Desarrollo de Aplicaciones Web

Docente: Brian Ducca

Node Js

Node Js es un entorno del lado del servidor basado en el motor Javascript V8 de Chrome. Es orientado a eventos, y trabaja sin bloquear el I/O (funcionando de manera asíncrona)

Se pueden construir aplicaciones de línea de comando, aplicaciones web, chats realtime, Api REST, entre otros.

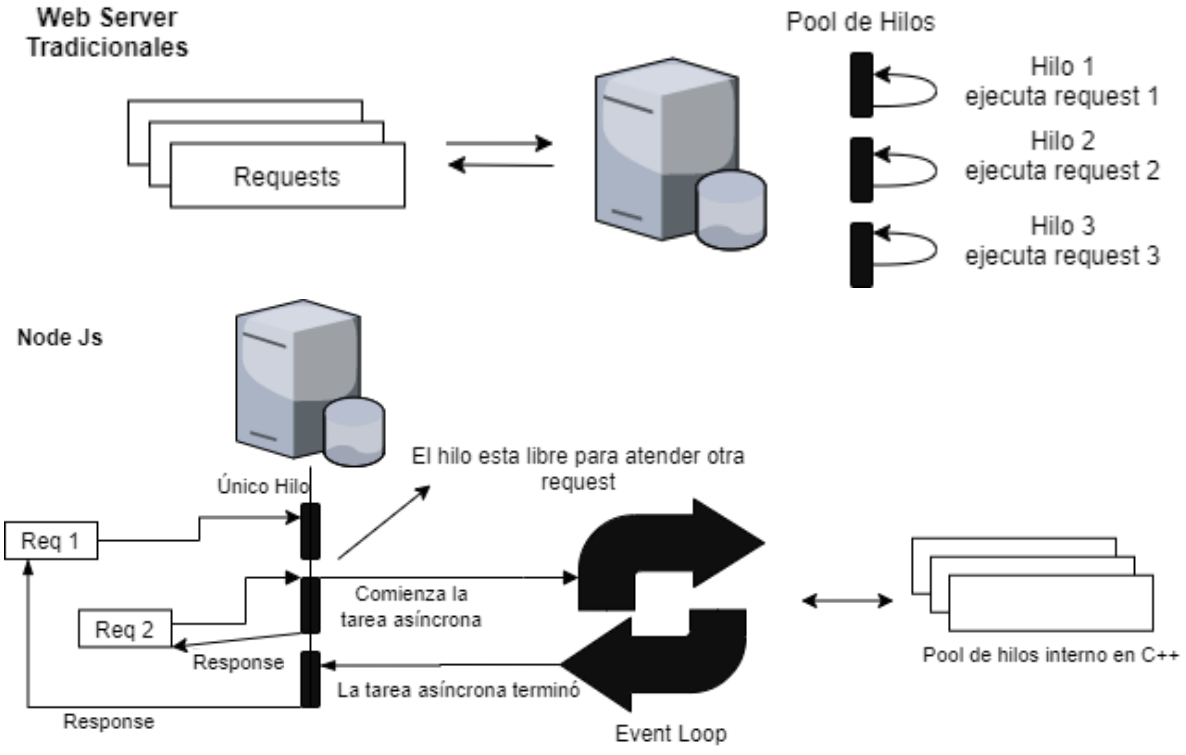
Ventajas de Node JS

- Open Source
- Utiliza Javascript
- Asíncrono por defecto(mejor performance)
- Multiplataforma (corre en Windows, MacOs y Linux)
- Liviano

