

TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE:

- Uso de callback y problemas comunes.
- Orden de código vs Orden de ejecución.
- Introducción y uso de promesas.
- Uso de async await.
- Módulo fs de Node.js.

Comenzaremos a revisar el comportamiento del código asíncrono, y el uso de **callbacks** en situaciones reales.

Dentro del mundo de la programación con JavaScript, existe un concepto llamado **“callback hell”**, que se traduce como “infierno de **callbacks**”, y se refiere a cuando necesitamos hacer uso de muchas **callbacks** consecutivas, lo cual vuelve nuestro código ilegible y muy difícil de controlar.

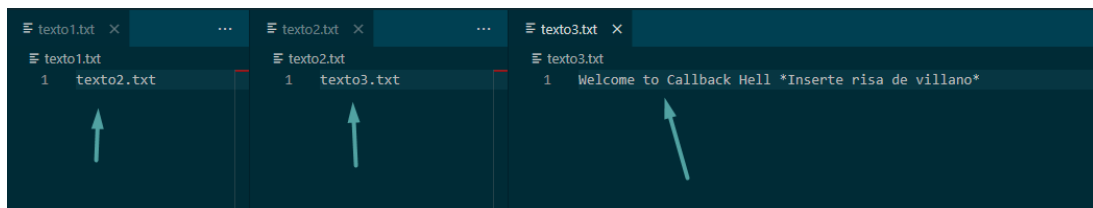
```
1  var floppy = require('floppy');
2
3  floppy.load('disk', function(data) {
4    floppy.load('disk', function(data) {
5      floppy.load('disk', function(data) {
6        floppy.load('disk', function(data) {
7          floppy.load('disk', function(data) {
8            floppy.load('disk', function(data) {
9              floppy.load('disk', function(data) {
10               floppy.load('disk', function(data) {
11                });
12              });
13            });
14          });
15        });
16      });
17    });
18  });
```

Para mostrar un ejemplo práctico respecto al **"Callback Hell"**, primero hablaremos sobre unos módulos o paquetes especiales, que son nativos de **node**, es decir, se encuentran instalados por defecto, y solo necesitamos requerirlos.

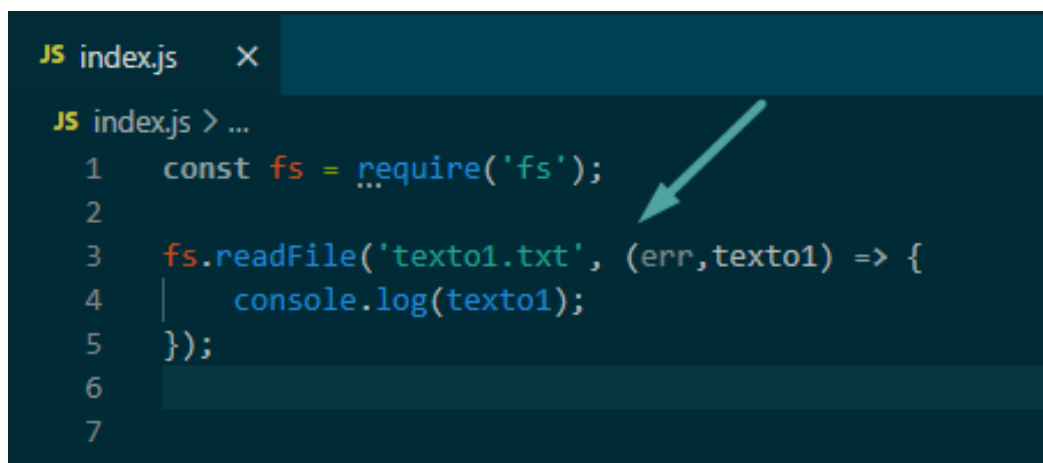
A continuación, conoceremos el módulo **"fs"** (File System), que es de mucha utilidad pues nos permite acceder al sistema de archivos de nuestro computador, para realizar distintas acciones.

Crea una nueva carpeta, y dentro de ella, tres archivos de texto llamados: "texto1.txt", "texto2.txt", y "texto3.txt".

El archivo texto1.txt tendrá como contenido "texto2.txt"; el archivo texto2.txt tendrá escrito "texto3.txt"; y el archivo texto3.txt tendrá escrito "Welcome to Callback Hell *Inserte risa de villano*"



En primer lugar, accederemos al archivo texto1.txt, para ello creamos un archivo index.js, y requerimos el módulo fs. Utilizamos el método **readFile()**, que toma como primer argumento el nombre del archivo al cual queremos acceder, y como segundo argumento, toma una función de callback, la que a su vez, tendrá como parámetros: el error en caso de ocurrir, y el contenido del archivo a leer.



