



**Argentina
programa
4.0**



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*

Clase 8: API sistema de archivos

Agenda de hoy

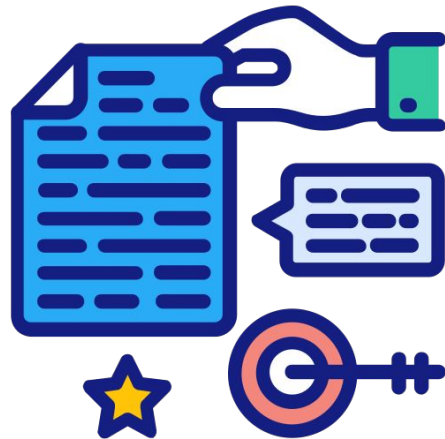
- A. FileSystem API
 - a. manejo de archivos del lado del servidor
- B. FileSystem
 - a. abrir un archivo (readFile)
 - b. leer su contenido
 - c. agregar información al archivo (appendFile)
 - d. guardar el contenido del archivo (writeFile)
 - e. manejo de directorios
- C. Mecanismos de codificación
 - a. codificar versus encriptar
 - b. Base64
- D. Actividad práctica



FileSystem API

Si bien hemos tenido una aproximación a lo que este módulo brinda dentro de Node.js, **FileSystem API** es una herramienta mucho más completa y efectiva cuando se trata de manipular, tanto archivos, como también carpetas o directorios.

Profundicemos un poco más en sus características desarrollando ejemplos prácticos, con todo lo que este módulo nos ofrece.

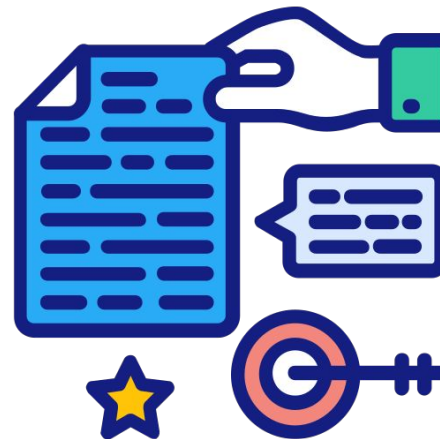


Qué es FileSystem API

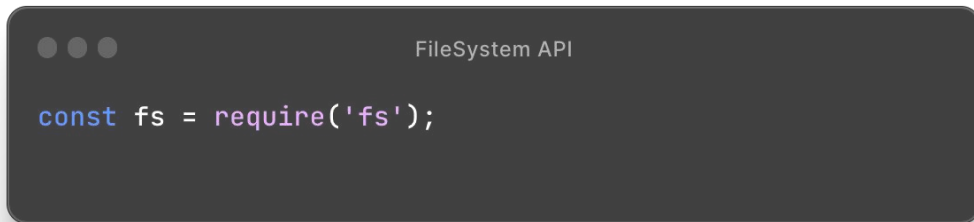
FileSystem API

El **módulo fs** integrado en Node.js proporciona una API para interactuar con el sistema de archivos en el servidor. Dentro de todas sus prestaciones, podemos destacar la creación, escritura, modificación y eliminación de archivos dentro de un sistema operativo.

Nuestras pruebas y ejemplos se enfocarán íntegramente dentro de un proyecto Node.js.



FileSystem API



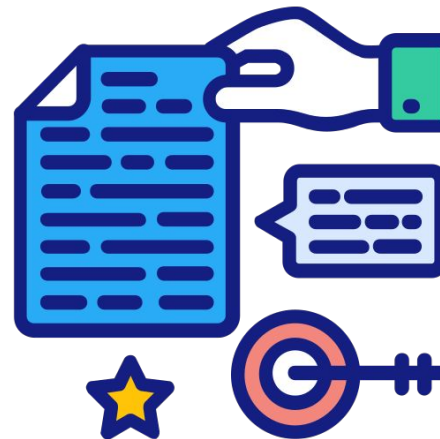
```
const fs = require('fs');
```

Para integrar FileSystem API dentro de un proyecto Node.js, **debemos declarar el módulo fs utilizando la función JS require()**. De esta forma, ya queda disponible para que lo utilicemos dentro de nuestros proyectos.

FileSystem API

Una vez declarado, este módulo cuenta con un montón de métodos JS, los cuales nos facilitan realizar todo tipo de interacción sobre un archivo, o varios, integrados mayormente en nuestro proyecto.

Veamos a continuación cuáles son los métodos principales para manipular las diferentes operaciones sobre archivos:



FileSystem API

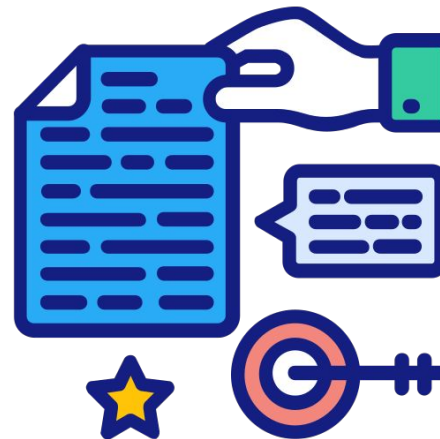
Método	Descripción
readFile()	Nos permite leer el contenido de un archivo plano.
writeFile()	Nos permite escribir dentro de un archivo plano, el contenido que necesitemos volcar.
appendFile()	Nos permite agregar contenido dentro de un archivo ya creado, respetando lo que éste tenga previamente almacenado.
unlink()	Elimina el archivo que le indiquemos. Debemos tener precaución siempre que utilicemos este método, porque es irrecuperable la acción en cuestión.

Crear un archivo

Crear un archivo

Nuestro primer paso, será crear un archivo en un proyecto Node.js. Para ello utilizaremos el método **.writeFile()**.

Éste, cuenta con una serie de parámetros donde podemos definir el nombre del archivo a crear, y el texto que deseamos incluir en el mismo.



Crear un archivo

```
FileSystem API

const fs = require('fs');

fs.writeFile('miarchivo.txt', 'Hola, archivo!', (error)=> {

});
```

Este método recibe tres parámetros en cuestión.

El primero de ellos corresponde al **nombre del archivo a crear**, **el segundo al texto que deseamos agregar** y finalmente, **el tercero** corresponde al **callBack que controla el éxito de la operación** y/o cualquier posible error.

