

NodeJS es monohilo

# VARIABLES DE ENTORNO

Variables de entorno:

Process.env.NOMBRE || ‘Sin Nombre’

Ejecutando archivos de node:

node rutaArchivo/archivo.js

para darle variables

NOMBRE=Andres node rutaArchivo/archivo.js

# HERRAMIENTAS

NODEMON: herramienta para usar en local y no tener que volver a ejecutar un archivo.

Instalar: npm install -g nodemon

Ejecutar: nodemon ruta/archivo.js

PM2: parecido a nodemon pero mas complejo, usado más en producción

Instalar: npm install -g pm2

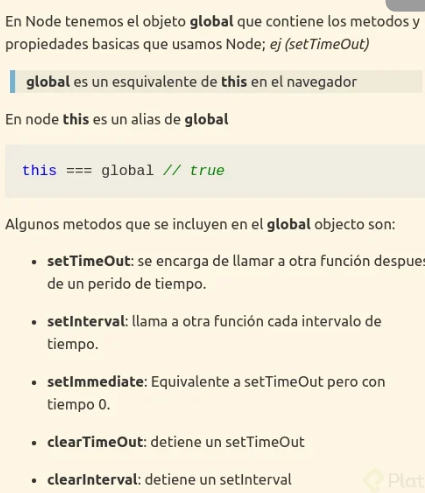
Ejecutar: pm2 start ruta/archivo.js

Detener: pm2 stop idProceso

# TIPOS DE FUNCIONES ASINCRONAS

* Callbacks
* Promise
* Async Await

# GLOBALES GLOBALS



Crear una variable global:

global.miVariable = ‘hola’;

clg(miVariable)

# FYLESYSTEM

const fs = require('fs');

function leer(ruta,cb){

    fs.readFile(ruta,(err,data)=>{

        cb(data.toString())

    })

}

function escribir(ruta,contenido,cb){

    fs.writeFile(ruta,contenido,function(err){

        if(err){

            console.error('No he podido escribirlo',err)

        }else{

            console.log('Se ha escrito correctamente')

        }

    })

}

function borrar(ruta,cb){

    fs.unlink(ruta,cb)

}

# Console

Con console podemos imprimir todo tipo de valores por nuestra terminal.

* console.log: recibe cualquier tipo y lo muestra en el consola.
* console.info: es equivalente a log pero es usado para informar.
* console.error: es equivalente a log pero es usado para errores.
* console.warn: es equivalente a log pero es usado para warning.
* console.table: muestra una tabla a partir de un objeto.
* console.count: inicia un contador autoincremental.
* console.countReset: reinicia el contador a 0.
* console.time: inicia un cronometro en ms.
* console.timeEnd: Finaliza el cronometro.
* console.group: permite agrupar errores mediante identación.
* console.groupEnd: finaliza la agrupación.
* console.clear: Limpia la consola.

# PROCESOS HIJO

El módulo de procesos secundarios de Node.js (**child\_process**) tiene dos funciones **spawn** y **exec**, mediante las cuales podemos iniciar un proceso secundario para ejecutar otros programas en el sistema.

La diferencia más significativa entre child\_process.spawn y child\_process.exec está en lo que spawn devuelve un stream y exec devuelve un buffer.

* Usa **spawn** cuando quieras que el proceso hijo devuelva datos binarios enormes a Node.
* Usa **exec** cuando quieras que el proceso hijo devuelva mensajes de estado simples.
* Usa **spawn** cuando quieras recibir datos desde que el proceso arranca.
* Usa **exec** cuando solo quieras recibir datos al final de la ejecución.

En node podemos crear procesos hijos que ejecuten cualquier accion de nuestro sistema operativo, como si nos encontraramos en la linea de comandos.

