NodeJS es un entorno de ejecución para Javascript construido sobre el motor de Javasript V8 de Chrome, es decir, es un entorno en el que podemos ejecutar nuestras aplicaciones de JS sin necesidad de un browser, y nos permite utilizar toda la potencia de Javascript con todas sus nuevas features. NodeJS tiene distintas características importantes, como

Nos permite hacer desarrollos basados en Javascript,

Basado en operaciones de E/S sin bloqueos (interesante a la hora de crear un chat),

Orientado a eventos. La asíncronía es un efecto colateral de esos eventos, por lo que con NodeJS, podemos hacer nuestros desarrollos asíncronos,

Por estas caracerísticas, podemos decir que NodeJS es liviano y eficiente, permitiendo utilizar NodeJS en distintos dispositivos sin gran capacidad de procesamiento.

Por todas estas características, podemos decir que NodeJS es buena opción para desarrollar Servidores Web, aplicaciones basadas en sockets y para desarrollar aplicaciones para IOT.

NPM es una de las partes más importantes de NodeJS.

NPM o Node Package Manager, es el gestor de paquetes que nos ofrece NodeJS, o es el gestor de paquetes para Javascript, permitiendonos descubrir nuevos paquetes de JS. Nos permite instalar paquetes de JS, compartirlos, distribuirlos, manejar dependencias y recibir feedback de otros usuarios.

Una de las peculiaridades cuando usamos NPM y creamos un proyectos, es el fichero package.json, que tiene una serie de características, entre ellas, almacena las dependencias de nuestro proyecto con la versión que estamos usando. De esta forma, no tenemos que compartir o almacenar todos los ficheros de todas las librerias que utilizamos, ya que al tener una referencia a nuestras dependencias en el fichero package.json, podemos instalar estas dependencias en cualquier momento. Además de este fichero, a partir de la versión 5 de NPM, tambien se genera el fichero package-lock.json que genera una snapshot del arbol de dependencias de los paquetes que usemos en nuestro proyecto, de tal forma que nos aseguramos que nuestro proyecto va a utilizar una versión en concreto de las distintas dependencias de nuestro proyecto.

Existe otro gestor de paquetes, alternativo a NPM, llamado Yarn. Surgio como alternativa a NPM e intentaba solucionar el problema que habia con las versiones anteriores de NPM 5. Yarn genera un fichero yarn.lock que es equivalente al fichero package-lock.json

NodeJS, al igual que el resto de lenguajes de programación, no sirve para todo, pero si esta muy bien preparado para ciertos desarrollos.

El primer caso que se nos ocurre es un chat, ya que es la forma más típica en tiempo real, suele hacer una aplicación ligera, se espera un alto tráfico de datos, y funciona en dispositivos distribuidos.

Otra opción es el desarrollo de una API Rest. En una API Rest, se utilizan los objetos JSON, al igual que ocurre en JS. Tambien podemos onectar con MongoDB, una base de datos NoSQL que nos permite incrustar código en Javascript. A la hora de realizar una API Rest, tenemos paquetes en NPM que nos pemriten hacer el desarrollo muy facil, como Express, Koa, Hapi.js, …

Además, debido a que NodeJS esta orientado a eventos, y con E/S sin bloqueos nos permite procesar peticiones HTTP, ya que las peticiones HTTP son eventos aislados, por lo que podriamos procesar ficheros, etc.

Un segundo caso para lo que NodeJS es muy buena opción, es para entradas en espera, como puede ser el caso de trabajar con bases de datos. Trabajar con bases de datos es bloqueante, por lo que la entrada estaría en espera a que acabe el Sistema Gestor de Base de Datos. Al no estar basado en bloqueos, podemos utilizar un gestor de colas como RabbitMQ, y por todo ello, NodeJS es buena opción para gestionar la concurrencia.

Por otro lado, NodeJS es muy buena opción para desarrollar un servidor Proxy, ya que se espera que pueda manejar una gran cantidad de conexiones simultaneas sin bloqueos.

Podemos utilizar NodeJS para IOT, ya que NodeJS consume pocos recursos, utilizariamos un solo lenguaje, Javascript además de permitirnos conexiones simultaneas y con E7S sin bloqueos.

