



¡Les damos la bienvenida!

¿Comenzamos?

Certificados oficialmente por



CODERHOUSE

Esta clase va a ser

- grabada
a

Clase 04. PROGRAMACIÓN BACKEND

Manejo de archivos en Javascript

Certificados oficialmente por



CODERHOUSE

Temario

03

Programación sincrónica y asincrónica

- ✓ Funciones en Javascript
- ✓ Callbacks
- ✓ Promesas
- ✓ Sincronismo vs. Asincronismo

04

Manejo de archivos en Javascript

- ✓ [Archivos
síncronos](#)
- ✓ [Manejo de
archivos](#)
- ✓ [File system](#)
- ✓ [Promesas](#)

05

Administradores de paquetes - NPM

- ✓ Node Js
- ✓ Módulos
nativos
- ✓ NPM
- ✓ Instalaciones
globales y
locales

Objetivos de la clase

- Utilizar la programación sincrónica y asincrónica y aplicarla en el uso de archivos
- Conocer el módulo nativo de Nodejs para interactuar con los archivos.
- Conocer la utilización de archivos con callbacks y promesas
- Conocer las ventajas y desventajas del FileSystem, así también como ejemplos prácticos.

Glosario

Función definida: Función que se declara con un nombre desde el inicio. Usualmente se usan para tareas específicas y no suelen reasignarse.

Función anónima: Función que se declara sin tener un nombre, sino que se asigna a una variable desde el inicio. Se suele utilizar para reasignaciones o para **callbacks**.

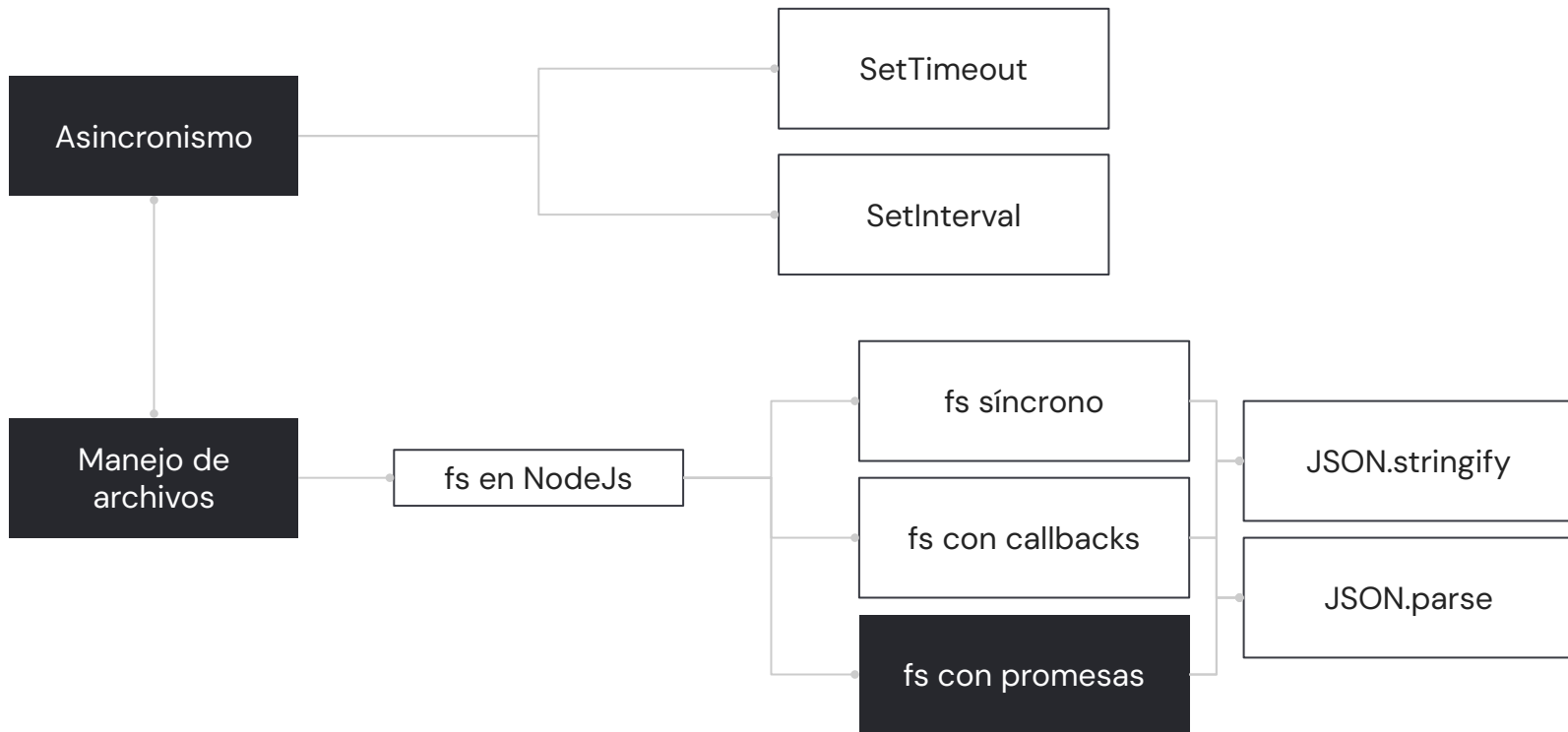
Callback: Bloque de código que encapsula una o más instrucciones, para ser utilizadas en cualquier otro momento del programa

