# TEMA 1 INTRODUCCIÓN A NODE.JS

Parte IV – Ficheros de texto y formato JSON

# El formato JSON

- Son las siglas de *JavaScript Object Notation*, una sintaxis propia de Javascript para poder representar objetos como cadenas de texto, y poder así serializar y enviar información.
- Un objeto Javascript se define mediante una serie de propiedades y valores.

```
let persona = {
nombre: "Nacho",
edad: 39
};
```

Formato JSON:

```
{"nombre":"Nacho","edad":39}
```

- Si hablamos de un vector:

```
let personas = [
{ nombre: "Nacho", edad: 39},
{ nombre: "Mario", edad: 4},
{ nombre: "Laura", edad: 2},
{ nombre: "Nora", edad: 10}
]
```

```
[{"nombre":"Nacho","edad":39},
    {"nombre":"Mario","edad":4},
{"nombre":"Laura","edad":2},
    {"nombre":"Nora","edad":10}]
```

# Conversión a y desde JSON

- Crear un archivo llamado "prueba_json.js“ y pon este vector:

```
let personas = [
{ nombre: "Nacho", edad: 39},
{ nombre: "Mario", edad: 4},
{ nombre: "Laura", edad: 2},
{ nombre: "Nora", edad: 10}
]
```

- CONVERTIR a JSON: empleamos el método **JSON.stringify**

```
let personasJSON = JSON.stringify(personas);
console.log(personasJSON);
```

- CONVERTIR desde JSON: empleamos el método **JSON.parse:**

```
let personas2 = JSON.parse(personasJSON);
console.log(personas2);
```

# Acceso a Ficheros

- Debemos emplear el módulo FS
- Para leer: readFileSync
- Para escribir: writeFileSync
- Son funciones síncronas, pero es normal esperar que acaben estas operaciones antes de seguir. En cualquier caso esta sincronia es solo para la petición concreta en curso, no afecta al funcionamiento asíncrono del servidor principal.

# Ejemplo acceso a ficheros

- Crea el fichero prueba_fs_texto.js:

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

const archivo = "datos.txt";
const textoPredefinido = "En un lugar de la mancha\n"+
"de cuyo nombre no quiero acordarme";

let guardarDatos = () => {
fs.writeFileSync(archivo, textoPredefinido);
};

let leerDatos = () => {
return fs.readFileSync(archivo, 'utf-8');
};

guardarDatos();
console.log("Texto del fichero:");
console.log(leerDatos());
```

# Lectura y escritura en formato JSON

- Combinamos lo anterior con el uso de JSON:

```
let objetoJavascript =
JSON.parse(fs.readFileSync(nombre_archivo,
    'utf8'));
fs.writeFileSync(nombreArchivo,
    JSON.stringify(objetoJavascript));
```

- Con el ejemplo anterior:

```
fs.writeFileSync(archivo,
    JSON.stringify(personas));
let personas2 =
    JSON.parse(fs.readFileSync(archivo, 'utf8'));
console.log(personas2);
```

# Ejemplo. Estructura básica LIBRO

- En primer lugar, vamos a definir el modelo de datos que vamos a almacenar en esta aplicación de ejemplo. Trataremos con libros, que tendrán cuatro atributos:
  - Un id (numérico entero)
  - Un título (texto)
  - Un autor (texto)
  - Un precio (numérico real)
- Por lo tanto, un objeto de este tipo tendrá una estructura similar a esta:

```
let libro = {
id: 1,
titulo: "El Señor de los Anillos",
autor: "J.R.R. Tolkien",
precio: 19.95
};
```

# Ejemplo. Estructura de ficheros

- Crea un carpeta **modelo** y crea 2 ficheros dentro:
- **libros.js:** Tendremos la definición de libro con los métodos que afecten a los mismos.
- **index.js:** Tendrá el require que use lo anterior
- Fuera de esta carpeta modelo habrá otro **index.js** que será el programa principal.

# Ejemplo. Libro.js

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

const archivo = 'libros.json';

let cargarLibros = () => {
try {
	return JSON.parse(fs.readFileSync(archivo, 'utf8'));
} catch (e) {
	return [];
}
};

let guardarLibros = (libros) => {
	fs.writeFileSync(archivo, JSON.stringify(libros));
};
```

- Podemos comprobar que capturamos una excepción en caso que falle la lectura del fichero, devolviendo un vector vacío, por ejemplo cuando el fichero este vacío.
- Grabar, guardará en el fichero en forma JSON.

