Introducción a NodeJS

Fuente: https://www.youtube.com/watch?v=BhvLIzVL8_o

1 – Hola Mundo

El primer paso en NodeJS es enviar mensajes por consola. La instrucción es la misma que en javascript.

```
console.log('Hola mundo')
```

La diferencia en este caso es que no estamos escribiendo un script para ejecutar en el navegador, sino que al ser NodeJS un entorno de ejecución, puede hacerse directamente en la terminal del sistema operativo con el comando `node index.js`

2 – Módulos

En general los proyectos grandes se separan en distintos archivos con la finalidad de poder controlar mejor el proyecto, realizar pruebas, mantener y actualizar el código. En nodejs estos archivos se llaman módulos.

Pongamos como ejemplo que queremos implementar funciones que resuelvan operaciones matemáticas en un archivo externo. Para esto creamos el archivo **math.js** con las funciones sumar, restar multiplicar y dividir (teniendo en cuenta que no se puede dividir por 0). En formáto clásico de js estás funciones quedarían definidas de la siguiente manera.

```
function sumar(x1, x2){
    return x1 + x2
}

function restar(x1, x2){
    return x1 - x2
}

function multiplicar(x1, x2){
    return x1 * x2
}

function dividir(x1, x2){
    if( x2 == 0 ){
        console.log("no se puede dividir por 0")
    } else{
        return x1 + x2
    }
}
```

Mas adelante las vamos a ir modificando para mostrar las distintas formas de exportar funcionalidad.

Para importar un modulo en nodejs, en esta caso el módulo math.js la sintaxis es

```
const math = require('./math.js')
```

En este punto ya tenemos el módulo importado, y si intentamos llamar a una de las funciones declaradas en el mismo vemos que la reconoce

Sin embargo, al intentar ejecutar nos tira un error

```
sumar(x1: any, x2: any): any
console.log(math.sumar(1,2));
```

```
carlos@carlos-ThinkPad-T430:~/Escritorio/Programación/JavaScript/Node/Proyectos/Intruducción a nodejs/2 - Modulos$ node index.js
/home/carlos/Escritorio/Programación/JavaScript/Node/Proyectos/Intruducción a nodejs/2 - Modulos/index.js:5
console.log(math.sumar(1,2));
                ^

TypeError: math.sumar is not a function
    at Object.<anonymous> (/home/carlos/Escritorio/Programación/JavaScript/Node/Proyectos/Intruducción a nodejs/2 - Modulos/index.js:5:18)
    at Module._compile (module.js:652:30)
    at Object.Module._extensions..js (module.js:663:10)
    at Module.load (module.js:565:32)
    at tryModuleLoad (module.js:505:12)
    at Function.Module._load (module.js:497:3)
    at Function.Module.runMain (module.js:693:10)
    at startup (bootstrap_node.js:188:16)
    at bootstrap_node.js:609:3
```

Esto se debe a que en el archivo math.js hay que exportar la funcionalidad, las variable, o lo que sea que queramos usar desde otra parte. Todo modulo en nodejs es visto como un onjeto json, donde, entre otras cosas tiene un atributo llamado **exports**, en el que podemos agregar mas objetos, ya sea variables o funciones.

De

```
Module {
  id: '.',
  exports: {},
  parent: null,
  filename: '/home/carlos/Escritorio/Programación/JavaScript/Node/Proyecto
  loaded: false,
  children:
   [ Module {
       id: '/home/carlos/Escritorio/Programación/JavaScript/Node/Proyectos/
       exports: {},
```

hecho, si todavía no exportamos nada en el archivo math.js, y si lo importamos en index.js e imprimimos el atributo **exports** muestra undefined, ya que aún no tiene ningún valor asignado.

```
const math = require('./math.js')

console.log(math.exports);
```

Entonces, empecemos a exportar funciones desde math.js. La primer forma de exportar algo es agregandole al objeto exports un atributo que esté referenciando, en este caso, a una función.

```
module.exports.sumar = sumar;
```

Vemos que ahora sí funciona el llamado a la función

```
console.log(math.sumar(1,2)); // 3
```

Otra forma de agregar atributos al objeto exports es asignando directamente al elemento que estamos creando una función

```
exports.restar = function (x1, x2){
    return x1 - x2
}
```

o una función anónima.

```
exports.multiplicar = (x1, x2) => x1*x2;
```

Por último, también
se puede exportar varios elementos juntos asignando un json al elemento exports

```
module.exports = {
    dividir : (x1, x2) =>{
        if( x2 == 0 ){
            console.log("no se puede dividir por 0")
        } else{
            return x1 / x2
        }
    }
}
```

Hay que tener cuidado con este último método, ya que solo va a permitir usar fuera del módulo solo lo que se exportó dentro del json. También permite exportar solo una función en lugar de un objeto (module.exports = function (){}).

