



# NODE.JS

© 2017, ACTIBYTI PROJECT SLU, Barcelona  
Autor: Ricardo Ahumada



MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

**red.es**



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



**UNIÓN EUROPEA**

Fondo Social Europeo  
*"El FSE invierte en tu futuro"*

# ÍNDICE DE CONTENIDOS

1. Introducción
2. Instalación
3. Use cases
4. Primera app
5. Modules
6. Callbacks
7. Eventos
8. Streams
9. Buffers

1

# Introducción

# Node.js

- Node.js es una plataforma construida en tiempo de ejecución de JavaScript de Chrome para construir fácilmente aplicaciones de red rápidas y escalables.
- Node.js utiliza un modelo orientado a eventos, sin bloqueo de E/S que hace que sea ligero y eficiente, ideal para aplicaciones en tiempo real de datos intensivos que se ejecutan a través de dispositivos distribuidos.
- Node.js también proporciona una amplia biblioteca de varios módulos de JavaScript que simplifica el desarrollo de aplicaciones web usando Node.js en gran medida.



# Características de Node.js

- **Asíncrono y controlado por eventos:** todas las API de la biblioteca Node.js son asíncronas, es decir, sin bloqueo. Significa esencialmente que un servidor basado en Node.js nunca espera a que una API devuelva datos. El servidor se mueve a la siguiente API después de llamarla y un mecanismo de notificaciones de Eventos de Node.js ayuda al servidor a obtener una respuesta de la API llamada anteriormente.
- **Muy rápido:** Construido sobre el motor de JavaScript V8 de Google Chrome, la biblioteca Node.js es muy rápida en la ejecución de código.
- **Único subprocesso pero altamente escalable:** Node.js utiliza un único modelo de roscado con el evento de bucle.
- **Sin búfer:** las aplicaciones de Node.js nunca almacenan cualquier dato. Estas aplicaciones simplemente dan salida a los datos en fragmentos.

2

## Instalación

# Cómo instalar Node.js en Windows

- Para la instalación y/o configuración de Node.js es necesario los siguientes dos softwares, instalados en el equipo:
  - Editor de texto.
  - Los binarios instalables Node.js.
- Los binarios de Node.js son por decirlo de una forma explícita y sencilla, las librerías de la codificación para el uso de node, sin las librerías al hacer un llamado a node, éste no sabrá el significado de dicho código de llamado por lo tanto no se ejecutará la aplicación con las funciones de Node.js.



## Archivos necesarios

- Editores de Texto de su preferencia, algunos recomendados:
  - SublimeText 3 -> <https://www.sublimetext.com/3>
  - Atom -> <https://atom.io/>
  
- Binarios de Node.js -> <https://nodejs.org/en/download/>

# Descargando binarios de Node.js



- En primer lugar vemos que podemos seleccionar entre:
  - **LTS** (*recomendada para la mayoría de usuarios*): es la versión de Node.js con Long Term Support (LTS), es decir a la que se le da soporte a largo plazo. Esta versión puede **no tener** disponibles las últimas tecnologías que todavía no se consideran estables.
  - **Current**: esta es la versión más reciente de Node.js e incluye todas las funcionalidades, incluso aquellas más novedosas y que no se consideran estables.
- Por norma general, seleccionar la versión LTS es la opción recomendada y estable.

# Descargando binarios de Node.js



- En cuanto a los paquetes de instalación en Windows disponemos de:
  - **Windows Installer (.msi)**: es un instalador con extensión .msi que se encargará de automatizar todo el proceso de instalación, agregando las variables de sistema automáticamente y además incluye el gestor de módulos de Node.js conocido como NPM (Node Package Manager).
  - **Windows Binary (.exe)**: con esta opción descargaremos únicamente el ejecutable de Node.js, el cual no incluye NPM.
- La opción recomendada es descargar el instalador .msi, ya que es el más completo y además se encarga automáticamente de configurar nuestro sistema.