Promesa: Operación **asíncrona** que tiene dos canales de salida: resolverse o rechazarse. Permiten mejor control que los callbacks

Operación síncrona: Operación **bloqueante** en la cual, una tarea no puede comenzar hasta que haya finalizado la otra tarea.

Operación asíncrona: Operación **no bloqueante** en la cual se pueden iniciar múltiples tareas independientemente de que no hayan finalizado las tareas previas.

MAPA DE CONCEPTOS





PARA RECORDAR

Sincronismo

Las operaciones síncronas o bloqueantes, nos sirven cuando necesitamos que las operaciones se ejecuten **una detrás de otra**, es decir, se utiliza cuando **deseamos que las tareas sean secuenciales, independientemente del tiempo que demore cada operación.**



PARA RECORDAR

Asincronismo

Las operaciones asíncronas o **no bloqueantes**, nos sirven cuando necesitamos que haya múltiples tareas ejecutándose, sin tener que esperar a las tareas que ya se están ejecutando. **Úsalas cuando necesites hacer alguna operación, sin afectar al flujo principal.**

Ejemplos de asincronismo: **setTimeout y setInterval**

setTimeout

setTimeout se utiliza para establecer un temporizador que ejecute una tarea después de un determinado tiempo. permite entender en un par de líneas la idea del asincronismo.

A diferencia de una operación síncrona, podremos notar como setTimeout inicia su ejecución, y una vez que haya transcurrido el tiempo, veremos el resultado, **aun cuando el resto de las operaciones hayan terminado.**

Ejemplo de uso de setTimeout

JS sincrono.js X

JS sincrono.js

```
1 //Ejemplo de operación síncrona
2 console.log("¡Iniciando tarea!");
3 console.log("Realizando operación");
4 console.log("Continuando operación");
5 console.log("¡Tarea finalizada!");
6 //Orden de salida:
7 /**
8  * ¡Iniciando tarea!
9  * Realizando operación
10 * Continuando operación
11 * ¡Tarea finalizada!
12 */
13
14 //Hasta aquí todo en orden, una va detrás de otra.
15 //¿Qué pasa con una operación asíncrona?
```

JS setTimeout.js X

JS setTimeout.js > ...

```
1 const temporizador = (callback) =>{
2   |   setTimeout(()=>{
3   |     |   callback();
4   |     |   },5000)
5   |   }
6
7 let operacion = () => console.log("Realizando operación");
8
9 console.log("¡Iniciando tarea!");
10 temporizador(operacion); //Metemos la "operacion" al temporizador
11 console.log("¡Tarea finalizada!")
12
13 /**
14  * Orden de salida:
15  *
16  * ¡Iniciando tarea!
17  * ¡Tarea finalizada!
18  * Realizando operación
19  *
20  *
21  * La tarea "operación" tuvo que esperar 5000 milisegundos (5 segundos)
22  * para poder ejecutarse, pero al ser asíncrono, el programa pudo continuar
23  * y pudo finalizar sin esperar dicha operación
24  */
25
```

setInterval

setInterval funciona como setTimeout, la diferencia radica en que éste reiniciará el conteo y ejecutará la tarea nuevamente cada vez que se cumpla dicho intervalo de tiempo.

Un timer devuelve un **apagador** el cual permite detener el intervalo cuando se cumpla cierta operación.

Suele utilizarse mucho para poner tiempos límites en alguna página para llenar formularios (Hay ciertas páginas que te dan tiempo límite para hacer la operación, O TE BOTAN).