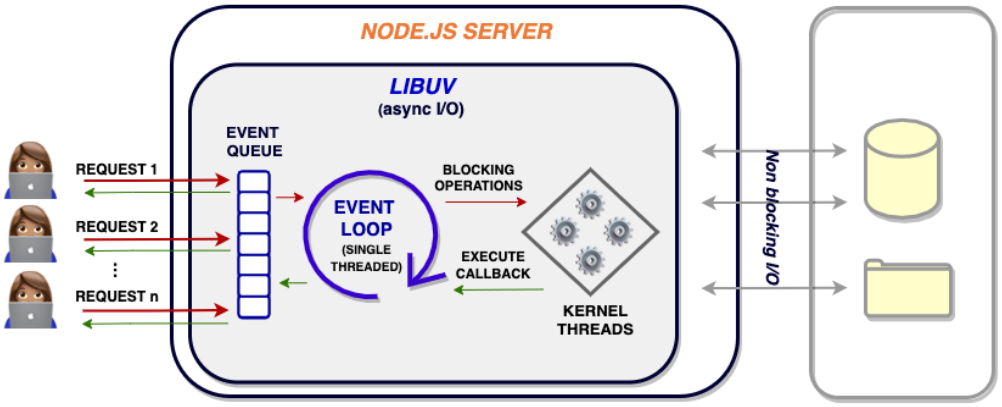
¿Cómo funciona?

NodeJS funciona distinto a los web server tradicionales, dado que procesa las request de los usuarios de manera diferente, tiene un **único** proceso y el código de la aplicación corre en un solo hilo, es por ello que consume menos recursos que otras plataformas. Todo el trabajo I/O se va a realizar de manera asíncrona, para que el único hilo que posee, no tenga que esperar a que se complete esa request y queda libre para manejar otras request. Cuando el trabajo de I/O asíncrono se completo, termina de procesar la request y manda la response.

Posee un “event loop” que constantemente está mirando eventos entrantes para realizar el trabajo asíncrono y ejecutar el callback cuando la tarea termine.



Entonces, tendremos un único hilo principal (Main Thread), cuya responsabilidad es ejecutar la parte sincrónica de la request, y luego para la operación asincrónica se crea un Background Thread que hará la operación de I/O. A pesar de tener múltiples hilos en el “background” , todas las request llegan al hilo principal y luego Node internamente maneja la ejecución asincrónica de la tarea.



I/O (Input/output)

Cuando nos referimos a I/O, hacemos referencia a la interacción del programa con el disco y la red. Algunos ejemplos de I/O serían leer/escribir datos en el disco, realizar llamadas HTTP, realizar operaciones con la base de datos. Estos son más “lentos” que si las comparamos con las operaciones que realizan trabajo con la CPU o acceden a la memoria (RAM)

Sincronismo vs Asincronismo

Sincronismo es cuando la ejecución de código es en secuencia, es decir línea por línea, una a la vez. Cada vez que se llama a una función, se espera al retorno de la misma antes de continuar con la siguiente línea de código.

Asincronismo por el otro lado, no espera a que se complete la tarea y continúa con la siguiente.

Ejemplo:

```
// Synchronous: 1,2,3
alert(1);
alert(2);
alert(3);

// Asynchronous: 1,3,2
alert(1);
setTimeout(() => alert(2), 0);
alert(3);
```

En este ejemplo utilizamos el setTimeout para simular un asincronismo.

Blocking vs Non-Blocking I/O

Cuando hablamos de operaciones que bloquean el I/O nos referimos a las que bloquean la ejecución de código hasta que la operación termine, y las que no bloquean continúan su ejecución.
Es decir, las que bloquean se ejecutan de manera sincrónica y las otras de manera asincrónica.

Ejemplo:

```
// Blocking
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until
file is read
console.log(data);
moreWork(); // will run after console.log

// Non-blocking
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
moreWork(); // will run before console.log
```

En el primer ejemplo, el console.log será llamado antes de llamar a moreWork. Por otra parte, en el segundo ejemplo, fs.readFile() es no bloquea la ejecución de Javascript, entonces continua y moreWork() va a ser llamado primero.

En las no bloqueantes, no se espera a que la operación se complete, sino que continúa con la ejecución de código y tienen un callback que es llamado cuando la operación se completa.

