



**¡Les damos la
bienvenida!**

¿Comenzamos?

Unidad 03. Programación Backend

Desarrollo Avanzado de Backend con Node.js

Objetivos de la clase

- Repasar qué es Node js y su uso en el backend
- Entender la diferencia entre un módulo nativo y uno de terceros
- Conocer la función de NPM y el proceso de instalación de dependencias
- Conocer el proceso de actualización de dependencias.

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.



Proyecto de node

Actividad de repaso para fortalecer conceptos de node y javascript

Duración: 10 min



ACTIVIDAD EN CLASE

Proyecto de node

- ✓ Crear un proyecto de node que genere 10000 números aleatorios en un rango de 1 a 20. Indicar por consola la finalización de esta operación con un mensaje.
- ✓ Mediante el uso de Promesas, crear un objeto cuyas claves sean los números salidos y el valor asociado a cada clave será la cantidad de veces que salió dicho número. Representar por consola los resultados.

Nota: Considerar que esta operación debe realizarse de forma asíncrona.



Hands on lab

En esta instancia de la clase **ahondaremos sobre el uso de los módulos nativos de Node js** con un ejercicio práctico

¿De qué manera?

El profesor mostrará 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: **20 minutos – 25 minutos**

Práctica de módulo nativo: crypto

¿Cómo lo hacemos? **Se creará una clase “UsersManager” que permitirá guardar usuarios en un atributo estático. El usuario se recibirá con una contraseña en string plano, y se deberá guardar la contraseña hasheada con crypto. Utilizar el módulo nativo crypto.**

El manager debe contar con los siguientes métodos:

- ✓ El método “Crear usuario” debe recibir un objeto con los campos:
 - Nombre
 - Apellido
 - Nombre de usuario
 - Contraseña

El método debe guardar un usuario en un atributo estático llamado “Usuarios”, **recordando que la contraseña debe estar hasheada por seguridad**

Práctica de módulo nativo: crypto

- ✓ El método "Mostrar Usuarios" imprimirá en consola todos los usuarios almacenados.
- ✓ El método "Validar Usuario" recibirá el nombre de usuario que quiero validar, seguido de la contraseña, debe poder leer el json previamente generado con el arreglo de usuarios y hacer la comparación de contraseñas, Si coinciden el usuario y la contraseña, devolver un mensaje "Logueado", caso contrario indicar error si el usuario no existe, o si la contraseña no coincide.



Calculadora de edad

Realizando un programa que dependa de módulos externos

Duración: 10 min



ACTIVIDAD EN CLASE

Calculadora de edad

Realizar un programa que utilice la dependencia **momentjs** (deberá instalarse por npm install).

- ✓ Debe contar con una variable que almacene la fecha actual (utilizar moment())
- ✓ Debe contar con una variable que almacene sólo la fecha de tu nacimiento (utilizar moment).
- ✓ Validar con un if que la variable contenga una fecha válida (utilizar el método isValid());
- ✓ Finalmente, mostrar por consola cuántos días han pasado desde que naciste hasta el día de hoy. (utilizar el método diff())
- ✓ Extra: Cambia tu moment a la versión 1.6.0, al no ser la misma versión mayor, nota el cambio al correr el programa.





Break

¡10 minutos y volvemos!

Ejemplos de asincronismo: **setTimeout y setInterval**

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
```

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  * Nuevamente, todo parece normal, la tarea finaliza hasta
23  * que el ciclo haya terminado de contar del 1 al 5
24  * ¿Cómo funcionará asíncrono con intervalos?
25  */
```

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  */
```

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

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.



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 })
```



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.

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:

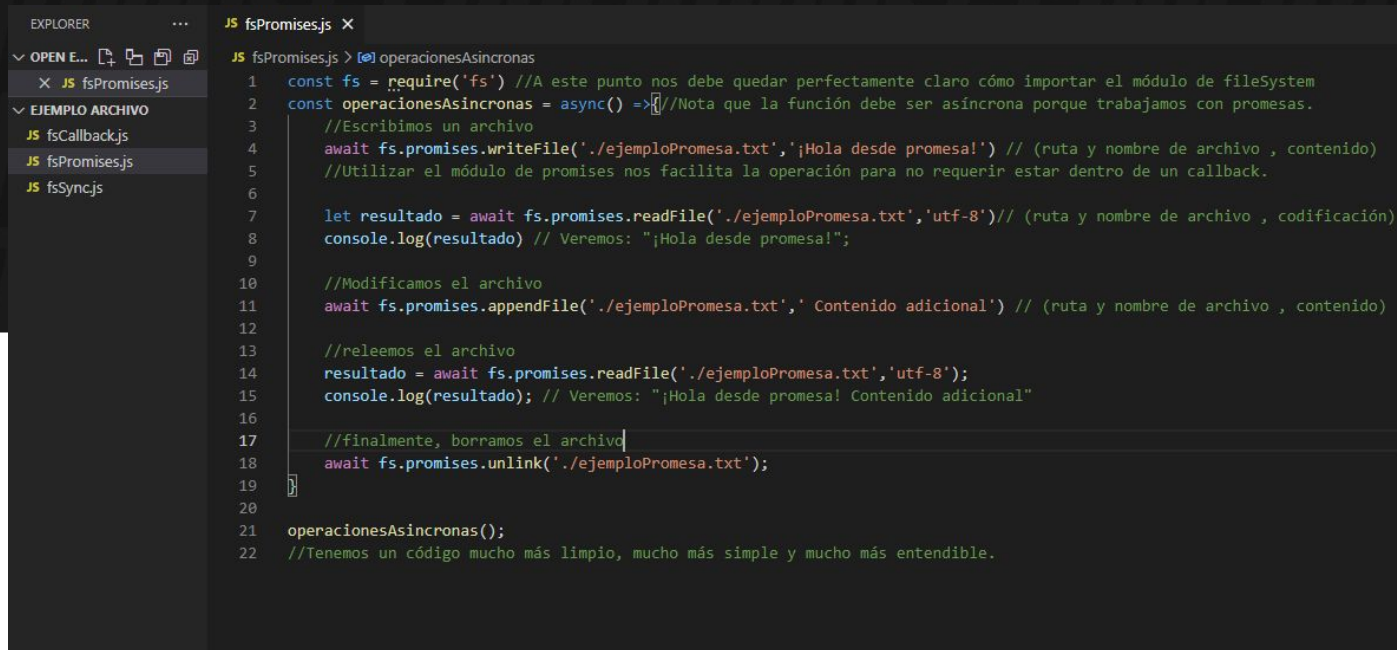
- ✓ **fs.promises.writeFile** = Para escribir contenido en un archivo. Si el archivo no existe, lo crea. Si existe, lo sobrescribe.
- ✓ **fs.promises.readFile** = Para obtener el contenido de un archivo.
- ✓ **fs.promises.appendFile** = Para añadir contenido a un archivo. ¡No se sobrescribe!
- ✓ **fs.promises.unlink** = 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.
```



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



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 mostrará 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 UsersManager.js (Como el realizado en la clase pasada).
- ✓ 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

Muchas gracias.

#DemocratizandoLaEducación