
JS

• Node JS the hard parts

to recap, JS stores data in memory, & executes code...
We have our execution context, global memory & call stack to keep our place. functions executed open "mini-programs" or execution contexts

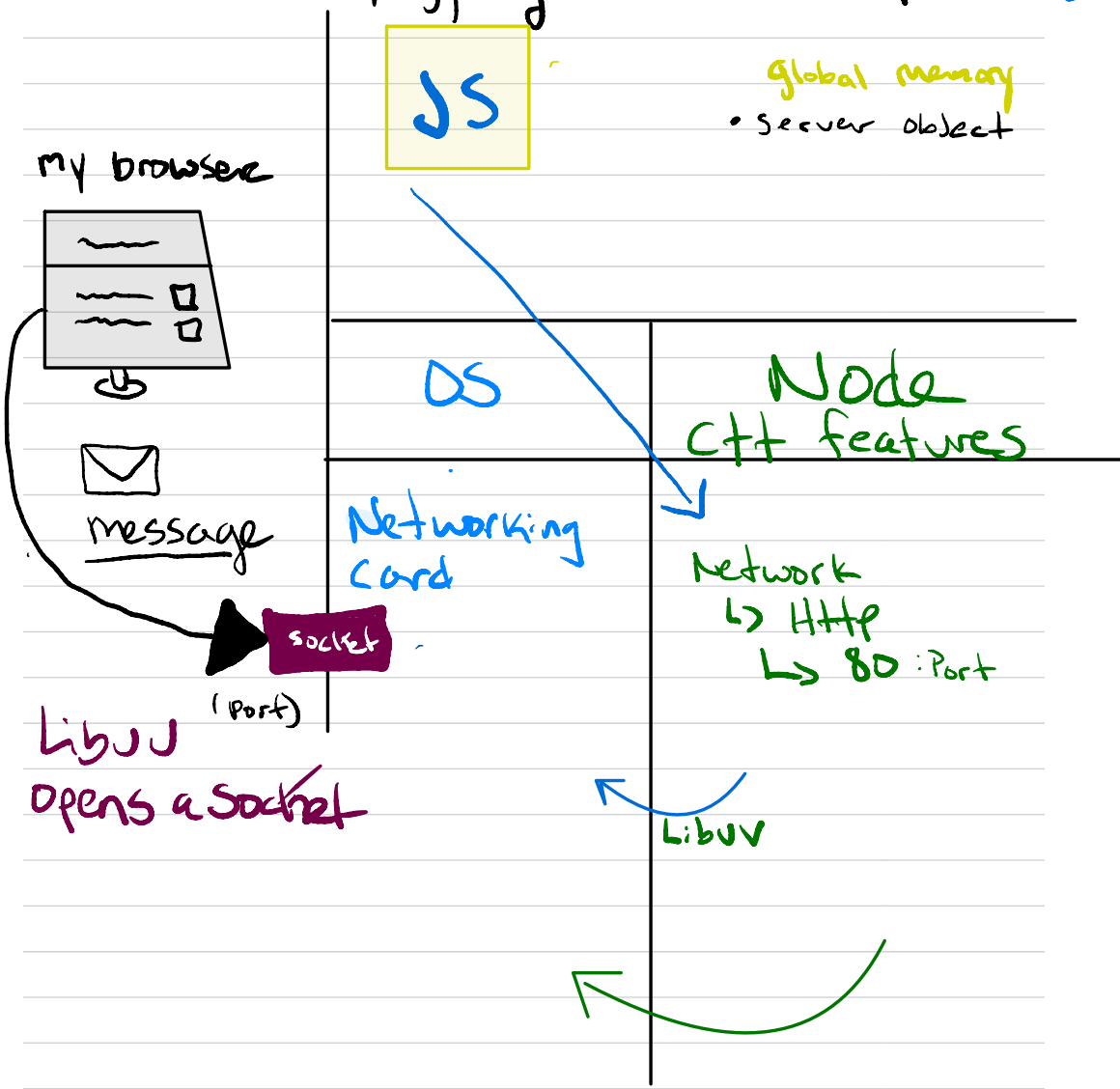
See JavaScript, the hard parts for more info on How this works

{ Node puts the ( ) on our functions for us }

1. Saves data

2. Uses that data by running functionality on it

3. Built-in labels, triggering ctt to use comp. internals



Node auto-runs the code (function) *for us* when a request arrives from a user

```
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    functionsToSetOutgoingData.end("Welcome to Twitter!")
}

const server = http.createServer(doOnIncoming)
server.listen(80)
```

1. We don't know when the inbound request would come - we have to rely on Node to trigger JS code to run
2. JavaScript is single-threaded & synchronous. All slow work (e.g. speaking to a database) is done by Node in the background (more on this later)

two things to running a function.

