

# Desarrollo Backend

Bienvenid@s

FileSystem API

Clase 07



Pon a grabar la clase



# Temario

- Introducción a Filesystem API
  - Qué es **FileSystem API**
  - Cómo incluirlo en nuestros proyectos **Node.js**
- Manipulación de archivos con el módulo **fs**
  - el método **.readFile()**
  - el método **.writeFile()**
  - el método **.appendFile()**
  - Borrar archivos
- Manipulación de Directorios
  - el método **.mkdir()**
  - el método **.readdir()**
  - el método **.rename()**
  - el método **.rmdir()**



# Introducción a 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

# Qué es FileSystem API

## ***Refresquemos conceptos:***

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 S.O.

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



# Qué es FileSystem API



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.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Qué es 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:





# Qué es FileSystem API

Método	Descripción
<b>readFile()</b>	Nos permite leer el contenido de un archivo plano.
<b>writeFile()</b>	Nos permite escribir dentro de un archivo plano, el contenido que necesitemos volcar.
<b>appendFile()</b>	Nos permite agregar contenido dentro de un archivo ya creado, respetando lo que éste tenga previamente almacenado.
<b>unlink()</b>	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

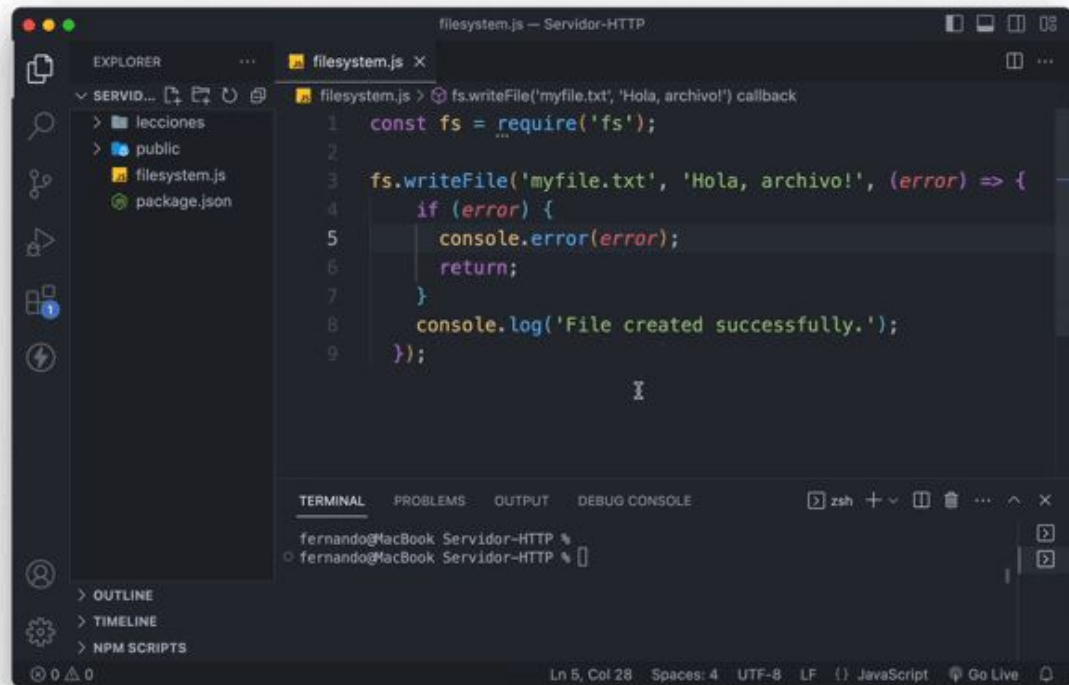
```
FileSystem API

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.');
```

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.

