



Clase 4. Programación Backend

Sync Async - Manejo de Archivos en Javascript



OBJETIVOS DE LA CLASE

- Ver ejemplos prácticos, ventajas y desventajas de cada uno de los modos de programación sincrónica y asincrónica .
- Conocer el módulo que usa Node.js para acceder al sistema de archivos.
- Utilizar la forma sincrónica y asincrónica para interactuar con los archivos.
- Como se utiliza callback en programación asincrónica.

CRONOGRAMA DEL CURSO

Clase 3



Programación sincrónica
y asincrónica

Clase 4



Manejo de Archivos en
Javascript

Clase 5



Administradores de
Paquetes - NPM

Repasando...

Sincronismo vs Asincronismo



Ejecución sincrónica vs. ejecución asincrónica

Ejecución Sincrónica: Repasemos

- Cuando escribimos más de una instrucción en un programa, esperamos que las instrucciones se ejecuten comenzando desde la primera línea, una por una, de arriba hacia abajo hasta llegar al final del bloque de código.
- Si una instrucción es una llamada a otra función, la ejecución se pausa y se procede a ejecutar esa función.
- Sólo una vez ejecutadas todas las instrucciones de esa función, el programa retomará con el flujo de instrucciones que venía ejecutando antes.

Ejemplo Ejecución Sincrónica

```
function funA() {  
  console.log(1)  
  funB()  
  console.log(2)  
}  
function funB() {  
  console.log(3)  
  funC()  
  console.log(4)  
}  
function funC() {  
  console.log(5)  
}  
  
funA()  
  
//Al ejecutar la función funA()  
//se muestra lo siguiente por pantalla:  
1  
3  
5  
4  
2
```

- En todo momento, sólo se están ejecutando las instrucciones de una sola de las funciones a la vez. O sea, **debe finalizar una función para poder continuar con la otra.**
- El fin de una función marca el inicio de la siguiente, y el fin de ésta, el inicio de la que le sigue, y así sucesivamente, describiendo una **secuencia que ocurre en una única línea de tiempo.**

Comportamiento de una función: Bloqueante vs no-bloqueante

Cuando alguna de las instrucciones dentro de una función intente acceder a un recurso que se encuentre fuera del programa (por ejemplo, enviar un mensaje por la red, o leer un archivo del disco) nos encontraremos con dos maneras distintas de hacerlo: en forma bloqueante, o en forma no-bloqueante (blocking o non-blocking).

Operaciones bloqueantes



- En la mayoría de los casos, precisamos que el programa ejecute todas sus operaciones en forma secuencial, y sólo comenzar una instrucción luego de haber terminado la anterior.
- A las operaciones que obligan al programa a esperar a que se finalicen antes de pasar a ejecutar la siguiente instrucción se las conoce como **bloqueantes**.
- Este tipo de operaciones permiten que el programa se comporte de la manera más intuitiva.
- Permiten la ejecución de una sola operación en simultáneo.
- A este tipo de ejecución se la conoce como **sincrónica**.

Operaciones no-bloqueantes



- En algunos casos esperar a que una operación termine para iniciar la siguiente podría causar grandes demoras en la ejecución del programa.
- Por eso que Javascript ofrece una segunda opción: las operaciones **no bloqueantes**.
- Este tipo de operaciones permiten que, una vez iniciadas, el programa pueda continuar con la siguiente instrucción, sin esperar a que finalice la anterior.
- Permite la ejecución de varias operaciones en paralelo, sucediendo al mismo tiempo.
- A este tipo de ejecución se la conoce como **asincrónica**.