Parans on the end, insert data

Node the function, adding paranthesis

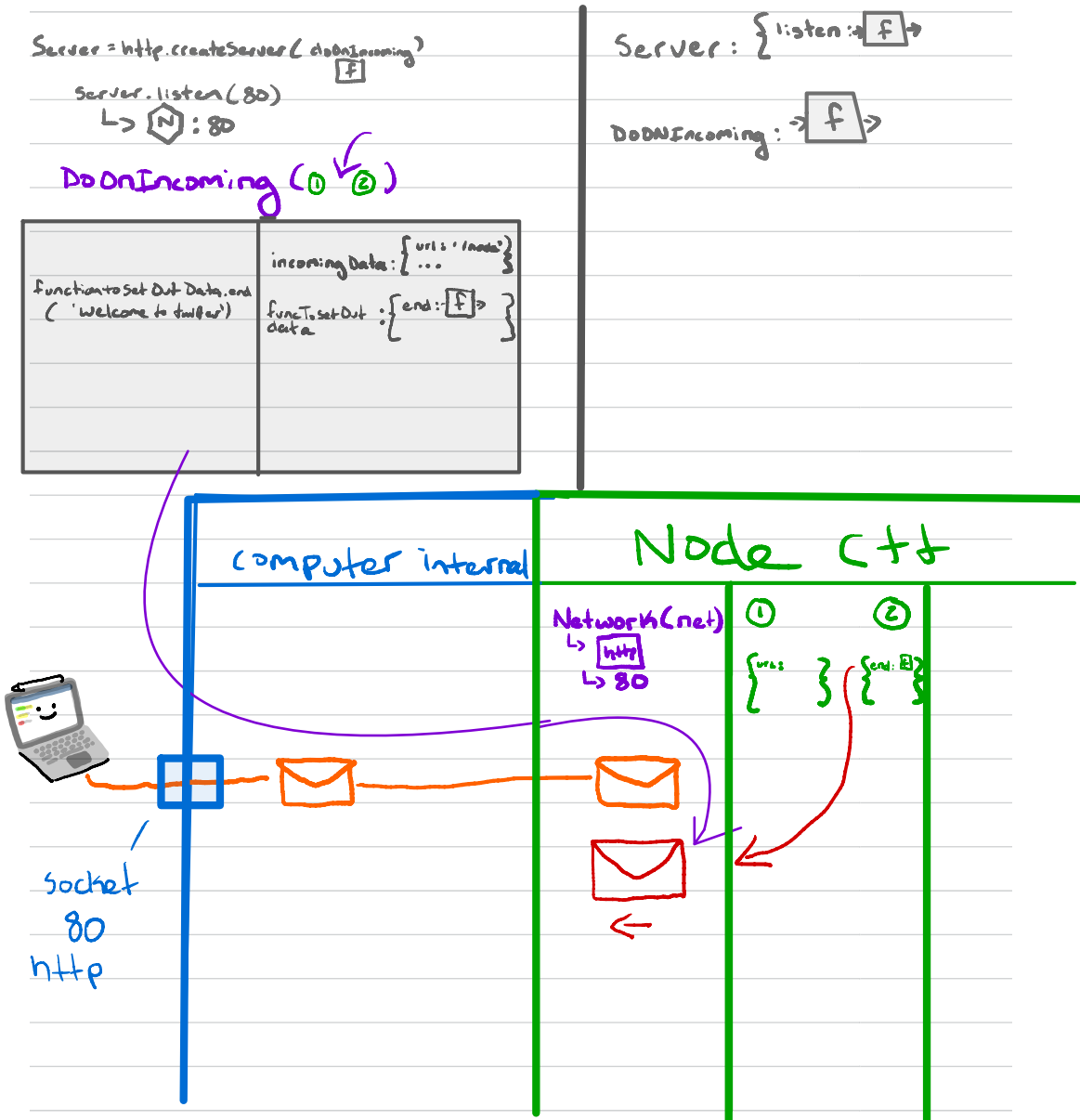
+ inserting the incoming data as an argument

