "framework tech" have less overhead. Apache is multi-threaded, while Nginx is not.

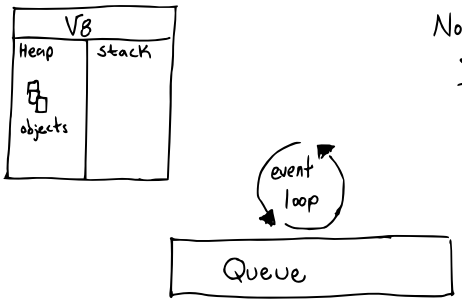
Definition of the event loop 02.14.2018

↳ is an entity that handles external events and converts them into callback invocations.

Here is another definition : A loop that picks events from the event queue and pushes their callbacks to the callstack. ★ To understand the event loop you should understand the data structures that are involved with the event loop.



Again the most important thing to learn is the callback and the concurrency model. Just like Ruby's event Machine, or Python's twisted. Node uses the event loop. The event loop is automatically started when you execute the node script. Node will exit the event loop when all callbacks are exhausted. The chrome browser has a similar mechanism. To understand the event loop. 02.16.2018



so V8 has this thing called stack ... it also has a heap.
the heap is where objects are stored in memory. It is the memory that is allocated by the Vm for "various" tasks.
When you invoke a function an area of the heap is allocated to act as a local scope for that function

Both the stack and the heap are part of the 'runtime engine' * and not node itself * Node provides timers (i.e. Apis) emmiters - wrappers around OS operations. It also provides event queue and the event loop using the libuv library. ϕ2.16.2ϕ18

* Remember that the libuv library provides node with the event loop. * ϕ2.18.2ϕ18 The event loop works between the event Queue and the callstack.

+ The call stack : FiFo : The first element we can pop out of the stack is the first element we have pushed it into it. In the case of the V8 call stack these elements are functions. Since JavaScript is single threaded it can only execute one function at a time. If the stack is executing something on a single thread nothing else will happen on that thread.

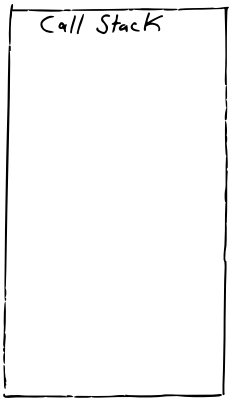
* When we call multiple functions that call each other we naturally create a stack. Then we backtrack the function invocation - all the way back to the first caller. * If you wish to implement recursion, without recursion (recursive) you will need to use the stack. ϕ2.18.2ϕ18 i.e a normal recursive function will use a function anyway.

Let's look at the stack, when we call a function (i.e. look into the stack)

three simple functions

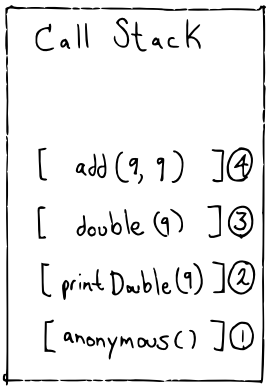
```
const add = (a, b) => a + b;

const double = a =>
    add(a, a);

const printDouble = a => {
    const output = double(a);
    console.log(output);
};
    printDouble(9);
```

printDouble() calls double



* all these functions are wrapped in an immediately invoked function expression.

V8 uses the stack to record where in the program it is executed. Everytime we step into a function it gets pushed into the stack, and every time we return from a function it gets popped out of the stack. 02.18.2018



① Start with the iife (immediately Invoked) function which is an anonymous function. This iife function defines other functions. But only execs printDouble(9) ~▸ print double calls double() ③ so now double gets pushed into the stack.

Double calls add(9, 9) (4), now the add function is pushed onto the stack. * So far all these functions are being executed.

* When the add() function returns with the value, we pop that function off the call stack. It is done. Now return the double function. (double(9)) ③ is popped. Now the execution returns in print double ②

When we execute ② => print double we also execute the console.log (explicitly) which will now return the function (2) and thus popped from the stack. And finally the anonymous iife is also popped out. Notice how that every time a function is added to the stack its arguments and local variables are added to the stack (in that same level) 02.18.2018

. The stack Frame is the definition of the function and its local variables on the stack.

Every time you get an error, the console will show the call-stack. If you try making a function call itself and succeed, you will cause an infinite loop on the call stack. This means you will be continueing to push the function on the call-stack until the V8 engine is out of memory.

* Maximum callstack size, exceeded.

16. ✯ Handling Slow Operations ✯ - as long as the operations executed on the stack is handled fast then Single Threaded is A-ok!!!! But however when we start dealing with slow operations

Handling Slow operations - cont - 02.18.2018

--> will block the execution, I repeat, Slow operations will block the operations. * Guess what you can still block operations in node ... e.g.

```
const slowAdd = (a, b) => {
    for(let i=0; i < 999999999; i++){}
    return a + b;
};
```

like this slow add function. So what happens to the call stack, when we execute a bunch of slow processing functions? It just takes more time. This style of programming goes against Ethos. This is blocking programming. Node's event loop has been created to prevent this style of programming.

So How do Callback's actually work ???

Everyone knows that Node's Api's are designed around callbacks. We pass functions to other functions as arguments. * Then those argument functions get executed at a later time somehow?

Now look at this 02.18.2018

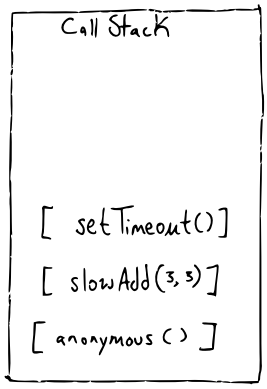
```
const slowAdd = (a, b) => {
    setTimeout(() => {
        console.log(a + b);
    }, 5000);
};

slowAdd(3, 3);
slowAdd(4, 4);
```

here we make sure our function has a set time out, of 5,000 milliseconds that right there is our callback !!!!!!!

① the anonymous() function gets called

② The slowAdd(3, 3) gets called,

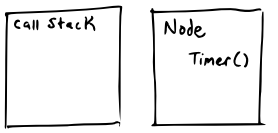
③ The setTimeout() gets called, gets pushed to the stack, but gets popped out of the stack



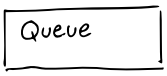
④ now slowAdd(3,3) gets popped out

* The setTimeout gets popped off quickly, because it has no args.

Then somehow console.log(6) gets executed 02.18.2017 thus pushed to the stack to be executed. and after that console.log(8) gets executed and pushed to the stack. The setTimeout is not part of V8. It is provided by node itself, just as it is provided by browsers too. * It is wired in a way to use the event loop asynchronously. * This is to why it behaves more strangely on the call stack.



* Okay so what's this event queue? AKA the message queue. Or the callback queue.



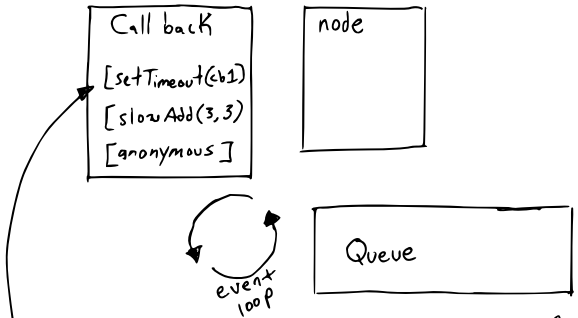
* The event queue is simply a list of things to be processed.

* Let's call these things in the list events.

When we store an event on the queue, we sometimes store a plain old function on it. This function is the good ol callback. Again as a reminder a queue data structure is a FIFO. So this means that the first event that gets queued is the first event to be dequed.

02.18.2018

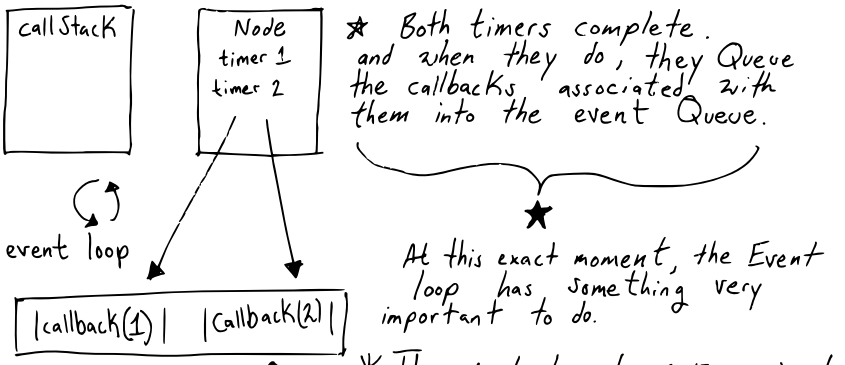
To dequeue and process the event from the event Q we just invoke the function associated with it. (The last event... pushes the function to the stack.



So let's start with the anonymous function (1) it pushes slowAdd(3,3) to the callstack, which in turn pushes setTimeout(callback 1, delay of 5 secs) to the stack

* note the setTimeout() functions are anonymous functions. At this point node sees a call to its setTimeout api, takes note of it, and instantiates a timer outside of the JavaScript Runtime.

The setTimeout function will be called and done; thus popped off the stack. — And while the node timer is running. The stack is free to process its items. It pops slowAdd(3,3) and pushes slowAdd(4,4), which in turn pushes setTimeout(callback 2, delay) Node will now kick off another timer for the new timeout call, and the stack continues to pop its done functions. ϕ2.19.2ϕ18



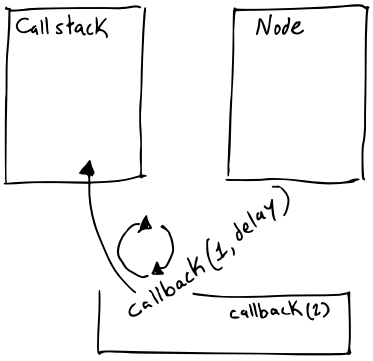
★ Both timers complete. and when they do, they Queue the callbacks associated with them into the event Queue.

★ At this exact moment, the Event loop has something very important to do.

Job to do -> It monitors the call stack and the event Queue. * The event loop has a very simple

When the stack is empty and the queue is not empty. There are events to be processed in the Queue. The event loop will dequeue one event from the queue and push its callback to the stack. ϕ2.19.2ϕ18

★ This is to why it's called the event loop. It loops this logic until the event Queue is empty.



① The stack is empty the event loop "churns" and pushes the callback(1) to the stack.

② Once callback(1) is on the stack, Callback(1) will push the console.log(6) to the stack. Since console.log(6) returns immediately, it gets popped out of the stack. and now the stack is empty.

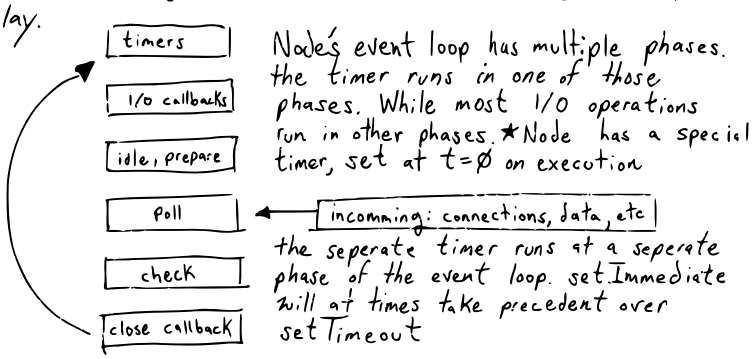
So now the stack is empty. 02.19.2018

This means that the event loop is gonna churn all over again. It pushes the callback 0(2) function to the 0 stack from the queue. Thus callback(2) now pushes console.log(8) to the stack. This is done at once and is returned, thus it is popped off the stack and subsequently the callback is also returned and popped off the stack. This brings the whole machinery, i.e. the callStack, the Event loop along with the event Queue and node to an idle state.

The stack is empty and the queue is empty.

★ Node will now exit the process when we reach this state. ★ Believe it or not, ★ all node processes work this way. i.e. Some process will go handle a certain I/O asynchronously while keeping track of a callback. When it is done it will queue the callback into the event Queue. Remember, this any "slow" code will block the event loop. Also be mindful! → too many events in the queue could undermine the queue. "This important to understand"

18. So what happens when the timer 02.20.2018 delay is 0 seconds, almost the same thing happens. The timers are not executed after 0 seconds but rather after the stack is empty. So if there is a slow operation on the stack, those operations have to wait. This means the delay we have in the timer is not the time we have for execution, ★ but rather a min time to execution. The timer will execute at the min of this delay.



Node's event loop has multiple phases. the timer runs in one of those phases. While most I/O operations run in other phases. ★ Node has a special timer, set at t=0 on execution

the seperate timer runs at a seperate phase of the event loop. setImmediate will at times take precedent over setTimeout

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

fs.readFile(__filename, () => {
  setTimeout() => {
   console.log('execute timeout');
 }, 0); //set timeout is at 0 milliseconds
  setImmediate(() => {
    console.log('execute setImmediate');
  });
});
```

```
/*
$ node setImmediate.js
execute setImmediate
execute setTimeout
$
*/
```

important thing to remember -> 02.21.2018
setImmediate can "top" the zero timeout.

★ When you want something to be executed on the next tick of the event loop → Node has process.nextTick api which is very similar to setImmediate, but node does NOT actually execute its callback on the next tick of the event loop. process.nextTick is not actually part of the event loop. And process.nextTick does not care about the phases of the event loop. Node processes the callbacks registered by the process.nextTick, after the current operation completes, and before the event loop continues. This can be useful but also dangerous! ★ when using process.nextTick recursively.

One good example for using process.nextTick is for making a standard function contract, e.g.

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

function fileSize (fileName, mycallBack) {
  if (typeof fileName !== 'string') {
    return mycallBack(new TypeError('argument should have been a string'));
  }

  fs.stat(fileName, (err, stats) => {
    if(err) {
      return mycallBack(err);
    }

    mycallBack(null, stats.size);
  });
}

fileSize(__filename, (err, size) => {
  if (err) throw err;

  console.log(`Size in KB: ${size/1024}`);
});

console.log('Hello, Multiverse!');
```

02.22.2018

So lets take a look at the standard function contract on line 3 in the nextTick.js script, the file size function receives a file name arg and a callback. It first makes sure that the file name argument is a string. And it executes the callback with an error if the argument is not a string. Then it executes the async function fs.stat on line 8, and it executes the callback with the filesize. i.e. line 17, where the filesize is logged. 17: fileSize(__filename, (err, size)=>

• Now if you execute the script with the filename, it will return "Hello, Multiverse" and the file size. * However * if you replace __filename, on line 17 in the fileSize function with a number or some other character, the script will error out as expected. * But it will not print "Hello, Multiverse!"

• Thats because you have synchronous and async code in nextTick.js.

* * * * This is bad design

→ case in point our old hub-node test rig.

So you how this code ran synch- 02.25.2018 and async ~ * Don't write code with both sync and async operations, now you got that! A function should always be sync or async - no duality. We can fix it like so...

```
function filsize (filename, mycallBack ) {
    if ( typeof fileName !== 'string') {
        return process.nextTick (
            callBack,
                new TypeError ('argument should have been...
... String' );
    }
```

Now the filesize function will always be async

Node's Common Built-in Modules 02.25.2018

The original way node handled async calls, was with callB--acks * This was a long time ago. -- This was before Java Script had native promises support and the async/await feature. Callbacks, again are just functions you can pass to other functions, you can do something like that in C++ using pointers.

Callback !== Async Code ~ callbacks do not indicate async code. * Remember how you used process dot next Tick to handle async operations, written with callbacks. Look at this example...

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

```
const readFileAsArray = function(file, callback) {
    fs.readFile(file, function(err, data) {
        if (err) {
            return callback(error);
        }

        const lines = data.toString().trim().split('\n');
        callback(null, lines);
    });
};

readFileAsArray('./numbers', (err, lines) => {
    if (err) throw err;

    const numbers = lines.map(Number);
    const oddNumbers = numbers.filter(number => number % 2 === 1);
    console.log('odd number count is : ', oddNumbers.length);
});
```

02.28.2018

readFileAsArray, takes a file path and a callback, at fs.readFile(file ....) { or line 4 in src file takes the and reads it. The function then splits the line into an array of strings, and then call the callback with that array. e.g. Say you have a file that has only numbers, parse those numbers into an array. (The numbers in the file are strings) after parsing into an array, count the odd numbers. This is just one example of using read file as an array. This is a good example of node's callback style. The first argument is an error, aka error first argument that's nullable, and we pass the callback as the last argument of the host function ✱ Always do this with your functions because node devs will always assume this. i.e. function(err, data) {
if (err) {
return callback(err);

Modern JavaScript ⇒ has promises ~ which are known as an alternative to callbacks for async apis. Instead of passing callback as an argument and handling the error in the same place. A promise object

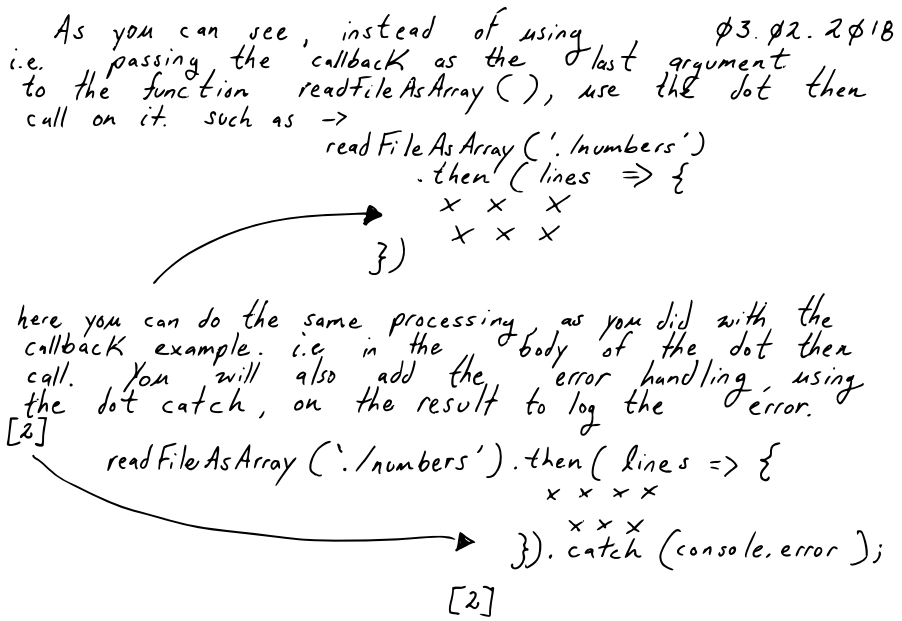
allows the dev to handle success and error cases

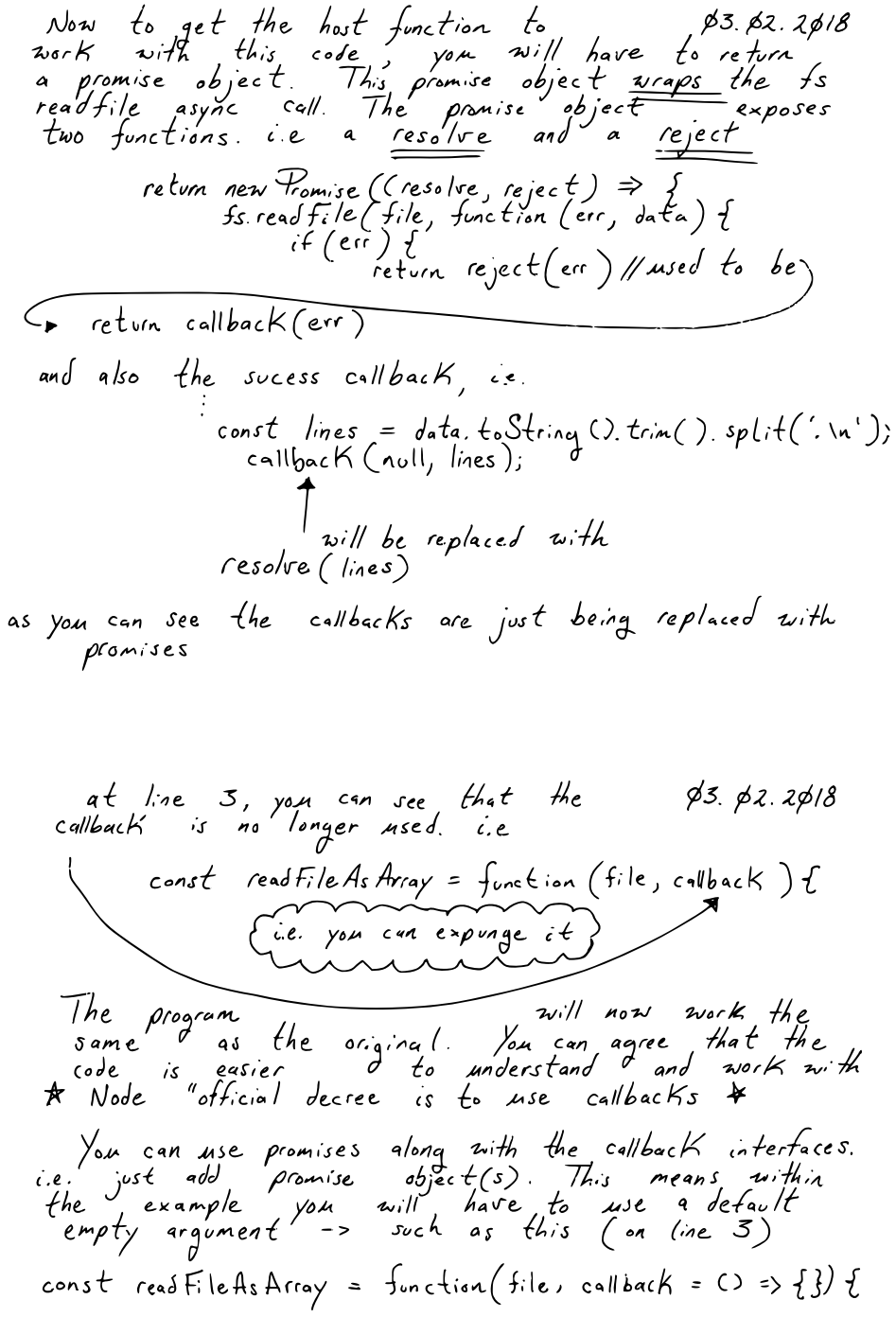
02.28.2018

~ Separately, and allows to chain async calls instead of nesting them as in callbacks.