Ejemplo de uso de setInterval

JS sincrono.js X

JS sincrono.js > ...

```
1 //Ejemplo de operación síncrona
2
3 console.log("¡Iniciando tarea!");
4 console.log("Realizando operación")
5 for(let contador = 1;contador<=5;contador++){
6   console.log(contador);
7 }
8 console.log("¡Tarea finalizada!")
9 //Orden de salida:
10 /**
11  * ¡Iniciando tarea!
12  * Realizando operación
13  * 1
14  * 2
15  * 3
16  * 4
17  * 5
18  * ¡Tarea finalizada!
19  */
20
21
22 /**
23  * Nuevamente, todo parece normal, la tarea finaliza hasta
24  * que el ciclo haya terminado de contar del 1 al 5
25  * ¿Cómo funcionará asíncrono con intervalos?
```

JS setInterval.js X

JS setInterval.js > [🔍] contador

```
1 //Ejemplo con setInterval
2 let contador = () =>{
3   let counter=1;
4   console.log("Realizando operación");
5   let timer = setInterval(()=>{
6     console.log(counter++);
7     if(counter>5){
8       clearInterval(timer); //Se después de contar 5
9     }
10  },1000)
11  /**
12   * Al ser un intervalo, el console.log(counter++) se ejecutará
13   * cada 1000 milisegundos (1 segundo)
14   */
15 }
16 console.log("¡Iniciando tarea!");
17 contador();
18 console.log('¡Tarea finalizada!')
19 /**
20  * Orden de salida:
21  * ¡Iniciando tarea!
22  * Realizando operación
23  * ¡Tarea finalizada!
24  * 1 (aquí pasa 1 segundo)
25  * 2 (aquí pasa 1 segundo)
26  * 3 (aquí pasa 1 segundo)
27  * 4 (aquí pasa 1 segundo)
28  * 5
29  */
```

Más allá de la memoria...

manejo de archivos

El problema: persistencia en memoria

Cuando comenzamos a manejar más información, nos encontraremos con una de las grandes molestias del programador: tener que comenzar desde 0 cada vez que el programa termina su ejecución.

Todas las cosas que creamos, movimos o trabajamos desaparecen, ya que **sólo persiste en memoria**, y esta se borra automáticamente al finalizar el programa.



El mundo real no te permite eso

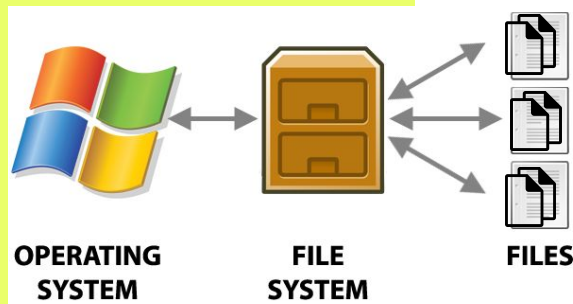
Si tratara de registrar usuarios para la página web de mi empresa ¡imagina si tuviera que hacer una actualización de la página!

Al actualizar la página (el servidor), necesitaría reiniciarse Y **todos los usuarios desaparecerían**. Tendríamos que pedirle a cada usuario que vuelva a registrarse y a subir su información.



Usuario volviendo a nuestra página buscando su información

¡La solución!



La **primera** solución al problema de persistencia en memoria fue guardar la información en **archivos**. Estos son un conjunto de información que podemos **almacenar**.

Así, cuando se requiera nuevamente la información, podemos leer el archivo que habíamos guardado y recuperar la información, aún si el programa había finalizado.

¡Tu Sistema Operativo funciona así!

Implementando archivos en nodejs: fs

fs en Nodejs

fs es la abreviación utilizada para FileSystem, el cual, como indica el nombre, es un sistema de manejador de archivos que nos proporcionará node para poder crear, leer, actualizar o eliminar un archivo, sin tener que hacerlo nosotros desde cero.

Así, crear un archivo con contenido será tan fácil como escribir un par de líneas de código, en lugar de tener que lidiar con los datos binarios y transformaciones complejas y de un nivel más bajo en la computadora.

¡Importante!

Contempla que el curso utilizará muchas soluciones “ya previamente desarrolladas”. La programación se basa en un principio clave: **no reinventes la rueda**. Los módulos a utilizar son soluciones ya proporcionadas por otros desarrolladores, con el fin de concentrarnos en la solución de problemas más específicos.

¿Cómo utilizamos el File
System de NodeJs en
nuestro propio código?

Utilizando fs

fs existe desde el momento en el que instalamos Nodejs en nuestro computador, por lo que, para utilizarlo, podemos llamarlo desde cualquier archivo que tengamos de nuestro código con la siguiente línea:

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

De ahí en adelante todo el módulo de FileSystem estará contenido en la variable fs. Sólo debemos utilizarlo llamando sus métodos como una clase. Esto podremos hacerlo de 3 formas: **síncrono**, con **callbacks** o con **promesas**.