# Crear un archivo



The screenshot shows a VS Code editor window titled 'filesystem.js - Servidor-HTTP'. The Explorer sidebar on the left shows a project structure with folders 'lecciones' and 'public', and files 'filesystem.js' and 'package.json'. The main editor area displays the following JavaScript code in 'filesystem.js':

```
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 bottom of the window features a terminal panel with the prompt 'fernando@MacBook Servidor-HTTP' and a cursor.

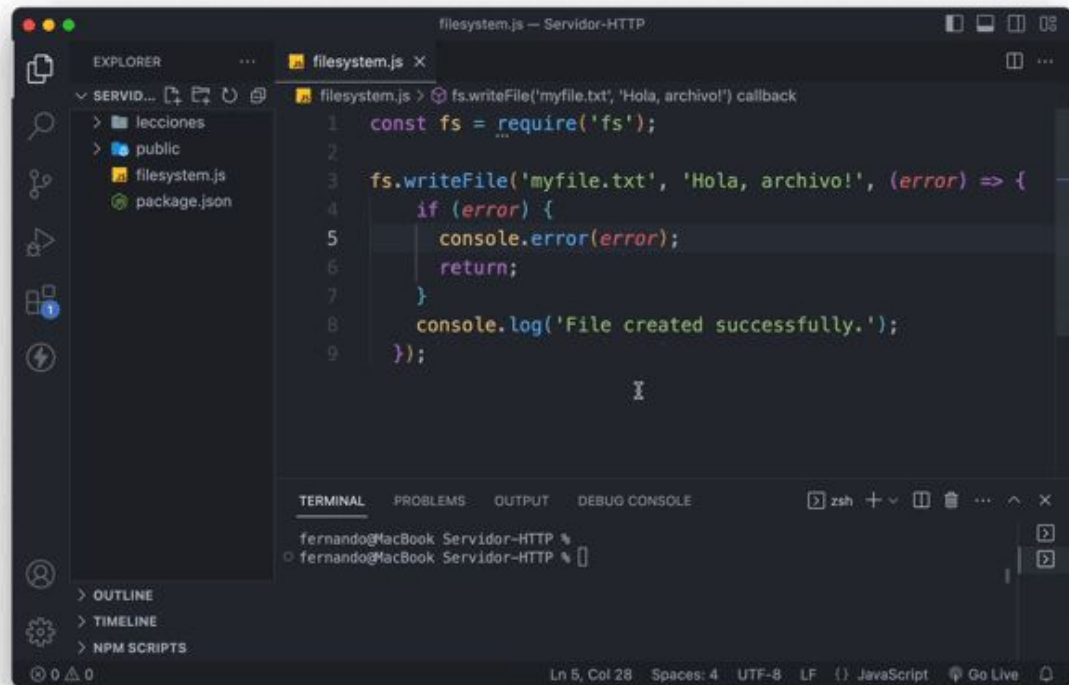
Como podemos ver en el ejemplo, el proceso es simple y directo.

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

# Crear un archivo

No hay verificaciones ni validaciones ni nada por el estilo de por medio.

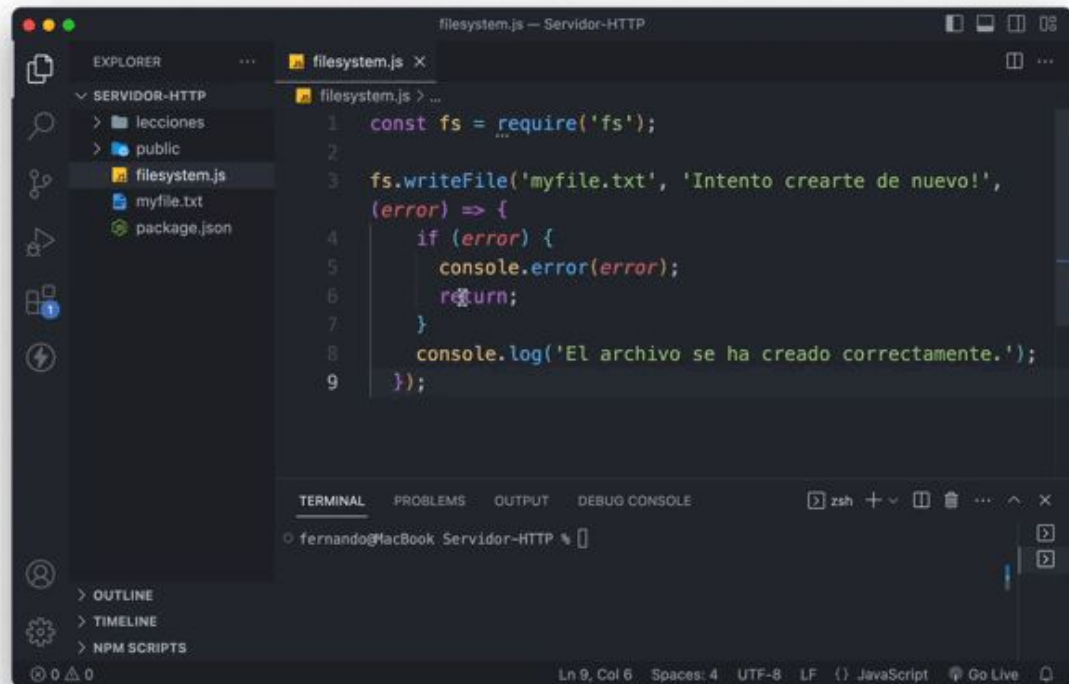
Esto, si bien es un beneficio para nosotros, también **debe ser una advertencia**, porque **si el archivo fue creado anteriormente y tiene contenido, podemos sobreescribirlo.**