Al ver por consola la respuesta, podemos notar que se está obteniendo un valor extraño.

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\codigo-asincrono>node index.js
<Buffer 74 65 78 74 6f 32 2e 74 78 74>

C:\Users\codigo-asincrono>

```

Esto se debe a que el método `readFile` devuelve un `stream` de bytes por defecto, salvo que le pasemos la indicación del formato en que queremos recibir el contenido del archivo, tal como un argumento extra.

```

JS index.js  X
JS index.js > ...
1  const fs = require('fs');
2
3  fs.readFile('texto1.txt', 'utf8', (err, texto1) => {
4    console.log(texto1);
5  });
6
7

```

```

PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\codigo-asincrono>node index.js
texto2.txt

C:\Users\codigo-asincrono>

```

Ahora que hemos logrado obtener el texto contenido en el archivo "texto1.txt", utilizaremos el valor para acceder al archivo "texto2.txt".

```

1  const fs = require('fs');
2
3  fs.readFile('texto1.txt', 'utf8', (err, texto1) => {
4
5      fs.readFile(texto1, 'utf8', (err, texto2) => {
6
7          console.log(texto2)
8
9      })
10 })
11

```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```

C:\Users\codigo-asincrono>node index.js
texto3.txt
C:\Users\codigo-asincrono>

```

Y, por último, utilizaremos el valor obtenido de "texto2.txt", para acceder al archivo "texto3.txt".

```

1  const fs = require('fs');
2
3  fs.readFile('texto1.txt', 'utf8', (err, texto1) => {
4
5      fs.readFile(texto1, 'utf8', (err, texto2) => {
6
7          fs.readFile(texto2, 'utf8', (err, textoFinal) => {
8
9              console.log(textoFinal)
10
11          })
12      })
13  })
14  });
15

```

PROBLEMS OUTPUT **TERMINAL** DEBUG CONSOLE

```

C:\Users\codigo-asincrono>node index.js
Welcome to Callback Hell *Inserte risa de villano*
C:\Users\codigo-asincrono>

```

Además, agregaremos una validación en cada callback para controlar un posible error.

```
JS index.js X
JS index.js > ...
1  const fs = require('fs');
2
3  fs.readFile('texto1.txt', 'utf8', (err, texto1) => {
4      if(err){
5          return console.log(err);
6      }
7      fs.readFile(texto1, 'utf8', (err, texto2) => {
8          if(err){
9              return console.log(err);
10         }
11         fs.readFile(texto2, 'utf8', (err, texto3) => {
12             if(err){
13                 return log(err);
14             }
15             console.log(texto3);
16         });
17     });
18 });
19
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\codigo-asincrono>node index.js
Welcome to Callback Hell *Inserte risa de villano*

C:\Users\codigo-asincrono>
```

Aquí solo estamos haciendo tres llamadas consecutivas al mismo método, y puedes ver cómo ha crecido rápidamente la cantidad de líneas de código, y lo complicado que se ha vuelto de leer. Antes de pasar al siguiente tema, el cual nos permitirá escribir código más limpio, observaremos otra problemática que se presenta con el uso de callbacks.

Veamos el siguiente ejemplo.

Crea un archivo lorem.txt, y pega el texto a continuación:

“Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec eleifend lobortis enim quis varius. Donec odio neque, efficitur non ante quis, dapibus ultricies ligula. Curabitur luctus lorem ac mauris laoreet, ut bibendum sapien gravida. Vivamus tempor velit ac arcu malesuada, ac congue nulla accumsan. Maecenas facilisis lacus vitae ante semper, a blandit arcu tempus. Integer ut convallis magna. Etiam malesuada augue a finibus efficitur. Praesent volutpat magna at augue aliquam fringilla. Mauris malesuada sem eu pharetra ultrices.”

```
lorem.txt x
lorem.txt
1 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec eleifend lobortis enim quis varius.
2 Donec odio neque, efficitur non ante quis, dapibus ultricies ligula. Curabitur luctus lorem ac mauris laoreet,
3 ut bibendum sapien gravida. Vivamus tempor velit ac arcu malesuada, ac congue nulla accumsan. Maecenas facilisis
4 lacus vitae ante semper, a blandit arcu tempus. Integer ut convallis magna. Etiam malesuada augue a finibus efficitur.
5 Praesent volutpat magna at augue aliquam fringilla. Mauris malesuada sem eu pharetra ultrices.
```

