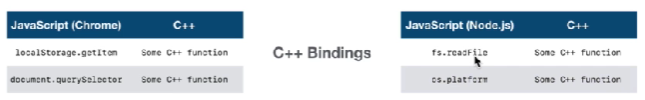
How Node.js Works - A Look Behind the Scenes

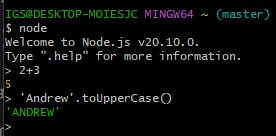
1. V8 engine

Tuesday, February 27, 2024

11:00 AM





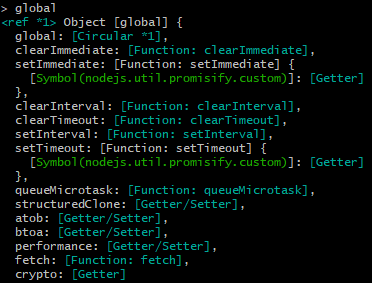


Frontend (browser) has “windows” object

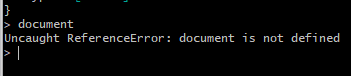
Yet, “windows” does not exist in server-side Node.js env

Only “V8” object exists in server-side Node.js env

V8 global



V8 has NO document



Chrome NO global

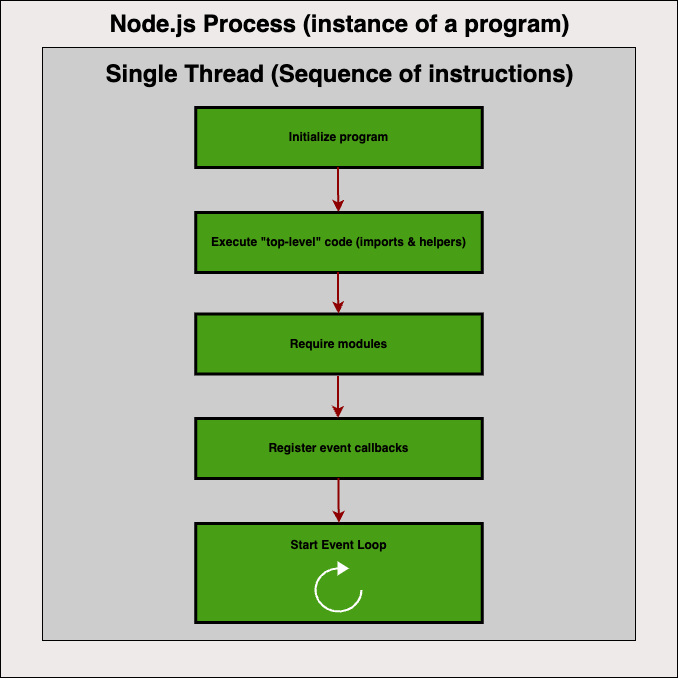
shutdown V8 terminal process.exit()

2. Processes, Threads & Thread Pool

Node.js = a process running a C++ program

Single Thread = A box running in a computer

We must be very careful for not blocking our only Thread



Some tasks registered in the Event Loop may be too heavy or expensive (CPU & RAM) draining to execute, thus blocking the Event Loop

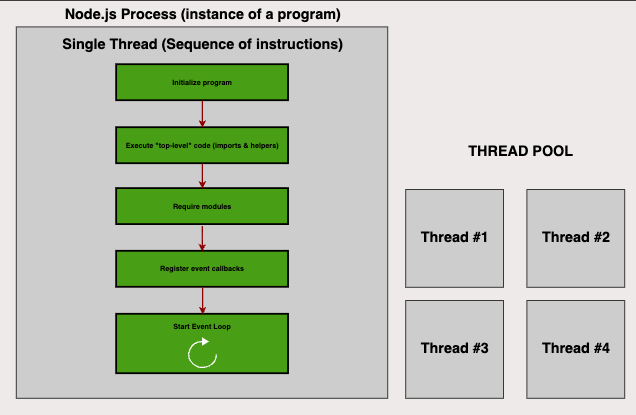
Libuv library provides us a **Thread Pool**

Thread Pool gives 4 additional threads (or more)

which is completely separated from our single thread

Libuv **default** uv\_threadpoolsize = 4

Max. Thread = 128 Threads

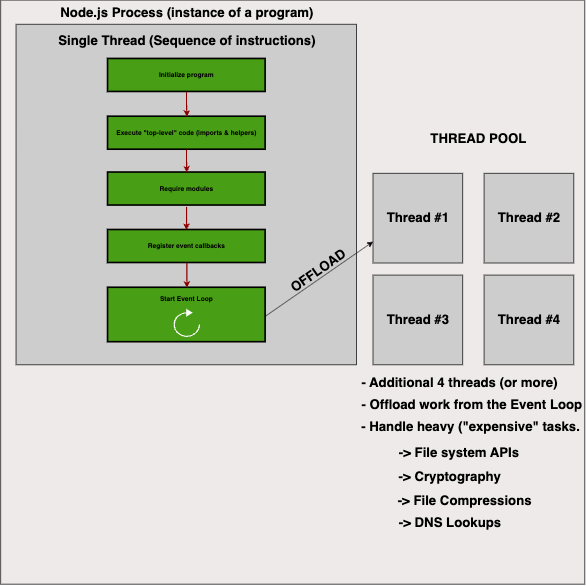


**Event Loop can automatically offload heavy tasks**

**e.g. File System APIs & Cryptography (Password hashing)**

**e.g. File Compressions & DNS Lookups**

**to Thread Pool to avoid Block our Event Loop**



File System APIs e.g. fs.writeFile() to extract users' input into files saved on host machine

Cyptography e.g. Bcrypt to hash passwords

Compression

DNS lookups

3. Node.js Event Loop

Heart of Node.js

**Event Loop:**

1. All Callback functions (non top-level code)

run in Event Loop

2. Node.js is built around Callbacks functions

(some operations that finish & return something in the future)

3. Event-driven architecture:

-Events are emitted

-Event loops pick them up

-Callbacks are called

Event Loop receives events each time something important happens:

-Orchestration:

1. Receives Events

2. Call their callback funcs

3. Off-load resource intensive tasks to Thread Pool

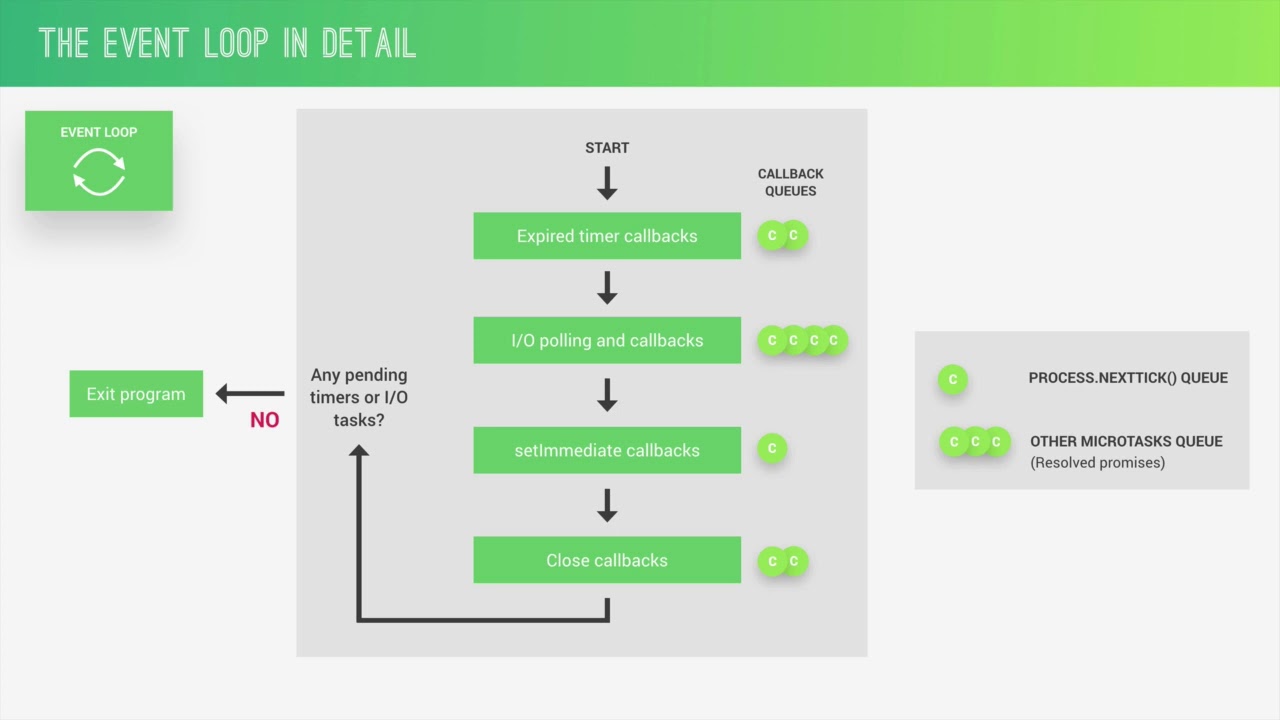
Node runtime starts

Callback Queues:

Callbacks coming from events that Event Loop receives,

sometimes, there's only 1 event queue,

each phase has its own callback queues.



Event Loop does **Orchestration,**

**by receiving events ->**

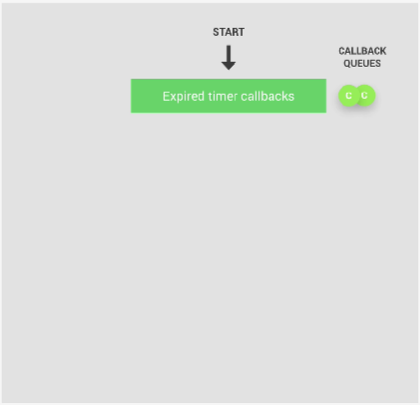
**calling Callback functions ->**

**offloading resource intensive tasks to Thread Pool**

**4 Most important Event Loop phases:**

**0. Start**

**1. Call 'Expired timer callbacks'**





Example:

setTimeout(() => {

console.log('Timer expired!');

}, ms)

If there're Callback funcs from timers that just expired,

these are the 1st ones to be proceeded by the Event Loop.

\*\*If timer expires later, during the time,

when 1 of the other phases are being processed,

then the Callback of that timer will only be called

when Event Loop comes back to this 1st phase.

**\*Callbacks in each queue are processed 1 by 1,**

**until there're no one left in the queue, only then Event Loop will enter next phase \***

**2. I/O polling & callbacks**

**(looking for new I/O events to callback queue)**



Example:

\*\* **I/O** = **Networking & File Access**, 99% of our code

gets executed here

fs.readFile('file.txt', (err, data) => {

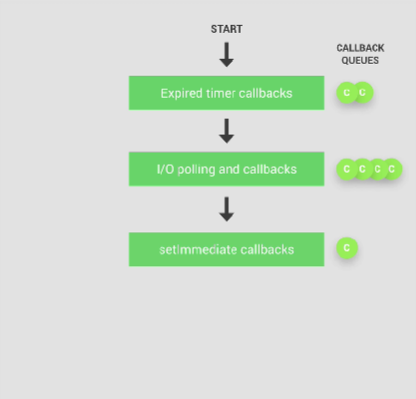
if (err) console.log(err);

console.log('File read!\n', 'data: \n', data);

return data;

});

**3. setImmediate callbacks**



A special kind of timer if we want to process callbacks

immediately after the I/O polling & callbacks phase.

**This is only used for some really advanced use-cases.**

**4. Close callbacks**

**Not that important**



All close events are processed

When a web server / web socket shuts down

**These are all phases in the Event Loop**

1. Expired timer callbacks

2. I/O polling and callbacks

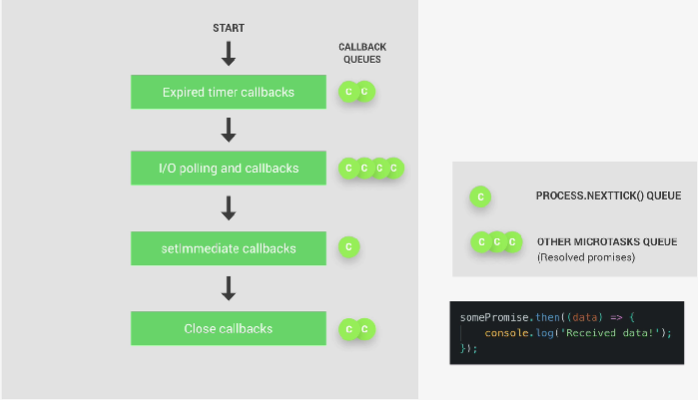
3. setImmediate callbacks

4. Close callbacks

**There're also 2 other Queues:**

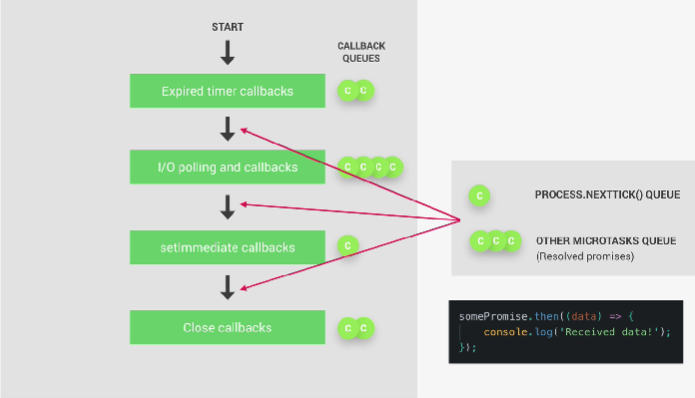
**i. Process.nextTick() queue:**

**ii. Other Microtasks queue: Resolved Promises**





**If there are any Callbacks in 1 of these special queues to be processed, they'll be executed right after the current phase finishes, instead of waiting for the entire Event Loop to finish.**



Execute Callbacks right after current Event Loop phase

Other Microtasks queue

somePromise.then((data) => {

console.log('Received data!');

});

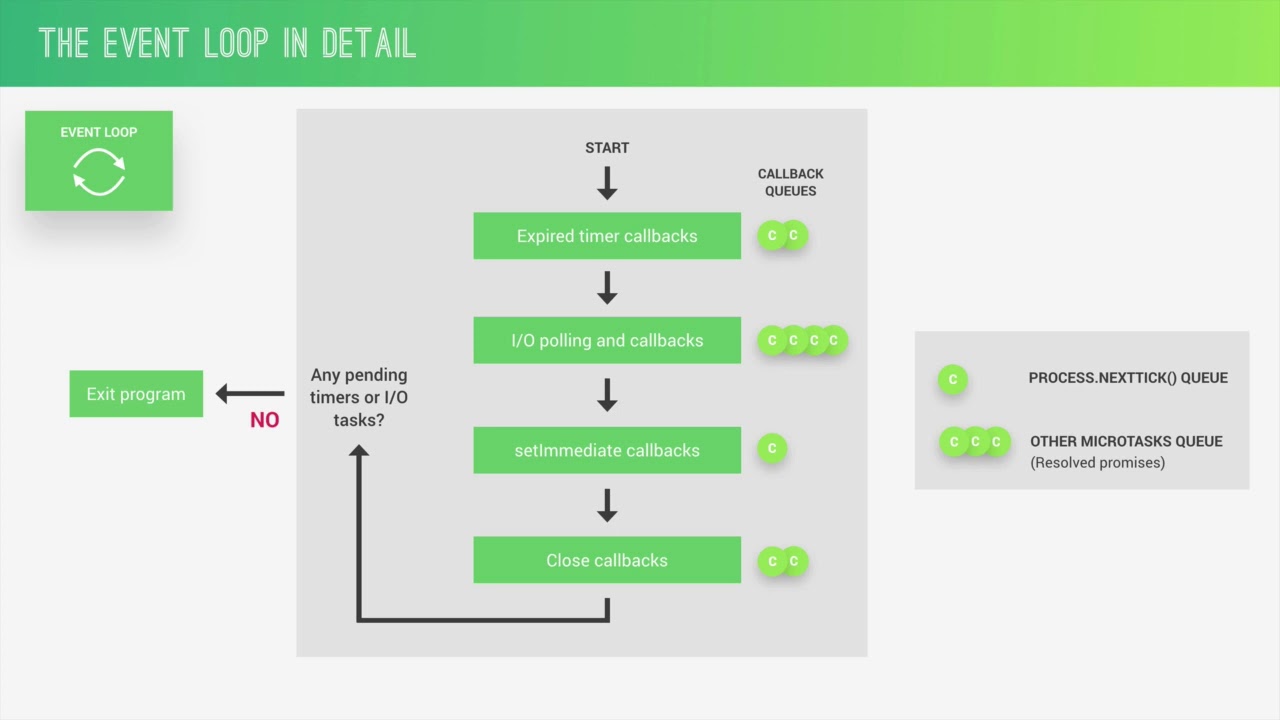
**i. Process.nextTick() queue:**

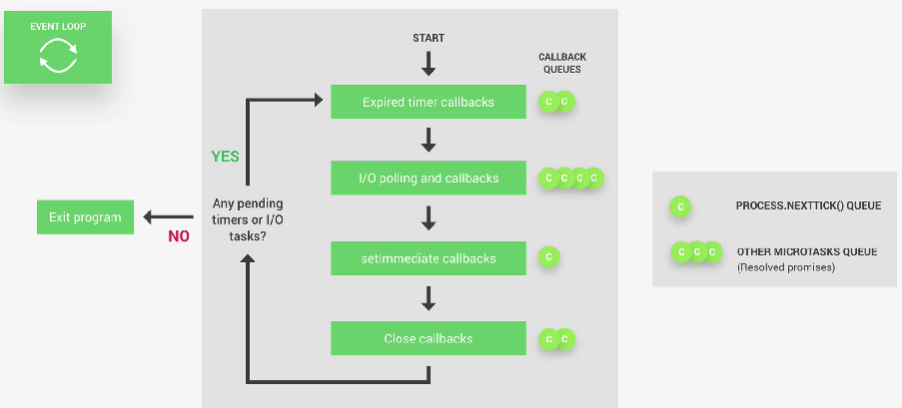
Process.nextTick() queue is only used, when we **really need to execute a certain Callback right after the current Event Loop phase**.

Similar to setImmediate, yet **setImmediate only runs right after I/O callback phase finishes**)

[really advanced use-cases]

1 Tick = 1 Event Loop cycle





Event Loop => Any pending timers or I/O tasks ?

( Proceed NextTick ) : ( Exit Program )

i.e. Node-farm project

-When we're listening for incoming HTTP requests,

we're basically running an I/O tasks,

thus Node.js keeps running & listening to

HTTP requests coming in,

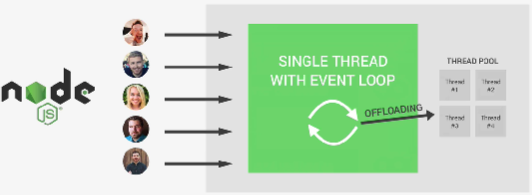
rather than Exiting the app

When fs.readFile or fs.writeFile,

there's also an I/O task, Node won't Exit

It's really important to grasp

the concepts of Node.js Event Loop to debug



It's the **Event Loop makes Asynchronous programming**

**possible in Node.js,**

**making it completely different from other platforms**

Event Loop does the **Orchestration** to

off-load heavy tasks such as File Access,

Networking to the Thread Pool & doing the simpler works itself.

**Why we need the Node.js Event Loop?**

**Cuz, in Node.js, everything works in 1 single thread,**

**thus you can have thousands of**

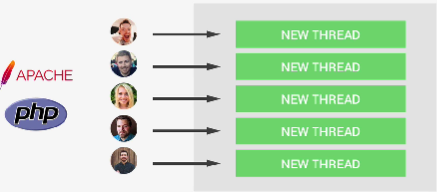
**users using the same thread at the same time**

**However, there's a danger of blocking the single thread,**

**making our entire app slow or even stopping**

**all users from accessing the entire app**

**Apache + PHP:**

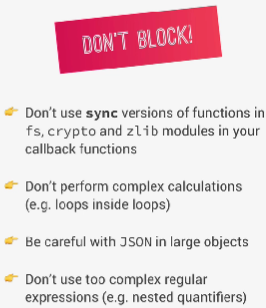


**A new thread is created for each new user.**

**Way more resource-intensive,**

**there's no danger of blocking Event Loop.**

**\*\*How NOT to block Event Loop in Node.js:**



**1. Do NOT use SYNC functions in fs,**

**crypto and zlib modules in Callback functions**

**2. Do NOT perform complex calculations in Event Loop**

**(e.g. Loops inside Loops)**

**3. Be careful with JSON in very large objects**

**It takes long-time to JSON.parse() || JSON.stringify()**

**4. Do NOT use too complex Regular Expressions**

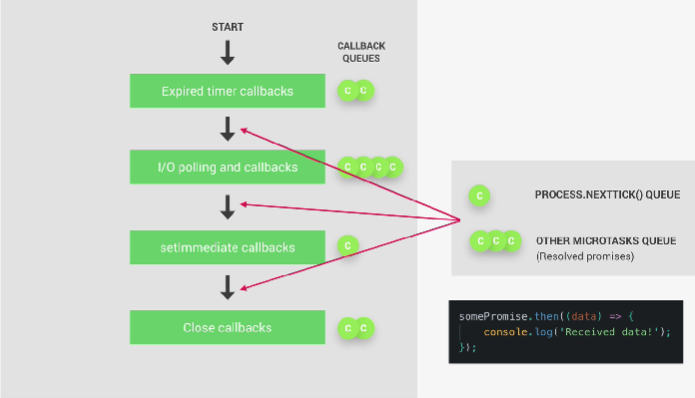
**(e.g. Nested Quantifiers)**

**Potential Solutions:**

**-Manually off-loading Heavy Tasks to Thread Pool**

**-Using Child processors**

4. Revised - The Event Loop in Practice



// import file system module

const fs = require('fs');

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

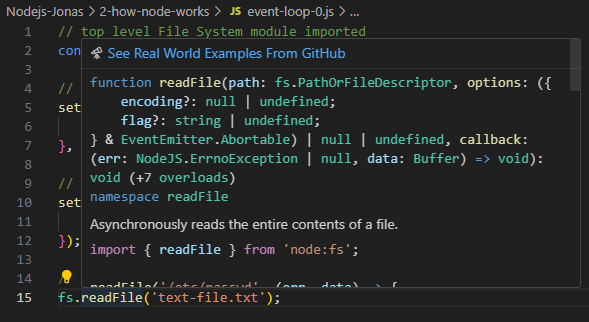
setImmediate(() => console.log('Immediate 1 finished'));

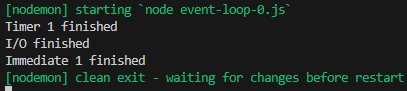
fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

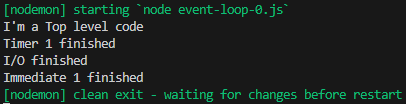
})



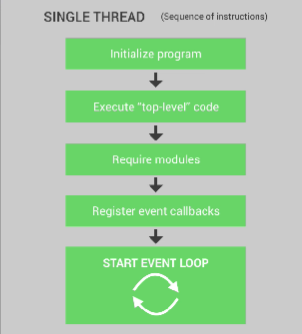


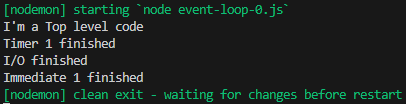


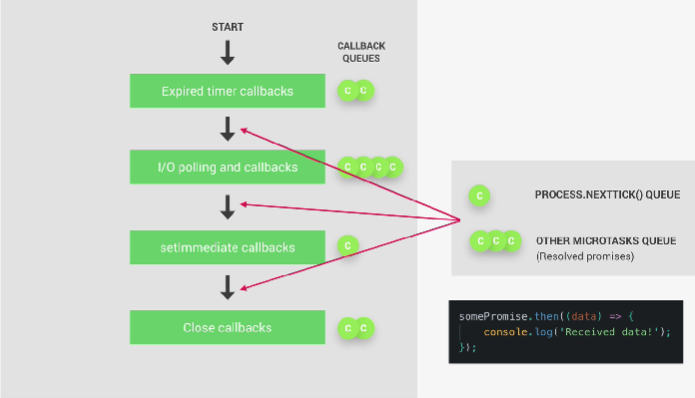




console.log(`I'm a Top level code`); is NOT inside any callbacks







**Time completion depends heavily on**

Resources needed to for a Callback

0. Start Node.js

1. Top-level code => console.log(`I'm Top level code`);

2. Require modules => const fs = require('fs');

3. Register callbacks

i. setTimeout(() => console.log(``));

ii. fs.readFile(filePath, () => {});

iii. setImmediate(() => console.log(``));

;

// ========================

// import file system module

const fs = require('fs');

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

// Event Loop finds Timer 3 as pending after I/O finished

    setTimeout(() => console.log('Timer 3 finished'), 3000);

// When there's no I/O callbacks in the queue

// Event Loop checks if there's any setImmediate()

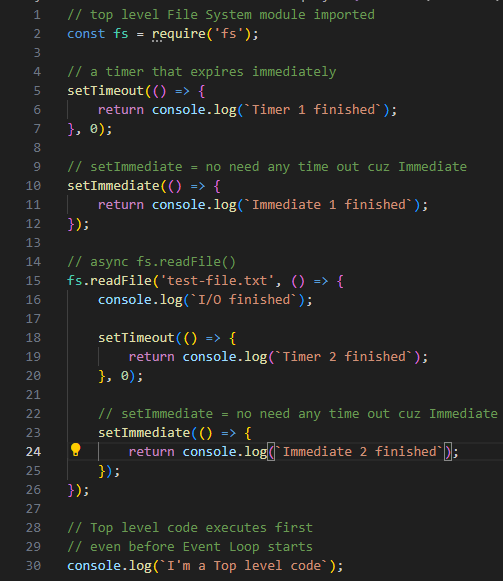
// And execute setImmediate() right away after I/O finished

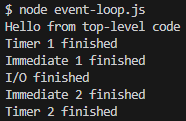
// Even before expired timers

    setImmediate(() => console.log('Immediate 2 finished'));

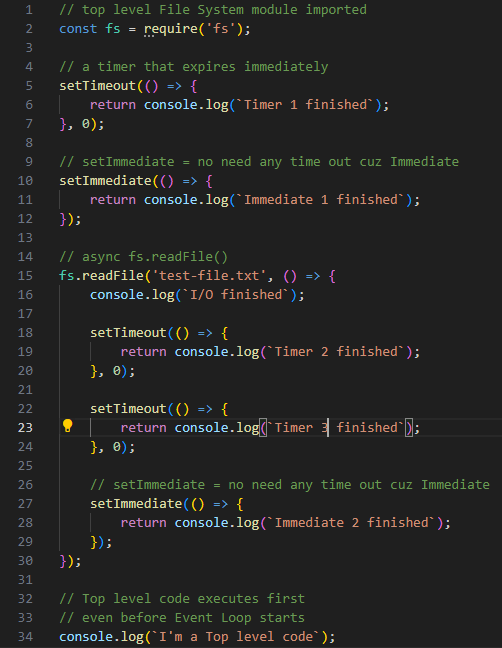
})