Messages Are sent in http format, the "protocol" for browser/server interaction

Request line GET

Headers

Body (optional)



Node Auto - runs do on incoming, adding parenthesis & inserting arguements ... it Auto - creates two objects, & inserts them into the function

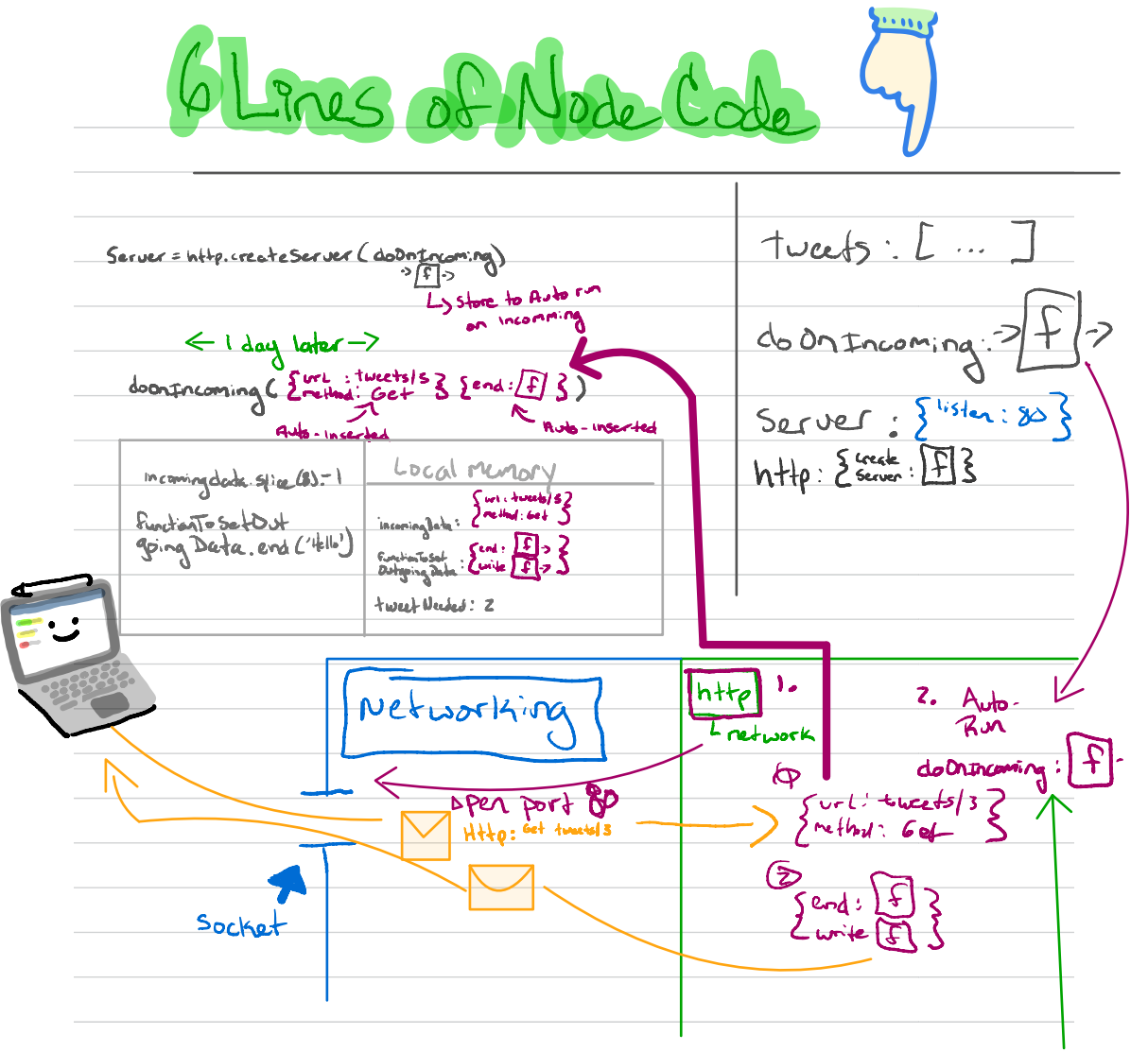
the Node http module

Auto creates two objects,

1. Access to the incoming data
2. functions to send outgoing data. via the .end() function

Built in Javascript lable, which sets up the Node c++ feature that has Access to the background features, we add a Javascript function that Node will Auto-run whenever the background feature gets Activity. Node adds 2 objects that are auto-created.