```
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.');
```

# Crear un archivo

**Aquí, un ejemplo de sobreescritura sin advertencias de contenido.**



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

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.



# Crear un archivo

```
FileSystem API

const fs = require('fs');

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

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.

# Crear un archivo

```
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.');
```

```
});
}
```

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.

# Leer un archivo

# Leer un archivo

El método **.readFile()** nos permite leer un archivo. Este ya lo utilizamos anteriormente, cuando creamos un servidor web para nuestro sitio frontend.

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



# Leer un archivo

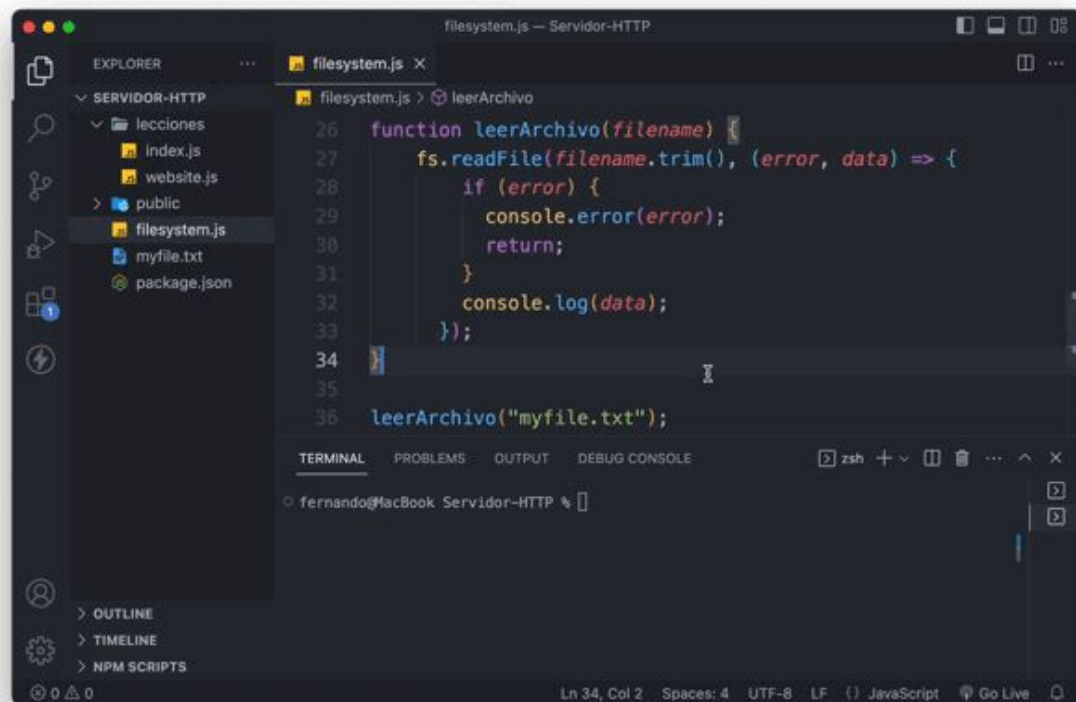
```
FileSystem API

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.

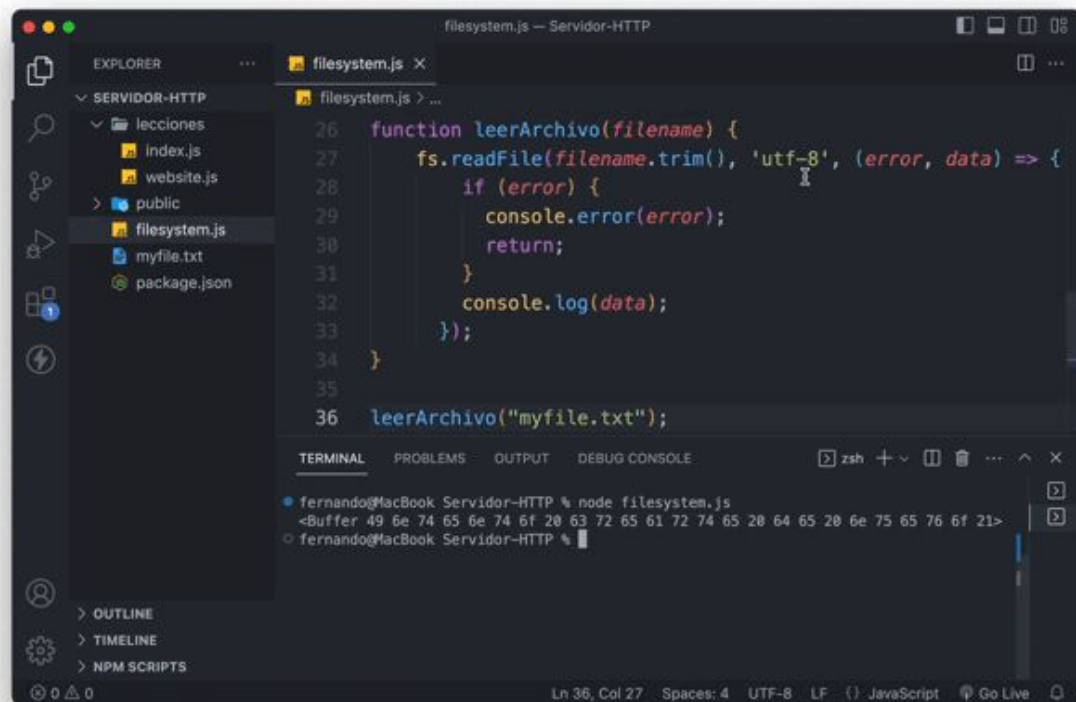
*¿Recuerdas por qué sucede esto?*

# Leer un archivo



**Cuando leíamos contenido de un archivo 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