Pero lo mejor de todo, es que para todo ello, solo necesitamos saber Javascript y algunas cosas de NodeJS.

Instalar NodeJS es un proceso muy sencillo, ya que podemos instalarlo en cualquier Sistema Operativo, ya sea Windows, Linux, …

El problema que nos podemos encontramos al instalar NodeJS en nuetsro sistema, es que solo podemos tener una versión de NodeJS, pero ¿que ocurre cuando necesitamos tener instaladas distintas versiones de NodeJS? NVM, o Node Versión Manager es una aplicación que nos permite gestionar distintas versiones de NodeJS y NPM en un mismo ordenador. En NVM existe un fichero llamado .nvmrc. Este fichero se define por cada proyecto, y nos permite definir la versión de NodeJS que queremos utilizar en nuestro proyecto, lo que es bastante util a la hora de desarrollar nuestros proyectos, ya que con este fichero, NVM podría detectar que versiónd e NodeJS debe usar en ese proyecto, y si no la encuentra en el sistema, instala dicha versión.

Existen algunos comandos más usados con NVM

> nvm ls-remote

Nos muestra todas las versiones de NodeJS que estan publicadas, incluso cuales son las versiones LTS.

> nvm ls

Nos muestra las versiones de NodeJS que tenemos instaladas en el sistema.

> nvm install 9.3.0

Instala la versión 9.3.0 de NodeJS

> nvm use 8.1.4

Carga la versión de NodeJS 8.1.4 en el directorio, y nos permite utilizar esta versión de Node.

La comunidad de NodeJS es una de las comunidades más grandes, y en parte, es gracias a NPM, su gestor de paquetes. Esto es debido también al lenguaje que se utiliza en NodeJS, Javascript y de todas las librerías que hay publicadas en este lenguaje en NPM.

Esta comunidad sigue creciendo cada vez más, y publicando nuevos paquetes, lo que permite que nuestros desarrollos sean agiles y más fáciles.

Todo ello gracias a Javascript, ya que cualquier persona que conozca este lenguaje, podría comenzar a desarrollar su aplicación en NodeJS, comparta conocimiento o cree y comparta sus librerias en NPM. Algunos de os paquetes más utilizados son:

Express: Nos permite crear nuestro servidor web, ya sea API Rest, o web renderizando nuestro servidor desde back.

React: Una de las librerías más utilizadas a la hora de crear Single Page Application

Vue: Una nueva experienca a la hora de crear Single Page Appications. Muy recomendable

Babel: Nos permite transpilar o transformar nuestro código de la última versión de ECMAScript a una versión de JS que entienda el navegador o servidor

Lodash: Un conjunto de funciones y helpers que nos ayudan a desarrollar más rápido, y mejor

Webpack: Gestor de tareas automáticas, y generador de bundles para nuestra aplicación

Ecosistema

Que necesitamos para trabajar con NodeJS? Pues es facil, solo necesitamos Javascript y un terminal ! Hablando del ecosistema, podemos decir que hay más de 2.760.000.000 paquetes que se descargan por semana en NPM, un número incrieble pero que va aumentando cada semana debido a lo activa que es la comunidad de NodeJS y Javascript.

**EJERCICIO**

Lo primero que haremos será ejecutar el comando npm init, el cual nos generará el fichero package.json despues de responder una serie de preguntas relacionadas con nuestro proyecto, quedando nuestro fichero package.json de la siguiente forma:

{

"name": "curso-nodejs",

"version": "1.0.0",

"description": "Curso NodeJS",

"main": "src/index.js",

"keywords": [

"nodejs"

],

"license": "MIT",

"author": "OpenWebinars",

"scripts": {

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

}

}

Luego, crearemos nuestro fichero index.js, que será el que ejecutaremos con NodeJS. En este caso, haremos que muestre Hola Mundo, quedando nuestro fichero de la siguiente manera

// src/index.js

console.log('Hola Mundo')

Para ejecutar nuestro fichero escribiremos

> node src/index.js

Otra opción, es crearnos un script en nuestro package.json

{

"name": "curso-nodejs",

"version": "1.0.0",

"description": "Curso NodeJS",

"main": "src/index.js",

"keywords": [

"nodejs"

],

"license": "MIT",

"author": "OpenWebinars",

"scripts": {

"start": "node src/index.js"

}

}

Para ejecutar la tarea start, haremos lo siguiente

> npm run start

En este apartado vamos a aprender algunas de las características que nos proporciona el nuevo estándar ES6.