# Instalación en Windows



- Utilice el archivo .msi y siga las instrucciones para instalar Node.js. De forma predeterminada, el instalador utiliza la distribución Node.js en C:\Archivos de programa\nodejs. El instalador debe establecer el directorio C:\Archivos de programa\nodejs\bin en la variable de entorno PATH de la ventana. Reinicie cualquier mensaje de comandos abierto para que el cambio surta efecto.



# Verificar la instalación: Ejecución de un archivo

- Creamos un archivo llamado JS **main.js** en su máquina (Windows o Linux), con el siguiente código:

```
/* Hello, World! Programs is node.js */  
Console.log("Hello, World!")
```

- Una vez instalado, vamos a probar que todo funciona correctamente, para ello abrimos la consola de comandos y ejecutamos el comando “node”. Si todo ha ido bien, estaremos dentro de la consola de Node.js y podemos ejecutar la siguiente línea de código:

```
console.log("hola mundo");
```

The screenshot shows a Windows command prompt window titled 'cmd' with the path 'C:\Windows\system32\cmd.exe'. The window displays the following text:  
Microsoft Windows [Versión 6.1.7601]  
Copyright © 2009 Microsoft Corporation. Todos los derechos reservados.  
C:\Users\UA>node  
> console.log("hola mundo");  
hola mundo  
undefined  
> .exit  
C:\Users\UA>

3

## Use cases

# Dónde debe utilizarse Node.js

- Node.js es un entorno Javascript del lado del servidor, basado en eventos. Node ejecuta javascript utilizando el motor V8, desarrollado por Google para uso de su navegador Chrome.
- El motor V8 permite a Node proporciona un entorno de ejecución del lado del servidor que compila y ejecuta javascript a velocidades increíbles. El aumento de velocidad es importante debido a que V8 compila Javascript en código de máquina nativo, en lugar de interpretarlo o ejecutarlo como bytecode.
- Node.js trabaja en tiempo real, recibiendo datos instantáneamente sin pasar de una página a otra; es multiusuario, se comunica con JSON de manera muy eficiente sin realizar tantas conversiones de un lenguaje a otro. Puede trabajar con conexiones simultaneas, ayudar en el trafico de la base de datos.
- Node es usado en páginas web como ebay, microsoft, paypal, wikipedia y muchas otras.
- Node es de código abierto, y se ejecuta en Mac OS X, Windows y Linux.

# Dónde debe utilizarse Node.js

## ➤ CHAT

- Chat es la aplicación más común en tiempo real y multiusuario, es realmente el ejemplo especial para Node.js: es una aplicación ligera, de alto tráfico, de datos intensivos (pero de bajo procesamiento/cálculo) que se ejecuta a través de dispositivos distribuidos. También es un gran caso de uso para aprender también, ya que es simple, pero cubre la mayoría de los paradigmas que usará en una aplicación típica de Node.js.

