Manejo de Archivos (1era parte)

Teoría

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.

Entre las ventajas de usar archivos encontramos las siguientes:

- 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.

Sin embargo, no serán la mejor opción cuando tengamos que hacer (con frecuencia):

- Consultas sobre algún dato puntual entre todos los datos almacenados (y no podamos guardarnos todo el lote de datos en memoria).
- Ediciones de datos puntuales (que no requieren sobreescribir el archivo por completo)
- Lecturas que combinen datos obtenidos de varios archivos (nuevamente, suponiendo que no podemos guardar todos los datos en memoria)

Para este segundo caso, probablemente sea mejor considerar en uso de un motor de base de datos. Este tema será abordado en otro módulo de la materia.

Manejo de Archivos en NodeJS

Al igual que la mayoría de los lenguajes, NodeJS cuenta con una librería (o módulo, en js) para interactuar con el sistema de archivos (o File System) de tu computadora, que recibe el nombre "fs", por razones obvias.
Para poder usar este módulo solo debemos importarla al comienzo de nuestro archivo fuente, utilizando la declaración *`import [ identifier ] from [ module | path ]`*:

```
import fs from 'fs'
```

Para más información sobre import/export, ver apéndice al final del apunte.

La mayoría de las funciones que contiene este módulo pueden usarse tanto de manera **sincrónica** como **asincrónica**. En este apunte estudiaremos primero las funciones **sincrónicas** (de uso más intuitivo), y luego en el próximo apunte pondremos más atención en

sendas versiones asincrónicas, ya que su funcionamiento es algo propio de NodeJS, y requiere de su estudio con más detenimiento.

Operaciones más comunes sobre archivos

Leer

Para leer un archivo usaremos la función `readFileSync(path, encoding)`
El primer parámetro es un string con la ruta del archivo que queremos leer, y 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).

Ejemplo de uso:

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

Algunas consideraciones:

Si la ruta comienza con un `''` o `'./'` se trata de una ruta relativa, es decir, que el programa se está ejecutando en la carpeta `'/user/documents/workspace/proyecto/'` y 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.

Escribir (pisando el contenido anterior, si lo hay)

Para sobreescribir el contenido de un archivo usaremos la función `writeFileSync(ruta, datos)`. El primer parámetro es un string con la ruta del archivo en el que queremos escribir y 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. Si la función no recibe este tercer parámetro, se usará el formato por defecto, `'utf-8'`.

Ejemplo de uso:

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

Importante:

Si la ruta provista fuera válida, pero el nombre de archivo no existiera, la función creará un nuevo archivo con el nombre provisto.

Escribir (agregando al final del contenido existente, si lo hay)

Para agregar contenido a un archivo usaremos la función `appendFileSync(ruta, datos)`
El primer parámetro es un string con la ruta del archivo al que le queremos agregar contenidos, y 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. Si la función no recibe este tercer parámetro, se usará el formato por defecto, `'utf-8'`.

Ejemplo de uso:

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

Importante:

Al igual que con la función `writeFileSync`, si la ruta provista fuera válida, pero el nombre de archivo no existiera, la función creará un nuevo archivo con el nombre provisto.

Otras operaciones sobre archivos

Además de leerlos y escribirlos, es posible que queramos hacer otras cosas con los archivos. Acá listamos algunas de esas operaciones:

Renombrar

Está función, que no devuelve nada, busca el archivo en la primera ruta provista y le asigna la segunda ruta provista.

Ejemplo de uso

```
fs.renameSync(rutaVieja, rutaNueva)
```

Borrar

Está función, que no devuelve nada, elimina al archivo ubicado en la ruta provista.

Ejemplo de uso

```
fs.unlinkSync(ruta)
```

Crear una carpeta

Está función, que no devuelve nada, crea una carpeta vacía en el directorio ubicado en la ruta provista.

Ejemplo de uso

```
fs.mkdirSync(ruta)
```

Leer el contenido de un directorio (carpeta)

Está función devuelve una lista de strings con los nombres de los archivos y carpetas del directorio que se encuentre en la ruta provista.

Ejemplo de uso

```
const listaDeNombresDeArchivos = fs.readdirSync(rutaDeLaCarpeta)
```

Manejo de errores

Es importante mencionar que todas las funciones que acabamos de describir pueden lanzar excepciones en el caso de encontrarse con algún imprevisto que impida su ejecución. La forma adecuada de manejar estas excepciones será ejecutando el código en cuestión dentro de una cláusula: `try / catch`.

Ejemplo de uso

```
try {
    const data = fs.readFileSync(path)
    console.log(data)
} catch (algunError) {
    // manejar el error!!
    console.log(algunError)
}
```

Últimas observaciones

Manejar archivos en forma sincrónica es muy fácil e intuitivo, y nos permite escribir código en forma similar a la mayoría de los demás lenguajes que usamos habitualmente. Sin embargo, Es recomendable aprender a trabajar en forma asincrónica para poder aprovechar al máximo las herramientas que nos brinda el lenguaje para mejorar la performance de nuestras aplicaciones.

Apéndice A: Modularización

En el caso de que quisiéramos repartir nuestras funciones entre distintos archivos para hacer más explícita la separación de responsabilidades, podremos hacerlo sin problemas.

Basta con crear un nuevo archivo en alguna carpeta, por ejemplo ‘miLib.js’, en el cual podremos escribir nuestras variables y funciones como lo haríamos normalmente. Luego, al final del archivo, agregamos la siguiente declaración:

```
export default {
  nombreDeVariable,
  nombreDeFuncion,
  .....
}
```

Aquello que incluyamos dentro de estas llaves será la parte “visible” a la que podremos acceder cuando importemos nuestro módulo desde otro archivo.

Para importar un módulo creado por nosotrxs, lo haremos igual que con los módulos propios de js, pero está vez incluiremos la ruta del archivo en cuestion. Ejemplo:

```
import lib from 'misLibrerias/lib.js'
```

Existen otras formas de importar y exportar cosas en js, que nos pueden brindar un control más fino y específico de los datos expuestos. Para mayor información, no duden en echarle un vistazo a la documentación oficial.