Podemos llamar a exec para ejecuciones sencillas:

**const** { exec } = require('child\_process')

exec('ls', (e, stdout) => {

(e) ?

console.log(e) :

console.log(stdout)

})

Podemos llamar a spawn para obtener el proceso: *La ventaja de este enfoque es que obtienes mayor control del proceso, y del estado en el que se encuenta*

**const** { spawn } = require('child\_process')

**const** myprocess = spawn('ls')

process.stdout.on("data", data => console.log(data.toString()));

process.on("exit", () => console.log("process end"));

# **Módulos nativos en c++**

JavaScript permite hacer uso de módulos nativos de c++. Para lograr esto debemos instalar sudo npm i -g node-gyp, este modulo de npm nos permite compilar módulos nativos de c++ en node.

Luego debemos tener listo nuestro archivo de código fuente en c++ junto a otro archivo .gyp, que nos ayudara hacer la compilación a JavaScript.

En este archivo .gyp le indicamos que va compilar, como se va llamar el archivo resultante y de donde va a tomar la info a convertir, todo esto lo dejamos como un json

{

"targets": [

{

"target\_name": "addon",

"sources": [ "hola.cc" ]

}

]

}

luego le decimos a node que configure este modulo, con le comando node-gyp configure, como resultado tendremos en un directorio nuevo donde se encontraran diferentes archivos de código nativo, para finalizar con node-gyp build creamos nuestro modulo y estará listo para ser usado.

# HTTP

const http = require('http')

function router(req,res){

    console.log('nueva petición')

    console.log(req.url)

    switch(req.url){

        case '/hola':

            res.write('Hola, ya se escuchar HTTP de NodeJS')

            res.end()

            break;

        default:

            res.write('Error 404: No sé lo que quieres')

            res.end()

    }

}

console.log('Escuchando http en el puerto 3000')

# PROCESS

*// Process viene ya en las variables globales, no tenemos que usar require*

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

    console.log('EL proceso va a terminar')

})

*// al terminar el proceso de nodejs*

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

    console.log('EL proceso acabó')

})

*// para capturar cualquier error no capturdo*

process.on('uncaughtException', (err,origen)=>{

    console.error('Vaya se nos ha olvidado capturar un error')

    console.error(err)

    console.log(origen)

})

 funcionNoExiste()

 console.log('Esto si el error no se recoje, no sale')

*// cuando se ha rechazado*

process.on('unhandleRejection')

# Gestión de paquetes: NPM y package.json

# Construyendo módulos: Require e Import

# Módulos útiles

<https://blog.logrocket.com/4-alternatives-to-moment-js-for-internationalizing-dates/>

Inicializar un package.json

npm init -y

Paquete de encriptado:

npm i bcrypt

const bcrypt = require('bcrypt')

const password  = '1234segura!'

bcrypt.hash(password,5,function(err,hash){

    console.log(hash)

*// comparar contraseña con el hash*

    bcrypt.compare(password,hash,function(err,res){

        console.log(err)

        console.log(res)

    })

})

## MOMENT

Instalar: npm i moment

const moment = require('moment')

let ahora = moment()

*// formatear fecha*

console.log(ahora.format('YYYY/MM/DD - HH:mm'))

## SHARP

Instalar: npm i Sharp

const sharp = require('sharp')

sharp('original.png')

    .resize(80)

    .grayscale()

    .toFile('resized.png')

# BUFFERS

## Buffer

Un buffer es un espacio de memoria (en la memoria ram), en el que se almacenan datos de manera temporal.

Es la forma mas cruda en la que se pueden almacenar los datos. (Se guardan en **bytes** y no se especifica el tipo de dato)

En la consola, **los datos se muestran en formato hexadecimal**.

