



M E A N

WEB FULL STACK DEVELOPER

Germán Caballero Rodríguez
germanux@gmail.com



Desarrollo con



INDICE

- 1) Modo estricto
- 2) Node-localstorage
- 3) El proceso de ejecución de NodeJS
- 4) Eventos en NodeJS
- 5) Depuración con NodeJS
- 6) Arquitectura en pipeline
- 7) Buffers
- 8) Streams

Modo estricto

- El modo strict es una manera de introducir una mejor comprobación de errores en el código.
- Cuando se utiliza el modo strict, no se puede, por ejemplo, usar variables declaradas de forma implícita, asignar un valor a una propiedad de solo lectura o agregar una propiedad a un objeto que no es extensible.
- Para obtener más información sobre el modo strict, vea Especificación del lenguaje ECMAScript, Quinta edición.

Modo estricto

- Puedes declarar el modo strict agregando "use strict"; al principio de un archivo, un programa o una función.
- Este tipo de declaración se conoce como prólogo de directivas.
- El ámbito de una declaración de modo strict depende del contexto.
- Si se declara en un contexto global (fuera del ámbito de una función), todo el código del programa está en modo strict.

Modo estricto

- Si se declara en una función, todo el código de la función está en modo strict.
- Por ejemplo, en el siguiente ejemplo todo el código está en modo strict y la declaración de variables fuera de la función produce el error de sintaxis "Variable sin definir en modo strict".

Modo estricto

- Si se declara en una función, todo el código de la función está en modo strict.
- Por ejemplo, en el siguiente ejemplo todo el código está en modo strict y la declaración de variables fuera de la función produce el error de sintaxis "Variable sin definir en modo strict".

```
"use strict";  
function testFunction(){  
    var testvar = 4;  
    return testvar;  
}  
  
// This causes a syntax error.  
testvar = 5;
```

Modo estricto

- En el ejemplo siguiente, solo el código contenido en testFunction está en modo strict.
- La declaración de variables fuera de la función no produce un error de sintaxis, pero la declaración dentro de la función, sí.

```
function testFunction(){  
    "use strict";  
    // This causes a syntax error.  
    testvar = 4;  
    return testvar;  
}  
testvar = 5;
```


Modo estricto

- En la tabla siguiente se indican las restricciones más importantes que se aplican en el modo strict.

Elemento del lenguaje	Restricción	Error	Ejemplo
Variable	Usar una variable sin declararla.	SCRIPT5042: Variable sin definir en modo strict	<div>JavaScript</div> <pre>testvar = 4;</pre>
Propiedad de solo lectura	Escribir en una propiedad de solo lectura.	SCRIPT5045: No se permite la asignación a propiedades de solo lectura en modo strict	<div>JavaScript</div> <pre>var testObj = Object.defineProperties({}, { prop1: { value: 10, writable: false // by default }, prop2: { get: function () { } } }); testObj.prop1 = 20; testObj.prop2 = 30;</pre>

Modo estricto

Elemento del lenguaje	Restricción	Error	Ejemplo
Propiedad no extensible	Agregar una propiedad a un objeto cuyo atributo extensible está establecido en false .	SCRIPT5046: No se puede crear ninguna propiedad para un objeto no extensible	<div>JavaScript</div> <pre>var testObj = new Object(); Object.preventExtensions(testObj); testObj.name = "Bob";</pre>
delete	Eliminar una variable, una función o un argumento. Eliminar una propiedad cuyo atributo configurable está establecido en false .	SCRIPT1045: No se permite llamar a delete en una <expresión> en modo strict	<div>JavaScript</div> <pre>var testvar = 15; function testFunc() {}; delete testvar; delete testFunc; Object.defineProperty(testObj, "testvar", { value: 10, configurable: false }); delete testObj.testvar;</pre>

Modo estricto

Elemento del lenguaje	Restricción	Error	Ejemplo
Duplicar una propiedad	Definir una propiedad más de una vez en un literal de objeto.	SCRIPT1046: No se permiten varias definiciones de una propiedad en modo strict	<div>JavaScript</div> <pre>var testObj = { prop1: 10, prop2: 15, prop1: 20 };</pre>
Duplicar un nombre de parámetro	Usar un nombre de parámetro más de una vez en una función.	SCRIPT1038: No se permiten nombres de parámetros formales duplicados en modo strict	<div>JavaScript</div> <pre>function testFunc(param1, param1) { return 1; };</pre>

Elemento del lenguaje	Restricción	Error	Ejemplo
this	El valor de this no se convierte al objeto global cuando es null o undefined .		<div>JavaScript</div> <pre>function testFunc() { return this; } var testvar = testFunc();</pre> <p>En modo no strict, el valor de testvar es el objeto global, pero en modo strict su valor es undefined.</p>
eval como identificador	La cadena "eval" no se puede usar como identificador (nombre de variable o función, nombre de parámetro, etc.).		<div>JavaScript</div> <pre>var eval = 10;</pre>
Función declarada dentro de una instrucción o un bloque	No se puede declarar una función dentro de una instrucción o un bloque.	SCRIPT1047: En modo strict, las declaraciones de función no pueden estar anidadas dentro de una instrucción o	<div>JScript</div> <pre>var arr = [1, 2, 3, 4, 5]; var index = null; for (index in arr) { function myFunc() {}; }</pre>

node-localstorage

- Un sustituto de la API Storage nativa del navegador que se ejecuta en node.js.
- Instalación:
 - npm install node-localstorage
- Uso:

JavaScript

```
if (typeof localStorage === "undefined" || localStorage === null) {  
  var LocalStorage = require('node-localstorage').LocalStorage;  
  localStorage = new LocalStorage('./scratch');  
}  
  
localStorage.setItem('myFirstKey', 'myFirstValue');  
console.log(localStorage.getItem('myFirstKey'));
```

El proceso de ejecución de NodeJS

- La naturaleza de NodeJS es la de un lenguaje de programación de propósito general, con el que podemos hacer todo tipo de aplicaciones.
- Normalmente comenzaremos con programas que se ejecutarán en la consola de comandos, o terminal, de tu sistema operativo, pero a partir de ahí sus aplicaciones son muy extensas.