Crear un archivo

Ejemplo funcional del código en cuestión, donde creamos el archivo con un texto simple (*en formato plano*), y luego mediante la función callback controlamos si hay un error (*lo informamos y visualizamos*), sino, notificamos que el archivo se creó correctamente.

```
FileSystem API

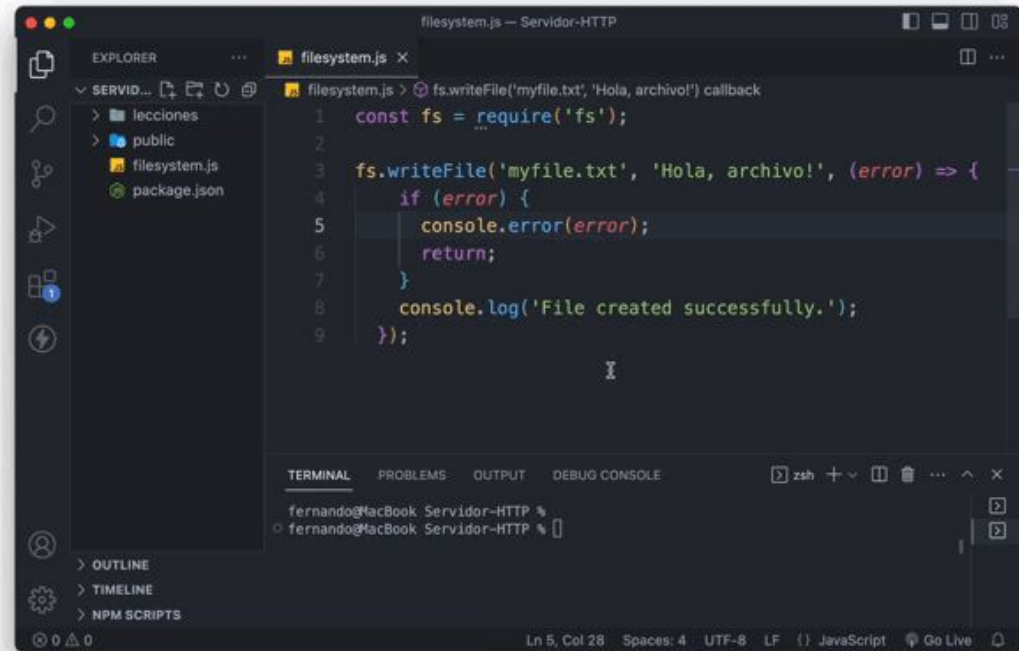
const fs = require('fs');

fs.writeFile('miarchivo.txt', 'Hola, archivo!', (error) => {
  if (error) {
    console.error(error);
    return;
  }
  console.log('El archivo se ha creado correctamente.');
```

Crear un archivo

Como podemos ver en el ejemplo, el proceso es simple y directo.

Ejecutamos la aplicación Node.js y el archivo se crea de forma inmediata.



```
filesystem.js — Servidor-HTTP  
1  const fs = require('fs');  
2  
3  fs.writeFile('myfile.txt', 'Hola, archivo!', (error) => {  
4      if (error) {  
5          console.error(error);  
6          return;  
7      }  
8      console.log('File created successfully.');
```

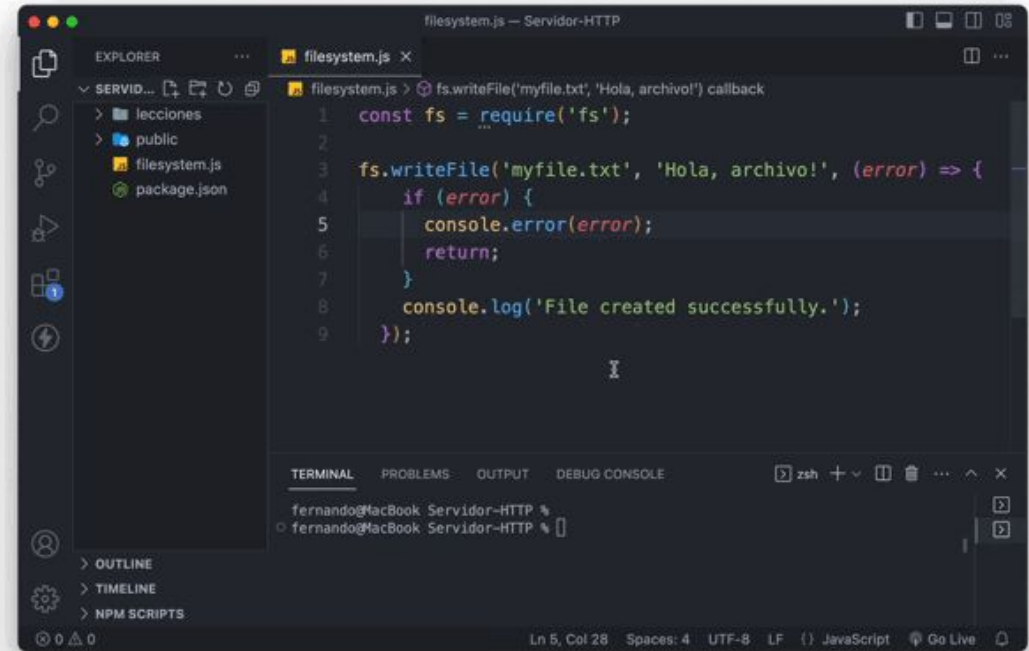
TERMINAL

```
fernando@MacBook: Servidor-HTTP %  
fernando@MacBook: Servidor-HTTP %
```

Crear un archivo

No hay verificaciones ni validaciones ni nada por el estilo durante este proceso.

Esto, si bien es un beneficio para nosotros, también debe ser una advertencia, porque **si el archivo fue creado anteriormente y tiene contenido, ¡podemos sobrescribirlo!**.



