Introducción a node.js

Manuel Molino Milla

18 de octubre de 2017

Índice

1. Introducción

1.1. ¿Qué es node.js

- Es una plataforma en el lado del servidor.
- Se basa en el motor *V8 de google*. Aprovechando el motor V8 permite a node.js proporcionar un entorno de ejecución del lado del servidor que compila y ejecuta javascript a velocidades increíbles.
- Fue desarrollado por Ryan Dahl en 2009.
- Se basa en eventos y sobre todo es no bloqueante.
- Es *open source* y corre en todos los sistemas operativos.
- Encontramos proyectos de node.js en compañias como *eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, ...*
- Ofrece muy buena gestión de paquetes (librerías) gracias a *npm*

1.2. ¿Por qué javascript del lado del servidor?

- Encontramos JavaScript en gran número de páginas web y cada vez más código.
- Generalmente la parte del servidor quedaba relegada a otras soluciones como *php+apache* o *servlets de java* o *microsoft IIS+.NET o C#*
- JavaScript tiene la ventaja de poseer un excelente modelo de eventos, ideal para la programación asíncrona.
- JavaScript es conocido por muchos programadores.

1.3. ¿Qué problema resuelve Node?

- En lenguajes como Java y PHP, cada conexión genera un nuevo hilo que potencialmente viene acompañado de 2 MB de memoria.
- En un sistema que tiene 8 GB de RAM, esto da un número máximo teórico de conexiones concurrentes de cerca de 4.000 usuarios.
- A medida que crece el número de usuarios se necesitará agregar más servidores.
- Por todas estas razones, el cuello de botella en toda la arquitectura de aplicación Web (incluyendo el rendimiento del tráfico, la velocidad de procesador y la velocidad de memoria) era el número máximo de conexiones concurrentes que podía manejar un servidor.
- Node resuelve este problema cambiando la forma en que se realiza una conexión con el servidor.

1.4. Procesos bloqueantes

- node.js se caracteriza porque es **no bloqueante** y **orientado a eventos**
- Es similar como un navegador maneja JavaScript.
- Fijándonos en los siguientes datos:

L1 cache read	0.5 nanoseconds
L2 cache read	7 nanoseconds
RAM	100 nanoseconds
Read 4 KB randomly from SSD	150,000 ns
Read 1 MB sequentially from SSD	1,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns

- Observamos que estas operaciones I/O son bastante costosas y cuando se ejecutan dentro de un programa, este deberá esperar a que termine dicha operación para seguir la ejecución normal del programa.

Ejemplo de programa:

```
console.log('1');
var log = fileSystemReader.read('./verybigfile.txt');
console.log('2');
```

- La función fileSystemReader.read() se encarga de llamar al SO para proceder a la lectura de un fichero de texto.
- Esto es un cuello de botella, hasta que no se acabe la lectura no podremos continuar el programa.
- Se trata de una operación bloqueante, hasta que no acabe no prosigue el programa.
- Este modelo es costoso y presenta mucha latencia, pues cada proceso tiene asociado una memoria y un estado que no se libera hasta que el programa no acabe.
- Una forma de solucionar esto es usando *hilos*, este es un ejemplo en *Java*:

```
public class TestHilos {
  public static void main(String[] args) {
  // lanzamos hilo independiente que se para 5 segundos
  // simula un proceso costoso I/O
    System.out.println("Antes de llamar al hilo");
    Thread lanzarHilo = new Thread(new HiloIndependiente());
    lanzarHilo.start();
    System.out.println("Despues del hilo");
    // paramos el programa principal 5 segundos
    // simula un proceso costoso I/O
    System.out.println("Sin hilos, comienzo");
    try {
      Thread.currentThread().sleep(5_000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    System.out.println("Sin hilos, fin");
  }
}
class HiloIndependiente implements Runnable{
  @Override
  public void run() {
    try {
      Thread.currentThread().sleep(5_000);
    } catch (InterruptedException e) {
    // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
}
```

- El uso de hilos hace que puedan compartirse datos en el mismo espacio de memoria donde se ejecuta el proceso.
- El problema que surge es el de compartición de recursos. Debemos sincronizar los mismos, esto dificulta la programación.
- En el caso de los navegadores, la gestión I/O es diferente.
- Ocurren fuera del *hilo principal* y un *evento* es emitido cuando I/O finaliza.
- El evento es manejado por una *callback*
- Este tipo de gestión I/O es *no bloqueante* y *asíncrona*
- Como I/O es no va a bloquear, el hilo principal continua y getiona otros eventos sin esperar a la culminación del proceso I/O
- node.js trabaja de forma similar.