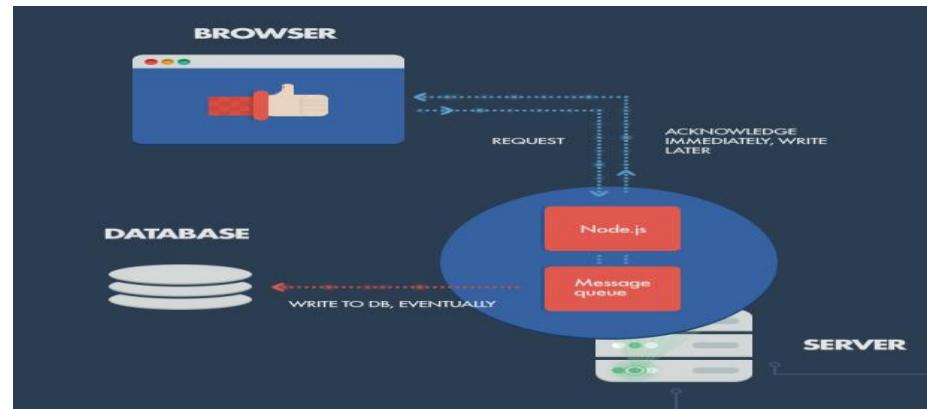
## ➤ API de manejo de una DB de objetos

- Aunque Node.js realmente es excelente con aplicaciones en tiempo real, también es de gran utilidad a la hora de exponer datos de una DB de objetos. Los datos almacenados como JSON permiten que Node.js funcione sin la diferencia de impedancia y la conversión de datos. Por ejemplo, si se usa Rails, convertirás de JSON a modelos binarios y luego volverás a exponerlos como JSON. Con Node.js, simplemente puede exponer sus objetos JSON con una API REST para que el cliente obtenga los datos. Además, no necesita preocuparse por la conversión entre JSON y cualquier otra cosa al leer o escribir desde su base de datos

# Dónde debe utilizarse Node.js

## ➤ Entradas en cola

- Si recibe una gran cantidad de datos concurrentes, su base de datos puede convertirse en un cuello de botella. Node.js puede manejar fácilmente las conexiones simultáneas. Pero como el acceso a la base de datos es una operación de bloqueo (en este caso), nos encontramos con problemas. La solución es reconocer el comportamiento del cliente antes de que los datos se escriban realmente en la base de datos. Con este enfoque, el sistema mantiene su capacidad de respuesta bajo una carga pesada.



- En resumen: con Node.js, puedes empujar los datos a escribirse en la base de datos a un lado y tratar con ellos más tarde, para proceder como si esta hubiera sido exitosa.

# Dónde debe utilizarse Node.js

## ➤ Streaming de datos

- En las plataformas web más tradicionales, las solicitudes HTTP y las respuestas se tratan como un evento aislado; De hecho, en realidad son arroyos. Esta observación se puede utilizar en Node.js para crear algunas características interesantes. Por ejemplo, es posible procesar archivos mientras se siguen cargando, ya que los datos entran a través de un flujo y podemos procesarlo de una manera en línea. Esto podría hacerse para la codificación de audio o video en tiempo real, y para el proxy entre diferentes fuentes de datos (vea la siguiente sección).

# Dónde debe utilizarse Node.js

## ➤ Proxy

- Node.js se emplea fácilmente como un proxy del lado del servidor donde puede manejar una gran cantidad de conexiones simultáneas de una manera no bloqueante. Es especialmente útil para la representación de diferentes servicios con diferentes tiempos de respuesta, o la recopilación de datos de múltiples puntos de origen.

## ➤ Monitoreo de aplicaciones

- Otro caso de uso común en el que Node.js encaja perfectamente es en el seguimiento de los visitantes de un sitio web y la visualización de sus interacciones en tiempo real. Un ejemplo sería: poder reunir estadísticas en tiempo real de su usuario, o incluso moverlo al siguiente nivel introduciendo interacciones específicas con sus visitantes

# Dónde debe utilizarse Node.js

## ➤ Tablero de monitoreo de sistemas

- Ahora, vamos a visitar el lado de la infraestructura de las cosas. Imagine, por ejemplo, un proveedor de SaaS que desee ofrecer a sus usuarios una página de supervisión de servicios (por ejemplo, la página de estado de GitHub). Con el bucle de eventos Node.js, podemos crear un potente panel de control basado en web que comprueba los estados de los servicios de manera asíncrona y empuja los datos a los clientes mediante websockets.

4

## Primera app

# Creación de una app con node.js

- Para crear una aplicación con Node.js debemos tener en cuenta los sientes tres pasos importantes:
  - Importación de módulos necesarios: Utilizamos la directiva “require” para cargar módulos Node.js.
  - Crear servidor: Un servidor que escuchará las solicitudes del cliente, similares a Apache HTTP Server.
  - Leer solicitud y devolver respuesta: El servidor creado leerá la solicitud HTTP hecha por el cliente que puede ser un navegador o una consola y devolver la respuesta.