The screenshot shows a code editor with a file named `filesystem.js` open. The code in the editor is as follows:

```
1 const fs = require('fs');
2
3 fs.writeFile('myfile.txt', 'Hola, archivo!', (error) => {
4   if (error) {
5     console.error(error);
6     return;
7   }
8   console.log('File created successfully.');
```

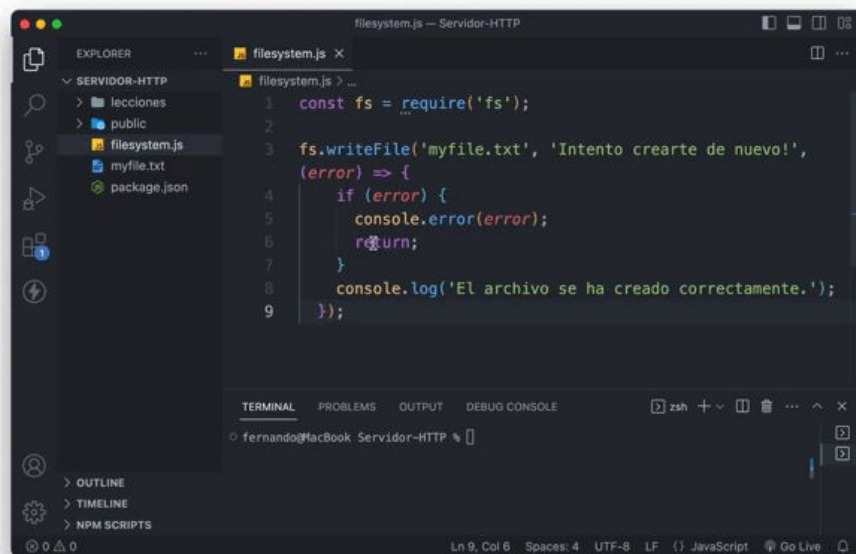
The Explorer panel on the left shows a project structure with folders `lecciones` and `public`, and files `filesystem.js` and `package.json`. The terminal at the bottom shows the command `fs.writeFile('myfile.txt', 'Hola, archivo!')` being executed, and the output `File created successfully.` is visible.

Crear un archivo

Aquí tenemos un ejemplo de sobreescritura de un archivo, sin advertencias. 🤯

El uso de este tipo de métodos, debe contenerse dentro de funciones y, a su vez, hacer validaciones de existencia de archivo previo a crearlo.

Como alternativa, utilizar datos aleatorios en el nombre del archivo, si es que creamos recursos temporales y luego los eliminamos rápidamente.



```
1 const fs = require('fs');
2
3 fs.writeFile('myfile.txt', 'Intento crear de nuevo!',
4   (error) => {
5     if (error) {
6       console.error(error);
7       return;
8     }
9     console.log('El archivo se ha creado correctamente.');
```


Crear un archivo

El método **.existsSync()** nos permite validar, retornando un valor booleano, si un archivo existe o no.

Aquí un ejemplo simple de cómo implementarlo en una función con retorno, para luego crearlo, o no.

```
FileSystem API

const fs = require('fs');

function fileExists(filename) {
  const existe = fs.existsSync(filename.trim());
  return existe ? true : false;
}
```

Crear un archivo

Finalmente, creamos una función dedicada en donde le pasamos el nombre del archivo y contenido que deseamos agregar en él.

Con una **estructura if()** utilizamos la función **fileExists()** para validar si existe. En el caso que exista, evitamos su creación.

```
FileSystem API

function crearArchivo(filename, content) {
  const archivo = `${filename.trim()}.txt`;

  if (fileExists(archivo)) {
    console.error('El archivo existe. No se puede sobrescribir.');
```

```
  } else {
    fs.writeFile(archivo, content.trim(), (error) => {
      if (error) {
        console.error(error);
        return;
      }
      console.log('El archivo se ha creado correctamente.');
```

```
    });
  }
}
```

Leer un archivo

Leer un archivo



El método **.readFile()** nos permite leer un archivo.

Creemos a continuación una función que nos permita leer el archivo y enviarlo a la consola JS.

Leer un archivo

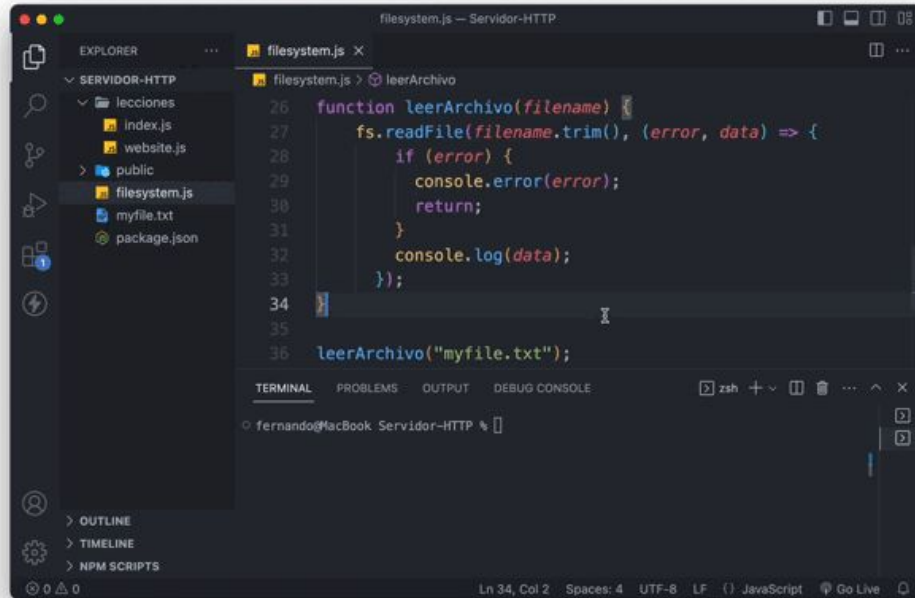
```
FileSystem API

function leerArchivo(filename) {
  fs.readFile(filename.trim(), (error, data) => {
    if (error) {
      console.error(error);
      return;
    }
    console.log(data);
  });
}
```

Definimos una función dedicada para la lectura del archivo, controlando por supuesto cualquier error que ocurra durante la misma.

Si todo va bien, leemos el contenido de este en la consola JS.

Leer un archivo

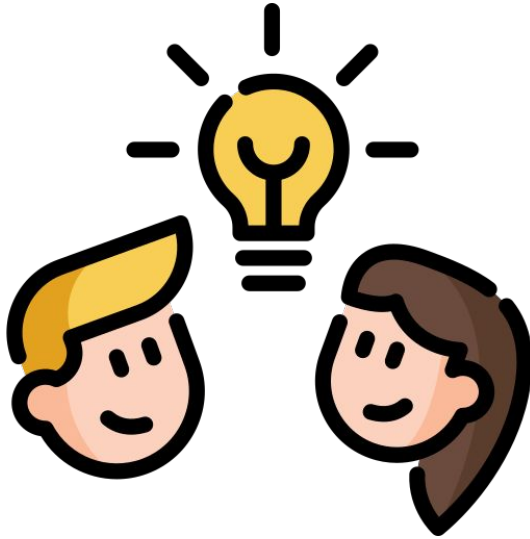


```
26 function leerArchivo(filename) {
27   fs.readFile(filename.trim(), (error, data) => {
28     if (error) {
29       console.error(error);
30       return;
31     }
32     console.log(data);
33   });
34
35
36 leerArchivo("myfile.txt");
```

Si todo “*sale correctamente*”, debemos toparnos con un pequeño error en la consola JS, en el momento en que nuestra aplicación Node devuelve el contenido del archivo.