Instalación

Para instalar Node Js con alguna distribución de linux (como ejemplo utilizaremos ubuntu) realizaremos los siguientes pasos:

1. Actualizar el packet manager con los siguientes comandos
    a. ***sudo apt-get update***
    b. ***sudo apt-get upgrade***
2. Instalamos las librerias necesarias (en caso de que no las tengamos ya instaladas)
    a. ***sudo apt-get install python-software-properties***
    b. ***sudo apt-get install curl***
3. Agregamos el archivo de paquetes personal de Node a nuestro S.O
    a. ***curl -sL https://deb.nodesource.com/setup_16.x | sudo -E bash −***

    Nota: En este caso estaremos instalando la versión 16 de node, en el caso de querer otra versión, simplemente reemplazar setup_16.x con setup_11.x por ejemplo.
4. Instalamos NodeJS y npm en nuestro ubuntu
    a. ***sudo apt-get install nodejs***
5. Verificamos que se haya instalado de manera correcta con los siguientes comandos
    a. ***node -v*** *o* ***node −version***
    b. ***npm -v*** *o* ***npm −version***

Errores comunes que pueden surgir :
https://stackoverflow.com/questions/62028180/ubuntu-19-04-error-404-not-found-ip-91-189-95-83-80-error-on-apt-update

Callbacks

Un Callback (llamada de vuelta) es una función que se ejecutará después de que otra función haya terminado de ejecutarse, de aquí el nombre de Callback.
Es decir, es una función que se pasa a otra función como un argumento, que luego se invoca dentro de la función externa para completar algún tipo de rutina o acción.

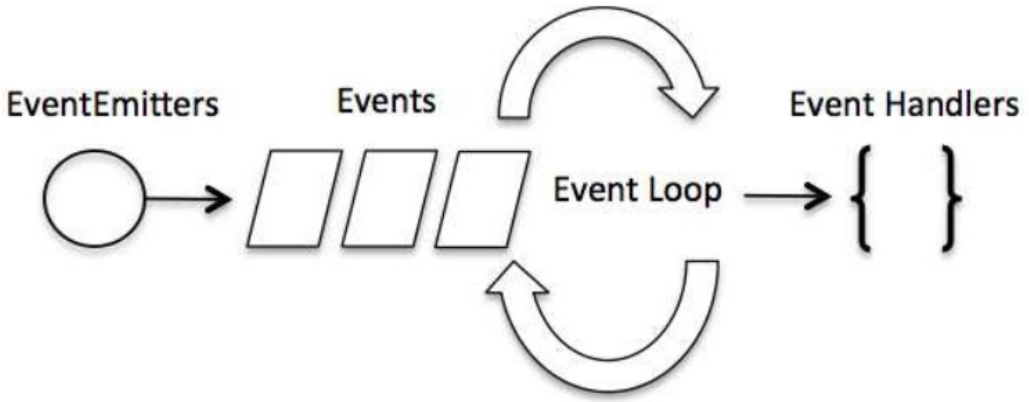
En JavaScript, las funciones admiten funciones como argumentos y pueden ser devueltas por otras. Las funciones que hacen esto se denominan funciones de orden alto (high-order).
Cualquier función que se pase como argumento se denomina función Callback.

Ejemplo:

```
function hacerTarea(nombre, callback) {
 alert(`Comenzando la tarea ${nombre}.`);
 callback();
}
function tareaTerminada(){
 alert('Fin de la tarea');
}
hacerTarea('Programación', tareaTerminada);
```

Programación orientada a eventos

Node JS utiliza este concepto junto con el de callbacks para poder dar soporte a la concurrencia. La programación orientada a eventos consiste en un loop principal que escucha eventos y dispara una función callback cuando uno de estos eventos se detectó.



La diferencia con los callbacks es que las funciones callback son llamadas una vez que una función asíncrona retornó el resultado, pero en eventos se trabaja con el patron “observer”. Las funciones que escuchan los eventos actuan como “Observers” y cuando se lanza un evento, la función que estaba escuchando este evento, comienza su ejecución.