```
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");
```

```
fernando@MacBook Servidor-HTTP % node filesystem.js
<Buffer 49 6e 74 65 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()**.

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO



# 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 un poco 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.

```
FileSystem API

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

# Escribir contenido en un archivo

```
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.');
```

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

# 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.



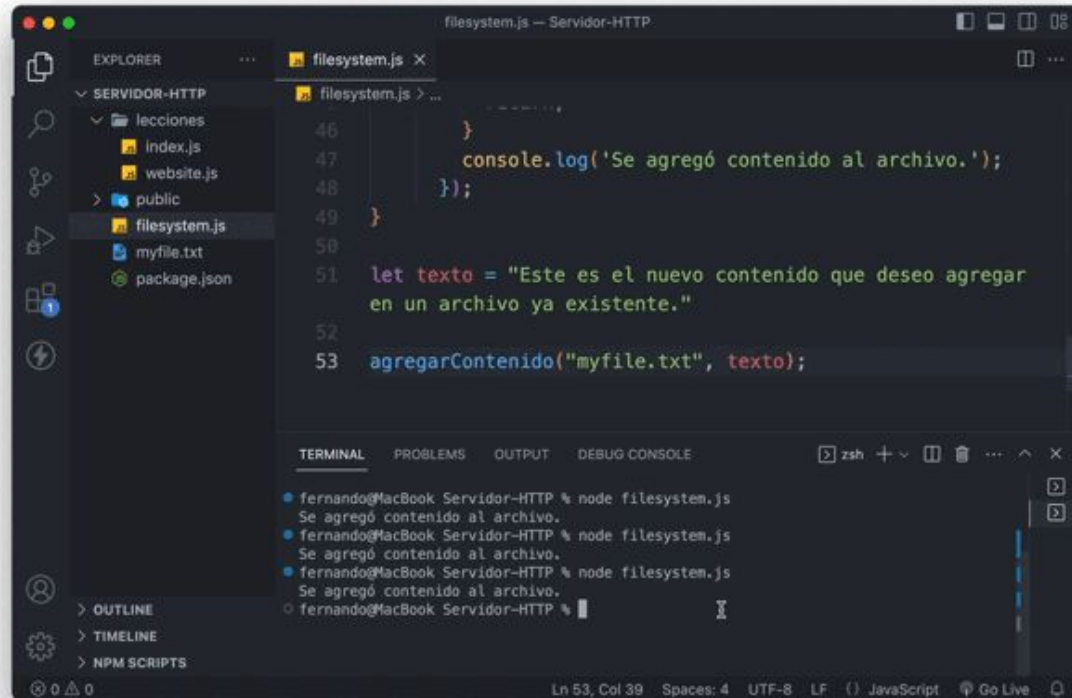
FileSystem API

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

**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Escribir contenido en un archivo