# Primera aplicación con node.js

# Importar módulo requerido y crear servidor



- Utilizamos la directiva “require” para cargar el módulo http y almacenar la instancia HTTP que éste devuelta en una variable http de la siguiente manera:

```
var http = require("http");
```

- Utilizamos la instancia http creada y llamamos al método http.createServer () para crear una instancia de servidor y luego la enlazamos en el puerto 8081 usando el método listen asociado con la instancia del servidor. Hay que pasar una función con parámetros de solicitud y respuesta

```
http.createServer(function (request, response) {  
    // Envía el encabezado HTTP  
    // HTTP Estado : 200 : OK  
    // Tipo de contexto: text/plain  
    response.writeHead(200, {'Content-Type': 'text/plain'});  
  
    // Enviar respuesta como "Hello World"  
    response.end('Hello World\n');  
}).listen(8081);  
  
// La consola imprimirá el mensaje  
console.log('Server running at http://127.0.0.1:8081/');
```

# Solicitud de pruebas y de respuesta



- Vamos a poner el paso 1 y 2 juntos en un archivo llamado main.js y comenzar nuestro servidor HTTP.
- Ahora ejecute el main.js para iniciar el servidor:

```
$ node main.js
```

- Abra `http://127.0.0.1:8081/` en cualquier navegador y observe el siguiente resultado.



5

# Modules

# Modules

- Node.js puede cargar dependencias utilizando la palabra clave “require”, como se puede ver en el ejemplo:

```
var http = require('http');  
var dns = require('dns');
```

- También podemos cargar archivos relativos:

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

- Puede requerir la instalación de módulos desde npm como lo haría con los nativos, sin necesidad de especificar la ruta absoluta o relativa:

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

# Modules

- Lo bueno de los módulos en Node.js es que no se inyectan automáticamente en el ámbito global, sino que simplemente se asignan a una variable de su elección. Eso significa que usted no tiene que preocuparse por dos o más módulos que tienen funciones con el mismo nombre. Al crear sus propios módulos, todo lo que tiene que hacer es tener cuidado al exportar algo. El primer enfoque sería exportar un solo objeto:

```
var person = { name: 'John', age: 20 };
```

```
module.exports = person;
```

- El segundo enfoque requiere agregar propiedades al objeto de exportación:

```
exports.name = 'John';
```

```
exports.age = 20;
```

# Modules

- Una cosa a tener en cuenta sobre los módulos es que no comparten el alcance, por lo que si desea compartir una variable entre los diferentes módulos, debe incluirlo en un módulo separado que es requerido por los otros módulos. Otra cosa interesante que debe recordar es que los módulos se cargan sólo una vez, y después se almacenan en memoria caché por nodo.
- A diferencia del navegador, Node.js no tiene un objeto global de ventana, sino que tiene otros dos: “globals” y “process”. Sin embargo, se debe evitar seriamente agregar propiedades en los dos.

6

# Callbacks

# Callbacks

- Una función Callbacks se llama al finalizar una tarea determinada. Todas las API de Node se escriben de tal manera que admiten Callbacks.
- Por ejemplo, una función para leer un archivo puede comenzar a leer el archivo y devolver el control al entorno de ejecución inmediatamente para que la siguiente instrucción se pueda ejecutar. Una vez que la E/S del archivo esté completa, llamará a la función Callbacks y ésta pasará el contenido del archivo como un parámetro.

# Ejemplo de código de bloqueo

- Creamos un archivo de texto denominado input.txt con un contenido de preferencia, ejemplo:

Dar contenido de auto aprendizaje.

Para enseñar al mundo de manera simple y fácil !!!!!

- Cree un archivo js llamado main.js con el siguiente código:

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
console.log("Program Ended");
```

- Ahora ejecutamos el main.js para ver el resultado

\$ node main.js

- Verificamos la salida:

Dar contenido de auto aprendizaje.

Para enseñar al mundo de manera simple y fácil !!!!!

Program Ended

# Ejemplo de código sin bloqueo

- Creamos un archivo de texto denominado input.txt con un contenido de preferencia, ejemplo:

Dar contenido de auto aprendizaje.

Para enseñar al mundo de manera simple y fácil !!!!!

- Actualizamos main.js para tener el siguiente código:

```
var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});
console.log("Program Ended");
```

- Ahora ejecutamos el main.js para ver el resultado \$ node main.js

- Verificamos la salida:

Program Ended

Dar contenido de auto aprendizaje.

Para enseñar al mundo de manera simple y fácil !!!!!

# Callbacks

- Estos dos ejemplos explican el concepto de bloqueo y llamadas no bloqueantes.
- El primer ejemplo muestra que el programa se bloquea hasta que lee el archivo y luego sólo continúa para finalizar el programa.
- El segundo ejemplo muestra que el programa no espera la lectura del fichero y procede a imprimir "Program Ended" y al mismo tiempo, el programa sin bloqueo sigue leyendo el fichero.
- Por lo tanto, un programa de bloqueo se ejecuta mucho en secuencia. Desde el punto de vista de programación, es más fácil implementar la lógica pero los programas no bloqueadores no se ejecutan en secuencia. En el caso de que un programa necesite utilizar cualquier dato a ser procesado, debe mantenerse dentro del mismo bloque para hacerlo secuencial.

# Callbacks

- Con este estilo, una función asíncrona invoca un Callbacks para continuar el programa una vez que haya finalizado.
- A continuación se muestra un ejemplo que busca direcciones IPv4 para un dominio:

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

dns.resolve4('www.google.com', function (err, addresses) {
    if (err) throw err;

    console.log('addresses: ' + JSON.stringify(addresses));

});
```

- Hemos pasado la devolución de la llamada, como el segundo argumento a la función asíncrona dns.resolve4. Una vez que la función asíncrona tenga la respuesta lista para nosotros, invocará la callbacks, continuando así la ejecución del programa.

7

# Eventos

# Eventos

- Node.js es una aplicación de un único subprocesso, pero puede admitir la simultaneidad a través del concepto de eventos y callbacks. Cada API de Node.js es asíncrona y de un único subprocesso, utilizan llamadas de función asíncrona para mantener la simultaneidad. El subprocesso de node mantiene un bucle de eventos y cada vez que una tarea se completa, se activa el evento correspondiente que señala la función de eventos a ejecutar.
- Node.js utiliza mucho los eventos y es también una de las razones por las que es bastante rápido en comparación con otras tecnologías similares. Tan pronto como Node inicia su servidor, simplemente inicia sus variables, declara funciones y luego simplemente espera que el evento ocurra.
- En una aplicación controlada por eventos, generalmente hay un bucle principal que escucha eventos y luego activa una función de callbacks cuando se detecta uno de esos eventos.

# Eventos

- Aunque los eventos parecen bastante similares a callbacks, la diferencia reside en el hecho de que las funciones callback se llaman cuando una función asíncrona devuelve su resultado, mientras que el manejo de eventos funciona en el patrón de observador. Las funciones que escuchan los eventos actúan como observadores. Cada vez que un evento se dispara, su función de escucha comienza a ejecutarse. Node.js tiene varios eventos incorporados disponibles a través del módulo de eventos y la clase EventEmitter que se utilizan para vincular eventos y events-listeners de la siguiente manera:

```
//Módulo de eventos de importación
var events = require('events');

//Crear un objeto eventEmitter
var eventEmitter = new events.EventEmitter();
```

# Eventos

- A continuación se muestra la sintaxis para enlazar un controlador de eventos con un evento:

```
/Vincular evento y controlador de eventos  
eventEmitter.on('eventName', eventHandler);
```

- Podemos lanzar un evento programáticamente de la siguiente manera:

```
eventEmitter.emit('eventName');
```

# Eventos

- Cree un archivo js llamado main.js con el siguiente código:

```
// Import events module
var events = require('events');
// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
// Create an event handler as follows
var connectHandler = function connected() {
    console.log('connection succesful.');
// Fire the data_received event
eventEmitter.emit('data_received'); }
// Bind the connection event with the handler
eventEmitter.on('connection', connectHandler);
// Bind the data_received event with the anonymous function
eventEmitter.on('data_received', function(){
    console.log('data received succesfully.'); });
// Fire the connection event
eventEmitter.emit('connection');
console.log("Program Ended.");
```