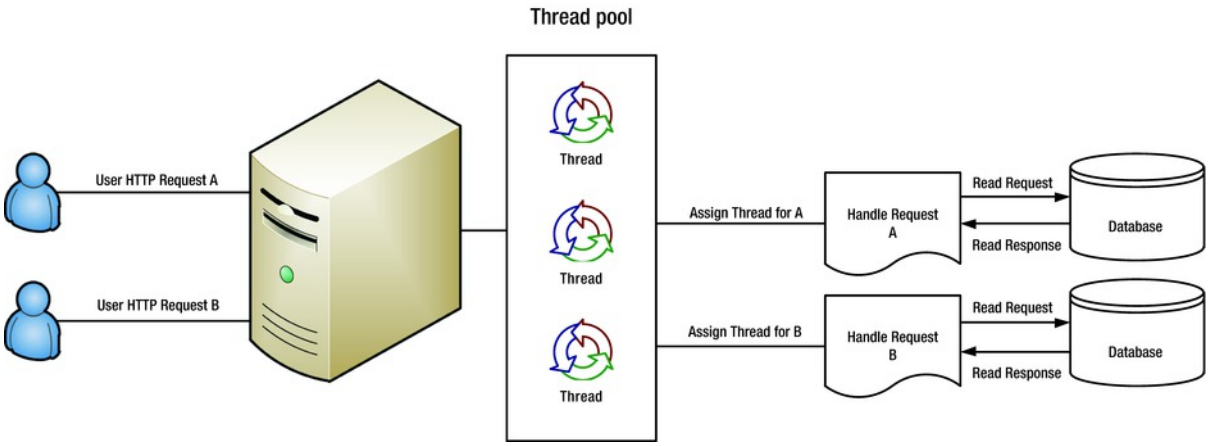

Figura 1: Servidor tradicional

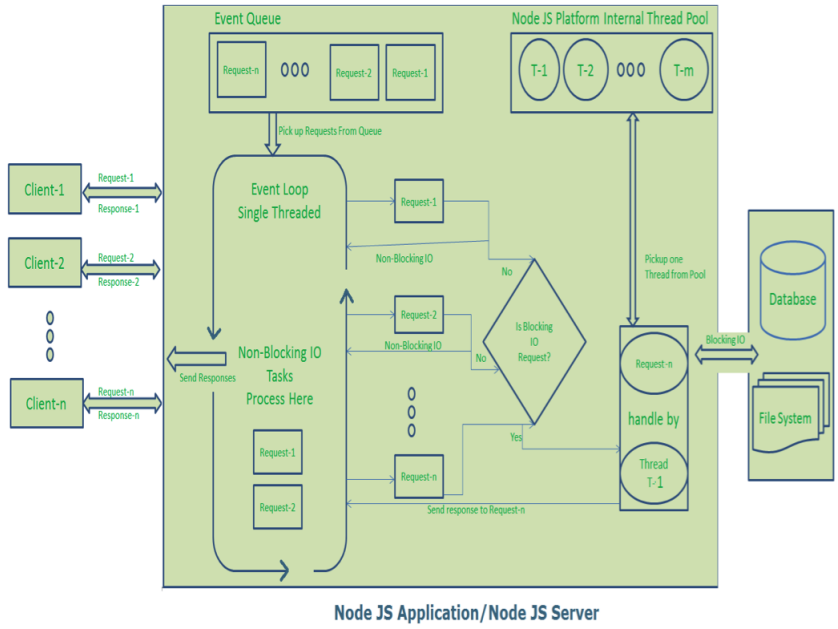

Figura 2: Servidor no bloqueante

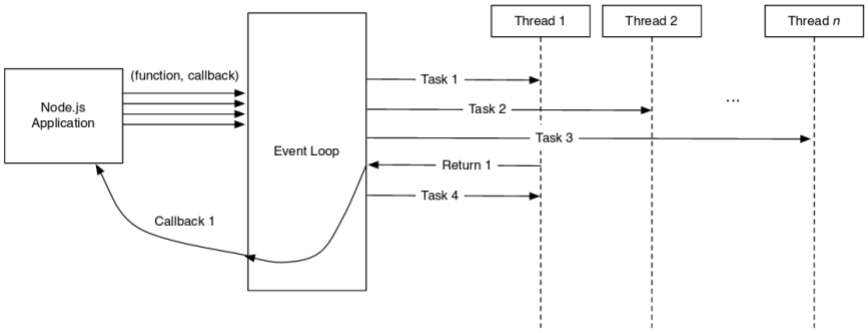

Figure 5: A closer look at the inner workings of the Event Loop.