Constantes y Left

Con ES6:

const constante = 10

let variable = 15

Equivalente a código javascript legacy:

var constante = 10;

var variable = 15;

Template String

Con ES6:

const lugar = 'Cádiz'

const NOMBRE = 'José Manuel'

let frase = `Soy ${nombre} y nací en ${lugar}`

Equivalente a código javascript legacy:

var lugar = 'Cádiz';

var NOMBRE = 'José Manuel';

var frase = 'Soy ' + nombre + ' y nac\xED en ' + lugar;

Mejoras en los objetos

Con ES6:

const lugar = 'Cádiz'

const datos = {

nombre: 'José Manuel',

edad: 35,

lugar,

random() {

return 'randomNumber'

}

}

Equivalente a código javascript legacy:

var lugar = 'Cádiz';

var datos = {

nombre: 'José Manuel',

edad: 35,

lugar: lugar,

random: function random() {

return 'randomNumber';

}

};

Arrow functions

Con ES6:

const fn = (item) => {

console.log(item)

return item

}

const otraFn = item => console.log(item)

Equivalente a código javascript legacy:

var fn = function fn(item) {

console.log(item);

return item;

};

var otraFn = function anotherFn(item) {

return console.log(item);

};

Default parameters

Con ES6:

function miFn(x, y, w = 10, h = 10) {

return x + y + w + h;

}

miFn(1, 2) === 23

Equivalente a código javascript legacy:

function miFn(x, y) {

var w = arguments.length > 2 && arguments[2] !== undefined ? arguments[2] : 10;

var h = arguments.length > 3 && arguments[3] !== undefined ? arguments[3] : 10;

return x + y + w + h;

}

miFn(1, 2) === 23;

Destructuring Array

Con ES6:

var [a, , b] = [1, 2, 3];

a === 1;

b === 3;

Equivalente a código javascript legacy:

var \_ref = [1, 2, 3],

a = \_ref[0],

b = \_ref[2];

a === 1;

b === 3;

Destructuring en Funciones

Con ES6:

function miFn({ nombre: x }) {

console.log(x);

}

miFn({ nombre: 5, random: 15 })

Equivalente a código javascript legacy:

function miFn(\_ref2) {

var x = \_ref2.nombre;

console.log(x);

}

miFn({ nombre: 5, random: 15 });

Default parameters y Destructuring

Con ES6:

function miFn({ x, y, w = 10, h = 10 }) {

return x + y + w + h;

}

miFn({ x: 1, y: 2 }) === 23

Equivalente a código javascript legacy:

function miFn(\_ref) {

var x = \_ref.x,

y = \_ref.y,

\_ref$w = \_ref.w,

w = \_ref$w === undefined ? 10 : \_ref$w,

\_ref$h = \_ref.h,

h = \_ref$h === undefined ? 10 : \_ref$h;

return x + y + w + h;

}

miFn({ x: 1, y: 2 }) === 23;

Y más novedades que, para etse curso, no nos haran falta o iremos explicando a lo largo del curso.

The compiler for writing next generation JavaScript

Babel, como lo definen en su propia web, es un compilador de javascript para escribir la próxima generación de Javascript.

Para utilizar babel lo primero que necesitamos es NodeJS, en una versión superior a la 6.x (preferiblemente), luego, instalaremos babel.

**> npm install --save babel-core babel-cli**

**sudo npm install --save-dev babel babel-preset-env babel-register**

En este caso, necesitaremos tanto el core de babel, como su cli para utilizarlo en nuestro terminal.