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

const readFileAsArray = function(file) {
```

```
    return new Promise((resolve, reject) => {
      fs.readFile(file, function(err, data) {
        if(err) {
          return reject(err);
        }

        const lines = data.toString().trim().split('\n');
        resolve(lines);
      });
    });
};

// here's where you make the call
readFileAsArray('./numbers').then(lines => {
  const numbers = lines.map(Number);
  const oddNumbers = numbers.filter(number => number % 2 === 1);
  console.log('odd numbers count:', oddNumbers.length);
})
.catch(console.error);
```






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

const readFileAsArray = function(file, callback = () => {}) { //notice the emp
  return new Promise((resolve, reject) => {
    fs.readFile(file, function(err, data) {
      if(err) {
        reject(err);
        return callback(err); // you can see you're still returning, this time
      }

      const lines = data.toString().trim().split('\n');
```

```
      resolve(lines);
      callback(null, lines);
    });
  });
};

// here's where you make the call
readFileAsArray('./numbers').then(lines => {
  const numbers = lines.map(Number);
  const oddNumbers = numbers.filter(number => number % 2 === 1);
  console.log('odd numbers count:', oddNumbers.length);
})
.catch(console.error);
```

03.02.2018

Adding promises makes code easier to work with - especially when there is a need to loop over async functions. Callbacks can become messy. However, you can name your callback functions, use promises or use function generators.

★★★ Async function ★★★ a new alternative that's hip. It allows you to treat async code in more linear fashion. Making it more readable when you need to process loops. Use the harmony flag.

Now to use the async function begin the function definition with the async keyword before the function name. It is important to understand how await works with async.

But to work with errors, you will have to wrap everything in a try catch block.

To execute this code run like so →

$ node --harmony-async-await filename.js

```
async function countEven() {
//  (%) is the modulus operator, it will let you have the remainder of place/s
try {
  const lines = await readFileAsArray('./numbers');
  const numbers = lines.map(Number);
  const evenCount = numbers.filter(number => number % 2 !== 1).length
  console.log('even number count is : ', evenCount);
} catch(err) {
  console.error(err);
}
  }
}
```

```
countEven(); // wow! async code running synchronously ... good for test automa
```

The Event Emitter 03.03.2018

The event emitter is a module that facilitates communi-cation between objects in node. ✯ Event emitter is at the core of node's async driven architecture ✯ * Many of Node's built in modules inherit from event emitter *

✯ Emitter objects emit named events that cause listeners to be called. ✯ An emitter object has two main features.

① emitting main events
② registering listener functions

① logger.emit('event');

② logger.on('event', listenerFunc);

Ⓐ * To work with eventEmitter, create a class that extends eventEmitter *

Ⓑ Emitter objects is instantiated from eventEmitter classes

Ⓒ At any point in their lifecycle, you can use the emit function to emitt any named event.

✯ Emitting an event is a [signal] 03.03.2018 that some condition has occured. This condition is usually a change of state (state change) in the emitting object. ✯

listener functions are added using the on() method.
e.g.

```
logger.emit('event');
logger.on('event', listenerFunc);
```

Now the listener function will be executed everytime the eventEmitter object emits an associated name event.
e.g.

```
//sync_event.js
const EventEmitter = require('events');

class Logger extends EventEmitter {
  execute(theTask) {
    console.log('State before execution');
    this.emit('Begin');
    theTask();
    this.emit('fin');
    console.log('State after execution');
```

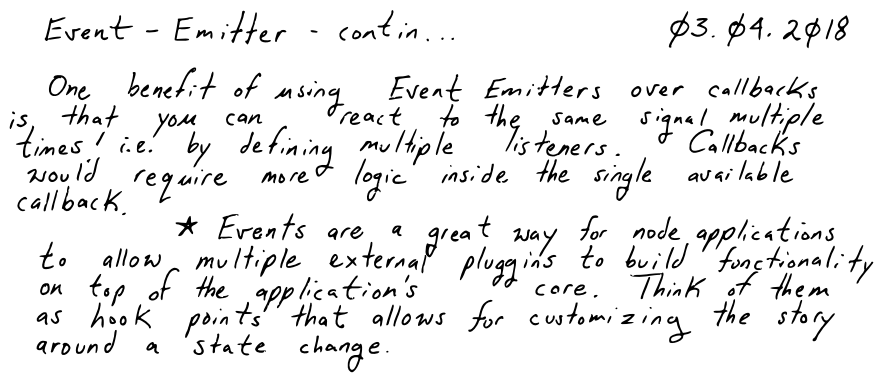
```
  }
}

const logger = new Logger();

logger.on('begin', () => console.log('About to execute'));
logger.on('end', () => console.log('Done with executing'));

logger.execute(() => console.log('***** execute task ******* '));
```

Shell Output

```
$ node sync_event.js
State before execution
***** execute task *******
State after execution
```

Notice that the execution of sync_event.js is synchronous.

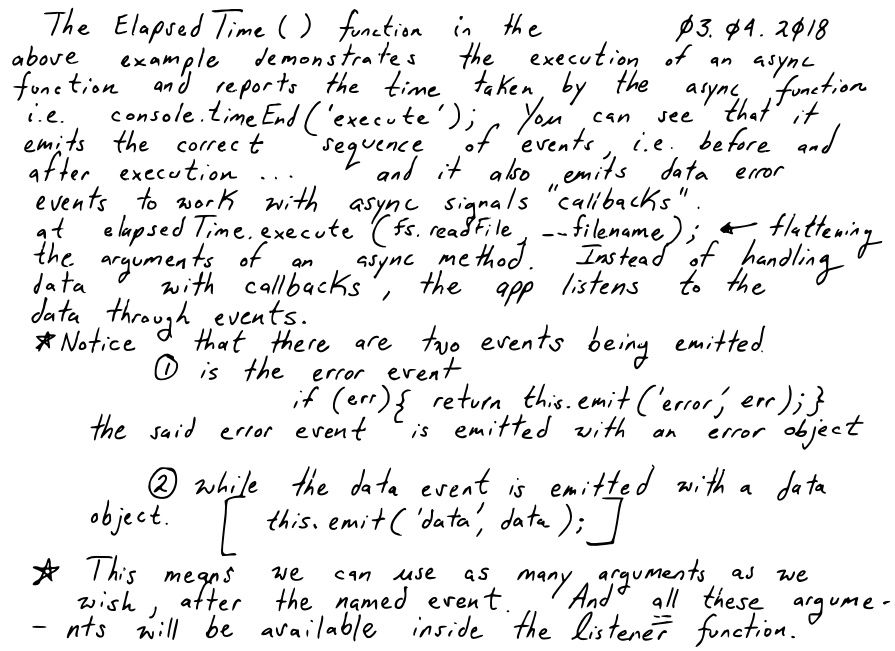
```
//sync_event2.js
const EventEmitter = require('events');

class Logger extends EventEmitter {
  execute(theTask) {
    console.log('State before execution');
    this.emit('Begin');
    theTask();
    this.emit('fin');
    console.log('State after execution');
  }
}

const logger = new Logger();

logger.on('begin', () => console.log('About to execute'));
logger.on('end', () => console.log('Done with executing'));

logger.execute(() => setTimeout(
  () => console.log('***** execute task ******* '),
  1000
  )
);
```

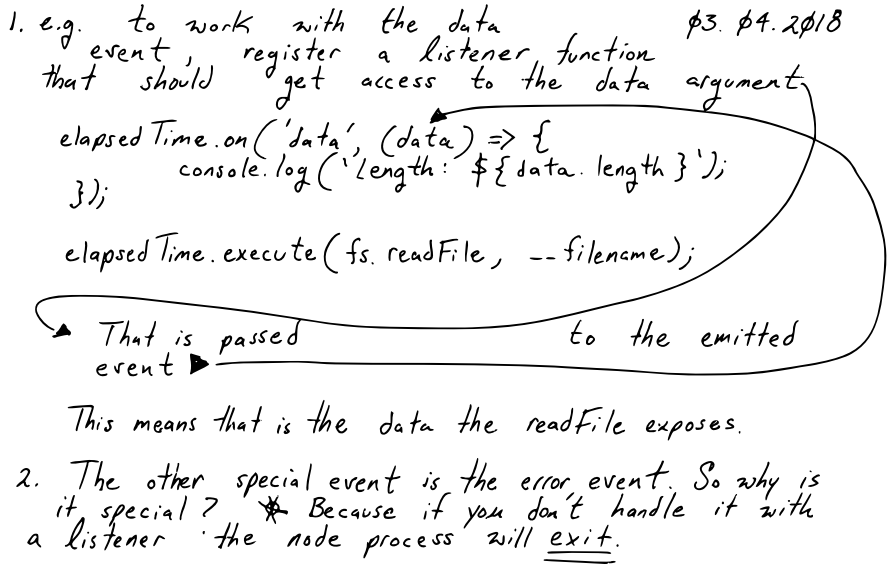



```
const fs = require('fs');
const EventEmitter = require('events');

class ElapsedTime extends EventEmitter {
execute(asyncFunc, ...args) {
  console.time('execute');
  this.emit('begin');
  asyncFunc(...args, (err, data) => {
    if(err) {
      return this.emit('error', err);
    }

    this.emit('data', data);
    console.timeEnd('execute');
    this.emit('end');
  });
  }
}

const elapsedTime = new ElapsedTime();

elapsedTime.on('begin', () => console.log('ready to exec'));
elapsedTime.on('end', () => console.log('completed execution'));

elapsedTime.execute(fs.readFile, __filename);
```

Shell Output

```
$ node async_event.js
ready to exec
execute: 47.151ms
completed execution
```



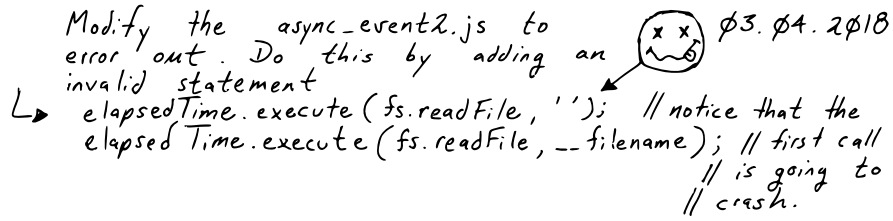



```
const fs = require('fs');
const EventEmitter = require('events');

class ElapsedTime extends EventEmitter {
execute(asyncFunc, ...args) {
  console.time('execute');
  this.emit('begin');
  asyncFunc(...args, (err, data) => {
    if(err) {
      return this.emit('error', err);
    }
```

```
    this.emit('data', data);
    console.timeEnd('execute');
    this.emit('end');
  });
  }
}

const elapsedTime = new ElapsedTime();

elapsedTime.on('data', (data) => {
  console.log(`Length: ${data.length}`);
});
elapsedTime.execute(fs.readFile, __filename);
```

Shell Output

```
$ node async_event2.js
Length: 571
execute: 3.550ms
```




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

class ElapsedTime extends EventEmitter {
execute(asyncFunc, ...args) {
  console.time('execute');
  this.emit('begin');
  asyncFunc(...args, (err, data) => {
    if(err) {
      return this.emit('error', err);
    }

    this.emit('data', data);
    console.timeEnd('execute');
    this.emit('end');
  });
```

```
  }
}

const elapsedTime = new ElapsedTime();

elapsedTime.on('data', (data) => {
  console.log(`Length: ${data.length}`);
});

elapsedTime.execute(fs.readFile, ''); //notice blank in argument, will crash
elapsedTime.execute(fs.readFile, __filename);
```

To prevent the node process from crashing 03.04.2018
and exiting. You will need to register a listener
i.e. for the special error event.

By doing so, the error will be reported and both lines will
be executed. * This will prevent the node Test
framework from crashing and stalling as it did with
A.'s design. *

The other method for handling errors is by listening
to the unhandled process event.

So in the case of the Node-Test-Framework, we
could have inserted a block of code to "clean up"
before the process.exit(1); i.e to exit to the OS.

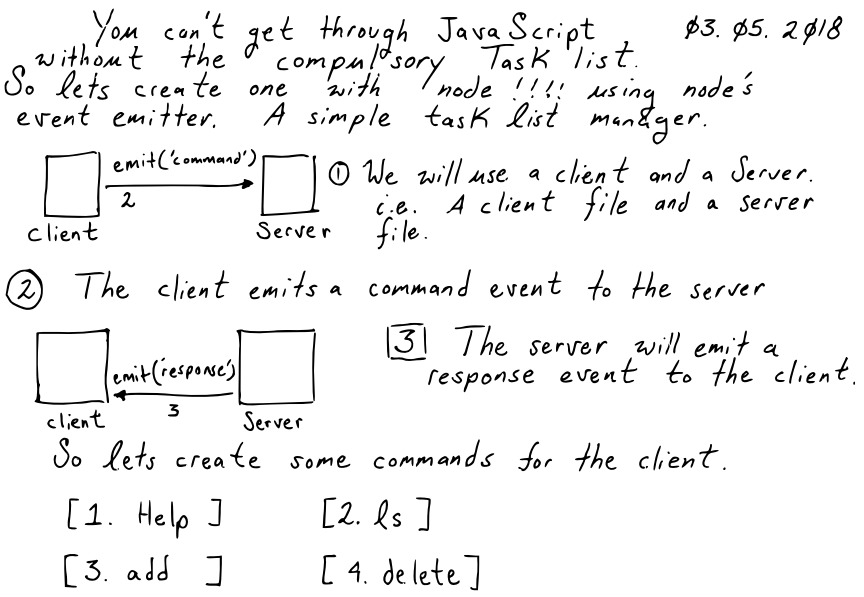
* But what happens when multiple error events occur?
this would mean that the uncaughtException listener
will be triggered multiple times. *

* This could pose a serious problem for the cleanup
code block *
You can mitigate this problem by
using the .once method on the event listener.

If the dev chooses to register 03.04.2018
multiple event listeners for the same event
then the invocations for those listeners should be
in order. + The first listener that is registered will
be the first to be invoked.

The prependListener() method - if you have a need to
define a new listener - but have that listener invoked
first, then just use the prependListener() method

If you wish to remove a listener method, then you
can use the obj.removeListener() method.




```
const EventEmitter = require('events');

class Server extends EventEmitter {
  constructor(client) { //define the constructor to receive the client *within
    super();
  }
}
//created a function not just an object
module.exports = (client) => new Server(client); // Server(client) instantiate
//the function (client) is going to receive the client
```

```
//client
const EventEmitter = require('events');
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin, output:process.stdout
});

//client event emmiter
const client = new EventEmitter(); //instantiate an object directly from Event
const server = require('./server')(client); //import server object
/* the client is going to emit events, while
the server is going to listen to those events
*/
```

```
rl.on('line', (input) => { //here register a listener for the line event, that

});
```

```
//client with latest update to read input from cli and echo input with readlin
const EventEmitter = require('events');
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

//client event emmiter
const client = new EventEmitter(); //instantiate an object directly from Event
const server = require('./server')(client); //import server object, also expor
/* the client is going to emit events, while
the server is going to listen to those events
*/
//use the readline interInterface
rl.on('line', (input) => { //here register a listener for the line event, that
  console.log(input); //log the inpout line, to test the client
});
```

```
//output
$ node client.js
yo
yo
```

```
const EventEmitter = require('events');
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

//client event emmiter
const client = new EventEmitter(); //instantiate an object directly from Event
const server = require('./server')(client); //import server object
/* the client is going to emit events, while
```

```
the server is going to listen to those events
*/
//use the readline interInterface
rl.on('line', (input) => { //here register a listener for the line event, that
  client.emit('command', input);  // now everytime the user presses ENTER,
}); // the client is going to emit an input EVENT to the server
```

The idea is to enter a value such as 'help' which is the command value and the Server will read and respond accordingly.

```
// client is able to clear the terminal when a command is entered
// client echos the command, and the correct command too. from the server
const EventEmitter = require('events');
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

//client event emmiter
const client = new EventEmitter(); //instantiate an object directly from Event
const server = require('./server')(client); //import server object
/* the client is going to emit events, while
the server is going to listen to those events
*/
server.on('response', (resp) => { //when the server emits a response event, th
  //console.log(`Response: ${resp}`);
  process.stdout.write('\u001B[2J\u001B[0;0f'); //clear terminal
  process.stdout.write(resp);
  process.stdout.write('\n\>');
});
//use the readline interInterface
rl.on('line', (input) => { //here register a listener for the line event, that
  client.emit('command', input);  // now everytime the user presses ENTER,
});  // the client is going to emit an input EVENT to the server
```

Server is able to take four commands and echo invalid commands

```
//server is able to handle four commands, and update accordingly, also can han
const EventEmitter = require('events');

class Server extends EventEmitter {
```

```
  constructor(client) { //define the constructor to receive the client *within
    super();
    client.on('command', (command) => {
      console.log(`Command: ${command}`);
      switch(command) {
        case 'help':
        case 'add':
        case 'del':
        case 'ls':
        this[command]();
        break;
      default: // case in which is unknown command
        this.emit('response', 'unknown command:' + `${command}`);
      }
      // help, add, del, ls
    });
  }
    help() { //create an instance method for each command
      this.emit('response', 'help is on the way ...');
    }
    add() {
      this.emit('response', 'adding ...');
      }
   del() {
     this.emit('response', 'deleting ...');
   }
   ls(){
     this.emit('response', 'ls-Ing...');
   }
}
//created a function not just an object
module.exports = (client) => new Server(client); // Server(client) instantiate
//the function (client) is going to receive the client
```

```
//terminal output
$ node client.js
help is on the way ...
>
```

Server

```
//complete server.js
const EventEmitter = require('events');
```

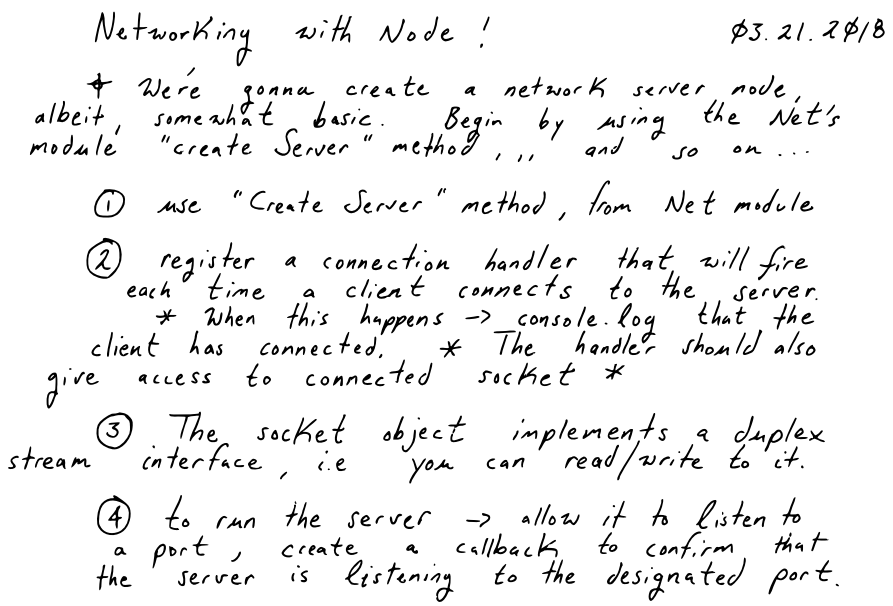
```
class Server extends EventEmitter {
    constructor(client) { //define the constructor to receive the client *with
        super();
        //this.emit('response', 'Welcome -- enter a command or help for list o
        /* this.emit does not work because it will be executed when
        const server = require(./server)(client) in client.js will be executed
         This is the sequential order, so the handler of the reponse event is
         */
        this.tasks = {}; //task object to hold task info
        this.taskId = 1; //task Id to keep track of number tasks starting at 1
        process.nextTick(() => {
            this.emit('response', 'Welcome - type a command or enter help for
        });
        client.on('command', (command, args) => { //now server can accept a co
            console.log(`Command: ${command}`);
            switch (command) {
                case 'help':
                case 'add':
                case 'del':
                case 'ls':
                    this[command](args); // NOW THE ARRAY OF ARGS CAN BE PASSE
                    break;
                default: // case in which is unknown command
                    this.emit('response', 'unknown command: ' + `${command}`);
            }
            // help, add, del, ls
        });
    }

    tasksString() {
      return Object.keys(this.tasks).map(key => { //loop over key, return task
        return `${key}: ${this.tasks[key]}`;
      }).join('\n');
    }

    help() { //create an instance method for each command
        this.emit('response', `Available commands:
        add, del {id}, ls`);
    }
    add(args) {
        this.tasks[this.taskId] = args.join(' ');  // each task is now added t
        this.emit('response', `Added task ${this.taskId}`); //emit the number
        this.taskId++;
    }
    del() {
       delete(this.tasks[args[0]]); //for deleting the task, delete the id ind
       this.emit('response', `Deleting task ${args[0]}`); //print to console t
    }
```

