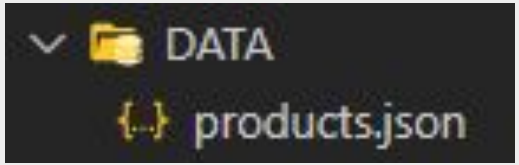

ICARO
Nuestro compromiso es con el futuro.

Introducción a la programación

Repaso

JSON

Los JSONs son un formato o tipo de archivo muy similares a los objetos literales, que nos permiten almacenar información de una manera muy ordenada y metódica, ya que deben respetarse sus reglas sí o sí para que éste pueda funcionar.



JSON

- JSON es un lenguaje de **intercambio de datos**.
- Su sintaxis se inspiró en la **notación literal** del objeto JavaScript.

Pero existen algunas diferencias…

En JSON, todas las **claves** deben estar **entre comillas**.
Mientras que, en los objetos literales, no es necesario.

Esto es porque, en JSON, el uso de palabras reservadas **no** **est** **itido**.

```
// JSON:
{ "foo": "bar" }

// Object literal:
var o = { foo: "bar" };
```

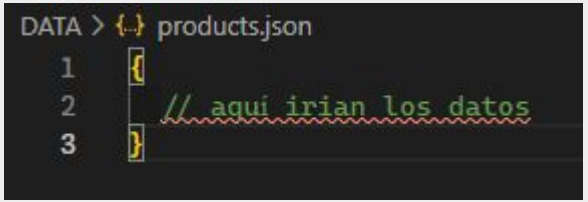
JSON

Los datos en JSON están restringidos a los siguientes tipos de dato

String
Number
Object
Array
true
false
null

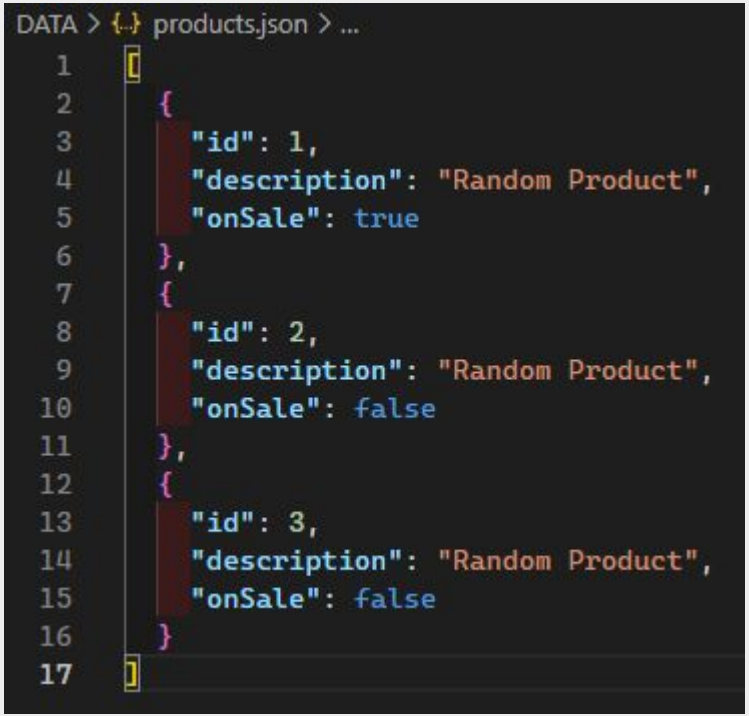
JSON

Un **JSON** puede ser o bien un **array** o un **objeto literal**, pero con algunas consideraciones: el contenido de un **JSON** no debe almacenarse en una variable o constante, sino que debe ser **literalmente** el dato a guardar, es decir el **array** u **objeto literal** puro.
Importante destacar que al ser archivos puramente de datos, no podemos cometer errores sintácticos ni utilizar comentarios.



JSON

Teniendo esto en cuenta, dentro de estos **objetos literales** o **arrays** podemos guardar cualquier dato como lo haríamos normalmente, con la diferencia de que las ***keys*** o ***propiedades*** del objeto literal deben ir **siempre** entre comillas, como **string**.
Más adelante veremos cómo podemos trabajar con estos **JSON** para leerlos, editarlos, sobreescribirlos, etc.