Pongamos el ejemplo siguiente, tenemos un código escriot en ES& en un fichero llamado es6-script.js, para transformalo a código legacy, ejecutamos lo siguiente:

> babel es6-script.js --out-file script.js

Babel se basa en presets, o configuraciónes dependiendo de la versión del estandar que queramos utilizar. Los más conocidos son

env

es2015

es2016

es2017

latest (deprecado, ahora se utiliza env)

react

flow

además, tenemos los presets experimentales en los que podremos probar las nuevas funcionalidades que aún no estan aprobadas del todo por el comite TC39, y por tanto, no han sido añadidas a EcmasScript.

Cuando queremos utilizar un preset, debemos crear un fichero de configuración, donde especificaremos que preset queremos utilizar. Nuetsro fichero de configuración puede venir definido en alguno de estos ficheros:

.babelrc

package.json

env option

**Manejo de eventos**

En este caso, vamos a trabajar con una de las ibrerias internas de Node Readline que publicará un evento dependiendo de la acción que se este ejecutando. Para ello, nos basaremos en un fichero de ejemplo sample.txt

Tambien utilizaremos la librería fs que nos permite leer un fichero

// Muestra el número total de lineas, y el número de palabras por linea

// Documentación: https://nodejs.org/api/readline.html

import fs from 'fs'

import readline from 'readline'

const file = process.argv[2]

let lines = 0

console.log(`Fichero selecionado: ${file}\n`)

// Emite el evento 'line' cuando lee una linea

const rl = readline.createInterface({

input: fs.createReadStream(file),

crlfDelay: Infinity

})

rl.on('line', (line) => {

++lines

console.log(`Número de caracteres por linea: ${line.length}`)

})

rl.on('close', () => {

console.log(`\nNúmero total de lineas: ${lines}`)

})

Para ejecutar nuestro código, ejecutaremos la siguiente tarea

> npm run start sample.txt

Nuestro código, lo que hace es leer el 2 parametro que le enviamos como argumento (sample.txt), y leera cada linea del fichero hasta el final, de tal forma que cuando se publique un evento de lectura de linea (line), y un evento cuando acabe de leer el fichero (close), y mostrará información sobre el fichero dependiendo del evento.

**Programación asíncrona**

COmo ya sabemos, NodeJS esta preparado para trabajar con eventoss y que las E/S sean sin bloqueos. Además de permitirnos trabajar con eventos, NodeJS tambien nos permite trabajar de forma síncrona, de tal forma que cada instrucción se ejecuta en orden, una detras de otra.

Vamos a leer nuestro ficherom sample.txt de forma síncrona

// Muestra el número total de lineas, y el número de palabras por linea

// Documentación: https://nodejs.org/api/fs.html

import fs from 'fs'

const file = process.argv[2]

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

if (err) {

return console.log(err)

}

const lines = contents.toString().split('\n')

for (let line of lines) {

console.log(`Número de caracteres por linea: ${line.length}`)

}

console.log(`\nNúmero total de lineas: ${lines.length}`)

})

console.log(`Fichero selecionado: ${file}\n`)

En este caso, vemos que nuestro código no nos muestra información del fichero de forma asíncrona, si no que nos muestra todos los datos una vez se haya acabado de leer todo el fichero.

Para ejecutar nuestro código, ejecutaremos la siguiente tarea

> npm run start sample.txt

**Modularización**

En este caso, vamos a crear una pequeña aplicación que modularizaremos, y nos permitirá leer un fichero de forma síncrona, o por eventos.

Por un lado, tendremos 2 ficheros que almacenaran esta funcionalidad

// events.js

import fs from 'fs'

import readline from 'readline'

export default file => {

console.log('Eventos')

let lines = 0

console.log(`Fichero selecionado: ${file}\n`)

// Emite el evento 'line' cuando lee una linea

const rl = readline.createInterface({

input: fs.createReadStream(file),

crlfDelay: Infinity

})

rl.on('line', (line) => {

++lines

console.log(`Número de caracteres por linea: ${line.length}`)

})

rl.on('close', () => {

console.log(`\nNúmero total de lineas: ${lines}`)

})

}

// async.js

import fs from 'fs'

export default file => {

console.log('Asíncrono')

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

if (err) {

return console.log(err)

}

const lines = contents.toString().split('\n')

for (let line of lines) {

console.log(`Número de caracteres por linea: ${line.length}`)

}

console.log(`\nNúmero total de lineas: ${lines.length}`)

})

console.log(`Fichero selecionado: ${file}\n`)

}

Ahora, vamos a crear un menu utilizando la librería interna de Node readLine, para que el usuario pueda seleccionar que métod quiere ejecutar. Los ficheros async.js y events.js los cargaremos en nuestro fichero utilizando import, del nuevo estandar de JS.