2.1 – Módulos preconstruidos

Son módulos que ya se encuentran instalados en el núcleo de NodeJS, se invocan de la misma manera que los módulos propios, utilizando require, excepto que no hace falta indicar el path del archivo.

Fuente: https://nodejs.org/docs/latest-v13.x/api/

Los módulos mas usados como ejemplo suelen ser el modulo os, que permite acceder a información del sistema operativo, y el módulo fs, para majejar el sistema de archivos.

3 – Llamadas asíncronas

El módulo file system (fs) es muy util para explicar como funciona nodejs con llamadas asíncronas. Se puede importar de la siguiente manera.

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

En nodejs las funciones asíncronas se identifican fácilmente porque requieren como parámetro otra función conocida como **callback**, que se ejecuta en al finalizar la operación requerida. En este caso llamamos a la función del sistema de archivos que permite crear y escribir un documento de texto.

```
fs.writeFile('./texto.txt', 'hola mundo en archivo', (err) =>{
    if (err){
        console.log('error al abrir escribir archivo')
    } else{
        console.log('escritura realizada')
    }
})
```

La función **writeFile** recibe tres parámetros, primero el path del archivo a crear/escribir, luego el texto, y por último una función que recibe como parámetro una variable que indica que ocurrió un error. Este **código** es **asíncrono** ya que la escritura de archivos no la realiza nodejs sino el sistema operativo, entonces el programa delega esta función y sigue ejecutando las siguientes lineas de código. Una vez que el SO termina de escribir el archivo se ejecuta la función de callback.

NodeJS está implementado de esta manera ya que originalmente fue pensado para atender a miles de usuarios de manera concurrente. Es por esto que una aplicación de servidor necesita que las funciones no sean bloqueantes, ya que esto evitaría que se pueda atender a otros clientes mientras se espera que finalice un proceso.

Es importante entender el funcionamiento de las funciones asíncronas ya que al crear aplicaciones en una arquitectura cliente servidor es muy común usar este tipo de funciones para gestionar las peticiones (request) y retornar información (response).

Para finalizar vemos el ejemplo de la lectura de archivos. La función readFile recibe dos paraámetros en su función de callback, uno para gestrionar errores, y otro para la información leida.

```
fs.readFile('texto.txt', (err, data)=>{
    if(err){
        console.log(err)
    }else{
        console.log(data.toString())
    }
})
```

4 – Servidor http

Para crear una aplicación de servidor por medio del protocolo http es necesario requerir el módulo **http.** Lo primero que hacemos es crear un servidor, que no es mas que una aplicación que escucha peticiones en un puerto especificado, esto se logra con la función **createServer()**. Esta función al llevar tiempo de ejecución también se comporta de manera asíncrona recibe como parámetro una función de callback, que a su vez tiene dos parámetros, generalmente llamados **req** y **res**. El primero es para almacenar información que llega con la petición del cliente (request) y el segundo para responderle (response).

En este ejemplo la petición viene vacía y en la respuesta devolvemos código html con el método **write**. Una vez terminada de cargar la respuesta se cierra el objeto res con el método **end**. Para finalizar se especifica en que puerto está escuchando el servidor.

```
1
2   const http = require('http')
3
4   const port = 3000
5
6   http.createServer((req, res)=>{
7       res.write('<h1> Hola mundo desde nodejs </h1>')
8       res.end
9   }).listen(port)
10
11  console.log (`Servidor a la escucha en puerto ${port}`)
12
13
```

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
carlos@carlos-ThinkPad-T430:~/Escritorio/Programación/JavaScript/Node/Proy
Servidor a la escucha en puerto 3000
```

En el navegador, entrando a **localhost:3000**

Hola mundo desde nodejs

Al crear un servidor es común responder no solo con el contenido de la página, sino con información acerca del resultado de la petición. Esto se hace agregando una cabecera al objeto res con un código de estado y un objeto json con distintos atributos html.

Códigos de estado HTTP: https://es.wikipedia.org/wiki/Anexo:Códigos_de_estado_HTTP

Agregamos por ejemplo información con un código de estado 200 y el tipo de contenido HTML.