# STREAMS

\*\* Stream\*\*

Podría decirse que un Stream es el proceso de ir consumiendo datos al tiempo en que se reciben. Por ejemplo, cuando vemos un video en Youtube estamos consumiendo datos por medio de streaming (readable stream, porque solo podemos ver los videos mas no editarlos) ya que lo vemos al mismo tiempo en que este se está descargando. de lo contrario habría que esperar a que se descargue el video por completo para poder verlo.

buffer

Si en el caso anterior, mientras vemos el video, fallara el internet, así sea por un segundo, la reproducción se pararía instantáneamente. Pero sabemos que en realidad no es así, el video continúa reproduciéndose por un tiempo mas. Esto es gracias a la implementación de un buffer el cuál es un espacio en memoria ram en donde la información proveniente del servidor llega por fragmentos (chunks), para luego ser consumido, y como ese almacenamiento de datos en el buffer se hace a bajo nivel, de forma binaria, el proceso es mucho mas rápido de lo que se consume. Es por eso que cuando reproducimos un video en Youtube vemos que este se carga mas rápido. (dependiendo del ancho de banda claro está)

# Benchmarking (console time y timeEnd)

console.time('todo')

let suma = 0;

*// TIEMPO QUE TARDA EN EJECUTARSE UN PEDASO DE CÓDIGO*

console.time('bucle')

for (let i = 0; i < 100000000; i++) {

    suma += 1;

}

console.timeEnd('bucle')

let suma2 = 0;

*// TIEMPO QUE TARDA EN EJECUTARSE UN PEDASO DE CÓDIGO*

console.time('bucle 2')

for (let i = 0; i < 1000000000; i++) {

    suma2 += 1;

}

console.timeEnd('bucle 2')

console.time('asincrono')

asincrona().then(()=>{

    console.timeEnd('asincrono')

})

console.timeEnd('todo')

function asincrona(){

    return new Promise(resolve=>{

        setTimeout(function(){

            console.log('Termina el proceso asincrono')

            resolve()

        })

    })

}

# DEBUGGER

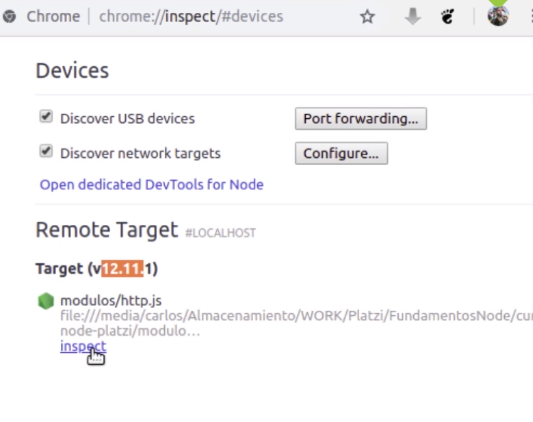
Al ejecutar el archive agregamos el –inspect