```
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);
```

TERMINAL

```
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 %
```

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.

# 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

```
FileSystem API

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

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.

# Borrar un archivo

```
FileSystem API

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

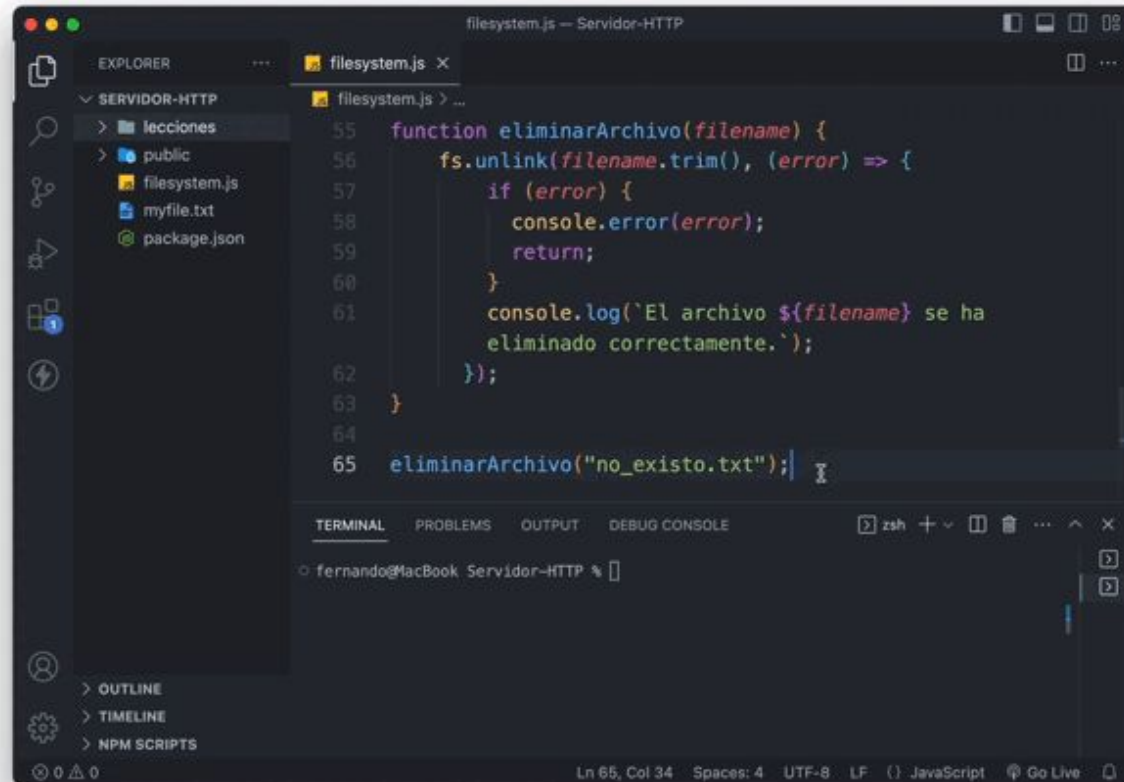
Armamos una función dedicada para eliminar archivos, pasándole como parámetro el nombre de éste.