```
http.createServer((req, res)=>{
    res.writeHead(200, {'content-type':'text/html'});
    res.write('<h1> Hola mundo desde nodejs </h1>');
    res.end;
}).listen(port);
```

Para actualizar el servidor es necesario cerrar el proceso en la terminal (ctrl+c) y volver a ejecutar el archivo index.js. Los cambios no se van a ver reflejados en la página ya que están en la cabecera, para verlos es necesario inspeccionarla desde las herramientas de desarrollador.

El código del servidor puede organizarse mejor almacenando el retorno de las llamadas a funciones en constantes, de este modo el servidor quedaría así.

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

const port = 3000

const handleServer = (req, res)=>{
    res.writeHead(200, {'Content-type':'text/html'});
    res.write('<h1> Hola mundo desde nodejs </h1>');
    res.end;
}

const server = http.createServer(handleServer);

const listenCallback = ()=> console.log (`Servidor a la escucha en puerto ${port}`);

server.listen(port, listenCallback);
```

5 – NPM

Al trabajar con NodeJS con aplicaciones grandes no es práctico hacerlo con javascript puro. Lo que suele hacerse es trabajar con frameworks y librerías. Para eso se utilizan módulos creados por terceros que son importados desde un repositorio de código llamado NPMJS (node package manager). Desde la página principal de NPM (https://www.npmjs.com) se puede buscar los módulos existentes para la funcionalidad que estamos queriendo desarrollar.

Para instalar módulos en el proyecto es necesario abrir una terminal en la carpeta donde está ubicado y ejecutar el comando `npm install <modulo>` (tambien se puede abreviar escribiendo solo i en lugar de install)**.** Al hacer esto, en la carpeta del proyecto se va a generar una nueva carpeta llamada **node_modules** que contiene todos los módulos que uno vaya importando al proyecto.

Al crecer el proyecto generalmente es necesario hacerlo portable, para poder desarrollar en equipo o instalarlo en un servidor remoto. Ya que los módulos son descargados de internet, no es necesario que la carpeta node_modules sea transportada todo el tiempo. Para esto existe el comando `npm init` que solicita información acerca del proyecto, como el nombre, la versión, repositorio de git, descripción, etc. y crea un archivo de configuración **package.json**.

Entre todo el contenido de este archivo un atributo escencial es la sección **dependencies**, donde se almacena la referencia a todos los módulos necesarios para el funcionamiento del programa. Luego al transportar el proyecto es necesario ejecutar en consola el comando `npm install` , el cual solicita al archivo package.json todas las dependencias del proyecto y las instala de manera automática.

En el ejemplo, antes de iniciar npm instalamos el modulo colors para dar colores y estilos a los mensajes de consola. Se puede ver como al ejecutar npm init en la sección dependencias aparece este módulo referenciado.

```
{
  "name": "npm",
  "version": "1.0.0",
  "description": "'curso nodejs, npm'",
  "main": "index.js",
  "dependencies": {
    "colors": "^1.4.0"
  },
  "devDependencies": {},
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

* Se puede realizar la prueba de eliminar la carpeta node_modules y volver a ejecutar npm init y se vuelve a instalar.

Otra sección importante del package.json es **scripts**. Esta permite definir distintas rutinas a ejecutar con un nombre específico. Por ejemplo, un script por defecto es **start**, donde se puede colocar el comando especifico para ejecutar el servidor (ahora es casi trivial porque es node index.js, pero mas adelante al usar express generator es mas útil ya que el comando es algo mas complicado). Entonces al ejecutar `npm run start` se ejecuta el script start (el comando **run** es general para cualquier script, aunque en realidad npm reconoce el script star así que en ese caso no necesitaria especificarlo).

Para finalizar, veamos un avance de lo que es **Express.**

Express es uno de los principales frameworks de nodejs, y sirve para implementar servidores de manera mas sencilla. Primero lo instalamos con el comando `npm i express` (notar que se agrega express a las dependencias, si no lo hace es porque al comando hay que agregar `- - save`).

Otra diferencia es que ahora al crear el servidor no se le asigna una función para atender una primer petición, sino que se se usa el método **get,** que recibe una **ruta** y una **funcion** con lo que debe retornar usando también los parámetros req y res.

Si lo que queremos es que al iniciar el servidor muestre un h1 este se envía con otro método de express llamado **send**. el código es el siguiente.

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

const server = express();
const port = 3000;
server.get("/", (req, res)=> {
    res.send('<h1> Hola mundo con express y nodejs </h1>');
    res.end();
});
server.listen (port, ()=> console.log(`servidor a la escucha en puerto ${port}`));
```