Figura 3: node js

1.5. callback

- Es una función que hace que node.js trabaje de forma *asíncrona*
- El caso de la lectura del fichero

En el caso *síncrono*:

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

var data = fs.readFile('/etc/passwd');
console.log(data); // => origina undefined

console.log('Fin del programa');
```

Y el caso *asíncrono*:

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

var data = fs.readFile('/etc/passwd','utf8', function(err,data){
  if (err)
   throw err;
  else
   console.log(data); // => muestra el contenido del fichero
});

console.log('fin del fichero');
```

Otro ejemplo sería un programa que con la ayuda de una función tarda medio segundo en calcular el valor de la suma, simulando un proceso bloqueante. Nos ayudamos de la función *setTimeout* cuyo segundo argumento es el tiempo de respuesta:
En el caso síncrono el código sería:

```
function suma(numero_uno,numero_dos){
   setTimeout(function(){
      var resultado = numero_uno + numero_dos;
      return resultado;
   }, 500);
}

var resultado = suma(2,5)
console.log(resultado);
```

Y en el caso asíncrono sería:

```
function suma(numero_uno,numero_dos,callback){
   setTimeout(function(){
      var resultado = numero_uno + numero_dos;
      callback(resultado);
   }, 500);
}
suma(2,5,function(resultado){
   console.log(resultado);
})
```

- El parámetro *callback*. Éste nos ayudará a retornar el resultado cuando esté listo.
- El valor devuelto por la función no se asigna a una variable, sino se usa como parámetro la función que devuelve el valor.

Conclusiones:

- Las callback en vez de retornar el resultado como la mayoría de las funciones, estás toman cierto tiempo en devolver el resultado.

Otro ejemplo usando como I/O una petición web:
Síncrono:

```
var request = require('request');
var status = undefined;
request('http://google.com', function (error, response, body) {
    if (!error && response.statusCode == 200) {
      status = response.statusCode;
    }
    });
console.log(status);
console.log('fin de programa');
```

Asíncrono:

```
var request = require('request');
var status = undefined;
function getSiteStatus(callback){
  request('http://google.com', function (error, response, body) {
    if (!error && response.statusCode == 200) {
      status_code = response.statusCode;
    }
    callback(status_code);
    });
}
function showStatusCode(status){
  console.log(status);
}
getSiteStatus(showStatusCode);
console.log('fin de programa');

/*tambien puede ser:
getSiteStatus(function(status){
  console.log(status);
});*/
```

Hay que tener cuidado en el caso que usemos funciones predefinidas que no sean asíncronas, como puede ser la función *sort* que ordena un array. Para esto usamos *process.nextTick*

```
function coleccion (valor) {
  var _valor = valor;
  var funcionDeComparacion = function (elem1, elem2) {return elem1
      -elem2;};
  return {
    ordenar : function(callback) {
      process.nextTick(function() {
        callback(_valor.sort(funcionDeComparacion));
      });
    }
  }
}
```

1.6. Programación orientada a eventos en node.js

- node.js funciona en un único hilo de ejecución, pero soporta concurrencia mendiante eventos y callback.
- Los eventos son parecidos a las callback, pero NO lo son.
- Las callback retornan un resultado cuando una función asíncrona las usa.
- En cambio existen funciones que escuchan eventos y actúan como observadoras.