Ejemplo:

```
// Importamos el modulo de eventos
var eventos = require('events');

// Creamos un objeto eventEmitter
var eventEmitter = new eventos.EventEmitter();

// Creamos un handler del evento
var connectHandler = function connected() {
   console.log('Conexión exitosa.');

   // Lanzamos el evento data_received
   eventEmitter.emit('data_received');
}

// Bindeamos el evento connection con el handler
eventEmitter.on('connection', connectHandler);

// Bindeamos el eventocon una función anónima
eventEmitter.on('data_received', function() {
   console.log('Llego la data.');
});

// Lanzamos el evento connection
eventEmitter.emit('connection');
```

Cabe destacar que el método “on” del EventEmitter no realiza verificaciones para ver si ya se había agregado el listener previamente, es decir que si esto se realiza múltiples veces, se ejecutará el evento cuantas veces se haya bindeado.

Module.exports & require

Antes de comprender para que sirven, debemos entender cúal es el propósito de los módulos en Node. Para ello, debemos pensar que un módulo, es código que agrupamos para poder reutilizar y compartir en otros lados de nuestra app.
Entonces, sabiendo la función que va a tener nuestro módulo, la instrucción module.exports le dice a Node, que porción de código va a exportar desde ese archivo, para que esté disponible a los otros.

Con esto podemos ver que “module” es simplemente un objeto que se pasa entre módulos y dentro de ese objeto , exports es una property que va a contener las variables y funciones que nosotros le vayamos asignando a lo largo de nuestra app.

Ejemplo en archivo datos.js:

```
module.exports = { nombre: 'Brian', apellido: 'Ducca' }
```

Si ya entonces exportamos el código que necesitamos, ahora necesitamos de alguna manera importarlo en otro archivo para poner tener a disposición el código. “Require” es una función que vamos a utilizar para dicha importación.

En el archivo index.js:

```
let persona = require('./datos.js');
console.log(persona.nombre + ' ' + persona.apellido);
```

Require va a buscar por módulos de la siguiente manera:

1. Busca por módulos “Core” con el path dado, estos módulos son:
    - http (https://nodejs.org/api/http.html)
    - url (https://nodejs.org/api/url.html)
    - querystring (https://nodejs.org/api/querystring.html)
    - path (https://nodejs.org/api/path.html)
    - fs (https://nodejs.org/api/fs.html)
    - util (https://nodejs.org/api/util.html)
2. Busca por algún package dentro de node_modules que tenga el nombre especificado en el path.
3. Busca un archivo o directorio con el nombre dado en el path.
4. Si no encuentra ninguno de ellos, arroja un error

Express JS

Express es un framework web, escrito en JavaScript y alojado dentro del entorno de ejecución NodeJS. Es robusto, rápido, flexible y muy simple. Soporta los métodos GET, POST, PUT, DELETE entre otros y posee un método de direccionamiento especial que no se deriva de ningún método HTTP (.all).

Instalación

```
npm install --save express
```

Ejemplo para levantar una API:

Archivo index.js

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

```
var app = express();

app.get('/', function (req, res) {
   res.send('Hola Mundo');
})

var server = app.listen(8081, function () {
   var host = server.address().address
   var port = server.address().port

   console.log("API levantada en http://%s:%s", host, port)
})
```

Y lo ejecutamos en la consola de la siguiente manera:

```
node index.js
```

La definición de ruta tiene la siguiente estructura:
app.MÉTODO(ruta,handler)

Donde:
app es una instancia de express.
MÉTODO es un método de solicitud HTTP.
RUTA es una vía de acceso a un servidor
HANDLER es la función que se ejecuta cuando se correlaciona la ruta.