// index.js

// Muestra el número total de lineas, y el número de palabras por linea

import readline from 'readline'

import async from './async'

import events from './events'

const file = process.argv[2]

const rl = readline.createInterface({

input: process.stdin,

output: process.stdout

})

rl.question(

`Como quiere leer el fichero?

1. De forma asíncrona (default)

2. Con eventos

Seleccione una opcion: `,

value => {

console.log(`Selecciono ${value}\n\n`)

switch (value) {

case '2':

events(file)

break

default:

async(file)

}

rl.close()

})

En este caso, utilizamos readLine para leer de la entrada estandar, el teclado, que valor ha seleccionado el usuario, y dependiendo de que haya seleccionado, ejecutaremos el método por eventos, o asíncrono.

Para ejecutar nuestro código, ejecutaremos la siguiente tarea

> npm run start sample.txt

**Introducción a Sockets**

Los sockets son una forma de comunicarse un cliente o servidor con otro cliente o servidor. Crea un canalbidireccional con ambos clientes, de tal forma que desde nuestro servidor Node podemos enviar un mensaje a otro cliente, ya sea una web, otro servidor, etc. Podemos tener tantas conexiones abiertas, con la única limitación que nos proporciona nuestro ordenador.

Para ello vamos a crear un fichero cliente. En este caso, vamos a utilizar la librería interna que nos proporciona Node, llamada net

import net from 'net'

const socket = net.Socket()

socket.connect(8000, 'localhost')

socket.write('Hola?')

socket.on('data', data => console.log(d.toString()))

En este caso, nuestro cliente se conectará al puerto 8000 de localhost, y publicará el mensaje Hola?. Cuando el servidor le envie un mensaje, nos mostrará dicho mensaje por la consola.

Vamos a definir nuestro servidor

import net from 'net'

const server = net.createServer(socket => {

socket.on('data', data => {

console.log(data.toString())

socket.write('Mundo?')

})

})

server.on('error', err => {

throw err

})

server.on('connect', () => console.log('socket connected'))

server.listen(

{

host: 'localhost',

port: 8000,

exclusive: true

},

() => console.log('Servidor socket abierto en ', server.address())

)

En este caso, abriremos un socket en el puerto 8000 en localhost. Una vez este conectado el servidor, mostraremos un mensaje en la consola. Cuando nuestro servidor reciba un mensaje, nos mostrará dicho mensaje en la consola, y enviará otro mensaje con Mundo?

Para ejecutar nuestro cliente, necesitaremos otra tarea en nuestro package.json, quedando así

{

"name": "curso-nodejs",

"version": "1.0.0",

"description": "Curso NodeJS",

"main": "src/index.js",

"keywords": [

"nodejs"

],

"license": "MIT",

"repository": {

"type": "git",

"url": "git@github.com:jmanuelrosa/curso-nodejs-openwebinars.git"

},

"homepage": "",

"author": {

"name": "José Manuel Rosa Moncayo",

"email": "josemanuel.rosamoncayo@gmail.com"

},

"bugs": {

"url": "https://github.com/jmanuelrosa/curso-nodejs-openwebinars/issues",

"email": "josemanuel.rosamoncayo@gmail.com"

},

"engines": {

"node": ">= 9.3.0",

"npm": ">= 5.5.1",

"yarn": ">=1.3.2"

},

"scripts": {

"start": "node --require 'babel-register' src/index.js",

"client": "node --require 'babel-register' src/client.js",

"lint": "standard 'src/\*\*/\*.js'"

},

"dependencies": {},

"devDependencies": {

"babel": "^6.23.0",

"babel-preset-env": "^1.6.1",

"babel-register": "^6.26.0",

"standard": "^10.0.3"