¿Cuál es el problema?

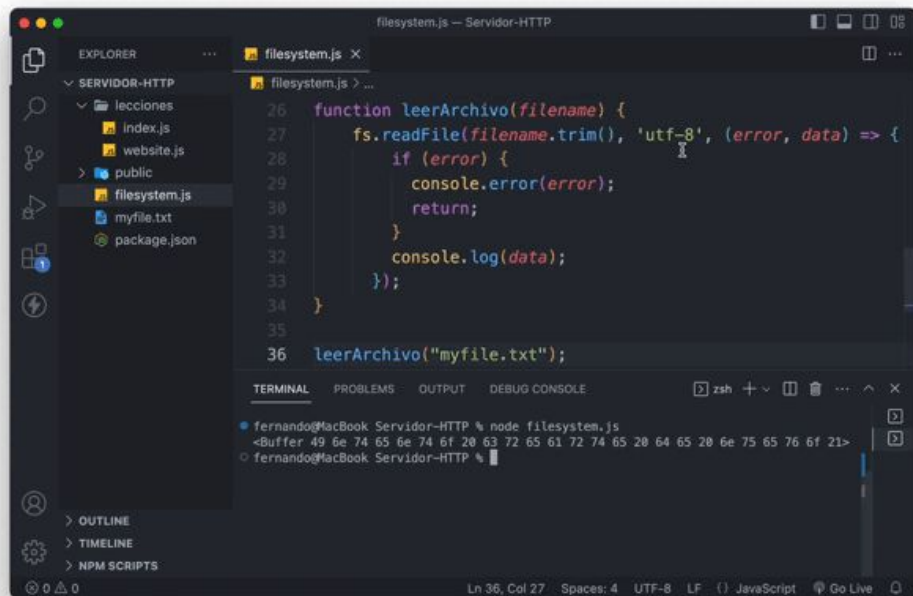
Leer un archivo



Cuando leemos contenido de un archivo alojado en un servidor web, este contenido se lee en forma de stream.

Por ello, debemos indicar la codificación de caracteres del archivo leído, para así evitar la generación de un buffer, en pos de poder visualizar correctamente el texto almacenado en éste.

Leer un archivo



```
filesystem.js — Servidor-HTTP
EXPLORER
  SERVER-HTTP
    lecciones
      index.js
      website.js
    public
      filesystem.js
      myfile.txt
      package.json
  ...
filesystem.js
26 function leerArchivo(filename) {
27   fs.readFile(filename.trim(), 'utf-8', (error, data) => {
28     if (error) {
29       console.error(error);
30       return;
31     }
32     console.log(data);
33   });
34 }
35
36 leerArchivo("myfile.txt");

TERMINAL
fernando@MacBook Servidor-HTTP % node filesystem.js
<Buffer 49 6e 74 65 6e 74 6f 20 63 72 65 61 72 74 65 20 64 65 20 6e 75 65 76 6f 21>
fernando@MacBook Servidor-HTTP %
```

El parámetro **“utf-8”** será quien acomode el contenido en *formato stream*, y haga que este sea legible en la codificación de caracteres en la cual fue escrita.

Agreguemos entonces el parámetro **‘utf-8’** en el método **.readFile()**.

Leer un archivo



El concepto de stream, y de lectura de archivos con Node.js es muy amplio. Si nos topamos en algún momento con archivos de gran tamaño, los mismos tienen una forma particular de lectura.

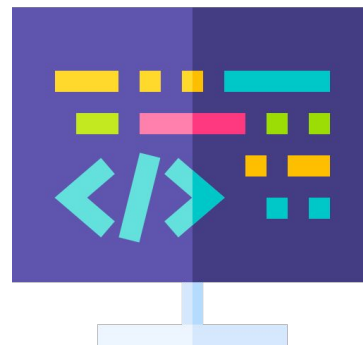
Te invitamos a [leer este artículo](#) para conocer más en profundidad *el concepto stream en Node.js*.

**Escribir contenido
en un archivo**

Escribir contenido en un archivo

Existe la posibilidad de ir agregando contenido de forma parcial dentro de un archivo ya creado.

Para ello, el método **.appendFile()** es el apropiado.



Escribir contenido en un archivo

Como todos los métodos de FileSystem API, es simple de utilizar.

Recibe como **primer parámetro** el **nombre del archivo**, como **segundo parámetro el contenido**, y finalmente, como **tercer parámetro una función callback** la cual maneja un error o notifica sobre la operación exitosa.

```
fs.appendFile(filename, content, (error) => {  
  if (error) {  
    console.error(error);  
    return;  
  }  
  console.log('Se agregó contenido al archivo.');
```



Escribir contenido en un archivo

Definimos una función dedicada para agregar diferente contenido en nuestro archivo en cuestión, de acuerdo a lo que venimos trabajando.

```
FileSystem API

function agregarContenido(filename, content) {
  const archivo = `${filename.trim()}.txt`;
  const texto = content.trim();

  fs.appendFile(archivo, texto, (error) => {
    if (error) {
      console.error(error);
      return;
    }
    console.log('Se agregó contenido al archivo.');
```