```
    ls() {
        this.emit('response', `List of Tasks\n${this.tasksString()}`);
    }
}
//created a function not just an object
module.exports = (client) => new Server(client); // Server(client) instantiate
//the function (client) is going to receive the client
```

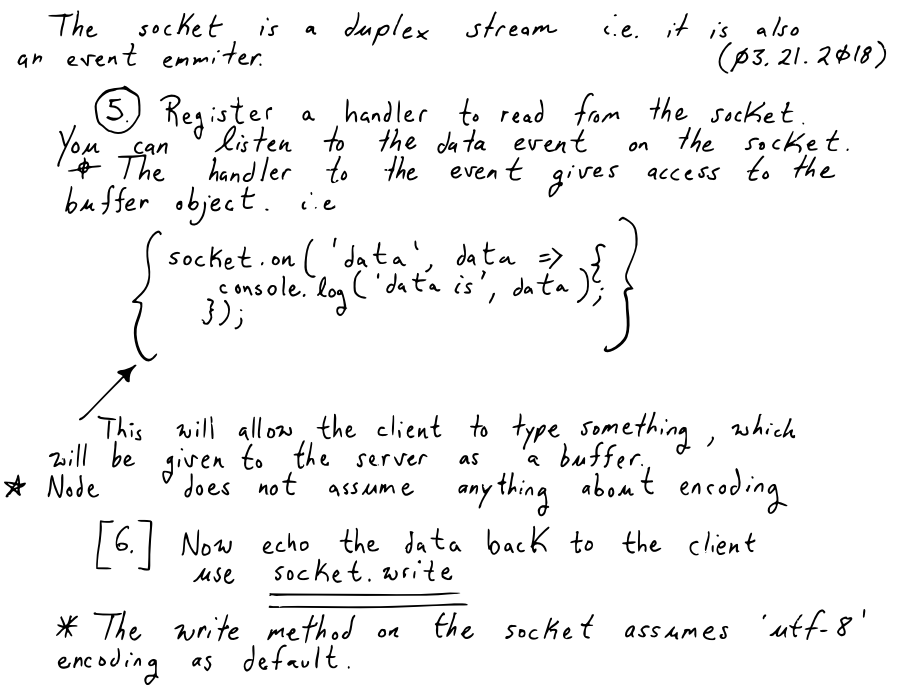
Client

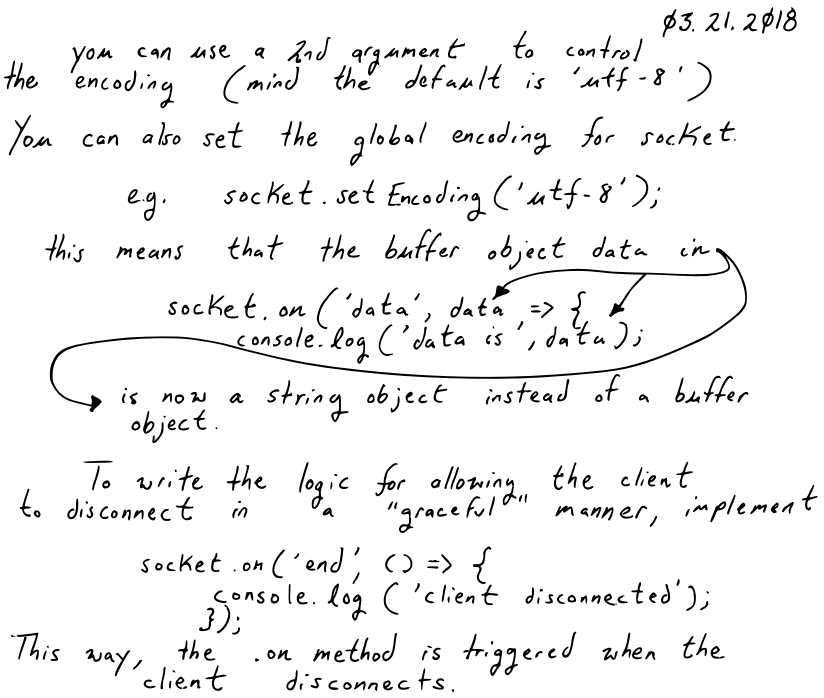
```
//complete client
const EventEmitter = require('events');
const readline = require('readline');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

//client event emmiter
const client = new EventEmitter(); //instantiate an object directly from Event
const server = require('./server')(client); //import server object
/* the client is going to emit events, while
the server is going to listen to those events
*/
server.on('response', (resp) => { //when the server emits a response event, th
    //console.log(`Response: ${resp}`);
    process.stdout.write('\u001B[2J\u001B[0;0f'); //clear terminal
    process.stdout.write(resp);
    process.stdout.write('\n\>');
});
//use the readline interInterface
let command, args;
rl.on('line', (input) => { //here register a listener for the line event, that
  [command, ...args] = input.split(' '); //split on space, the first "token" w
  client.emit('command', command, args); //pass both command and argument, it
}); // the client is going to emit an input EVENT to the server
```

Node Networking




```
process.stdout.write('\u001B[2J\u001B[0;0f'); //again clear node terminal

const server = require('net').createServer(); // here use the createServer met

server.on('connection', socket => {
  console.log('Client is connected!');
  socket.write('Welcome!');
});

server.listen(3000, () => console.log('I am Server'));
```






```
process.stdout.write('\u001B[2J\u001B[0;0f'); //again clear node terminal

const server = require('net').createServer(); // here use the createServer met

server.on('connection', socket => {
  console.log('Client is connected!');
  socket.write('Welcome!\n');
```

```
  socket.on('data', data => {
    console.log('data is', data);
    socket.write('data is ');
    socket.write(data);
  });

  socket.on('end', () => {
    console.log('Client is now disconnected');
  });
});

server.listen(3000, () => console.log('I am Server'));
```

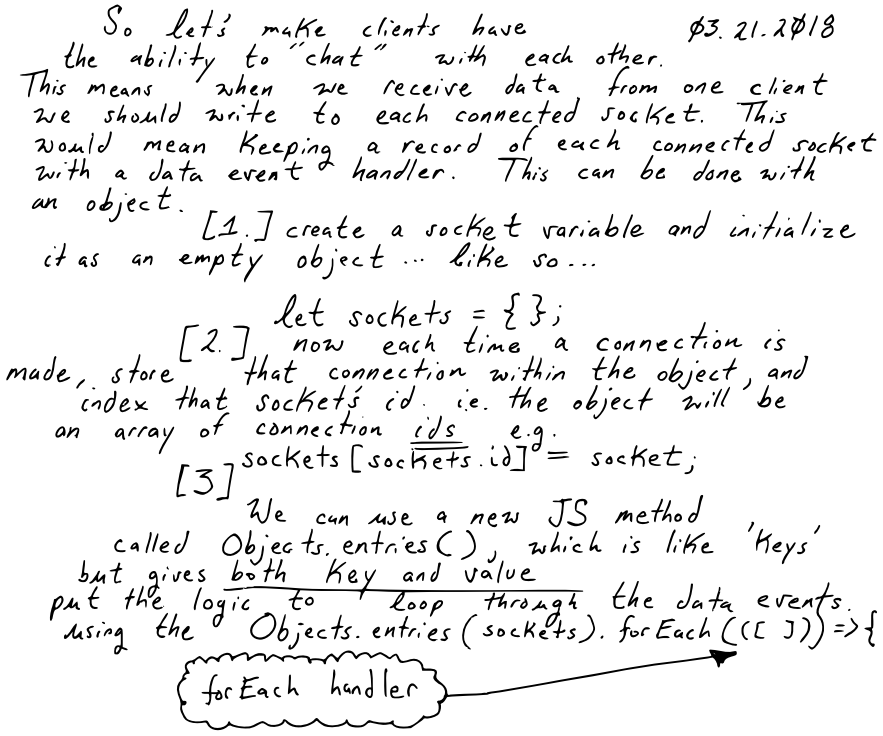
03.21.2018

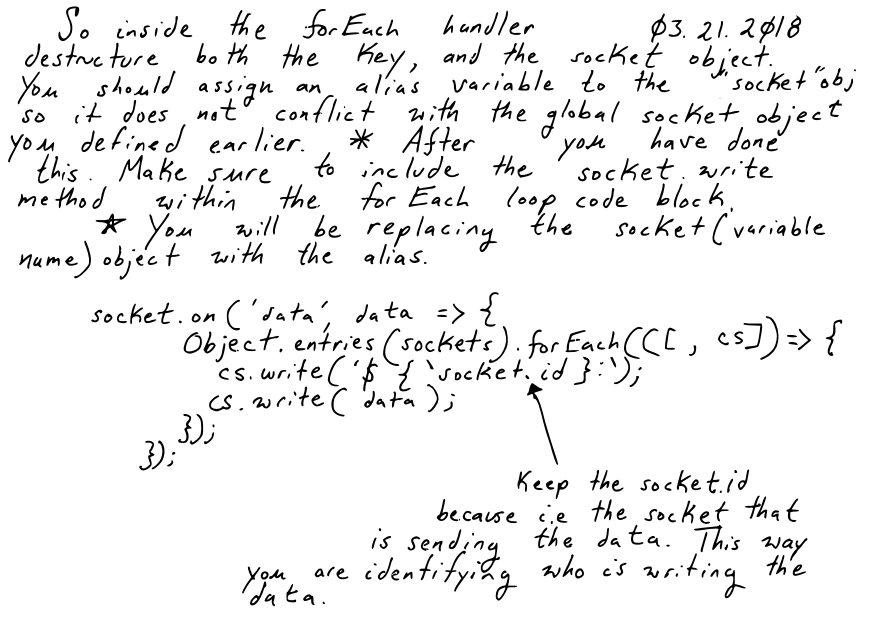
The server object created by the net module is an event emitter. The server object emits a connection event any time a client connects to the server, i.e by how we registered a handler on the connection event, thus the handler gives the client access to the connected socket.
☆ The socket (connected), is, also an event emitter, here it is emitting a data event i.e at [socket.on('data', data => {] whenever the end user types into, the connection. The connected socket also emits an 'end' event when client exists. This will close the connection. ☆

✳ At this point point more than 1 client can to the server. Each client connection will have its own connection socket.
So why don't we assign an id to each socket connection at the Server ?

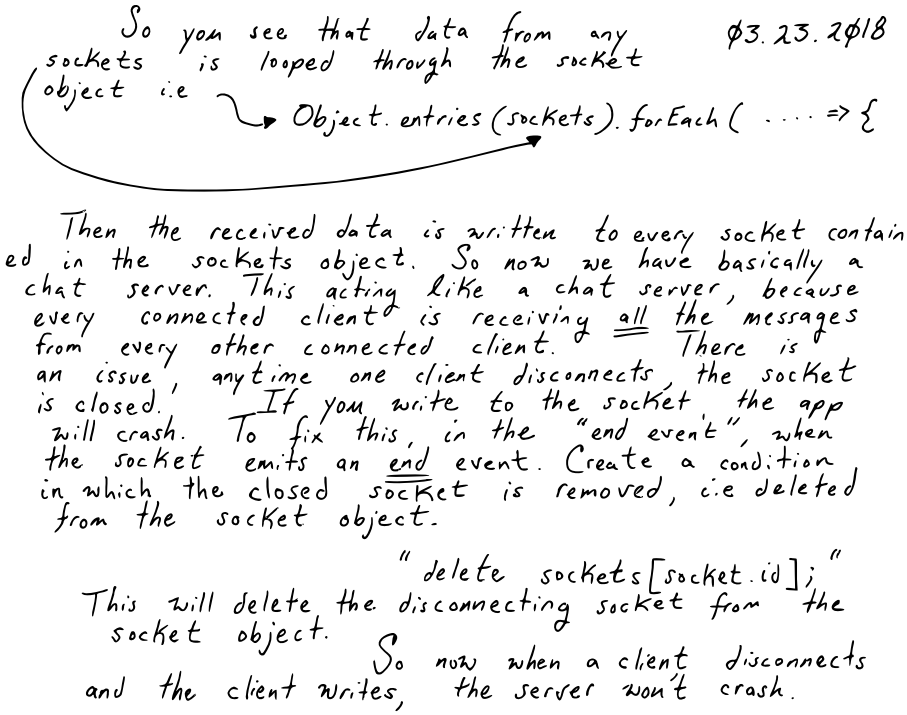
```
process.stdout.write('\u001B[2J\u001B[0;0f'); //again clear node terminal
let counter = 0;
const server = require('net').createServer(); // here use the createServer met

server.on('connection', socket => {
  socket.id = counter++; //everytime a client connects assign an id, and add t
  console.log('Client is connected!');
  socket.write('Welcome!\n');

  socket.on('data', data => {
    socket.write(`${socket.id}: `);
    console.log('data is', data);
    socket.write('data is ');
    socket.write(data);
  });
```

```
  socket.on('end', () => {
    console.log('Client is now disconnected');
  });
});

server.listen(3000, () => console.log('I am Server'));
// use tmux to connect with multiple clients -- i.e. fingers should
//never leave the keyboard!
```





```
process.stdout.write('\u001B[2J\u001B[0;0f'); //again clear node terminal
const server = require('net').createServer(); // here use the createServer met
let counter = 0;
let sockets = {};

server.on('connection', socket => {
  socket.id = counter++; //everytime a client connects assign an id, and add t
  sockets[sockets.id] = socket;

  console.log('Client is connected!');
  socket.write('Welcome!\n');

  socket.on('data', data => {
   Object.entries(sockets).forEach(([, cs]) => {  //the empty value would be k
    cs.write(`${socket.id}: `);
  //  console.log('data is', data);
    cs.write(data);
  });
});
  socket.on('end', () => {
    console.log('Client is now disconnected');
  });
});

server.listen(3000, () => console.log('I am Server'));
// use tmux to connect with multiple clients -- i.e. fingers should
//never leave the keyboard!
```




```
process.stdout.write('\u001B[2J\u001B[0;0f'); //again clear node terminal
const server = require('net').createServer(); // here use the createServer met
let counter = 0;
let sockets = {};

server.on('connection', socket => {
  socket.id = counter++; //everytime a client connects assign an id, and add t
  sockets[sockets.id] = socket;

  console.log('Client is connected!');
  socket.write('Welcome!\n');

  socket.on('data', data => {
   Object.entries(sockets).forEach(([key, cs]) => {  //the empty value would b
    if (socket.id == key)
      return;
    cs.write(`${socket.id}: `);
  //  console.log('data is', data);
    cs.write(data);
  });
});

  socket.on('end', () => {
    delete sockets[socket.id];
    console.log('Client is now disconnected');
  });
});
```

```
server.listen(3000, () => console.log('I am Server'));
// use tmux to connect with multiple clients -- i.e. fingers should
//never leave the keyboard!
```

Node's DNS module 03.24.2018

It can be used to translate network name to address e.g. the lookup method to lookup a host.

```
dns.lookup('nannerl.io', (err, address) => {
    console.log(address);
});
```

* The lookup method on the DNS module does not perform any network communication, but instead uses the underlying OS facilities to perform network operations. *(lookup) This means it will be using libuv threads. All other DNS methods does NOT work with libuv threads but rather uses the Network directly. e.g. the equivalent network method for the lookup method is

```
dns.resolve4('nannerl.io', (err, address) => {
    console.log(address);
});
```

"4 is the family of IP addresses"

* This will be returning an array of addresses, i.e in case the domain has multiple 'A' records.

using the dns.resolve (without '4') 03.24.2018

the default 2nd arg is 'A' record. e.g. you wish to use 'MX'

[1]
```
dns.resolve('nannerl.io', 'MX', (err, address) => {
    console.log(address);
});
```

OR instead all the equivalent types such as 'MX' or 'A' have an equivalent method name. e.g.

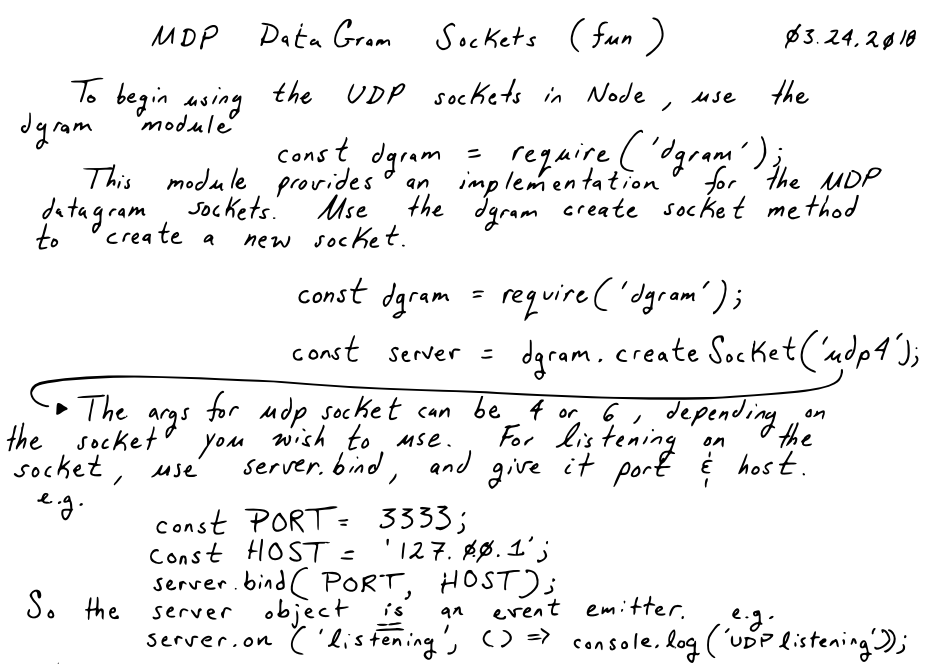
[2]
```
dns.resolveMx('nannerl.io', (err, address) => {
    console.log(address);
});
```

[2] is the same as [1]

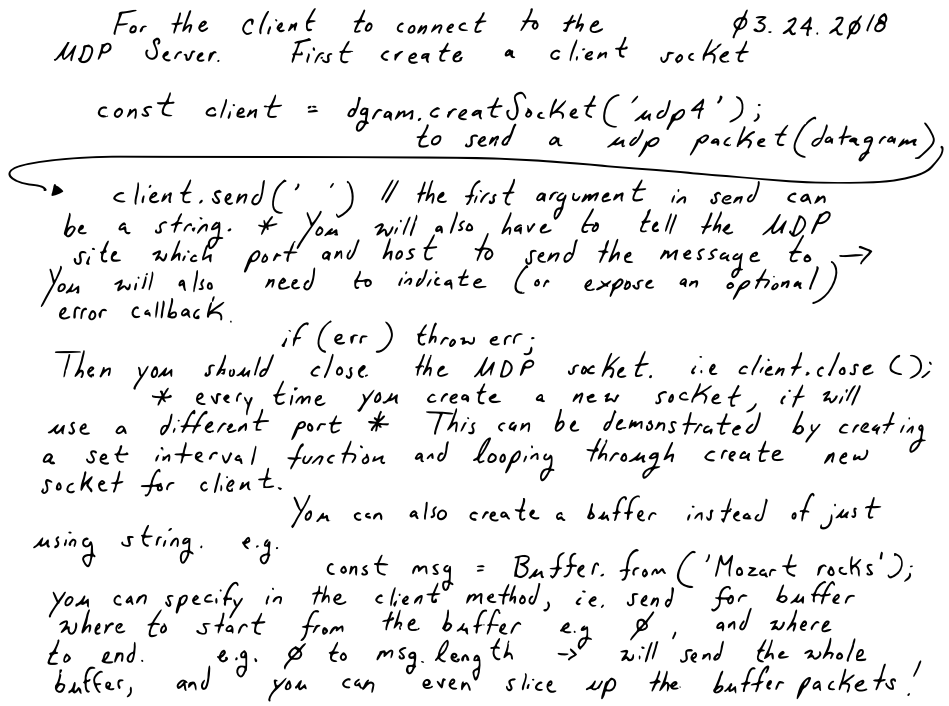
* The reverse method *

The reverse method takes an IP and translates it back to its hostname -> e.g.

```
dns.reverse('x.x.x.x', (err, hostname) => {
    console.log(hostnames);
});
```




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

const server = dgram.createSocket('udp4');

server.on('listening', () => console.log('UDP Server is listening up and liste

server.on('message', (msg, rinfo) => { // register handle for message event, c
  console.log(`${rinfo.address}:${rinfo.port} - ${msg}`); // remote address an
});

const PORT = 3333;
const HOST = '127.0.0.1'; //home
server.bind(PORT, HOST);
```




```
const dgram = require('dgram');
const PORT = 3333;
const HOST = '127.0.0.1'; //home

//server
const server = dgram.createSocket('udp4');

server.on('listening', () => console.log('UDP Server is listening up and liste

server.on('message', (msg, rinfo) => { // register handle for message event, c
  console.log(`${rinfo.address}:${rinfo.port} - ${msg}`); // remote address an
});


server.bind(PORT, HOST);

//clients
setInterval(function () {
  const client = dgram.createSocket('udp4');

  client.send('Mozart is awesome!', PORT, HOST, (err) => {
    if (err) throw err;

    console.log('message sent by UDP');
    client.close();
```

```
  });
}, 1000); // 1 second interval
```

```
$ node stanUDP.js
UDP Server is up and listening!
message sent by UDP
127.0.0.1:61164 - Mozart is awesome!
message sent by UDP
127.0.0.1:52090 - Mozart is awesome!
message sent by UDP
127.0.0.1:52091 - Mozart is awesome!
message sent by UDP
127.0.0.1:52092 - Mozart is awesome!
```

Here we're using sending a buffer instead of string

```
const dgram = require('dgram');
const PORT = 3333;
const HOST = '127.0.0.1'; //home


//server
const server = dgram.createSocket('udp4');

server.on('listening', () => console.log('UDP Server is up and listening! '));

server.on('message', (msg, rinfo) => { // register handle for message event, c
  console.log(`${rinfo.address}:${rinfo.port} - ${msg}`); // remote address an
});


server.bind(PORT, HOST);

//clients
setInterval(function () {
  const client = dgram.createSocket('udp4');
  const msg = Buffer.from('Mozart ROCKS!!!')

  client.send(msg, 0, msg.length, PORT, HOST, (err) => {
    if (err) throw err;

    console.log('message sent by UDP');
    client.close();
```

```
  });
}, 1000);
```

Output is the same

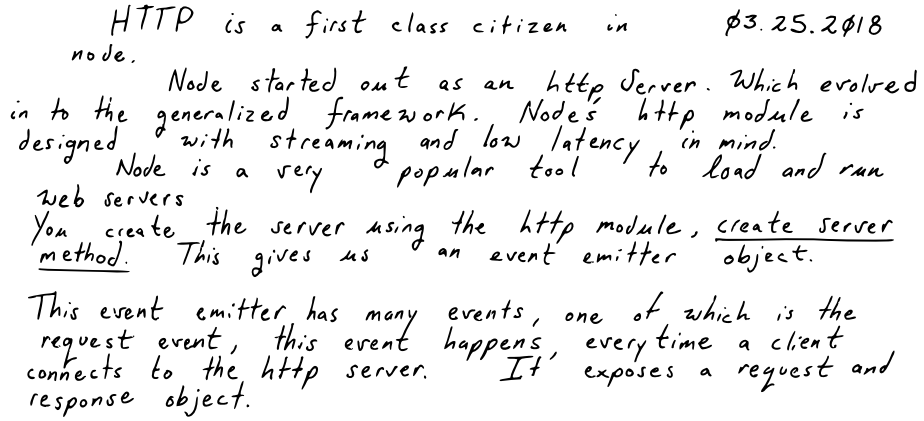
```
$ node stanUDPbuff.js
UDP Server is up and listening!
message sent by UDP
127.0.0.1:58048 - Mozart ROCKS!!!
message sent by UDP
127.0.0.1:58049 - Mozart ROCKS!!!
message sent by UDP
127.0.0.1:56642 - Mozart ROCKS!!!
message sent by UDP
127.0.0.1:56643 - Mozart ROCKS!!!
message sent by UDP
127.0.0.1:56644 - Mozart ROCKS!!!
```