Ejemplo de un servidor web usando eventos:

```
var http = require("http");
var server = http.createServer();

server.on("request", function (req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(3000);
```

Y usando callback (Sacado de la página oficial)

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

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

El mismo servidor, también usando callback, pero de diferente forma:

```
var http = require("http").createServer(function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
}).listen(3000);
console.log("Server is listening");
```

1.7. Módulos en node.js

- Es importante la organización del código, sobre todo cuando el número de líneas aumenta.
- Para esto podemos crear módulos independientes.
- Usamos la palabra reservada *exports*

Ejemplo 1, fichero *geometria.js*:

```
exports.area = function (r) {
  return 3.14 * r * r;
};
exports.circumference = function (r) {
  return 3.14 * 3.14 * r;
};
```

Qué lo podemos usar:

```
var geo = require('./geometria.js');
console.log(geo.area(2));
```

Otra forma de trabajar es usar *modules.exports* y clousures

```
function circunferencia (radio) {
  _radio = radio;
  return {
    area:      function() { return _radio * _radio * 3.14 ; },
    perimetro: function() { return 2 * 3.14 * _radio }
  }
}
module.exports = circunferencia;
#module.exports.circunferencia = circunferencia (de otra manera)
```

Y crear un main.js como:

```
var circunferencia = require('./circunferencia');

console.log(circunferencia(4).area());
console.log(circunferencia(4).perimetro());
```

Y usando un constructor quedaría:

```
function Circunferencia (radio) {
  this.radio = radio;
}

Circunferencia.prototype = {
  area     : function() { return this.radio * this.radio * 3.14; },
  perimetro: function() { return this.radio * 2 * 3.14; }
}

module.exports = Circunferencia;
```

Y el main.js:

```
var Circunferencia = require('./Circunferencia');

var circunferencia = new Circunferencia(4);

console.log(circunferencia.area());
console.log(circunferencia.perimetro());
```

Otro ejemplo puede ser la configuración del acceso a una BD:

```
var db_config = {
  server: "0.0.0.0",
  port: "3306",
  user: "mysql",
  password: "mysql"
};
module.exports = db_config;
```

Y su uso podría ser:

```
var config = require('./config.js');
console.log(config.user);
```

1.8. Ejercicios

- Crea un módulo denominado *estadistica.js* que dado una colección de números nos diga:
  - El número de números de la colección.
  - El valor mas alto.
  - El valor mas bajo.
  - Dado un número como argumento, nos indique cuantas veces se repite en la colección.
  - No entregue la colección ordenada.
  - El valor promedio de la colección.
- Convierte el ejercicio de *cadenas.js* en un módulo.
- Idem con el ejercicio de dni y nif.

1.9. npm

- Es el gestor de módulos de node.js
- Viene integrado en el proceso de instalación de node.js
- Es la herramienta perfecta y necesaria en cuando a manejo de dependencias se refiere.

Los comandos mas interesantes son:

npm init permite crear, modificar y generar un *package.json*

npm search *nombre-del-modulo* o *palabra-clave* permite buscar en el registro de npm, algun modulo acerca de la palabra clave

npm info nombre-del-modulo Muestra información en formato json acerca de nombre-del-modulo

npm ls, npm la, npm ll, npm list lista los paquetes actualmente instalados en un determinado directorio. En formato de tree.

npm outdated busca si hay versiones más nuevas acerca de los modulos instalados en el directorio actual.

npm install *nombre-del-módulo* instala el módulo en el directoria actual en una carpeta denominada *modules*. Si se hace con la opción *-g* se instala de forma global en el sistema. Y con la opción *–save* se añade como dependencia al fichero *package.json*

npm uninstall *nombre-del-módulo* desinstala el módulo determinado. Admite las opciones anteriores de la instalación.

```
{
  "name": "best-practices",
  "description": "A package using versioning best-practices",
  "author": "Charlie Robbins <charlie@nodejitsu.com>",
  "dependencies": {
    "colors": "0.x.x",
    "express": "2.3.x",
    "optimist": "0.2.x"
  },
  "devDependencies": {
    "vows": "0.5.x"
  },
  "engine": "node >= 0.4.1"
}
```

1.10. Ejercicio final

Tenemos el fichero *datos.csv* que contempla 1000 registros de datos con el nombre, primer apellido, email, edad y sexo, y queremos implementar un programa en *nodejs* que primero lea dicho fichero como argumento del programa, ejemplo:

node leerFicheroCSV.js datos.csv

Nos de un mensaje en el caso que no encuetre el fichero y la lectura no debe ser bloqueante, comprueba dicho requisito
A la vez que se leen los datos, se crea un array de objetos persona que tienen como atributos cada uno de los campos del fichero CSV.
En un módulo aparte, crea un programa que dada la colección de objetos anterior nos de:

- La mayor edad y menor edad como un objeto.
- Una colección de objetos de persona con mas edad.
- Dado el sexo (M o F) nos de una lista de personas de igual sexo.