Timers



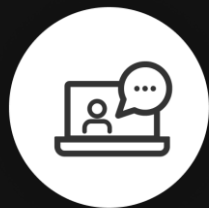
setTimeout

- ❑ ***setTimeout(function, milliseconds, param1, param2, ...)***
 - Es una función nativa, no hace falta importarla.
 - La función ***setTimeout()*** recibe un callback, y lo ejecuta después de un número específico de milisegundos.
 - Trabaja sobre un modelo asincrónico no bloqueante.

setInterval



- ❑ ***setInterval(cb, milliseconds, param1, param2, ...): Object***
 - Es una función nativa, no hace falta importarla.
 - La función *setInterval()* también recibe un callback, pero a diferencia de *setTimeout()* lo ejecuta una y otra vez cada vez que se cumple la cantidad de milisegundos indicada.
 - Trabaja sobre un modelo asincrónico no bloqueante.
 - El método *setInterval()* continuará llamando al callback hasta que se llame a *clearInterval()* o se cierre la ventana.
 - El objeto devuelto por *setInterval()* se usa como argumento para llamar a la función *clearInterval()*.



¡Vamos al código!

Concepto Ejecución Asíncronica

- Para poder usar funciones que realicen operaciones no bloqueantes debemos **aprender a usarlas adecuadamente**, sin generar efectos adversos en forma accidental.
- Cuando el código que se ejecuta en forma sincrónica, establecer el orden de ejecución consiste en decidir qué instrucción escribir primero.
- Cuando se trata de **ejecución asíncronica**, sólo sabemos en qué orden comenzarán su ejecución las instrucciones, pero **no sabemos en qué momento ni en qué orden terminarán de ejecutarse**.

Ejemplo Ejecución Asíncronica

```
const escribirArchivo = require('./escriArch.js')

console.log('inicio del programa')

// el creador de esta funcion la definió
// como no bloqueante. recibe un callback que
// se ejecutará al finalizar la escritura.
escribirArchivo('hola mundo', () => {
  console.log('terminé de escribir el archivo')
})

console.log('fin del programa')

// se mostrará por pantalla:
// > inicio del programa
// > fin del programa
// > terminé de escribir el archivo
```

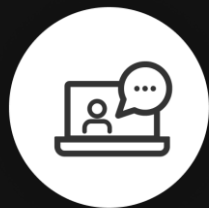
En el ejemplo no se bloquea la ejecución normal del programa y se permite que este se siga ejecutando. La ejecución de la operación de escritura “comienza” e inmediatamente cede el control a la siguiente instrucción, que escribe por pantalla el mensaje de finalización. Cuando la operación de escritura termina, ejecuta el callback que informará por pantalla que la escritura se realizó con éxito.

Ejemplo Ejecución Asíncronica : Aclaración

Si queremos que el mensaje de 'finalizado' **salga después** de haber grabado el archivo, **anidaremos las instrucciones dentro del callback** de la siguiente manera:

```
escribirArchivo('hola mundo', () => {  
  console.log('terminé de escribir el archivo')  
  console.log('fin del programa')  
})
```

Esto funciona porque lo (único) que podemos controlar en este tipo de operaciones es que el callback siempre se ejecuta luego de finalizar todas las demás instrucciones involucradas en ese llamado.



¡Vamos al código!



Determinaremos en los siguientes fragmentos de código el orden de salida de los mensajes a la consola y explicaremos el por que en cada caso

```
const delay = ret => {for(let i=0; i<ret*3e6; i++);}

function hacerTarea(num) {
  console.log('haciendo tarea ' + num)
  delay(100)
}

console.log('inicio de tareas');
hacerTarea(1)
hacerTarea(2)
hacerTarea(3)
hacerTarea(4)
console.log('fin de tareas')
console.log('otras tareas ...')
```

```
function hacerTarea(num, cb) {
  console.log('haciendo tarea ' + num)
  setTimeout(cb,100)
}

console.log('inicio de tareas');
hacerTarea(1, () => {
  hacerTarea(2, () => {
    hacerTarea(3, () => {
      hacerTarea(4, () => {
        console.log('fin de tareas')
      })
    })
  })
})
console.log('otras tareas ...')
```



Asincronismo y callbacks

Realizar un programa no bloqueante utilizando timers y
callbacks

Tiempo aproximado: 15 minutos

CODER HOUSE



Asincronismo y callback

Desarrollar una función 'mostrarLetras' que reciba un string como primer parámetro, un entero como segundo parámetro y una función callback como tercer parámetro:

```
const fin = () =>  
  console.log('terminé')
```

Al ejecutarse debe imprimir en consola de forma secuencial cada letra del string recibido con diferencias de tiempo de 0, 1500 y 3000 mS (un valor por invocación) Al finalizar, debe invocar la función callback recibida.