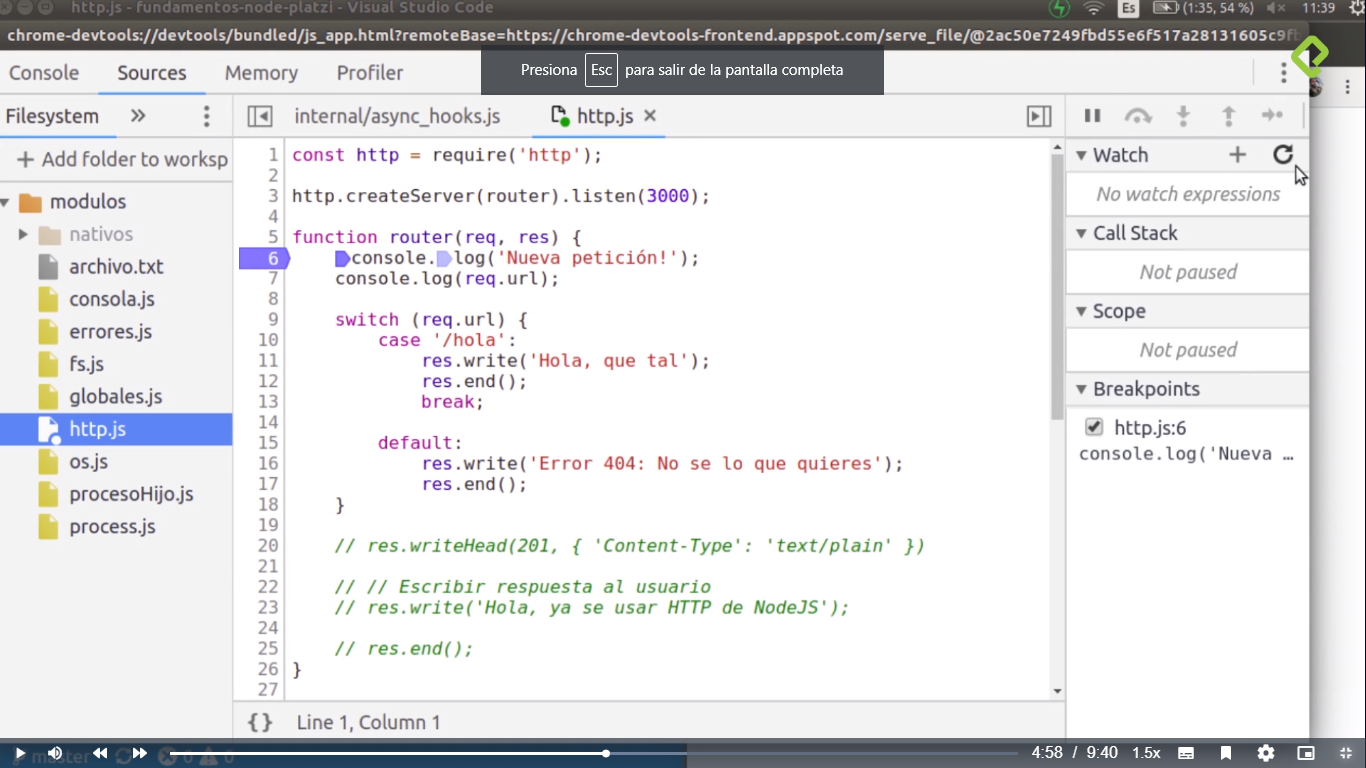
Ejemplo:

node – inspect modulos/http.js

luego ir a la web del proyecto

en otra pestaña colocar “chrome://inspect”



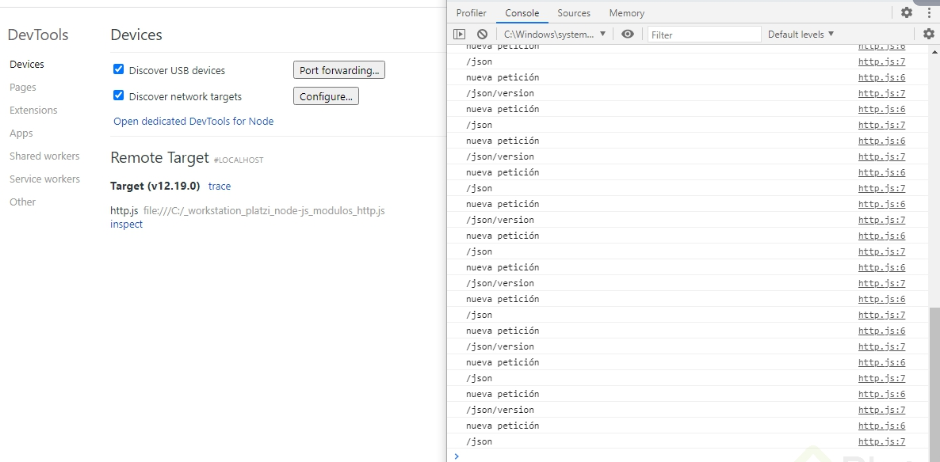


Node.js viene integrado con un modo de debug para poder conectarnos desde cualquier herramienta de inspección de código a nuestro código de node.js.

Podemos utilizar en la terminal el flag de --inspect con nodemon



Para poder acceder a debugger de chrome vamos a la url chrome://inspect/#devices y le dan a inspect en el remote target que quieres inspeccionar.