Usando fs de manera síncrona

fs síncrono

¡El uso de fs de manera síncrona es bastante sencillo! para ello, sólo utilizaremos la palabra Sync después de cada operación que queramos realizar. Hay muchas operaciones para trabajar con archivos, pero sólo abarcaremos las principales.

Las principales operaciones que podemos hacer con fs síncrono son:

- ✓ **writeFileSync** = Para escribir contenido en un archivo. Si el archivo no existe, lo crea. Si existe, lo sobrescribe.

- ✓ **readFileSync** = Para obtener el contenido de un archivo.

- ✓ **appendFileSync** = Para añadir contenido a un archivo. ¡No se sobrescribe!

- ✓ **unlinkSync** = Es el “delete” de los archivos. eliminará todo el archivo, no sólo el contenido.

- ✓ **existsSync** = Corrobora que un archivo exista!



Ejemplo 1

- ✓ Se profundizará sobre la sintaxis de las operaciones síncronas de archivos con fs.

Ejemplo de uso de fs en su modo síncrono

```
EXPLORER  ...  JS fsSyncjs  x  ejemplo.txt

v OPEN EDITORS
  x JS fsSyncjs
    ejemplo.txt
v EJEMPLO ARCHIVO
  ejemplo.txt
  JS fsSyncjs

JS fsSyncjs > ...
1  const fs = require('fs');
2  //Como comentamos en las diapositivas, fs nos permitirá acceder a las operaciones para archivos
3
4  fs.writeFileSync('./ejemplo.txt','¡Hola, Coders, estoy en un archivo!')
5  /**
6   * Primer operación: para escribir un archivo, el primer argumento/parámetro es la ruta y
7   * nombre del archivo sobre el que queremos trabajar. El segundo argumento es el contenido
8   * ¡Súper sencillo!
9   */
10 if(fs.existsSync('./ejemplo.txt')){//existsSync devuelve true si el archivo sí existe, y false si el archivo no existe
11   let contenido = fs.readFileSync('./ejemplo.txt','utf-8')
12   /**
13    * readFileSync lee el contenido del archivo, es importante que en el segundo parámetro coloquemos el tipo de
14    * codificación que utilizaremos para leerlo. En este curso sólo abarcaremos utf-8
15    */
16   console.log(contenido) //El resultado será lo que escribimos arriba en la línea 4: "¡Hola Coders, estoy en un archivo!"
17   fs.appendFileSync('./ejemplo.txt',' Más contenido')
18   /**
19    * appendFileSync buscará primero la ruta del archivo, si no encuentra ningún archivo, lo creará, en caso de sí
20    * encontrarlo, en lugar de sobrescribir todo el archivo, sólo colocará el contenido al final.
21    */
22   contenido = fs.readFileSync('./ejemplo.txt','utf-8')
23   //Volvemos a leer el contenido del archivo.
24   console.log(contenido);
25   //Esta vez el contenido será: "¡Hola, Coders, estoy en un archivo! Más contenido" esto gracias al appendFileSync.
26
27   fs.unlinkSync('./ejemplo.txt');
28   //Por último, esta línea de código eliminará el archivo, independientemente de todo el contenido que éste tenga.
29
```

fs con Callbacks

fs con callbacks

Funciona muy similar a las operaciones síncronas. Sólo que al final recibirán un último argumento, que como podemos intuir, **debe ser un callback**. Según lo vimos en las convenciones de callbacks de la clase pasada, el primer argumento suele ser un error.

Esto permite saber si la operación salió bien, o si salió mal. Sólo **readFile** maneja un segundo argumento, con el resultado de la lectura del archivo.

Por último: el manejo por callbacks es totalmente **asíncrono**, así que cuidado dónde lo usas.

fs con callbacks

Las principales operaciones que podemos hacer con fs con callbacks son:

- ✓ **writeFile** = Para escribir contenido en un archivo. Si el archivo no existe, lo crea. Si existe, lo sobrescribe. Al sólo escribir, su callback sólo maneja: **(error)=>**
- ✓ **readFile** = Para obtener el contenido de un archivo. Como pide información, su callback es de la forma: **(error, resultado)=>**

- ✓ **appendFile** = Para añadir contenido a un archivo. ¡No se sobrescribe!, al sólo ser escritura, su callback sólo maneja: **(error)=>**
- ✓ **unlink** = Es el “delete” de los archivos. eliminará todo el archivo, no sólo el contenido. Al no retornar contenido, su callback sólo es **(error)=>**



Ejemplo 2

- ✓ Se realizará el mismo procedimiento del ejemplo 1, haciendo énfasis en los callbacks y cómo se manejan.

Ejemplo de uso de fs con callbacks

EXPLORER

JS fsCallback.js X

Lo mismo ¡pero muy diferente!

```
1 const fs = require('fs'); //Volvemos a utilizar fs, sin él, no podremos trabajar con archivos.
2 fs.writeFile('./ejemploCallback.txt','Hola desde Callback',(error)=>{
3     /**
4      * Notemos que la operación es similar, el detalle es que ahora estamos metiendo un callback para preguntar si algo
5      * salió mal durante la operación de escritura del archivo.
6      */
7     if(error) return console.log('Error al escribir el archivo') //Pregunto si el "error" del callback existe.
8     fs.readFile('./ejemploCallback.txt','utf-8',(error,resultado)=>{
9         /**
10          * Los mismos argumentos del readFileSync, sólo que esta vez al final colocamos un callback, donde el primer
11          * argumento/parámetro sirve para saber si hubo algún error al leer el archivo, el segundo argumento es el
12          * resultado de esa lectura.
13          */
14         if(error) return console.log('Error al leer el archivo') //¡Nota que cada callback es consciente de su error!
15         console.log(resultado)//En caso de no haber error, el resultado será: 'Hola desde Callback'
16         fs.appendFile('./ejemploCallback.txt',' Más contenido',(error)=>{
17             /**
18              * Hasta este punto debes estar preocupándote... ¿Acaso estoy armando un callback Hell?
19              * ¡Mucho cuidado cuando trabajes con callbacks y con archivos!
20              */
21             if(error) return console.log('Error al actualizar el archivo') //Preguntamos si hubo error en el append.
22             fs.readFile('./ejemploCallback.txt','utf-8',(error,resultado)=>{
23                 /**
24                  * Volvemos a leer el archivo, para corroborar el nuevo cambio.
25                  */
26                 if(error) return console.log("Error al leer el archivo")
27                 console.log(resultado) //Si todo salió bien, debe mostrar "Hola desde Callback Más contenido"
28                 fs.unlink('./ejemploCallback.txt',(error)=>{
29                     if(error)return console.log('No se pudo eliminar el archivo');
30                 })
31             })
32         })
33     })
34 })
```


¡Importante!

Recuerda que los callbacks son peligrosos si necesitas encadenar múltiples tareas. Si necesitas hacer operaciones muy complejas con archivos, no uses callbacks o terminarás en un ***CALLBACK HELL***



Almacenar fecha y hora

Práctica para repasar los conceptos de archivos con
callbacks

Duración: 5-10 min



ACTIVIDAD EN CLASE

Almacenar fecha y hora

- ✓ Realizar un programa que cree un archivo en el cual escriba la fecha y la hora actual. Posteriormente leer el archivo y mostrar el contenido por consola.
- ✓ Utilizar el módulo fs y sus operaciones de tipo callback.



Break

¡10 minutos y volvemos!

fs utilizando promesas

fs con promesas

Ya sabemos trabajar con archivos, ya vimos cómo trabajarlos de manera asíncrona, ahora viene el punto más valioso: **trabajar con archivos de manera asíncrona, con promesas**. esto lo haremos con su propiedad **fs.promises**

JAVASCRIPT
PROMISES



fs.promises

Al colocar a nuestro módulo fs el **.promises** estamos indicando que, la operación que se hará debe ser ejecutada de manera **asíncrona**, pero en lugar de manipularla con un callback, lo podemos hacer con `.then` `+.catch`, o bien con `async/await`.
Los argumentos y estructura es casi idéntico al síncrono, por lo tanto sus operaciones principales serán:

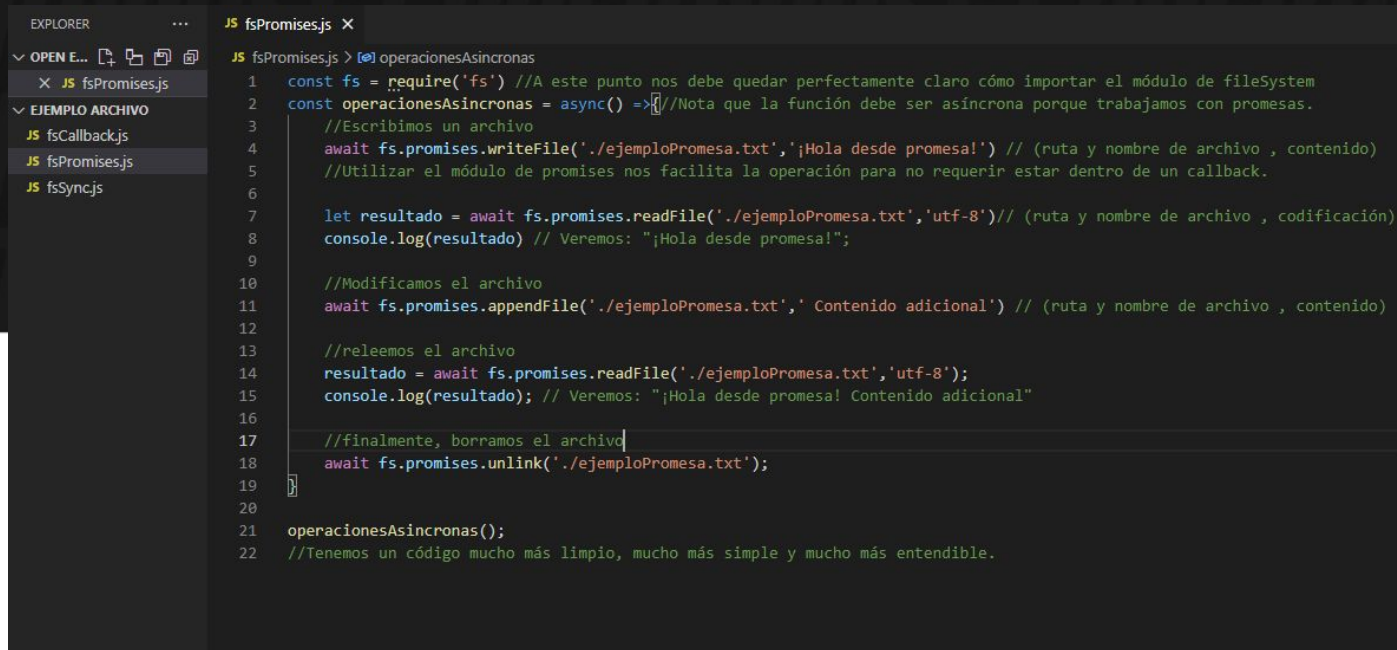
- ✓ **fs.promises.writeFile** = Para escribir contenido en un archivo. Si el archivo no existe, lo crea. Si existe, lo sobrescribe.
- ✓ **fs.promises.readFile** = Para obtener el contenido de un archivo.
- ✓ **fs.promises.appendFile** = Para añadir contenido a un archivo. ¡No se sobrescribe!
- ✓ **fs.promises.unlink** = Es el “delete” de los archivos. eliminará todo el archivo, no sólo el contenido.



Ejemplo 3

- ✓ Se realizará el mismo procedimiento que los ejemplos 1 y 2, pero trabajando fs con su submódulo *promises*. La implementación será con `async/await`

Ejemplo de fs con promesas usando async/await



```
1  const fs = require('fs') //A este punto nos debe quedar perfectamente claro cómo importar el módulo de fileSystem
2  const operacionesAsincronas = async() =>{//Nota que la función debe ser asíncrona porque trabajamos con promesas.
3      //Escribimos un archivo
4      await fs.promises.writeFile('./ejemploPromesa.txt','¡Hola desde promesa!') // (ruta y nombre de archivo , contenido)
5      //Utilizar el módulo de promises nos facilita la operación para no requerir estar dentro de un callback.
6
7      let resultado = await fs.promises.readFile('./ejemploPromesa.txt','utf-8')// (ruta y nombre de archivo , codificación)
8      console.log(resultado) // Veremos: "¡Hola desde promesa!";
9
10     //Modificamos el archivo
11     await fs.promises.appendFile('./ejemploPromesa.txt',' Contenido adicional') // (ruta y nombre de archivo , contenido)
12
13     //releemos el archivo
14     resultado = await fs.promises.readFile('./ejemploPromesa.txt','utf-8');
15     console.log(resultado); // Veremos: "¡Hola desde promesa! Contenido adicional"
16
17     //finalmente, borramos el archivo
18     await fs.promises.unlink('./ejemploPromesa.txt');
19
20
21     operacionesAsincronas();
22     //Tenemos un código mucho más limpio, mucho más simple y mucho más entendible.
```

Manejo de datos complejos con `fs.promise`

Manejo de datos complejos

Como ya podrás imaginar, no todo son archivos .txt, y por supuesto que no todo es una cadena de texto simple. ¿Qué va a pasar cuando queramos guardar el contenido de una variable, aun si esta es un objeto? ¿Y si es un arreglo? Normalmente los archivos que solemos trabajar para almacenamiento, son los archivos de tipo **json**.