# Eventos

- Ahora vamos a intentar ejecutar el programa anterior y comprobar su salida:

```
$ node main.js
```

- Esto debe producir el siguiente resultado:

```
connection successful.  
data received successfully.  
Program Ended.
```

# Eventos

- El estándar callbacks funciona bien para los casos de uso en los que queremos ser notificados cuando termine la función asíncrona. Sin embargo, hay situaciones que requieren ser notificado de diferentes eventos que no se producen al mismo tiempo. Veamos un ejemplo de un cliente IRC:

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

var client = new irc.Client('irc.freenode.net', 'myIrcBot', {
  channels: ['#sample-channel'] });
client.on('error', function(message) {
  console.error('error: ', message);
});
client.on('connect', function() {
  console.log('connected to the irc server');
});
client.on('message', function (from, to, message) {
  console.log(from + ' => ' + to + ':' + message);
});
client.on('pm', function (from, message) {
  console.log(from + ' => ME: ' + message);
});
```

# Eventos

- Estamos leyendo diferentes tipos de eventos en el ejemplo anterior:
  - El evento connect se emite cuando el cliente se ha conectado correctamente al servidor IRC.
  - El evento de error se activa en caso de que se produzca un error.
  - Los mensajes y los eventos pm se emiten para los mensajes entrantes.
- Los eventos mencionados anteriormente hacen esta situación ideal para usar el patrón EventEmitter, EventEmitter permite a los implementadores emitir un evento al que los consumidores pueden suscribirse si están interesados. Node tiene una clase EventEmitter en core que podemos usar para crear nuestros propios objetos EventEmitter. Crearemos una clase MemoryWatcher que hereda de EventEmitter y emite dos tipos de eventos:
  - Un evento de datos en un intervalo regular, que representa el uso de memoria en bytes
  - Un evento de error, en caso de que la memoria supere un cierto límite impuesto

# Eventos

- La clase MemoryWatcher se verá de la siguiente manera:

```
var EventEmitter = require('events').EventEmitter;
var util = require('util');

function MemoryWatcher(opts) {
  if (!(this instanceof MemoryWatcher)) {
    return new MemoryWatcher();
  }
  opts = opts || {
    frequency: 30000 // 30 seconds
  };
  EventEmitter.call(this);
  var that = this;
  setInterval(function() {
    var bytes = process.memoryUsage().rss;
    if (opts.maxBytes &gt; bytes &gt; opts.maxBytes) {
      that.emit('error', new Error('Memory exceeded ' + opts.maxBytes + ' bytes'));
    } else {
      that.emit('data', bytes);
    }
  }, opts.frequency);
  util.inherits(MemoryWatcher, EventEmitter);
}

Using it is very simple: <!-- code lang=javascript linenums=true -->
var mem = new MemoryWatcher({
  maxBytes: 12455936, frequency: 5000 });
mem.on('data', function(bytes) { console.log(bytes); })
mem.on('error', function(err) { throw err; })
```

# Eventos

- Una forma más fácil de crear objetos EventEmitter es crear nuevas instancias de la clase EventEmitter sin procesar:

```
var EventEmitter = require('events').EventEmitter; }
var emitter = new EventEmitter();