# Error First Callbacks

function asincrona(callback){

    setTimeout(function(){

        try {

            let a = 3+z

            callback(null,a)

        } catch (error) {

            callback(error)

        }

    },1000)

}

asincrona(function(err,dato){

    if(err){

        console.error('Tenemos un error');

        console.error(err)

*// throw err : ESTE SOLO FUNCIONA PARA funciones síncronas*

        return false

    }

    console.log('Todo ha ido bien, mi data es',dato)

})

# SCRAPING

Web scraping es una técnica utilizada mediante programas de software para extraer información de sitios web. Usualmente, estos programas simulan la navegación de un humano en la World Wide Web ya sea utilizando el protocolo HTTP manualmente, o incrustando un navegador en una aplicación.

const puppeteer = require('puppeteer');

(

    async () =>{

        console.log('Lanzamos navegador!')

*// const browser = await puppeteer.launch();*

*// para poder ver el navegador*

        const browser = await puppeteer.launch({headless:false});

        const page = await browser.newPage();

        await page.goto('https://e-consultaruc.sunat.gob.pe/cl-ti-itmrconsruc/jcrS00Alias');

        await page.evaluate(()=>{

            const inputRuc = document.getElementById("txtRuc");

            inputRuc.value = '20512613218'

            const btnBuscar = document.getElementById("btnAceptar");

            btnBuscar.click();

        })

        console.log('Cerramos navegador...');

*// browser.close();*

        console.log('Navegador cerrado');

    }

)();

# Automatización de procesos

npm i

npm i gulp gulp-server-livereload

archivo “GULPFILE.JS”

const gulp = require('gulp')

const server = require('gulp-server-livereload')

gulp.task('build',function(callback){

    console.log("Construyendo el sitio");

    setTimeout(callback,1200);

})

gulp.task('serve',function(cb){

    gulp.src('www')

        .pipe(server({

            livereload:true,

            open:true

        }));

})

*// tareas en serie*

gulp.task('default',gulp.series('build','serve'))

EN PACKAGE.JSON

"scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "start": "gulp",

    "build": "gulp build",

    "serve": "gulp serve"

  },

# Aplicaciones de escritorio

Electron

const {app, BrowserWindow} = require('electron')

let mainWindow;

app.on('ready',createWindow)

function createWindow(){

    mainWindow = new BrowserWindow({

        width:800,

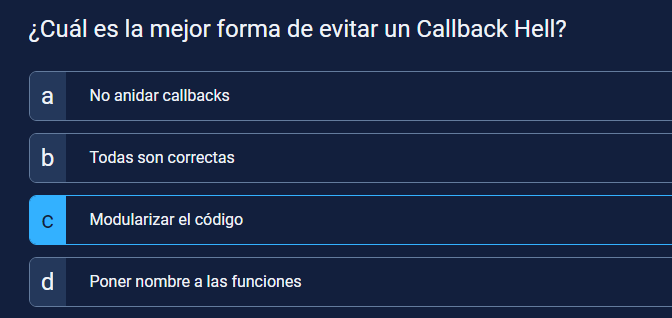
        height:600

    })

    mainWindow.loadFile('index.html')