```
FileSystem API

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

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

# Borrar un archivo



```
filesystem.js — Servidor-HTTP
EXPLORER
  SERVADOR-HTTP
    lecciones
    public
    filesystem.js
    myfile.txt
    package.json
  OUTLINE
  TIMELINE
  NPM SCRIPTS

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");
```

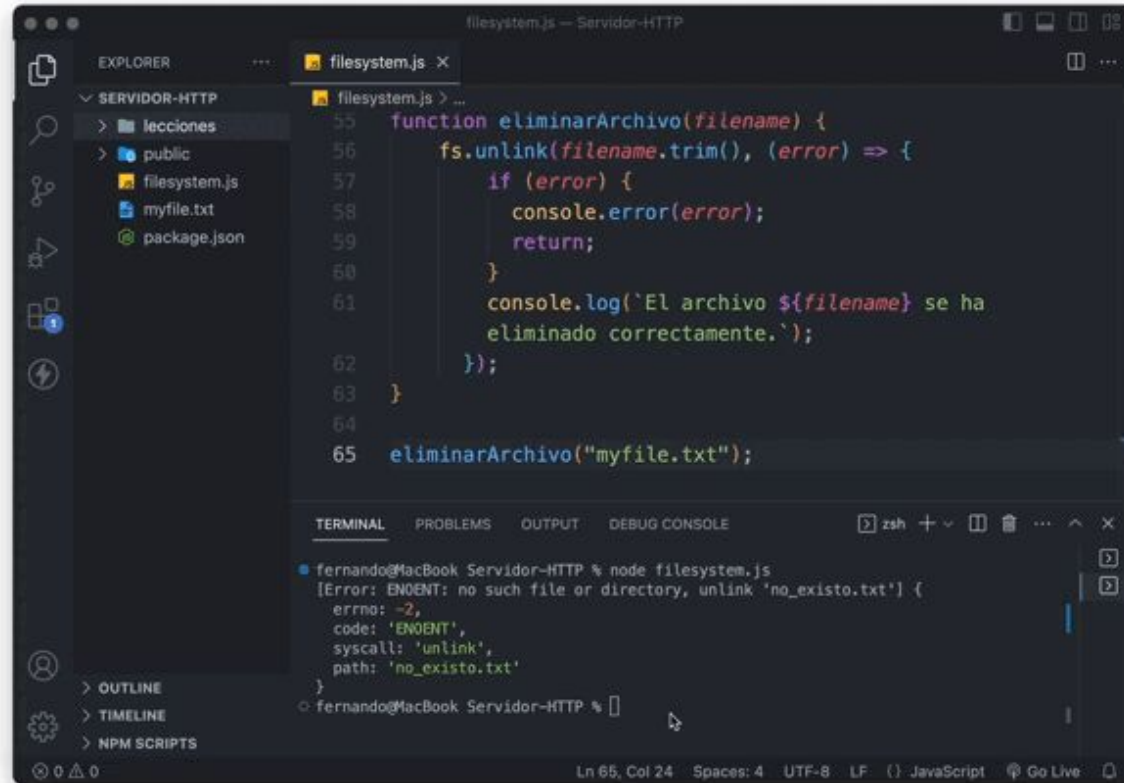
TERMINAL

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

Ln 65, Col 34 Spaces: 4 UTF-8 LF () JavaScript Go Live

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.

# Borrar un archivo



The screenshot shows a Visual Studio Code editor with a project named 'SERVIDOR-HTTP'. The Explorer sidebar on the left shows the file structure: 'lecciones', 'public', 'filesystem.js', 'myfile.txt', and 'package.json'. The main editor window displays the 'filesystem.js' file with the following code:

```
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");
```

Below the editor, the TERMINAL panel shows the command 'node filesystem.js' being executed. The output displays an error message: '[Error: ENOENT: no such file or directory, unlink 'no\_existo.txt'] { errno: -2, code: 'ENOENT', syscall: 'unlink', path: 'no\_existo.txt' }'. The status bar at the bottom indicates 'Ln 65, Col 24', 'Spaces: 4', 'UTF-8', 'LF', 'JavaScript', and 'Go Live'.