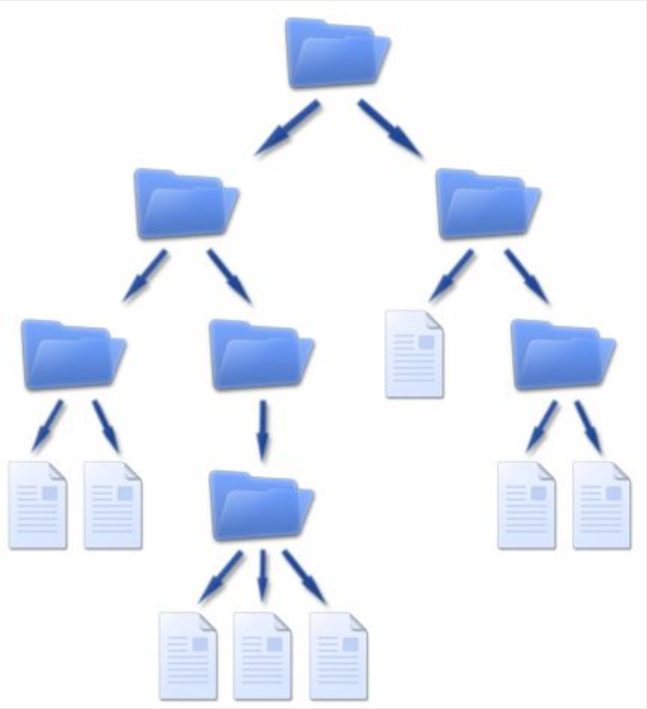


Clase 6

File System

¿Qué es un file system?

Un sistema de archivos o ficheros es un elemento que controla cómo se **almacenan y recuperan** los datos.
Estructura la información (documentos, fotos, videos, canciones) guardada en un dispositivo de almacenamiento (interno, como el **disco** de la computadora o externa, como un dispositivo de entrada **USB**) que luego será representada al usuario utilizando un **gestor de archivos** (explorador de Windows, Finder, Nautilus).



fs - File System

Cuando trabajamos en programación, nuestro elemento principal siempre son los datos. En esta línea, sabemos como guardar **datos** en variables, modificarlos, trabajar con ellos, etcétera. Pero aún no hemos visto como poder almacenar estos datos en algún lado.

Para ellos **node.js** tiene un módulo nativo que nos permite trabajar con la lectura y escritura de archivos: **File System**.

fs

Si hacemos un **console.log** del objeto fs encontramos que contiene muchos **métodos**.
Son estos métodos los que nos van a permitir trabajar con nuestros **datos**.

fs

Como siempre, lo primero será importar este objeto desde su módulo correspondiente.

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

fs

Puntualmente, nos interesan dos métodos particulares que veremos a continuación:

- **ReadFileSync**
- **WriteFileSync**

ReadFileSync

ReadFileSync

El primero de estos dos métodos es el **ReadFileSync**, método que nos permitirá leer de manera sincrónica (de ahí el “sync” del final) un archivo que le pasemos por parámetro.

```
function readFileSync(path:
fs.PathOrFileDescriptor, options?: {
    encoding?: null;
    flag?: string;
}): Buffer (+2 overloads)
```

ReadFileSync

Para nuestros propósitos, utilizaremos este método con dos parámetros:

```
console.log(fs.readFileSync('./data/data.json', 'utf-8'))
```

- El primero será la ruta del archivo que queremos **leer**.

- El segundo será la codificación del archivo que estamos intentando **leer**. Esta codificación permite al **fs** interpretar los caracteres que va a contener ese archivo para que lea correctamente.

ReadFileSync

Si hacemos un **console.log** de lo que obtenemos al ejecutar el **ReadFileSync** veremos que nos devuelve el contenido del **archivo**.