Para poder almacenar elementos más complejos, nos apoyaremos del elemento `JSON.stringify()` y `JSON.parse()`

```
▼ {  
  "id": 1001,  
  "type": "donut",  
  "name": "Cake",  
  "description": "https://www.jqueryscript.net",  
  "price": 2.55,  
  ▶ "available": { 2 items },  
  ▼ "topping": [  
    ▼ {  
      "id": 5001,  
      "type": "None"  
    },  
    ▼ {  
      "id": 5002,  
      "type": "Glazed"  
    }  
  ]  
}
```

Así se ve un JSON. Muy similar a un objeto, ¡pero no lo es!

JSON.stringify

Una vez que tenemos el objeto que queremos guardar en el archivo, tenemos que recordar que éste no puede guardarse sólo incrustándolo. **Necesitamos convertirlo a formato json**, el cual es un formato estándar de guardado y envío de archivos.

La sintaxis para hacer la conversión es:

```
JSON.stringify(objetoAConvertir, replacer, '\t')
```

JSON.parse

Ahora que entendemos cómo se convierte un objeto a un JSON, es claro mencionar que JSON.parse representa la operación contraria. **Cuando leemos un archivo, el contenido no es manipulable**, así que, para recuperar el objeto que había guardado y no sólo una string representativa de él, entonces hay que transformarlo de vuelta, esto se hace con JSON.parse su sintaxis es:

```
JSON.parse(json_que_quiero_transformar_a_objeto)
```



Lectura y escritura de archivos

Duración: 10–15 min



ACTIVIDAD EN CLASE

Lectura y escritura de archivos

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

- ✓ 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*
- ✓ Lea el archivo *package.json* y declare un objeto con el siguiente formato y datos:

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



ACTIVIDAD EN CLASE

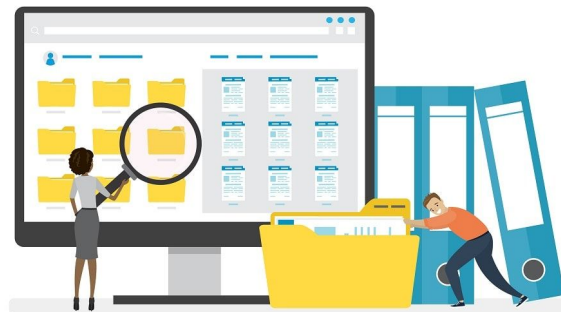
Lectura y escritura de archivos

- ✓ Muestre por consola el objeto info luego de leer el archivo
- ✓ Guardar el objeto info en un archivo llamado info.json dentro de la misma carpeta de package.json
- ✓ Incluir el manejo de errores (con throw new Error)
- ✓ Utilizar el módulo promises de fs dentro de una función async/await y utilizar JSON.stringify + JSON.parse para poder hacer las transformaciones json-→objeto y viceversa

Ventajas y desventajas de utilizar archivos

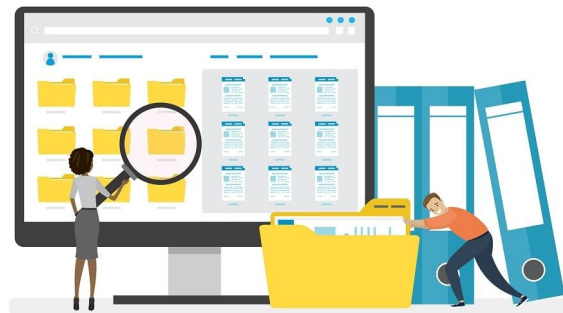
¿Por qué usarlos?

- ✓ Son excelentes para empezar a aprender persistencia, ya que son muy fáciles de usar
- ✓ Al ser nativo de node js, no tenemos que hacer instalaciones externas.
- ✓ Es muy fácil de manipular dentro o fuera de nuestro programa, además de se transferible.



Desventajas

- ✓ Conforme la información crece, nos daremos cuenta que, para querer modificar una sola cosa, necesitamos leer todo el archivo, lo cual consume recursos importantes.
- ✓ Similar al punto anterior, una vez modificado un dato puntual del archivo, tengo que **reescribir el archivo** completamente, lo cual es un proceso innecesario y pesado cuando la información es grande.
- ✓ Al final, puede ser peligroso tener toda la información en un archivo fácilmente extraíble con un drag&drop a otra carpeta.





Para pensar

Una vez vistas las ventajas y desventajas de su uso, toca preguntarnos **¿en el desarrollo web es utilizado, o es algo más propio sólo de Sistemas Operativos?**



Hands on lab

En esta instancia de la clase **ahondaremos sobre creación de promesas y el uso de async await** con un ejercicio práctico

¿De qué manera?