Verificar que los mensajes de salida se intercalan.



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

Archivos



Introducción



- En todo sistema, es posible que nos topemos con la necesidad de que algunos **datos persistan más allá de la ejecución del programa**.
- Una de las opciones con las que contamos es el uso de archivos.
- Según el caso, existen ventajas y desventajas en utilizar el sistema de archivos como medio de almacenamiento de información.



Ventajas del uso de archivos



- Son fáciles de usar.
- No requieren el uso de programas externos para su creación, lectura o edición.
- En ocasiones, pueden ser abiertos y editados desde programas de edición de texto simples como un bloc de notas (¡siempre que se trate de texto!).
- Son fáciles de compartir o enviar a otros usuarios/programas.

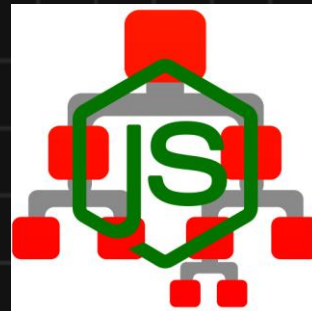
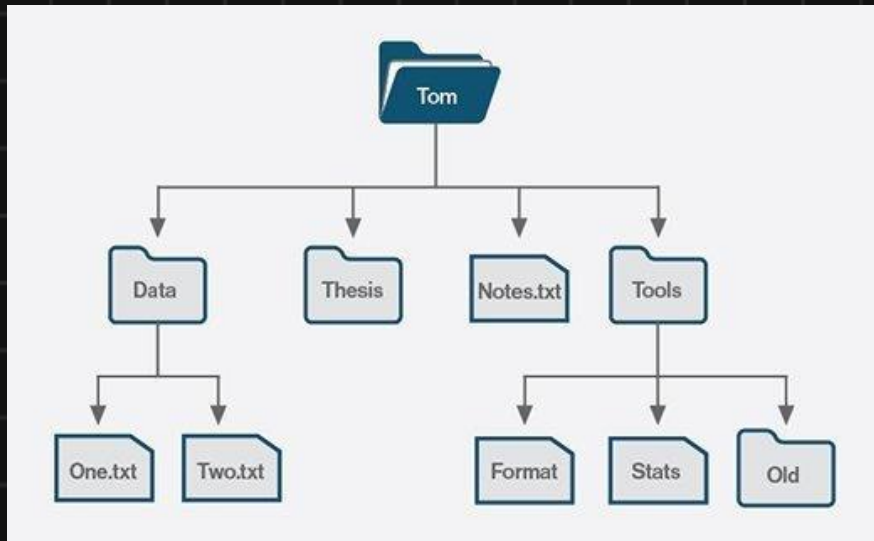


Desventajas del uso de archivos



- Consultas sobre algún dato puntual entre todos los datos almacenados (y no podemos guardar todo el lote de datos en memoria).
- Ediciones de datos puntuales (que no requieren sobrescribir el archivo por completo).
- Lecturas que combinen datos obtenidos de varios archivos (nuevamente, suponiendo que no podemos guardar todos los datos en memoria).
- Probablemente sea mejor considerar el uso de un motor de base de datos.

Manejo de Archivos en NodeJS



Módulo nativo file system: fs

- **fs** es la abreviatura en inglés para file system o sistema de archivos y es, además, uno de los módulos más básicos y útiles de Node.js.
- En Node.js es posible manipular archivos a través de fs (crear, leer, modificar, etc.).
- La mayoría de las funciones que contiene este módulo pueden usarse tanto de manera sincrónica como asincrónica.

Aclaración: Hay que tener en cuenta que esto sólo aplica a Node.js, desde el navegador no es posible manipular archivos dado que sería muy inseguro.