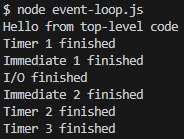
console.log('Hello from top-level code');

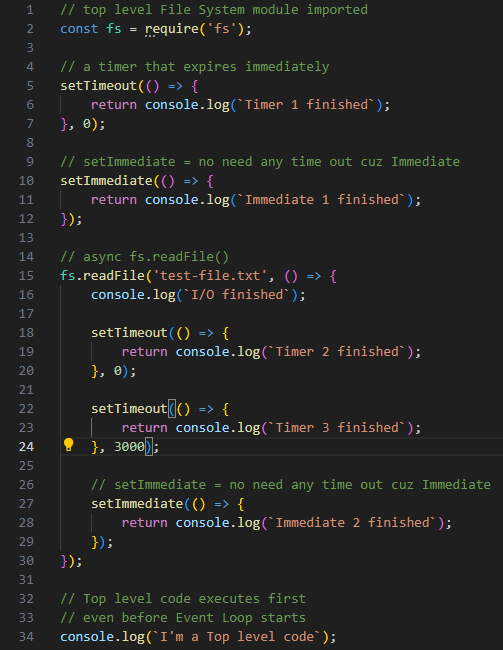


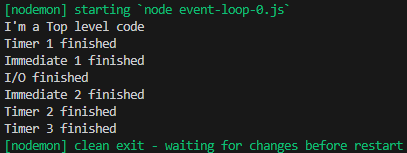


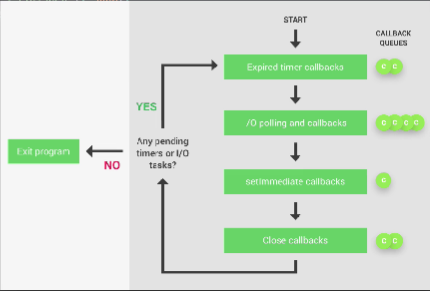
Program kept running until Timer3 finished











**1. After all 1st level Expired timer Callbacks**

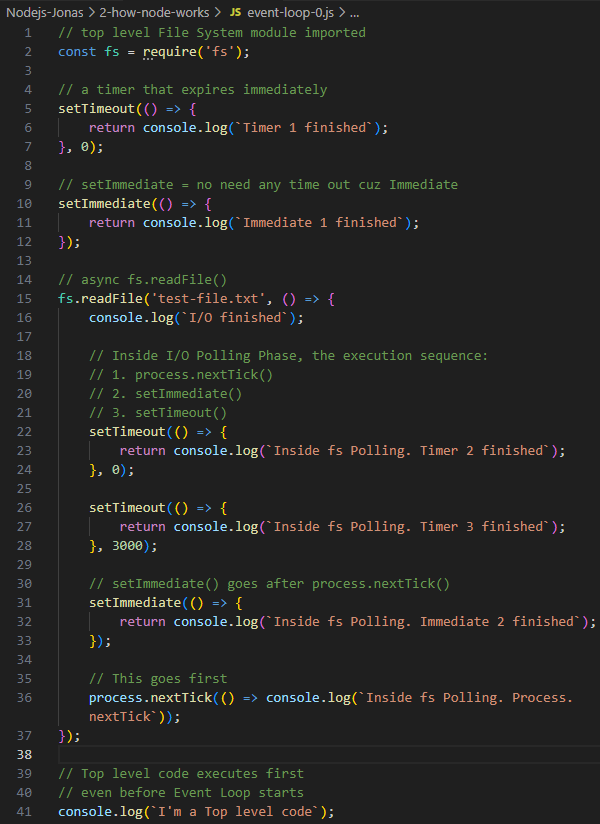
**2. I/O Polling Phase starts Inside the 1st I/O callback,**

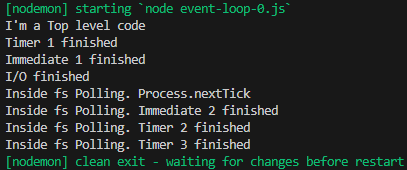
**since Event Loop is inside the Polling Phase,**

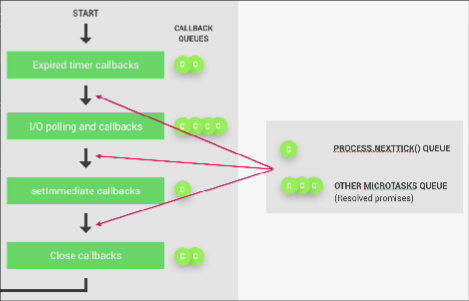
**all setImmedaite callbacks are executed**

**right away after Polling Phase**

**even before Expired Timers setTimeout callbacks**







process.nextTick() is 1 of Micro-task queue

that is executed after each phase of:

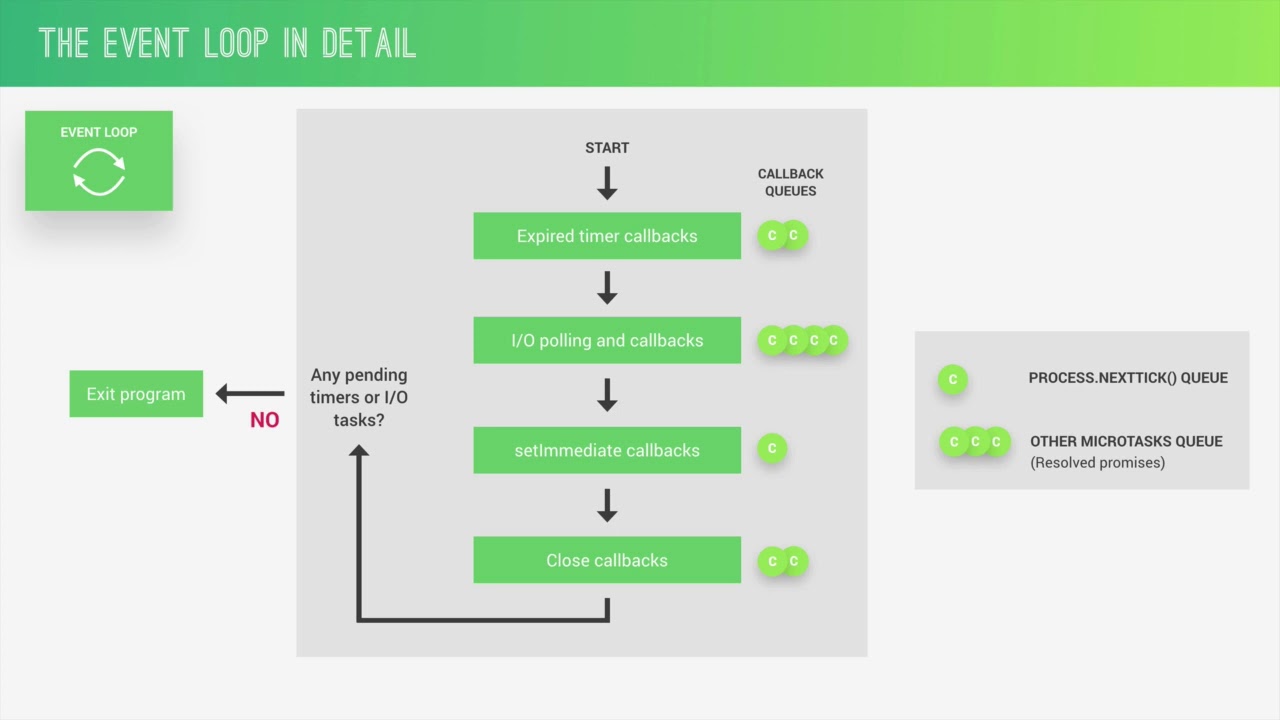
1. Expired timer callbacks

2. I/O Polling and callbacks

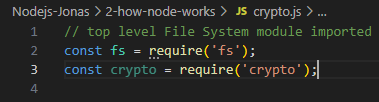
3. setImmediate callbacks

4. Close callbacks

**\*\* process.nextTick() happens before the next Loop phase \*\***



**Using Cryptography to encrypt a password**



crypto.js

// Using Libuv to set Thread Pool Size = 1

// Terminal: export UV\_THREADPOOL\_SIZE=1

// we'll only have 1 Thread in our Thread Pool

require('dotenv').config();

console.log(`process.env.UV\_THREADPOOL\_SIZE:\n${process.env.UV\_THREADPOOL\_SIZE}`);

// top level File System module imported

const fs = require('fs');

const crypto = require('crypto');

const startTime = Date.now();

// a timer that expires immediately

setTimeout(() => {

    return console.log(`Timer 1 finished`);

}, 0);

// setImmediate = no need any time out cuz Immediate

setImmediate(() => {

    return console.log(`Immediate 1 finished`);

});

// async fs.readFile()

fs.readFile('test-file.txt', () => {

    console.log(`I/O finished`);

    // Inside I/O Polling Phase, the execution sequence:

    // 1. process.nextTick()

    // 2. setImmediate()

    // 3. setTimeout()

    setTimeout(() => {

        return console.log(`Inside fs Polling. Timer 2 finished`);

    }, 0);

    setTimeout(() => {

        return console.log(`Inside fs Polling. Timer 3 finished`);

    }, 3000);

    // setImmediate() goes after process.nextTick()

    setImmediate(() => {

        return console.log(`Inside fs Polling. Immediate 2 finished`);

    });

    // This goes first

    process.nextTick(() => console.log(`Inside fs Polling. Process.nextTick`));

    // crypto.pbkdf2('password', 'salt', iterations, keyLength, digest='typeOfAlgorithm', callback)

    // const password = process.env.PASSWORD;

    const password = 'password';

    crypto.pbkdf2(password, 'salt', 100000, 1024, 'sha512', (err, derivedKey) => {

        if (err) console.log(`Error: ${err}`);

        // Convert Buffer to string

        const encryptedPassword = derivedKey.toString('hex');

        console.log(Date.now() - startTime, `ms taken to encrypt this Password`);

        //console.log(`Encrypted Password: ${encryptedPassword}`);

    });

    crypto.pbkdf2(password, 'salt', 100000, 1024, 'sha512', (err, derivedKey) => {

        if (err) console.log(`Error: ${err}`);

        // Convert Buffer to string

        const encryptedPassword = derivedKey.toString('hex');

        console.log(Date.now() - startTime, `ms taken to encrypt this Password`);

        //console.log(`Encrypted Password: ${encryptedPassword}`);

    });

    crypto.pbkdf2(password, 'salt', 100000, 1024, 'sha512', (err, derivedKey) => {

        if (err) console.log(`Error: ${err}`);

        // Convert Buffer to string

        const encryptedPassword = derivedKey.toString('hex');

        console.log(Date.now() - startTime, `ms taken to encrypt this Password`);

        //console.log(`Encrypted Password: ${encryptedPassword}`);

    });

    crypto.pbkdf2(password, 'salt', 100000, 1024, 'sha512', (err, derivedKey) => {

        if (err) console.log(`Error: ${err}`);

        // Convert Buffer to string

        const encryptedPassword = derivedKey.toString('hex');

        console.log(Date.now() - startTime, `ms taken to encrypt this Password`);

        //console.log(`Encrypted Password: ${encryptedPassword}`);

    });

});

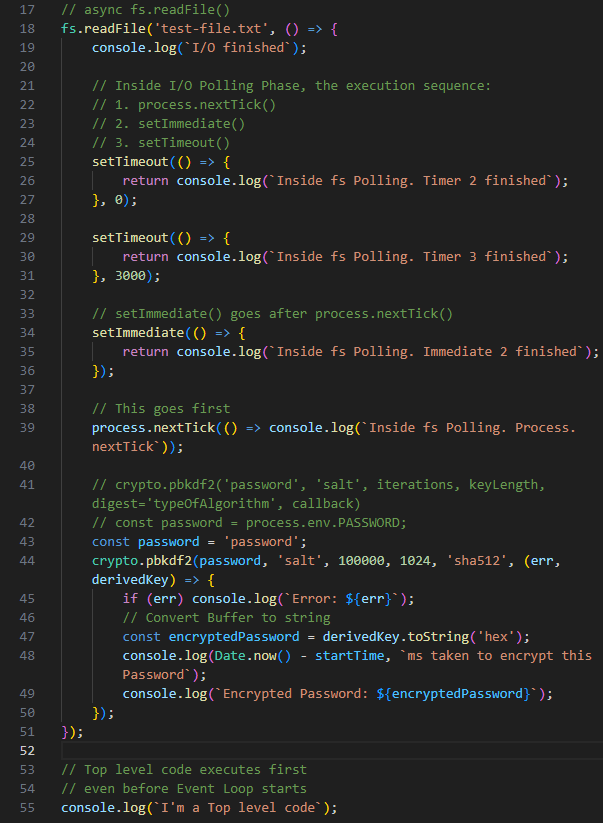
// Top level code executes first

// even before Event Loop starts

console.log(`I'm a Top level code`);

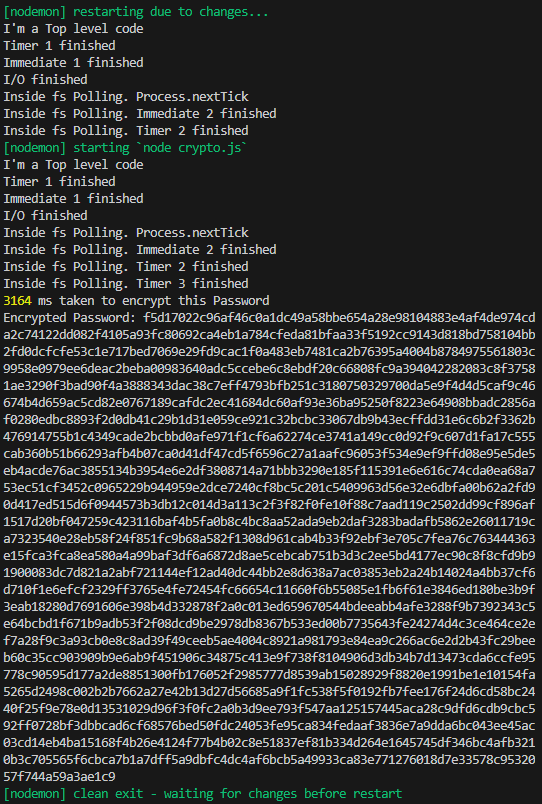
crypto.pbkdf2 method implementation





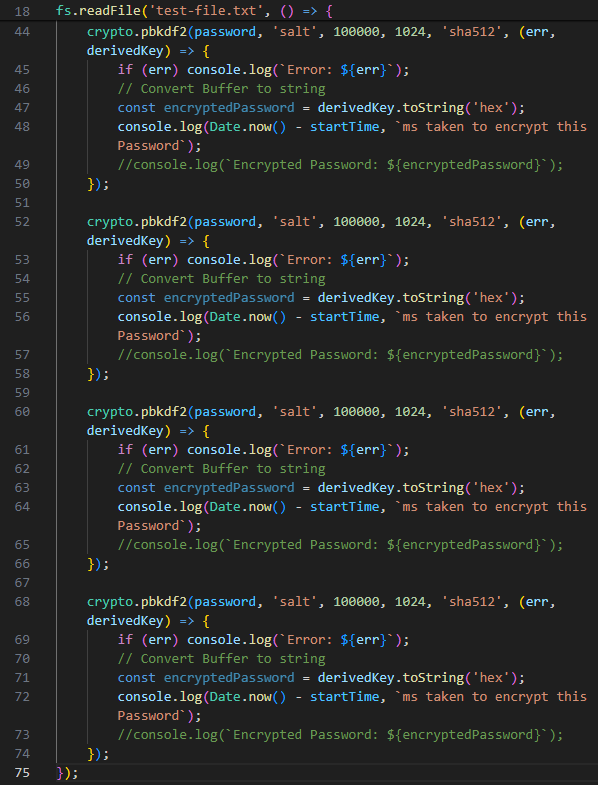
Async nature for crypto.pbkdf2()

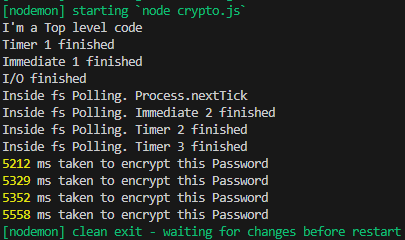
Try encrypting a password with SHA-512 =>



Try running 4 instances of crypto.pbkdf2()

inside fs.readFile('test-file.txt', () => {...});





All 4 password encryption callbacks take

almost the same time to complete

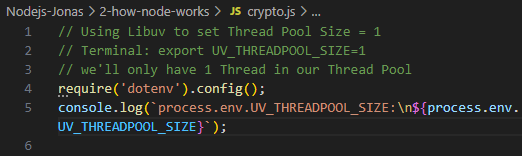
due to async nature & **default UV\_THREADPOOL\_SIZE=4;**

**\*\* Regarding the use of .env file**

**Need to locate projectFolder as rootDir first, otherwise, Node.js runtime may not be able to recognize process.env.PARAMS declared in .env file**

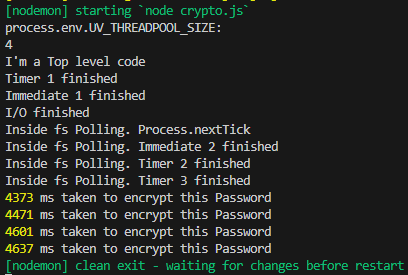
**\*\***

**Assigning Number of Threads in our Thread Pool**

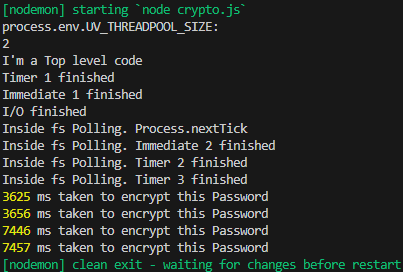


**Setting Number of Node.js ThreadPool using Terminal**

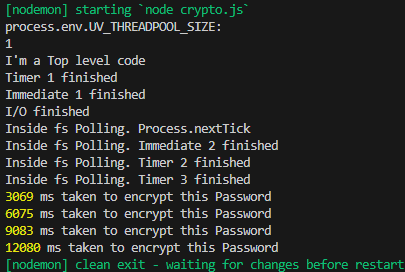
export UV\_THREADPOOL\_SIZE=4 && npm start;



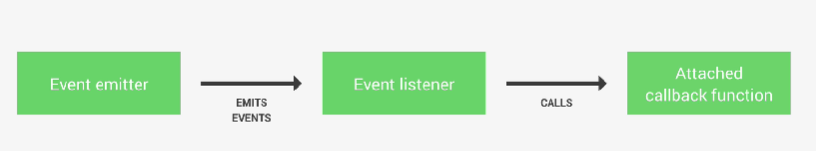
export UV\_THREADPOOL\_SIZE=2 && npm start;

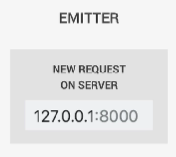


export UV\_THREADPOOL\_SIZE=1 && npm start;

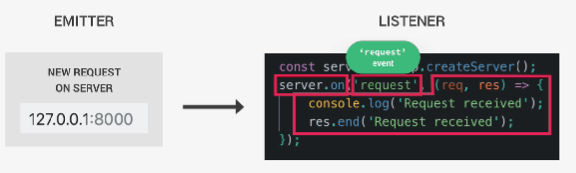


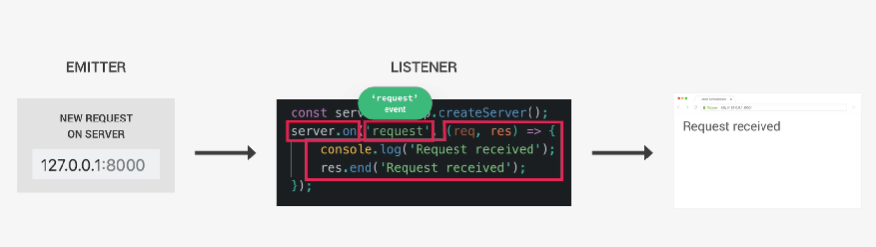
5. Revised - Events & Event-driven Architecture





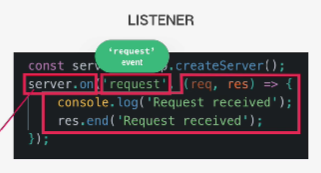






a function calling another function

A method that listens to Events, trigger callbacks (req, res) => {...}



Everything is more de-coupled as callbacks are more

Reactive rather than Called

**Each module can have its own Event Listeners =>**

**more de-coupled**

**Event-driven Architecture makes it way more**

**straightforward to react multiple times to the same Event**

6. Node.js Event Loop & Thread Pool

node.js:

-V8 (JS & C++)

-libuv (C++)

File Access, Network I/O, Event Loop,

Thread Pool (Tread#1, #2, #3, #4)

Access to C++ capabilities through JS

**-http-parser**

**-c-ares (DNS requests)**

**-OpenSSL (crypto)**

**-zlib (compression)**

====**Processes, Threads, and Thread Pool**

Threads & Thread Pool:

Node.js process running in a server

-Single Threaded architecture(Sequence of instructions)

0. Initialize program

1. Execute 'top-level' code

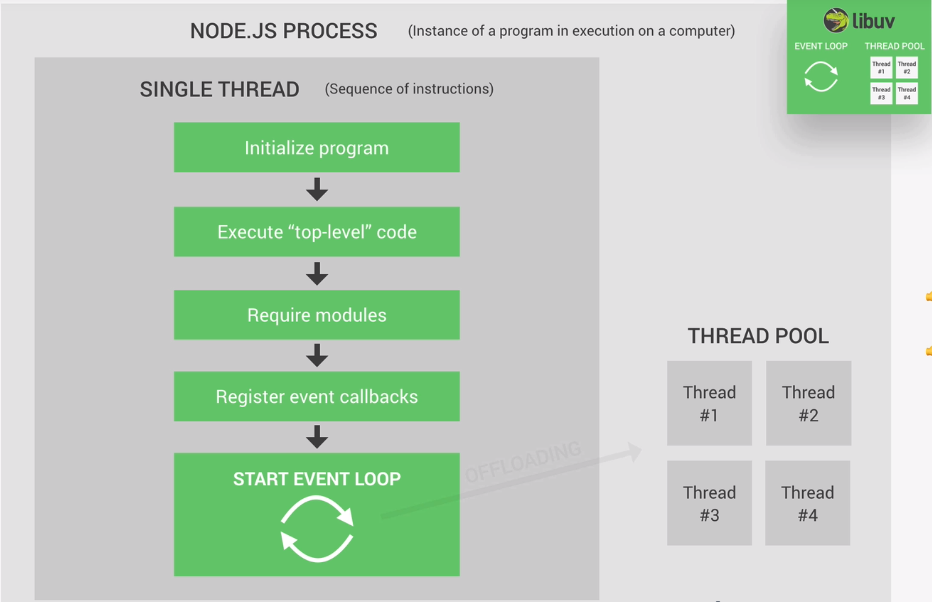
2. Require modules

3. Register Event Callbacks

4. Start Event Loop (most of the work)

Thread Pool (by **libuv**):

-4 additional threads (up to 128 threads ~ 32 CPUs)



Event Loop can auto-distribute tasks to Thread Pool

**Heavy tasks:**

**-File System APIs**

**-Crypto (Password Encryption)**

**-Compression**

**-DNS lookups (map domain name to public IPs)**

====**Node.js Event Loop**

Heart of Node.js

**Event Loop:**

1. All Callback functions (non top-level code) run in Event Loop

2. Node.js is built around Callbacks functions

(some operations that finish & return something in the future)

3. Event-driven architecture:

-Events are emitted

-Event loops pick them up

-Callbacks are called

Event Loop receives events each time

something important happens:

-Orchestration:

1. Receives Events

2. Call their callback funcs

3. Off-load resource intensive tasks to Thread Pool

**=========================================**

**Start - Official Documentation for Offloading**

2 options for a destination Worker Thread Pool to

which offload work.

1. Use the built-in Node.js Worker Pool by

developing a **C++ addon**.

Older versions of Node C++ addon,

built using

<https://github.com/nodejs/nan>

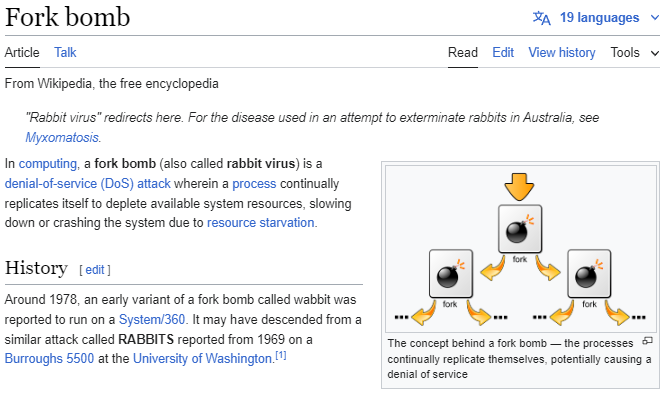
Newer version **N-API**

<https://nodejs.org/api/n-api.html>

Child Process or Cluster

<https://nodejs.org/api/child_process.html>

We should NOT simply create a Child Process for every client, as Node.js can receive client requests more quickly than it can create & manage children



**Some suggestions for Offloading**

We may wish to distinguish between CPU-intensive &

I/O-intensive tasks, as they have markedly different characteristics.

A **CPU-intensive** task **only makes progress when**

its **Worker** is **scheduled**, and the Worker must be

scheduled onto 1 of our machine's logicial cores.

If we have 4 Logical Cores & 5 Workers, 1 of these

Workers cannot make progress. We will be

paying overhead (memory & scheduling costs)

for this Worker & getting no return for it.