Ahora, intentaremos acceder al sistema de archivos del computador, para que nuestro programa muestre por pantalla el texto contenido en "lorem.txt", guardando nuestra función dentro de una variable, y retornando el valor del texto.

```
lorem.txt JS index.js x
JS index.js > ...
1 const fs = require('fs');
2
3 const textoLorem = fs.readFile('lorem.txt', 'utf8', (err, texto) => {
4   if(err){
5     return err;
6   }
7   return texto;
8 });
9
10 console.log("Texto lorem");
11 console.log(textoLorem);
12
13
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
C:\Users\codigo-asincrono>node index.js
Texto lorem
undefined
C:\Users\codigo-asincrono>[]
```

Parece que nuestra función está retornando indefinido. Revisemos qué sucede si tratamos de acceder a la respuesta dentro de la función, tal como lo hicimos en el ejemplo anterior.



```
JS index.js > ...
2
3 const textoLorem = fs.readFile('lorem.txt', 'utf8', (err, texto) => {
4   if(err){
5     return err;
6   }
7   console.log(texto);
8   return texto;
9 });
10
11 console.log("Texto lorem");
12 console.log(textoLorem);
13
14
```

C:\Users\codigo-asincrono>node index.js

Texto lorem

undefined

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec eleifend lobortis enim quis varius. Donec odio neque, efficitur non ante quis, dapibus ultricies ligula. Curabitur luctus lorem ac mauris laoreet, ut bibendum sapien gravida. Vivamus tempor velit ac arcu malesuada, ac congue nulla accumsan. Maecenas facilisis lacus vitae ante semper, a blandit arcu tempus. Integer ut convallis magna. Etiam malesuada augue a finibus efficitur. Praesent volutpat magna at augue aliquam fringilla. Mauris malesuada sem eu pharetra ultrices.

C:\Users\codigo-asincrono>

Aquí se observa que el texto se encuentra dentro de la función, entonces ¿por qué no podemos acceder fuera de ella? Si recuerdas, cuando hablamos del código asíncrono, la línea 12 del código es ejecutada **mientras** nuestro archivo está siendo leído. Esta es una operación que lleva tiempo, independientemente de lo grande que sea el contenido del archivo que estemos leyendo.

PROMESAS

¿Qué sucede si queremos acceder al contenido de nuestros archivos de forma secuencial, sin tener que pensar si este contenido estará listo para ser leído y utilizado en nuestro código? Para todas estas tareas que llevan tiempo, se ha introducido el concepto de promesas. Una promesa no es más que una porción de código que tendrá dos posibles salidas: o se cumple de forma exitosa, o no se cumple y falla.

Para empezar a utilizarlas, tenemos dos formas. Actualmente, la mayoría de los módulos crean sus propios métodos con soporte para promesas; es decir, generalmente éstos retornan una promesa, aunque depende de cada módulo e implementación. Por otra parte, podemos crear nuestras propias promesas.

Para construir una promesa, debemos utilizar el constructor `new Promise`, y pasar como argumento una función que, a su vez, recibe otros dos argumentos: `resolve` o `reject`. Ya dentro del cuerpo de esta función, se decidirá en qué condiciones queremos resolverla o rechazarla. En

el ejemplo a continuación, estamos definiendo la creación de un número al azar, y evaluando si es mayor a 7 o no. Si lo es, la promesa es rechazada, y en caso contrario, ésta se resuelve.

```
JS index.js X
JS index.js > ...
15  const numeroAzar = Math.floor(Math.random() * 10);
16
17  const promesa = new Promise((resolve, reject) => {
18
19      if( numeroAzar > 7 ){
20
21          reject("La promesa ha fallado");
22
23      } else {
24
25          resolve("El número resultante fue menor a 7");
26
27      }
28  })
29
30
```

Para finalizar la implementación de una promesa, primero debemos utilizar el método `then()`, el cual recibe una función de callback con el resultado de ésta.

```
JS index.js X
JS index.js > ...
15  const numeroAzar = Math.floor(Math.random() * 10);
16
17  const promesa = new Promise((resolve, reject) => {
18
19      if( numeroAzar > 7 ){
20
21          reject("La promesa ha fallado");
22
23      } else {
24
25          resolve("El número resultante fue menor a 7");
26
27      }
28  })
29
30
31  promesa.then( resultado => {
32      console.log(resultado);
33  });
34
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\codigo-asincrono>node index.js
El número resultante fue menor a 7
C:\Users\codigo-asincrono>
```