Uso de fs en nuestro código



Para poder usar este módulo solo debemos **importarlo** con la función *require* al comienzo de nuestro archivo fuente:

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

FS: modo sincrónico

Operaciones Sincrónicas



- Las funciones sincrónicas terminan con “Sync”
- Son **operaciones bloqueantes** que **devuelven un resultado**

Podemos listar algunas de ellas:

- ☐ **readFileSync**: lectura de un archivo en forma sincrónica
- ☐ **writeFileSync**: escritura de un archivo en forma sincrónica
- ☐ **appendFileSync**: actualización de un archivo en forma sincrónica
- ☐ **unlinkSync**: borrado de un archivo en forma sincrónica
- ☐ **mkdirSync**: creación de una carpeta

Leer un archivo



❑ *fs.readFileSync(path, encoding)*

```
const data = fs.readFileSync('./test-input-sync.txt', 'utf-8')
console.log(data)
```

- El **primer parámetro** es un **string** con la **ruta del archivo** que queremos **leer**
- El **segundo parámetro** indica el **formato de codificación de caracteres** con que fue escrito el dato que estamos leyendo
- El formato que utilizaremos con más frecuencia será **'utf-8'** (inglés: 8-bit Unicode Transformation Format, español: Formato de Codificación de caracteres Unicode).

Acerca de las rutas...

- Si la ruta comienza con un '' o './' se trata de una *ruta relativa*.
 - Supongamos que el programa se está ejecutando en la carpeta '/user/documents/workspace/proyecto/'
 - Si llamamos a alguna función con la ruta: './mi-archivo.txt' o 'mi-archivo.txt', estaremos en realidad leyendo la ruta: '/user/documents/workspace/proyecto/mi-archivo.txt'.
- Si la ruta, en cambio, comienza con '/', estaremos leyendo exactamente esa ruta.

Sobreescribir un archivo



❏ *fs.writeFileSync(ruta, datos) //sobreescribe archivo*

```
fs.writeFileSync('./test-output-sync.txt', 'ESTO ES UNA PRUEBA\n')
```

- El **primer parámetro** es un **string** con la **ruta del archivo** en el que queremos **escribir**
- El **segundo parámetro** indica lo que queremos escribir.
- La función admite un **tercer parámetro opcional** para indicar el **formato de codificación de caracteres** con que queremos escribir el texto: por defecto *'utf-8'*.
- Si la **ruta** provista fuera **válida**, pero el nombre de **archivo no existiera**, la función creará un **nuevo archivo** con el nombre provisto.

Agregar contenidos a un archivo



❏ `fs.appendFileSync(ruta, datos)` //agregar contenido a archivo

```
fs.appendFileSync('./test-output-sync.txt', 'ESTO ES UN AGREGADO\n')
```

- El **primer parámetro** es un **string** con la **ruta** del **archivo** al que le queremos **agregar contenidos**
- El **segundo parámetro** indica lo que queremos **agregar**.
- La función admite un **tercer parámetro opcional** para indicar el **formato de codificación de caracteres** con que queremos escribir el texto: por defecto `'utf-8'`.
- Si la **ruta** provista fuera **válida**, pero el **nombre de archivo no existiera**, la función creará un **nuevo archivo** con el nombre provisto

Borrar un archivo



❑ `fs.unlinkSync(ruta)`

```
fs.unlinkSync('./test-output-sync.txt')
```

El único **parámetro** es un **string** con la **ruta** del **archivo** que queremos borrar.



Manejo de errores

```
try {  
  const data = fs.readFileSync('/ruta/que/no/existe')  
} catch (err) {  
  console.log(err)  
}
```

Ante una situación de error, las excepciones se lanzan inmediatamente y se pueden manejar usando **try... catch**. Esta forma de capturar errores se puede utilizar en todas las funciones sincrónicas de acceso al sistema de archivos.



Fecha y hora

Vamos a practicar lo aprendido hasta ahora



Realizar un programa que:

- A) Guarde en un archivo llamado *fyh.txt* la fecha y hora actual.
- B) Lea nuestro propio archivo de programa y lo muestre por consola.
- C) Incluya el manejo de errores con try catch (progresando las excepciones con throw new Error).