Escribir contenido en un archivo

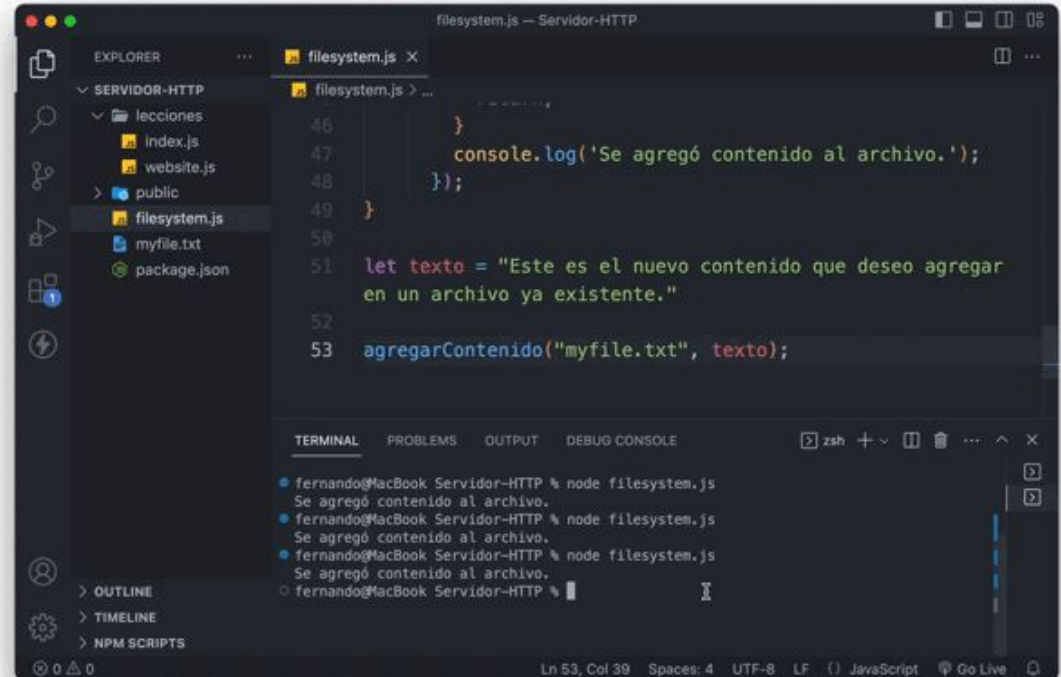
Luego, en una variable, definimos el contenido en formato texto para realizar una prueba simple. Finalmente, invocamos a nuestra función **agregarContenido()**, y luego verificamos que el contenido se haya agregado correctamente en el archivo en cuestión.

FileSystem API

```
let texto = "Este es el nuevo contenido que deseo  
agregar en un archivo ya existente."  
  
agregarContenido("myfile.txt", texto);
```

Escribir contenido en un archivo

Ejecutada nuestra aplicación Node.js, ya podemos validar a continuación el archivo .TXT creado, que el mismo contiene el texto agregado con esta última función JS.



The screenshot shows a Visual Studio Code editor window with the file `filesystem.js` open. The Explorer sidebar on the left shows the project structure: `SERVERIDOR-HTTP` (containing `lecciones`), `index.js`, `website.js`, `public` (containing `filesystem.js`, `myfile.txt`, and `package.json`). The main editor displays the following JavaScript code in `filesystem.js`:

```
46 }
47 console.log('Se agregó contenido al archivo.');
```

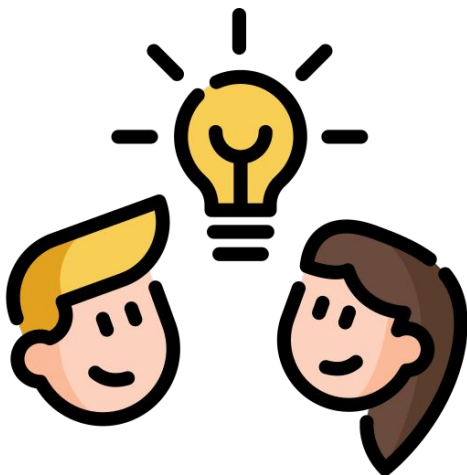
```
48 });
49 }
50
51 let texto = "Este es el nuevo contenido que deseo agregar
52 en un archivo ya existente."
53 agregarContenido("myfile.txt", texto);
```

Below the editor, the TERMINAL panel shows the output of running `node filesystem.js` three times, each resulting in the message: `Se agregó contenido al archivo.`

```
fernando@MacBook Servidor-HTTP % node filesystem.js
Se agregó contenido al archivo.
fernando@MacBook Servidor-HTTP % node filesystem.js
Se agregó contenido al archivo.
fernando@MacBook Servidor-HTTP % node filesystem.js
Se agregó contenido al archivo.
fernando@MacBook Servidor-HTTP %
```

The status bar at the bottom indicates the current position is Line 53, Column 39, with 4 spaces, using UTF-8 encoding and LF line endings, in a JavaScript file.

Escribir contenido en un archivo



Al ejecutar la función `.appendFile()`, pudimos comprobar que el contenido agregado al archivo `.TXT` se grabó de forma automática.

El método en cuestión, además de agregar contenido en un archivo también lo guarda, evitando así que tengamos que recurrir a una segunda función para esto último.

Borrar un archivo

Borrar un archivo

Nos queda ver cómo podemos eliminar un archivo existente utilizando FileSystem API.

Para ello, el método **.unlink()** es la herramienta que nos ayudará a realizar esta tarea.



Borrar un archivo

Como bien dijimos, el método es fácil de utilizar, recibiendo como **primer parámetro el nombre del archivo** en cuestión, y como **segundo parámetro el callback que controla cualquier posible error, o la eliminación efectiva del archivo** en cuestión.

```
FileSystem API

fs.unlink(filename, (error) => {
  if (error) {
    console.error(error);
    return;
  }
  console.log(`El archivo ${filename} se ha eliminado correctamente.`);
});
```

Borrar un archivo

Armamos una función dedicada para eliminar archivos, pasándole como parámetro el nombre de éste.

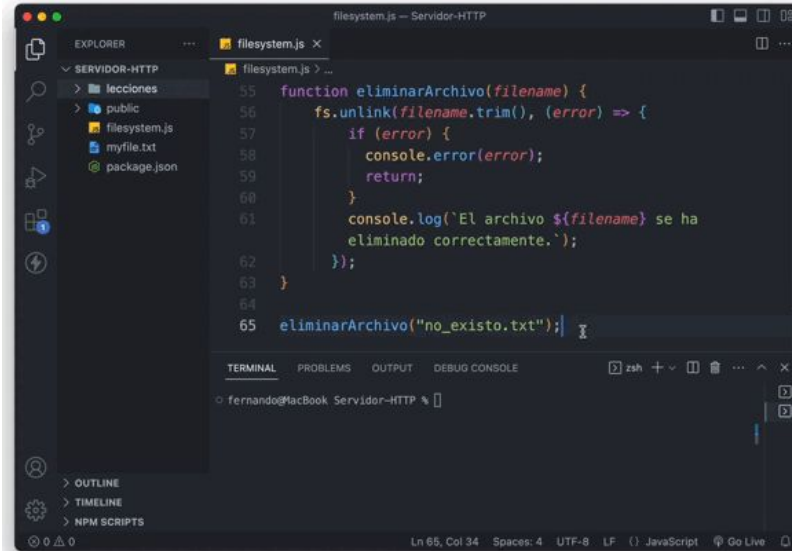
```
function eliminarArchivo(filename) {  
  fs.unlink(filename.trim(), (error) => {  
    if (error) {  
      console.error(error);  
      return;  
    }  
    console.log(`El archivo ${filename} se ha eliminado correctamente.`);  
  });  
}
```