Here we specify an offset, you can even send an array of messages as 1st argument

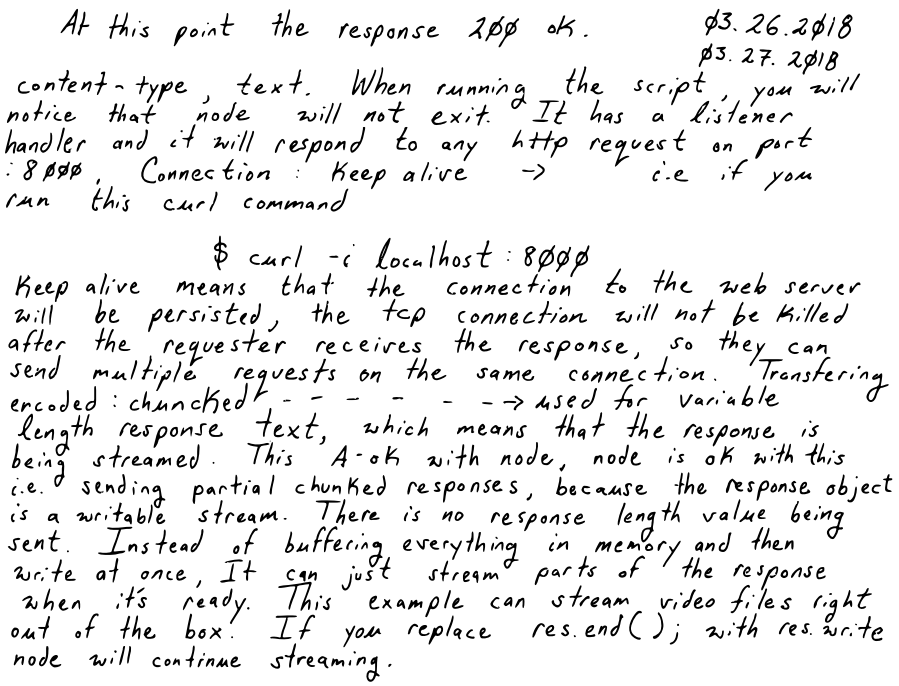
```
const dgram = require('dgram');
const PORT = 3333;
const HOST = '127.0.0.1'; //home


//server
const server = dgram.createSocket('udp4');

server.on('listening', () => console.log('UDP Server is up and listening!'));

server.on('message', (msg, rinfo) => { // register handle for message event, c
  console.log(`${rinfo.address}:${rinfo.port} - ${msg}`); // remote address an
});


server.bind(PORT, HOST);

//clients
setInterval(function () {
  const client = dgram.createSocket('udp4');
  const msg = Buffer.from('Mozart ROCKS!!!')
```

```
  client.send(msg, 0, 7,  PORT, HOST, (err) => {
    if (err) throw err;

  client.send(msg, 7, 6, PORT, HOST, (err) => {
      if (err) throw err;

    console.log('message sent by UDP');
    client.close();
  });
});
}, 1000);
```

```
UDP Server is up and listening!
message sent by UDP
127.0.0.1:51492 - Mozart
message sent by UDP
127.0.0.1:51494 - Mozart
127.0.0.1:51494 - ROCKS!
message sent by UDP
127.0.0.1:51495 - Mozart
127.0.0.1:51495 - ROCKS!
```




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

server.on('request', (req, res) => {
  res.writeHead(200, { 'content-type': 'text/plain' });
  res.end('Hola, Multiverse(s)!\n');
});

server.listen(8000);
```




```
//stanHTTP_2.js
const server = require('http').createServer();

server.on('request', (req, res) => {
  res.writeHead(200, { 'content-type': 'text/plain' });
  res.write('Hola, Multiverse(s)!\n');

setTimeout(function () {
  res.write('Mozart rocks!\n');
  }, 1000);

setTimeout(function () {
  res.write('Beethoven rules!\n');
  }, 1000);
});

server.listen(8000);
```

The server will continue to stream, unless node's http response object is terminated. If you extend the timeout function to a longer delay → e.g. 20 - 30 seconds, the server is not in a sleep state → rather it is idling. It can handle other requests via the event loop. i.e. during this idling phase. More than one request can be handled concurrently with the same node process. * Terminating the response object with a call to the end method is not optional. You must do this for every request. * If you do not call the end method the response obj will time out at the default of 2 min. To control the time out, use the .timeout() function from the http module. e.g. server.timeout = 1000, would timeout in 1 second. 03.27.2018

★ HTTPS ★

Https is the http protocol over TLS - SSL, https is handled by node's https module. Just require the https module and provide an options object to the create Server method. e.g.

```
const server = require('https').createServer({
    key:
    cert:
});
```

In the options object for https, you may combine key, cert with pbx. 03.27.2018

Run the following:

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -nodes
```

The command should create two files in your working directory. Use the fs module

```
key: fs.readFileSync('./key.pem');
cert: fs.readFileSync('./cert.pem');
```

The readFileSync method, indicates that the file is to be read once. And also change the port to default https port (443)

```
const fs = require('fs');
const server = require('https')
  .createServer({
  key: fs.readFileSync('./key.pem'), //readFileSync for reading file only once
  cert: fs.readFileSync('./cert.pem'),
});
//use openssl toolkit
server.on('request', (req, res) => {
  res.writeHead(200, { 'content-type': 'text/plain' });
  res.write('Hola, Multiverse(s)!\n');
});

server.listen(443);
```

```
//openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -nodes
```

Using node as a client for requesting http and http data 03.28.2018

There are 5 main classes in Node HTTP module.

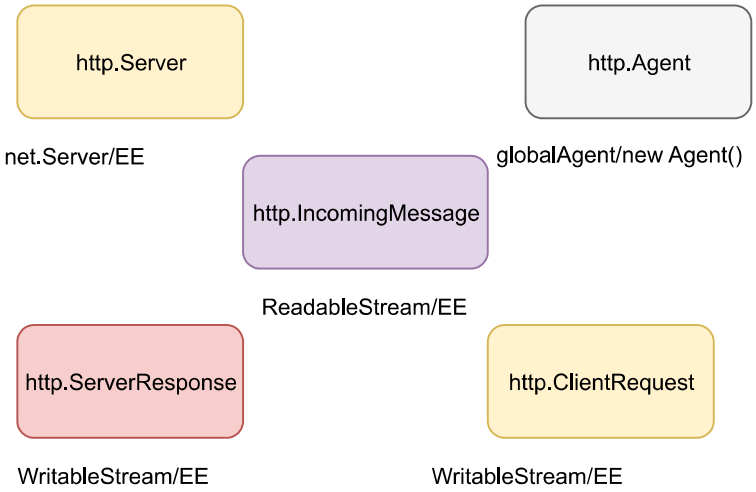
① [http.Server] * the server class is what is used to create a basic server. It inherits from net Server So it is an ee (event emitter) [A server response object is created internally by an http server]

② [http.ServerResponse]

③ [http.Agent] class is used to manage pooling sockets used in http client requests. * Node uses a global agent by default, but the dev can create a different agent when needed.

④ [http.ClientRequest] - when we initiate an http request the dev will be using this. This object is NOT the same as the server obj. That request object is an [incoming message object] ⑤

Both Client Request & Server Response implement Writable Stream / event emitter interface.



⑤ [http.IncomingMessage] 03.28.2018 03.29.2018

incoming Message objects implement (Readable Stream) interface, which is also an event emitter.

The default "request" for http.request is 'get,' e.g.

```
const http = require('http');
                                        , method: 'POST'};
http.request ({ hostname: 'www.nannerl.io' },
              (res) => {
                  console.log(res);
              });
```

The request method takes an option argument and gives access to a callback, for the response of the host you wish to request.

```
http.request(
    { },
    (res) => {

    }
);
```

Notice the handler being defined in the 2nd argument, it does not have an error argument. This is because the handler gets registered as an event listener. Also the error is

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

http.request({hostname: 'www.nannerl.io'},
(res) => {
  console.log(res);
});
```

is handled with an event listener. 03.30.2018

So this mean the http method -> http.request

① returns an object. Which would be an 'event emitter.'

Now we can register a handler for the error event

```
const req = http.request({...
req.on('error', (e) => console.log(e));
req.end();
```

* This request object is a writable stream. This means you can write with a post method.

e.g. req.write('....');

and don't forget to terminate the stream !!!

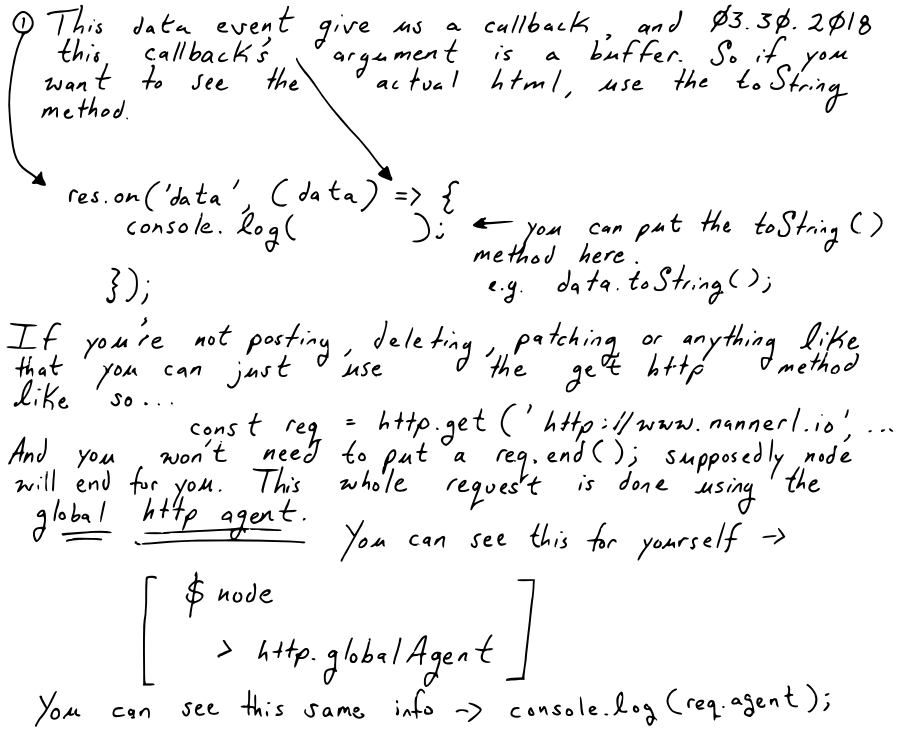
e.g. req.end();

You should get the whole incomming message. You should be able to read info on the response object like so ... console.log(res.statusCode);

console.log(res.headers); <- [for header info]

* Most important * this response object is an event emitter... It emits a data event when it receives data from the hostname.

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

const req = http.request(
  { hostname: 'www.nannerl.io'},
(res) => {
  console.log(res);
  }
);

req.on('error', (e) => console.log(e));

req.end();
```

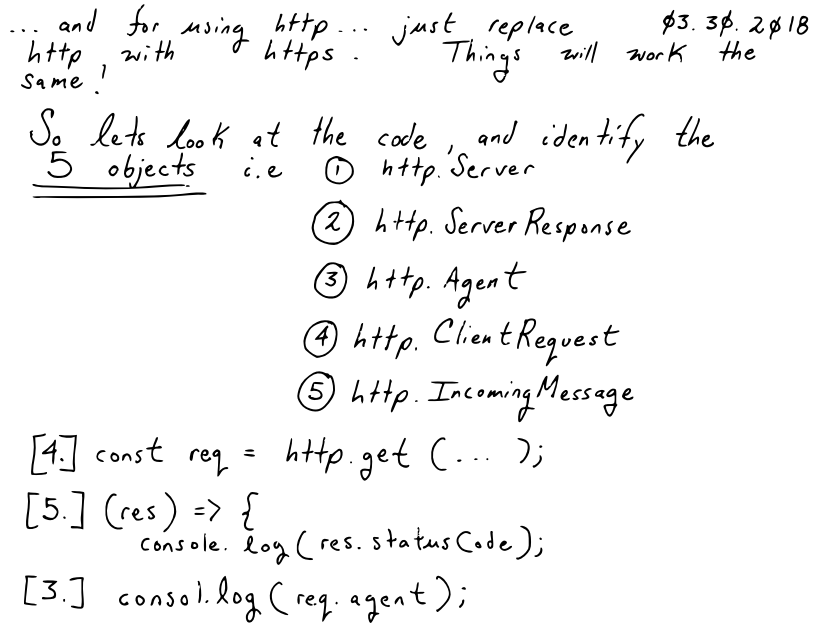



```
$ node stanRequest.js
```

https://github.com/stan-alam/NodeJS/blob/develop/coreNode/10/15-30/src/web/stanRequestOutput.txt

Node REPL output for http.globalAgent (used by node to manage sockets ...etc)

```
$ node
> http.globalAgent
Agent {
  domain:
   Domain {
     domain: null,
     _events: { error: [Function: debugDomainError] },
     _eventsCount: 1,
     _maxListeners: undefined,
     members: [] },
  _events: { free: [Function] },
  _eventsCount: 1,
  _maxListeners: undefined,
  defaultPort: 80,
  protocol: 'http:',
  options: { path: null },
  requests: {},
  sockets: {},
  freeSockets: {},
  keepAliveMsecs: 1000,
  keepAlive: false,
  maxSockets: Infinity,
  maxFreeSockets: 256 }
>
```



In the server 03.30.2018
[1.] const server = require('http').createServer();

The request object inside the request listener is from the incoming message class. [5.]

i.e server.on('request', (req, res) => {

And the response object is from the server response class

i.e.

res.writeHead(200, { 'content-type': 'text ~

Routes in Node

Working with Routes using Node's http module. 03.31.2018

lets support a few routes, a home route, an api route, a burger route.

e.g. curl localhost:8000/home
/api
/burgers

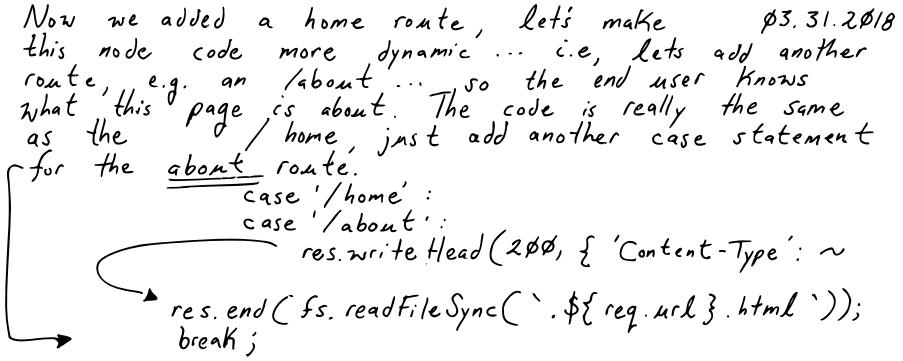
The first thing to do is read the URL info... which can be done by (req.url) The easiest way to handle routes is to use the switch statement. Like so ....

```
server.on('request', (req, res) => {
  switch (req.url) {
  case '/home':

    break;
  case '/':

    break;
  default:
```

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

server.on('request', (req, res) => {
  switch (req.url) {
  case '/home':
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(fs.readFileSync('./home.html'));
    break;
  case '/':

    break;
  default:

  }
});
```

```
server.listen(8000);
```

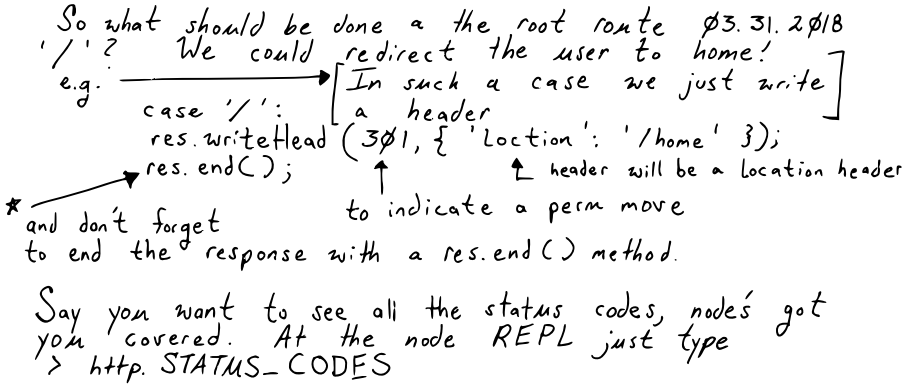
```
$ curl localhost:8000/home
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   289    0   289    0     0   2627      0 --:--:-- --:--:-- --:--:--
3074<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset ="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale-1.0">
  <met http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Nannerl.io</title>
</head>
<body>
  Welcome to Nannerl.io
</body>
</html>
```




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

server.on('request', (req, res) => {
  switch (req.url) {
  case '/home':
  case '/about':
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(fs.readFileSync(`.${req.url}.html`));
    break;
  case '/':
```

```
    break;
  default:

  }
});

server.listen(8000);
```

```
$ curl localhost:8000/about
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   294    0   294    0     0    294      0 --:--:-- --:--:-- --:--:--
287k<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset ="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale-1.0">
  <met http-equiv="X-UA-Compatible" content="ie=edge">
  <title>About</title>
</head>
<body>
  Nannerl.io, by Stan Enterprises
</body>
</html>
```




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

server.on('request', (req, res) => {
  switch (req.url) {
  case '/home':
  case '/about':
```

```
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(fs.readFileSync(`.${req.url}.html`));
    break;
  case '/':
    res.writeHead(301, { 'Location': '/home' });
    res.end();
    break;
  default:

  }
});

server.listen(8000);
```

```
$ curl -i localhost:8000/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0HTTP/1.1 301 Moved Permanently
Location: /home
Date: Sat, 31 Mar 2018 18:45:03 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

```
const fs = require('fs');
const server = require('http').createServer();
const colorData = {
  "colors": [
    {
      "color": "black",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,255,255,1],
        "hex": "#000"
      }
    },
    {
      "color": "white",
      "category": "value",
      "code": {
        "rgba": [0,0,0,1],
        "hex": "#FFF"
      }
```

```
    },
    {
      "color": "red",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,0,0,1],
        "hex": "#FF0"
      }
    },
    {
      "color": "blue",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [0,0,255,1],
        "hex": "#00F"
      }
    },
    {
      "color": "yellow",
      "category": "hue",
      "type": "primary",
      "code": {
        "rgba": [255,255,0,1],
        "hex": "#FF0"
      }
    },
    {
      "color": "green",
      "category": "hue",
      "type": "secondary",
      "code": {
        "rgba": [0,255,0,1],
        "hex": "#0F0"
      }
    },
  ]
}

server.on('request', (req, res) => {
  switch (req.url) {
  case '/api':
    res.writeHead(200, { 'Content-Type': 'application/json' });
    res.end(JSON.stringify(colorData));
    break;
  case '/home':
  case '/about':
    res.writeHead(200, { 'Content-Type': 'text/html' });
```

```
    res.end(fs.readFileSync(`.${req.url}.html`));
    break;
  case '/':
    res.writeHead(301, { 'Location': '/home' });
    res.end();
    break;
  default:
    res.writeHead(404);
    res.end();
  }
});

server.listen(8000);
```

What if you wish to work with routing JSON data? Well, in the switch case, add an api route, like so: 03.31.2018

```
case '/api':
res.writeHead(200, {'Content-Type': application/json'});
```

* You would want to return the JSON Data as a stringfied version of the data.

e.g.

```
res.end( JSON.stringify (data)); // you put a variable for the json data like so  const data = { };
```

★ But what happens when send a request with some garbage route, something that does not exist? At this time we don't have logic in place that will handle said scenerio. If you enter a garbage string for the route, the webServer will time out. In such a case you can add the logic at the default switch/case.

```
default:
  res.writeHead (404);
    res.end(); // make sure to end the response
```

```
$ curl -i localhost:8000/api
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   559    0   559    0     0  34937      0 --:--:-- --:--:-- --:--:--
34937HTTP/1.1 200 OK
Content-Type: application/json
Date: Sat, 31 Mar 2018 19:14:47 GMT
Connection: keep-alive
Transfer-Encoding: chunked

{"colors":[{"color":"black","category":"hue","type":"primary","code":
{"rgba":[255,255,255,1],"hex":"#000"}},
{"color":"white","category":"value","code":{"rgba":
[0,0,0,1],"hex":"#FFF"}},
{"color":"red","category":"hue","type":"primary","code":{"rgba":
```

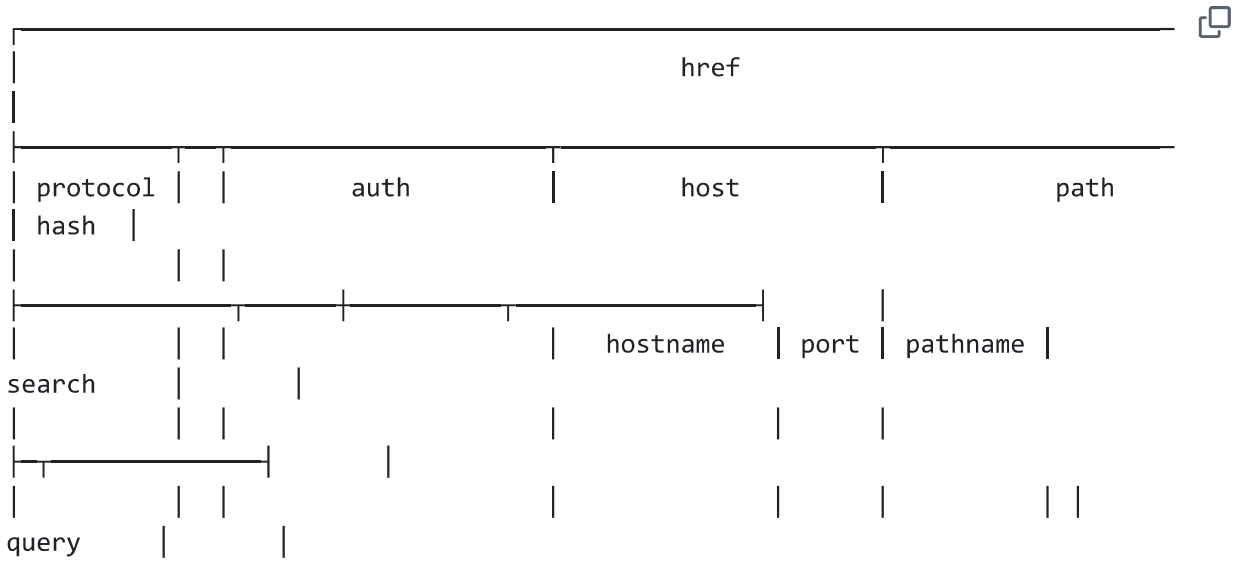
```
[255,0,0,1],"hex":"#FF0"}},
{"color":"blue","category":"hue","type":"primary","code":{"rgba":
[0,0,255,1],"hex":"#00F"}},
{"color":"yellow","category":"hue","type":"primary","code":{"rgba":
[255,255,0,1],"hex":"#FF0"}},
{"color":"green","category":"hue","type":"secondary","code":{"rgba":
[0,255,0,1],"hex":"#0F0"}}]}
```