El método `then` solo tomará el resultado de la promesa cuando ésta ha sido resuelta, pero en caso de que sea rechazada, obtendremos el siguiente mensaje de error.


```

15
16 const numeroAzar = Math.floor(Math.random() * 10);
17
18 const promesa = new Promise((resolve, reject) => {
19
20   if( numeroAzar > 7 ){
21
22     reject("La promesa ha fallado");
23
24   } else {
25
26     resolve("El número resultante fue menor a 7");
27
28   }
29 })
30
31 promesa.then( resultado => {
32   console.log(resultado);
33 });
34

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\codigo-asincrono>node index.js
(node:13640) UnhandledPromiseRejectionWarning: La promesa ha fallado
(Use 'node --trace-warnings ...' to show where the warning was created)
(node:13640) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node process on unhandled promise rejection, use the CLI flag '--unhandled-rejections=strict' (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode). (rejection id: 2)
(node:13640) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future, promise rejections that are not handled will terminate the Node.js process with a non-zero exit code.
C:\Users\codigo-asincrono>]

Esto sucede porque debemos controlar el caso de una promesa rechazada. Para ello, utilizamos el método **catch()**, el cual recibe un **callback** con el error como argumento.

```

15
16 const numeroAzar = Math.floor(Math.random() * 10);
17
18 const promesa = new Promise((resolve, reject) => {
19
20   if( numeroAzar > 7 ){
21
22     reject("La promesa ha fallado");
23
24   } else {
25
26     resolve("El número resultante fue menor a 7");
27
28   }
29 })
30
31 promesa.then( resultado => {
32   console.log(resultado);
33 })
34 .catch( error => {
35   console.log(error);
36 });
37

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\codigo-asincrono>node index.js
La promesa ha fallado
C:\Users\codigo-asincrono>]

Y como puedes ver, en caso de que la promesa sea rechazada, ahora tenemos control sobre la excepción.

Para utilizar el módulo `fs` implementando promesas, primero debemos requerirlo de la siguiente manera.

```
JS index.js  X
JS index.js > ...
1  const fs = require('fs/promises');
2
3
```

Esto nos permite ahora cambiar la sintaxis del primer ejemplo, de la siguiente forma.

```
JS index.js  X
JS index.js > ...
44
45  fs.readFile('texto1.txt', 'utf8')
46    .then( texto1 => fs.readFile(texto1, 'utf8'))
47    .then( texto2 => fs.readFile(texto2, 'utf8'))
48    .then( textoFinal => console.log(textoFinal))
49    .catch( error => console.log(error));
50
51
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
C:\Users\codigo-asincrono>node index.js
Welcome to Callback Hell *Inserte risa de villano*
C:\Users\codigo-asincrono>
```


Utilizando una cadena de `then()`, podemos llegar a un código que tiene la misma funcionalidad que el del ejemplo de uso de callbacks; pero escribiendo uno muchísimo más ordenado y secuencial.

ASYNC - AWAIT

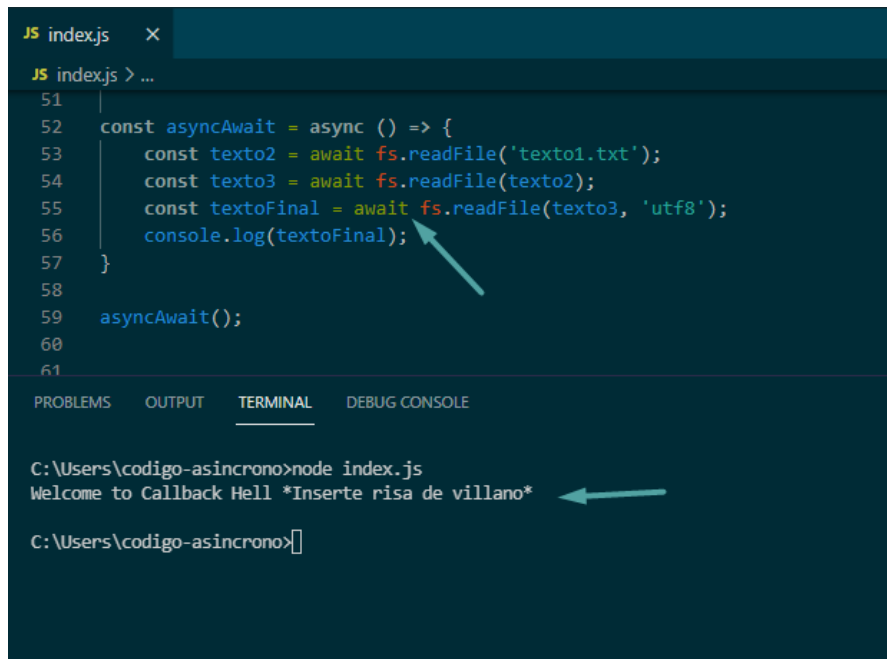
Además de las promesas, existe otra forma más de sintaxis, la cual nos permite escribir un código aún más limpio, y tener la sensación de “código síncrono”. Con ésta será mucho más fácil ordenar la ejecución de tu programa cuando existan tareas asíncronas, pues da la posibilidad de detenerla, hasta que se obtenga la respuesta que estás esperando.