El proceso de ejecución de NodeJS

- Una de las características de NodeJS es su naturaleza "Single Thread".
- Cuando pones en marcha un programa escrito en NodeJS se dispone de un único hilo de ejecución.
- Esto, en contraposición con otros lenguajes de programación como Java, PHP o Ruby, donde cada solicitud se atiende en un nuevo proceso, tiene sus ventajas.
- Una de ellas es que permite atender mayor demanda con menos recursos, uno de los puntos a favor de NodeJS.

El proceso de ejecución de NodeJS

Objeto process

- El objeto process es una variable global disponible en NodeJS que nos ofrece diversas informaciones y utilidades acerca del proceso que está ejecutando un script Node.
- Contiene diversos métodos, eventos y propiedades que nos sirven no solo para obtener datos del proceso actual, sino también para controlarlo.

<https://nodejs.org/docs/latest/api/process.html>

El proceso de ejecución de NodeJS

Objeto process

- Se pueden hacer muchas cosas con el proceso y consultar diversos datos de utilidad.

```
console.log('id del proceso: ', process.pid);  
console.log('título del proceso: ', process.title);  
console.log('versión de node: ', process.version);  
console.log('sistema operativo: ', process.platform);
```

El proceso de ejecución de NodeJS

Objeto process

- Salir de la ejecución de un programa Node:

```
process.exit();
```

- El comportamiento normal de un programa será salir automáticamente cuando haya terminado su trabajo.
- Técnicamente esto quiere decir, que se haya terminado la secuencia de ejecución de instrucciones de un script y que no haya trabajo pendiente en el "event loop".

El proceso de ejecución de NodeJS

Objeto process

- Básicamente provocará que el programa acabe, incluso en el caso que haya operaciones asíncronas que no se hayan completado o que se esté escuchando eventos diversos en el programa.
- El método `exit` puede recibir opcionalmente un código de salida. Si no indicamos nada se entiende "0" como código de salida.

```
process.exit(3);
```

El proceso de ejecución de NodeJS

Objeto process

Evento exit

- Otra de las cosas básicas que podrás hacer mediante el objeto process es definir eventos cuando ocurran cosas diversas, por ejemplo la salida del programa.

El proceso de ejecución de NodeJS

Objeto process

Evento exit

- Los eventos en Javascript se definen de manera estándar mediante el método `on()`, indicando el tipo de evento que queremos escuchar y una función callback que se ejecutará cuando ese evento se dispare.
- Dado que el evento `exit` pertenece al `process`, lo definiremos a partir de él.

El proceso de ejecución de NodeJS

Objeto process

Evento exit

```
process.on('exit', function(codigo) {  
    console.log('saliendo del proceso con código  
de salida', codigo);  
})
```

- En este caso la función callback asociada al evento exit recibe aquel código de error que se puede generar mediante la invocación del método exit() que conocimos en el anterior punto.

El proceso de ejecución de NodeJS

Objeto process

Process Events

- Event: 'beforeExit'
- Event: 'disconnect'
- Event: 'exit'
- Event: 'message'
- Event: 'rejectionHandled'
- Event: 'uncaughtException'
 - Warning: Using 'uncaughtException' correctly
- Event: 'unhandledRejection'
- Event: 'warning'
 - Emitting custom warnings
- Signal Events

El proceso de ejecución de NodeJS

`process.abort()`
`process.arch`
`process.argv`
`process.argv0`
`process.channel`
`process.chdir(directory)`
`process.config`
`process.connected`
`process.cpuUsage([previousValue])`
`process.cwd()`
`process.disconnect()`
`process.env`
`process.emitWarning(warning[, name][, ctor])`

- Avoiding duplicate warnings

`process.execArgv`
`process.execPath`

`process.exit([code])`
`process.exitCode`
`process.getegid()`
`process.geteuid()`
`process.getgid()`
`process.getgroups()`
`process.getuid()`
`process.hrtime([time])`
`process.initgroups(user, extra_group)`
`process.kill(pid[, signal])`
`process.mainModule`
`process.memoryUsage()`
`process.nextTick(callback[, ...args])`
`process.pid`
`process.platform`
`process.release`
`process.send(message[, sendHandle[, options]][, callback])`
`process.setegid(id)`
`process.seteuid(id)`
`process.setgid(id)`
`process.setgroups(groups)`

<https://nodejs.org/docs/latest/api/process.html>

Eventos en NodeJS

- Los eventos que se producen en el servidor pueden ser de diversos tipos dependiendo de las librerías o clases que estemos trabajando.
- Para hacernos una idea más exacta, pensemos por ejemplo en un servidor HTTP, donde tendríamos el evento de recibir una solicitud.
- Por poner otro ejemplo, en un stream de datos tendríamos un evento cuando se recibe un dato como una parte del flujo.

Eventos en NodeJS

Módulo de eventos

- Los eventos se encuentran en un módulo independiente que tenemos que requerir en nuestros programas creados con Node JS.
- Lo hacemos con la sentencia "require" que conocimos en artículos anteriores cuando hablábamos de módulos.

```
var eventos = require('events');
```

Eventos en NodeJS

Módulo de eventos

- Dentro de esta librería o módulo tienes una serie de utilidades para trabajar con eventos.
- Veamos primero el emisor de eventos, que encuentras en la propiedad EventEmitter.

```
var EmisorEventos = eventos.EventEmitter;
```

- Ese "EmisorEventos" es una clase de programación orientada a objetos (POO), por eso se le ha puesto en el nombre de la clase la primera letra en mayúscula.
- Por convención se hace así con los nombres de las clases en POO.

Eventos en NodeJS

Cómo definir un evento en NodeJS

- En "Node" existe un bucle de eventos, de modo que cuando tú declaras un evento, el sistema se queda escuchando en el momento que se produce, para ejecutar entonces una función.
- Esa función se conoce como "callback" o como "manejador de eventos" y contiene el código que quieres que se ejecute en el momento que se produzca el evento al que la hemos asociado.

Eventos en NodeJS

Cómo definir un evento en NodeJS

- Primero tendremos que "instanciar" un objeto de la clase EventEmitter, que hemos guardado en la variable EmisorEventos en el punto anterior de este artículo.

```
var ee = new EmisorEventos();
```

Eventos en NodeJS

Cómo definir un evento en NodeJS

- Luego tendremos que usar el método `on()` para definir las funciones manejadoras de eventos, o su equivalente `addEventListener()`.
- Para emitir un evento mediante código Javascript usamos el método `emit()`.

Eventos en NodeJS

Cómo definir un evento en NodeJS

- Por ejemplo, voy a emitir un evento llamado "datos", con este código.

- ```
ee.emit('datos', Date.now());
```
- 

- Ahora voy a hacer una función manejadora de eventos que se asocie al evento definido en "datos".

```
ee.on('datos', function(fecha){
 console.log(fecha);
});
```

# Eventos en NodeJS

## Cómo definir un evento en NodeJS

- Si deseamos aprovechar algunas de las características más interesantes de aplicaciones NodeJS quizás nos venga bien usar `setInterval()` y así podremos estar emitiendo datos cada cierto tiempo:

```
setInterval(function(){
 ee.emit('datos', Date.now());
}, 500);
```



# Eventos en NodeJS

## Código completo del ejemplo de eventos y ejecución

- Con esto ya habremos construido un ejemplo NodeJS totalmente funcional.
- El código completo sería el siguiente:

```
var eventos = require('events');

var EmisorEventos = eventos.EventEmitter;
var ee = new EmisorEventos();
ee.on('datos', function(fecha){
 console.log(fecha);
});
setInterval(function(){
 ee.emit('datos', Date.now());
}, 500);
```

<http://www.desarrolloweb.com/articulos/eventos-nodejs.html>

# Depuración en NodeJS

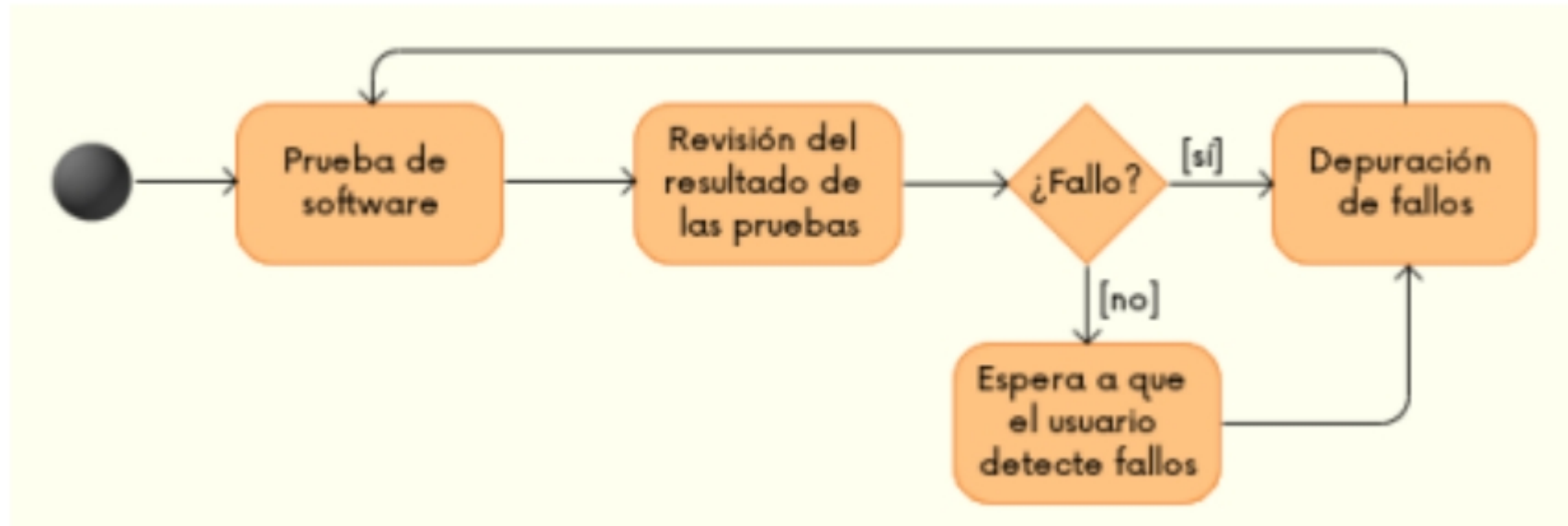
- La depuración es algo que todo usuario debe conocer para desenvolverse con soltura en Node y así resolver problemas de programación fácil y rápidamente.
- No hay una buena base de Node sin conocer un depurador con el que encontrar fallos en el software.
- Formalmente, la depuración (debugging) es el proceso mediante el cual se corrige uno o más defectos o fallos en el software.

# Depuración en NodeJS

- Por un lado, se busca la causa del problema, es decir, por qué se está dando el error y, una vez encontrado, se resuelve o corrige.
- Una vez finalizada la depuración, se vuelve a someter el software a la batería de pruebas para validar que realmente se ha resuelto el fallo sin añadir ninguno nuevo, es decir, que la corrección del defecto no tiene efectos colaterales en otras partes del software que no los presentaba antes.

# Depuración en NodeJS

- A continuación, se muestra un diagrama de estado que muestra el ciclo de vida del software, en cuanto a fallos se refiere:



# Depuración en NodeJS

- Tal como se puede observar, nunca se da por finalizada la prueba, más pronto o más tarde, aparecerá un problema.
- Como buenos desarrolladores, nuestro objetivo es intentar que llegue el menor número de defectos al usuario, para que así no nos saquen los colores y nuestra organización o el cliente no se vea perjudicada por un fallo importante

# Depuración en NodeJS

- Tal como se puede observar, nunca se da por finalizada la prueba, más pronto o más tarde, aparecerá un problema.
- Como buenos desarrolladores, nuestro objetivo es intentar que llegue el menor número de defectos al usuario, para que así no nos saquen los colores y nuestra organización o el cliente no se vea perjudicada por un fallo importante

# Depuración en NodeJS

Los depuradores principalmente proporcionan las siguientes características:

- Ejecución paso a paso (stepping).
  - Permite ejecutar una proposición cada vez con objeto de ver cómo quedan las cosas tras la ejecución de la proposición.
- Puntos de interrupción (breakpoints).
  - Permite indicar puntos en los que el depurador detendrá la ejecución.
  - En ese punto, podremos comprobar los valores de las variables, decidir seguir ejecutando paso a paso o hasta el siguiente punto de interrupción, comprobar la pila de llamadas o finalizar la ejecución.

# Depuración en NodeJS

Los depuradores principalmente proporcionan las siguientes características:

- Observadores (watchers).
  - Permite visualizar el valor de una determinada variable o expresión.
- Consola de consulta (drawer).
  - Permite ejecutar expresiones de JavaScript usando como variables el contexto actual de ejecución.
- Pila de llamadas (call stack).
  - Permite visualizar qué funciones hay abiertas en el punto bajo depuración.



# Depuración en NodeJS

## Depuración 1: con Supervisor

- node-inspector, supervisor y forever.
- npm install -g node-inspector supervisor forever
  - **Node-inspector** será el que nos permita depurar nuestra aplicación
  - **Supervisor** nos evita tener que tumbar y volver a levantar nuestro servidor cada vez que hagamos un cambio,
  - y **forever**, nos va a permitir dejar ejecutando node-inspector en background por tiempo indefinido.

# Depuración en NodeJS

## Depuración 1: con Supervisor

- Para que node inspector pueda depurar, debemos ejecutar supervisor, que se encargará de levantar el servidor, y forever que se encargará de ejecutar el depurador indefinidamente.
- Ejecutar en 2 ventanas distintas de la terminal.
  - `supervisor --debug server.js`
  - `forever /usr/local/bin/node-inspector --web-port=9999`
    - Veremos que forever nos da un mensaje como el siguiente:
    - Visit <http://127.0.0.1:9999/debug?port=5858> to start debugging.
    - `debugger;` es una instrucción JS para declarar breakpoint

# Depuración en NodeJS

## Depuración 2: opción --inspect

- Instalamos node-inspector
- Node Inspector (npm: node-inspector) es un interfaz de depuración para nodejs para poder utilizar el inspector de Chrome.
- Como lo utilizaremos para depurar cualquier aplicación de nodejs, lo instalamos de manera global:
  - `npm install -g node-inspector`
- Para arrancar node-inspector y dejarlo en segundo plano ejecutamos:
  - `node-inspector &`
- En la consola veremos algo como:
  - Info - socket.io started
  - visit `http://0.0.0.0:8080/debug?port=5858` to start debugging

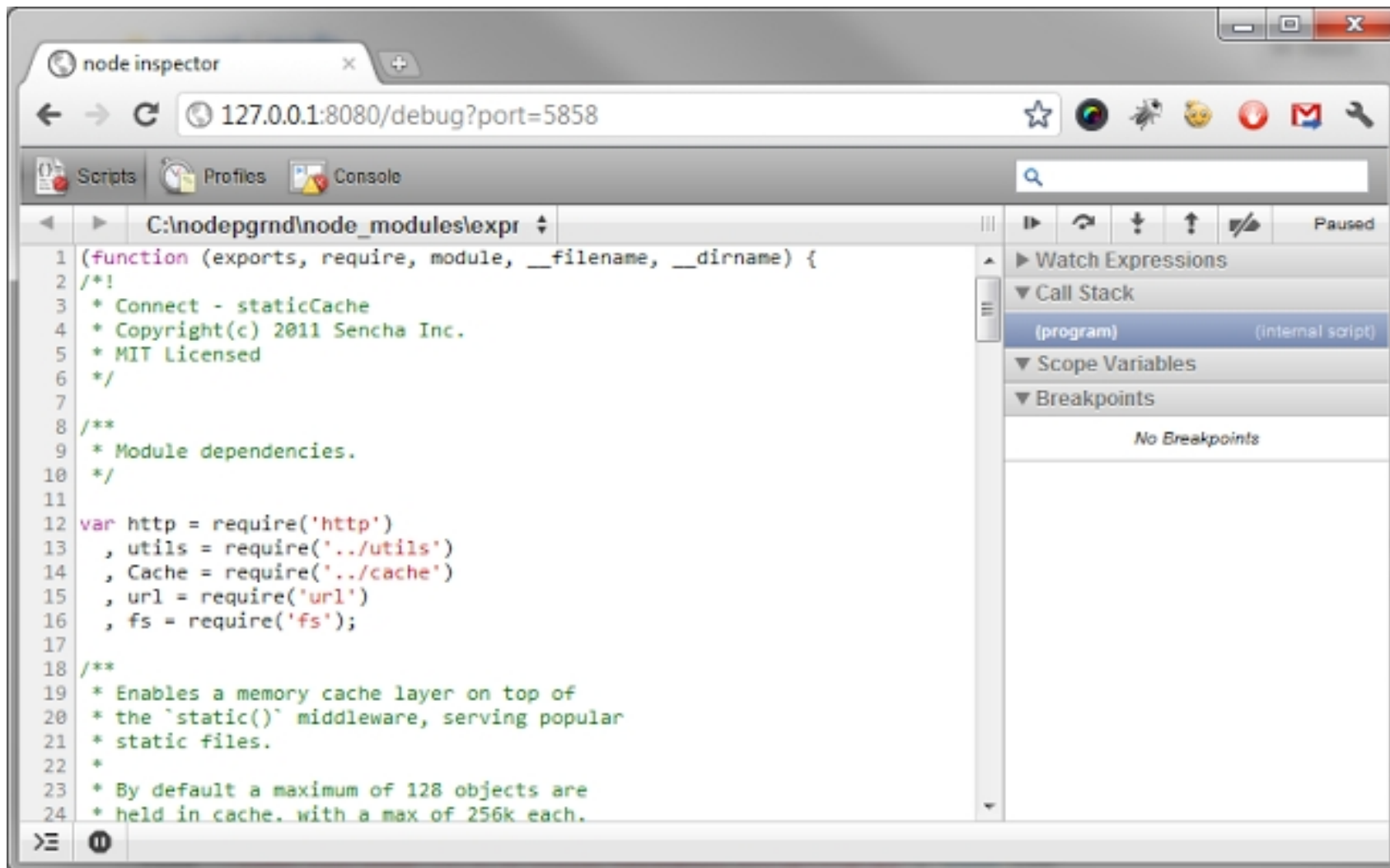
# Depuración en NodeJS

## Depuración 2: opción --inspect

- Arrancamos el servidor en modo depuración:
- `node --debug server.js`
- Con Google Chrome accedemos a `http://0.0.0.0:8080/debug?port=5858` ( `http://127.0.0.1:8080/debug?port=5858` ).
- Y veremos la consola:

# Depuración en NodeJS

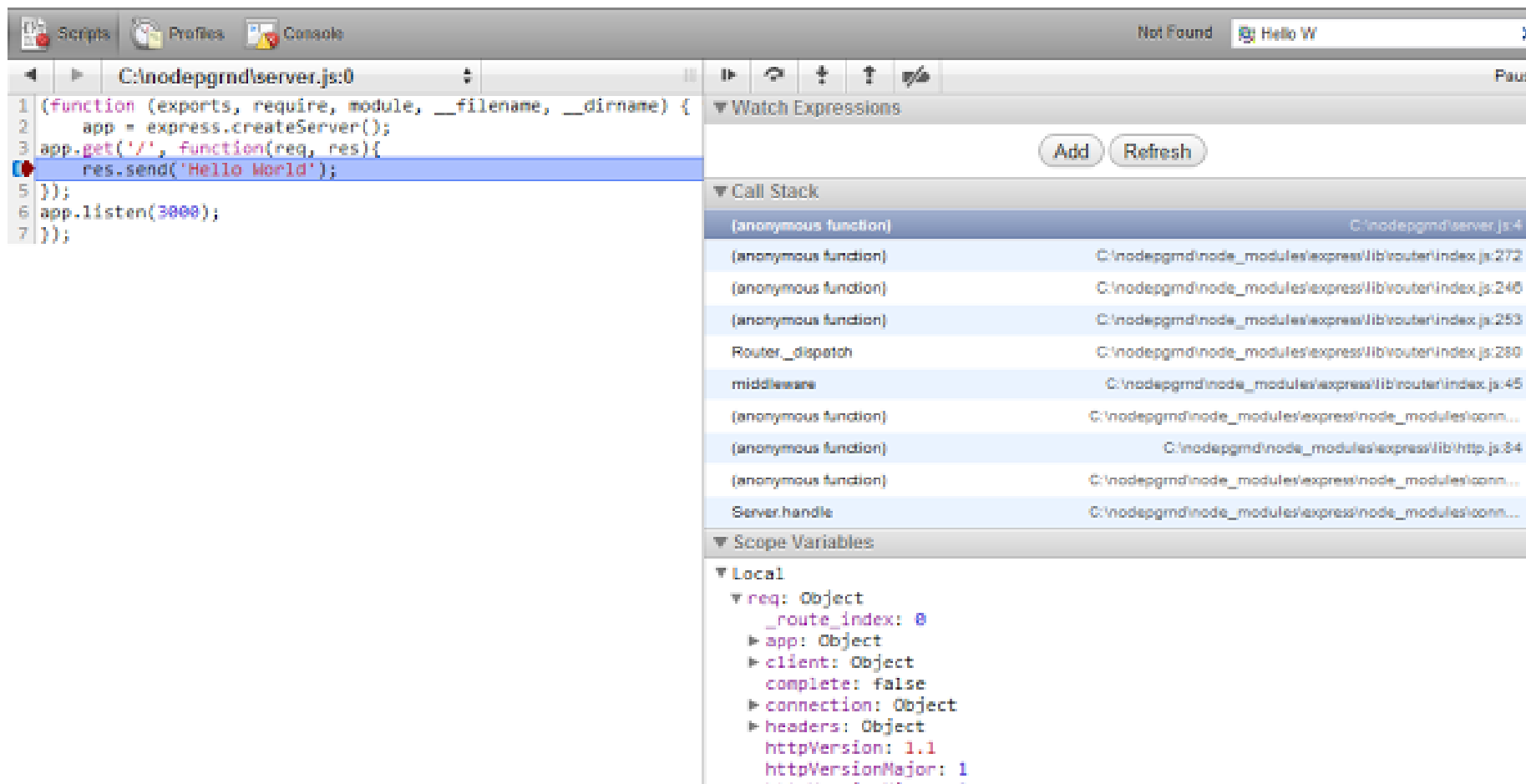
## Depuración 2: opción --inspect



# Depuración en NodeJS

## Depuración 2: opción --inspect

- Desde aquí podemos poner puntos de interrupción, evaluar expresiones, interactuar con la consola, ver el stack de llamadas, etc.



The screenshot displays the Node.js DevTools interface. The top bar shows 'Scripts', 'Profiles', and 'Console' tabs. The 'Scripts' tab is active, showing a file named 'C:\nodepgmd\server.js:0'. The script content is as follows:

```
1 (function (exports, require, module, __filename, __dirname) {
2 app = express.createServer();
3 app.get('/', function(req, res){
4 res.send('Hello World');
5 });
6 app.listen(3000);
7 });
```

A red breakpoint is set on line 4. The right-hand pane shows the 'Call Stack' with the following entries:

- (anonymous function) C:\nodepgmd\server.js:4
- (anonymous function) C:\nodepgmd\node\_modules\express\lib\router\index.js:272
- (anonymous function) C:\nodepgmd\node\_modules\express\lib\router\index.js:246
- (anonymous function) C:\nodepgmd\node\_modules\express\lib\router\index.js:253
- Router.dispatch C:\nodepgmd\node\_modules\express\lib\router\index.js:280
- middleware C:\nodepgmd\node\_modules\express\lib\router\index.js:45
- (anonymous function) C:\nodepgmd\node\_modules\express\node\_modules\conn...
- (anonymous function) C:\nodepgmd\node\_modules\express\lib\http.js:84
- (anonymous function) C:\nodepgmd\node\_modules\express\node\_modules\conn...
- Server.handle C:\nodepgmd\node\_modules\express\node\_modules\conn...

The 'Scope Variables' pane shows the local scope with the following variables:

- req: Object
  - \_route\_index: 0
  - app: Object
  - client: Object
  - complete: false
  - connection: Object
  - headers: Object
  - httpVersion: 1.1
  - httpVersionMajor: 1

# Arquitectura en pipeline

- La arquitectura en pipeline (basada en filtros) consiste en ir transformando un flujo de datos en un proceso comprendido por varias fases secuenciales, siendo la entrada de cada una la salida de la anterior.
- Esta arquitectura es muy común en el desarrollo de programas para el intérprete de comandos, ya que se pueden conectar comandos fácilmente con tuberías (pipe).

# Arquitectura en pipeline

- También es una arquitectura muy natural en el paradigma de programación funcional, ya que equivale a la composición de funciones matemáticas.
- Una tubería (pipe, cauce o '|') consiste en una cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo.
- Permiten la comunicación y sincronización entre procesos.
- Es común el uso de buffer de datos entre elementos consecutivos.



# Arquitectura en pipeline

- La comunicación por medio de tuberías se basa en la interacción productor/consumidor, los procesos productores (aquellos que envían datos) se comunican con los procesos consumidores (que reciben datos) siguiendo un orden FIFO.
- Una vez que el proceso consumidor recibe un dato, éste se elimina de la tubería.

# Arquitectura en pipeline

- Las tuberías (pipes) están implementadas en forma muy eficiente en los sistemas operativos multitarea:
  - Iniciando todos los procesos al mismo tiempo
  - Y atendiendo automáticamente los requerimientos de lectura de datos para cada proceso cuando los datos son escritos por el proceso anterior.
- De esta manera el planificador de corto plazo va a dar el uso de la CPU a cada proceso a medida que pueda ejecutarse minimizando los tiempos muertos.

# Arquitectura en pipeline

- Para mejorar el rendimiento, la mayoría de los sistemas operativos implementan las tuberías usando buffers, lo que permite al proceso proveedor generar más datos que lo que el proceso consumidor puede atender inmediatamente.
- Podemos distinguir dos tipos de tuberías:

# Arquitectura en pipeline

Podemos distinguir dos tipos de tuberías:

- **Tubería sin nombre:**

- Las tuberías sin nombre tienen asociado un fichero en memoria principal, por lo tanto, son temporales y se eliminan cuando no están siendo usados ni por productores ni por consumidores.
- Permiten la comunicación entre el proceso que crea un cauce y procesos hijos tras la creación de la tubería.

# Arquitectura en pipeline

Podemos distinguir dos tipos de tuberías:

- **Tubería con nombre**

- Su diferencia respecto a las tuberías sin nombre radica en que el cauce se crea en el sistema de archivos, y por lo tanto no tienen carácter temporal.
- Se manejan mediante llamadas al sistema (open, close, read y write) como el resto de ficheros del sistema.
- Permiten la comunicación entre los procesos que usen dicha tubería, aunque no exista una conexión jerárquica entre ellos.

# Buffer en NodeJS

- Los buffer son conjuntos de datos en crudo, datos binarios, que podemos tratar en NodeJS para realizar diversos tipos de acciones.
- Los implementa Node a través de una clase específica llamada Buffer, que era necesaria porque Javascript tradicionalmente no era capaz de trabajar con tipos de datos binarios.
- Los buffer son similares a arrays de enteros, en los que tenemos bytes que corresponden con datos en crudo de posiciones de memoria fuera de la pila de V8.

# Buffer en NodeJS

- Aunque desde Javascript ES6 con los **TypedArray** ya es posible trabajar con un buffer de datos binarios, NodeJS mantiene su clase Buffer que realiza un tratamiento de la información más optimizado para sus casos de uso.
  - Un objeto **TypedArray** describe una vista tipo matriz de un búfer de datos binario subyacente.
  - No hay ninguna propiedad global denominada TypedArray, ni hay un constructor de TypedArray directamente visible.

```
new TypedArray(length);
new TypedArray(typedArray);
new TypedArray(object);
new TypedArray(buffer [, byteOffset [, length]]);
```

# Buffer en NodeJS

- Entre los casos de uso más populares de la plataforma NodeJS se encuentran muchos en los que los buffer son verdaderos protagonistas, como por ejemplo
  - En la comunicación bidireccional que tenemos cuando manejamos sockets
  - También al manipular imágenes
  - O streams de datos.



# Buffer en NodeJS

## Crear un Buffer

- Para trabajar con buffers debemos apoyarnos en la clase Buffer, que es global en NodeJS, por lo que no necesitas hacer el require de ningún módulo para poder usarla.
- Ten en cuenta que el tamaño de un buffer se establece en el momento de su creación y luego ya no es posible cambiarlo.

# Buffer en NodeJS

## Crear un Buffer

- Podríamos crear un buffer de diversas maneras, la más sencilla es la siguiente:

```
var b1 = Buffer.alloc(20);
```

- Esto crea un buffer con tamaño de 20 bytes, en el que no hemos definido su contenido, de modo que cada uno de sus bytes estará inicializado a cero.

# Buffer en NodeJS

## Crear un Buffer

- Si queremos crear un buffer incluyendo ya su contenido podemos usar otros métodos de la clase Buffer, como from().
- El método from() permite crear un nuevo buffer en el que contendremos una cadena de caracteres.
- Además podemos asignar una codificación, siendo "utf8" la codificación predeterminada.

```
var b2 = Buffer.from('Mañana más');
```

# Buffer en NodeJS

## Escribir un buffer por pantalla

- El buffer es una cadena de bytes, en binario, por lo que si lo mostramos por pantalla obtendremos una secuencia de valores que inicialmente pueden no tener mucho sentido

# Buffer en NodeJS

## Escribir un buffer por pantalla

- Por ejemplo con los buffer b1 y b2 creados en el punto anterior, podríamos imprimirlos en pantalla con este código.

```
console.log('Este es mi buffer inicializado a cero');
console.log(b1);
console.log('-----');
console.log('Este es mi buffer creado con un string');
console.log(b2);
```

```
Este es mi buffer inicializado a cero
<Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00>

Este es mi buffer creado con un string
<Buffer 4d 61 c3 b1 61 6e 61 20_6d c3 a1 73>
```

# Buffer en NodeJS

## Escribir un buffer por pantalla

- Ahora bien, si lo que queremos es ver la cadena tal cual y no los datos en binario, entonces podemos usar el método `toString()` del buffer.

```
console.log(b2.toString());
```

# Buffer en NodeJS

## El buffer como un array de enteros

- Cada una de las posiciones del buffer se acceden con un índice numérico, comenzando con el índice cero, como los arrays en general en Javascript.

```
console.log(b2[0]);
```

- Obtenemos el valor 77, que en hexadecimal corresponde con el valor 4d y que a su vez es equivalente a la "M".

# Buffer en NodeJS

## El buffer como un array de enteros

- Podríamos modificar de manera arbitraria varias posiciones de nuestro buffer

```
b2[0] = 46;
b2[2] = 54;
b2[3] = 55;
b2[4] = 77;
console.log(b2.toString())
```



# Buffer en NodeJS

## Un poco más

Class Method: Buffer.alloc(size[, fill[, encoding]])

Class Method: Buffer.allocUnsafe(size)

Class Method: Buffer.allocUnsafeSlow(size)

Class Method: Buffer.byteLength(string[, encoding])

Class Method: Buffer.compare(buf1, buf2)

Class Method: Buffer.concat(list[, totalLength])

Class Method: Buffer.from(array)

Class Method: Buffer.from(arrayBuffer[, byteOffset[, length]])

Class Method: Buffer.from(buffer)

Class Method: Buffer.from(string[, encoding])

Class Method: Buffer.isBuffer(obj)

Class Method: Buffer.isEncoding(encoding)

Class Property: Buffer.poolSize

buf[index]

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

# Buffer en NodeJS

## Un poco más

- `buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])`
- `buf.entries()`
- `buf.equals(otherBuffer)`
- `buf.fill(value[, offset[, end]][, encoding])`
- `buf.indexOf(value[, byteOffset][, encoding])`
- `buf.includes(value[, byteOffset][, encoding])`
- `buf.keys()`
- `buf.lastIndexOf(value[, byteOffset][, encoding])`
- `buf.length`
- `buf.readDoubleBE(offset[, noAssert])`
- `buf.readDoubleLE(offset[, noAssert])`
- `buf.readFloatBE(offset[, noAssert])`
- `buf.readFloatLE(offset[, noAssert])`
- `buf.readInt8(offset[, noAssert])`
- `buf.readInt16BE(offset[, noAssert])`
- `buf.readInt16LE(offset[, noAssert])`
- `buf.readInt32BE(offset[, noAssert])`
- `buf.readInt32LE(offset[, noAssert])`
- `buf.readIntBE(offset, byteLength[, noAssert])`
- `buf.readIntLE(offset, byteLength[, noAssert])`

- `buf.readUIntBE(offset, byteLength[, noAssert])`
- `buf.readUIntLE(offset, byteLength[, noAssert])`
- `buf.slice([start[, end]])`
- `buf.swap16()`
- `buf.swap32()`
- `buf.swap64()`
- `buf.toString([encoding[, start[, end]]])`
- `buf.toJSON()`
- `buf.values()`
- `buf.write(string[, offset[, length]][, encoding])`
- `buf.writeDoubleBE(value, offset[, noAssert])`
- `buf.writeDoubleLE(value, offset[, noAssert])`
- `buf.writeFloatBE(value, offset[, noAssert])`
- `buf.writeFloatLE(value, offset[, noAssert])`
- `buf.writeInt8(value, offset[, noAssert])`
- `buf.writeInt16BE(value, offset[, noAssert])`
- `buf.writeInt16LE(value, offset[, noAssert])`
- `buf.writeInt32BE(value, offset[, noAssert])`
- `buf.writeInt32LE(value, offset[, noAssert])`
- `buf.writeIntBE(value, offset, byteLength[, noAssert])`

<https://nodejs.org/api/buffer.html>

# Streams en NodeJS

- En general son flujos de información, o chorros de información que usamos en la transmisión de datos binarios.
- El flujo de información que forma un stream se transmite en pedazos, conocidos habitualmente con su término en inglés "chunk".
- Los chunk no son más que objetos de la clase Buffer.

# Streams en NodeJS

- Los streams los vamos a recibir desde muchas fuentes de información como la manipulación de imágenes, las request del protocolo HTTP, de ficheros o los sockets.
- Existen tres tipos de streams, según el uso que queramos realizar mediante estos flujos de datos.
- Los tenemos de lectura, escritura y duplex (que permiten ambas operaciones a la vez).

# Streams en NodeJS

- El método `createReadStream()` del objeto `fs` nos devuelve un stream a cambio de la ruta del archivo que pretendemos leer.
- `var streamLectura = fs.createReadStream('./archivo-texto.txt');`
- Al usar `createReadStream()` recibiremos un stream de lectura.
- Con él podremos hacer todas las cosas que se encuentran disponibles en el API de NodeJS para los streams.

# Streams en NodeJS

## Eventos en streams

- Existen varios eventos que podemos usar con los streams, para realizar acciones cuando ocurran ciertos sucesos.
- Los eventos disponibles dependen del tipo de stream que tenemos con nosotros.
- Por ejemplo, para streams de lectura tenemos los eventos "close", "data", "end", "error", "readable".

# Streams en NodeJS

## Eventos en streams:

- El evento "data" ocurre cada vez que se reciben datos desde un stream, lo que ocurre al invocar diversos métodos del objeto stream.
- Además, al crearse un manejador de evento para el evento "data", comienza también la lectura del stream.

# Streams en NodeJS

## Eventos en streams:

- Cuando el dato se haya leído y se encuentre disponible se ejecutará el propio manejador de evento asociado, recibiendo como parámetro un buffer de datos.

```
streamLectura.on('data', (chunk) => {
 //chunk es un buffer de datos
 console.log(chunk instanceof Buffer); //escribe "true"
});
```



# Streams en NodeJS

## Eventos en streams:

- Podríamos ver su contenido de esta manera:

```
streamLectura.on('data', (chunk) => {
 console.log('He recibido ' + chunk.length + ' bytes de datos.');
```

```
 console.log(chunk.toString());
});
```

# Streams en NodeJS

## Stream de escritura `process.stdout`

- Para hacer alguna cosa con streams de tipo de escritura vamos a basarnos en una propiedad del objeto global `process` de Node, llamada `stdout`.
- No es más que un stream de escritura con el que podemos generar salida en nuestro programa.
- La salida de `stdout` es la salida estándar de NodeJS, la propia consola.

# Streams en NodeJS

## Stream de escritura process.stdout

- Mediante el método `write()` se escribe en un stream de escritura.
- Para ello tenemos que enviarle un buffer y otra serie de parámetros opcionales como el tipo de codificación y una función callback a ejecutar cuando termine la operación de escritura.

```
process.stdout.write(objetoBuffer);
```

# Streams en NodeJS

## Stream de escritura process.stdout

- Nuestro manejador de evento anterior podría apoyarse en este método `write()` para conseguir la escritura, en lugar de usar el `console.log()`.

```
streamLectura.on('data', (chunk) => {
 process.stdout.write(chunk)
});
```

# Streams en NodeJS

## Usar la entrada estándar `process.stdin`

- Comenzamos con la configuración de la entrada de datos por consola.
  - `process.stdin.setEncoding('utf8');`
- Luego podemos mostrar un mensaje en la consola para que se sepa qué dato se está solicitando al usuario.
  - `process.stdout.write('mensaje por salida stdout');`

# Streams en NodeJS

## Usar la entrada estándar `process.stdin`

- Posteriormente podemos asociar un manejador de eventos a `stdin`, para el evento "data".
- Él produce que la lectura comience y una vez que se tenga algún dato, ejecutará la función `callback`.
- En la función `callback` recibiremos el chunk (buffer) de datos escritos en la consola hasta la pulsación de la tecla `enter`.

# Streams en NodeJS

## Usar la entrada estándar process.stdin

- 

```
process.stdin.once('data', function(res) {
 process.stdout.write('Has respondido: ');
 process.stdout.write(res);
 process.stdin.pause();
});
```

# Streams en NodeJS

## Usar la entrada estándar `process.stdin`

- Por último ejecutamos el método `pause()` que producirá que el stream pare de emitir datos, por lo que se dejará de leer en `stdin` y por tanto el programa acabará.



# Streams en NodeJS

## Generar tuberías entre streams

- Podemos conectar un stream de lectura a un stream de escritura, produciendo una tubería que enviará los datos del origen para el destino.
- Para ello usamos el método `pipe()`.
- Para crear esa tubería tenemos que invocar el método `pipe` sobre un stream de lectura.
- Ahora, además del stream de lectura del archivo de texto de antes, vamos a crear un stream de escritura sobre otro archivo del sistema.

# Streams en NodeJS

## Generar tuberías entre streams

- `var streamLectura = fs.createReadStream('./archivo-texto.txt');`
- `var streamEscritura = fs.createWriteStream('./otro-archivo.txt');`
- Ahora vamos a usar el método `pipe()` para realizar ese flujo de datos de un stream a otro.
- `streamLectura.pipe(streamEscritura);`
- Si ejecutamos ese método produciremos la copia del contenido del archivo "archivo-texto.txt" hacia el fichero "otro-archivo.txt"

# Streams en NodeJS

## Generar tuberías entre streams

- Podemos saber cuándo terminó esa copia del fichero si añadimos un evento "end" a nuestro stream de lectura.
- 

```
streamLectura.on('end', function() {
 console.log('La lectura del fichero se ha completado');
});
```

<https://nodejs.org/api/stream.html>