When trying curl -i localhost:8000/garbage

```
$ curl -i localhost:8000/garbage
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--
0HTTP/1.1 404 Not Found
Date: Sat, 31 Mar 2018 19:14:55 GMT
Connection: keep-alive
Transfer-Encoding: chunked
```

This diagram is from the NODE.org api page for the URL module

parsing the following url: http://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash



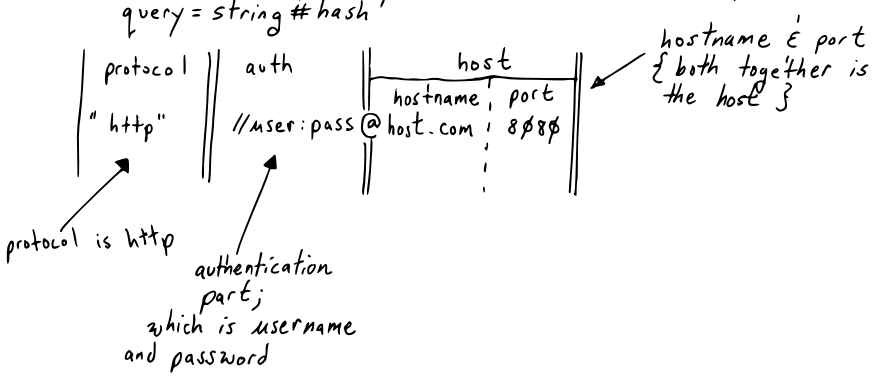
```
"  https:   //    user   :   pass   @ sub.host.com : 8080   /p/a/t/h  ?
query=string   #hash "
│          │  │          │          │   hostname    │ port │          │
│       │
│          │  │          │          ├─────────────────┴──────┤          │
│       │
│ protocol │  │ username │ password │        host           │          │
│       │
├──────────┴──┼──────────┴──────────┼───────────────────────┤          │
│       │
│   origin    │                    │       origin          │ pathname │
search     │ hash  │
├─────────────┴────────────────────┴──────────────────────┴──────────┴─────────
│                                          href
│
└──────────────────────────────────────────────────────────────────────────────
```

(all spaces in the "" line should be ignored -- they are purely for formatting)

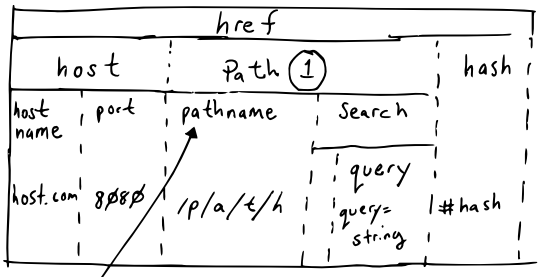
[Parsing URLs and Query Strings] 03.31.2018

So lets say, you wish to work with URLs -> node's got you covered! From the Node REPL, type > url And this also covers parsing!!! The url module is awesome! Take a gander at * https://nodejs.org/api/url.html * This is the Api reference page for the url module.

The diagram is manually parsing 'http://user:pass@sub.host.com:8080/p/a/t/h?query=string#hash'



03.31.2018



These are all the elements within a url object

★ So if you have this object and you wish to parse it into these elements You are in Luck!!! Node's url module has you covered.

pathname which comes after the host, and before the query string

The query string including the question mark (?) is 'Search'

★ Without the question mark, it is just 'query' while pathname (+) 'Search' is the path [1]

And if there is a hash location, it will be called hash!

Remember this URL object is a string object.

04.01.2018

you can use the url.parse() method at the node REPL. e.g.

```
> url.parse('http://nannerl.io')
```

you may see something such as this.

```
Url {
  protocol: 'http:',
  slashes: true,
  auth: null,
  host: 'nannerl.io',
  port: null,
  hostname: 'nannerl.io',
  hash: null,
  search: null,
  query: null,
  pathname: '/',
  path: '/',
  href: 'http://nannerl.io/' }
>
```

★ Since we don't see a query string, we can't parse it. However, it would not be difficult to do so. You should be able to do so with a second argument.

```
> url.parse('http://nannerl.io', true)
```

What if you have the inverse, i.e what if you wish to format the url? 04.01.2018
It's all good, node's got you covered ... simply (at the node REPL) use the url.format() method. like so

```
> url.format ( { protocol : http
                 host     : nannerl.io
               ~  ~  ~  ~  ~  ~
               })
```

this will format all the object element properties into a string, i.e. it will put them back to the nicely formatted url, that everyone is so used to.

There is also another module that should be discussed. This module also has some great methods available. This would be the parse method and the stringify method. e.g. →

```
> {
... name: 'Ludwig Wittgenstein', website: 'nannerl.io/L-W'}
...
> querystring.stringify({ .... }) -> using this method on the object )
```

name=Ludwig%20Wittgenstein&website=nannerl.io%2FL-W'

*notice how the stringify method escaped special characters by default. If you have the inverse situation, where you wish to parse a query string into an object. Just use querystring.parse(); it will return a JSON of that query string.

Common Built-in Modules in Node 04.01.2018

Node provides a 'plethora' of utilities for acting on the OS. use the OS module for this. * const os = require('os'); Try this at node REPL > os.cpus()] this will give you info on CPUs their models, speed, times..etc * You can even read information on network interfaces.

↳ > os.networkInterfaces().en0.map(i => i.address) * gives you IP addresses, but you can also read mac addresses. netmasks, families → you name it Node's got it Node is so good it can probably do your taxes!!!

> os.freemem(), will tell you about free memory (you can also check for total mem)
> os.type(), will tell you what type of OS your special node was compiled on. * This way you can write code that is specific to your operating System.
> os.release(), will allow you to read the release info on the operating system.
> os.userInfo() → is also very handy. It returns the user info on the current user. This would be good if you're writing a script custom to the shell user.
On windows the shell attrib is null, both uid and gid = -1

> os.constants.signals, is another handy method. It will list a 'whole 'plethora' of [os exit codes] and signal codes. 04.01.2018

os.constants

★ The fs module ★ : provides simple I/O functions to use with node. * All fs functions have sync, and async forms. (you should not mix these types together) You should pick one form that is best for your code logic.
e.g. Say you're reading a file during a server initialization process. Use the readFileSync --▸ fs.readFileSync( )
This will read the file only once, and then it is done.
When you're reading a file in which a user may be writing to, or the file is changing (reacting), then use the async function.

• Most important thing to know, besides that these two function handle read/write differently, they also handle errors in a different way.

The async function pass any errors, as the first argument in the callback, while the sync function throws an immediate error. Recall this was an issue with the hub-node test rig. i.e when the webdriver could not write to the log it would throw an error to the OS, not to node, node was just spawning the webdriver process. That's why the try-catch block would not work, and Jenkins would crash!!!!

here is example of an async fs and Sync fs 04.01.2018

★ The readFile method on default returns a buffer

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

fs.readFile(__filename, (err, data) => {
  if (err) throw err;

});

const data = fs.readFileSync(__filename); // sync
      // here exception will be immediatly thrown
```

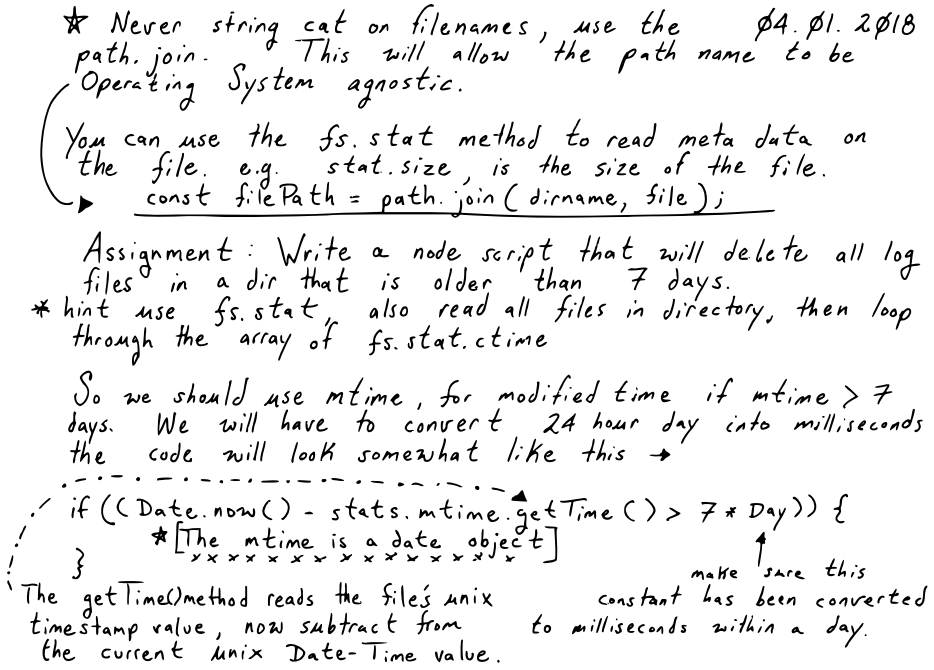
Take a look at this script ->

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

const files = fs.readdirSync(dirname);
files.forEach(file => {
```

[The readdir method is for reading the directory for a list of files.]

★ this will be an array of file names, stored in the file constant. (Just the names) not the full paths

To get the full path use the paths module.




```
const fs = require('fs');
const path = require('path');
const dirname = path.join(__dirname, 'files');
const files = fs.readdirSync(dirname);
const Day = 24*60*60*1000; // millaseconds in a Day

files.forEach(file => {
  const filePath = path.join(dirname, file);
  fs.stat(filePath, (err, stats) => {
    if (err)
      throw err;
    if((Data.now() - stats.mtime.getTime() > 7*Day)) {
      fs.unlink(filePath, (err) => {
        if (err)
          throw err;
        console.log(`deleted ${filePath}`);
      });
    }
  });
});
```

the fs.unlink(); will delete files. 04.01.2018
use fs.watch(); for watching a change in a file
* Write a node script that will watch a directory for three events. These events are a file is added, removed, changed. fs.watch(); will not give you the ability to act on all these aforementioned events. e.g. both 'add' and delete events are registered as 'rename' to fs.watch() method.

① first read the files in the specific directory that is to be watched.
↳ const currentFiles = fs.readdirSync('dirname');
② Start the watch listener
↳ fs.watch(dirname, (eventType, filename) => {
↳ if the event type is 'rename', means a file is deleted or added
↳ if (eventType === 'rename') {
③ If the file exists in the currentFiles array, then the event is a remove event
↳ const index = currentFiles.indexOf(filename);
↳ which the current file array will be updated to reflect the event.
i.e update the removed file → currentFiles.splice(index, 1);
logWithTime(`${filename} was removed`);
this just outputs the time in UTC

④ at this point of the code 04.01.2018
↳ currentFiles.push(filename);
logWithTime(`${filename} is added`);
return;
}
means a file is being added to the directory.
It says to update the current file array [push] and log the message
⑤ So if the event type is not rename {which covered both del/add} the the code will reach the 'change' event which is supported by the fs.watch method. Here the code will log the UTC stamp at which the file 'change' event occurs.
↳ logWithTime(`${filename} has been modified`);

Remember fs module (i.e. the API) is not universal for all OS environments. "Test the code."

```
const fs = require('fs');
const path = require('path');
const dirname = path.join(__dirname, 'files');
const files = fs.readdirSync(dirname);

const logWithTime = (message) =>
  console.log(`${new Date().toUTCString()}: ${message}`);

fs.watch(dirname, (eventType, filename) => {
  if (eventType === 'rename') {
    const index = currentFiles.indexOf(filename);
    if (index >= 0) {
      currentFiles.splice(index, 1);
      logWithTime(`${filename} has been expunged`);
      return;
```

```
    }

    currentFiles.push(filename);
    logWithTime(`${filename} has been added`);
    return;
  }

  logWithTime(`${filename} a change has occured`);
});
```

Console & Utilities 04. 01. 2018

The console module is designed to match the console object provided by web browsers. In node there is a console class that can be used by devs to write to any node stream. There is also a global console that is configured to write to stdout and stderror. ✱ Remember that these are two different things. ✱ i.e. if you wish to write to a socket or a file. All you need to do is instantiate a new console object from the console class & pass the desired output and error streams as args. e.g.)

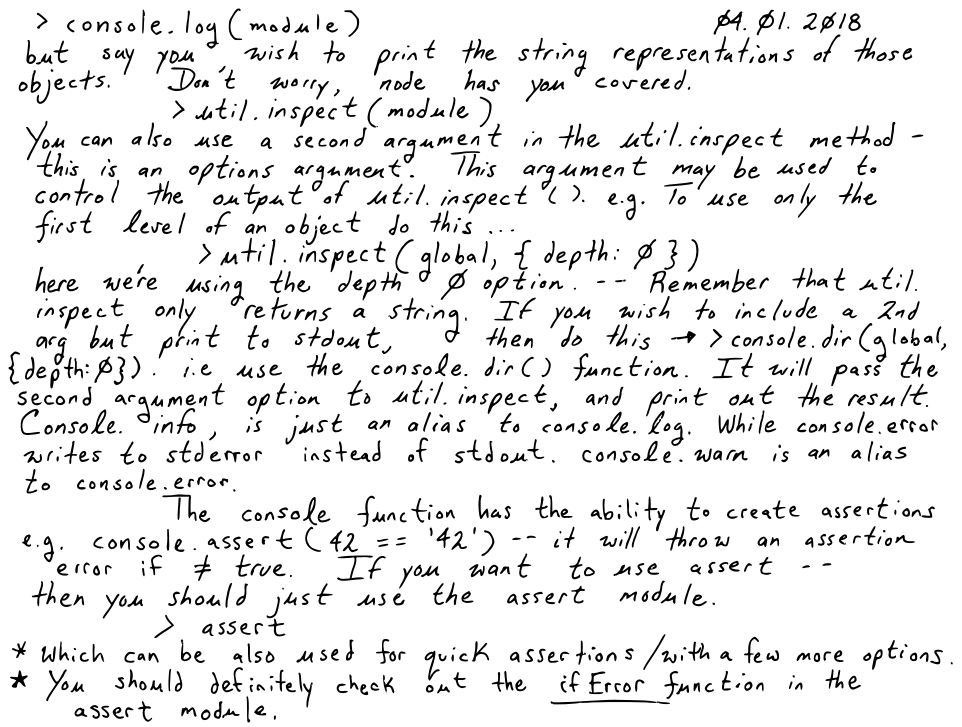
```
const myConsole = new console.Console(out, err);
```

✱ console.log, uses the util module, i.e. under the hood. i.e to format and output a message with a new line. You can even use printf formatting elements, or print multiple args and print multiple elements on the same line. %d for number, %s for stream.✱ ✱ There is also %j for JSON object, yay !!!

If you want to use the printf substitutions - without console log -> you can use the util dot format method ↴

```
> util.format('Some %s', 'thing');
```

this will return the formatted string.
You can even console log objects, like so ->



> console

```
> console
Console {
  log: [Function: bound consoleCall],
  info: [Function: bound consoleCall],
  warn: [Function: bound consoleCall],
  error: [Function: bound consoleCall],
  dir: [Function: bound consoleCall],
  time: [Function: bound consoleCall],
  timeEnd: [Function: bound consoleCall],
  trace: [Function: bound consoleCall],
  assert: [Function: bound consoleCall],
  clear: [Function: bound consoleCall],
  count: [Function: bound consoleCall],
  countReset: [Function: bound countReset],
  group: [Function: bound consoleCall],
  groupCollapsed: [Function: bound consoleCall],
  groupEnd: [Function: bound consoleCall],
  Console: [Function: Console],
  debug: [Function: debug],
  dirxml: [Function: dirxml],
  table: [Function: table],
  markTimeline: [Function: markTimeline],
  profile: [Function: profile],
  profileEnd: [Function: profileEnd],
  timeline: [Function: timeline],
```

```
  timelineEnd: [Function: timelineEnd],
  timeStamp: [Function: timeStamp],
  context: [Function: context],
  [Symbol(counts)]: Map {} }
>
```

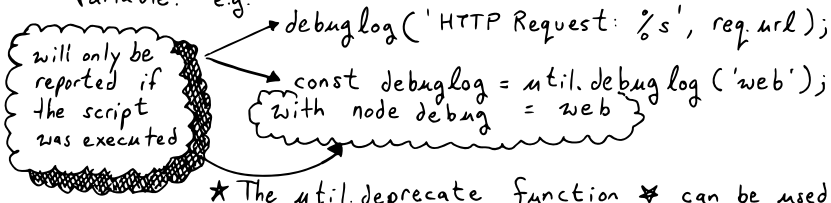
> assert

```
$ node
> assert
{ [Function: ok]
  fail: [Function: fail],
  AssertionError: [Function: AssertionError],
  ok: [Circular],
  equal: [Function: equal],
  notEqual: [Function: notEqual],
  deepEqual: [Function: deepEqual],
  deepStrictEqual: [Function: deepStrictEqual],
  notDeepEqual: [Function: notDeepEqual],
  notDeepStrictEqual: [Function: notDeepStrictEqual],
  strictEqual: [Function: strictEqual],
  notStrictEqual: [Function: notStrictEqual],
  throws: [Function: throws],
  doesNotThrow: [Function: doesNotThrow],
  ifError: [Function: ifError] }
>
```

The ifError function throws value if the value is "truthy." Which is usually done on any errors argument in callbacks. 04.01.2018

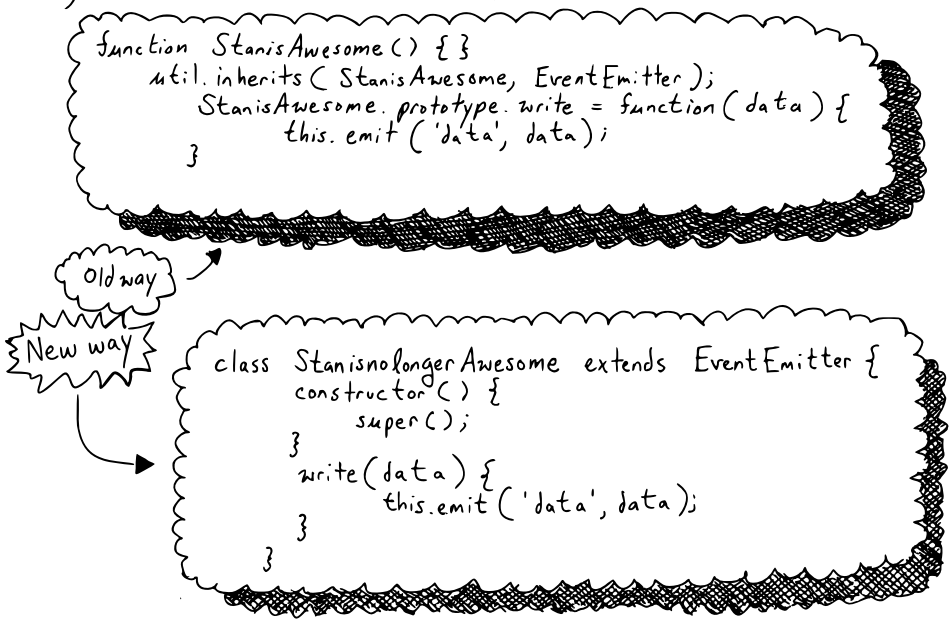
★ console.trace("here") ★ is just like console error but it also prints the call stack at the point where the error is placed. Which is cool when you need to debug problems. You can use console.time(), and console.timeEnd() to start and timers and even report the duration of an operation. * The argument you pass to this timers should be a unique label for that operation.
e.g. > console.time("test") ⟶ > console.timeEnd("test")

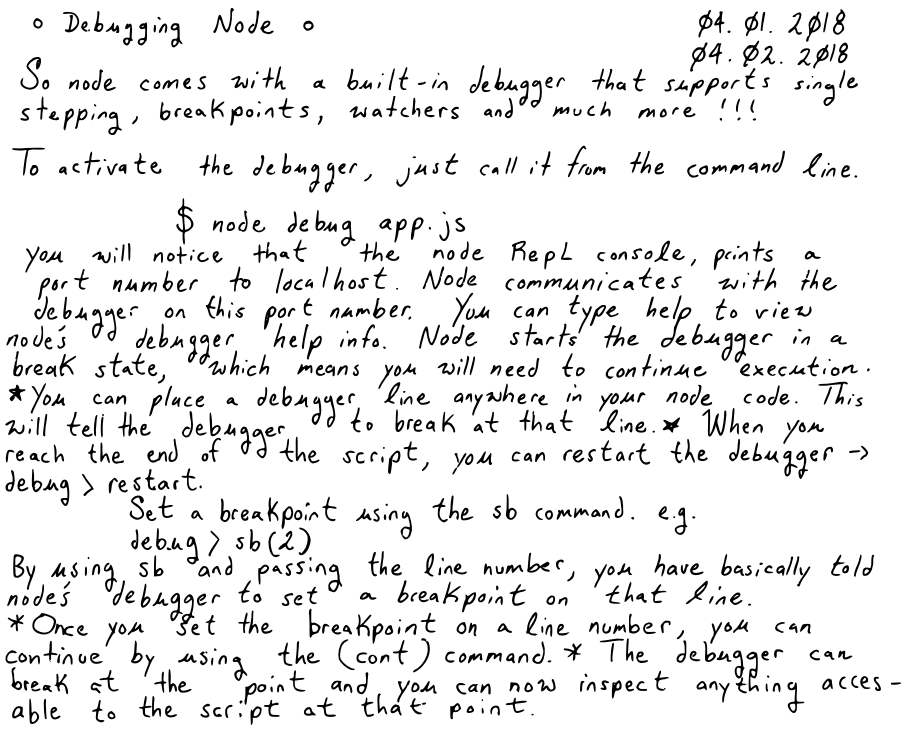
The util module has some more handy options, e.g. There is a debuglog -> if you wish to write debug errors (conditionally) to stderror -> depending on the existence of the node debug variable. e.g.