Aclaración: utilizar las funciones sincrónicas de lectura y escritura de archivos del módulo fs de node.js

Tiempo: 5/10 minutos

***FS: modo asincrónico vía
Callbacks***

Introducción: fs con Callbacks



- Las funciones asincrónicas tiene el **mismo nombre** que sus versiones sincrónicas, pero **sin** la palabra “Sync” al final
- Son operaciones **no bloqueantes**
- **Reciben un nuevo último parámetro: un callback.**
- Los callbacks pueden recibir un primer parámetro destinado al error (si lo hubiere) para saber cómo manejarlo y un segundo parámetro, en caso de que la función en cuestión devuelva algún resultado, para indicar qué hacer con el mismo.
- Para **manejar los errores** que pueden surgir de su ejecución, **no será necesario** ejecutarlas utilizando **try / catch**.

Operaciones Asíncronas



Podemos listar algunas de ellas:

- ☐ **readFile**: lectura de un archivo en forma asíncrona
- ☐ **writeFile**: escritura de un archivo en forma asíncrona
- ☐ **appendFile**: actualización de un archivo en forma asíncrona
- ☐ **unlink**: borrado de un archivo en forma asíncrona
- ☐ **mkdir**: creación de una carpeta

Leer un archivo



❑ `fs.readFile(ruta, encoding, callback)`

```
fs.readFile('/ruta/al/archivo', 'utf-8', (error, contenido) => {  
  if (error) {  
    // hubo un error, no pude leerlo, hacer algo!  
  } else {  
    // en este punto del código, puedo acceder a todo el contenido  
    // del archivo a través de la variable "contenido".  
    console.log(contenido)  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback**. La función se encarga internamente de **abrir y cerrar el archivo** una vez finalizado su uso.

Sobreescribir un archivo



❏ `fs.writeFile(ruta, datos, callback)` //sobreescribe archivo

```
fs.writeFile('/ruta/al/archivo', 'TEXTO DE PRUEBA\n', error => {  
  if (error) {  
    // hubo un error, no pude sobreescribirlo, hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('guardado!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**. La función se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

Agregar contenidos a un archivo



❏ `fs.appendFile(ruta, datos, callback)` //agregar contenido a

```
fs.appendFile('/ruta/al/archivo', 'TEXTO A AGREGAR\n', error => {  
  if (error) {  
    // hubo un error, no pude agregarlo, hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('guardado!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**. La función se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

Borrar un archivo



❏ `fs.unlink(ruta, callback)`

```
fs.unlink(ruta, error => {  
  if (error) {  
    // hubo un error, no pude borrarlo, hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('borrado!')  
  }  
})
```

Recibe los **mismos parámetros** que su versión sincrónica, más el **callback** con un **parámetro** para **manejar** algún eventual **error**. La función se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

Otras funciones útiles

Crear una carpeta



❏ *fs.mkdir(ruta, callback)*

```
fs.mkdir(ruta, error => {  
  if (error) {  
    // hubo un error, no pude crear la carpeta! hacer algo!  
  } else {  
    // no hubo errores, hacer algo (opcional)  
    console.log('carpeta creada!')  
  }  
})
```

Recibe los mismos parámetros que su versión sincrónica, más el **callback** con un **parámetro** para manejar algún eventual error.

Esta función también se encuentra en su versión sincrónica (mkdirSync).

Leer el contenido de una carpeta



❑ `fs.readdir(ruta, callback)`

```
fs.readdir(ruta, (error, nombres) => {  
  if (error) {  
    // hubo un error, no pude leer la carpeta! hacer algo!  
  } else {  
    // hacer algo con los nombres!  
    console.log(nombres)  
  }  
})
```

Recibe los mismos parámetros que su versión sincrónica, más el **callback** con un **parámetro** para manejar algún eventual error.

Esta función también se encuentra en su versión sincrónica (readdirSync).



Lectura y escritura de archivos

Práctica del modo asincrónico

Tiempo: 10/15 minutos



Escribir un programa ejecutable bajo node.js que realice las siguientes acciones:

A) Abra una terminal en el directorio del archivo y ejecute la instrucción: `npm init -y`.

Esto creará un archivo especial (lo veremos más adelante) de nombre `package.json`

B) Lea el archivo `package.json` y muestre su formato y datos:

```
const info = {  
  contenidoStr: (contenido del archivo leído en formato string),  
  contenidoObj: (contenido del archivo leído en formato objeto),  
  autor: ''  
}
```

C) Muestre por consola el objeto info luego de leer el archivo

D) Guardar el objeto info en un archivo llamado info.txt dentro de la misma carpeta



Aclaraciones:

- Utilizar la lectura y escritura de archivos en modo asincrónico con callbacks.
- Consigna B): Para deserializar un string con contenido JSON utilizar `JSON.parse` (convierte string en object).
- Consigna C): Para serializar un objeto (convertirlo a string) y guardarlo en un archivo utilizar `JSON.stringify`.

Ayuda:

Para el Punto 3 considerar usar `JSON.stringify(info, null, 2)` para preservar el formato de representación del objeto en el archivo (2 representa en este caso la cantidad de espacios de indentación usadas al representar el objeto con **CODER HOUSE**).



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

FS: modo asincrónico vía Promesas

Introducción: fs con Promesas



- El módulo **fs** nos permite operar tanto de forma **sincrónica** como **asincrónica**.
- **fs** inicialmente ofrecía funciones que reciben un callback para manejar el asincronismo.
- En una **actualización** de este módulo se agregaron versiones de **funciones asincrónicas** que en lugar de recibir callbacks, **operan mediante promesas** con then/catch.
- Posteriormente se incluyó una **sintaxis simplificada** utilizando las nuevas palabras reservadas **“async”** y **“await”**.

Leer un archivo



`fs.promises.readFile(ruta, encoding)`



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

//Leo el archivo usando sintaxis then/catch
function leerTC() {
  fs.promises.readFile('/ruta/al/archivo', 'utf-8')
    .then( contenido => {
      console.log(contenido)
    })
    .catch( err => {
      // hubo un error, no pude leerlo, hacer algo!
      console.log('Error de lectura!',err)
    })
}
leerTC()

//Leo el archivo usando sintaxis async/await
async function leerAA() {
  try {
    const contenido = await fs.promises.readFile('/ruta/al/archivo', 'utf-8')
    console.log(contenido)
  }
  catch (err) {
    // hubo un error, no pude leerlo, hacer algo!
    console.log('Error de lectura!',err)
  }
}
leerAA()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

¡Aclaraciones!

- En el caso de querer hacer algo con la variable **fuera** del **bloque try/catch**, la **declaración** debería hacerse **fuera** del mismo.
- Recordar que debemos anteponer la palabra “**await**” al llamado a la función para que ésta se comporte de manera bloqueante.
- Si se omitiera la palabra “**await**” la instrucción *console.log(contenido)* se ejecutaría ANTES de que a la variable contenido se le asigne el resultado de la operación de lectura del archivo.
- Recordar también que la palabra “**await**” puede usarse ÚNICAMENTE dentro de una función de tipo “**async**” Dado que estas funciones ya no poseen un parámetro que nos permite elegir cómo manejar los errores que pueden surgir de su ejecución, vuelve a ser necesario ejecutarlas utilizando **try / catch**

Sobreescribir un archivo



❏ *fs.promises.writeFile(ruta, datos)*

```
async function escribir() {  
  try {  
    await fs.promises.writeFile('/ruta/al/archivo', 'TEXTO DE PRUEBA\n')  
    console.log('guardado!')  
  }  
  catch (err) {  
    // hubo un error, no pude escribirlo, hacer algo!  
  }  
}  
escribir()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

Agregar contenidos a un archivo



❏ `fs.promises.appendFile(ruta, datos)`

```
async function agregar() {  
  try {  
    await fs.promises.appendFile('/ruta/al/archivo', 'TEXTO DE PRUEBA\n')  
    console.log('agregado!')  
  }  
  catch (err) {  
    // hubo un error, no pude escribirlo, hacer algo!  
  }  
}  
agregar()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.

Renombrar un archivo



❏ *`fs.promises.rename(rutaVieja, rutaNueva)`*