Nuestra primera prueba será invocar un nombre de archivo inexistente, para ver el error que arroja.

```
eliminarArchivo("no_existo.txt");
```

Borrar un archivo

El error en la Consola JS valida de que el archivo que intentamos eliminar, no existe en el sistema de archivos de nuestro proyecto Node.js.



```
filesystem.js — Servidor-HTTP

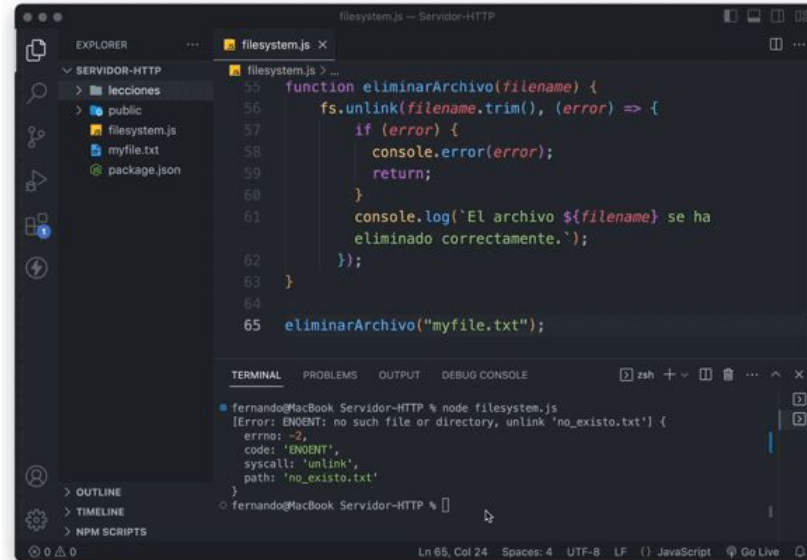
EXPLORER
  SERVADOR-HTTP
    lecciones
    public
    filesystem.js
    myfile.txt
    package.json

filesystem.js
55 function eliminarArchivo(filename) {
56   fs.unlink(filename.trim(), (error) => {
57     if (error) {
58       console.error(error);
59       return;
60     }
61     console.log(`El archivo ${filename} se ha
62       eliminado correctamente.`);
63   });
64 }
65 eliminarArchivo("no_existo.txt");

TERMINAL
fernando@MacBook: Servidor-HTTP %
```

Borrar un archivo

Definiendo un nombre de archivo existente, vemos que efectivamente éste se elimina del sistema de archivos de nuestro proyecto Node.js.



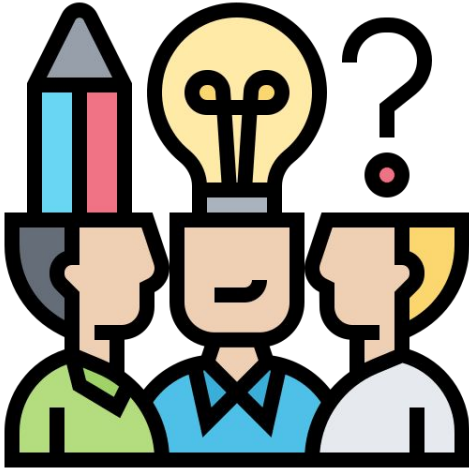
The screenshot shows a Visual Studio Code editor window with a file explorer on the left and a code editor in the center. The file explorer shows a project structure with a 'public' folder containing 'filesystem.js' and 'myfile.txt', and a 'package.json' file. The code editor shows the following JavaScript code in 'filesystem.js':

```
55 function eliminarArchivo(filename) {  
56     fs.unlink(filename.trim(), (error) => {  
57         if (error) {  
58             console.error(error);  
59             return;  
60         }  
61         console.log(`El archivo ${filename} se ha  
        eliminado correctamente.`);  
62     });  
63 }  
64  
65 eliminarArchivo("myfile.txt");
```

The terminal at the bottom shows the command 'node filesystem.js' being executed, resulting in an error message: '[Error: ENOENT: no such file or directory, unlink 'no_existo.txt'] { error: -2, code: 'ENOENT', syscall: 'unlink', path: 'no_existo.txt' }'. This indicates that the file 'myfile.txt' was successfully deleted, but the code is attempting to delete a file that no longer exists.

Implementaciones de FileSystem API

Implementaciones de FileSystem API



Como podemos ver, FileSystem API en Node.js es muy útil para realizar tareas de administración de archivos y directorios en aplicaciones backend.

Te compartimos, a continuación, algunas tareas más cercanas al día día, que se pueden realizar utilizando FileSystem API:

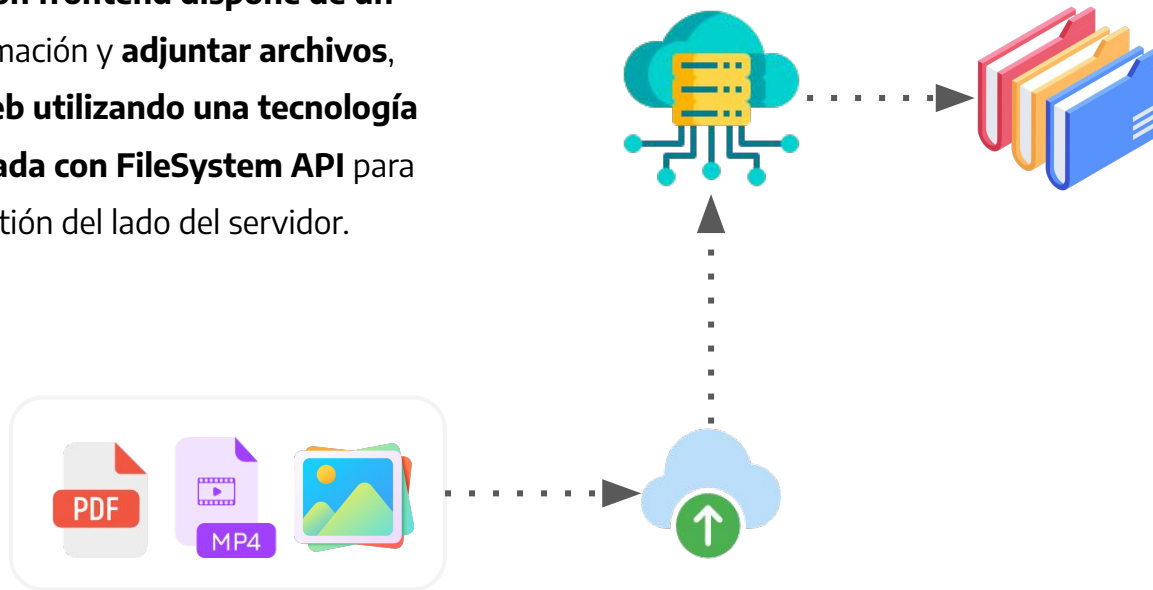
Implementaciones de FileSystem API

Estado	Descripción
Leer y escribir archivos	leer archivos existentes y/o escribir nuevos archivos en el sistema de archivos del servidor.
Borrar de archivos	eliminar archivos existentes del sistema de archivos del servidor.
Gestionar directorios y archivos	crear nuevos directorios, leer y enumerar el contenido de éste y eliminar los existentes. También copiar y mover archivos de un lugar a otro en el sistema de archivos del servidor.
Manipulación de archivos temporales	crear y manipular archivos temporales que se utilizan para almacenar datos temporales, por ejemplo, durante el procesamiento de una solicitud.
Procesamiento de archivos de registro	leer archivos de registro y procesarlos para extraer información útil o realizar análisis.



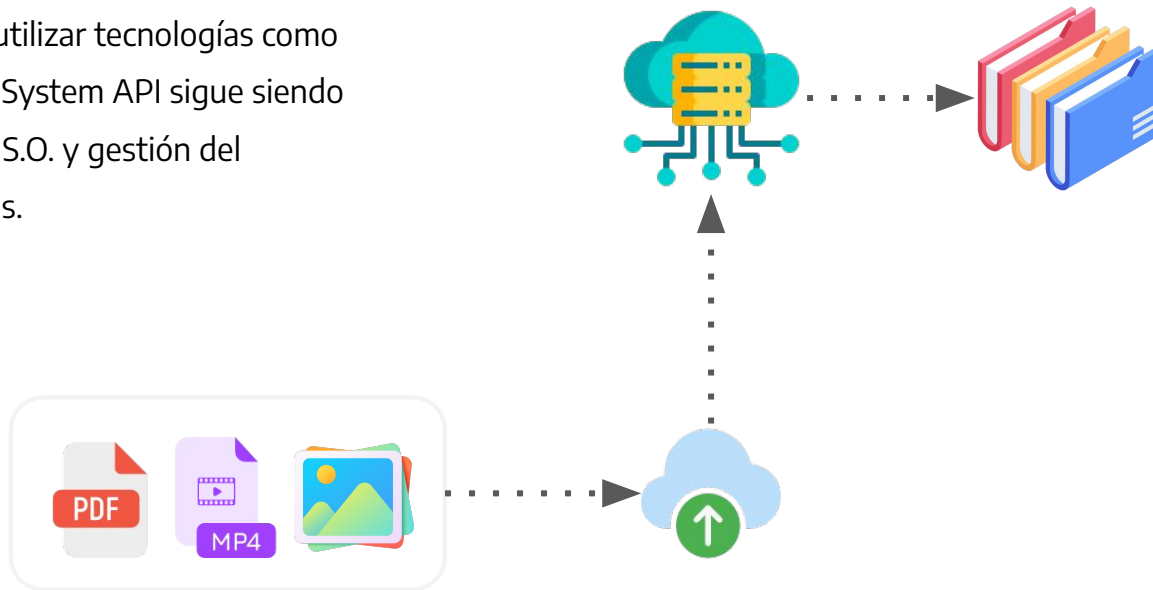
Implementaciones de FileSystem API

También, **cuando una aplicación frontend dispone de un formulario** de llenado de información y **adjuntar archivos**, **estos se suben al servidor web utilizando una tecnología denominada Stream, combinada con FileSystem API** para escribir el/los archivo(s) en cuestión del lado del servidor.

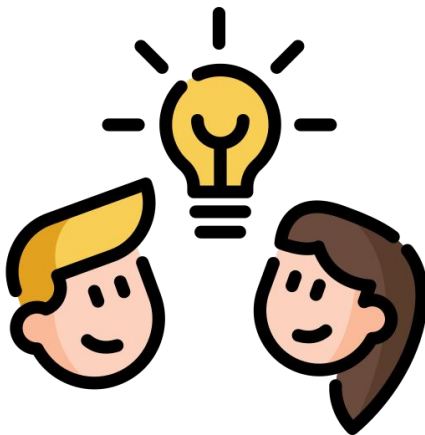


Implementaciones de FileSystem API

Si bien para esto último, debes utilizar tecnologías como [busboy](#) o [multer](#), el uso de FileSystem API sigue siendo requerido para el diálogo con el S.O. y gestión del contenido de archivos y carpetas.



Implementaciones de FileSystem API



FileSystem API en Node.js es muy útil en aplicaciones backend para todo tipo de tareas de administración de archivos y directorios, lo que permite que una aplicación interactúe con el sistema de archivos del servidor para leer, escribir, eliminar y manipular archivos y directorios.

Manejo de Directorios

Manejo de directorios

¿Y cómo estructuramos un cúmulo de archivos generados con 'fs'?

El mismo módulo FileSystem nos permite trabajar con carpetas o directorios del servidor. Como vimos al inicio de esta presentación, existen métodos que nos permiten crear, leer y actualizar directorios, como también eliminarlos. Veamos entonces cuáles son y cómo se implementan.



Manejo de directorios

Estado	Descripción
.mkdir()	crea un nuevo directorio dentro de la ruta de nuestro proyecto.
.readdir()	realiza una lectura del contenido especificado. Como respuesta, nos devuelve un array de nombres de archivos y directorios.
.rename()	cambia el nombre del directorio especificado como parámetro.
.rmdir()	elimina el nombre de un directorio del sistema de archivos del servidor.
Procesamiento de archivos de registro	leer archivos de registro y procesarlos para extraer información útil o realizar análisis.

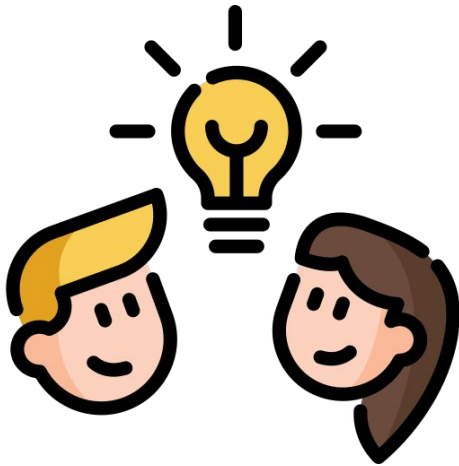
Manejo de directorios



Muchos de estos métodos respetan el nombre de los comandos dentro de los File System tipo Linux - Mac.

Si tienes experiencia en los mismos, serán fáciles de recordar, sino, aprender estos métodos te ayudarán mucho a luego poder manejar esta estructura de directorios sin problema.

Manejo de directorios



El módulo FileSystem cuenta con todas las herramientas necesarias para trabajar en profundidad con archivos y directorios.

Son simples de utilizar pero, en determinadas situaciones, requieren de cuidados particulares porque la reescritura de contenido o eliminación de un archivo o carpeta, debe ser controlada por la lógica de nuestro código,

Mecanismos de codificación

Mecanismos de codificación



Los mecanismos de codificación suelen utilizarse para tomar una porción de texto, legible por cualquier persona, y convertir dicha cadena de caracteres en lo que se conoce como una cadena codificada.

A diferencia de la encriptación, la cual utiliza mecanismos de codificación más complejos, la codificación de cadenas de textos es algo más fácil de manejar aunque no convierte los datos codificados en información 100% segura.

Base64



La codificación Base64 es un proceso en el que se toma un conjunto de datos binarios y se los convierte en una cadena de texto ASCII.

Esta cadena de texto puede ser transmitida a través de redes que sólo admiten caracteres ASCII, como por ejemplo en los correos electrónicos. Luego, el receptor puede decodificar la cadena Base64 para obtener los datos originales.

Base64

Base64

JavaScript incluye funciones nativas para codificar y decodificar texto utilizando Base64.

- **función btoa():** se utiliza para codificar una cadena de texto en Base64
- **función atob():** se utiliza para decodificar una cadena Base64



Veamos unos ejemplos a continuación.

Codificar Base64

Aquí un ejemplo de cómo **codificar una cadena de texto en Base64** utilizando Node.js:

```
Base64

const textoOriginal = "Hola, mundo!";
const textoCodificado = Buffer.from(textoOriginal).toString('base64');
console.log(textoCodificado);
```

Codificar Base64

El resultado en formato codificado del ejemplo anterior, sería:

```
Base64

//Resultado del texto codificado anteriormente (sin las comillas)

'SG9sYSwgbXVuZG8h'
```


Decodificar Base64

Para decodificar una cadena Base64, utilizamos el siguiente código:

```
Base64

const textoCodificado = "SG9sYSwgbXVuZG8h";
const textoDecodificado = Buffer.from(textoCodificado, 'base64').toString('ascii');
console.log(textoDecodificado);
```

Decodificar Base64

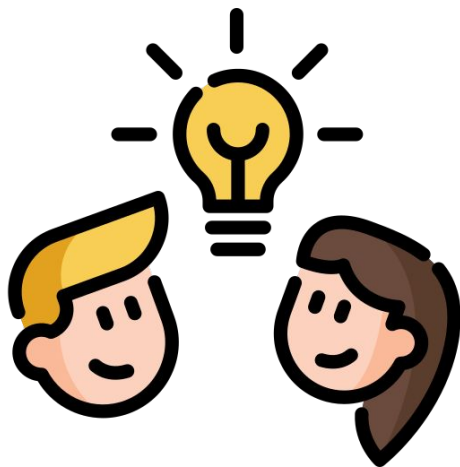
Nuevamente, obtenemos el texto decodificado, tal como lo generamos inicialmente.

```
Base64

//Resultado del texto decodificado (sin las comiilas)

'Hola, mundo!'
```

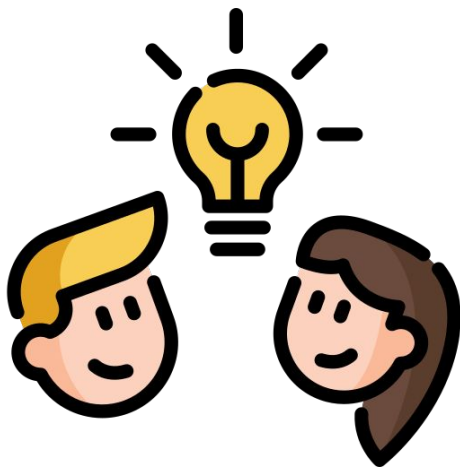
Base64



Estos mecanismos de codificación son utilizados por muchas aplicaciones, desde que la computación es tal.

Por ejemplo, el cuerpo de los Emails, si miramos un archivo de correo electrónico .EML convencional, tiene gran parte de su estructura codificada en Base64.

Base64



Este formato también es muy común encontrarlo aplicado para codificar imágenes y archivos de texto ligero, que deben guardar su contenido en una base de datos.

Es óptimo para, por ejemplo, codificar imágenes que requieran guardarse en una base de datos, y no como un archivo de imagen.

Base64

Codificar una imagen como la siguiente en formato Base64, nos devuelve una estructura de texto ASCII como la que vemos aquí abajo (*sin las comillas*).

El sitio web base64decode.org nos puede servir para investigar más en profundidad.



smile.png

```
Base64

'(data:image/png;base64,iVBORw0KGgoAAAANSUHEUgAAADIAAAAYCAYAAAAeP4ixAAAAAXNSR0I
Ars4c6QAAAAARnQU1BAACxjwv8YQAAAAAGUEXURQAAAAAAD/+v79/f3///M4Gq3AAAAA3dFJ0UwAQD
xwBPwj2bd0zAAAAfULEQVRo3u3YsQ0AIAzDsDsP7v+8igBknV7cavdSKsJ+7Bv8trYIPV7QjJrgBAAA
AP//b+/k+QAAABrSURBV0dEzY6FoMwGIUfFC6JdF6ZDR0g1Q2zF/8/3r3zD0M6LBIU6CKU6CKU6C
KU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6CKU6
K5CYII='
```

Prácticas

Prácticas

Al proyecto construído en las prácticas de la clase anterior, debemos sumarle el archivo con la estructura .JSON que nos compartirá la Profe.

Integraremos este archivo con los endpoint que creamos la clase anterior. Los mismos eran:

- /productos
- /productos/:id
- /productos/:nombre



Prácticas

Este archivo con extensión **.json**, debe ser leído utilizando el módulo FileSystem API. Luego, su contenido debe ser convertido a un array de objetos JS, y almacenado en una variable.

Finalmente, integrarás la lógica necesaria en los tres endpoints creados, para poder:

- listar todos los objetos del array
- devolver un objeto buscando el mismo por su propiedad **:id**
- devolver uno o más objetos filtrando los mismos por su propiedad nombre. *(puede recibir parte del nombre)*



Muchas gracias.



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*