6 Lines of Node Code

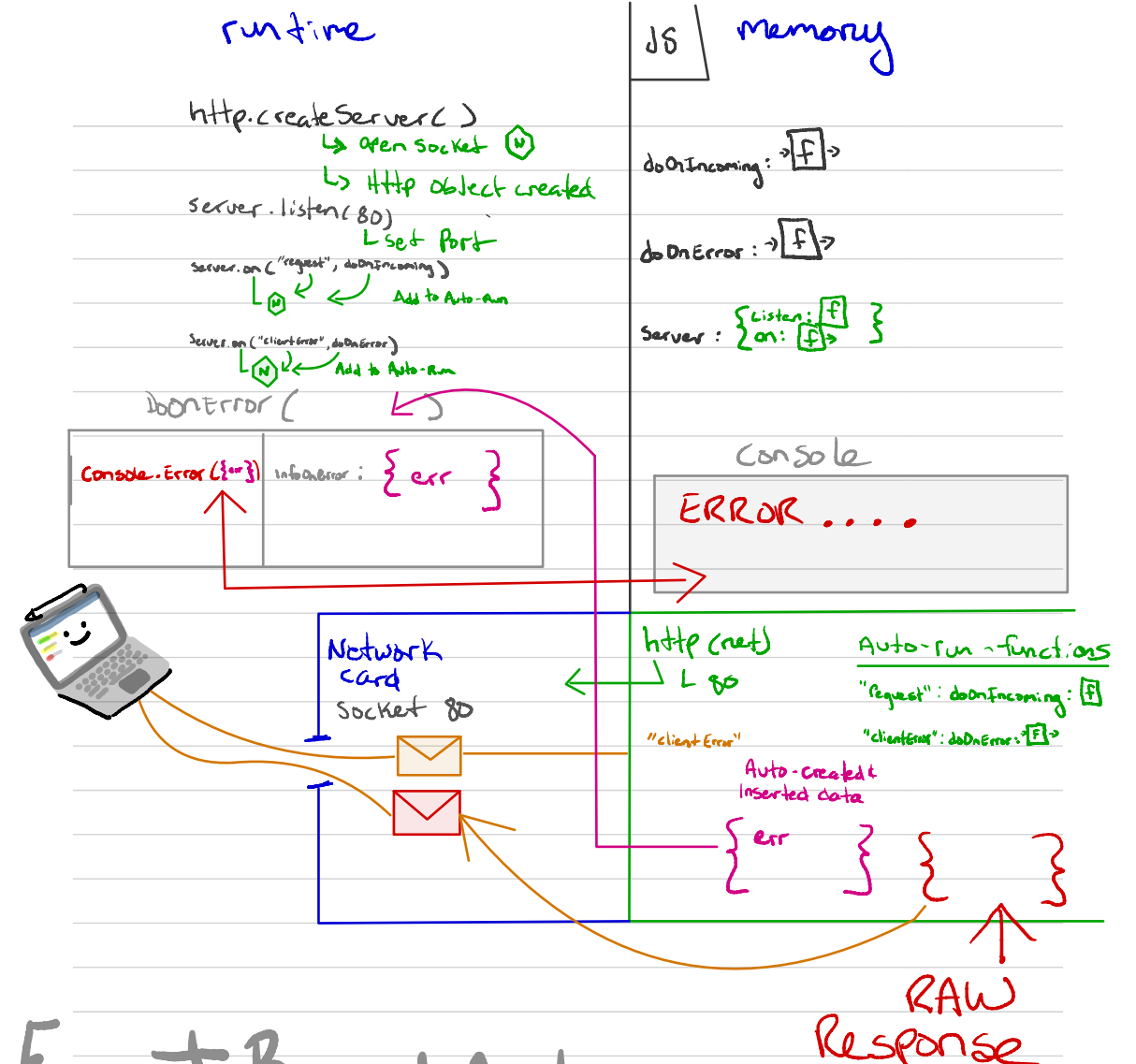


Messages are sent in HTTP format - The 'protocol' for browser-server interaction

HTTP message: Request line (url, method), Headers, Body (optional)

```
const tweets = ["Hi", "😂", "Hello", "👋", "☠️"]
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    const tweetNeeded = incomingData.url.slice(8)-1
    functionsToSetOutgoingData.end(tweets[tweetNeeded])
}

const server = http.createServer(doOnIncoming)
server.listen(80)
```

function to Auto-run on incoming traffic



Event Based Arch.

Node will automatically send out the appropriate event depending on what it gets from the computer internals (http message or error 😭)

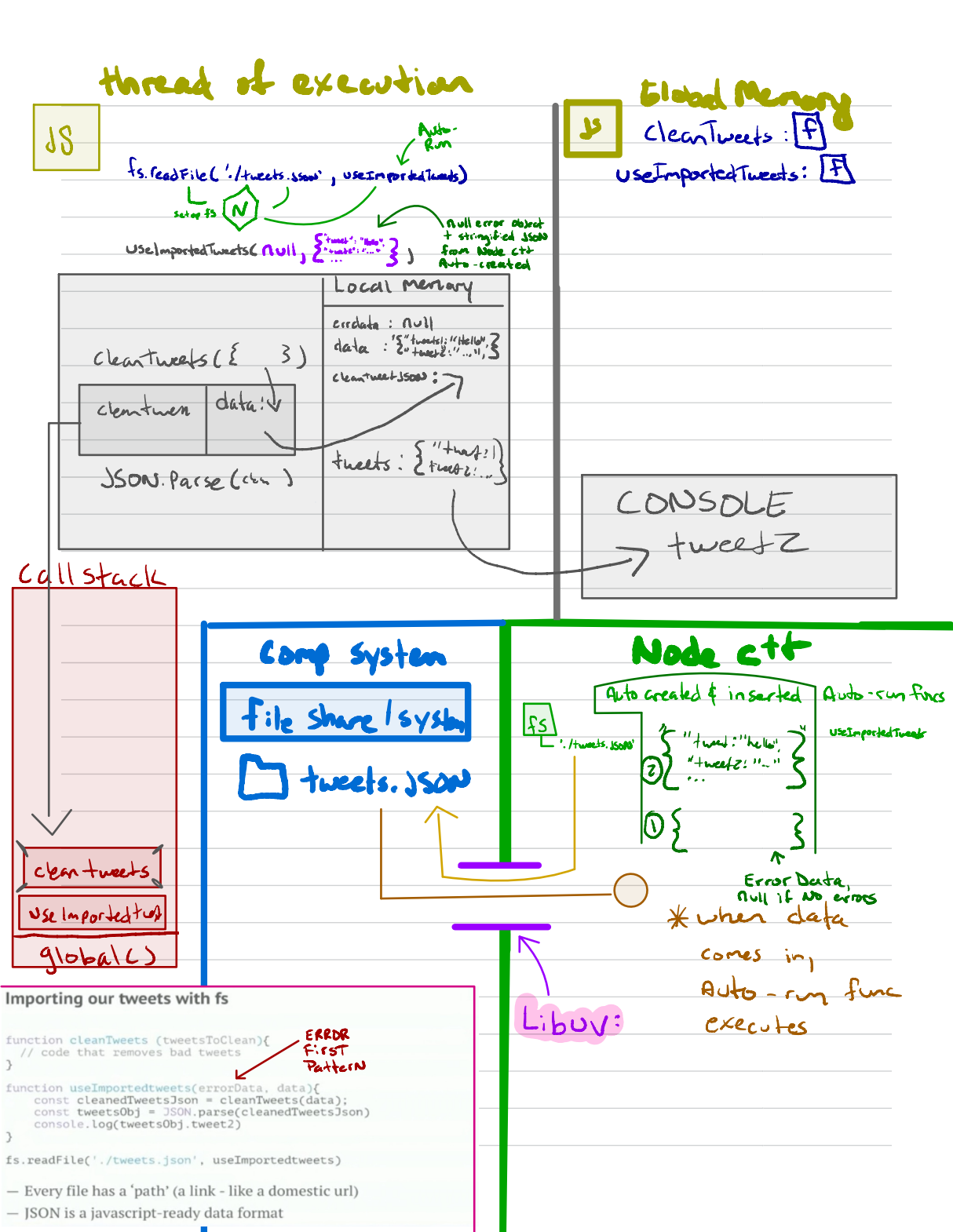
```
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    functionsToSetOutgoingData.end("Welcome to Twitter")
}

function doOnError(infoOnError){
    console.error(infoOnError)
}

const server = http.createServer();
server.listen(80)

server.on('request', doOnIncoming)
server.on('clientError', doOnError)
```

Node - http (net)
JS - object w/ methods to edit http.

EVENT System

Node, upon setting up server, has "Request" & "client error" Available. They're Node keywords

Events like "Request" or "client error" Arg messages, those messages get flashed out on "Activity"


thread of execution
JS
Auto-Run
fs.readFile('./tweets.json', useImportedTweets)
setup fs
useImportedTweets(null, {"tweet":"hello", "tweet2":"..."})
null error object + stringified JSON from Node ctr Auto-created
Global Memory
cleanTweets : f
useImportedTweets : f
Local Memory
errdata : null
data : '{"tweet":"hello", "tweet2":"..."}'
cleanTweetsJSON :
cleanTweets({ })
cleantwen
data:
JSON.Parse(clean)
tweets : {"tweet":"hello", "tweet2":"..."}
CONSOLE
tweet2
Call Stack
cleantweets
useImportedtwt
global()
Comp System
file share / system
tweets.JSON
Node ctr
Auto created & inserted
Auto-run fncs
fs
'./tweets.JSON'
useImportedTweets
{"tweet":"hello", "tweet2":"..."}
2
1
Error Data, null if no errors
* when data comes in, Auto-run func executes
Libuv:
Importing our tweets with fs
function cleanTweets (tweetsToClean){
// code that removes bad tweets
}
ERROR First Pattern
function useImportedtweets(errorData, data){
const cleanedTweetsJson = cleanTweets(data);
const tweetsObj = JSON.parse(cleanedTweetsJson)
console.log(tweetsObj.tweet2)
}
fs.readFile('./tweets.json', useImportedtweets)
— Every file has a 'path' (a link - like a domestic url)
— JSON is a javascript-ready data format

- things to Note,
  - fs doesn't bring in stringified JSON, instead, it brings in data of type Buffer. 0 & 1's
  - JSON.parse() will stringify & parse the buffer, converting it to an object.
  - Another issue, we have to wait for the entire Buffer to finish, then clean the tuzzets

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

const bufferFileData = fs.readFileSync('./test.json');

console.log(bufferFileData) //! will print out <Buffer of bytes>

{/* <Buffer 7b 0a 20 20 20 20 22 73 75 63 63 65 73 73 22 3a 20 74 72 75 65 2c 0a 20 20 20 20

const objectiveData = JSON.parse(bufferFileData); //! typeof object

const stringData = JSON.stringify(objectiveData, null, 2) //! string


console.log(stringData)
```

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

let cleanedTweets = '',
function cleanTweets(tweetsToClean) {
    //! algo to clean tweets
}
function doOnNewBatch(data) {
    cleanedTweets += cleanTweets(data)
}
const accessTweetsArchive = fs.createReadStream(`./tweetsArchive.json`);
accessTweetsArchive.on('data', doOnNewBatch)
```

We can work on Buffers in Batches, we can set it ourselves But the Default Batch size is 64 Kb

accessTweetArchive = fs.createReadStream('./tweets.json')

{ } - JS ⊥L N - libuv thread

accessTweetsArchive.on('data', doOnNewBatch [f])

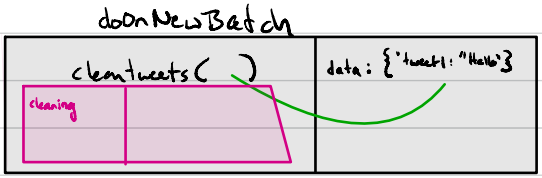
Auto run Func

cleanedTweets : ""

cleanTweets : [f]

doOnNewBatch : [f]

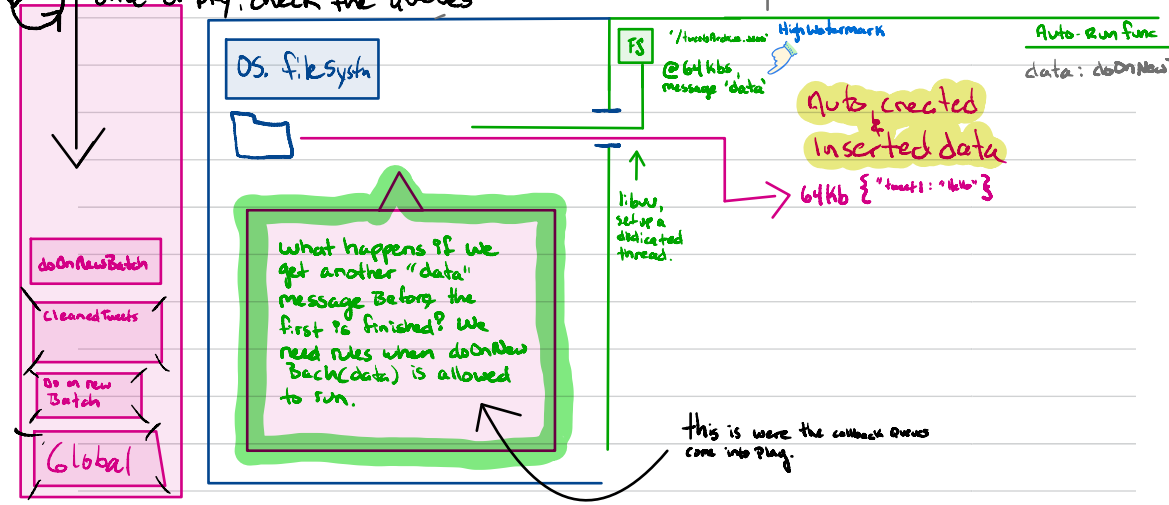
accessTweetArchive : { on: [f] }



Callback Queue

doOnNewBatch([f])

once empty, check the queues



Of Note, we don't start getting data until we set up an .on(data) event

Batches of data =

The High watermark

fs.createReadStream

↓

Sounds intimidating, it's more Batches of data... less a random stream...

Node will broadcast a "message" w/ each batch.

We will have a function attatched to the "message"

The "message" is an event, in the previous code, it's 'data'

.on('data', function)

This is Node's Proudest Feature

We can get super efficient

the fs module will do

1. thing in Node

   Libuv - set up a dedicated thread to get data & start pulling it

1 thing in JS

   return out an object w/ methods that have a connection to the libuv stream

Key Take Aways

1. We can break our data up into chunks
2. Node emits 'events' on incoming I/O that we can run functions on
3. The event loop steps in to handle Async load.

Oh & Libuv is the Bomb.com

Async Node

The CallStack:

JS keeps track of what function is being run & where it is run from. Whenever a function is to be run, it's added to the call stack

The Callback queue

Any function delayed from running, & run Automatically by Node are added to the callback queue when the background Node task has completed.. or there has been some Activity