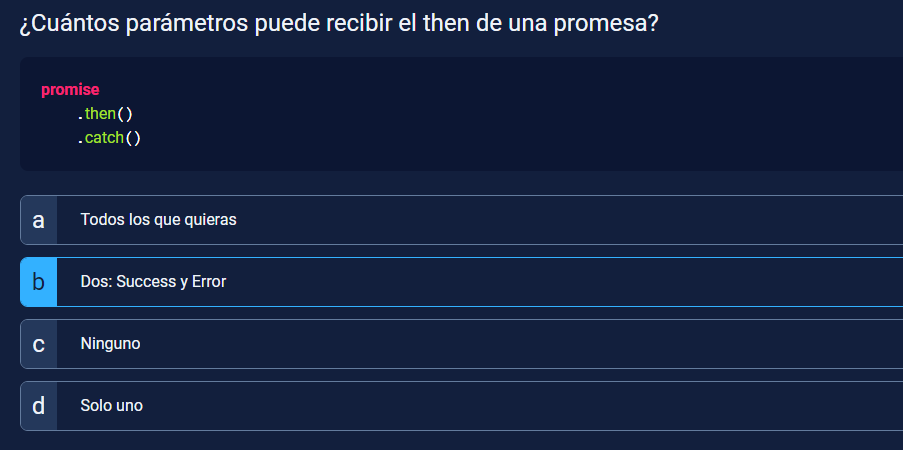
}

¿Cuál es la mejor forma de evitar un Callback Hell?



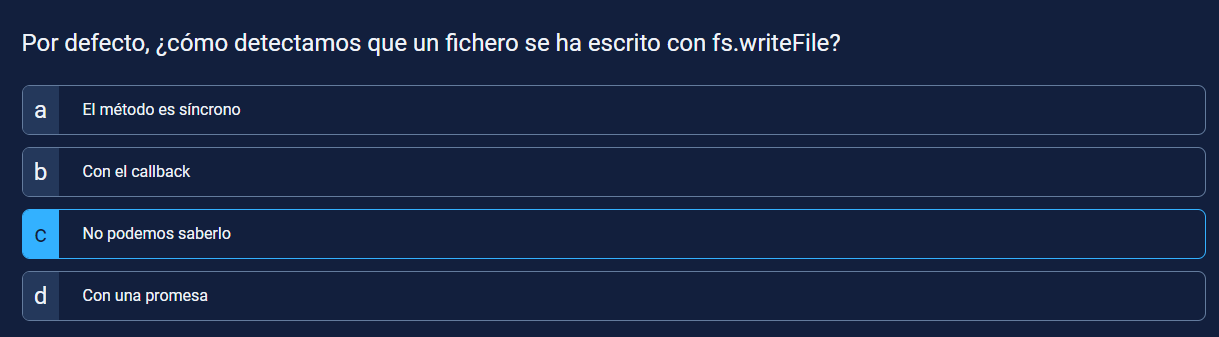
Rpt: Poner nombre a las funciones. LUEGO SOLO QUEDA “TODAS SON CORRECTAS”

¿Cuántos parámetros puede recibir el then de una promesa?



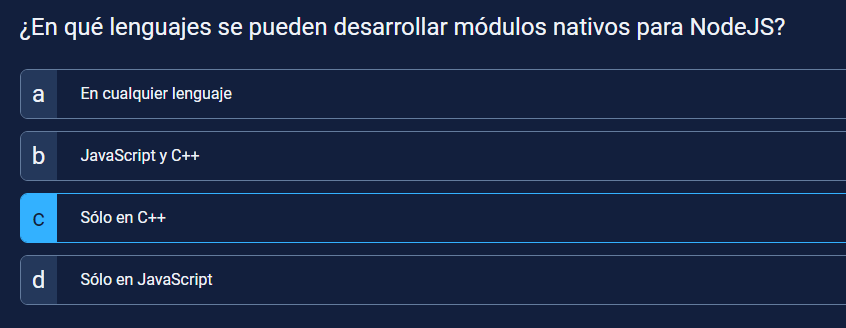
RPTA: SOLO UNO, LUEGO SOLO QUEDA “NINGUNO”

Por defecto, ¿cómo detectamos que un fichero se ha escrito con fs.writeFile?



RPTA: CON EL CALLBACK, LUEGO SOLO QUEDA “EL METODO ES SINCRONO”

¿En qué lenguajes se pueden desarrollar módulos nativos para NodeJS?



RPTA: JAVASCRIPT Y C++ , LUEGO SOLO QUEDA “SOLO EN JAVASCRIPT”