setInterval(function() {
  console.log(process.memoryUsage().rss);
}, 30000);
```

8

# Streams

# Streams

- Los Streams son objetos que le permiten leer datos de un origen o escribir datos en un destino de manera continua. En Node.js, hay cuatro tipos de streams:
  - Readable: stream que se utiliza para la operación de lectura.
  - Writable: stream que se utiliza para la operación de escritura.
  - Duplex: stream que se puede utilizar para operaciones de lectura y escritura.
  - Transform: Tipo de stream duplex en el que la salida se calcula en función de la entrada.
- Cada tipo de Stream es una instancia de EventEmitter y lanza varios eventos en diferentes instancias de tiempos. Por ejemplo, algunos de los eventos de uso común son:
  - Data: Este evento se activa cuando hay datos disponibles para leer.
  - End: Este evento se activa cuando no hay más datos que leer.
  - Error: Este evento se activa cuando hay algún error al recibir o escribir datos.
  - Finish: Este evento se activa cuando todos los datos han sido enjuagados al sistema subyacente.

# Streams

- Al igual que con los tubos Unix, las corrientes de nodo implementan un operador de composición llamado `.pipe()`. Los principales beneficios del uso de streams son que usted no tiene que almacenar todos los datos en la memoria y son fácilmente componibles.
- Para tener una mejor comprensión de cómo funcionan los streams, crearemos una aplicación que lea un archivo, lo cifre usando el algoritmo AES-256 y luego lo comprima usando gzip. Todo esto utilizando streams, lo que significa que para leer cada fragmento lo cifrará y comprimirá.

# Streams

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

var password = new Buffer(process.env.PASS || 'password');
var encryptStream = crypto.createCipher('aes-256-cbc', password);
var gzip = zlib.createGzip();

var readStream = fs.createReadStream(**filename);    // archivo actual
var writeStream = fs.createWriteStream(**dirname + '/out.gz');

readStream    // Lee el archivo actual
  .pipe(encryptStream)    // Cifra
  .pipe(gzip)    // Comprime
  .pipe(writeStream)    // Escribe en archivo
  .on('finish', function () {    // todo listo
    console.log('done');
});
```

- Aquí tomamos una secuencia legible, la canalizamos en una secuencia de cifrado, luego canalizamos esa en una secuencia de compresión gzip y finalmente la canalizamos a una secuencia de escritura.

# Streams

- Después de ejecutar ese ejemplo deberíamos ver un archivo llamado out.gz. Ahora es el momento de implementar el reverso, que es descifrar el archivo y enviar el contenido al terminal:

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

var password = new Buffer(process.env.PASS || 'password');
var decryptStream = crypto.createDecipher('aes-256-cbc', password);
var gzip = zlib.createGunzip();
var readStream = fs.createReadStream(__dirname + '/out.gz');

readStream // Lee el archivo actual
  .pipe(gzip) // Descomprime
  .pipe(decryptStream) // Descifra
  .pipe(process.stdout) // Escribe en el terminal
  .on('finish', function () { // terminado
    console.log('done');
});
```

9

# Buffers

# Buffers

- Node.js tiene una clase llamada “Buffer” que proporciona instancias para almacenar datos sin procesar similares a una matriz de números enteros.
- La clase buffer es una clase global a la que se puede acceder en una aplicación sin importar el módulo.
- Podemos realizar las siguientes características con los buffers:
  - Se pueden crear mediante el llamado de su clase.
  - Se puede escribir nuevos datos en el buffer ya creada.
  - Podemos leer los datos escritos en él.
  - podemos convertir un buffer en JSON.
  - Se pueden concatenar diferentes buffers



## Hagámoslo en practica

- > Creación de buffers
- > Escribiendo en buffers
- > Leyendo buffers
- > Convertir buffers a JSON
- > Concatenando buffers

# Creación de buffers



- Método 1: A continuación se muestra la sintaxis para crear un buffer no iniciado de 10 octetos:

```
var buf = new Buffer(10);
```

- Método 2: A continuación se muestra la sintaxis para crear un Buffer de una matriz determinada:

```
var buf = new Buffer([10, 20, 30, 40, 50]);
```

- Método 3: A continuación se muestra la sintaxis para crear un Buffer de una cadena dada y opcionalmente el tipo de codificación:

```
var buf = new Buffer("Simply Easy Learning", "utf-8");
```

- Aunque "utf8" es la codificación por defecto, se puede utilizar cualquiera de las siguientes "ascii", "utf8", "utf16le", "ucs2", "base64" o "hex".

# Escribiendo en buffers



- Se puede escribir en un buffer ya creado, a continuación se muestra la sintaxis del método para escribir en un buffer:

```
buf.write(string[, offset][, length][, encoding])
```

- Descripción de los parametros utilizados:
  - String: Esta es la cadena de datos que se escribirá en el buffer.
  - Offset: Este es el índice del buffer para comenzar a escribir en un punto establecido, el valor predeterminado es 0.
  - Length: Este es el número de bytes a escribir.
  - Encoding: Codificación a utilizar. 'Utf8' es la codificación predeterminada.
- Valor del retorno: Este método devuelve el número de octetos escritos. Si no hay suficiente espacio en el buffer para ajustar la cadena entera, escribirá solo una parte de la cadena (hasta donde alcance el espacio).



# Escribiendo en buffers

- A continuación un ejemplo de la utilización de este método:

```
buf = new Buffer(256);
len = buf.write("Simple y fácil de aprender");

console.log(" Octetos escritos: " + len);
```

- Al ejecutar el programa, produce el siguiente resultado:

```
Octets written : 26
```

# Lectura de buffers



- Se puede leer un buffer ya creado, a continuación se muestra la sintaxis del método para leer un buffer:

```
buf.toString([encoding][, start][, end])
```

- Descripción de los parametros utilizados:
  - Encoding: Codificación a utilizar. 'Utf8' es la codificación predeterminada.
  - Start: Índice inicial para empezar a leer, el valor predeterminado es 0.
  - End: El índice final de la lectura, por defecto es el buffer completo.
- Valor del retorno: Este método decodifica y devuelve una cadena de datos lo que esta escrito en el buffer.



# Lectura de buffers

- A continuación un ejemplo de la utilización de este método:

```
buf = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
    buf[i] = i + 97;

console.log( buf.toString('ascii')); // Salida: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5)); // Salida: abcde
console.log( buf.toString('utf8',0,5)); // Salida: abcde
console.log( buf.toString(undefined,0,5)); // Codificación por defecto 'utf8', salida abcde
```

- Al ejecutar el programa, produce el siguiente resultado:

```
abcdefghijklmnpqrstuvwxyz
abcde
abcde
abcde
```

# Convertir buffers a JSON



- A continuación se muestra la sintaxis del método para convertir un buffer en un objeto JSON

```
buf.toJSON()
```

- Valor del retorno: Este método devuelve una representación JSON de la instancia del buffer.
- A continuación un ejemplo de la utilización de este método:

```
var buf = new Buffer('Simple y fácil de aprender ');
var json = buf.toJSON(buf);
console.log(json);
```

- Al ejecutar el programa, produce el siguiente resultado:

```
[ 83, 105, 109, 112, 108, 121, 32, 69, 97, 115, 121, 32, 76, 101, 97, 114, 110, 105, 110, 103 ]
```

- Este resultado es la representación de cada letra en código ASCII.

# Concatenando buffers



- A continuación se muestra la sintaxis del método para concatenar dos buffers a un único buffer.

```
Buffer.concat(list[, totalLength])
```

- Descripción de los parametros utilizados:
  - List: Lista array de objetos de los buffers a concatenar.
  - TotalLength: Ésta es la longitud total de los buffers cuando están concatenados.
- Valor del retorno: Este método devuelve solo una instancia de buffer.



# Concatenando buffers

- A continuación un ejemplo de la utilización de este método:

```
var buffer1 = new Buffer('Tutorial ');
var buffer2 = new Buffer('Simple y fácil de aprender');
var buffer3 = Buffer.concat([buffer1,buffer2]);

console.log("buffer3 contiene: " + buffer3.toString());
```

- Al ejecutar el programa, produce el siguiente resultado:

```
buffer3 contiene: Tutorial Simple y fácil de aprender
```



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123  
08018 Barcelona  
Tel. 93 304.17.20  
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7  
28020 Madrid  
Tel. 91 442.77.03  
Fax. 91 442.77.07

[www.netmind.es](http://www.netmind.es)



GOBIERNO  
DE ESPAÑA

MINISTERIO  
DE ENERGÍA, TURISMO  
Y AGENDA DIGITAL

**red.es**



ESTRATEGIA DE  
EMPRENDIMIENTO Y  
EMPLEO JUVENIL  
*garantía juvenil*



Agenda Digital para España



**UNIÓN EUROPEA**

Fondo Social Europeo  
*“El FSE invierte en tu futuro”*