**I/O-intensive** tasks involve querying an

external provider such as **fetching API servers**,

**DNS**, **file system**

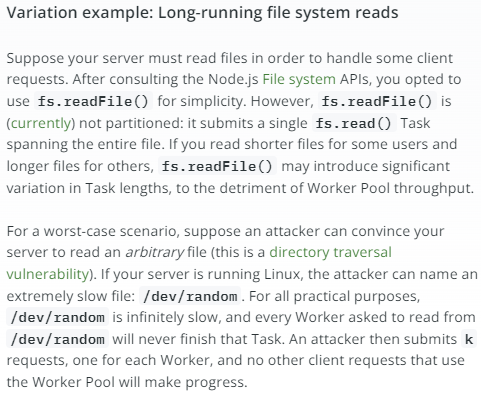
**\*\* Dealing with Variations in task-lengths \*\***

**Uneven resource distributions across**

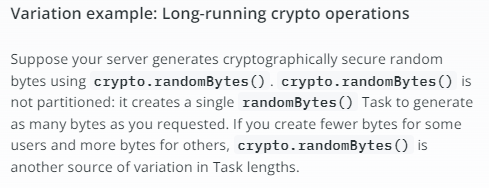
**the same Essence of I/O-driven tasks**

**e.g.**

**File System fs.readFile() fs.writeFile()**



**Cryptography crypto.randomBytes()**



**Task partitioning**

Tasks with variable time costs can harm the

throughput of the Worker Pool.

Minimizing Task times by partitioning each

Task into comparable-cost sub-Tasks.

**Manually partitioning** via **fs.read()**

**Automatically partitioned** via **ReadStream**

For CPU-bound tasks, **asyncAvg** might be

inappropriate for Event Loop, but it is well suited to Worker Pool.

Shorter Tasks expand into a small number of

sub-Tasks, and longer Tasks expand into

a larger number of sub-Tasks.

**Number** of sub-Tasks completed is **NOT**

a userful metric for the throughput of the Worker Pool.

**Concern with the number of Tasks completed**.

**Avoiding Task partitioning**

Purpose of **Task partitioning** is to **minimize**

the **variation** in **Task times**.

e.g.

Summing an array = Shorter Tasks

Sorting an array = Longer Tasks

**2 Ways to reduce variation in Task times**

1. We can create 1 Worker Pool for each class of Task.

2. Routing shorter Tasks & longer Tasks to separate Worker Pool

However, **Partitioning Tasks** incurs **Overhead**

[The costs of creating a Worker Pool Task representation

Manipulating the Worker Pool queue]

**Measuring Overhead in Node.js**

**1. Computational cost**

How much CPU time the API consumes

**2. Memory usage**

How much memory the API operation uses

**3. Response time**

How long does it take to get a response from the API

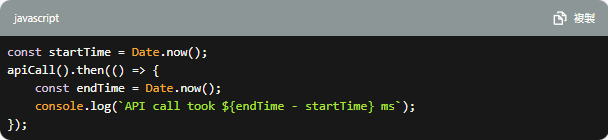
**4. Bandwidth**

How much data is being sent & received

a. Logging Execution Time

You can manually log the start &

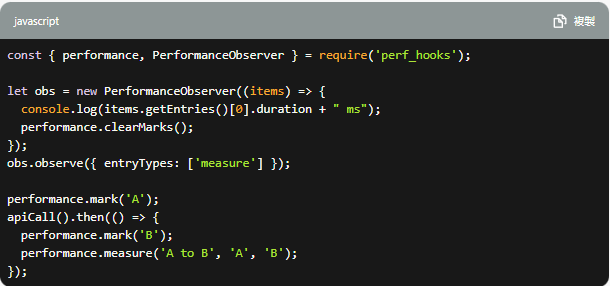
end time of your API calls to measure how long they take



b. Profiling CPU Usage

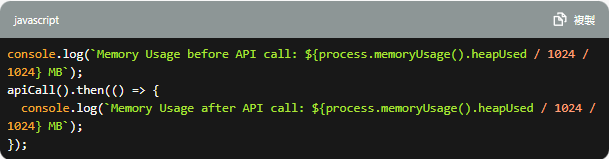
Node.js provides built-in modules like

perf\_hooks that can help in profiling:



c. Monitoring Memory Usage

Monitor memory usage using the process global object:



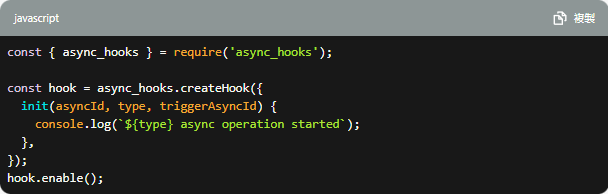
d. Using Profilers & Node.js Tools

clinic.js, 0x, or Node Inspector

**3. Analyze Asynchronous Behavior**

Understanding how asynchronous API calls

impact the Event Loop is crucial in Node.js:



**End of Official Documentations Offloading**

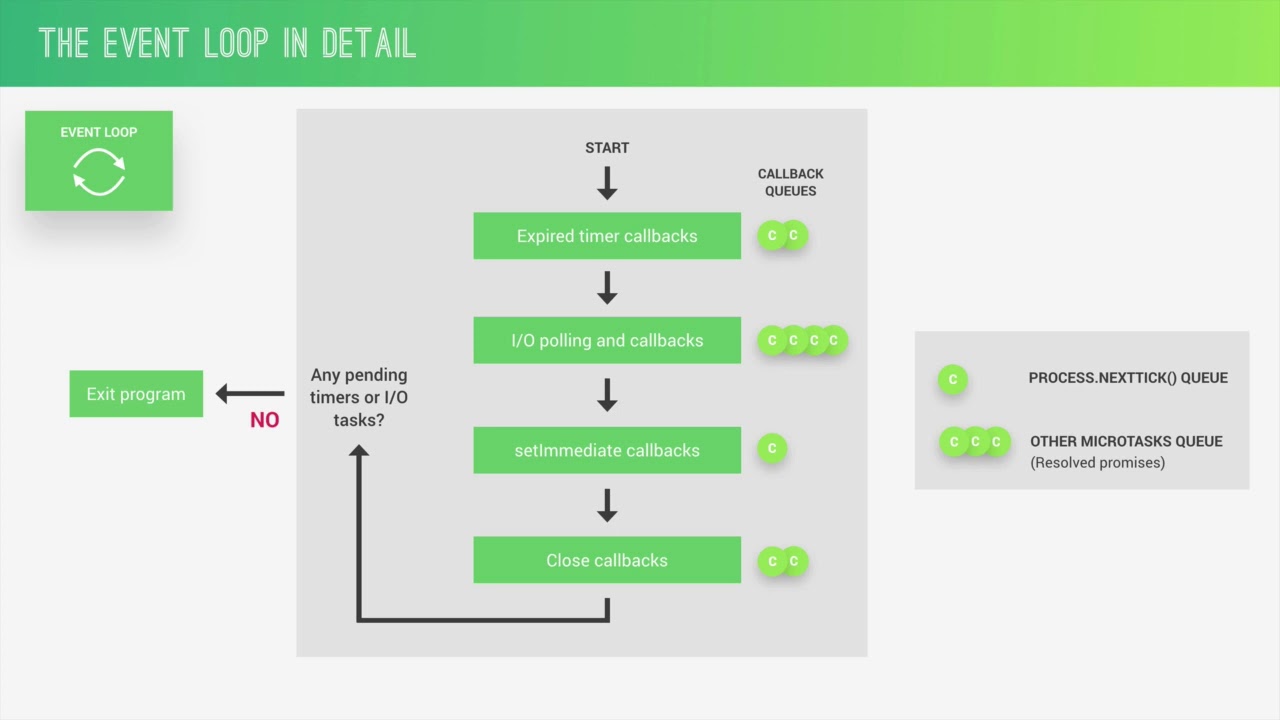
=========================================

Node runtime starts

Callback Queues:

Callbacks coming from events that Event Loop receives,

sometimes, there's only 1 event queue, each phase has its own callback queues.



**4 Most important Event Loop phases:**

**0. Start**

**1. Call 'Expired timer callbacks'**

Example:

setTimeout(() => {

console.log('Timer expired!');

}, ms)

If there're Callback funcs from timers that

just expired, these are the 1st ones to be

proceeded by the Event Loop.

\*\*If timer expires later, during the time,

when 1 of the other phases are being processed,

then the Callback of that timer will only be

called when Event Loop comes back to this 1st phase.

\*Callbacks in each queue are processed 1 by 1,

until there're no one left in the queue,

only then Event Loop will enter next phase.

**2. I/O polling & callbacks (looking for new I/O events to callback queue)**

Example:

\*\*Networking & File Access, 99% of

our code gets executed here

fs.readFile('file.txt', (err, data) => {

if (err) console.log(err);

console.log('File read!\n', 'data: \n', data);

return data;

});

**3. setImmediate callbacks**

A special kind of timer if we want to

process callbacks immediately after the I/O polling & callbacks phase.

**This is only used for some really advanced use-cases.**

**4. Close callbacks**

**Not that important**

All close events are processed

When a web server / web socket shutdown

**There're also 2 other Queues:**

**i. process.nextTick() queue:**

**==========================================**

**Start of Official Documentation - The Node.js Event Loop**

<https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick>

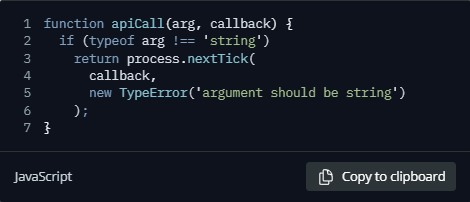
**process.nextTick() will be processed after the current operation is completed**, regardless of the current phase of the Event Loop.

process.nextTick() will be resolved before the event loop continues.

This behaviour can create some bad situations because **it allows us to "starve" our I/O by making recursive process.nextTick() calls**, which prevents the event loop from reaching the **poll** phase.

Q/A Why would that be allowed?

All Node.js APIs are designed to be always asynchronous even it does NOT have to be asynchronous.



This passes an error back to the user but

only after we have allowed the rest of the user's code to execute.

By using **process.nextTick()** ,

we guarantee that **apiCall()** always runs

its callback after the rest of the code &

before the event loop is allowed to proceed.

To achieve this, the JS call stack is allowed

to unwind then immediately execute

the provided callback which allows a person

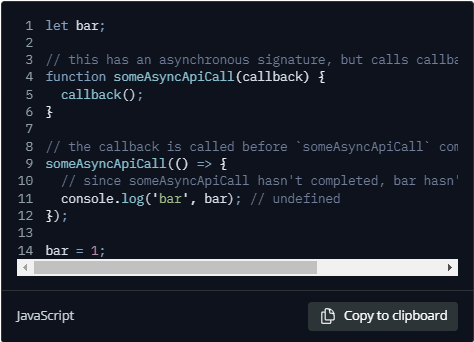
to make recursive calls to **process.nextTick()**

without reaching a

**RangeError: Maximum call stack size exceeded from v8.**

**\*\***

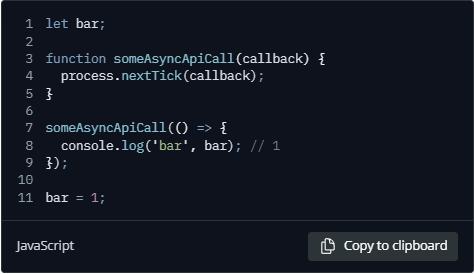
Example of problematic situations



The code above is NOT compiled,

bar is undefined during runtime => Errors

Solution:



Inside function someAsyncApiCall(),

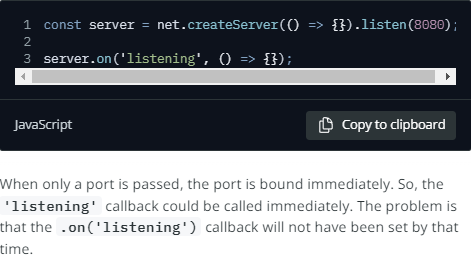
by placing the callback in a **process.nextTick()**,

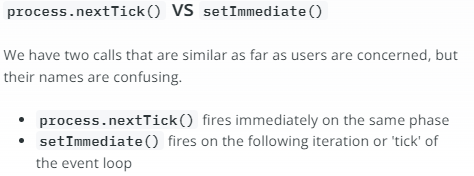
the script is still able to run to completion,

allowing all variables, functions etc. to be

initialized prior to the callback being called.

It also has the advantage of not allowing the event loop to continue.





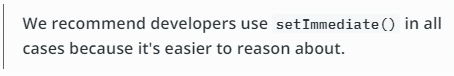
There's a Generic Artifact error in naming process.nextTick() & setImmediate() , however the NPM team cannot

fix this naming issue as there're a large number of

npm packages built & published to npm everyday,

risks of potential breakages are extremely high.

**process.nextTick() fires more immediately than setImmediate()** ,



Two main reasons:

1. Allow users to handle errors,

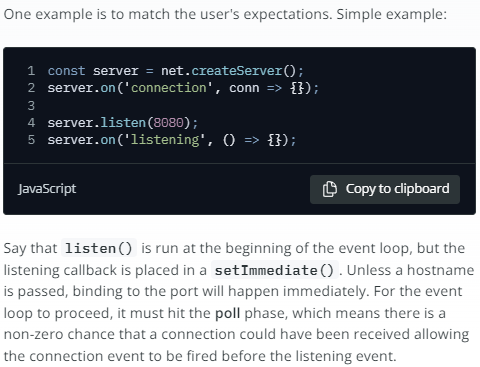
cleanup any then unneeded resources,

or perhaps try the request again before the event loop continues.

2. At times it's necessary to allow

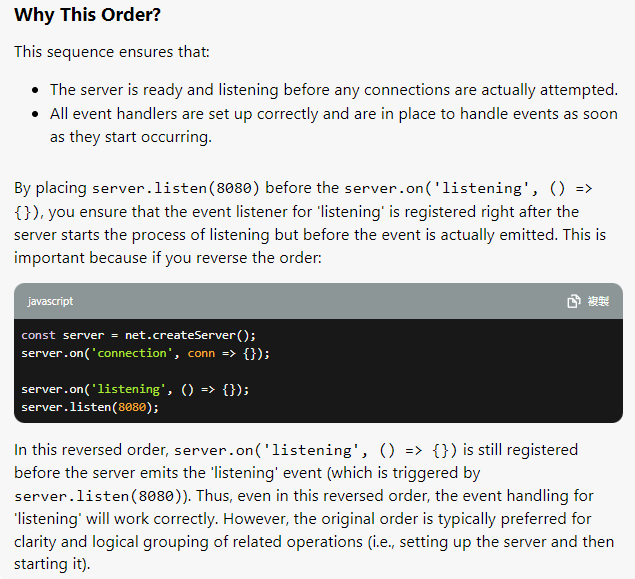
a callback to run after the call stack

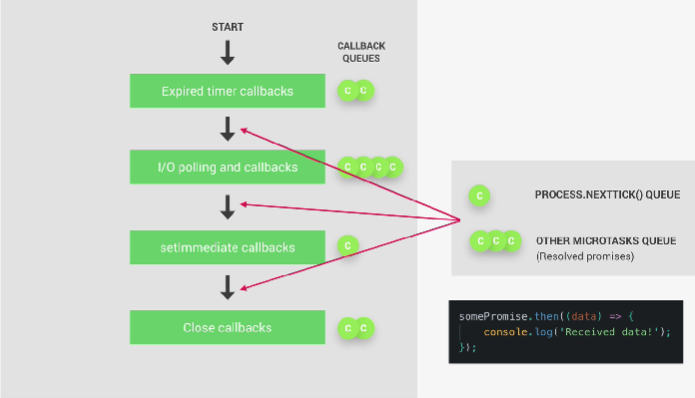
has unwound but before the event loop continues.



server.on() & server.listen() are both top level code

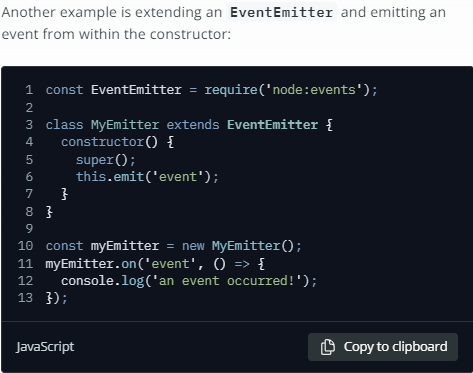
server.listen(8080); should be at the bottom





**\*\* Referring to Event Loop life-cycle \*\***

|  |  |  |
| --- | --- | --- |
| Code | Behaviour in Event Loop | Essence |
| server.listen | runs at the **beginning** | top-level code |
| server.on | runs during **Poll phase** | top-level code |





**\*\* Important concepts \*\***

We cannot emit an event during inheritance

phase from the constructor immediately

because MyEmitter class has NOT been instantiated yet.

Thus, we can use **process.nextTick()** to set

a callback to emit the event after

the constructor has finished, which provides the expected results.

<https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick>

**End of Official Documentation - The Node.js Event Loop**

**========================================**

**ii. Other Microtasks queue: Resolved Promises**

**If there are any Callbacks in 1 of**

**these special queues to be processed,**

**they'll be executed right after the current phase finishes,**

**instead of waiting for the entire Event Loop to finish.**

Execute Callbacks right after current Event Loop phase

Other Microtasks queue

somePromise.then((data) => {

console.log('Received data!');

});

**i. Process.nextTick() queue:**

Process.nextTick() queue is only used,

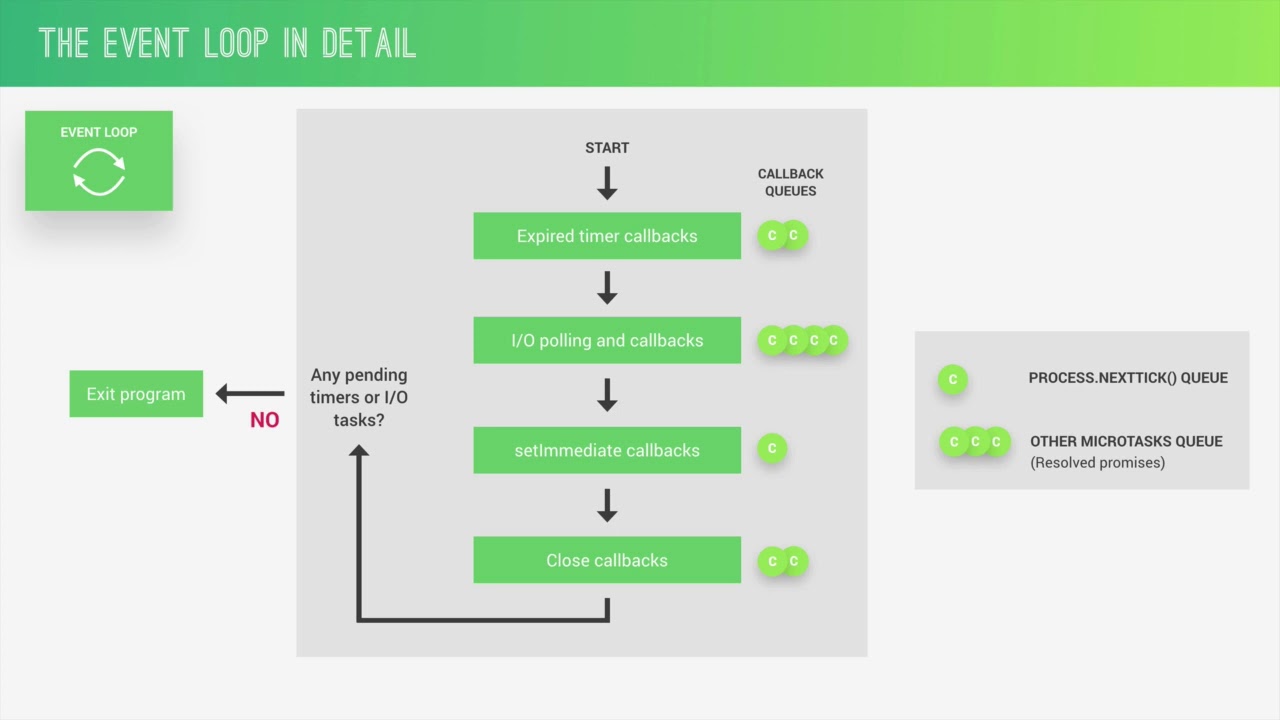
when we **really need to execute a certain**

**Callback right after the current Event Loop phase**.

Similar to setImmediate, yet **setImmediate only runs right after I/O callback phase finishes**)

[really advanced use-case]

1 Tick = 1 Event Loop cycle



Event Loop => Any pending timers or I/O tasks ?

( Proceed NextTick ) : ( Exit Program )

i.e. Node-farm project

-When we're listening for incoming HTTP requests,

we're basically running an I/O tasks, thus node.js

keeps running & listening to HTTP requests coming in,

rather than Exiting the app

-When fs.readFile or fs.writeFile, there's also an I/O task,

node won't Exit

It's really important to grasp the concepts of

Node.js Event Loop to debug.

It's the **Event Loop makes**

**Asynchronous programming possible in Node.js,**

**making it different from other platforms.**

Event Loop does the orchestration to

off-load heavy tasks such as File Access,

Networking to the Thread Pool & doing the simpler works itself.

**Why we need the Node.js Event Loop?**

**Cuz, in Node.js, everything works in 1 single thread,**

**thus you can have thousands of users**

**using the same thread at the same time.**

**However, there's a danger of blocking the single thread,**

**making our entire app slow or**

**even stopping all users from accessing the entire app.**

**Apache + PHP:**

**A new thread is created for each new user.**

**Way more resource-intensive,**

**there's no danger of blocking Event Loop.**

**\*\*How NOT to block Event Loop in Node.js:**

**1. Do NOT use SYNC functions in fs,**

**crypto and zlib modules in Callback functions**

**2. Do NOT perform complex calculations in Event Loop**

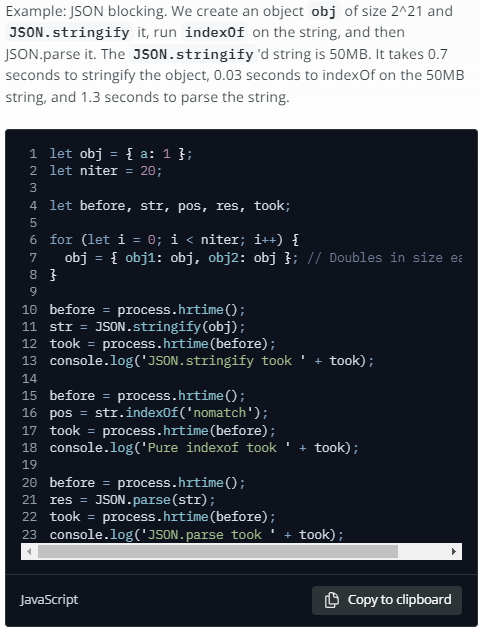
**(e.g. Loops inside Loops)**

**3. Be careful with JSON in very large objects**

**It takes long-time to**

**JSON.parse() || JSON.stringify()**

These are **O(n)** in the length of the input, for large n they can take surprisingly long



npm modules that offer asynchronous JSON APIs:

- JSONStream

<https://www.npmjs.com/package/JSONStream>

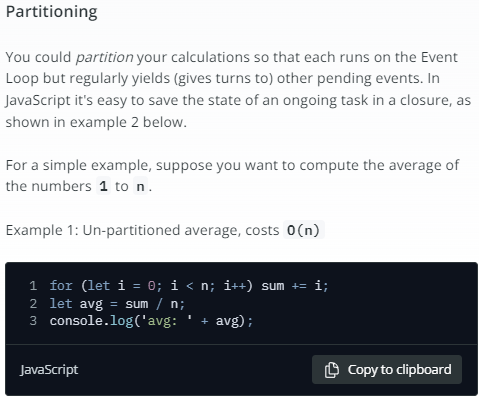
- Big-Friendly JSON

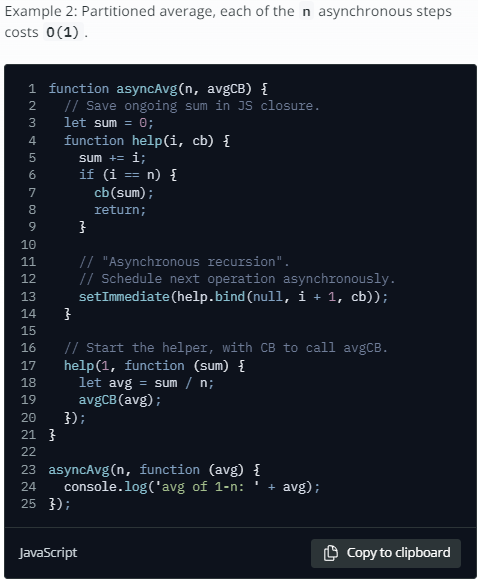
<https://www.npmjs.com/package/bfj>

This uses partitioning-on-the-Event-Loop paradigm outlined below

**Complex calculations without blocking the Event Loop**

**Partitioning**





**Offloading**

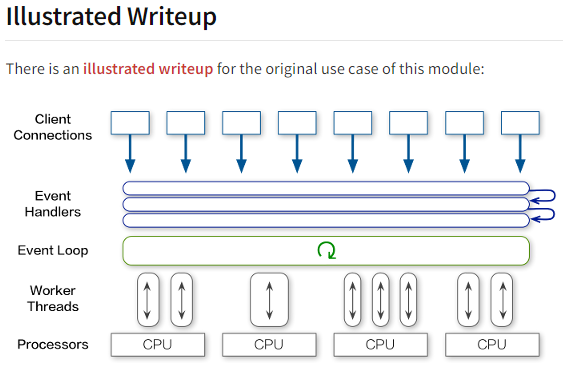
Moving expensive tasks from

Event Loop onto the Worker Thread Pool

<https://www.npmjs.com/package/webworker-threads>

**Event Loop Orchestration:**

**Juggle Events, Listeners and Callbacks quickly & without any hiccups nor interruptions that would ruin its performance**



**4. Do NOT use too complex Regular Expressions**

**e.g. Nested Quantifiers**

**Vulnerable Regex = Taking O(n^2) exponential time**

i. Avoid nested quantifiers like (a+)\*

V8's regexp engine can only handle some of these quickly

ii. Avoid **OR**'s with overlapping clauses, like (a|a)\*

iii. **indexOf** for string match = cheaper resources & never > O(n)

example of vulnerable regexp



This has doubly-nested quantifier:

If a client queries with filePath ///.../\n

[100 /'s followed by a newline character

that the regexp's "." won't match),

then the Event Loop will take effectively forever,

blocking the Event Loop.

This client's REDOS attack causes

all other clients not to get a turn until the regex match finishes.

**Tools to check for REDOS (vulnerable) regex**

<https://github.com/davisjam/safe-regex>

<https://github.com/superhuman/rxxr2>

**Expensive / heavy lifting / resource draining APIs:**

**- Encryption**

**- Compression**

**- File system**

**- Child process**

These APIs are only intended for

scripting convenience, but are not intended for

use in the server context.