}

}

Para ejecutar nuestro código, ejecutaremos las siguientes tareas en dos terminales

> npm run start

y

> npm run client

**Manejo de errores**

En los lenguajes que son compilados, tenemos el proceso de compilación en el que se nos va avisando de errores. En el caso de NodeJS, no existe el proceso de compilación, pero tenemos una serie de eventos que se publican cuando se produce un error, y que podemos capturar para realizar distintas acciones dependiendo del error que capturemos.

process.on('unhandledRejection', (err, p) => {

console.log('Custom Error: An unhandledRejection occurred')

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

})

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

console.log('Custom Error: An uncaughtException occurred')

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

})

setTimeout(() => console.log('This will still run.'), 500)

// El error es JSON.pasre

Promise(resolve => JSON.pasre({ color: 'azul' }))

// test()

// throw 'casa'

En este caso, vemos que estamos capturando 2 errores, unhandledRejection y uncaughtException. Para probar estos errores, podemos ejecutar una promesa cuyo código esta mal, lanzar una excepción con throw, o invicoar una función que no esta definda.

**NodeMon**

Nodemon es un paquete que se encuentra en NPM que nos permite trabajar y monitorizar nuestros cambios en nuestra aplicación de Node, es decir, podemos ejecutar nuestra aplicación, y conforme modifiquemos nuestro código, que se vaya reiniciando nuestro código y volviendo a ejecutarse con nuestros cambios, algo muy cómodo a la hora de desarrollar.

Lo primero es instalar nodemon

> npm install --save-dev nodemon

Una vez instalado, debemos configurarlo mediante el fichero nodemon.json

{

"restartable": "rs",

"verbose": true,

"ignore": [

"node\_modules"

],

"execMap": {

"js": "node --require 'babel-register'"

},

"watch": [

"src"

],

"ext": "js"

}

En este fichero, estamos definiendo algunos parametros como restartable, que hará que nodemon se reinicie cuando detecte un cambio en los dicheros que esta haciendo watch, y definimos en watch.

Luego, añadiremos una tarea watch a nuestro package.json, quedando de la siguiente forma

{

"name": "curso-nodejs",

"version": "1.0.0",

"description": "Curso NodeJS",

"main": "src/index.js",

"keywords": [

"nodejs"

],

"license": "MIT",

"repository": {

"type": "git",

"url": "git@github.com:jmanuelrosa/curso-nodejs-openwebinars.git"

},

"homepage": "",

"author": {

"name": "José Manuel Rosa Moncayo",

"email": "josemanuel.rosamoncayo@gmail.com"

},

"bugs": {

"url": "https://github.com/jmanuelrosa/curso-nodejs-openwebinars/issues",

"email": "josemanuel.rosamoncayo@gmail.com"

},

"engines": {

"node": ">= 9.3.0",

"npm": ">= 5.5.1",

"yarn": ">=1.3.2"

},

"scripts": {

"start": "node --require 'babel-register' src/index.js",

"watch": "nodemon",

"lint": "standard 'src/\*\*/\*.js'"

},

"dependencies": {},

"devDependencies": {

"babel": "^6.23.0",

"babel-preset-env": "^1.6.1",

"babel-register": "^6.26.0",

"nodemon": "^1.14.3",

"standard": "^10.0.3"

}

}

Para ejecutar nuestro código con nodemon, ejecutaremos la siguiente tarea

> npm run watch

Existen otras alternativas a Nodemon, aunque este paquete es uno de los más configurables y potentes.

**Módulo HTTP**

El módulo HTTP es un módulo interno de NodeJS en el que podemos hacer varias cosas con el.

Existen varios métodos, y uno de los más interesantes es http.ClientRequest y publica una serie de eventos dependiendo del caso que ocurra. Además tenemos http.ServerResponse y http.IncomingMessage. Como podemos observar, son métodos orientados al protocolo HTTP.

Existen otros temas interesantes, como http.STATUS\_CODES

Nos centraremos en http.createServer para crear nuestro servidor web. EN este caso, crea un servidor web, basado en HTTP sin necesidad de utilizar librerias externas como Express. Este método, nos devuelve una instancia de http.Server, que posee métodos como listen, que nos permite definir en que ruta y puerto queremos que nuestro servidor web se publique.