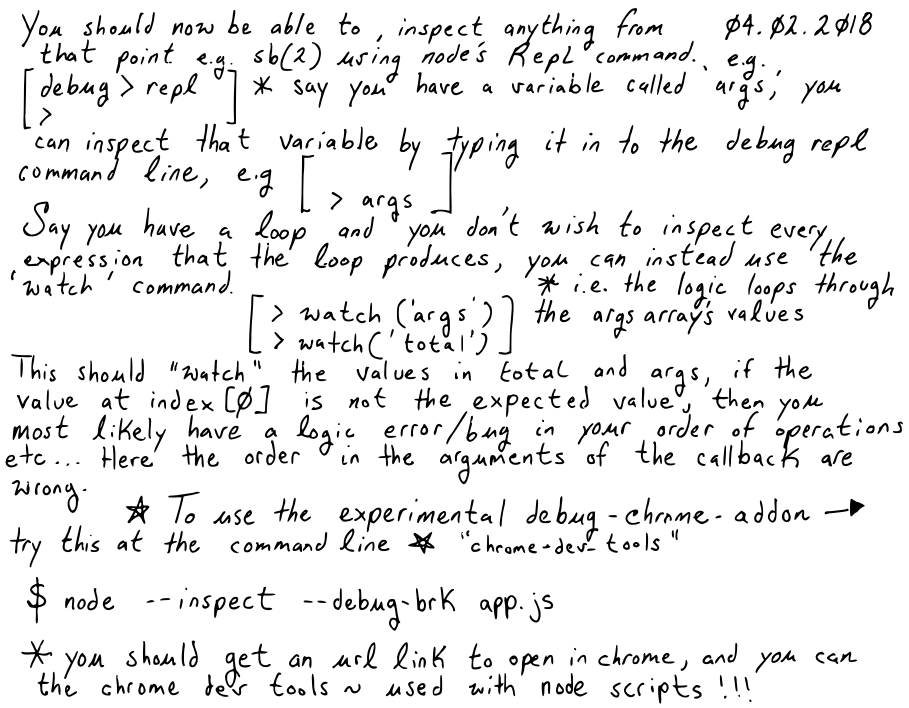
★ The util.deprecate function ★ can be used to wrap a function, before you export it. To mark that function as deprecated. So this way users of that function or module will get a warning! The first time they use that function/module.

Let's also take a look at the util.inherits() 04.01.2018 method -> it was heavily used by node devs, until the introduction of Es6 classes, for inheriting prototype methods from one constructor to another.






```
function negativeSummation(...args) {
  return args.reduce((arg, total) => {
    return total-arg;
  }, 0);
}

console.log((negativeSummation, 1, 3, 11));
//this is not working? So lets use the debugger to
//step through the code!
```




```
$ node debug negativeSummation.js
(node:18124) [DEP0068] DeprecationWarning: `node debug` is deprecated.
Please us
e `node inspect` instead.
< Debugger listening on ws://127.0.0.1:9229/14f57090-774a-40b4-820a-
9a83cd90e7b3

< For help see https://nodejs.org/en/docs/inspector
Break on start in negativeSummation.js:1
> 1 (function (exports, require, module, __filename, __dirname) { function
negat
iveSummation(...args) {
  2   return args.reduce((arg, total) => {
  3     return total-arg;
debug>
```

◦ Working with Streams in Node ◦ 04.02.2018

If you're going to be working with large amounts of data in Node, means you will be working with streams. To quote Dominic Tarr, "Streams are node's best and most misunderstood idea." There are many npm packages available to devs for making working with streams easier. We we will be focusing on the native node stream API. Take for example, the pipe feature in Unix. You can create powerful Unix/linux commands by piping smaller commands. The idea is the same with streams in Node.

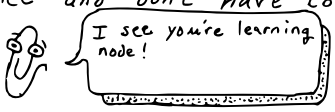
★Many of the built in modules in Node implement the streaming interface.

Readable Streams	Writable Streams
http resp, on the client	http responses, on the client
http requests, on the server	http requests, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout & stderr	child process stdin
process.stdin	process.stdout, process.stderr

★you can see there are inverse relationships within the list. Basically one object produces, and the other object consumes. vice versa ★

So what the heck is a stream? 04.02.2018

"Streams are collections of data that might not be available all at once and don't have to fit in memory" - microsoft clippy



Collections of data like arrays and strings but with the difference being that the data is not available all at once. Streams are powerful because you don't have to store the data in memory, which is good for large amounts of data (huge chunks of data, or data coming from an external source.

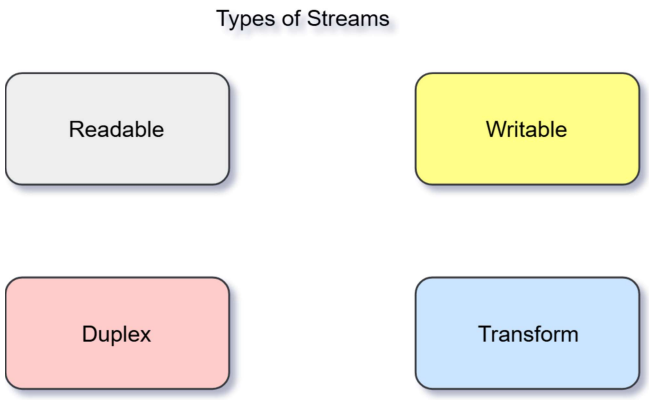
★you can use the fs module's stream interface to read & write streams! Remember that the response object in node is also a writable stream. So say, you have an ascii text file that is 450 megabytes. It would be highly inefficient to load all that in memory and serve it all at once. ★ Remember time & timeEnd for streaming time intervals over http? ★ This is basically the same use of those type of methods.

The fs module allows to pipe any file using the createReadStream method which you pipe into the writable stream, e.g. src.pipe(res);

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

server.on('request', (req, res) => {
  const src = fs.createReadStream('./giganticFile.txt');
  src.pipe(res); //here wer'e piping the contents of the file to the res obj
});
```

```
server.listen(8888);
```

Four Types of Streams in Node



★ Types of Streams ★ 04.04.2018

A readable stream - is an abstraction for a source in which data can be consumed. e.g. fs.createReadStream method.

A writable stream - is an abstraction for a destination to which data can be written. e.g. fs.createWriteStream method.

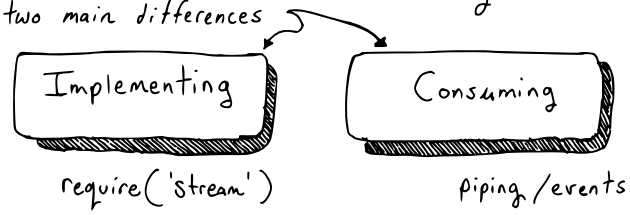
A Duplex Stream - is both readable and writable, e.g a socket, net.socket

A transform stream - is basically a duplex stream that can used to modify or transform data as it is being read. e.g. zlib.createG-zip stream. zlib.create Gzip stream compresses data using G-zip. ★ You can think of a transform stream as a function, where the input is the writable Stream part, and the output is the readable stream part. ★ Sometimes you may hear of transform streams as "through" streams.

★ All Streams are instances of Event Emitter ★

This means that all streams are EventEmitters. You can consume streams using the pipe method. e.g. src.pipe(dst); Here the readable stream, src is being piped to a writable stream (dst) destination. src(source) has to a readable stream, and (dst) destination has to be a writable stream. ★ They can be both duplex streams as well. ★★★ If you're piping duplex streams - You can chain pipe calls !!!

Just like piping in linux!!! 04.04.2018
So in linux it will look like this ->
a | b | c | d
in node it will look like this ->
a.pipe(b).pipe(c).pipe(d); a pipe into b, b into c, and finally c is piped into d. * given that b and c are both duplex streams. * It is advised to use either pipes or events, but not both at the same time. (mixing them) usually when you're using pipes you don't need to use events. If you need to consume streams in a more customized fashion -> then use events.
*** Streams *** - When talking about streams in node, there are two main differences

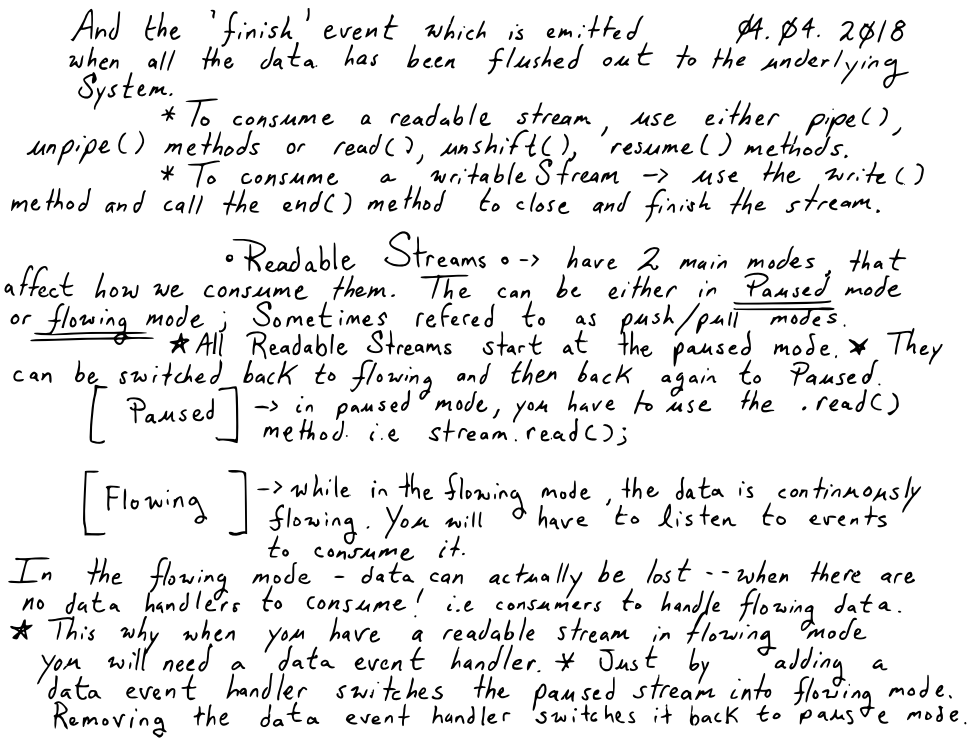


There is the task of implementing streams, and then there is the task of consuming streams. Stream Implementers usually use the stream module. For consuming all you need to do is listen to the stream event or use pipe.

Readable & Writable streams have functions 04.04.2018
that are ' related -> Which means they are usually used to-gether

Readable Streams	Writable Streams
Events	Events
- data	- drain
- end	- finish
* - error	- error
* - close	- close
- readable	- pipe/unpipe
Functions()	Functions()
- pipe(), unpipe()	- write()
- read(), unshift(), resume()	- end()
- pause(), isPaused()	- cork(), uncork()
- setEncoding()	- setDefaultEncoding()

Some events are similar, like error and close* events*. The most important events on the readable Stream are the data event. - which is emitted whenever an event passes a chunk of data to the consumer and the 'end' event which is emitted when there is no more data to be consumed from the stream. * The most important events on the writable stream are the drain event { which is a signal that the writable stream can receive more data.




```
//readStream2.js
const { Readable } = require('stream'); // implement a readable stream, requir

const streamIn = new Readable({
  read(size) {
    setTimeout(() => {
      if(this.currentCharCode > 90){ //stop at letter z
        this.push(null);
        return;
      }
      this.push(String.fromCharCode(this.currentCharCode++));
  }, 100);
 }
});

streamIn.currentCharCode = 65;
streamIn.pipe(process.stdout);
```

```
//writeStream.js
const { Writable } = require('stream');

const streamOut = new Writable({
  write(chunk, encoding, callback) {
  console.log(chunk.toString());
  callback();
```

```
  }
});

// basically anything this receives will echo right back.
process.stdin.pipe(streamOut); // piping stdin which is a readable stream into
```

At bash, type node readStream3.js | head -c3

```
//readStream3.js
//run this on linux
const { Readable } = require('stream'); // implement a readable stream, requir

const streamIn = new Readable({
  read(size) {
    setTimeout(() => {
      if(this.currentCharCode > 90){ //stop at letter z
        this.push(null);
        return;
      }
      this.push(String.fromCharCode(this.currentCharCode++));
  }, 100);
 }
});

streamIn.currentCharCode = 65;
streamIn.pipe(process.stdout);

process.on('exit', () => {
  console.error(`\n\ncurrentCharCode : ${streamIn.currentCharCode}`);
});
//stdout is not a tty got this on git-bash for win
process.stdout.on('error', process.exit); //register an error on stdout, and c
```

✦ Duplex Streams !!!! 04.04.2018

With duplex streams - you can implement both readable and writable streams with the same object. It's like you can inherit from both interfaces. For every duplex stream, you will have to implement both read/write streams. So e.g. You can create a node script that can write a message to console and read input with stdin -> at the same time, and echo the user input at the same time.

★ process.stdin.pipe(streamINnOut).pipe(process.stdout); ★
here the standard in is piped into the duplex stream (to echo) and pipe the duplex stream itself into the standard out. (for reading letters A-Z)

★ Very Important → The readable and writable sides of a duplex stream - operate completly independent of each other. ★

{Here we are just grouping two features into an object}

A tranform Stream's ouput is computed from its input. In a transform method, you don't have to implement read and/or write. You only have to combine a transform method, that will combine both of them. It has a signature of a write method -> and with this you can push data, as well.

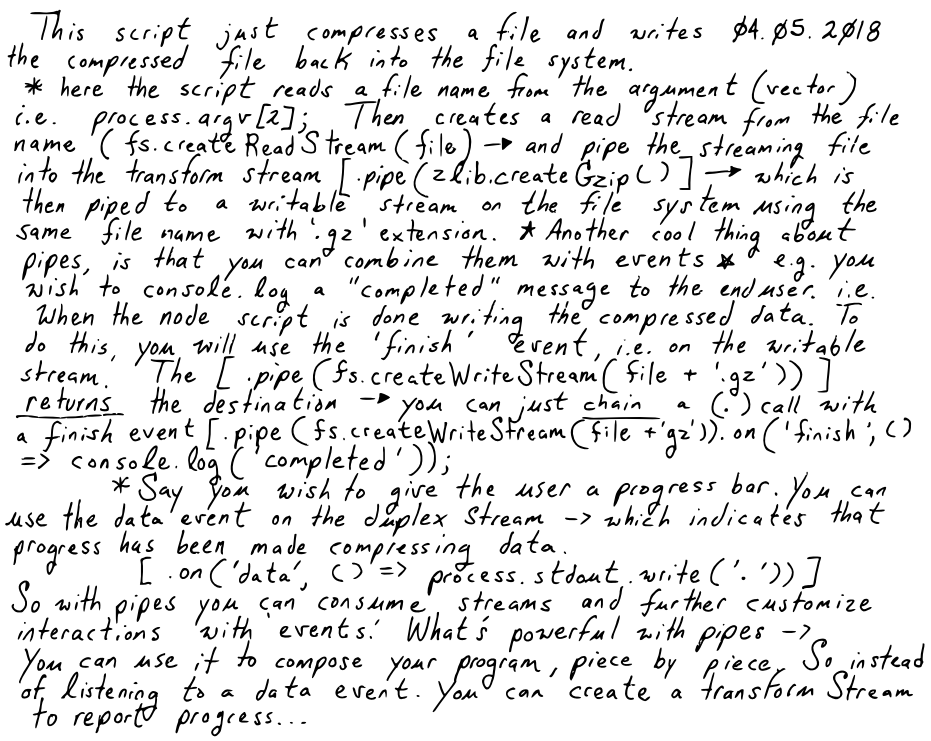
04.04.2018

```
const { Transform } = require('stream');

const upperCaseTr = new Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});
process.stdin.pipe(upperCaseTr).pipe(process.stdout);
```

here we're just converting the chunk (within the transform method) to an uppercase version, and then the uppercase version will be pushed as the readable part. * You will still have to 'call' the callback!!! So basically what this code is doing is the same echo, but converting the user input to all upper case, yay !!!

Zlib.createGzip() function

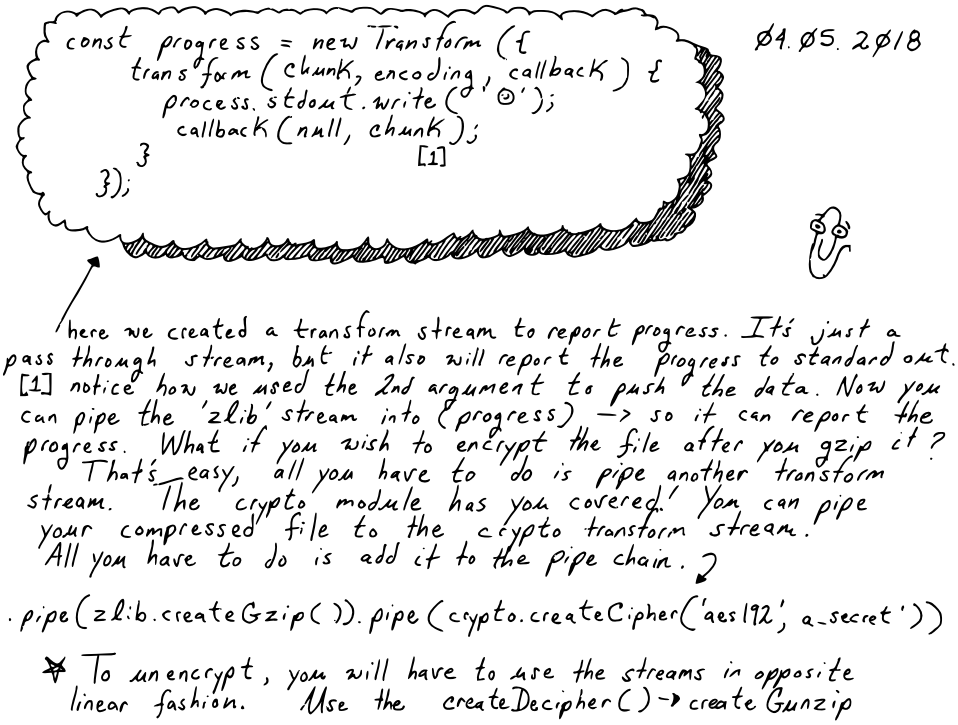
```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];
        fs.createReadStream(file).pipe(zlib.createGzip())
        .pipe(fs.createWriteStream(file + '.gz'));
```




```
//stanZip2.js
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .on('data', () => process.stdout.write('0'))  //listening to data and update
  .pipe(fs.createWriteStream(file + '.gz'))
  .on('finish', () => console.log('Completed'));
```

console output

```
$ node stanZip2.js tux.txt
00Completed
```




```
//stanZip3.js
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];
const crypto = require('crypto');

const { Transform } = require('stream');

const progress = new Transform({
  transform(chunk, encoding, callback) {
    process.stdout.write('☻');
    callback(null, chunk);
  }
});

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(crypto.createCipher('aes192', 'a_secret'))
  .pipe(progress)
  .pipe(fs.createWriteStream(file + '.zz'))
  .on('finish', () => console.log('Completed'));
```

Child processes and Clusters 04.05.2018

Single threaded - non-blocking performance is awesome! But eventually one process on one cpu is not going to be enough to handle an increasing workload of an application. It doesn't matter how powerful your server could be, what a single thread can support is limited.

* Scaling Node Apps * -- The advantage of node's single thread is the ability to take advantage of multiple processes and multiple machines. *Using multiple processes is the only way to scale a node application.

* Node is designed for building distributed applications with many nodes. Hence, the name - Node JS. Scalability is not some afterthought when it comes to Node. Scalability is baked into node.

Workload is one of the popular reasons for scaling applications. Availability would be another, and fault tolerance

It is important to understand and discuss Strategies for Scalability. There are mainly 3 things that you can do to scale an application.

Scalability Strategies - cont... 04.05.2018

Cloning (the application) - is the easiest approach to scale an application - multiple times and have each cloned instance handle a part of the workload. * Does not cost much dev time & is highly effective.

Decomposing : is another way an application can be scaled up. i.e decomposing the application based on functionality & services. {microservices} This means having multiple application's with their own dedicated code-bases and databases, {along with user interfaces} The size of the service is not important -- rather the enforcement of loose coupling & high cohesion between services. The implementation of this strategy is not easy.

Splitting : Another awesome strategy - in which the application is split into multiple instances, in which each instance is responsible of only a portion of the applications data. This strategy is called "horizontal partitioning." - Sharding in database terms. + This requires a "lookup" step to see which instance of the application to use -> e.g. Say you wish to partition users according to language or country -- you would have to look up this info first.

Child Processes events and Standard IO 04.06.2018

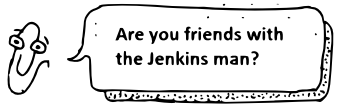
It is easy to spin child processes in Node, i.e using the child process node module. These child processes can communicate with each other via a messaging system.

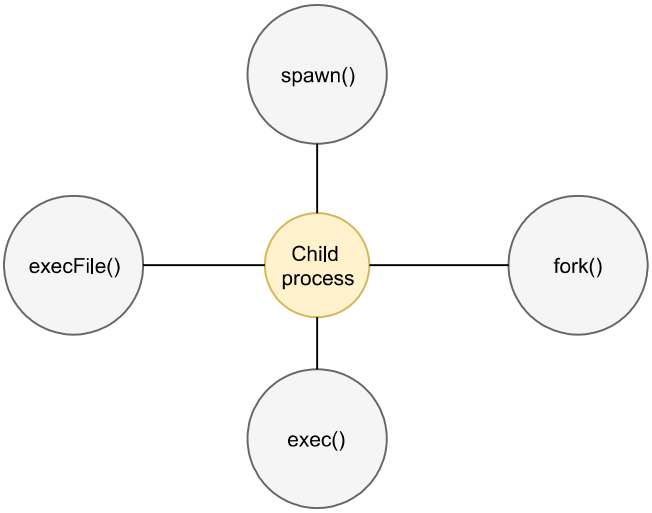
✦ Node and the OS ✦

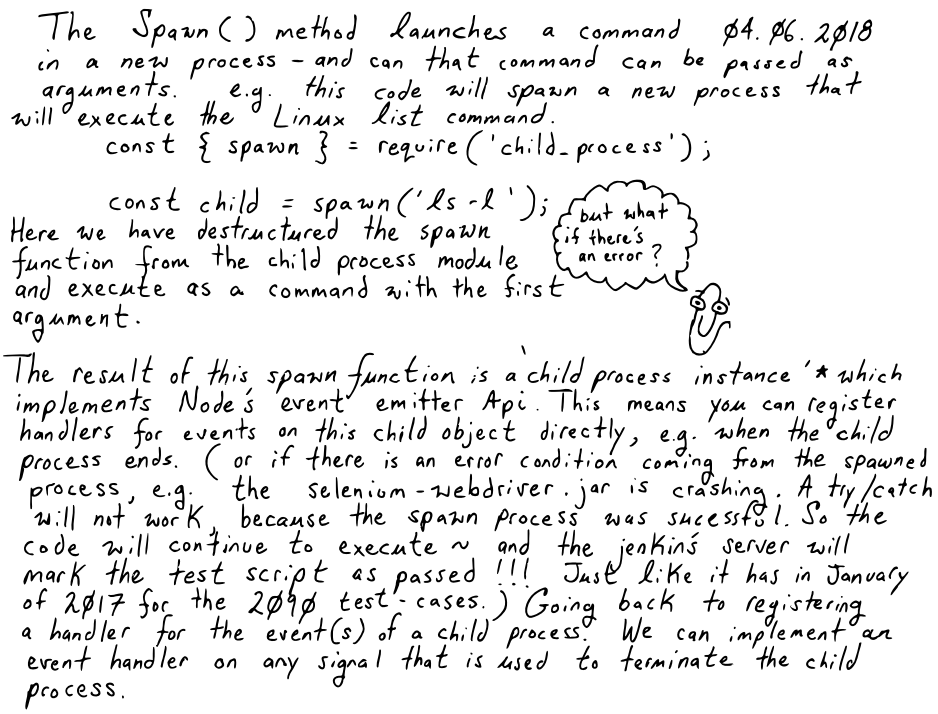
The child process module allows the dev to access Operating System functionalities by running any system command while running inside the child process, e.g. using shell. exec to start Webdriver. Furthermore, control the input stream, and listen in on the output stream. You can control the arguments passed to the command, and do whatever we want with its output. e.g. You can pipe the input of one command into another command as all inputs and outputs of these commands can be presented to the dev using node streams. There 4 different ways you can create child processes in node.

★ Spawn ★ fork ★ exec ★ execFile

▸ Remember these are all functions