If you execute them on the Event Loop,

they will take far longer to complete than

a typical JavaScript instruction, blocking the Event Loop.

In a Node.js server, we should NOT use these Synchronous APIs:

- Encryption:

- **crypto.randomBytes** (synchronous version)

- **crypto.randomFillSync**

- **crypto.pbkdf2Sync**

- Compression:

- **zlib.inflateSync**

- **zlib.deflateSync**

- File system:

- Do not use the synchronous file system APIs e.g. NFS

- **fs.readFileSync**

- **fs.writeFileSync**

- Child process:

- **child\_process.spawnSync**

- **child\_process.execSync**

- **child\_process.execFileSync**

**Potential Solutions:**

**-Manually off-loading Heavy Tasks to Thread Pool**

**-Using Child processors**

====**Node.js Event Loop in Practice**

// import file system module

const fs = require('fs');

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

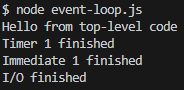
setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

})

console.log('Hello from top-level code');



// ========================

// import file system module

const fs = require('fs');

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

// Event Loop finds Timer 3 as pending after I/O finished

    setTimeout(() => console.log('Timer 3 finished'), 3000);

// When there's no I/O callbacks in the queue

// Event Loop checks if there's any setImmediate()

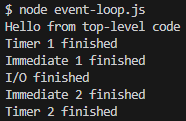
// And execute setImmediate() right away after I/O finished

// Even before expired timers

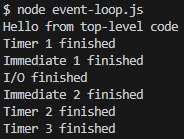
    setImmediate(() => console.log('Immediate 2 finished'));

})

console.log('Hello from top-level code');



Program kept running until Timer3 finished



// import file system module

const fs = require('fs');

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

    setTimeout(() => console.log('Timer 3 finished'), 3000);

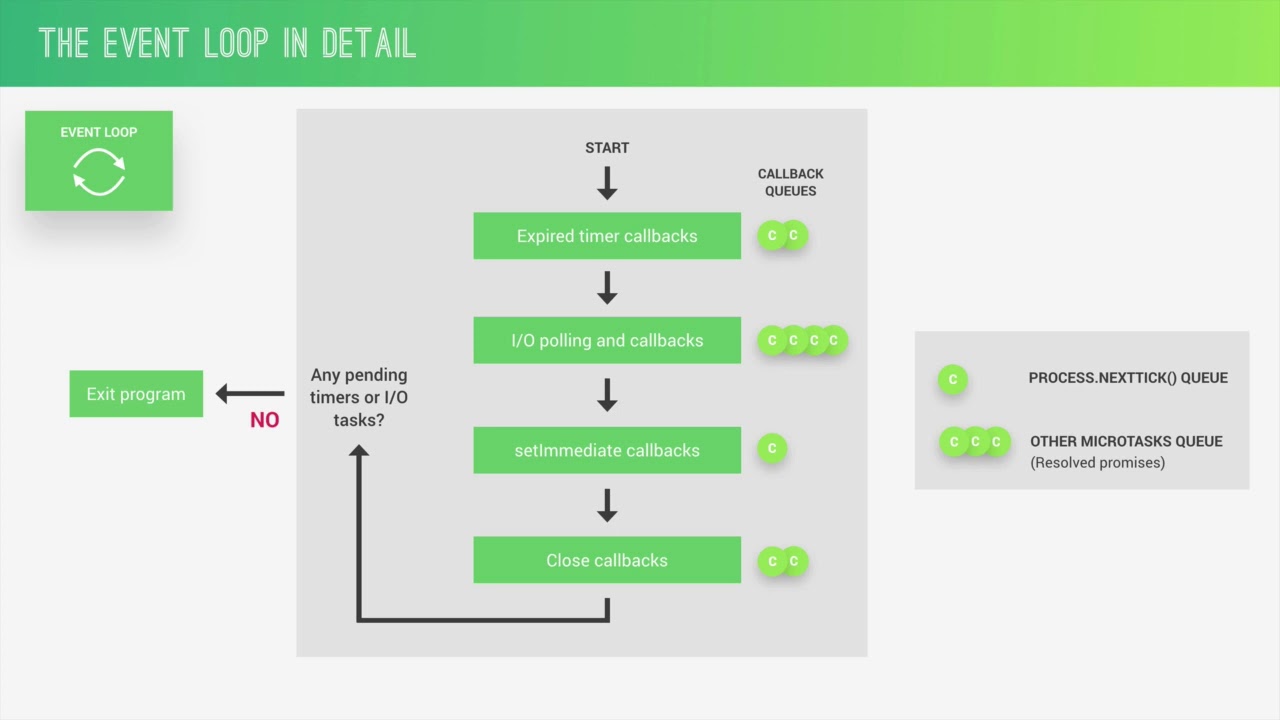
    // When there's no I/O callbacks in the queue

    // Event Loop checks if there's any setImmediate()

    // And execute setImmediate() right away after I/O finished

    // Even before expired timers

    setImmediate(() => console.log('Immediate 2 finished'));



// process.nextTick() can run after each phase

// In this case, process.nextTick() is in I/O polling and callbacks

// Thus, process.nextTick() is run right after I/O finishes &&

// before setImmediate callbacks

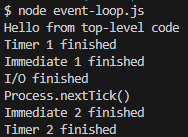
// process.nextTick() is actuall process.nextPhase() that

// happens before the next Event Loop phase, instead of entire Tick

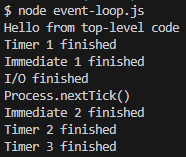
    process.nextTick(() => console.log('Process.nextTick()'));

})

console.log('Hello from top-level code');



Program kept running until Timer3 finished



By the way, these are really advanced use-cases

----Thread Pool => Complex operations to be

off-loaded to Thread Pool =>

Time how long it takes to run &

how to change Thread Pool size =>

Crypto to encrypt a password

// import file system module

const fs = require('fs');

const crypto = require('crypto');

const start = Date.now(); // Current Date in ms

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

    setTimeout(() => console.log('Timer 3 finished'), 3000);

    // When there's no I/O callbacks in the queue

    // Event Loop checks if there's any setImmediate()

    // And execute setImmediate() right away after I/O finished

    // Even before expired timers

    setImmediate(() => console.log('Immediate 2 finished'));

    process.nextTick(() => console.log('Process.nextTick()'));

    // To encrypt a password

    // crypto.pbkdf2('password', 'salt', key length, algorithm, callback)

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

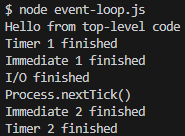
        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

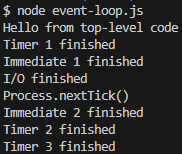
    });

})

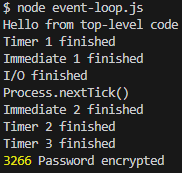
console.log('Hello from top-level code');



// program kept running



// program kept running



// almost 3.2 secs to encrypt this password

// Making 4 instances of Password Encryptions

// import file system module

const fs = require('fs');

const crypto = require('crypto');

const start = Date.now(); // Current Date in ms

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

    setTimeout(() => console.log('Timer 3 finished'), 3000);

    // When there's no I/O callbacks in the queue

    // Event Loop checks if there's any setImmediate()

    // And execute setImmediate() right away after I/O finished

    // Even before expired timers

    setImmediate(() => console.log('Immediate 2 finished'));

    process.nextTick(() => console.log('Process.nextTick()'));

    // Making 4 instances of password encryption

    // To encrypt a password

    // crypto.pbkdf2('password', 'salt', key length, algorithm, callback)

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

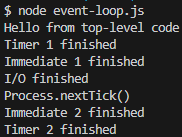
        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

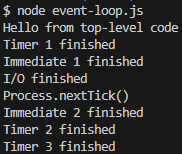
    });

})

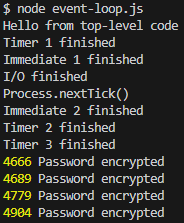
console.log('Hello from top-level code');



// Program kept running



// Program kept running



// All 4 instances of Password Encryptions almost completed at same time

// By default, Thread Pool has 4 threads

// There're 4 full threads working at the same time

// Thus, these 4 instances of Password Encryptions almost completed

// at the same time

// We can change Thread Pool size

// In Windows, we'll have to create a package.json to specify Thread Pool

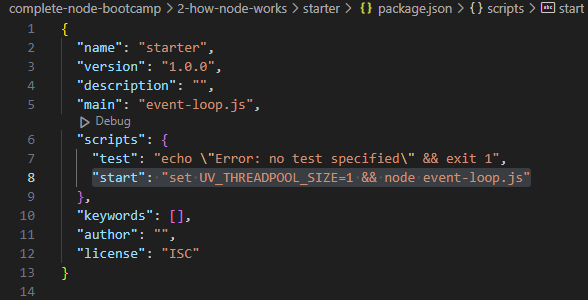
npm init -y

package.json:

"script": {

"start": "set UV\_THREADPOOL\_SIZE=1 && node event-loop.js"

}



event-loop.js:

// import file system module

const fs = require('fs');

const crypto = require('crypto');

const start = Date.now(); // Current Date in ms

// environmental variable

// UV => libuv

// We will only have 1 thread in Thread Pool

process.env.UV\_THREADPOOL\_SIZE = 1;

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

    setTimeout(() => console.log('Timer 3 finished'), 3000);

    // When there's no I/O callbacks in the queue

    // Event Loop checks if there's any setImmediate()

    // And execute setImmediate() right away after I/O finished

    // Even before expired timers

    setImmediate(() => console.log('Immediate 2 finished'));

    process.nextTick(() => console.log('Process.nextTick()'));

    // Making 4 instances of password encryption

    // To encrypt a password

    // crypto.pbkdf2('password', 'salt', key length, algorithm, callback)

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

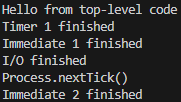
        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

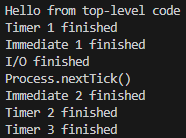
    });

})

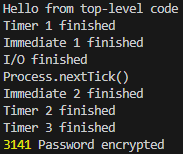
console.log('Hello from top-level code');



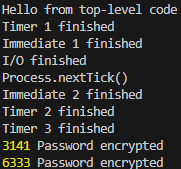
// program kept running



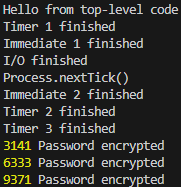
// program kept running



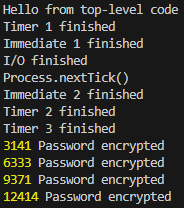
// program kept running



// program kept running



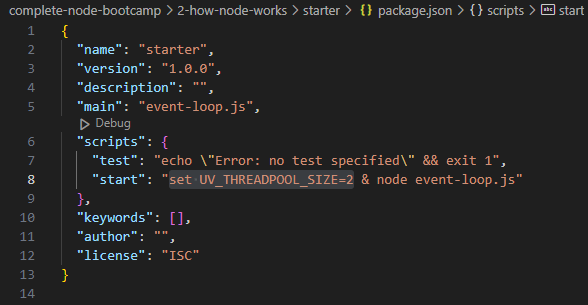
// program kept running

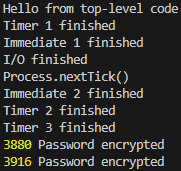


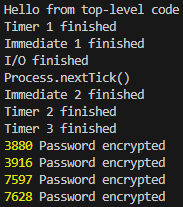
// Each instance of Password Encryption completes 1 after another

----

// If we set UV\_THREADPOOL\_SIZE=2







// Let's test set UV\_THREADPOOL\_SIZE=4

// Execute 4 instances of Synchronous Password

Encryptions in a callback

// import file system module

const fs = require('fs');

const crypto = require('crypto');

const start = Date.now(); // Current Date in ms

// environmental variable

// UV => libuv

// We will only have 1 thread in Thread Pool

process.env.UV\_THREADPOOL\_SIZE = 1;

// 0. Start app

// 1. Execute 'top-level' code

// 2. Require module

// 3. Register Event Callbacks

// 4. Start Event Loop

setTimeout(() => console.log('Timer 1 finished'), 0);

setImmediate(() => console.log('Immediate 1 finished'));

fs.readFile('./test-file.txt', () => {

    console.log('I/O finished');

    setTimeout(() => console.log('Timer 2 finished'), 0);

    setTimeout(() => console.log('Timer 3 finished'), 3000);

    // When there's no I/O callbacks in the queue

    // Event Loop checks if there's any setImmediate()

    // And execute setImmediate() right away after I/O finished

    // Even before expired timers

    setImmediate(() => console.log('Immediate 2 finished'));

    process.nextTick(() => console.log('Process.nextTick()'));

    // Making 4 instances of SYNC password encryption

    // To encrypt a password

    // crypto.pbkdf2('password', 'salt', key length, algorithm, callback)

    crypto.pbkdf2Sync('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2Sync('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2Sync('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

    crypto.pbkdf2Sync('password', 'salt', 100000, 1024, 'sha512', () => {

        // To time how long it takes to encrypt a password

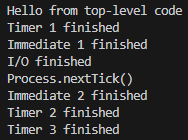
        // Date.now() - start

        console.log(Date.now() - start, 'Password encrypted');

    });

})

console.log('Hello from top-level code');



// Entire Event Loop execution was BLOCKED!

====**Events & Event-driven Architecture**

Event emitter =emits events> Event Listener

=>calls> Attached callback func

i.e.

-requests hitting server

-timer expiring

-file finishing to read

const server = http.createServer();

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

console.log('Request received');

res.end('Request received');

});

// server.on('', () => {} ) is an Event Listener

// Server acts as an emitter =>

// auto-emit a 'request' event each time request hits the server

New Request on Server 127.0.0.1:8000

Behind the scenes,

Server = Instance of EventEmitter class

Observer pattern = Reacting, rather than calling funcs

====**Events in Practice**

We'll need EventEmitter for Events in Node.js

7. C++ addons

**V8**:

C++ library Node.js uses to provide JS implementation.

Providing mechanisms for creating objects,

calling functions etc. V8's API is

documented mostly in the v8.h header file

(deps/v8/include/v8.h in the Node.js source tree).

**libuv**:

The C library that implements the Node.js event loop,

its worker threads & all of the

Asynchronous behaviour of the platform.

Serving as a cross-platform abstraction library,

giving easy, POSIX-like access across all major OS &

many common system tasks

e.g. File System

Web Sockets

Timers

System Events

**Internal Node.js libraries**:

Node.js itself exports C++ APIs that addons can use,

the most important of which is the **node::ObjectWrap** class

Example

**Hello World C++ addon**

**hello.cc**

#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, "world").ToLocalChecked());

}

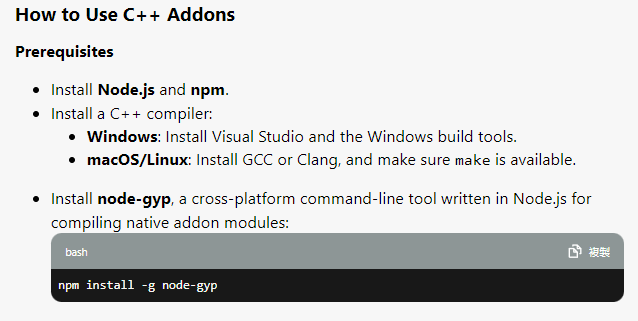
void Initialize(Local<Object> exports) {

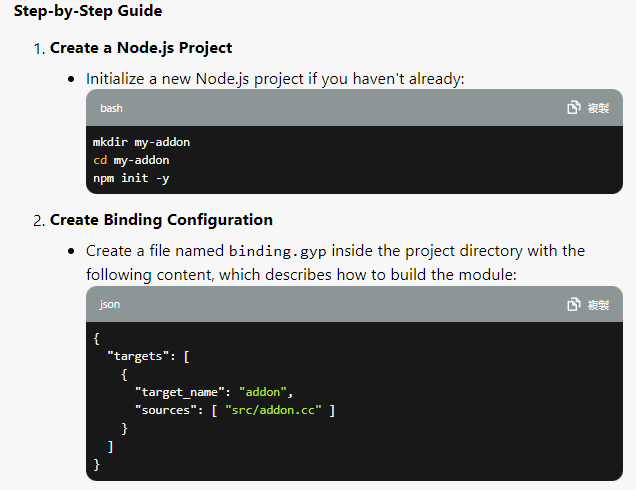
NODE\_SET\_METHOD(exports, "hello", Method);

}

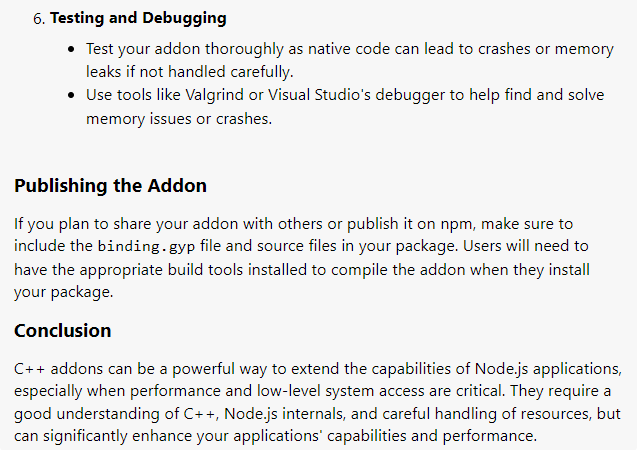
NODE\_MODULE(NODE\_GYP\_MODULE\_NAME, Initialize)

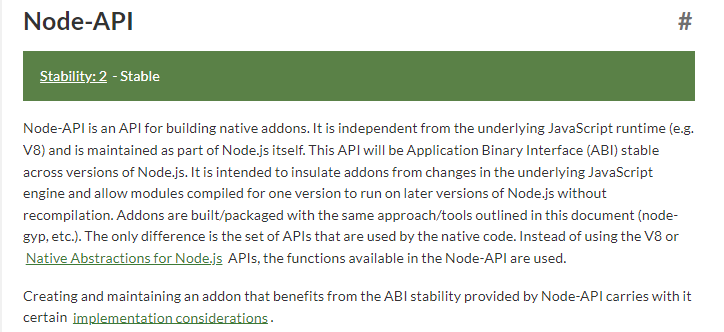
} // namespace demo

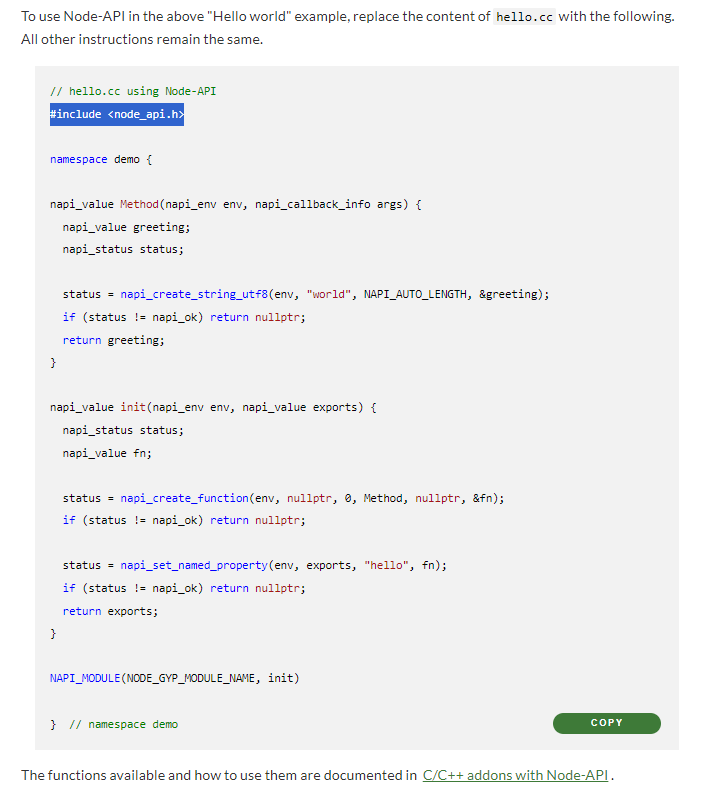












8. Streams

Used to process (read & write) data I/O

piece by piece (chunks),

without completing the whole read or write operation,

and thus without keeping all the data in memory.

When we read a file using streams:

It reads part of the data -> do sth with it->

free memory -> repeat until whole file is processed

Similar to Netflix & Youtube, every clip

is read piece by piece => return to users asap

when it **finishes reading the parts**

|  |  |  |  |
| --- | --- | --- | --- |
| 4 types of Streams: | Description | Example | Important Events & Funcs |
| 1. Readable streams | Streams from which we can consume data    All streams can emit & listen to named events | -http requests    -fs read streams | -data  -end    \*\*pipe()  read() |
| 2. Writable streams | Streams to which we can write data | -http responses    -fs write streams | -drain  -finish    write()  end() |
| 3. Duplex streams | Streams are both readable & writable | net web socket |  |
| 4. Transform streams | Duplex streams that transform data as it is written or read | zlib Gzip creation |  |

**Streams are instances of the EventEmitter class!**

**We'd rather learn 'how to consume Streams',**

**instead of implementing Streams!**

**Streams in Practice**

Reading a large size .txt file => send to clients

streams.js:

const fs = require('fs');

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

// Listen to a request event

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

    // Solution 1

    // Node.js will have to load entire large text file before

    // sending back data to clients

    // App will crash

    // fs.readFile('./test-file.txt', (err, data) => {

    //     if (err) console.log(err);

    //     res.end(data);

    // });

    // Solution 2

    // Create a Stream to consume data piece by piece

    // Return each chunk of data to clients

    const readable = fs.createReadStream('./testtt-file.txt');

    // readable.on('data', chunk => {res.write(chunk)}) -->

    // readable.on('end', ()=>{res.end()})

    readable.on('data', chunk => {

        // Write it to a writable stream

        res.write(chunk);

    })

    // When stream is done reading entire file

    readable.on('end', () => {

        res.end();

    });

    // Error

    readable.on('error', err => {

        console.log(err);

        // if using express.js

        // res.status(500);

        res.writeHead(500, {

            'Content-type': 'text/html',

            'Custom-header': 'ooops, page NOT found :(',

            'Status-code': res.statusCode = 500,

        })

        res.statusCode = 500; // Server error

        res.end('<h1>File NOT found<h1>');

    });

// Solution 3

// To overcome Back Pressure issues

// Use pipe() on all 'Readable Streams'

// to pipe OUTPUT of Readable Streams right into INPUT of Writable Streams

// Auto-handle speed of coming in & speed of going out

const readable = fs.createReadStream('./test-file.txt');

readable.pipe(res);

// readableSource.pipe(writeableDestination);

// readableSource.pipe(duplexStream);

// readableSource.pipe(transformStream);

});

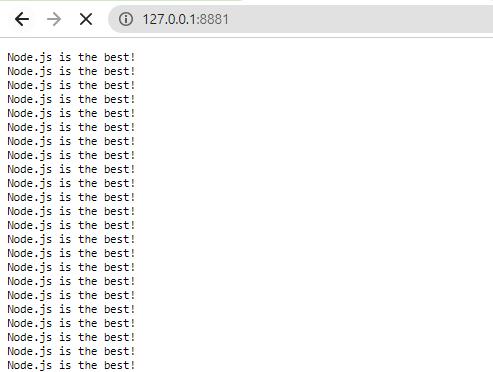
const localhost = '127.0.0.1';

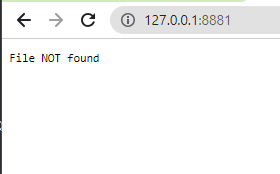
const port = 8881;

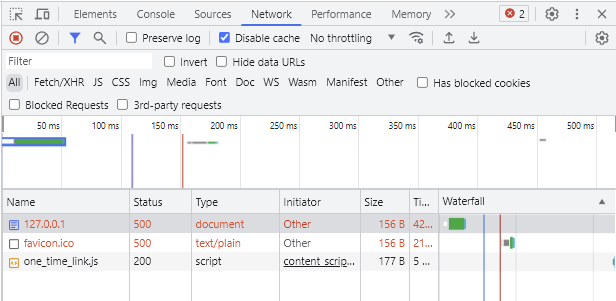
server.listen(port, localhost, () => {

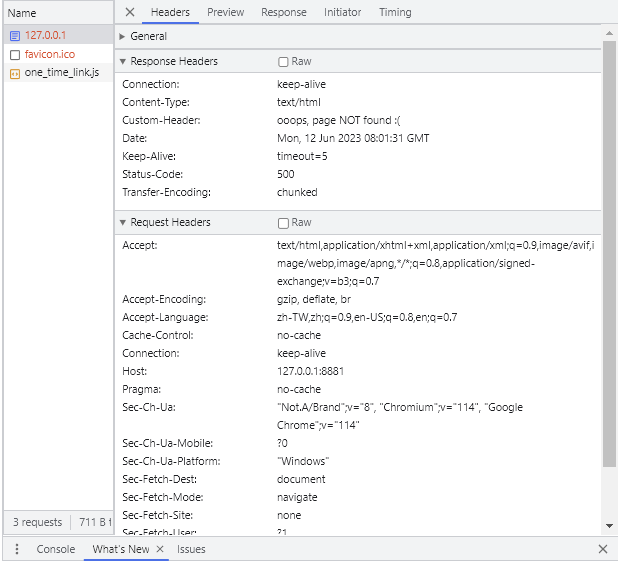
    console.log(`Server has been started on ${localhost}:${port}`);

})









9. Express.js

**Introduction**

Express is a minimal Node.js framework, a higher level of abstraction;

Express.js is written in 100% Node.js;

Express contains a very robust set of features:

-Complex routing

-Easier handling of Requests & Response

-Middleware

-Server-side rendering

Express allows for rapid development of Node.js apps:

we don't have to re-invent the wheel

Express makes it easier to organize our app into MVC architecture

====**Install Postman**

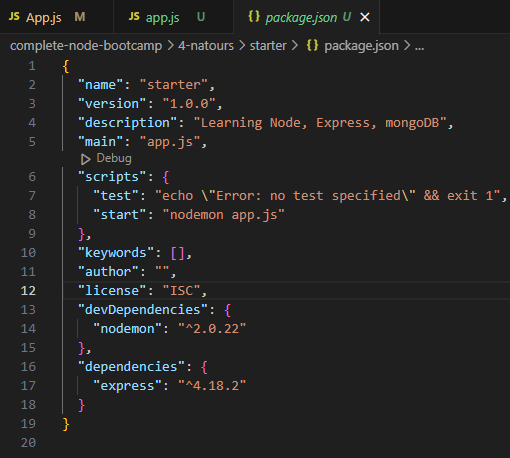
====**Setting up Express & Basic routing**

cd /Desktop/Web/vscode/complete-node-bootcamp/4-natours/starter

npm init -y && npm i nodemon --save-dev && npm i express@4

touch app.js

package.json:



app.js:

const express = require('express');