**Servidor básico**

Vamos a implementar un servidor básico basándonos en la librería interna de NodeJS http.

import http from 'http'

const server = http.createServer((request, response) => {

response.write('<h1>Curso NodeJS de OpenWebinars!</h1>')

response.end()

})

server.listen(8000, 'localhost', err => {

if (err) {

return console.log('Error: ', err)

}

console.log('Server opened listen on http://localhost:8000')

})

const http = require('http');

// Create an HTTP tunneling proxy

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

res.write('<h1> Curso de Node </h1>')

res.end

//res.writeHead(200, { 'Content-Type': 'text/plain' });

//res.end('okay');

});

proxy.listen(8000,'localhost',err => {

if (err) {

return console.log('Eror',err);

}

console.log('Server abierto en http://localhost:8000');

});

En este caso, estamos creando un servidor web, que escribe en una response un título, utilizando HTML, para luego, cerrar ese response avisando de que dicho response no va a devolver nada mas.

Luego, creamos nuestro servidor web en el puerto 8000 y en el host, localhost. Una vez se ejecute nuestro servidor web, podemos ver en consola un mensaje avisandonos de ello.

Para probar nuestro servidor web, basta con acceder desde un navegador a http://localhost:8000 y veremos el código HTML que devuelve la response que hemos definido.

**El objeto Response**

En la librería http, tenemos el objeto request y response. Una request es una petición a un servidor web, mientras que una response, es la respuesta de nuestro servidor web a una request.

En una response podemos definir varias cosas, como cabeceras, códigos de estado, enviar datos … al igual que en una request, podemos modificar sus cabeceras, escribir sobre una request, saber cuando se ha realizado la request, …

Vamos a ver un ejemplo muy sencillo de una request y una response

import http from 'http'

const server = http.createServer((request, response) => {

if (request.method === 'GET') {

response.write('<h1>Metodo valido</h1>')

return response.end()

}

response.write('<h1>Esta intentando acceder con un metodo no valido</h1>')

return response.end()

})

server.listen(8000, 'localhost', err => {

if (err) {

return console.log('Error: ', err)

}

console.log('Server opened listen on http://localhost:8000')

})

En este caso, estamos creando un servidor web en localhost:8000 el cual comprueba que si el método de la request es GET, nos devuelve un información, mientras que si el método no es GET, nos muestra otra información mediante el objeto response.

Con esto, podriamos empezar a mejorar nuestro servidor, como por ejemplo, definir rutas.

**Encabezados**

Los encabezados son información que se envía tanto en una request como en una response, y en el caso de una response, podemos definir que es lo que esa respuesta del servidor, va a devolver, como por ejemplo, definir la codificación de caracteres o el tipo de dato que devuelve (HTML, JSON, …)

En este caso, vamos a definir la cabecera de nuestra respuesta, definiendo el tipo y codificación del contenido de nuestra respuesta, además de definir el código de estado 200.

import http from 'http'

const server = http.createServer((request, response) => {

response.writeHead(200, { 'Content-Type': 'text/html; charset=UTF-8' })

// response.setHeader('Content-Type', 'application/json')

// response.statusCode = 404

if (request.method === 'GET') {

response.write('<h1>Metodo valido</h1>')

return response.end()

}

response.write('<h1>Esta intentando acceder con un metodo no valido</h1>')

return response.end()

})

server.listen(8000, 'localhost', err => {

if (err) {

return console.log('Error: ', err)

}

console.log('Server opened listen on http://localhost:8000')

})

Para ver las cabeceras de la response, podemos acceder a http://localhost:8000 a traves del navegador, y desde la pestaña Network de la consola de de desarrollo de Chrome, acceder a la petición y ver las cabeceras tanto de la request como de la response.

**MongoDB**

**sudo npm install mongoose**

**Postgres sequelize**

**sudo npm install pg sequelize --save**

**sudo npm install express --save**

**Para monitoriar el server**

**sudo npm install --save-dev morgan**

**sudo npm install --save body-parser**

**Trabajar con Templates**

**sudo npm install --save pug**