{} data.json

data > {} data.json > ...

```
[
  { "id": 1, "desc": "Lapiz" },
  { "id": 2, "desc": "Goma de borrar" },
  { "id": 3, "desc": "Lapicera" }
]
```

```
$ node app
[
  { "id": 1, "desc": "Lapiz" },
  { "id": 2, "desc": "Goma de borrar" },
  { "id": 3, "desc": "Lapicera" }
]
```

ReadFileSync

Si prestamos atención podemos ver que lo que nos está imprimiendo es literalmente el contenido del archivo. ¡Esto es genial!, pero aún no podemos trabajar con estos datos porque están en el formato **SON**, y nosotros necesitamos que sean un **objeto literal** de JavaScript para poder sacarle provecho.

Es por esto que utilizaremos el objeto **JSON** de **JavaScript** para poder ***interpretar*** o ***parsear*** el tipo de dato de un **JSON** a un **objeto literal**. También podemos aprovechar y almacenar el resultado del método en una variable para que sea más manejable.

JSON.parse

```
let data = fs.readFileSync('./data/data.json', 'utf-8')

console.log(JSON.parse(data))
```

```
$ node app
[
  { id: 1, desc: 'Lapiz' },
  { id: 2, desc: 'Goma de borrar' },
  { id: 3, desc: 'Lapicera' }
]
```

Como podemos ver, ahora la consola nos imprime un **objeto literal**, es decir que son **datos** que podemos manejar con **JavaScript** para poder manipularlos y trabajarlos según necesitemos.

ReadFileSync

Por ejemplo, podemos utilizar una función que reciba los **datos** y los **imprima** uno por uno por consola.

```
let data = JSON.parse(fs.readFileSync('./data/data.json', 'utf-8'))

const imprimirUnoPorUno = (datos) => {
  if (datos.length !== 0) {
    for (let i = 0; i < datos.length; i++) {
      console.log('-------')
      console.log('Dato:')
      console.log(datos[i])
    }
    console.log('-------')
  }
}
imprimirUnoPorUno(data)
```

```
-------
Dato:
{ id: 1, desc: 'Lapiz' }
-------
Dato:
{ id: 2, desc: 'Goma de borrar' }
-------
Dato:
{ id: 3, desc: 'Lapicera' }
-------
```

WriteFileSync

WriteFileSync

El segundo método es el **WriteFileSync**, que nos permitirá escribir de manera sincrónica un archivo. Este método **sobreescribirá** lo que ya exista en un archivo, por lo cual siempre deberíamos tener cuidado al utilizarlo.

Haciendo uso de este método podremos generar nuestras primeras instancias de **persistencia de datos**, es decir, trabajar con datos y guardar esos resultados en nuestra computadora de manera permanente.

```
function writeFileSync(file: fs.PathOrFileDescriptor, data: string |
NodeJS.ArrayBufferView, options?: fs.WriteFileOptions): void
```

WriteFileSync

Igual que en el anterior, para poder utilizarlo deberemos pasarle ciertos parámetros. En este caso serán tres:

- La **ruta del archivo** que queremos escribir.
- Los **datos** que queremos incluir dentro del archivo, es decir, el contenido.
- **Opciones** varias (este parámetro es opcional). Aquí podremos incluir ciertas personalizaciones sobre cómo queremos que se dé el proceso de la escritura.

```
fs.writeFileSync('./random/random.txt', 'esto es un texto random')
```

WriteFileSync

Ahora que sabemos cómo escribir un **archivo**, debemos tener en cuenta que para guardar los datos que manipulamos con **JavaScript** debemos hacer el proceso inverso al de lectura para guardarlos, es decir, debemos *parsear* nuevamente estos datos de **JavaScript** a **JSON** para poder utilizar el método y obtener un **JSON** con nuestros **datos**.

```
let data = [{ id: 1, desc: 'Lapiz' }]
let dataParseada = JSON.stringify(data, null, 2)

fs.writeFileSync('./data/data.json', dataParseada)
```

En el método JSON.stringify, le pasamos la data como primer parámetro, y luego el null y el número 2 para que, al escribirse, el archivo tenga un formato más legible. También podríamos simplemente pasarle sólo la data.

Vamos a practicar!

Muchas gracias!

ICARO Asociación Civil
CUIT 30716564815
info@icaro.org.ar
www.icaro.org.ar