El profesor demostrará cómo hacerlo y tú lo puedes ir replicando en tu computadora. Si surgen dudas las puedes compartir para resolverlas en conjunto de la mano de los tutores.

Tiempo estimado: **15 minutos**

Manager de usuarios

¿Cómo lo hacemos? **Se creará una clase que permita gestionar usuarios usando fs.promises, éste deberá contar sólo con dos métodos: Crear un usuario y consultar los usuarios guardados.**

- ✓ El Manager debe vivir en una clase en un archivo externo llamado ManagerUsuarios.js
- ✓ El método "Crear usuario" debe recibir un objeto con los campos:
 - Nombre
 - Apellido
 - Edad
 - Curso

El método debe guardar un usuario en un archivo "Usuarios.json", **deben guardarlos dentro de un arreglo, ya que se trabajarán con múltiples usuarios**

- ✓ El método "ConsultarUsuarios" debe poder leer un archivo Usuarios.json y devolver el arreglo correspondiente a esos usuarios



Manejo de archivos

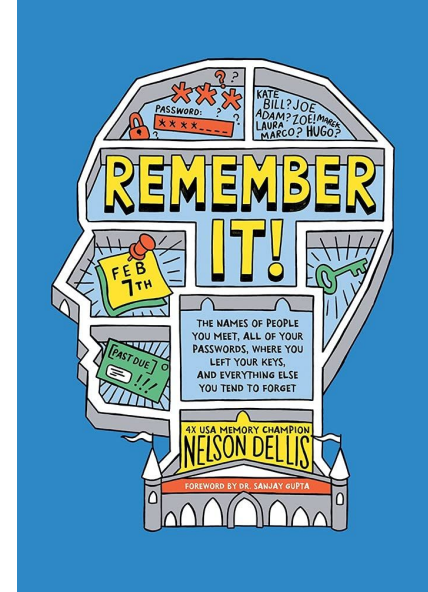
Agregamos `fileSystem` para cambiar el modelo de persistencia actual

Recordemos...

Basado en el entregable
de la clase 2

Estructuramos nuestra primera
clase

Agregamos los métodos necesarios
a nuestra clase para trabajar con un
arreglo de productos



Ahora... agregamos `fileSystem` para cambiar el modelo de persistencia actual



Manejo de archivos

Consigna

- ✓ Realizar una clase de nombre "ProductManager", el cual permitirá trabajar con múltiples productos. Éste debe poder agregar, consultar, modificar y eliminar un producto y manejarlo en persistencia de archivos (basado en entregable 1).
- ✓ Debe guardar objetos con el siguiente formato:
 - id (se debe incrementar automáticamente, no enviarse desde el cuerpo)
 - title (nombre del producto)
 - description (descripción del producto)
 - price (precio)
 - thumbnail (ruta de imagen)
 - code (código identificador)
 - stock (número de piezas disponibles)

Aspectos a incluir

- ✓ La clase debe contar con una variable `this.path`, el cual se inicializará desde el constructor y debe recibir la ruta a trabajar desde el momento de generar su instancia.



DESAFÍO ENTREGABLE

Aspectos a incluir

- ✓ Debe tener un método `addProduct` el cual debe recibir un objeto con el formato previamente especificado, asignarle un id autoincrementable y guardarlo en el arreglo (recuerda siempre guardarlo como un array en el archivo).
- ✓ Debe tener un método `getProducts`, el cual debe leer el archivo de productos y devolver todos los productos en formato de arreglo.
- ✓ Debe tener un método `getProductById`, el cual debe recibir un id, y tras leer el archivo, debe buscar el producto con el id especificado y devolverlo en formato **objeto**

- ✓ Debe tener un método `updateProduct`, el cual debe recibir el id del producto a actualizar, así también como el campo a actualizar (puede ser el objeto completo, como en una DB), y debe actualizar el producto que tenga ese id en el archivo.
NO DEBE BORRARSE SU ID
- ✓ Debe tener un método `deleteProduct`, el cual debe recibir un id y debe eliminar el producto que tenga ese id en el archivo.

Formato del entregable

- ✓ Archivo de javascript con el nombre `ProductManager.js`

Proceso de testing de este entregable ✓

¿Preguntas?

Resumen de la clase hoy

- ✓ Repaso rápido de asincronismo
- ✓ Archivos síncronos
- ✓ Archivos con callbacks
- ✓ Archivos con promesas

Opina y valora
esta clase

Muchas gracias.

#DemocratizandoLaEducación