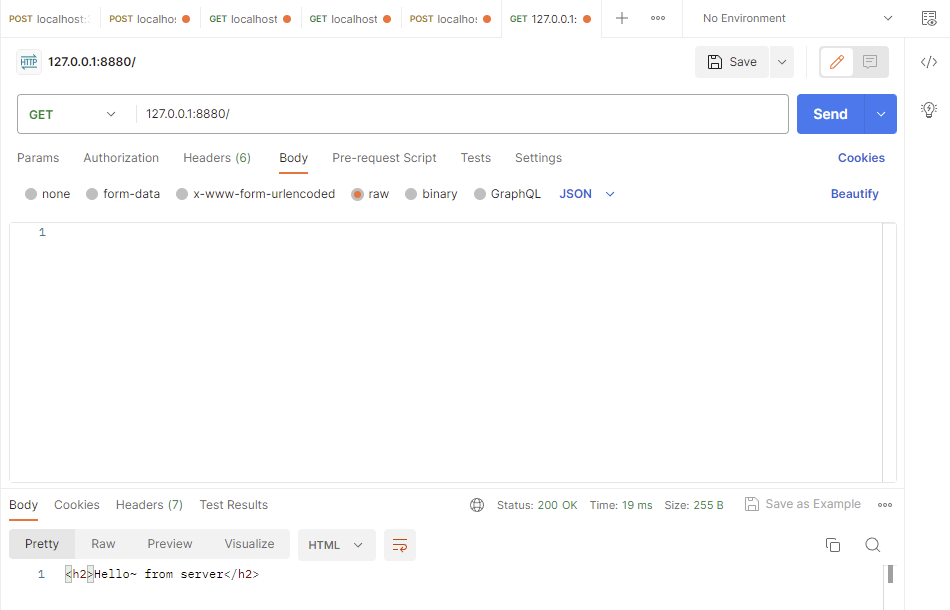
const app = express();

app.get('/', (req, res) => {

    console.log('Request received');

    console.log('req.body: \n', req.body);

    res.status(200).send('<h2>Hello~ from server</h2>');

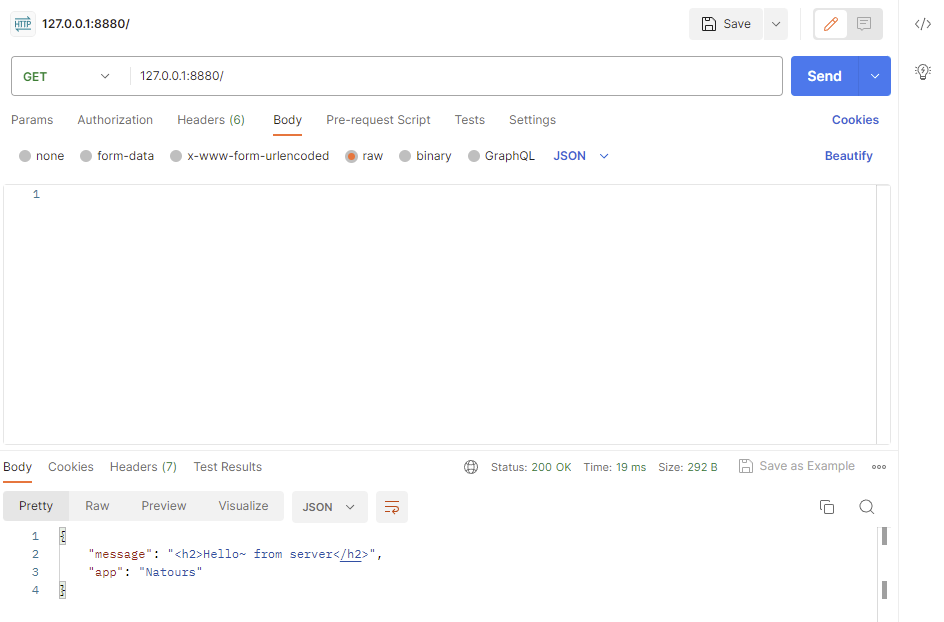


    res.status(200).json({

        message: '<h2>Hello~ from server</h2>',

        app: 'Natours'

    });



})

// To start a web server

const port = 8880;

const localhost = '127.0.0.1';

app.listen(port, localhost, () => {

    console.log(`Server is listening on ${localhost}:${port}`);

})

====**APIs & RESTful API design**

A software can be used by another software,

allowing to talk to each other.

Database => JSON data => API => Browsers, iOS, Android etc...

Node.js' fs or http APIs ("node APIs");

Browser's DOM JavaScript API;

With OOP, when exposing methods to the public,

we're creating an API;

REST architecture:

Represtation State Transfer

1. Separate API into logical **resources**

2. Expose structured, **resource-based URLs**

3. Use **HTTP methods** (get, post, put, delete)

4. Send data as **JSON**

5. **Stateless**

**Resource:**

Object / representation of something,

which has data associated to it.

Any info that can be **named** can be a resource.

tours

users

reviews

[https://www.natours.com/**addNewTour**](https://www.natours.com/addNewTour) **(Endpoint)**

**CRUD**

/addNewTour --> POST /tours (**Create**) --> Data in --> Database

/getTour --> GET /tours/7 (**Read**) --> Data out

/updateTour --> PUT /tours/7 (**Update**) --> Data in --> Database

/deleteTour --> DELETE /tours/7 (**Delete**) --> Data --> Database

login/search are not CRUD

/login

/search?

**originalData**: {

"id": 5,

"tourName": "The Park Camper",

"rating": "4.9",

"guides": [

{

"name": "Steven Miller",

"role": "Lead Guide"

},

{

"name": "Lisa Brown",

"role": "Tour Guide"

}

]

}

Response Formatting:

-JSend

-JSOPN:API

-OData JSON Protocol

**JSend**: {

"status": "success",

"statusCode": "200",

"data": {

"id": 5,

"tourName": "The Park Camper",

"rating": "4.9",

"guides": [

{

"name": "Steven Miller",

"role": "Lead Guide"

},

{

"name": "Lisa Brown",

"role": "Tour Guide"

}

]

}

}

Stateless RESTful API:

All state is handled on **Client**.

Each request must contain **all** info

necessary to process a certain request.

The server should **NOT** have to remember previous requests.

Examples of state:

loggedIn

currentPage

GET /tours/nextPage [Bad]

currentPage=5

GET /tours/nextPage --> Web Server --> State on server:

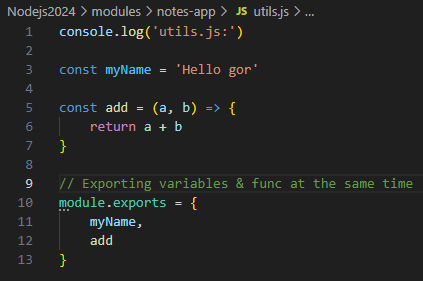
nextPage = currentPage + 1;

send(nextPage); [Bad practice]

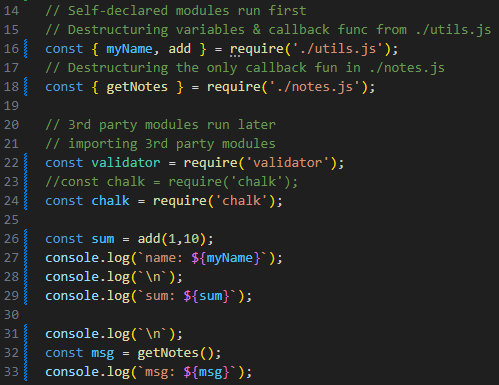
GET /tours/page/6 (State coming from client) --> Web Server --> send(6)

10. moduling

utils.js:



Using destructuring to import variables & callback func to app.js:



11. using modules (no npm)

Creating our own custom modules for other script.js to call:

mkdir ./modules

cd ./modules

touch replaceTemplate.js

\*\* This kind of Node.js templating is outdated in 2024 \*\*

index.js:

Cut the highlighted part => Paste to ./modules/replaceTemplate.js

// Server

// Arrow function to replace {%%} with product props in data.json

const replaceTemplate = (temp, product) => {

    // Avoid direct mutation of original template by storing temp in a new var

    let output = temp.replace(/{%PRODUCTNAME%}/g, product.productName);

    // this function mutates all {%%} in template-xx.html with each obj in json

    // const cardsHtml = dataObj.map(element => replaceTemplate(tempCard, element)).join('');

    // Start mutating now

    // Replacing all {%%} with

    output = output.replace(/{%IMAGE%}/g, product.image);

    output = output.replace(/{%PRICE%}/g, product.price);

    output = output.replace(/{%FROM%}/g, product.from);

    output = output.replace(/{%NUTRIENTS%}/g, product.nutrients);

    output = output.replace(/{%QUANTITY%}/g, product.quantity);

    output = output.replace(/{%DESCRIPTION%}/g, product.description);

    output = output.replace(/{%ID%}/g, product.id);

    if (!product.organic) output = output.replace(/{%NOT\_ORGANIC%}/g, 'not-organic')

    return output; // output final Html after replacing

}

// Top-level code only executes once

// Can only use Sync for top-level code

const tempOverview = fs.readFileSync(`${\_\_dirname}/templates/template-overview.html`, 'utf-8');

const tempCard = fs.readFileSync(`${\_\_dirname}/templates/template-card.html`, 'utf-8');

const tempProduct = fs.readFileSync(`${\_\_dirname}/templates/template-product.html`, 'utf-8');

const data = fs.readFileSync(`${\_\_dirname}/dev-data/data.json`, 'utf-8');

const dataObj = JSON.parse(data);

replaceTemplate.js:

// Export an Arrow function as a module

// Using module.exports = (arugment1, argument2) => {...} to export

// an Arrow Function, without an ES6 package.json --> "type": "module",

module.exports = (temp, product) => {

    // Avoid direct mutation of original template by storing temp in a new var

    let output = temp.replace(/{%PRODUCTNAME%}/g, product.productName);

    // this function mutates all {%%} in template-xx.html with each obj in json

    // const cardsHtml = dataObj.map(element => replaceTemplate(tempCard, element)).join('');

    // Start mutating now

    // Replacing all {%%} with

    output = output.replace(/{%IMAGE%}/g, product.image);

    output = output.replace(/{%PRICE%}/g, product.price);

    output = output.replace(/{%FROM%}/g, product.from);

    output = output.replace(/{%NUTRIENTS%}/g, product.nutrients);

    output = output.replace(/{%QUANTITY%}/g, product.quantity);

    output = output.replace(/{%DESCRIPTION%}/g, product.description);

    output = output.replace(/{%ID%}/g, product.id);

    if (!product.organic) output = output.replace(/{%NOT\_ORGANIC%}/g, 'not-organic')

    return output; // output final Html after replacing

}

index.js:

// Read-Eval-Print-Loop (REPL)

// const repl = require('repl');

const fs = require('fs');

// Networking capabilities => Building a HTTP server

const http = require('http');

// URL module

const url = require('url');

// import our custom module

const replaceTemplate = require('./modules/replaceTemplate');

12. Node

a runtime on machines

// Check version #

node -v



Johnny-Five

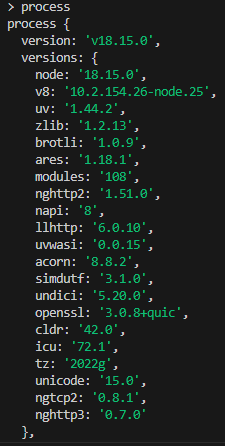
Raspberry Pi

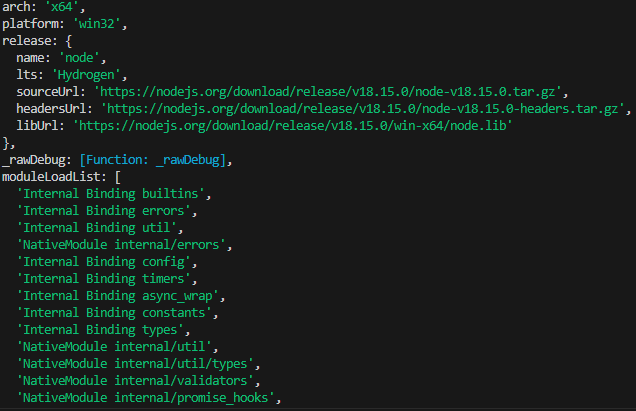
terminal:

node

global

process





// Go Back to terminal:

process.exit();

=================================================

**fetch() in Node:**

<https://blog.logrocket.com/fetch-api-node-js/>

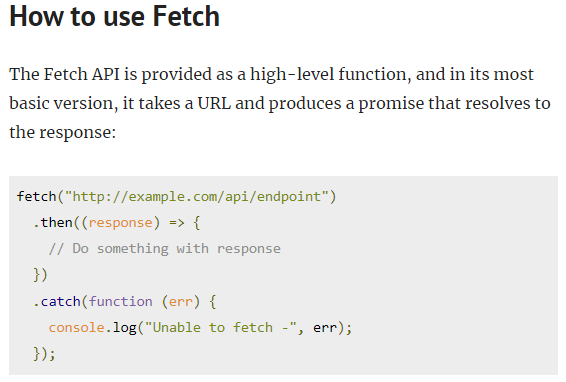
It was difficult to perform async requests across websites.

until 1998, IE5's XMLHttpRequest that was designed to fetch XML data via HTTP.

In 2015, Fetch API was launched as a modern successor to XMLHttpRequest.

Fetch uses promises, allowing simpler &

cleaner API, avoiding callback hell.



\*\* Node.js use node-fetch module for some production platforms… \*\*

13. node commands

terminal:

node -> ctrl + D (exit) || .exit || process.exit()

tab (check global variables i.e. constructors & modules:

Math, global, fs, https)

\_ (previous result)

String. -> press tab to check all props

14. Running script.js in Node

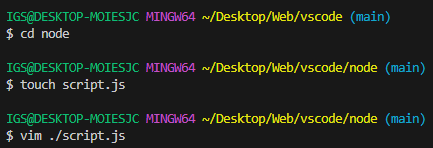
**vscode terminal**

mkdir ./node;

cd ./node;

touch script.js;

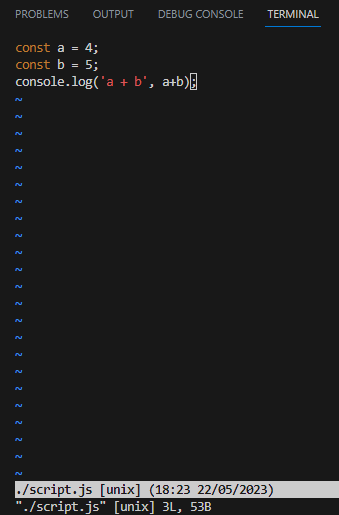
vi ./script.js;



const a = 4;

const b = 5;

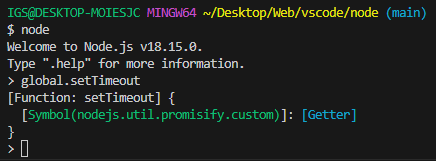
console.log('a+b', a+b);



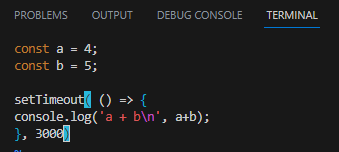
:wq

// Using Node.js runtime to run scripts

node ./script.js;

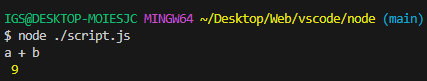


vim ./script.js;



// Node runtime ./script.js will run in 3 seconds

node ./script.js



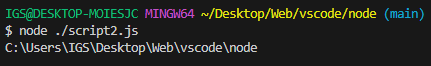
// Create script2.js

const a = 4;

const b = 5;

console.log(\_\_dirname);

node ./script2.js;



15. Promises, Async/Await

Problem with Callbacks = Callback Hell

Dog API to get a random Dog with certain breed:

<https://dog.ceo/dog-api/>

<https://dog.ceo/api/breed/retriever/images/random>

const urls = ['<https://dog.ceo/api/breed/retriever/images/random>'];

Promise.all(urls.map(url => fetch(url).then(res => res.json() ))).then(results => {

if (results) console.log(results);

else throw Error;

})

.catch( err => console.log(err) );

----

// Need package.json & superagent module

npm init -y && npm i superagent

----

dog.txt:

retriever

----

index.js

const fs = require('fs');

const superagent = require('superagent');

fs.readFile(`${\_\_dirname}/dog.txt`, (err, data) => {

if (err) {

return console.error(`Error reading File Content: ${err}`);

}

    console.log(`Breed: ${data}`);

console.log(`\n`);

    superagent

    .get(`https://dog.ceo/api/breed/${data}/images/random`)

    .end((err, res) => {

        if (err) {

            return console.log(err.message);

        };

        console.log('res.body', res.body);

        fs.writeFile('dog-img.txt', res.body.message, err => {

            if (err) {

                return console.log(err.message);

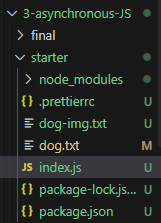
            }

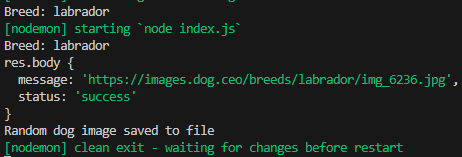
            console.log('Random dog image saved to file')

        });

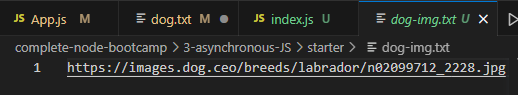
    });

});





dog-img.txt



==== **from Callback Hell => Promises**

index.js

const fs = require('fs');

const superagent = require('superagent');

// Create Arrow func to return new Promise that fs.readFile

const readFilePro = (file) => {

    return new Promise((resolve, reject) => {

// executer func

        fs.readFile(file, (err, data) => {

            // if there's an error --> reject() --> piped err into .catch()

            if (err) reject('Could NOT find file');

            // Promise returns data to us will be piped into .then()

            resolve(data);

        })

    });

}

const writeFilePro = (file, data) => {

    return new Promise((resolve, reject) => {

        fs.writeFile(file, data, (err) => {

            if (err) reject('Could NOT write file');

            // fs.writeFile doesn't need to return data

            resolve('write succeeded');

        })

    })

}

// readFilePro(fileName)

// returns a Promise before calling each of them

readFilePro(`${\_\_dirname}/dog.txt`)

.then(data => {

    console.log(`Breed: ${data}`);

console.log(`\n`);

    // To keep chaining .then(), must return a Promise

    // return a Promise before calling each of them

    return superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

})

.then(res => {

    console.log('res.body.message', res.body.message);

console.log(`\n`);

    // To keep chaining .then(), must return a Promise

    // return a Promise before calling each of them

    return writeFilePro('dog-img.txt', res.body.message);

    // fs.writeFile('dog-img.txt', res.body.message, err => {

    //     if (err) return console.log(err.message);

    //     console.log('Random dog image saved to file')

    // });

})

.then(() => {

    console.log('Random dog image saved to file!');

})

.catch(err => {

    console.log('err', err);

});

// Callback Hell

// Callbacks inside of Callbacks inside of Callbacks

// fs.readFile(`${\_\_dirname}/dog.txt`, (err, data) => {

//     if (err) {

//         return console.log(err.message);

//     }

//     console.log(`Breed: ${data}`);

//     superagent

//     .get(`https://dog.ceo/api/breed/${data}/images/random`)

//     .then(res => {

//         console.log('res.body', res.body);

//         fs.writeFile('dog-img.txt', res.body.message, err => {

//             if (err) {

//                 return console.log(err.message);

//             }

//             console.log('Random dog image saved to file')

//         });

//     })

//     .catch(err => {

//         console.log('errors...', err);

//     });

// });

====**Consuming Promises with Async/Await**

index.js

const fs = require('fs');

const superagent = require('superagent');

// Create Arrow func to return new Promise that fs.readFile

const readFilePro = (filePath) => {

    return new Promise((resolve, reject) => {

// executer func

        fs.readFile(filePath, (err, fileContent) => {

        // if there's an error --> reject() --> piped err to .catch()

        if (err) reject('Could NOT find file');

        // Promise returns data to us will be piped into .then()

        resolve(fileContent);

        })

    });

}

const writeFilePro = (filePath, data) => {

    return new Promise((resolve, reject) => {

        fs.writeFile(file, data, err => {

            if (err) reject('Could NOT write file');

            // fs.writeFile doesn't need to return data

            resolve('write succeeded');

        })

    });

}

// Async func lets other tasks keep running in Event Loop

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const data = await readFilePro(`${\_\_dirname}/dog-1.txt`);

console.log(`\n`);

        console.log(`Breed: ${data}`);

console.log(`\n`);

        const res = await superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        console.log('res.body.message', res.body.message);

console.log(`\n`);

        await writeFilePro('dog-img.txt', res.body.message);

        console.log('Random dog image saved to file');

    } catch (err) {

        console.log(err);

    }

}

getDogPic();

==== **How Async Await works behind the scene**

const fs = require('fs');

const superagent = require('superagent');

// Create Arrow func to return new Promise that fs.readFile

const readFilePro = (filePath) => {

    return new Promise((resolve, reject) => { // executer func

        fs.readFile(filePath, (err, fileContent) => {

            // if there's an error --> reject() --> piped into .catch()

            if (err) reject('Could NOT find file');

            // Promise returns data to us will be piped into .then()

            resolve(fileContent);

        })

    });

}

const writeFilePro = (file, data) => {

    return new Promise((resolve, reject) => {

        fs.writeFile(file, data, err => {

            if (err) reject('Could NOT write file');

            // fs.writeFile doesn't need to return data

            resolve('write succeeded');

        })

    })

}

// Async func lets other tasks keep running in Event Loop

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const data = await readFilePro(`${\_\_dirname}/dog-1.txt`);

console.log(`\n`);

        console.log(`Breed: ${data}`);

console.log(`\n`);

        const res = await superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        console.log('res.body.message', res.body.message);

        await writeFilePro('dog-img.txt', res.body.message);

        console.log('Random dog image saved to file');

    } catch (err) {

        console.log(err);

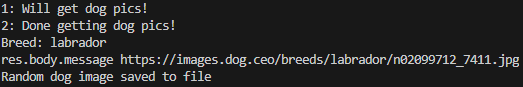
    }

}

console.log('1: Will get dog pics!');

getDogPic();

console.log('2: Done getting dog pics!');



// Async func lets other tasks keep running in Event Loop

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const data = await readFilePro(`${\_\_dirname}/dog-1.txt`);

console.log(`\n`);

        console.log(`Breed: ${data}`);

console.log(`\n`);

        const res = await superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        console.log('res.body.message', res.body.message);

console.log(`\n`);

        await writeFilePro('dog-img.txt', res.body.message);

        console.log('Random dog image saved to file');

    } catch (err) {

        console.log(`Error writing file as a promise:\n${err}\n`);

    }

console.log(`Returning something:`);

    return '2: READY';

};

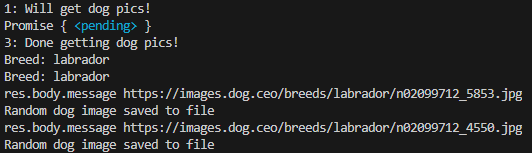
console.log('1: Will get dog pics!');

const x = getDogPic();

console.log(x);

getDogPic();

console.log('3: Done getting dog pics!');



// We'd get Promise { <pending> }, rather the string '2: READY';

// instead of logging '2: READY', const x (Async func) is still running

// Async func lets other tasks keep running in Event Loop

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const data = await readFilePro(`${\_\_dirname}/dog-1.txt`);

        console.log(`Breed: ${data}`);

console.log(`\n`);

        const res = await superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        console.log('res.body.message', res.body.message);

console.log(`\n`);

        await writeFilePro('dog-img.txt', res.body.message);

        console.log('Random dog image saved to file');

    } catch (err) {

        console.log(`Error node fetching API:\n${err}\n`);

    }

    return '2: READY';

};

console.log('1: Will get dog pics!');

getDogPic()

.then(x => {

    console.log(x);

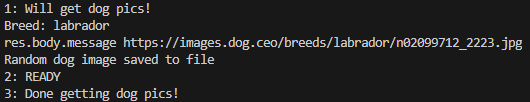
    console.log('3: Done getting dog pics!');

})

.catch(err => {

    console.log('Error!');

});



// We can use Immediately Invoke Function Execution

// instead of Flat Async.then().catch()

// Use IIFE

(async () => {

    try {

        console.log('1: Will get dog pics!');

        const x = await getDogPic();

        console.log(x);

    } catch(err) {

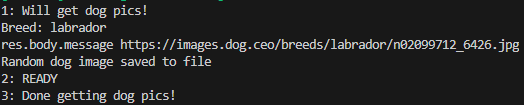
        console.log(`Error!\n${err}\n`);

    } finally {

        console.log('3: Done getting dog pics!');

    }

}) ();



Async funcs called from other Async funcs

**\*\* Async func() => {...} automatically returns a Promise**

**\*\* Value returned from an Async func = Resolved value of Promise**

==== **Waiting for Multiple Promises simultaneously**

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const fileContent = await readFilePro(`${\_\_dirname}/dog-1.txt`);

        console.log(`Breed: ${fileContent}`);

console.log(`\n`);

        // Suppose we wanna get 3 random Dog images at the same time

        // Storing a const resPro = superagent.get(`url`);

        // will NOT get us a resolved Promise

        const res1Pro = superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        const res2Pro = superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        const res3Pro = superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

        const all = await Promise.all([res1Pro, res2Pro, res3Pro]);

        const imgs = all.map(element => {

console.log('element.body:\n', element.body);

console.log('element.body.message:\n', element.body.message);

console.log(`\n`);

          return element.body.message;

        })

        console.log('imgs\n', imgs);

        // Writing the 3 images to './dog-img.txt'

        // join images each by a new line

        await writeFilePro('dog-img.txt', imgs.join('\n'));

        console.log('Random dog image saved to file');

    } catch (err) {

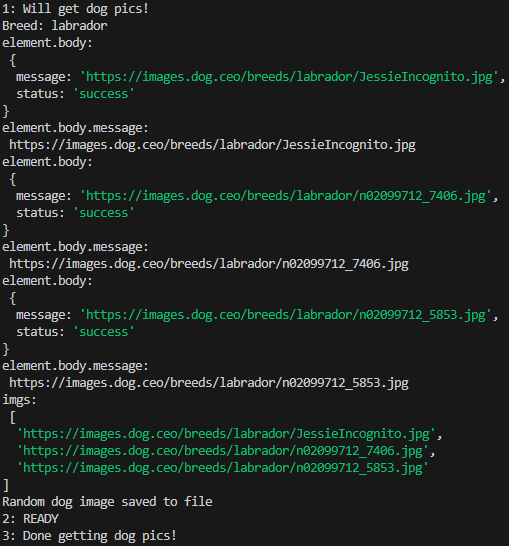
        console.log(`Error:\n${err}\n`);

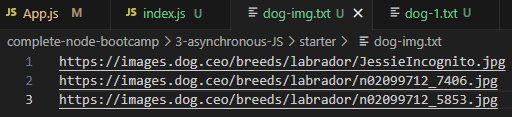
        throw Error;

    }

    return '2: READY';

};





// Solution 2

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const data = await readFilePro(`${\_\_dirname}/dog-1.txt`);

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

// Suppose we wanna get 3 random Dog images at the same time

        // Storing a const resPro = superagent.get(`url`);

        // will NOT get us a resolved Promise

// Store urls as an array

const urls = [

     `https://dog.ceo/api/breed/${data}/images/random`,

     `https://dog.ceo/api/breed/${data}/images/random`,

     `https://dog.ceo/api/breed/${data}/images/random`

   ];

// Destructuring urls to Promise.all => fetch each url

const [res1Pro, res2Pro, res3Pro] = await Promise.all(urls.map(url =>

            fetch(url).then(res => res.json())))

            console.log('res1Pro.message: \n', res1Pro.message)

            console.log('res2Pro.message: \n', res2Pro.message)

            console.log('res3Pro.message: \n', res3Pro.message)

    // Using template strings to writeFilePromise for each url

    await writeFilePro('./dog-img.txt', `${res1Pro.message}\n${res2Pro.message}\n${res3Pro.message}\n`);

   } catch (err) {

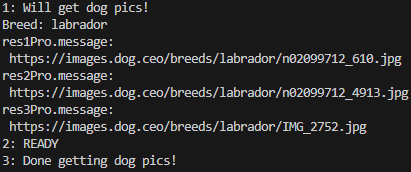
      console.log(err);

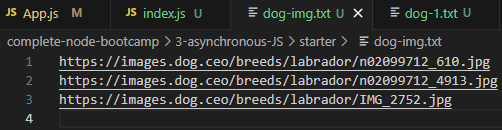
      throw Error;

   }

   return '2: READY';

};





==== **Entire index.js:**

const fs = require('fs');

const superagent = require('superagent');

// Create Arrow func to return new Promise that fs.readFile

const readFilePro = (filePath) => {

    return new Promise((resolve, reject) => { // executer func

        fs.readFile(filePath, (err, fileContent) => {

        // if there's an error --> reject() --> piped into .catch()

        if (err) reject('Could NOT find file');

        // Promise returns data to us will be piped into .then()

        resolve(fileContent);

        })

    });

}

// Create Arrow func to return new Promise that fs.writeFile

const writeFilePro = (filePath, data) => {

    return new Promise((resolve, reject) => {

        fs.writeFile(filePath, data, err => {

            if (err) reject('Could NOT write file');

            // fs.writeFile doesn't need to return data

            resolve('write succeeded');

        })

    })

}

// Async func lets other tasks keep running in Event Loop

const getDogPic = async () => {

    try {

        // Stop operation of this line below, until it returns

        // & finally stores results to const data

        const data = await readFilePro(`${\_\_dirname}/dog-1.txt`);

        console.log(`Breed: ${data}`);

        // Suppose we wanna get 3 random Dog images at the same time

        // Storing a const resPro = superagent.get(`url`);

        // will NOT get us a resolved Promise

        // Solution 1

        const res1Pro = superagent.get(

            `https://dog.ceo/api/breed/${data}/images/random`

        );

        const res2Pro = superagent.get(

            `https://dog.ceo/api/breed/${data}/images/random`

        );

        const res3Pro = superagent.get(

            `https://dog.ceo/api/breed/${data}/images/random`

        );

        const all = await Promise.all([res1Pro, res2Pro, res3Pro]);

        const imgs = all.map(element => {

            // console.log('element: \n', element);

            console.log('element.body: \n', element.body);

            console.log('element.body.message: \n', element.body.message);

            return element.body.message;

        });

        console.log('imgs: \n', imgs);

        await writeFilePro('./dog-img.txt', imgs.join('\n'));

        console.log('Random dog image saved to file');

        // Solution 2

        // Store urls as an array

        // const urls = [

        //     `https://dog.ceo/api/breed/${data}/images/random`,

        //     `https://dog.ceo/api/breed/${data}/images/random`,

        //     `https://dog.ceo/api/breed/${data}/images/random`

        // ];

        // // Destructuring urls to Promise.all => fetch each url

        // const [res1Pro, res2Pro, res3Pro] = await Promise.all(urls.map(url =>

        //     fetch(url).then(res => res.json())))

        //     console.log('res1Pro.message: \n', res1Pro.message)

        //     console.log('res2Pro.message: \n', res2Pro.message)

        //     console.log('res3Pro.message: \n', res3Pro.message)

        // // Using template strings to writeFilePromise for each url

        // await writeFilePro('./dog-img.txt', `${res1Pro.message}\n${res2Pro.message}\n${res3Pro.message}\n`);

        // For 1 url only

        // console.log('all Promise.all\n', all);

        // console.log('res.body.message', res.body.message);

        // await writeFilePro('dog-img.txt', res.body.message);

        // Writing the 3 images to './dog-img.txt'

        // join images each by a new line

    } catch (err) {

        console.log(err);

        throw Error;

    }

    return '2: READY';

};

/\*

console.log('1: Will get dog pics!');

// const x = getDogPic();

// console.log(x);

// getDogPic();

getDogPic()

.then(x => {

    console.log(x);

    console.log('3: Done getting dog pics!');

})

.catch(err => {

    console.log('Error!');

});

\*/

// Use IIFE

(async () => {

    try {

        console.log('1: Will get dog pics!');

        const x = await getDogPic();

        console.log(x);

    } catch(err) {

        console.log('Error!\n', err);

    } finally {

        console.log('3: Done getting dog pics!');

    }

}) ();

/\*

// readFilePro(fileName)

// return a Promise before calling each of them

readFilePro(`${\_\_dirname}/dog-1.txt`)

.then(data => {

    console.log(`Breed: ${data}`);

    // To keep chaining .then(), must return a Promise

    // return a Promise before calling each of them

    return superagent.get(`https://dog.ceo/api/breed/${data}/images/random`);

})

.then(res => {

    console.log('res.body.message', res.body.message);

    // To keep chaining .then(), must return a Promise

    // return a Promise before calling each of them

    return writeFilePro('dog-img.txt', res.body.message)

    // fs.writeFile('dog-img.txt', res.body.message, err => {

    //     if (err) return console.log(err.message);

    //     console.log('Random dog image saved to file')

    // });

})

.then(() => {

    console.log('Random dog image saved to file!');

})

.catch(err => {

    console.log('err', err);

});

\*/

Types of modules

3 kinds of modules:

**1. Custom**

[ps: should NOT add "type": "module", to package.json

if using require() ]

script.js:

const c = require('./script2.js');

const b = c.largeNumber;

const a = 5;

console.log('a + b', a+b);

script2.js:

const largeNumber = 356;

module.exports = {

largeNumber: largeNumber

};

====

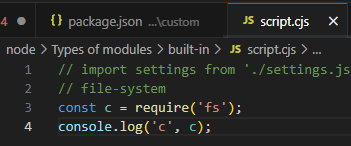
**2. Built-in modules**

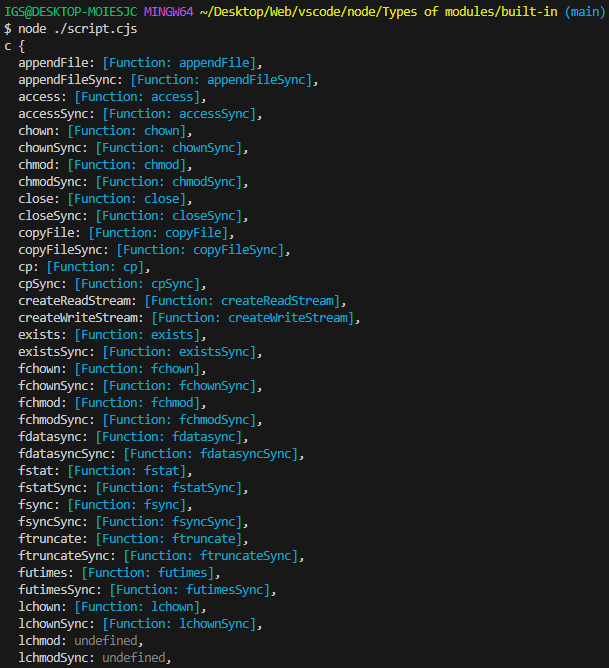
script.cjs:

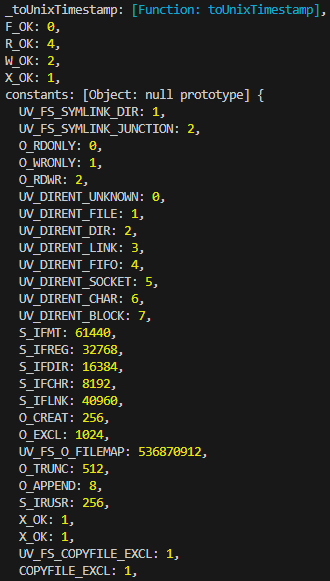
// file system

const c = require('fs');

console.log('c', c);





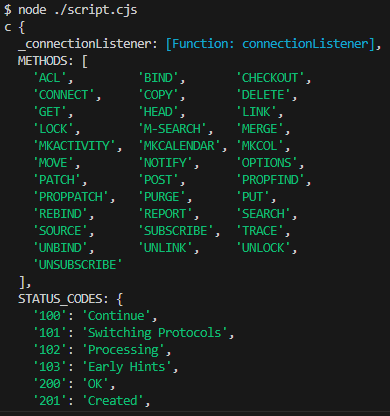


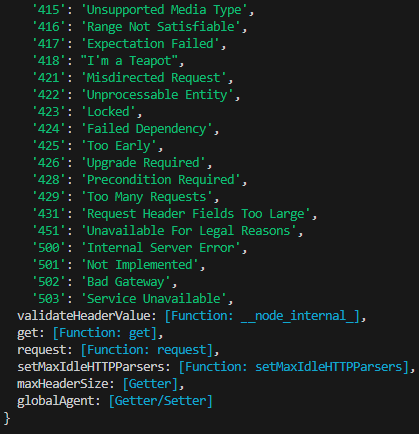
// Used for building API servers

const c = require('http');

console.log('c', c);







**3. NPM modules**

nodemon

npm init -y

npm install nodemon --save-dev

"devDependencies" are only used when developing



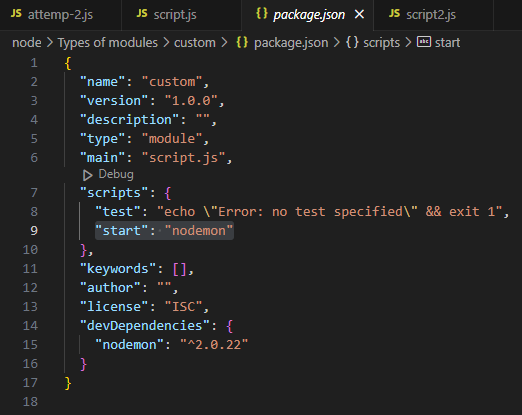
nodemon will not be included in build for Prod

Using vscode\node\Types of modules\custom

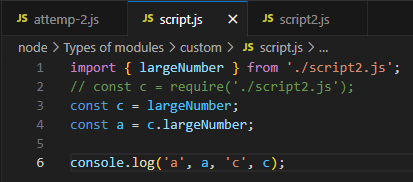
// Adding "script": {

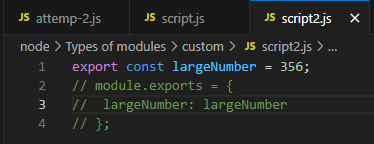
"start": "nodemon"

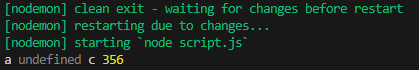
},



script.js & script2.js for testing:

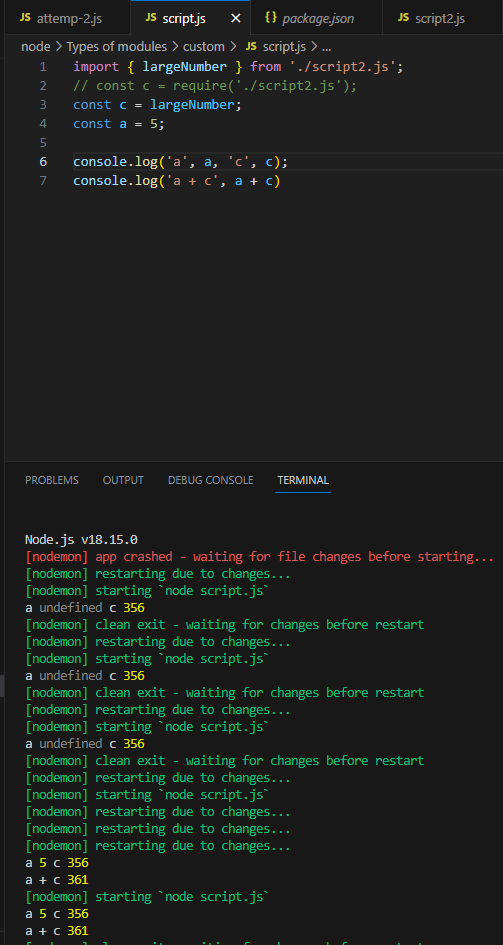






nodemon will keep listening to changes &&

output



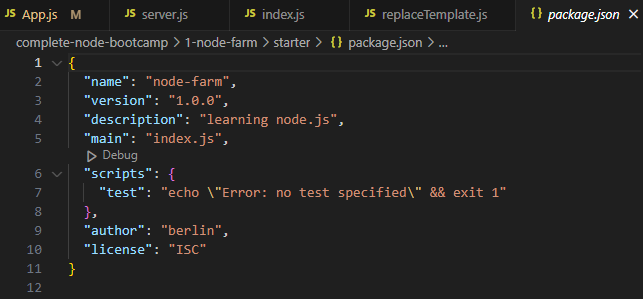
npm & package.json

npm = 3rd party package manager & command line interface app

npm: react, react-router-dom, ionic icons, express, ....

// Create jackage.json

npm init



====**2 Types & packages installs**

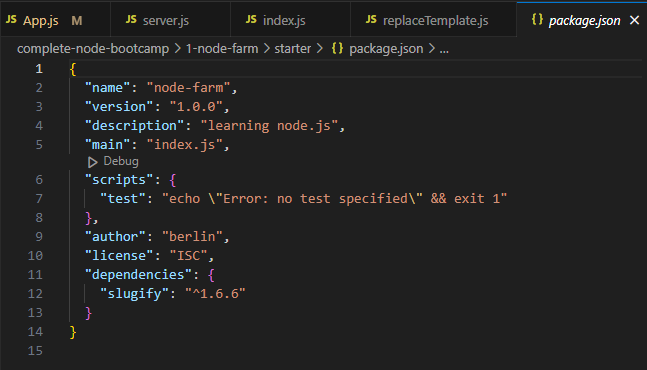
-Regular dependencies

-Dev dependencies

express.js = node framework (a dependency)

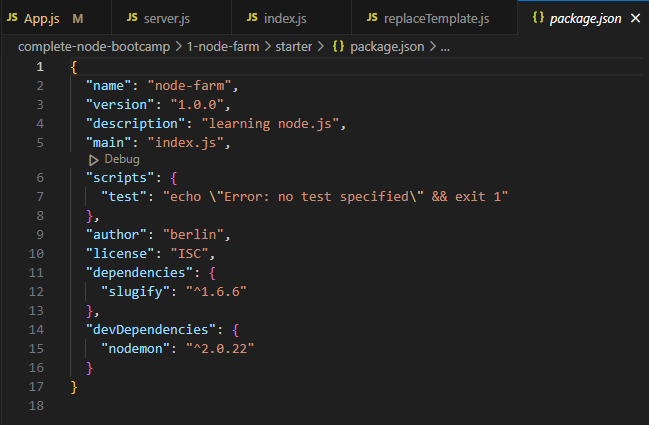
// Install 'slugify' as a regular dependency

npm install slugify



// Dev dependencies i.e. code bundler, webpack, debugger tool, test lib...

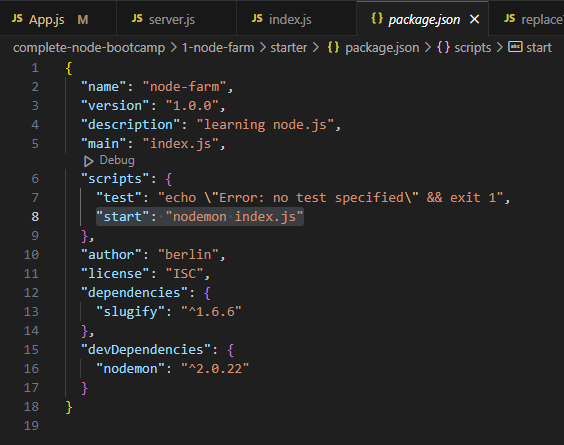
npm install nodemon --save-dev



// Global install nodemon

npm i nodemon -global

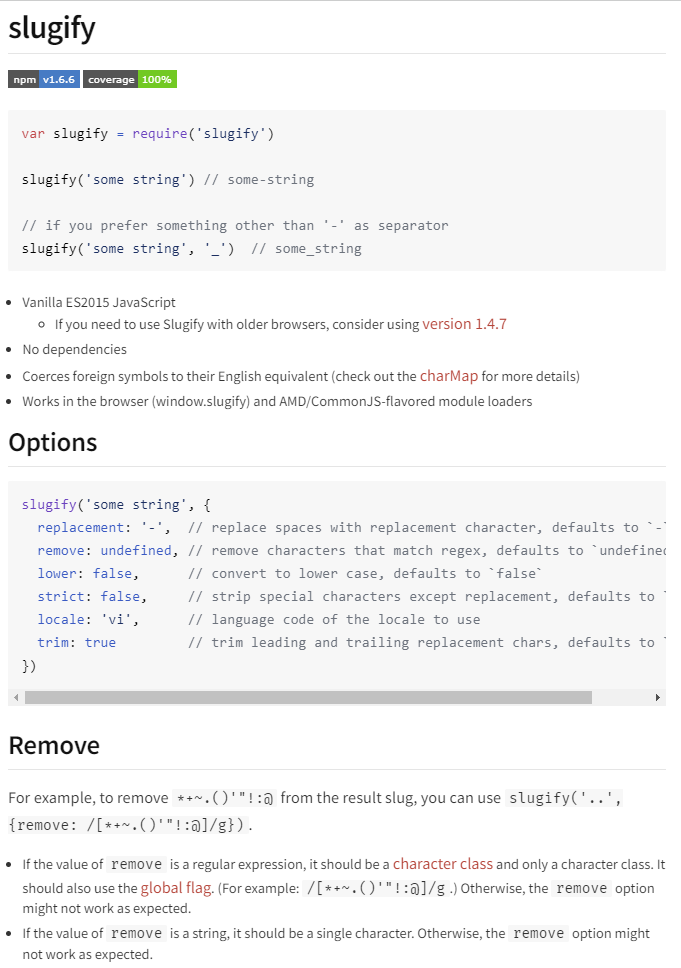
// Need to specify an npm script to use nodemon locally



====**Using 3rd party modules**

Slugify

<https://www.npmjs.com/package/slugify>



index.js:

// import local modules

const fs = require('fs');

// Networking capabilities => Building a HTTP server

const http = require('http');

// URL module

const url = require('url');

// import 3rd party module

const slugify = require('slugify');

// slugify can specify routes i.e. 127.0.0.1:8880/product/fresh-avocados

// instead of Query Strings /product?id=0

// import our custom module

const replaceTemplate = require('./modules/replaceTemplate');

**Package versioning & updating**

^major.minor.patch (version)

patch = fix bugs

minor = introducing new changes of backward compatibility

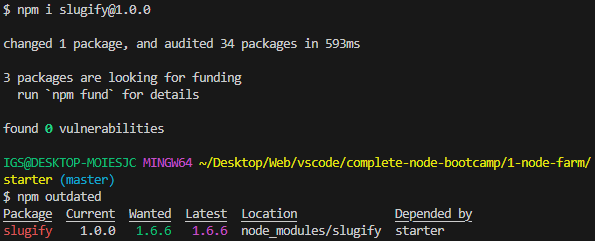
major = introducing new changes that can break codes

// Check any outdated packages

npm outdated

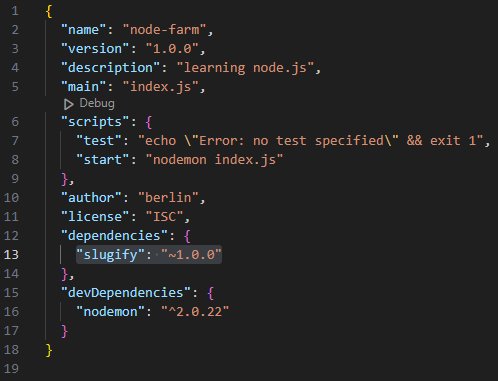
// Install old slugify

npm install slugify@1.0.0



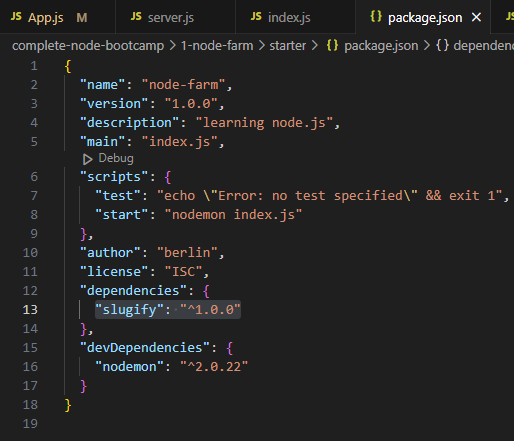
// Only accept patch releases

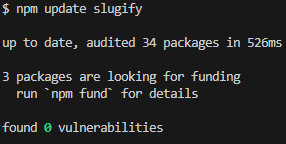
"slugify": "~1.0.0"



// To update a package

npm update slugify





// To update to all versions

"slugify": "\*1.3.4"

npm update slugify

// To install express => delete express

npm i express && npm uninstall express

// To uninstall slugify => install slugify

npm uninstall slugify && npm i slugify

// After uploading to github w/o node\_modules folder

// Download project dependencies back using package.json & package-lock.json

npm install

====**Setting up Prettier ext for vscode**

-DotENV

-ESLint

-Image preview in vscode for href

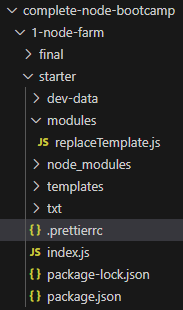
-Pug beautify (Pug is used to build templates)

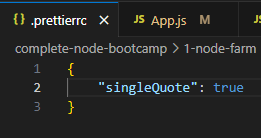
-TODO Highlight (TODO, BUG, FIXME)

-Theme - Oceanic Next

-Prettier - Code formatter

// Using a prettier config





====Web

1. Client on Browser => DNS => resolve domain name to IP => server

2. TCP/IP socket connection

3. Client => HTTP request => Server

|  |  |
| --- | --- |
| Get /maps (resource) HTTP/1.1 | => Start line: HTTP method + request target + HTTP version |
| HTTP request headers | many diff. possibilities |
| Host: [www.google.com](http://www.google.com) |  |
| User-Agent: Mozilla/5.0 |  |
| Accept-Language: en-US |  |
| <BODY> | => Request body (only when sending data to server e.g. POST) |

4. Client <= HTTP response <= Server

HTTP/1.1 200 OK

Date: new Date()

Content-Type: text/html

Transfer-Encoding: chunked

<BODY>

5.

index.html is the 1st to be loaded

Scanned for assets: JS, CSS, images

**Process is repeated for each file**

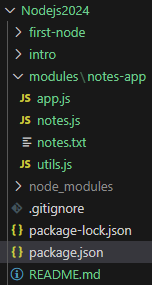
using npm 3rd party modules

Wednesday, February 28, 2024

12:01 PM

npm init -y;

package.json comes up



// Install locally

npm install validator@13.11.0;

// Install locally

npm install chalk@2.4.1;

// Install Nodemon globally

npm install nodemon@1.18.5 -g;

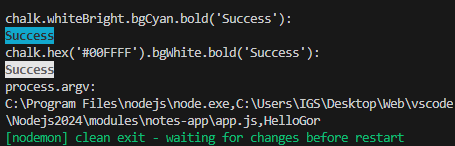
// Importing & Using 3rd party modules



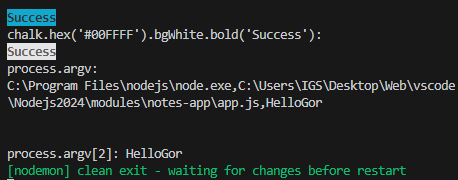
// Using nodemon with arguments

process.argv

nodemon app.js HelloGor;



console.log(`process.argv[2]: ${process.argv[2]}`);



16. File System module (fs)

// Read-Eval-Print-Loop (REPL)

// const repl = require('repl');

const fs = require('fs');

// Sync func can only be top-level code

const textInput = fs.readFileSync('./txt/input.txt', (err, data) => {

    if (err) {

        console.log('error: ', err)

    }

    return data;

})

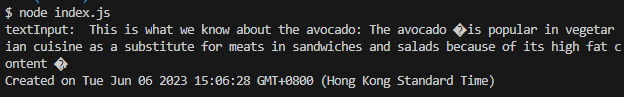
// const textOutput = textInput.toString();

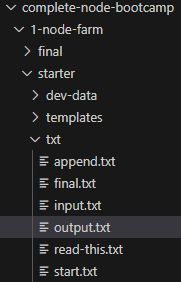
const textOutput = `This is what we know about the avocado: ${textInput}.\nCreated on ${new Date()}`;

console.log('textInput: ', textOutput);

// Save output to a file

fs.writeFileSync('./txt/output.txt', textOutput);





Node.js File System Module – Official docs

**File Stats**

Node.js File System module fs.state() method gets the file details it will call the callback function we pass, with **2 parameters**:

**Asynchronous** fs.stat()

fs.stat('pathToFile', (err, stats) => {

if (err) {

console.error(err);

}

// we have access the file stats in `stats`

});

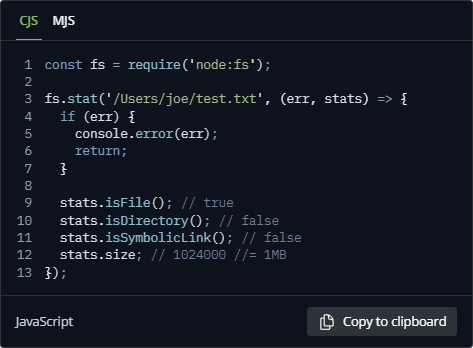


**Synchronous** fs.statSync()

[This blocks Event Loop]



Now, we can access the file information



**Promise-based**

const fs = require('**node:fs/promises**');



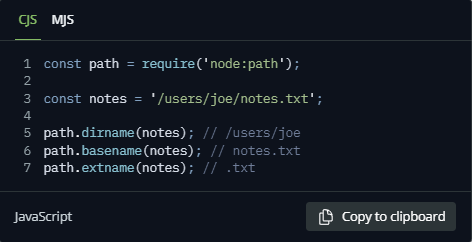
**Node.js File Paths**

We can also extract information using:

- **dirname**: gets the parent folder of a file

- **basename**: gets the filename part

- **extname**: gets the file extension



We can also get file name without extension

const path = require('node:path');

const notes = '/usr/joe/notes.txt';

path.dirname(notes); // /usr/joe/notes.txt

path.basename(notes); // notes.txt

path.extname(notes); // .txt

**To extract file name only**

path.basename(notes, path.extname(notes)); // notes.txt

path.basename(notes, .txt); // notes

**Path joining**

const name = 'user1';

**/usr/user1/notes.txt**

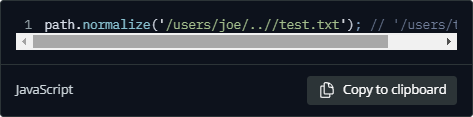
path.join('/', 'usr', name, 'notes.txt');

**/etc/joe.txt**

path.resolve('/etc', 'joe.txt');

**path.normalize()** is another useful function, that will try & calculate the actual path, when it contains relative specifiers like **.** or **. .** or double slashes:

path.normalize('/usr/joe/..//test.txt');



**\*\* Neither path.resolve() nor path.normalize() methods will check if the path exists \*\***

They just calculate a path based on the information they got.

**Working with File Descriptors in Node.js**

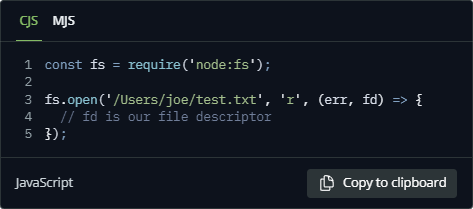
Before even we can read a file in our filesystem,

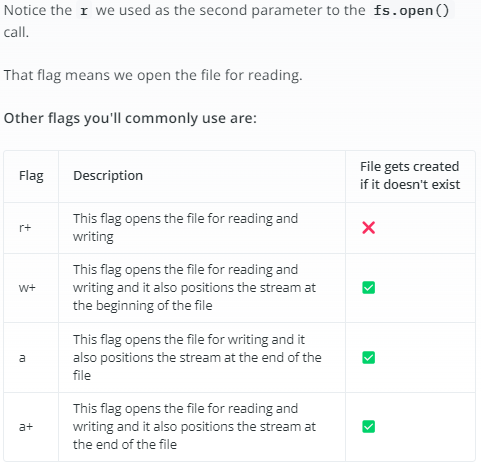
we must get a **file descriptor**.

File Descriptor = Reference to an open file

fd = a number a file descriptor (fd) returned by opening the file using **open()** method offered by **fs** module.

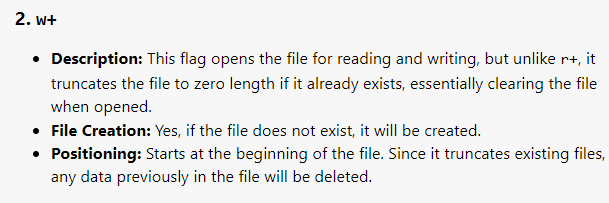
This number (**fd**) **uniquely identifies** an open file in OS





A close up of text

Description automatically generated



A black text on a white background

Description automatically generated

A text on a white background

Description automatically generated

r+ = does NOT create a new file

w+ a a+ = will create a new file if it does NOT exist

r+ and w+ start writing from the Start of file

a and a+ start writing from the End of file

**This Synchronous code gives us a unique identifier of a file**

**File Descriptor (fd)**

**instead of making it an Asynchronous callback**

A screenshot of a computer program

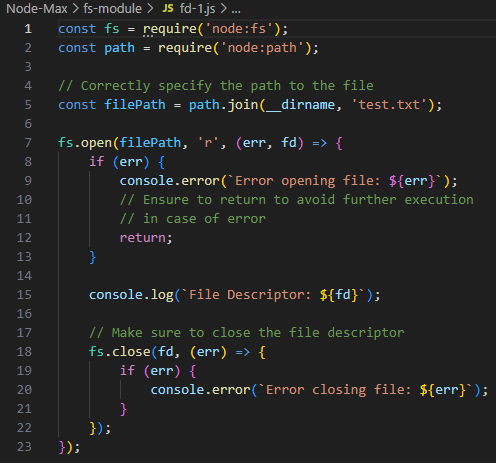
Description automatically generated

Acquiring the File Descriptor from a file ./test.txt



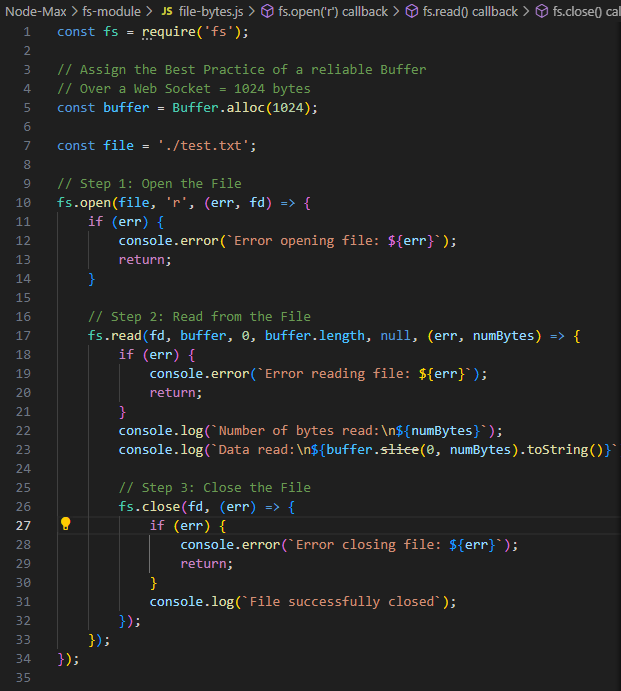
**This just console log the File Descriptor, doing nothing**

fd-1.js



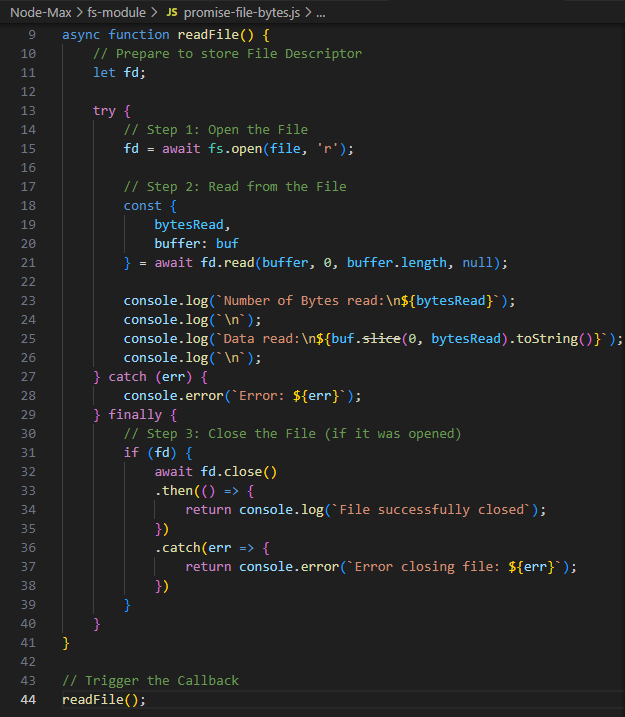
**Reading Number of File Bytes => Close the File**

file-bytes.js



**Promise-based Async Code to Open File => Acquire File Descriptor => Read contents => Close File**

promise-file-bytes.js



A screenshot of a computer program

Description automatically generated

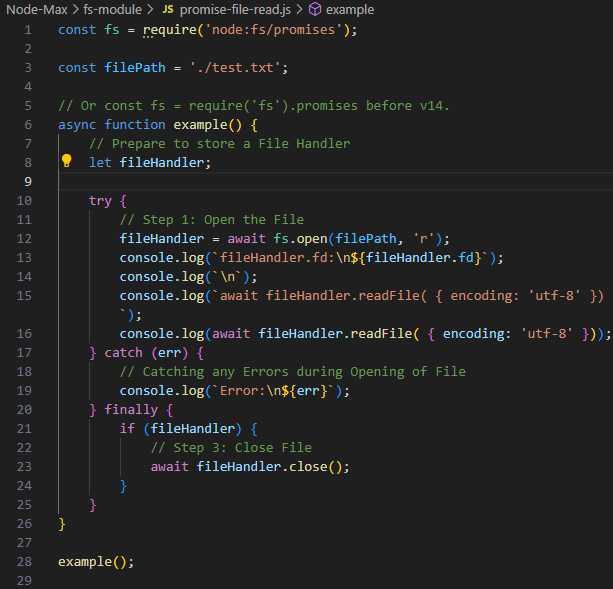
Checking against local File System properties

Size: 4 bytes

A screenshot of a computer

Description automatically generated

promise-file-read.js



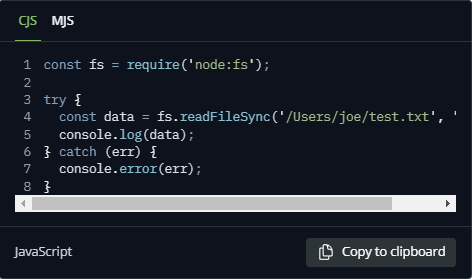
**Reading Files with Node.js**

Asynchronous code

A screen shot of a computer program

Description automatically generated

Synchronous code



Promise-based

A screen shot of a computer program

Description automatically generated

===================================================

**Writing files with Node.js**

Async



Synchronous

A screen shot of a computer program

Description automatically generated

Promise-based



**We can modify the default by specifying a flag:**

const fs = require('node:fs/promises');

async function example() {

try {

const content = 'Some content';

await fs.writeFile('/usr/user1/test.txt', content, { flag: 'a+' }, err => {});

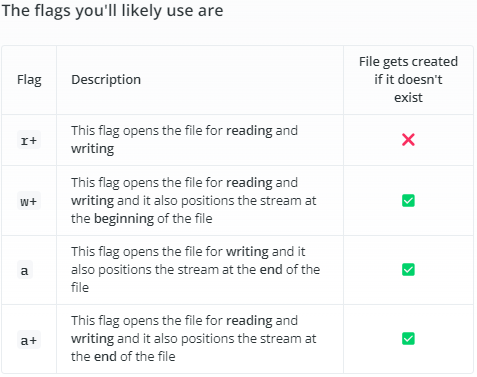
} catch (err) {

console.log(`Error: ${err}`);

}

}

example();



**Appending content to a file**

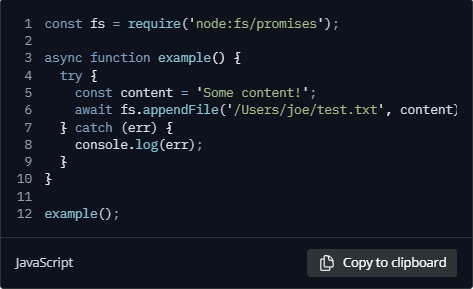
When we do NOT wanna overwrite a file with new content, but rather add to it.

Async

A screen shot of a computer program

Description automatically generated

Promise-based



=====================================

**Working with folders in Node.js**

Check if a folder exists

**Async**

fs.access()

**Promise-based**

fsPromises.access()

**Create a new folder**

A screenshot of a computer program

Description automatically generated

**Read the content of a directory**

To read contents of a directory

Async

fs.readdir()

A screenshot of a computer program

Description automatically generated

getting the full path

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Rename a folder**

Asynchronous

A screen shot of a computer program

Description automatically generated

Synchronous

A screen shot of a computer program

Description automatically generated

Promise-based

A screen shot of a computer program

Description automatically generated

**Remove a folder**

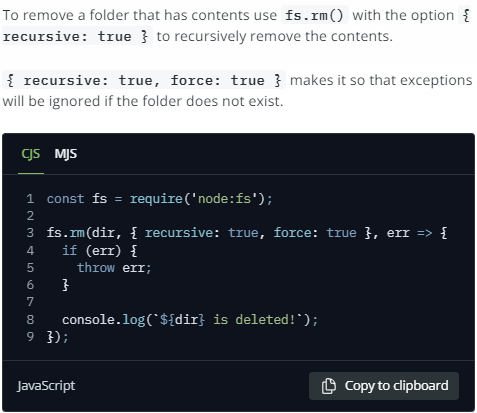
Asynchronous

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated



fs, http, url Routing

Sync = fs.readFileSync

Sync => Blocking (1 after another)

Async => Non-blocking => Heavy tasks as Callbacks

Heavy tasks:

-Networking

-File Access

const fs = require('fs');

// Sync

const textInput = fs.readFileSync('./txt/input.txt', (err, data) => {

    if (err) {

        console.log('error: ', err)

    }

    return data;

})

// const textOutput = textInput.toString();

const textOutput = `This is what we know about the avocado:\n${textInput}.\nCreated on ${new Date()}`;

console.log('textOutput: ', textOutput);

// Save output to a file

// fs.writeFileSync('./txt/output.txt', textOutput);

// Async - Non-blocking I/O & Network requests

// PHP - 1 new thread for each user

// Callbacks != Async

// Callbacks are NOT Asynchronous by nature

fs.readFile('./txt/input.txt', 'utf-8', (err, data) => {

    if (err) {

        console.log('error: ', err);

    }

    console.log('data: ', data);

    return data;

});

ES6 Promises || ES8 Async Await to tackle Callback hell

====**Reading & Writing files asynchronously**

// Async - Non-blocking I/O & Network requests

// PHP - 1 new thread for each user

// Callbacks != Async

// As soon as node reads this line => move on

fs.readFile('./txt/start.txt', 'utf-8', (err, data) => {

    if (err) console.log('error: ', err);

    console.log('data: ', data);

    return data;

});

// Finish reading this line first

console.log('Wil read file');



----

// Callback hell

fs.readFile('./txt/start.txt', 'utf-8', (err, data1) => {

    if (err) return console.log('error: ', err);

    console.log('data1: ', data1);

    // return data1;

    fs.readFile(`./txt/${data1}.txt`, 'utf-8', (err, data2) => {

        if (err) return console.log('error2: ', err);

        console.log('data2: ', data2);

        fs.readFile('./txt/append.txt', 'utf-8', (err, data3) => {

            if (err) return console.log('error3: ', err);

            console.log('data3: ', data3);

            fs.writeFile('./txt/final.txt', `${data2}\n${data3}`, 'utf-8', err => {

                if (err) return console.log('error: ', err);

                console.log('File has been written');

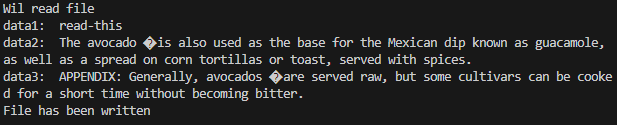
            })

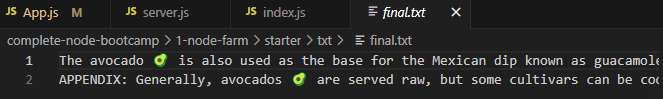
        })

    })

});

console.log('Wil read file');





====**Creating a simple server**

const fs = require('fs');

// Networking capabilities => Building a HTTP server

const http = require('http');

// Server

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

    console.log('req.body ', req.body);

    res.end('Hello from server')

});

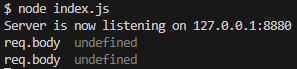
// server.listen(port, loalhost||loopback address)

server.listen(port=8880, localhost='127.0.0.1', () => {

    console.log(`Server is now listening on ${localhost}:${port}`);

})

// Listens to incoming requests from clients





// After editing the script => ctrl + C => node script.js again

====**Routing**

const fs = require('fs');

// Networking capabilities => Building a HTTP server

const http = require('http');

// URL module

const url = require('url');

// Server

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

    const pathName = req.url;

    if (pathName === '/' || pathName === '/overview') {

        res.end('This is the overview');

    } else if (pathName === '/product') {

        res.end('This is the product page');

    } else {

        res.writeHead(404);

        res.end('Page not found');

    }

});

// server.listen(port, loalhost||loopback address)

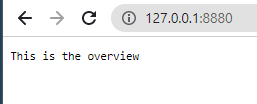
server.listen(port=8880, localhost='127.0.0.1', () => {

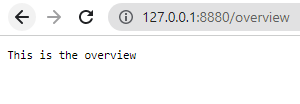
    console.log(`Server is now listening on ${localhost}:${port}`);

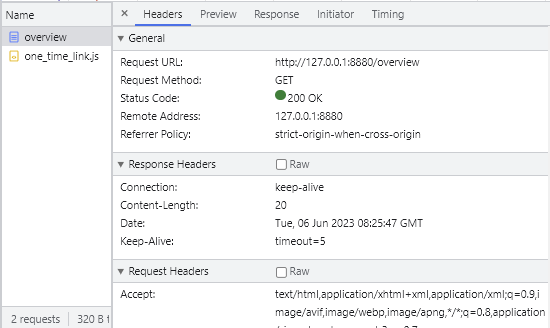
})

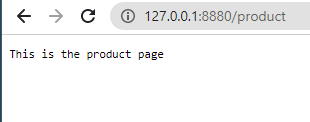
// Listens to incoming requests from clients

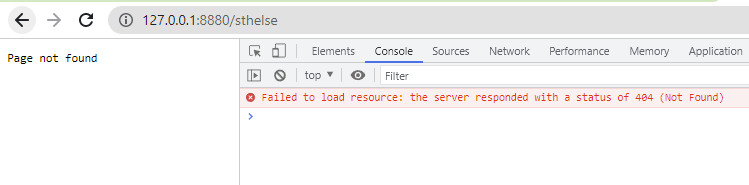
// 127.0.0.1:8880/product

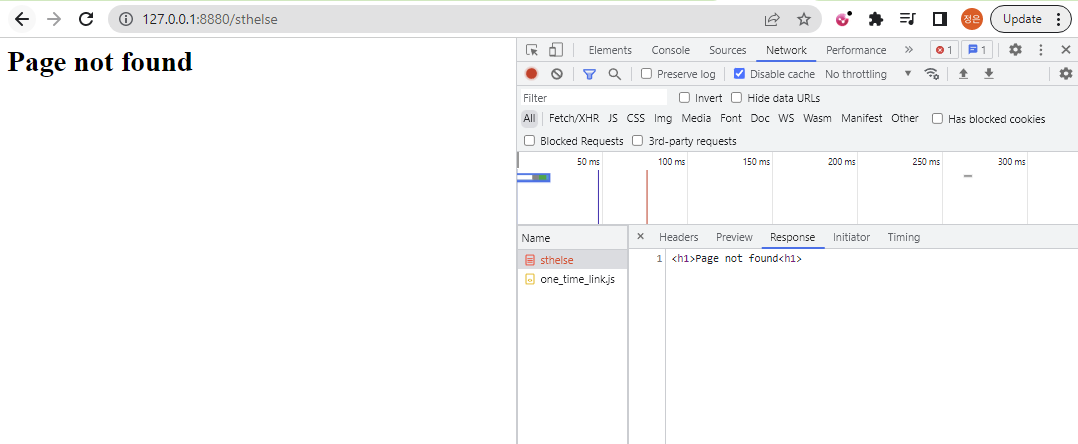


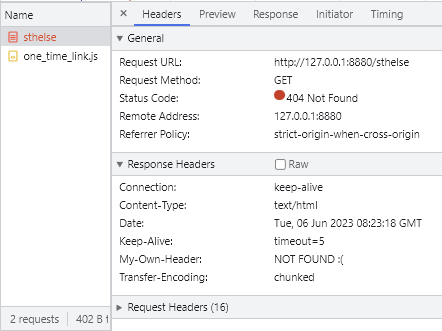












18. how Require() modules work

-Each JS file is treated as a separate module;

-Node.js uses **CommonJS module system**:

require()

exports

module.exports

-**ES module system** used in browsers: import/export;

-Native ES modules (.mjs):

Not popular using .mjs

require('module');

**Where does it come from & How does it work behind the scenes?**

1. Resolving & Loading:

3 types of modules (as mentioned)

i. Core modules:

require('http');

ii. Developer mdules:

require('./lib/controller');

iii. 3rd-party modules (from NPM):

require('express');

Mechanism:

Path Resolving: **How Node decides which module to load**

1. Start with **core modules**

2. If begins with '**./**' or '**../**' => Try to **load developer modules**;

3. If no file found => Try to **find folder** with index.js in it;

4. Else => Go to **node\_modules/** & try to find module there

2. Wrapping:

require();

// Like global variables injected

// Node puts require() into Immediately Invoked Functional Expression (IIFE) that returns objects

(function(exports, require, module, \_\_filename, \_\_dirname) {

// Module code lives here...

});

// This keeps 'top-level' variables we defined in our modules private

// Scoped only to current module

// will NOT mess up with npm modules

**require**: function to require modules;

**module**: reference to the current module;

**exports**: a reference to module.exports, used to export obj from a module;

**\_\_filename**: absolute path of the current module's file;

**\_\_dirname**: directory name of the current module (current dir)

3. Execution:

Where the modules get executed by Node.js runtime

4. Returning exports:

-require function in our module => required module **exports**

-module.exports is the returned object (in required modules)

-Use module.exports to export 1 single variable

i.e.

(-module.exports = Calculator);

-Use exports to export multiple named variables

(exports.add = (a, b) => a + b);

-This is how we import data from 1 module into another;

5. Caching:

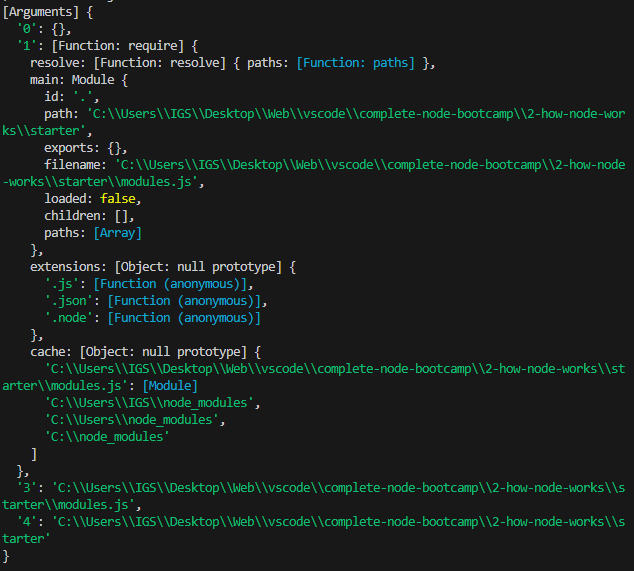
Required modules are only executed at the 1st call => Cached to memory & wait to be called again

====**Requiring Modules in Practice**

modules.js:

// arguments = array in JS are objs we pass into a func

console.log(arguments);



0: exports

empty cuz we have NOT exported any modules yet

1: require()

2: module

3: \_\_filename

'C:\\Users\\IGS\\Desktop\\Web\\vscode\\complete-node-bootcamp\\2-how-node-works\\starter\\modules.js',

4: \_\_dirname

'C:\\Users\\IGS\\Desktop\\Web\\vscode\\complete-node-bootcamp\\2-how-node-works\\starter'

wrapper function:

console.log(require('module').wrapper);

[

'(function (exports, require, module, \_\_filename, \_\_dirname) { ',

'\n});'

]

Exporting a class variable:

test-module-1.js:

class Calculator {

    add(a, b) {

        return a+b

    }

    multiply(a, b) {

        return a\*b

    }

    devide(a,b) {

        return a/b

    }

    remainder(a,b) {

        return a%b

    }

}

module.exports = Calculator;

modules.js:

const C = require('./test-module-1');

const calc1 = new C();

console.log(calc1.add(2, 5));



test-module-1.js:

// Anonymous module.exports

module.exports = class {

    add(a, b) {

        return a+b

    }

    multiply(a, b) {

        return a\*b

    }

    devide(a,b) {

        return a/b

    }

    remainder(a,b) {

        return a%b

    }

};



==== **When to use 'exports '**

test-module-2.js:

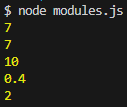
// exports.xx = (argu1, argu2) => {...}

exports.add = (a, b) => a+b;

exports.multiply = (a, b) => a\*b;

exports.divide = (a, b) => a/b;

exports.remainder = (a, b) => a%b;



// ES6 destructuring:

// exports - ES6 destructuring

const { add, multiply, divide, remainder } = require('./test-module-2');

console.log(add(2, 5));

console.log(multiply(2, 5));

console.log(divide(2, 5));

console.log(remainder(2, 5));

modules.js:

// exports

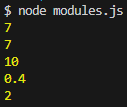
const calc2 = require('./test-module-2');

console.log(calc2.add(2, 5));

console.log(calc2.multiply(2, 5));

console.log(calc2.divide(2, 5));

console.log(calc2.remainder(2, 5));



// Caching

modules.js:

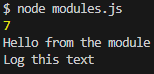
// Caching

require('./test-module-3')();

test-module-3.js:

console.log('Hello from the module');

module.exports = () => console.log('Log this text');

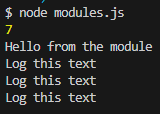


modules.js:

require('./test-module-3')();

require('./test-module-3')();

require('./test-module-3')();



// Technically, test-module-3.js only loaded once

// module.exports has exported the function to be cached & called

19. EventEmitter

Events:

1. EventEmitter

2. EventListener

// Import events modules

const EventEmitter = require('events');

// Listen to Events => React accordingly

// Similar to setting up an Event Listener that listens to Button clicks

// myEmitter will eventually emit a named event

const myEmitter = new EventEmitter();

// myEmitter.on = Observers

// Observers listen to Events

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('There was a new sale!')