Objeto Request

Representa a la request HTTP y algunas de sus properties más útiles son:

- req.body
  - Contiene un par de clave-valor de los datos que se enviaron en el cuerpo de la request. Por defecto, su valor es “undefined” y se va a llenar con valores cuando usemos el “body-parsing” middleware

Ejemplo:

```
var express = require('express')

var app = express()

app.use(express.json()) // para parsear application/json

app.post('/perfil, function (req, res, next) {
```

```
  console.log(req.body)
  res.json(req.body)
})
```

- req.ip
  - Contiene la dirección IP de la request
- req.params
  - Esta propiedad es un objeto de propiedades vinculadas a los parámetros de ruta (GET), por ejemplo, si tenemos la ruta "/usuario/:id", la propiedad "id" la podremos ver usando "req.params.id". Por defecto el valor del objeto es {}
- req.secure
  - Booleano para ver si se estableció una conexión TLS.

Objeto Response

Representa la respuesta HTTP que Express envía cuando recibe una request HTTP.

Algunos métodos que podemos utilizar con Express:

METODO	DESCRIPCIÓN
res.download(path [, filename] [, options] [, fn])	Solicita un archivo para descargarlo.
res.end()	Finaliza el proceso de respuesta sin data.
res.json([body])	Envía una respuesta JSON.
res.jsonp()	Envía una respuesta JSON con soporte JSONP.
res.redirect()	Redirecciona una solicitud.
res.render()	Representa una plantilla de vista.

res.send([body])	Envía una respuesta de varios tipos. En el body puede ir un objeto Buffer, un String, un objeto o un Array por ejemplo.
res.status(code)	Setea el estado de la respuesta HTTP, permite encadenarse con los otros métodos.
res.sendStatus()	Establece el código de estado de la respuesta y envía su representación de serie como el cuerpo de respuesta.

Algunos ejemplos:

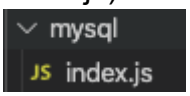
```
res.download('/reporteVentas.pdf', 'ventas.pdf');

res.json({ nombre: 'brian' , apellido:'ducca' });

res.redirect('http://www.google.com');

res.send({ usuario: 'bducca' });

res.send([1, 2, 3]);

res.send('test');

res.status(403).end();

res.status(400).send('Bad Request');

res.sendStatus(404) // igual a hacer res.status(404).send('Not Found')
```

MySql + Express JS

Para conectar una base de datos MySql con express, primero debemos instalar el módulo mysql realizando:

```
npm install --save mysql
```

Luego nos vamos a crear nuestro módulo de conexión a la base de datos e ingresaremos lo siguiente (para ello generamos una nueva carpeta mysql en nuestro raiz del proyecto de API y dentro de ella creamos un nuevo archivo index.js):




```
var mysql = require('mysql');
var connection = mysql.createConnection({
    host: '127.0.0.1',
    port: '3307',
    user: 'root',
    password: 'userpass'
});

connection.connect(function(err) {
    if (err) {
        console.error('Error al conectar: ' + err.stack);
        return;
    }

    console.log('Conectado con id ' + connection.threadId);
});

module.exports = connection;
```

Una vez realizado esto, importaremos nuestro módulo cuando queramos realizar alguna consulta a la base de datos de las siguientes posibles maneras:

- La primera consiste en pasar los siguientes parámetros (sqlString,callback)

```
connection.query('SELECT * FROM `books` WHERE `author` = "David"',
function (error, results, fields) {
  // error contendra los errores si hubiese en la query
```

```
  // results tendra la salida de la query
  // fields contedra si lo hay, información sobre los campos
});
```

- La segunda consiste en pasar (sqlString,valores,callback)

```
connection.query('SELECT * FROM `books` WHERE `author` = ?',
['David'], function (error, results, fields) {
  // error contendra los errores si hubiese en la query
  // results tendra la salida de la query
  // fields contedra si lo hay, información sobre los campos
});
```

Y lo haremos de la siguiente manera (Ejemplo dentro de index.js de la API):

```
var express = require('express');
var app = express();
//importamos el contenido de la conexion a la base de datos en la variable
mysql
var mysql = require('./mysql');
app.use(express.json()); // para parsear application/json
...

app.get('/', function(req, res, next) {
   mysql.query('Select * from DAW.Devices', function(err, rta, field) {
       if (err) {
           res.send(err).status(400);
           return;
       }
       res.send(rta);
   });
});

...
```