Para poder utilizar esta sintaxis, primero se debe definir una función con la palabra clave `async`.

```
1  
2  const asyncAwait = async () => {  
3  
4  }  
5  
6
```



Esta palabra clave nos indica que, dentro del cuerpo de la función, se ejecutará un código asíncrono, lo que nos permite “esperar” la respuesta de cada llamada, utilizando la palabra clave `await`. Veamos cómo quedaría la implementación del ejemplo que hemos usado para `callbacks` y promesas, utilizando el módulo `fs`.



```
JS index.js X
JS index.js > ...
51
52 const asyncAwait = async () => {
53   const texto2 = await fs.readFile('texto1.txt');
54   const texto3 = await fs.readFile(texto2);
55   const textoFinal = await fs.readFile(texto3, 'utf8');
56   console.log(textoFinal);
57 }
58
59 asyncAwait();
60
61

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\codigo-asincrono>node index.js
Welcome to Callback Hell *Inserte risa de villano*
C:\Users\codigo-asincrono>
```

Cada vez que utilizamos la palabra clave **await** frente a una llamada asíncrona, estaremos esperando a que termine la ejecución de esta línea, antes de pasar a la siguiente, lo que nos permite guardar el valor dentro de una variable, y luego pasarla a la siguiente llamada.

Para controlar los errores que puedan ocurrir dentro de la ejecución de una tarea con la palabra clave **await**, utilizamos un bloque **try-catch**, de la siguiente forma.

```

JS index.js x
JS index.js > ...
51
52 const asyncAwait = async () => {
53   try {
54     const texto2 = await fs.readFile('texto1.txt');
55     const texto3 = await fs.readFile(texto2);
56     const textoFinal = await fs.readFile(texto3, 'utf8');
57     console.log(textoFinal);
58   } catch(error) {
59     console.log('Ha ocurrido un error al leer el archivo');
60     console.log(error);
61   }
62 }
63
64 asyncAwait();
65
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

C:\Users\codigo-asincrono>node index.js
Ha ocurrido un error al leer el archivo
[Error: ENOENT: no such file or directory, open 'C:\Users\codigo-asincrono\texto1.txt'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\Users\codigo-asincrono\texto1.txt'
}

C:\Users\codigo-asincrono>

```

Como puedes ver, logramos un código secuencial, mucho más limpio que con **callbacks**, e incluso que con el uso de promesas.

BLOCKING VERSUS NON BLOCKING

Este concepto tiene que ver con la ejecución sincrónica o asincrónica de un código. Cuando hablamos de bloqueo, hablamos de que la ejecución del código debe esperar hasta que se complete una operación para continuar su flujo, por el contrario, la biblioteca estándar de node.js proporciona versiones asincrónicas de todos los métodos, es decir, que no bloquean la ejecución del código y aceptan funciones de devolución de llamada (callback) como vimos previamente en esta lectura.

Haciendo una comparación entre bloqueo y no bloqueo, usando un módulo del sistema de archivos, la lectura de un archivo sincrónica sería la siguiente:

```
const fs = require("fs");
const data = fs.readFileSync("/file.md"); // blocks here until file is read
```

Por su parte, podemos utilizar un módulo asincrónico que recibe las funciones de devolución de llamadas y permite que el código continúe ejecutándose.

```
const fs = require("fs");
fs.readFile("/file.md", (err, data) => {
  if (err) throw err;
});
```

HELPERS

Un Helpers (ayudates) es una forma de agrupar funciones de uso común para ayudar en los procesos. Se componen de funciones genéricas que se encargan de realizar acciones complementarias y pueden ser aplicables a cualquier elemento de un sistema. Un punto importante con respecto a los helpers es que estos pueden ser una agrupación de funciones “sueltas” que puedan servir para la solución de distintos procesos o tipo objetos de clases.

```
static helpers(name) {
  if (!name) {
    return container.helpers;
  }
  return container.helpers[name];
}
```

Fragmento del uso de un helpers de Handlebard.

```
describe(`parameterNameAndType helper`, () => {
  test(`sould compile`, () => {
    const data = project.findReflectionByName('objectLiteral');
    const result = Handlebars.helpers.parameterNameAndType.call(data);
    expect(result).toMatchSnapshot();
  });
});
```