})

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('Customer name: Jonas');

})

myEmitter.on('newSale', stock => {

    // amount of items left

    console.log(`There are now ${stock} items left in stock.`);

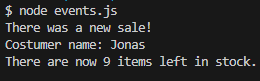
})

// Emitter

// An online store

// can pass 2nd arguments into an Emitter

myEmitter.emit('newSale', 9); // as if we're clicking on a button



// myEmitter.on are run Synchronously

Using ES6 class to construct a new class named 'Sales'

const EventEmitter = require('events');

// Listen to Events => React accordingly

// Similar to setting up an Event Listener that listens to Button clicks

// myEmitter will eventually emit a named event

// const myEmitter = new EventEmitter();

// ES6

// class Sales inherits all classes from EventEmitter class

class Sales extends EventEmitter {

    constructor() {

        super();

    }

}

const myEmitter = new Sales();

// myEmitter.on = Observers

// Observers listen to Events

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('There was a new sale!')

})

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('Costumer name: Jonas');

})

myEmitter.on('newSale', stock => {

    // amount of items left

    console.log(`There are now ${stock} items left in stock.`);

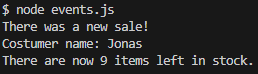
})

// Emitter

// An online store or something

// can pass 2nd arguments into an Emitter

myEmitter.emit('newSale', 9); // as if we're clicking on a button



Creating a small Web Server that listens to Events it emits

const EventEmitter = require('events');

const http = require('http');

// Listen to Events => React accordingly

// Similar to setting up an Event Listener that listens to Button clicks

// myEmitter will eventually emit a named event

// const myEmitter = new EventEmitter();

// class Sales inherits all classes from EventEmitter class

// http, fs modules all implement inheritance of EventEmitter internally

class Sales extends EventEmitter {

    constructor() {

        super();

    }

}

const myEmitter = new Sales();

// myEmitter.on = Observers

// Observers listen to Events

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('There was a new sale!')

})

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('Costumer name: Jonas');

})

myEmitter.on('newSale', stock => {

    // amount of items left

    console.log(`There are now ${stock} items left in stock.`);

})

// Emitter

// An online store or something

// can pass 2nd arguments into an Emitter

myEmitter.emit('newSale', 9); // as if we're clicking on a button

// ======Create a small web server that listens to Events that it emits

const server = http.createServer();

// Listens to different Events that the server will emit

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

    console.log('Request received! ');

    res.end('Request received');

})

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

    console.log('Another request received! ');

    res.end('Another request received');

})

// Listens to Server shutdown

server.on('Close', () => {

    console.log('Server closed');

})

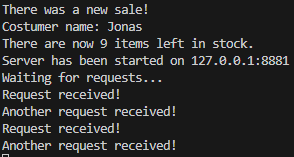
const localhost = '127.0.0.1';

const port = 8881;

server.listen(port, localhost, () => {

    console.log(`Server has been started on ${localhost}:${port}\nWaiting for requests...`);

})





====**Creating a small Web Server that listens to Events it emits**

const EventEmitter = require('events');

const http = require('http');

const url = require('url');

// Listen to Events => React accordingly

// Similar to setting up an Event Listener that listens to Button clicks

// myEmitter will eventually emit a named event

// const myEmitter = new EventEmitter();

// class Sales inherits all classes from EventEmitter class

// ES6

// http, fs modules all implement inheritance of EventEmitter internally

class Sales extends EventEmitter {

    constructor() {

        super();

    }

}

const myEmitter = new Sales();

// myEmitter.on = Observers

// Observers listen to Events

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('There was a new sale!')

})

// myEmitter listens on newSale event, followed by a Callback func

myEmitter.on('newSale', () => {

    console.log('Costumer name: Jonas');

})

myEmitter.on('newSale', stock => {

    // amount of items left

    console.log(`There are now ${stock} items left in stock.`);

})

// Emitter

// An online store or something

// can pass 2nd arguments into an Emitter

myEmitter.emit('newSale', 9); // as if we're clicking on a button

// ======Create a small web server that listens to Events that it emits

const server = http.createServer();

// Listens to different Events that the server will emit

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

    console.log('Request received! ');

    console.log(req.url);

    res.end('Request received'); // Can only send 1 response

})

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

    console.log('Another request received! ');

})

// Listens to Server shutdown

// Server will NOT shutdown as long as it's still listening on Events

server.on('close', () => {

    console.log('Server closed');

})

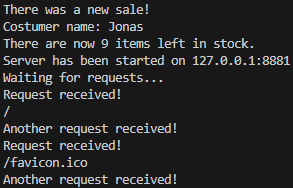
const localhost = '127.0.0.1';

const port = 8881;

server.listen(port, localhost, () => {

    console.log(`Server has been started on ${localhost}:${port}\nWaiting for requests...`);

})



// Browser tries to request a favicon.icon for each website

=== Node - Max

Thursday, May 23, 2024

3:38 PM

3. Basics

Thursday, May 23, 2024

3:39 PM

Node.js Server

Node Core Modules

e.g. File System module

Working with Requests & Responses (Basics)

Async code & Event Loop



modules

http -> send & receive req

https -> SSL server

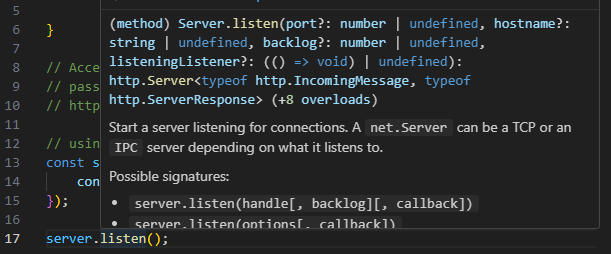
fs

path

os

Node server

1



app.js

// import a global module named http

const http = require('http');

// function rqListener(req, res) {

// }

// Accessing a method createServer to

// pass callback rqListener

// http.createServer(rqListener);

// using ES6 arrow

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

    console.log(req.url); // Log the request URL to the console

    res.statusCode = 200; // HTTP res status code

    res.setHeader('Content-Type', 'text/plain'); // Set head type

    res.end('Hello world\n'); // Send a res back to the client

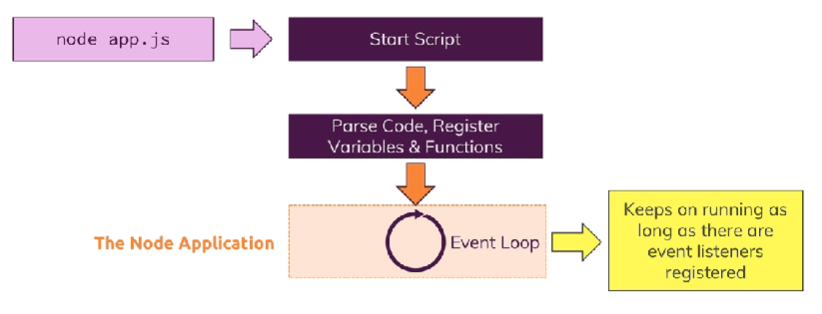
});

const port = 3005

server.listen(port, () => {

    console.log(`Server running at [http://localhost:${port}`](http://localhost:$%7bport%7d%60));

});



**Understanding Requests**

Logging out request method & request header

// using ES6 arrow

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

    console.log(req.url, req.method, req.headers); // Log the request URL to the console

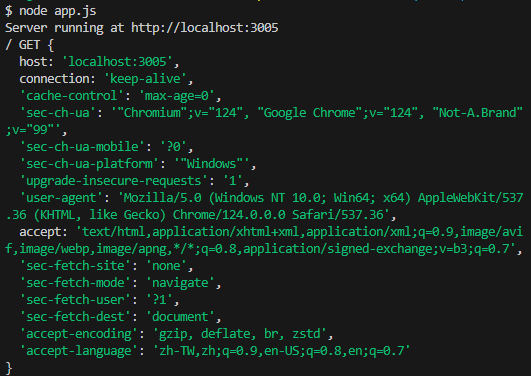
    res.statusCode = 200; // HTTP res status code

    res.setHeader('Content-Type', 'text/plain'); // Set head type

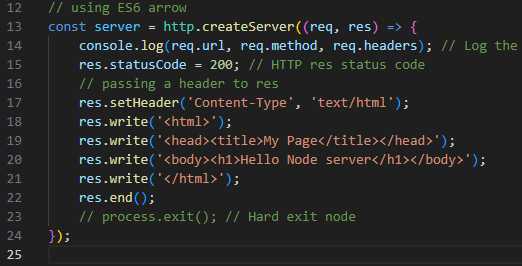
    res.end('Hello world\n'); // Send a res back to the client

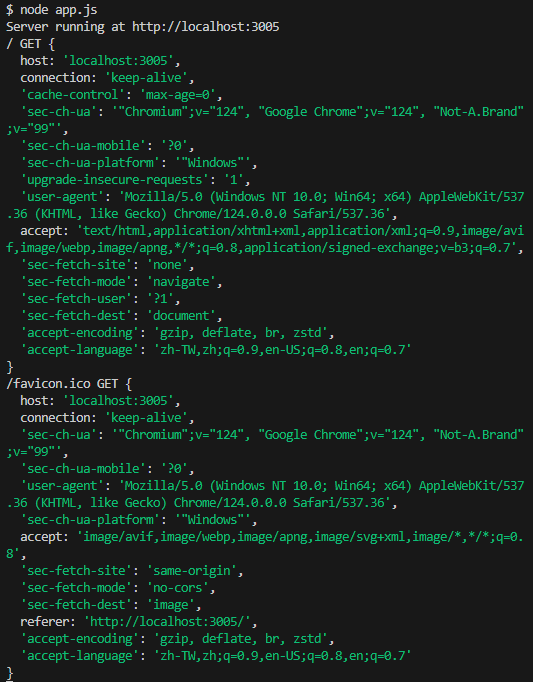
    // process.exit(); // Hard exit node

});

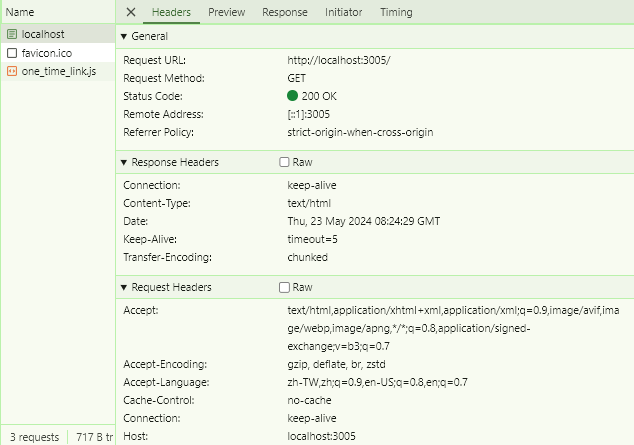


**Sending responses with a HTML**





Content-Type: text/html



**Routing requests**

app.js

// import a global module named http

const http = require('http');

// using ES6 arrow

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

    const url = req.url;

    if (url === '/') {

        console.log(req.url, req.method, req.headers); // Log the request URL to the console

        res.statusCode = 200; // HTTP res status code

        // passing a header to res

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

        res.write('<html>');

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

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

        res.write('</html>');

        // returning to stop subsequent code execution

        return res.end();

    }

    console.log(req.url, req.method, req.headers); // Log the request URL to the console

    res.statusCode = 201; // HTTP res status code for CREATED

    // passing a header to res

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

    res.write('<html>');

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

    res.write('<body><h1>Hello Node server</h1></body>');

    res.write('</html>');

    res.end();

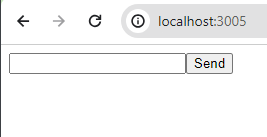
});

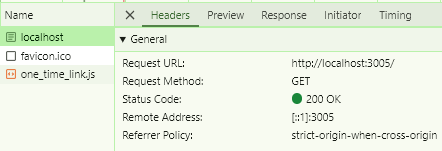
const port = 3005;

server.listen(port, () => {

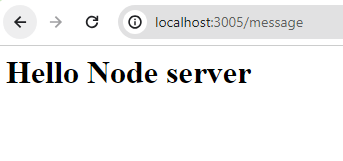
    console.log(`Server running at [http://localhost:${port}`](http://localhost:$%7bport%7d%60));

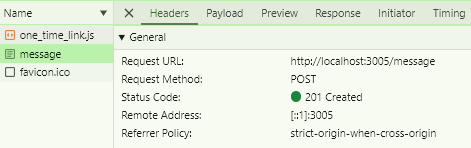
});





Once we submit any request with send button





Coding challenge

1. Save all submitted text into ./message folder

2. Upon saving ./message/text-1.txt -> redirect to /

app.js

// import a global module named http

const fs = require('fs');

const http = require('http');

const path = require('path');

// to format dateTime into 'yyyy-MM-dd\_\_HH\_mm\_ss'

const { format } = require('date-fns');

// using ES6 arrow

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

    const url = req.url;

    const method = req.method;

    if (url === '/') {

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

// Log the request URL to the console

        res.statusCode = 200; // HTTP res status code

        // passing a header to res

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

        res.write('<html>');

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

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

        res.write('</html>');

        // returning to stop subsequent code execution

        return res.end();

    }

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

        // preparing to store data chunk in an array

        const body = [];

        // Record dateTime for Message-${now}.txt

        const now = new Date();

        const formattedNow = format(now, 'yyyy-MM-dd\_\_HH\_mm\_ss');

        req.on('data', (chunk) => {

            body.push(chunk);

        });

        req.on('end', () => {

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

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

            const messagePath = path.join(\_\_dirname, 'message', `message-${formattedNow}.txt`);

            fs.writeFile(messagePath, message, err => {

                if (err) {

                    console.error(`Error:\n${err}`);

                    res.statusCode = 501;

                    res.end();

                    return;

                }

                res.statusCode = 302; // HTTP status code for redirection

                res.setHeader('Location', '/');

                res.end();

            });

        });

    } else {

        res.statusCode = 404;

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

        res.write('<html>');

        res.write('<head><title>Not Found</title></head>');

        res.write('<body><h1>Page Not Found</h1></body>');

        res.write('</html>');

        res.end();

    }

});

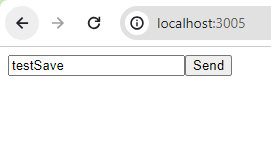
const port = 3005;

server.listen(port, () => {

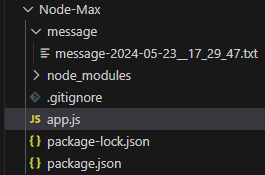
    console.log(`Server running at [http://localhost:${port}`](http://localhost:$%7bport%7d%60));

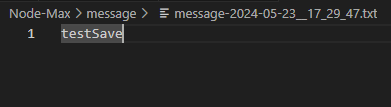
});

Test Node app



Send



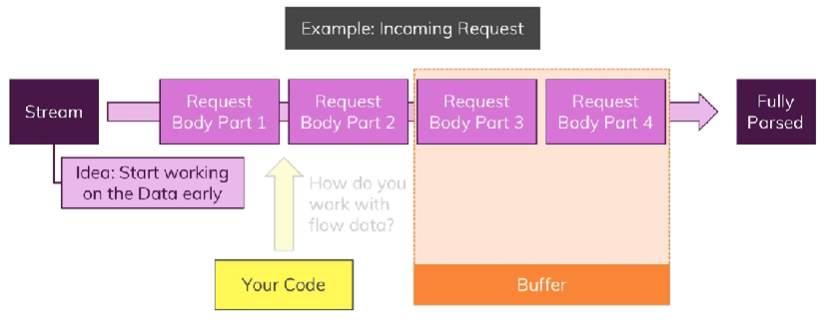


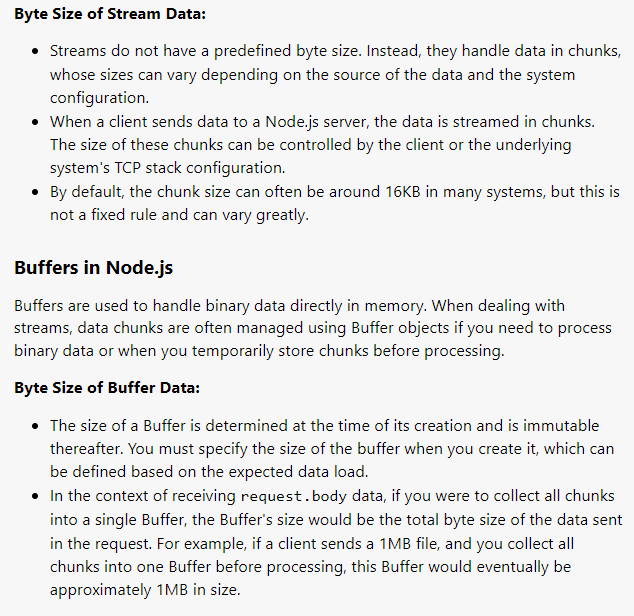
**Parsing request body**

sending request.body data in chunks

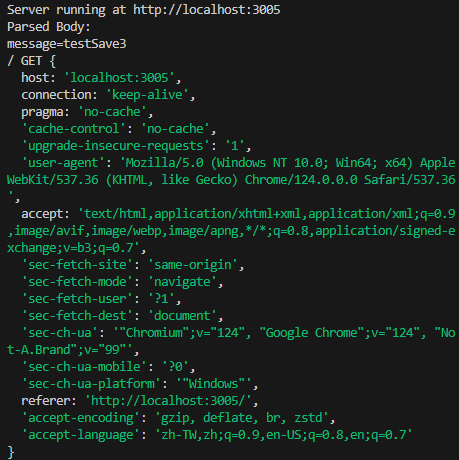
Buffer = Bus Stop = Hold multiple chunks before they're released

Buses = Always driving needing a Bus Stop to board Pessangers before keep moving on



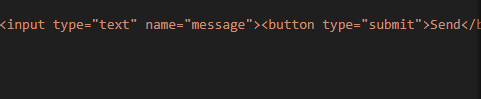


Console logging parsedBody data chunk



message=testSave3

cuz we named <input> as message



// import a global module named http

const fs = require('fs');

const http = require('http');

const path = require('path');

const { format } = require('date-fns');

// using ES6 arrow

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

    const url = req.url;

    const method = req.method;

    if (url === '/') {

        console.log(req.url, req.method, req.headers); // Log the request URL to the console

        res.statusCode = 200; // HTTP res status code

        // passing a header to res

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

        res.write('<html>');

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

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

        res.write('</html>');

        // returning to stop subsequent code execution

        return res.end(`OK`);

    }

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

        // preparing to store data chunk in an array

        const body = [];

        // Record dateTime for Message-${now}.txt

        const now = new Date();

        const formattedNow = format(now, 'yyyy-MM-dd\_\_HH\_mm\_ss');

        // Listen on Data events using anonymous ES6 arrow function

        req.on('data', (chunk) => {

            // appending each data chunk

            body.push(chunk);

        });

        req.on('end', () => {

            // Data is received in chunks (Instances of Buffer)

            // Chunks are collected into an array

            // Once all chunks are received (end Event Loop)

            // they are concatenated using Buffer.concat()

            // Need to know incoming data to be String

            // Will need to change data type if parsing a File

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

            console.log(`Parsed Body:\n${parsedBody}`);

            // message = spliting message=input

            // [1] taking 2nd element => input

            // [0] taking 1st element => message

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

            const messagePath = path.join(\_\_dirname, 'message', `message-${formattedNow}.txt`);

            // async fs writeFile method

            fs.writeFile(messagePath, message, err => {

                if (err) {

                    console.error(`Error:\n${err}`);

                    res.statusCode = 501;

                    return res.end(`Server Internal Error`);

                }

                res.statusCode = 302; // HTTP status code for redirection

                // redirect to /

                res.setHeader('Location', '/');

                // return to exit a function

                return res.end(`Redirected to /`);

            });

        });

    } else {

        res.statusCode = 404;

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

        res.write('<html>');

        res.write('<head><title>Not Found</title></head>');

        res.write('<body><h1>Page Not Found</h1></body>');

        res.write('</html>');

        res.end(`Page Not Found`);

    }

});

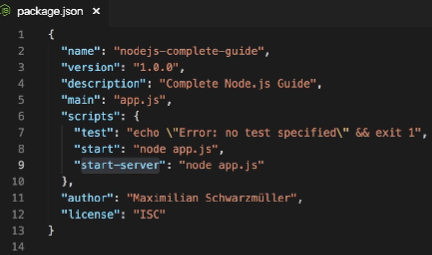
const port = 3005;

server.listen(port, () => {

    console.log(`Server running at [http://localhost:${port}`](http://localhost:$%7bport%7d%60));

});

Understanding NPM Scripts



This is a custom npm script

"scripts": {

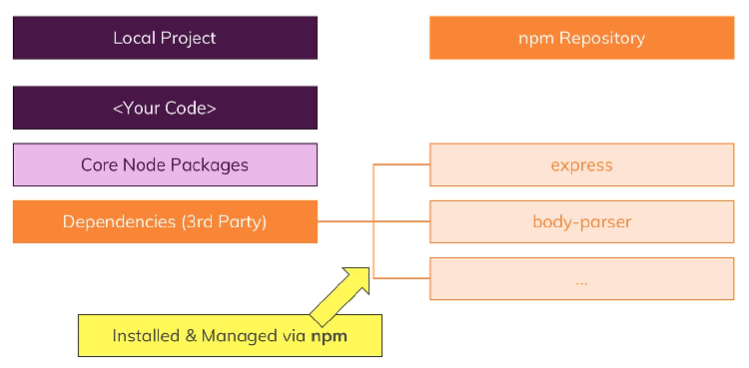
"start-server": "node app.js"

}

**We need to run custom npm script this way**

npm run start-server;

Installing 3rd Party Packages



**Installing nodemon as a dev dependency**

npm install nodemon --save-dev;

**Installing packages as a Prod dependency**

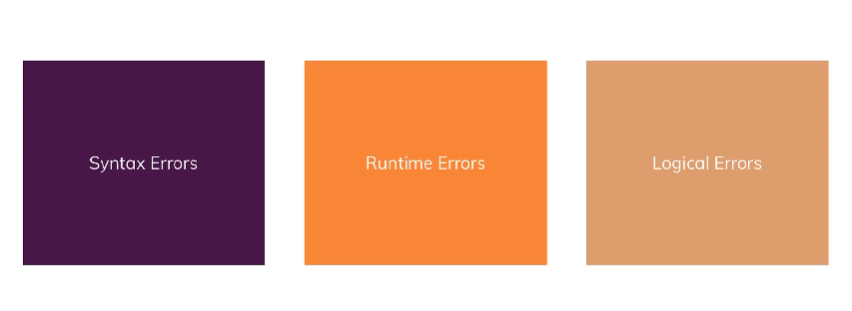
npm install packageName --save;

**Installing packages globally**

npm install packageName --global;

Understanding different Error Types

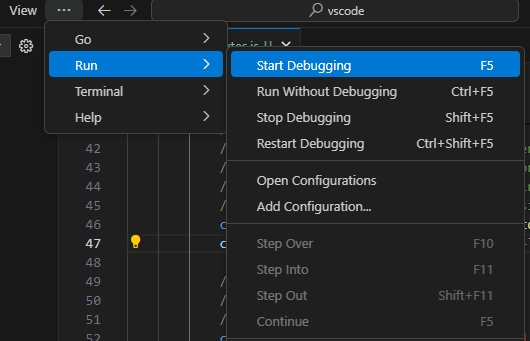
**Types of Errors**



Logical Errors

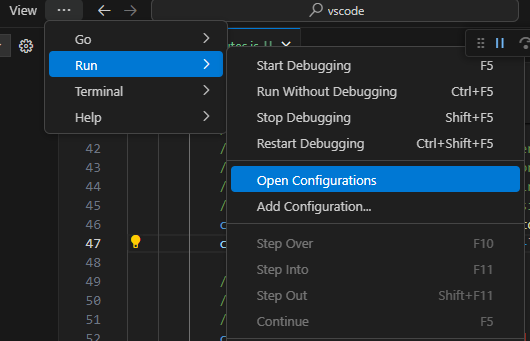
We've got no error messages, the app works, yet we do NOT know the root cause of unintentional behaviours.

**Use debugger**

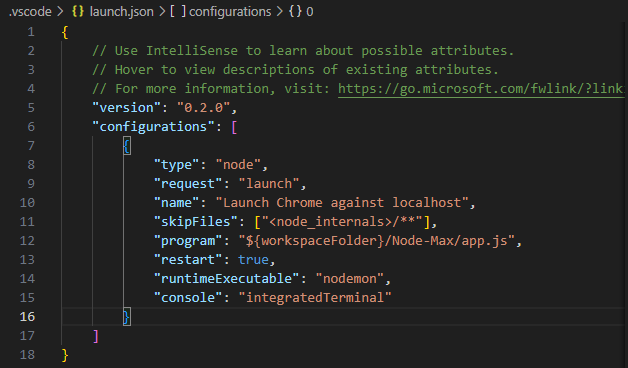


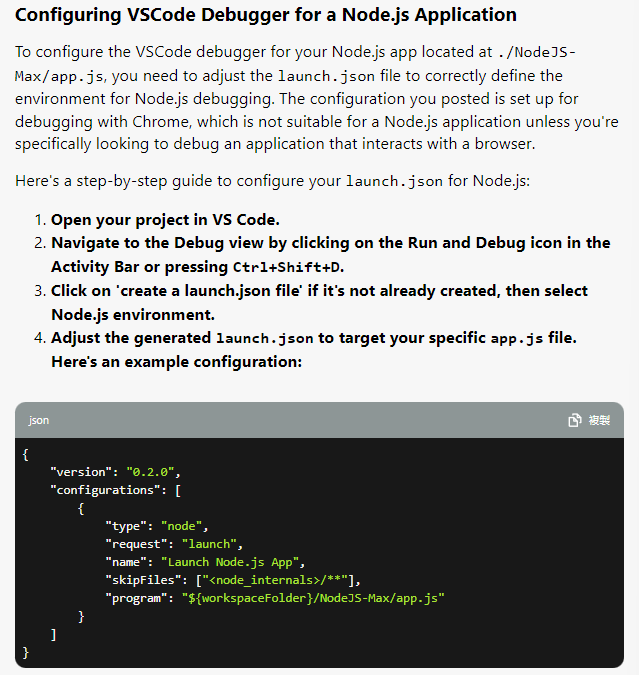
**To configure the web app we are debugging**

Open Configurations



**Configure debugging url to fit our in-progress app**





**Deep-diving vscode Debugger capabilities**

<https://code.visualstudio.com/docs/nodejs/nodejs-debugging>