# Ejemplo. Libros.js II. Buscar Libro

```
let buscarLibroPorId = (id) => {
let libros = cargarLibros();
let resultado = libros.filter((libro) => libro.id == id);
if (resultado.length > 0)
return resultado[0];
};
```

- Una función interesante puede ser que busque un id de un libro, si existe lo devuelva y si no que devuelva undefined.
- Observar la función filter, que busca por un criterio y devuelve el subconjunto que lo cumple.

# Ejemplo. Libro.js III. Insertar libro

```
let nuevoLibro = (id, titulo, autor, precio) => {
    if (!buscarLibroPorId(id))
    {
        let libros = cargarLibros();
        let nuevo = {
            id: id,
            titulo: titulo,
            autor: autor,
            precio: precio
        };
        libros.push(nuevo);
        guardarLibros(libros);
        return true;
    }
};
```

# Ejemplo. Libro.js III. Borrar libro

```
let borrarLibro = (id) => {
    let libros = cargarLibros();
    let librosFiltrados =
    libros.filter((libro) => libro.id != id);
    if (librosFiltrados.length !== libros.length)
            guardarLibros(librosFiltrados);
    return librosFiltrados.length !== libros.length;
}
```

- Para borrar buscamos el que deseamos borrar y nos quedamos con el resto.

# Ejemplo. Libros.js IV. Exportar

- Para que sean visibles fuera:

```
module.exports = {
listarLibros: cargarLibros,
nuevoLibro: nuevoLibro,
borrarLibro: borrarLibro
};
```

- Y en index.js de la carpeta modelo:

```
const libros = require('./libros');

module.exports = {
libros: libros
};
```

# Ejemplo. Index.js. PRINCIPAL

```
const modelo = require('./modelo');

let libros1 = modelo.libros.listarLibros();
console.log(libros1);

modelo.libros.nuevoLibro(1, "El Señor de los Anillos",
"J.R.R. Tolkien", 19.95);
modelo.libros.nuevoLibro(2, "El juego de Ender",
"Orson Scott Card", 9.90);

let libros2 = modelo.libros.listarLibros();
console.log(libros2);

modelo.libros.borrarLibro(1);

let libros3 = modelo.libros.listarLibros();
console.log(libros3);
```

# EJERCICIO

- Como ejercicio de esta sesión, vamos a hacer algo similar al ejemplo visto, pero con otro modelo de datos. En este caso, vamos a trabajar con videojuegos en una carpeta llamada "Tema1_VideojuegosJSON" en vuestra carpeta de "Ejercicios". Cada objeto videojuego tendrá la siguiente información:
  - Un id (numérico entero)
  - Un título (cadena)
  - Un fabricante (cadena)
  - Una categoría (cadena)
  - Un año de lanzamiento (numérico entero)
- Se pide hacer una estructura similar a la del ejemplo anterior, para poder añadir/quitar videojuegos del catálogo (almacenado en "videojuegos.json", en la carpeta apropiada). Además de lo visto en el ejemplo anterior, el modelo de videojuegos debe tener dos funciones más (accesibles desde fuera):
- Una llamada listarVideojuegosDesdeAnyo que devuelva un subconjunto con los videojuegos cuya fecha de lanzamiento sea igual o posterior al año indicado como parámetro
- Otra llamada listarVideojuegosPorCategoria que devuelva un subconjunto de todos los videojuegos de la categoría indicada como parámetro.
- En el programa principal, habrá pruebas de funcionamiento de cada uno de los métodos públicos del modelo de videojuegos (añadir/quitar/filtrar videojuegos).