```
async function renombrar(rutaVieja, rutaNueva) {  
  try {  
    await fs.promises.rename(rutaVieja, rutaNueva)  
    console.log('renombrado!')  
  }  
  catch (err) {  
    // hubo un error, no pude escribirlo, hacer algo!  
  }  
}  
renombrar()
```

Esta función recibe los mismos parámetros que su versión sincrónica y se encarga internamente de abrir y cerrar el archivo una vez finalizado su uso.



Lectura y escritura con promises

Tiempo: 5/10 minutos



Realizar un programa que ejecute las siguientes tareas:

- A) Lea el archivo info.txt generado en el desafío anterior deserializándolo en un objeto llamado info.
- B) Mostrar este objeto info en la consola.
- C) Modifique el author a "Coderhouse" y guarde el objeto serializado en otro archivo llamado package.json.coder
- D) Mostrar los errores por consola.



Aclaraciones:

Trabajar con fs.promises (then/catch).

Ayuda:

Para el punto 3 considerar usar `JSON.stringify(info.contenidoObj, null, 2)` para preservar el formato de representación del objeto en el archivo.



MANEJO DE ARCHIVOS

Manejo de archivos

Formato: carpeta comprimida con el proyecto.

Desafío
entregable



>> Consigna: Implementar programa que contenga una clase llamada Contenedor que reciba la ruta del archivo con el que va a trabajar e implemente los siguientes métodos:

- `save(Object): Number` - Recibe un objeto, lo guarda en el archivo, devuelve el id asignado.
- `getId(Number): Object` - Recibe un id y devuelve el objeto con ese id, o null si no está.
- `getAll(): Object[]` - Devuelve un array con los objetos presentes en el archivo.
- `deleteById(Number): void` - Elimina del archivo el objeto con el id buscado.
- `deleteAll(): void` - Elimina todos los objetos presentes en el archivo.

Manejo de archivos

Formato: carpeta comprimida con el proyecto.

Sugerencia: usar un archivo para la clase y otro de test, que la importe

Desafío
entregable



>> Aspectos a incluir en el entregable:

- El método save incorporará al producto un id numérico, que deberá ser siempre uno más que el id del último objeto agregado (o id 1 si es el primer objeto que se agrega) y no puede estar repetido.
- Tomar en consideración el contenido previo del archivo, en caso de utilizar uno existente.
- Implementar el manejo de archivos con el módulo fs de node.js, utilizando promesas con async/await y manejo de errores.
- Probar el módulo creando un contenedor de *productos*, que se guarde en el archivo: “productos.txt”
- Incluir un llamado de prueba a cada método, y mostrando por pantalla según corresponda para verificar el correcto funcionamiento
- El formato de cada producto será :

```
{  
  title: (nombre del producto),  
  price: (precio),  
  thumbnail: (url de la foto del producto)  
}
```

Manejo de archivos

Formato: carpeta comprimida con el proyecto.

Desafío
entregable



>> Ejemplo:

Contenido de "productos.txt" con 3 productos almacenados

```
[
  {
    title: 'Escuadra',
    price: 123.45,
    thumbnail: 'https://cdn3.iconfinder.com/data/icons/education-209/64/ruler-triangle-stationary-school-256.png',
    id: 1
  },
  {
    title: 'Calculadora',
    price: 234.56,
    thumbnail: 'https://cdn3.iconfinder.com/data/icons/education-209/64/calculator-math-tool-school-256.png',
    id: 2
  },
  {
    title: 'Globo Terráqueo',
    price: 345.67,
    thumbnail: 'https://cdn3.iconfinder.com/data/icons/education-209/64/globe-earth-geograhpy-planet-school-256.png',
    id: 3
  }
]
```

CODER HOUSE

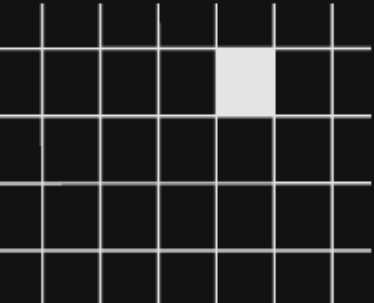
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- Funciones
 - Callbacks
 - Promesas
 - Ejecución sincrónica/asincrónica
 - Manejo de archivos en Node.js
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDOLAEDUCACIÓN