

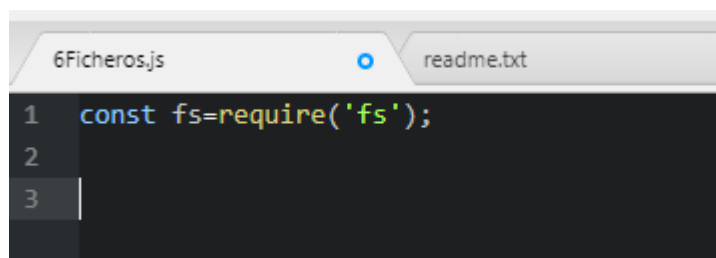
Índice

1. Archivos y directorios
2. Streams y buffers
3. Pipes

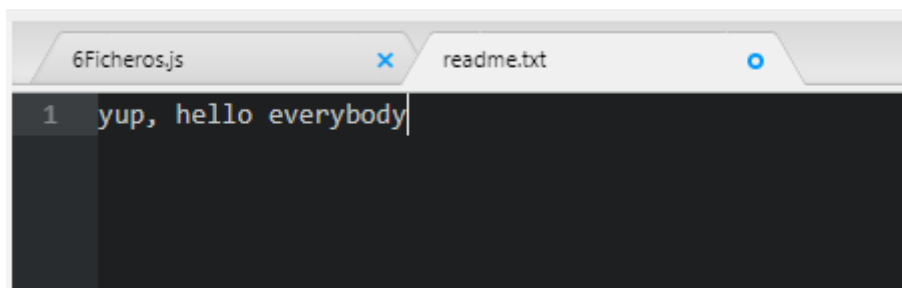
1. Archivos (módulo fs)

El módulo fs se encarga de soportar la lectura y escritura de archivos.

Lo importamos y también creamos un archivo de texto para trabajar.



```
6Ficheros.js  readme.txt
1  const fs=require('fs');
2
3
```



```
6Ficheros.js  readme.txt
1  yup, hello everybody
```

Hay dos formas de trabajar con archivos:

- sincrónico: mientras se lee o escribe un archivo, se paraliza la ejecución del código posterior.



```
1  const fs=require('fs');
2
3  var leeme=fs.readFileSync('readme.txt','utf8');
4  console.log(leeme);
5  fs.writeFileSync('escribeme.txt',leeme);
6

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
1: bash
manuela@manuela-SATELLITE-L50-B-1CE:~/Escritorio/DI/interface design with node and electron/02. Node/03. Files$ node
Ficheros.js
yup, hello everybody!!
manuela@manuela-SATELLITE-L50-B-1CE:~/Escritorio/DI/interface design with node and electron/02. Node/03. Files$
```

- asíncrono: otros procesos pueden continuar ejecutándose mientras se lee o escribe el archivo. En ese caso, tenemos que pasar una función de devolución de llamada a readFile y

writeFile , que será la que se ejecute al finalizar la acción. Esta función tomará como parámetros el error que salta si falla la operación y los datos que leemos o escribimos.

```
7 fs.readFile('./readme.txt', 'utf8', (err, data) => {
8   if (err) throw err;
9   console.log('Reading asynchronous file');
10  console.log(data);
11
12  fs.writeFile('./writeMe.txt', data, (err) => {
13    if (err) throw err;
14    err
15    ? console.log(err)
16    : console.log('File written correctly');
17  })
18 });
19
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: bash + [] [] ^ x

manuela@manuela-SATELLITE-L50-B-1CE:~/Escritorio/DI/interface design with node and electron/02. Node/03. Files\$ node Ficheros.js
Reading asynchronous file
yup, hello everybody!!
File written correctly

Hay más funciones disponibles en fs, podemos consultar la documentación de node

- fs.rename: renombrar archivos
 - fs.stat: información del archivo
 - fs.appendFile; agregar datos de forma asíncrona. Cree el archivo si no existe.
 - fs.unlink: eliminar un archivo
- ```
fs.unlink('writeMe.txt', (err) => { si (err) tirar err;
console.log ('writeMe.txt fue eliminado'); });
```

```

18 });
19
20 fs.rename('readme.txt', 'leeme.txt', (err) => {
21 if (err) throw err;
22 console.log('Nombre Editado Satisfactoriamente');
23 });
24 fs.stat('readme.txt', (err, stats) => {
25 if (err) throw err;
26 console.log(stats);
27 });
28

```



< archivo escrito correctamente  
 C:\Users\Dept Informática\node> node 6Ficheros

```

Stats {
 dev: 2863553481,
 mode: 33206,
 nlink: 1,
 uid: 0,
 gid: 0,
 rdev: 0,
 blksize: undefined,
 ino: 12384898975391016,
 size: 22,
 blocks: undefined,
 atimeMs: 1564668514350.6443,
 mtimeMs: 1564669965217.9336,
 ctimeMs: 1564669965217.9336,
 birthtimeMs: 1564668514350.6443,
}

```

```

fs.unlink('./stuff/writeMe.txt', function(){
 fs.rmdir('stuff',function (err) {
 if(err){
 throw err+' borrando carpeta stuff';
 }
 });
},function (err) {
 if(err){
 throw err+ 'borrando archivo stuff/writeMe.txt';
 }
});

```

```

fs.unlink('writeMe.txt', (err) => { if (err) throw err; console.log('writeMe.txt was deleted'); });

fs.rename('readme.txt', 'leeme.txt', (err) => {
 if (err) throw err;
 console.log('Name edited');
});
fs.stat('leeme.txt', (err, stats) => {
 if (err) throw err;
 console.log(stats);
});

fs.mkdirSync('mydir');
fs.rmdirSync('mydir');
fs.mkdir('./stuff/things', { recursive: true }, (err) => {
 if (err) throw err;
});

fs.rmdir('./stuff/things', (err) => {
 if (err) {
 throw err + ' borrando carpeta things';
 }
});

```

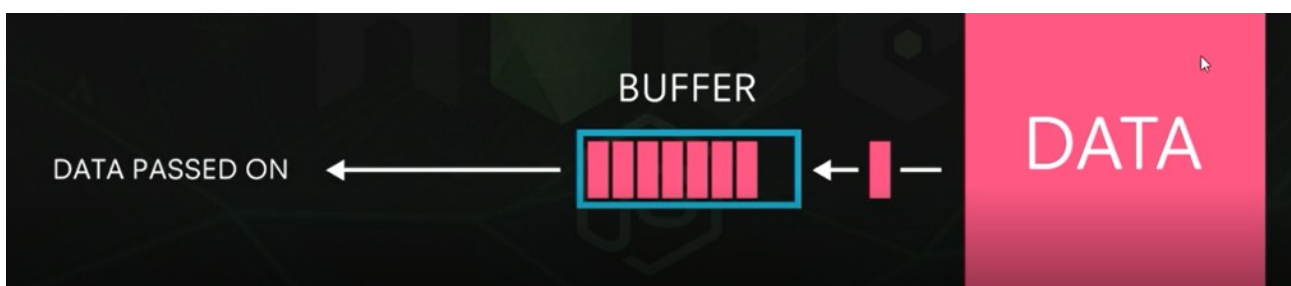
```

fs.unlink('./stuff/writeMe.txt', function(){
 fs.rmdir('stuff', function (err) {
 if (err) {
 throw err + ' borrando carpeta stuff';
 }
 });
}, function (err) {
 if (err) {
 throw err + ' borrando archivo stuff/writeMe.txt'; // throw error when deleting writeMe.txt
 }
});

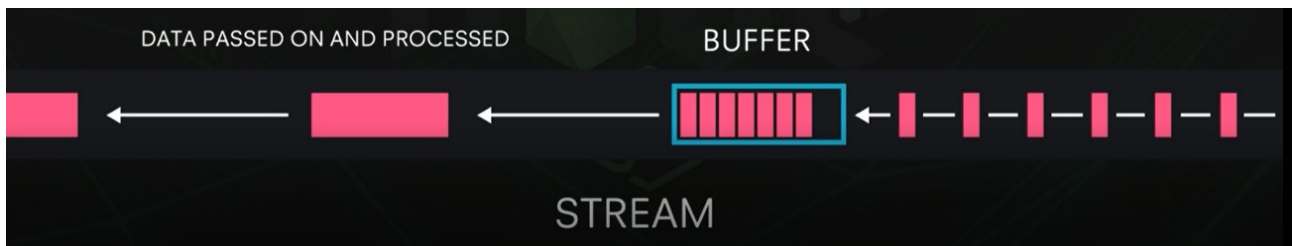
```

## 2. Streams and buffers

Un búfer es un almacenamiento temporal de una pieza de datos que pertenece a un conjunto de datos mucho más grande que se transporta de un lugar a otro. El búfer se llena de datos y se transfiere al destino final.



Cuando el búfer está lleno, estos datos se transfieren a través de un flujo, que sería el propio canal.



Este proceso ocurre, por ejemplo, al ver películas en streaming: no esperamos a que la película completa se haya transferido al pc, sino que visualizamos los datos en estos pequeños fragmentos que se transfieren a los búferes.

En node podemos crear búferes y flujos para pasar datos, lo que aumenta la eficiencia y el rendimiento de la aplicación.

### Streams de lectura

Una secuencia de lectura se utiliza para leer un archivo grande, por ejemplo. Cree un archivo con suficiente texto, por ejemplo, la historia de los hermanos del gueto del bronx.

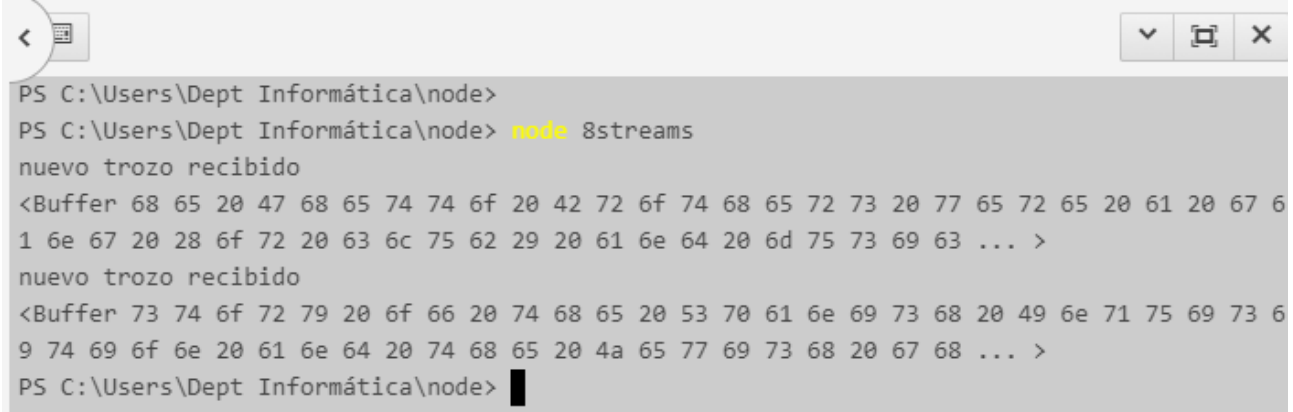
```
const fs=require('fs');

var miStream=fs.createReadStream(__dirname+'/ghettobrothers.txt');
```

En esta variable miStream se almacenarían bits del archivo que se han almacenado en el búfer.

createReadStream hereda de EventEmitter y tiene un evento llamado data que se activa cada vez que se llena la transmisión, por lo que lo usaremos para ver los fragmentos.

```
5 var miStream=fs.createReadStream(__dirname+'/ghettobrothers.txt');
6
7 miStream.on('data',function (chunk) {
8 console.log('nuevo trozo recibido ');
9 console.log(chunk);
10
11 })
12
```



```
PS C:\Users\Dept Informática\node>
PS C:\Users\Dept Informática\node> node 8streams
nuevo trozo recibido
<Buffer 68 65 20 47 68 65 74 74 6f 20 42 72 6f 74 68 65 72 73 20 77 65 72 65 20 61 20 67 6
1 6e 67 20 28 6f 72 20 63 6c 75 62 29 20 61 6e 64 20 6d 75 73 69 63 ... >
nuevo trozo recibido
<Buffer 73 74 6f 72 79 20 6f 66 20 74 68 65 20 53 70 61 6e 69 73 68 20 49 6e 71 75 69 73 6
9 74 69 6f 6e 20 61 6e 64 20 74 68 65 20 4a 65 77 69 73 68 20 67 68 ... >
PS C:\Users\Dept Informática\node>
```

Si en vez de verlo en hexadecimal queremos ver el texto en sí, añadimos la codificación

```
var miStream=fs.createReadStream(__dirname+'/ghettobrothers.txt','utf8');

miStream.on('data',function (chunk) {
 console.log('nuevo trozo recibido ');
 console.log(chunk);
})
```

Para controlar el tamaño del búfer, por ejemplo 1bytes

```
const fs=require('fs');
var readStream=fs.createReadStream(__dirname+'/ghettobrothers.txt',{highWaterMark: 1});

readStream.on('data',function(chunk){
 console.log('new piece recieved');
 console.log(chunk);
})
```

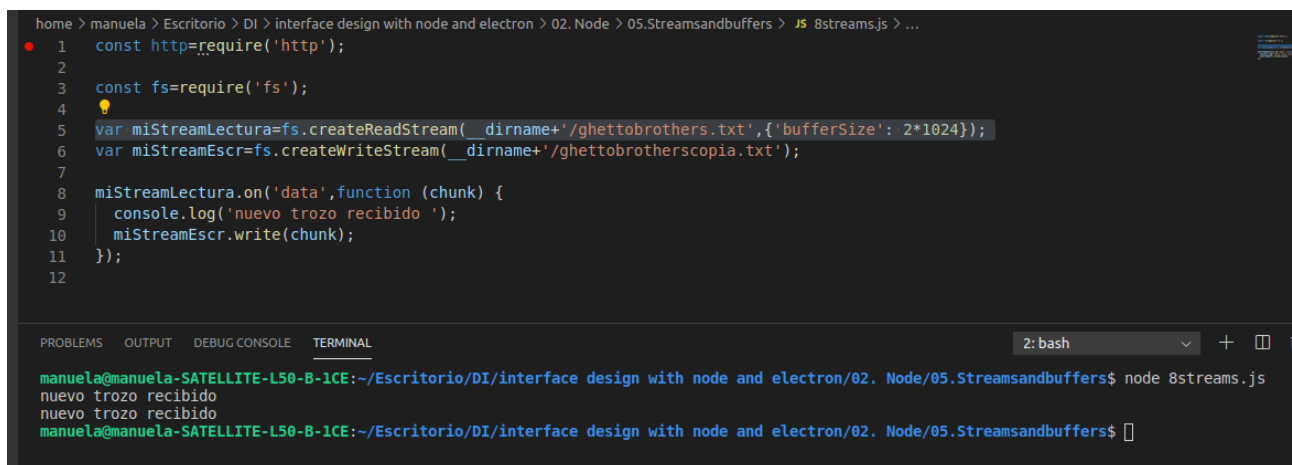
Cada dato podría pasarse al cliente con un flujo de escritura en lugar de esperar a tener el archivo completo.

## Stream de escritura

Por lo general, queremos pasar los datos almacenados en búfer a un archivo o cliente, no solo mostrarlos en la consola. Para esto necesitamos un flujo de escritura. Para crear uno que escriba en un archivo:

```
var miStreamEscr=fs.createWriteStream(__dirname+'/ghettobrotherscopia.txt');
```

Cada vez que llegue un fragmento de datos, lo escribiremos en este nuevo archivo.



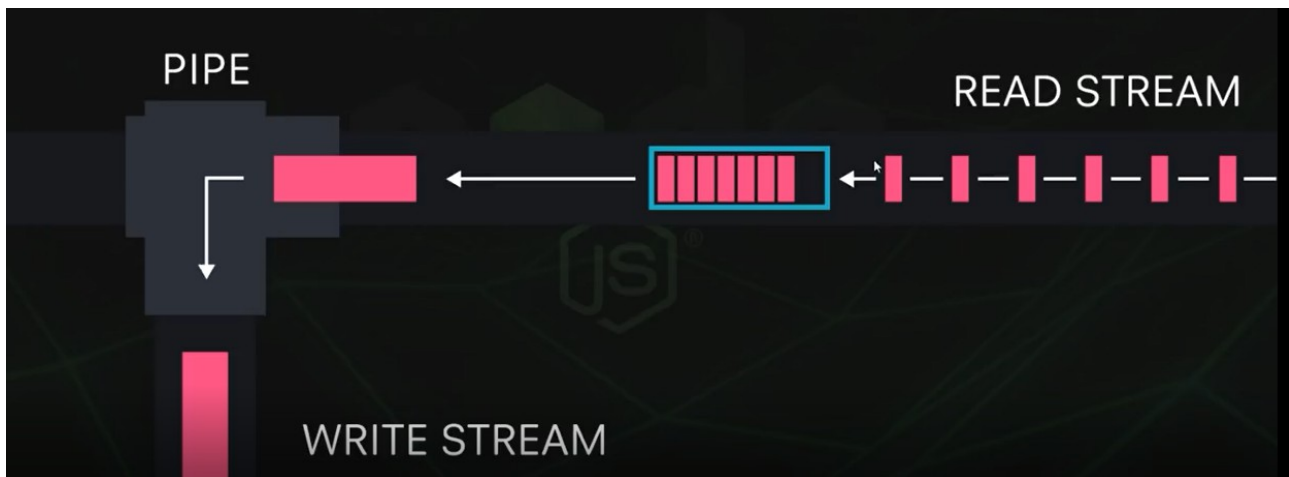
```
home > manuela > Escritorio > DI > interface design with node and electron > 02. Node > 05.Streamsandbuffers > JS 8streams.js > ...
1 const http=require('http');
2
3 const fs=require('fs');
4
5 var miStreamLectura=fs.createReadStream(__dirname+'/ghettobrothers.txt',{'bufferSize': 2*1024});
6 var miStreamEscr=fs.createWriteStream(__dirname+'/ghettobrotherscopia.txt');
7
8 miStreamLectura.on('data',function (chunk) {
9 console.log('nuevo trozo recibido ');
10 miStreamEscr.write(chunk);
11 });
12
```

manuela@manuela-SATELLITE-L50-B-1CE:~/Escritorio/DI/interface design with node and electron/02. Node/05.Streamsandbuffers\$ node 8streams.js  
nuevo trozo recibido  
nuevo trozo recibido  
manuela@manuela-SATELLITE-L50-B-1CE:~/Escritorio/DI/interface design with node and electron/02. Node/05.Streamsandbuffers\$

La diferencia entre este método y el readFile writeFile que hicimos anteriormente es que en este caso los datos se pueden utilizar a medida que llegan a su destino, y es mucho más rápido. Hay una forma aún más rápida de hacerlo: tuberías.

### 3. Tubería

La canalización nos ahorra la molestia de "escuchar" manualmente cuando llega un dato para escribirlo en el flujo de escritura, lo hace automáticamente.



El código anterior se simplifica utilizando el método de tubería de flujo legible

```
var miStreamLectura=fs.createReadStream(__dirname+'/ghettobrothers.txt');
var miStreamEscr=fs.createWriteStream(__dirname+'/ghettobrotherscopia.txt');

miStreamLectura.pipe(miStreamEscr);
```

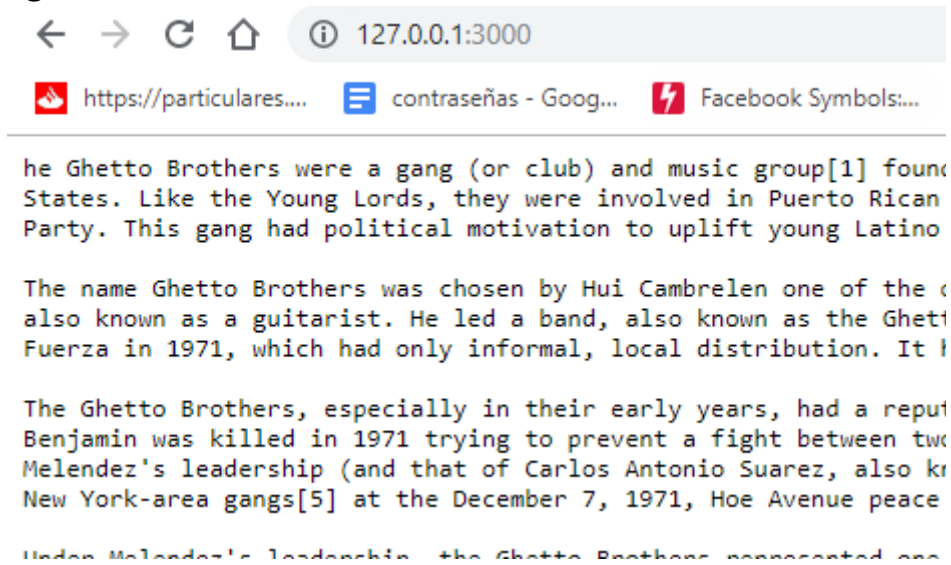
Vamos a usar esta idea para enviar datos a un cliente usando el servidor que creamos hace días, enviándole el archivo Ghetto Brothers para que pueda aprender sobre la cultura urbana. Para esto, en lugar de hacer que la salida de la canalización sea un flujo grabable, lo convertimos en la respuesta del cliente.

```
1 const http=require('http');
2 const fs=require('fs');
3
4 var server=http.createServer(function (req,res) {
5 console.log(req.url);
6 res.writeHead(200, {'Content-type':'text/plain'});
7 var miStreamLectura=fs.createReadStream(__dirname+'/ghettobrothers.txt');
8 miStreamLectura.pipe(res);
9 });
10
11 server.listen(3000,'127.0.0.1');
12
```

PS C:\Users\Dept Informática\node> node 9pipes



En el navegador:



Se muestra en modo texto porque la codificación predeterminada es utf8.

Si queremos enviar un archivo html como por ejemplo

Tenemos que cambiar el tipo de texto que se envía para que el navegador trate el archivo como html y no como texto sin formato

```
res.writeHead(200, {'Tipo de contenido':'texto/html'});
```

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <style>
5 body{background:skyblue;font-family:verdana;color:#fff;padding:30px;}
6 h1{font-size:48px;text-transform:uppercase;letter-spacing:2px;text-align:center;}
7 </style>
8 </head>
9 <body>
10 <h1>La mejor página ever</h1>
11 Aquí hay un contenido chachi
12 </body>
13 </html>
```

en el navegador



# LA MEJOR PÁGINA EVER