```
//spawnList.js
const { spawn } = require('child_process');
const child = spawn('ls');

child.on('exit', function (code, signal) {
  console.log(`child process exited with code ${code}, ${signal}`)}); //here i
  //test will be marked as failed.
```

The other events you can register handlers for 04.07.2018
i.e. this child process instance are -> disconnect, error, message, close
° The disconnect is triggered when the parent process manually calls the child disconnect method. While an error event is triggered if a process can't be spawned or Killed. Finally, the message event is important to understand. °

The message event is triggered when the child process uses the process.send() method to send messages. ✯ This is how parent-child processes can communicate with each other ✯

The close event is emitted when the standard I/O stream of a child process get closed. Every child process gets 3 standard I/O streams - which can be accessed using child.stdin, child.stdout, child.stderr. When the standard I/O of a child process close - the close event is triggered - which is not the same as the exit event. i.e multiple child processes may share the same standard I/O streams. Also just because a child processes exited does not mean that the stream was closed. Since all streams are event emitters - you can listen to different events on those streams attached to every child process. Unlike a normal process - a child process stdout - and stderr are Readable, while stdin is writable - this is the inverse of the types found in a normal process.

Most important - the events on the mentioned 04.07.2018
streams are the standard ones. On the readable streams you can listen to the data event, which will have the output of the command or any error encountered while executing the command.
e.g.

```
child.stdout.on('data', (data) => {
    console.log(`child stdout: \n${data}`);
});
child.stderr.on('data', (data) => {
    console.log(`child stderr: \n${data}`);
});
```

These two handlers will log both cases to the main process standard out and standard error. When the script is executed the output of the ls command is printed and the child process exits with code 0; meaning no error occured.

You can also pass an array of arguments to the spawn command. const child = spawn('ls', [...]); Say for example to find all files in your working directory - use the linux command $ find . -type f. To use this in node, use the spawn process.

```
const child = spawn('find', ['.' '-type', 'f']);
```

```
//spawnPipeWC.js
const { spawn } = require('child_process');
const child = spawn('wc');

process.stdin.pipe(child.stdin);

child.stdout.on('data', (data) => {
```

```
  console.log(`child stdout:\n${data}`);
});
```

★ If there is an error during the execution of the command, the stderr event will be triggered - and a code 1, will be emitted to the OS. (linux) 04.07.2018

A child process stdin is a writable stream. It can be used to send a command some input. Just like any writable stream the easiest way to consume it -> is using the pipe function. i.e when we pipe a readable stream into a writable one. Since the top level process stdin is a readable stream, you can pipe that to a child process stdin stream. e.g.

```
const child = spawn('wc');
process.stdin.pipe(child.stdin)
```

Here we pipe the main process stdin (readable stream) to the child process stdin which is a writable stream.

So whatever you type will become the input for the wc command. You can also pipe the stdin and stdout on multiple processes on each other just like you can do in linux. e.g. You can pipe the [find command] to the standard in of the WC command (stdin) to count all the files in the current directory. All that needs to be done -> is define both child processes and pipe the output of the first one to the input of the 2nd one.

```
//spwnPipeFindWC.js
const { spawn } = require('child_process');
const find = spawn('find', ['.', '-type', 'f']);
const wc = spawn('wc', ['-l']);

process.stdin.pipe(wc.stdin);

wc.stdout.on('data', (data) => {
  console.log(`child stdout:\n${data}`);
});
```

Shell syntax, exec(), & execFile() 04.08.2018

By default the spawn method does not create a shell to execute the command that's passed to it. Making it more efficient than the exec method. (which does create a shell) * Another major difference in exec vs spawn -> exec method buffers the command's generated output and passes the whole value to a callback function. Since exec uses the shell -> You can use the default pipe [1]

```
exec('find . -type f | wc -l', (err, stdout, stderr) => {
```

[1]

Here exec buffers the output and passes it to the stdout[2] - - - { which is the standard out for the callback } and the ouput which is to be printed. If there is an error in the command, the error first argument will be set. Exec is a good choice if you need to use shell syntax, and providing that the data is not too large. Because the data is buffered by exec. Use spawn when the data is large, because the data will be streamed with the standard I/O objects. You can make the child process inherit the standard I/O objects of its parents. This code will spawn the find command but inherit the process stdin and stdout, stderr.

```
const { spawn } = require('child_process');
const child = spawn('find', ['.', '-type', 'f'], {
    stdio: 'inherit'
});
```

When this code is executed, the data event will be triggered on the process

```
//spawnFind.js
const { spawn } = require('child_process');
const child = spawn('find', ['.', '-type', 'f'], {
  stdio: 'inherit'
});
```

~(process.stdout) ~ allowing the script to output the results out right away. * If you want to use the shell (04.09.2018) syntax and still get the advantage of streaming data that the spawn method gives you --▸ Then use the shell option, i.e set it to true. e.g., shell: true, This way spawn will still use the shell but it won't buffer the data. ~ Which is the best of both worlds! e.g.

```
const child = spawn('find . -type f | wc -l', {
      stdio: 'inherit',
      shell: true,
      cwd: '/Users/stan/UFOstuff'
    });
```

Another option would be to use env option, e.g.

```
const child = spawn('echo $HOME', {
      stdio: 'inherit',
      shell: true,
    });
```

(1) Now any command will have access to the current process environment by default

← you can add an empty object for env (like so env: {}, and now the command will not have access to the parent process env object anymore. Doing this overrides the behavior of the default [env process (1)] You can use the env option to manually define environment variables and access them in a child command. const child = spawn('echo $ans');

```
↳ env: { ans: 21 },
    });
```

Lets take a look at the detached option ϕ4.ϕ9.2ϕ18
This option allows the child process to run independantly of its parent process. (this behavior depends on the underlying OS)
e.g. on Windows, the detached process will have its own console window. While in linux the detached child process will be made the leader of a new process group session.
If the unref method is called on the detached process, the parent process can exit independantly of the child ~ which can be useful if the child is executing a long running process but to keep it running in the background the child's stdio configuration also has to be independant of the parent.

```
const { spawn } = require('child_process');
const child = spawn('node', ['timer.js'], {
  detached: true;
  stdio : 'ignore',
});
child.unref();
```

This will run a node script in the background, by detaching and ignoring its parent's stdio file descriptor. This way the parent can terminate, while the child can keep running in the background. You will see that the timer.js is still running. [$ ps -ef | grep timer] * Now say you want to execute a file without using a shell, the exec file is what you would use.

It basically works like the exec method, but without the shell, thus being more efficient. i.e exec file () --▸ but behaviors like I/O redirection and file clobbing are not supported. On windows some files cannot be (ϕ4.ϕ9.2ϕ18)
used, e.g. *.bat, *.cmd exec, or spawn with shell set to true will suffice. All the child process module functions have synchronise blocking versions that will wait until the spawn process exists. (useful for simplifying scripting tasks or any start up processing tasks.

The fork () function

The fork function is a variation of the spawn function. The biggest difference between spawn() and fork() is that a communication channel is established to the child process when using the fork () So you can use the send method on the forked process and the global process object, to exchange messages between the parent and forked processes with an interface similar to the event emitter module. In the parent.js module, we fork 'child.js' and listen on the message event, which will be triggered anytime the child uses process.send, which is being done every second. To pass down messages from the parent to the child -> we execute the send function on the forked object itself
e.g. forked.send ({ hola! : 'multiverse'}); , while in the child we listen to the message event i.e. on the global process object. process.on ('message', (msg) => {

```
//parent.js
const { fork } = require('child_process');
const forked = fork('child.js');

forked.on('message', (msg) => {
  console.log('Message from child', msg);
});
```

```
forked.send({ hola: 'multiverse' });
```

```
//child.js
process.on('message', (msg) => {
  console.log('Message from parent:', msg);
});

let counter = 0;

setInterval(() => {
  process.send({ counter: counter++ });
}, 1000);
```

Say you have an http server, that handles 04.10.2018
multiple endpoints. One of these endpoints is computationally expensive and will take a few seconds to complete, which we can simulate with this long for loop -> for (let i=0; i < 1e9; i++){. So when the ###/compute endpoint is requested the server will not be able to handle any other requests because the event loop is busy with the long for-loop operation. We should move this "heavy" computational job to another process, using fork. Move the whole job to a file and then within the node script fork that file. And use the message interface to communicate with that file. i.e the server and the forked process. So when we receive a request we will send a message to the forked process to start processing. * The forked process can read the message, using the message event on the global process object. In the file we will call the "long" operation. The operation will return its result to the parent process via process.send(); * In the parent process we just listen to the message event on the forked process itself. Once the forked file is complete, the parent will be listening and will send that result to the requesting http process. This code is limited by the number of processes we can fork. But when you execute the code you will notice that the server is not blocked at all. The server is able to take further requests. Node's cluster module is based on the idea child process forking and load balancing the requests on the many forks you can create.

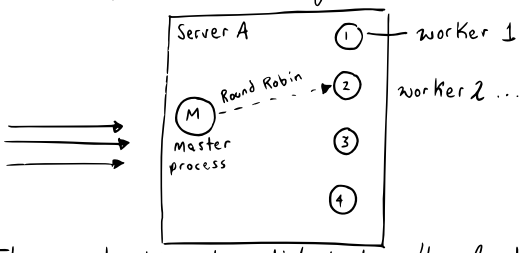
Cluster Module - is used to enable load-balancing over an environment of multiple CPU cores. It's based on the fork function. It allows you to fork your main operation (application) process over as many times as you have cpu cores. ★ It will then take over and load-balance all requests to the main process across all forked processes. The cluster module is node's helper for you to implement the "Cloning" Scalability Strategy *but only on one machine* So if you have a big machine with a lot of resources or when it's easier to add more resources to one machine rather than adding new machines. Then the cluster module is a great choice for implementing the cloning strategy. Most machines have multiple - 8 cores. Even if you are not worried about load on your node Server, you should enable the cluster module anyway. This will increase your scalability and fault tolerance. It's a simple step - when using a process manager like pm2 - all you have to do is provide an argument to the launch command. We will look at the native implementation of the cluster module.

04.10.2018

The first thing that occurs when using the cluster module.

#1 We create a master process, forks a number of worker processes and manages them. Each worker process represents an instance of the application you are scaling. All incomming requests are handled by the master process. The master process decides which worker process should handle incoming requests. The master process job is easy. It just uses a round robin algorithm to pick a worker process. [this is enabled on default, except on windows) it can be globally modified for the load balancing to be handled by the operating system itself.

04.10.2018



The round robin algo distributes the load evenly across all available processes on a rotational basis. The first request is forwarded to the first worker process - the 2nd, to the next worker process and so on... And when the end of the list is reached the algo starts over again. There are more featured algorithms allow assigning priorities, selecting the least loaded server or the one with the fastest response rate ... etc.

Lets Load Balance a simple Http Server!

```
//HttpServer.js
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i=0; i<1e7; i++); //to simulate CPU work
  res.end(`Work handled by process ${pid}`);

}).listen(8080, () => {
```

```
  console.log(`Beginning process start ${pid}`);
});
```

Apache AB on a Windows machine

```
C:\xampp\apache\bin>ab -c200 -t10 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Finished 1442 requests


Server Software:
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        28 bytes

Concurrency Level:      200
Time taken for tests:   10.002 seconds
Complete requests:      1442
Failed requests:        0
Total transferred:      148526 bytes
HTML transferred:       40376 bytes
Requests per second:    144.17 [#/sec] (mean)
Time per request:       1387.246 [ms] (mean)
Time per request:       6.936 [ms] (mean, across all concurrent requests)
Transfer rate:          14.50 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1  12.6      0     478
Processing:  1100 1239 303.8   1133    2479
Waiting:     1100 1239 303.8   1133    2479
Total:       1100 1240 304.0   1133    2480

Percentage of the requests served within a certain time (ms)
  50%   1133
  66%   1137
  75%   1146
  80%   1154
  90%   1678
  95%   2095
  98%   2329
```

```
  99%   2403
 100%   2480 (longest request)

C:\xampp\apache\bin>
```

$ ab -c 200 -t 10 http://localhost:8080/ 04.10.2018
here we will load the server with 200 concurrent connections, for 10 seconds. This little node server was able to handle 144.17 requests per second. Results will vary. But we will use this result as our reference measurement.

04.21.2018

Now we will run ab on cluster.js
As you can see that cluster.js is able to handle 315.26 requests per second. In cluster.js we called the cluster.fork() method, which will create as many number of workers as there are cpus. Which will take advantage of all the available processing power. When the cluster.fork() line is executed from the master process, the current main module, cluster.js is run again but this time in worker mode, with the isMaster flag set to false. We will require our http Server.

★ In this load test the requests per minute is at 298

```
//cluster.js
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
  const cpus = os.cpus().length;

  console.log(`Forking for ${cpus} CPUs`);
  for (let i = 0; i <cpus; i++) {
    cluster.fork();
  }
} else {
  require('./HttpServer');
}
```

```
C:\xampp\apache\bin>ab -c200 -t10 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Finished 2989 requests


Server Software:
Server Hostname:        localhost
```

```
Server Port:            8080

Document Path:          /
Document Length:        28 bytes

Concurrency Level:      200
Time taken for tests:   10.003 seconds
Complete requests:      2989
Failed requests:        0
Total transferred:      307867 bytes
HTML transferred:       83692 bytes
Requests per second:    298.82 [#/sec] (mean)
Time per request:       669.296 [ms] (mean)
Time per request:       3.346 [ms] (mean, across all concurrent requests)
Transfer rate:          30.06 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.5      0      24
Processing:    85  645 107.6    661     795
Waiting:       81  644 107.7    660     795
Total:         85  645 107.6    661     795

Percentage of the requests served within a certain time (ms)
  50%    661
  66%    685
  75%    696
  80%    703
  90%    735
  95%    766
  98%    776
  99%    779
 100%    795 (longest request)

C:\xampp\apache\bin>
```

Broadcasting Messages to all Workers 04.22.2018

The communication between the master process and workers is pretty simple. The cluster module is using the child process fork api - which means we are using communication channels between the worker and master. We can access the list of worker objects using cluster.workers e.g. console.dir(cluster.workers, {depth: 0}); which is an object that holds references to the workers each object holds a reference to each worker and can be used to read information on those workers. To broadcast a message to all the workers -> create a loop, iterate through an object values array. Node has this -> Object.values(cluster.workers).forEach(worker => { });

```
//BroadCast/cluster.js
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
  const cpus = os.cpus().length;

  console.log(`Forking for ${cpus} CPUs`);
  for (let i = 0; i <cpus; i++) {
    cluster.fork();
  }

  console.dir(cluster.workers, { depth: 0 });
  Object.values(cluster.workers).forEach(worker => {
    worker.send(`Hi Worker ${worker.id}`)
  });
} else {
  require('./HttpServer');
}
```

Terminal Output for BroadCasting messages via fork api in cluster.js

```
$ node cluster.js
Forking for 4 CPUs
{ '1': [Object], '2': [Object], '3': [Object], '4': [Object] }
Beginning process start 5980
Beginning process start 9816
Beginning process start 7144
Beginning process start 3212
```

```
//BroadCast/HttpServer.js broadcasting messages!
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i=0; i<1e7; i++); //to simulate CPU work
  res.end(`Work handled by process ${pid}`);

}).listen(8080, () => {
  console.log(`Beginning process start ${pid}`);
});

process.on('message', msg => {
```

```
  console.log(`Message from master: ${msg}`);
});
```

messages are not in order

```
$ node cluster.js
Forking for 4 CPUs
{ '1': [Object], '2': [Object], '3': [Object], '4': [Object] }
Message from master: Hi Worker 1
Beginning process start 428
Message from master: Hi Worker 2
Message from master: Hi Worker 3
Beginning process start 7192
Message from master: Hi Worker 4
Beginning process start 8336
Beginning process start 7092
```

We're gonna make a more practical app !!! 04.22.2018

The product owner has asked us to have the server reply with the number of users we have created in our database.

Acceptance criteria: A function (mock) that returns a number of users that is in the database, and double its value every time it's called.

So the first thing we should do, is cache our db requests as to avoid multiple requests to the db. So let's cache the call at ten seconds. We don't want each worker to make their own request per ten seconds. We will have the master process do one request, which will communicate to each worker (on my machine -> 4 cpus) the new value i.e the user count using the communication interface. * In the process master mode, create a function named update workers * And use a loop to broadcast the updated value (this time, instead of a text message) and then add an invoke workers function that will update, every ten seconds with a timer. So now every ten seconds all workers will receive a new value over the process communication channel. In the Http server code, we would use this value with the message variable ${msg}

We can create a variable, that is global to the Http Server module & we can use it anywhere we want. We will also 'respond using the same variable. 04.22.2018

* The user count variable is controlled by the message interface The master process * i.e cluster.js - resets the value of the user count variable every 10 seconds by invoking the mock db call.

```
//cluster2.js
const cluster = require('cluster');
const os = require('os');

//make db call (mock)
const usersDB = function() {
  this.count = this.count || 5;
  this.count = this.count * this.count;
  return this.count;
}
if (cluster.isMaster) {
  const cpus = os.cpus().length;

  console.log(`Forking for ${cpus} CPUs`);
  for (let i = 0; i <cpus; i++) {
    cluster.fork();
  }

const updateWorkers = () => {
  console.dir(cluster.workers, { depth: 0 });
  const usersCount = usersDB();
  Object.values(cluster.workers).forEach(worker => {
      worker.send({usersCount}); //broadcast number of users

  });
};

setInterval(updateWorkers, 1000); //update workers on 10 second interval, ever
} else {
  require('./HttpServer2');
}
```

```
//HttpServer2.js
const http = require('http');
const pid = process.pid;

let usersCount; //variable is set at the global level

http.createServer((req, res) => {
  for (let i=0; i<1e7; i++); //to simulate CPU work
  res.write(`Work handled by process ${pid}`); //changed write to output both
  res.end(`Users: ${usersCount}`); //here the userCounts variable is controlle
}).listen(8080, () => {
  console.log(`Beginning process start ${pid}`);
});
```

```
process.on('message', msg => {
//  console.log(`Message from master: ${msg.toString()}`);
  usersCount = msg.usersCount;
});
// the master process in cluster2.js will reset the userscount every 10 secs t
```

Availability and zero downtime Restarts 04.23.2018
When running a single node instance and the instance crashes there will be downtime. You can increase the availability of the application by increasing its instances. If for example a unexpected error occurs on the server process (relative) to the master. i.e a worker process exits. You can notify the master by using the exit event on the cluster object. So we can register a handler for that event.

cluster.on('exit', (worker, code, signal) => {
↳ here you would fork a new worker process, i.e when any worker process exits. You should add the condition here to make sure the worker process actually crashed and not manually disconnected or killed by the master process. e.g. the master decides that were using too much resource based on load patterns it detects and it needs to kill some of the worker processes. (use .kill() or .disconnect() on any worker)
↳ if (code !== 0 && !worker.exitedAfterDisconnect) {
* here the code will not fork a new worker for that exact case. * We can also measure the availability using ab (apache benchmark) or BeesWithMachineGuns

In our http server we created a setTimeout() 05.15.2018 to simulate a crash. And we are able to notify the master using the exit event on the cluster object. So we register a handler on that event and within that block -> fork a new worker process.
↳ cluster.on('exit', (worker, code, signal) => {
}
* So this way we start a new worker process any time a worker crashes. It is also important to add a logical condition to the handler to diff between a 'crash' and a legit termination by the master process. e.g the master proc may decide that too much resources are being consumed by a worker proc. The master may then shutdown the process based on the load patterns it detects. In such a case use the dot kill or dot disconnect. In which the code below will set the disconnect flag to true.
e.g.
if (code !== 0 && !worker.exitedAfterDisconnect) {
}
When you run the code -> you will see that some workers crash.