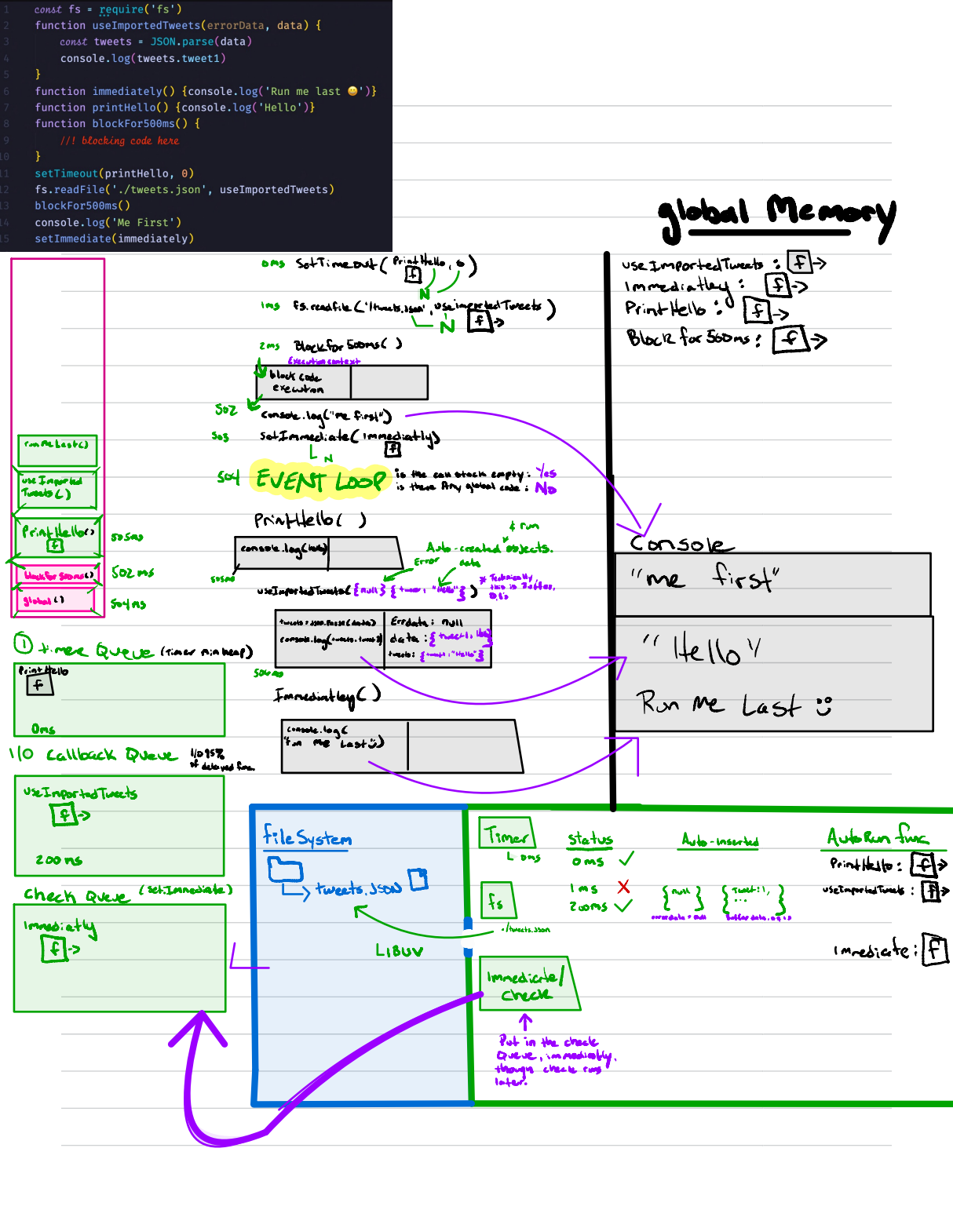
The Event Loop

Determins what code to run next from the Queues

to understand the eventloop, were going to use a contrived example.

```
const fs = require('fs')
function useImportedTweets(errorData, data) {
    const tweets = JSON.parse(data)
    console.log(tweets.tweet1)
}
function immediately() {console.log('Run me last 😀')}
function printHello() {console.log('Hello')}
function blockFor500ms() {
    //! blocking code here
}
setTimeout(printHello, 0)
fs.readFile('./tweets.json', useImportedTweets)
blockFor500ms()
console.log('Me First')
setImmediate(immediately)
```

```
const fs = require('fs')
function useImportedTweets(errorData, data) {
    const tweets = JSON.parse(data)
    console.log(tweets.tweet1)
}
function immediately() {console.log('Run me last 😀')}
function printHello() {console.log('Hello')}
function blockFor500ms() {
    //! blocking code here
}
setTimeout(printHello, 0)
fs.readFile('./tweets.json', useImportedTweets)
blockFor500ms()
console.log('Me First')
setImmediate(immediately)
```



All set timeout does is hold our code w/in node until the timmer is Done, once the timer is done, it gets pushed to the timer queue. It will sit there untill the event loop pushes it to the callstack.

on the fs read file, we get two Auto-created & Auto inserted objects error Data & the fill contents. Error will be null if things went smoothly

Why would we use setImmediate? (using the check queue)

Pass a function to setImmediate, it will absolutely not run Immediately, in fact, it's the opposite.

But why?

if we want to ensure all I/O Has finished (@ that point) - use setImmediate

I/O work is all Auto created & run I/O functions are completed in that "tick"

Not Ever, currently finished callbacks

promises:
rather than using a callback to Auto-run a function, Data will be Auto-updating on JavaScript, that will trigger a function to run. Promises are handled via a "microtask queue, which takes prescidence over the Queues mentioned here"

MICRO TASK
- process.nextTick
- Promise

Timer Q

I/O

check Q

close

The Event Loop doesn't go from timer to I/O to check directly but checks the MICRO-TASK Queue in-between each time

1. Hold each deffered task in one of the task Q's when Node background API completes
2. Add the function to the callstack ONLY when the call stack is totally empty, (Have the event loop check this condition)
3. Prioritize functions in Timer 'Q' over I/O, over setImmediate (check) Q

C++