```
//MasterCluster.js
const cluster = require('cluster');
const os = require('os');
```

```
if (cluster.isMaster){
  const cpus = os.cpus().length;
  for (let i = 0; i < cpus; i++) {
    cluster.fork();
   }

   cluster.on('exit', (worker, code, signal) => {
     if (code !== 0 && !worker.exitedAfterDisconnect) {
       console.log(`Worker ${worker.id} has crashed. ` +
         'Spinning up a new worker...');
      cluster.fork();
     }
  });
} else {
  require('./HttpServer');
}
```

```
//Availability/HttpServer.js
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i=0; i<1e7; i++); //to simulate CPU work
  res.end(`Work handled by process ${pid}`);

}).listen(8080, () => {
  console.log(`Beginning process start ${pid}`);
});

setTimeout(() => {
  process.exit(1) // kill at random

}, Math.random() * 10000);
```

Terminal Output

```
$ node MasterCluster.js
Beginning process start 9692
Beginning process start 7092
Beginning process start 5532
Beginning process start 7192
Worker 4 has crashed. Spinning up a new worker...
Beginning process start 6472
Worker 2 has crashed. Spinning up a new worker...
Beginning process start 10716
```

```
Worker 6 has crashed. Spinning up a new worker...
Beginning process start 12180
Worker 1 has crashed. Spinning up a new worker...
Beginning process start 12204
Worker 8 has crashed. Spinning up a new worker...
Beginning process start 796
Worker 5 has crashed. Spinning up a new worker...
Beginning process start 10224
Worker 9 has crashed. Spinning up a new worker...
Beginning process start 10688
Worker 3 has crashed. Spinning up a new worker...
Beginning process start 11208
Worker 12 has crashed. Spinning up a new worker...
```

AB terminal output for MasterCluster.js

```
C:\xampp\apache\bin>ab -c200 -t10 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
apr_socket_recv: An existing connection was forcibly closed by the remote
host.
  (730054)
Total of 482 requests completed
```

But what if the dev wishes to restart all 05.15.2018
the workers. In this case we have multiple workers running
The best approach would be to restart them one by one.
To allow other workers to continue to serve requests while
one worker is restarting. The first thing we should
do in the master cluster is to add a signal
listener e.g >

process.on('SIGUSR2', () => {

This could be a good time to brush up on UNIX signals

$ kill -SIGUSR2 PID

here just send a kill signal on the process ID
, this way the master process will not be killed.
* Do not use the SIGUSR1, 'cause node's debugger
uses that one. You can also watch a process PID file and
see if there's a remove event.
So the first thing you should do -> create a reference
to all current workers using the clusters.workers object
- We can now store the workers in an array.
const workers = Object.values(cluster.workers);
- We will create a restart worker function that receives
the index of the worker to be restarted. * This way
we can call the workers in sequence - and restarting

const restartWorker = (workerIndex) => { 05.16.2018
}; ↗ So this function will now call itself when it's ready
to start another worker. To start the sequence we
call the function with Index 0 e.g. restartWorker(0);
Inside the restartWorker function we get a reference
to the worker to be restarted, using the workers array
and the received index, and since this function will be
called recursively to form a sequence -> add a stop cond-
ition { when there is no longer workers then return }
and disconnect the worker. Before starting another worker
you will need to fork a new worker to replace the
one that was just disconnected. You should be able to
use the exit event on the worker itself to be
able to fork a new worker when the other one exits
To do this we should use the disconnect flag, e.g.
-> if (!worker.exitedAfterDisconnect) return;

↗ if this flag is not true, this exit was caused by some-
thing other than disconnect, in that case, just return.
But remember the fork process is not synchronous -
So you can't just start a new worker (fork) after the
but instead you should be able to monitor the listening
event or the forked worker. -> which tells us that this
worker is disconnected and ready

When you get this event - you can safely, 05.16.2018
restart the next worker in sequence, pretty much
all you need for 0 downtime to work with a cluster
module. * make sure to console.log the master process
ID

```
//zeroDwnTime/MstrCluster.js
const cluster = require('cluster');
```

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

if (cluster.isMaster){
  const cpus = os.cpus().length;
  for (let i = 0; i < cpus; i++) {
    cluster.fork();
   }

   cluster.on('exit', (worker, code, signal) => {
     if (code !== 0 && !worker.exitedAfterDisconnect) {
       console.log(`Worker ${worker.id} has crashed. ` +
         'Spinning up a new worker...');
      cluster.fork();
     }
  });

process.on('SIGUSR2', () => {
  const workers = Object.values(cluster.workers);

  const restartWorker = (workerIndex) => {
    const worker = workers[workerIndex];
    if (!worker)
      return;

    worker.on('exit', () => {
      if (!worker.exitedAfterDisconnect) //set flag if true, fork new worker
      console.log(`Exited process ${worker.process.pid}`)
      cluster.fork().on('Listening ... ', () => {
        restartWorker(workerIndex + 1);
      });
    });
      worker.disconnect();
  };

  restartWorker(0);
});

} else {
  require('./HttpServer');
}
```

PM2 - is cool!

just use pm2 reload all for zero downtime reload

https://github.com/Unitech/pm2

(3) BDD with Node

Ø3 Ø2.11.2Ø19

TDD & BDD with Mocha

TDD, is one of the main agile development techniques. TDD is increased quality of code, faster dev time, resulting from greater confidence (on the dev's part) & improved bug detection.

Certain parts such as standalone services Rest Apis should be tested thoroughly by the TDD. UI/user experience should be tested with selenium/puppeteer.

Think about testing as the time saver. With proper tests in place & a bit of time spent on writing them, devs save time in the long term. The longer the long term, the more the payoff. It's not uncommon for a good module to have (2-3x) more tests then the code itself. It's not overkill, its sound strategy.

BDD → The behavior-driven development is based on TDD. It differs from TDD in language/interface, which is more natural. BDD is the preferred way of writing tests. Ø2.11.2Ø19

e.g. a BDD interface would be 'expect.'

e.g. expect(response.status).to.equal(2Ø1)

while TDD → with assert →

assert.equal(response.status, 2Ø1)

Objectives →

- Install Mocha TDD with assert
- BDD with expect.js
- Project: Write the BDD for a blog!

```
$ npm -i -g mocha@4.Ø.1
```

You can have a seperate version of Mocha for each project by pointing to the local version of Mocha

```
$ ./node_modules/.bin/mocha test Ø1
```

The main process of TDD summed up in 3 steps. 02.11.2019

1. Implement a test
2. Implement the code to make the test pass
3. Verify that the test passes / repeat the cycle.

BDD is a specialized version of TDD that specifies what needs to be unit-tested from the perspective of business requirements
use mocha → you get
- reporting
- Asynch support
- Rich configurability
- Notifications
- Debugger support
- common interface with before, after hooks File watcher support

Mocha Hooks 02.11.2019
a hook is some logic, typically a function with a few statements.

Mocha has hooks to explore different parts of suites -- before the whole suite, before each test, etc. In addition there are after() & afterEach() hooks. → These will be used for test cleanup, cleanup of database.

All hooks support asynchronous modes. The same is true for tests as well. e.g. 1

```
// Code Stub 01
```

But as soon as you add a done parameter to the test's function, the test case waits for the Http request to come back. This is by calling the done() to let Mocha (Jasmine/Jest) understand that if done() is not implemented then the test will time out, because the test runner will not know when its done. 2

```
// Code Stub 02
```

Test cases (describe) can be nested 02.11.2019
inside other test cases, & hooks such as
before & beforeEach can be mixed in with
different test cases on different levels.
Nesting 'describe' constructions is a good
idea in large test files.
Devs who wish to skip a testcase i.e.
(describe.skip()) or it.skip() or make
a test run exclusive → describe.only()
or describe.only().
An alternate to the BDD interfaces 'describe',
it, before,) & others, Mocha does also support
more traditional' TDD interfaces:

- suite : Analogous to 'describe'
- test : Analogous to 'it'
- setup : Analogous to 'before'
- teardown : Analogous to 'beforEach'
- suiteTeardown : Analogous to 'afterEach'

BDD with Expect 02.11.2019

'Expect is one of the BDD languages. Its very
popular because its syntax allows for chaining.
It has "better" features vs assert.
The purpose is for better understanding/human
readability. The 3 amigos can read them.
There are 2 flavors of Expect to choose
from:

- 1 Standalone : Install as a expect.js
  module
- 2 Chai : Install as part of the chai
  library ~ the latter is recommended.

must have your package.json already there
npm init -y

```
$ npm install chai@4.1.2 --save-exact
```

use import → const expect = require('chai').expect
use ES6 destructing assignment

```
const {expect} = require ('chai')
```

Each assert assertion can be rewritten 02.11.2019
with Expect. The idea is to use expect()
& pass an object we are going to test to
it as an argument, e.g. → expect(
(current.length).

Then use the properties & methods by chaining them
in some resemblance to the English language.
e.g. expect(current.length).to.equal(30)
To write this in Chai BDD style → use to.be.true,
equal & to.equal
3

```
// Code Stub 03
```

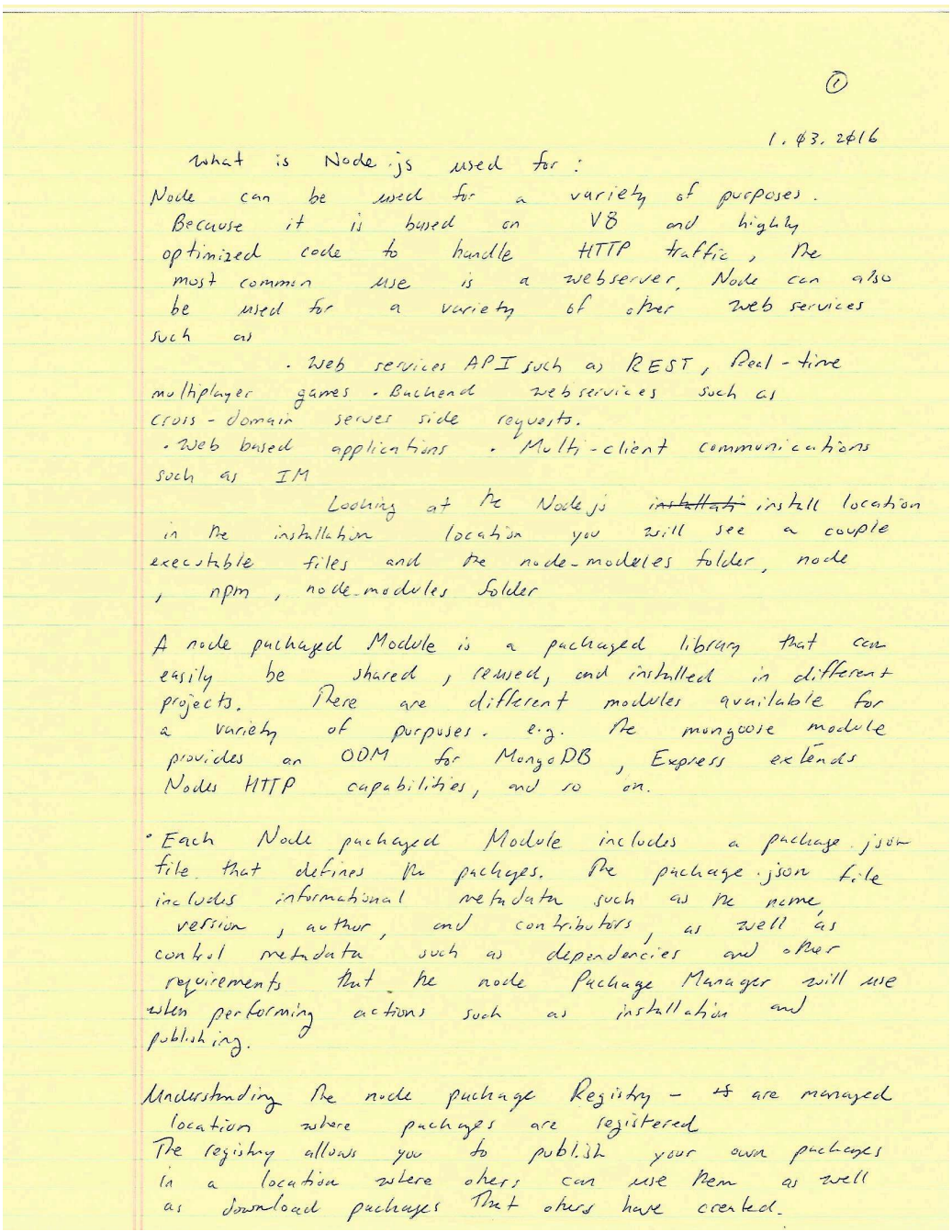
The expect.js is not 100% compatible with
chai.expect

$ npm install expect.js

replace the chai expect const {expect}
= require('chai')

```
// Code Stub 04
```

Old Notes (scanned)



①

1.03.2016

What is Node.js used for:

Node can be used for a variety of purposes. Because it is based on V8 and highly optimized code to handle HTTP traffic, the most common use is a webserver, Node can also be used for a variety of other web services such as

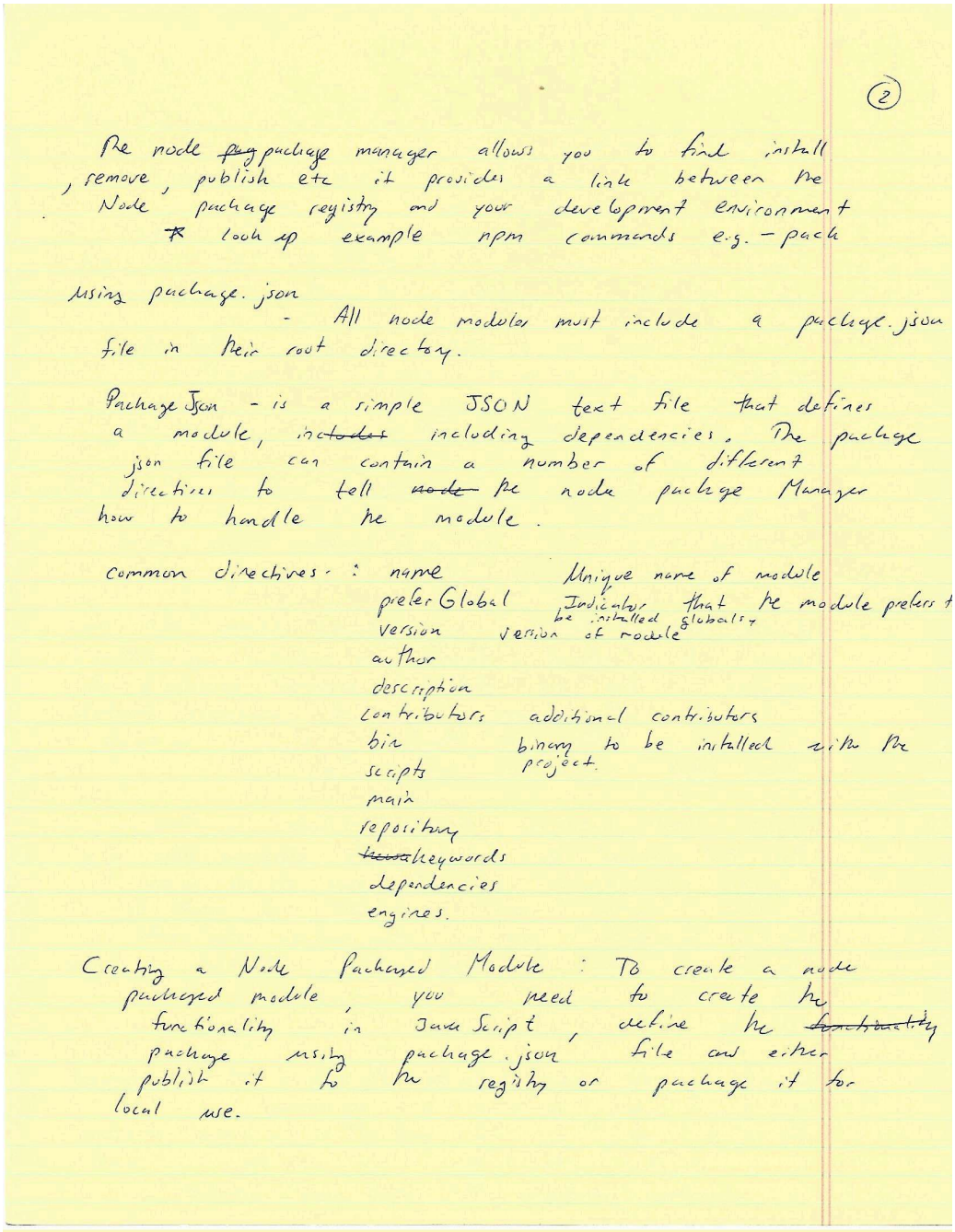
· Web services API such as REST, Real-time multiplayer games · Backend web services such as cross-domain server side requests.
· Web based applications · Multi-client communications such as IM

Looking at the Node.js ~~installati~~ install location in the installation location you will see a couple executable files and the node_modules folder, node, npm, node_modules folder

A node packaged Module is a packaged library that can easily be shared, reused, and installed in different projects. There are different modules available for a variety of purposes. e.g. the mongoose module provides an ODM for MongoDB, Express extends Nodes HTTP capabilities, and so on.

· Each Node packaged Module includes a package.json file that defines the packages. The package.json file includes informational metadata such as the name, version, author, and contributors, as well as control metadata such as dependencies and other requirements that the node Package Manager will use when performing actions such as installation and publishing.

Understanding the node package Registry - is are managed location where packages are registered
The registry allows you to publish your own packages in a location where others can use them as well as download packages that others have created.



(2)

The node ~~pkg~~ package manager allows you to find, install, remove, publish etc it provides a link between the Node package registry and your development environment

* look up example npm commands e.g. - pack

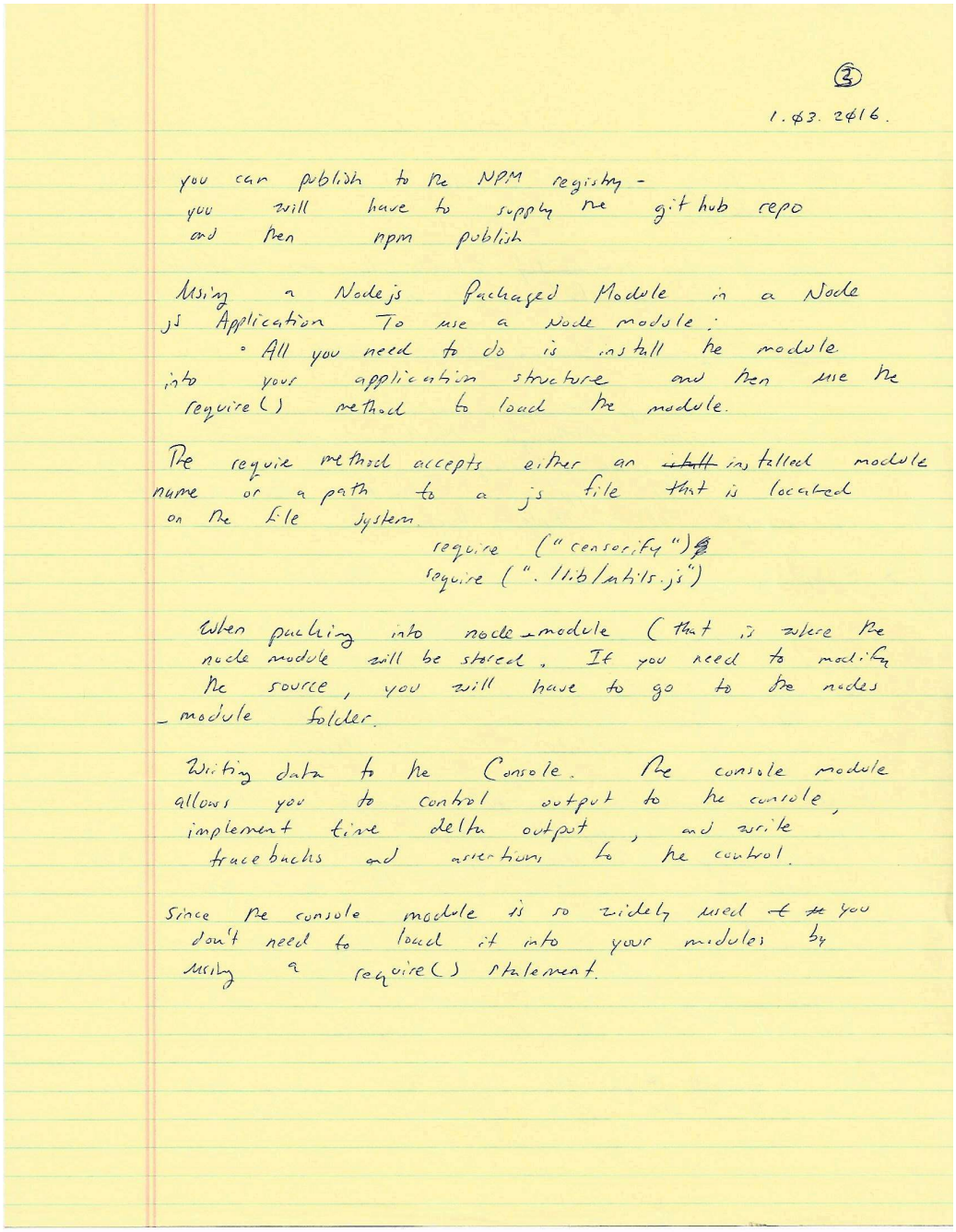
using package.json

- All node modules must include a package.json file in their root directory.

Package Json - is a simple JSON text file that defines a module, ~~includes~~ including dependencies. The package json file can contain a number of different directives to tell ~~node~~ the node package Manager how to handle the module.

common directives: :
- name — Unique name of module
- preferGlobal — Indicates that the module prefers to be installed globally
- version — version of module
- author
- description
- contributors — additional contributors
- bin — binary to be installed with the project
- scripts
- main
- repository
- ~~keywords~~ keywords
- dependencies
- engines.

Creating a Node Packaged Module: To create a node packaged module, you need to create the functionality in Java Script, define the ~~functionality~~ package using package.json file and either publish it to the registry or package it for local use.



③

1.03.2016.

you can publish to the NPM registry - you will have to supply the git hub repo and then npm publish

Using a Nodejs Packaged Module in a Node js Application To use a Node module:

- All you need to do is install the module into your application structure and then use the require() method to load the module.

The require method accepts either an ~~install~~ installed module name or a path to a js file that is located on the file system.

```
require ("censorify")
require (". /lib/utils.js")
```

When pushing into node_module (that is where the node module will be stored. If you need to modify the source, you will have to go to the nodes_module folder.

Writing data to the Console. The console module allows you to control output to the console, implement time delta output, and write tracebacks and assertions to the control.

Since the console module is so widely used ~~t~~ ~~the~~ you don't need to load it into your modules by using a require() statement.

Node Style Guide >>>

https://github.com/felixge/node-style-guide