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

# 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:



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

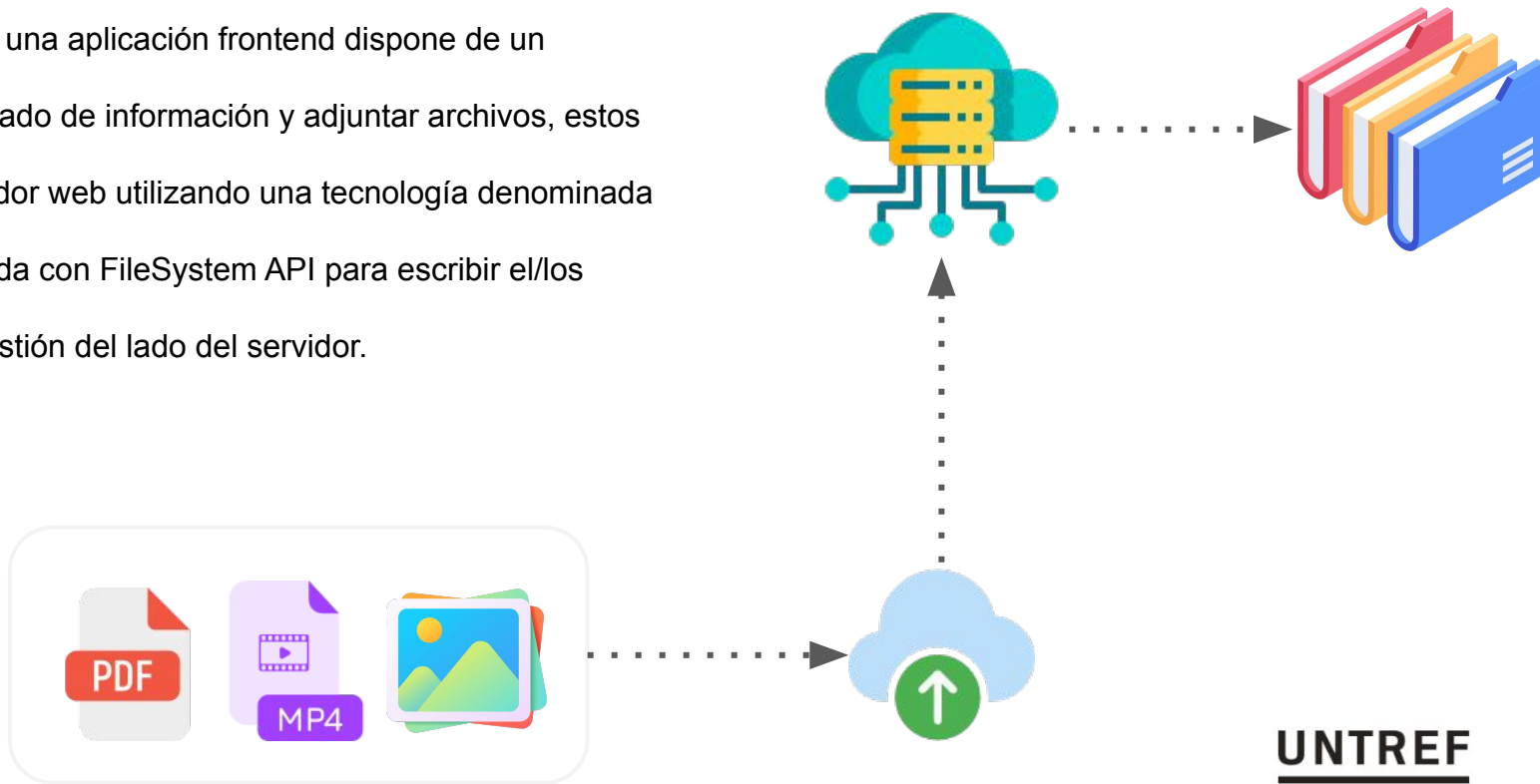


# Implementaciones de FileSystem API

Estado	Descripción
<b>Leer y escribir archivos</b>	leer archivos existentes y/o escribir nuevos archivos en el sistema de archivos del servidor.
<b>Borrar de archivos</b>	eliminar archivos existentes del sistema de archivos del servidor.
<b>Gestionar directorios y archivos</b>	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.
<b>Manipulación de archivos temporales</b>	crear y manipular archivos temporales que se utilizan para almacenar datos temporales, por ejemplo, durante el procesamiento de una solicitud.
<b>Procesamiento de archivos de registro</b>	leer archivos de registro y procesarlos para extraer información útil o realizar análisis.

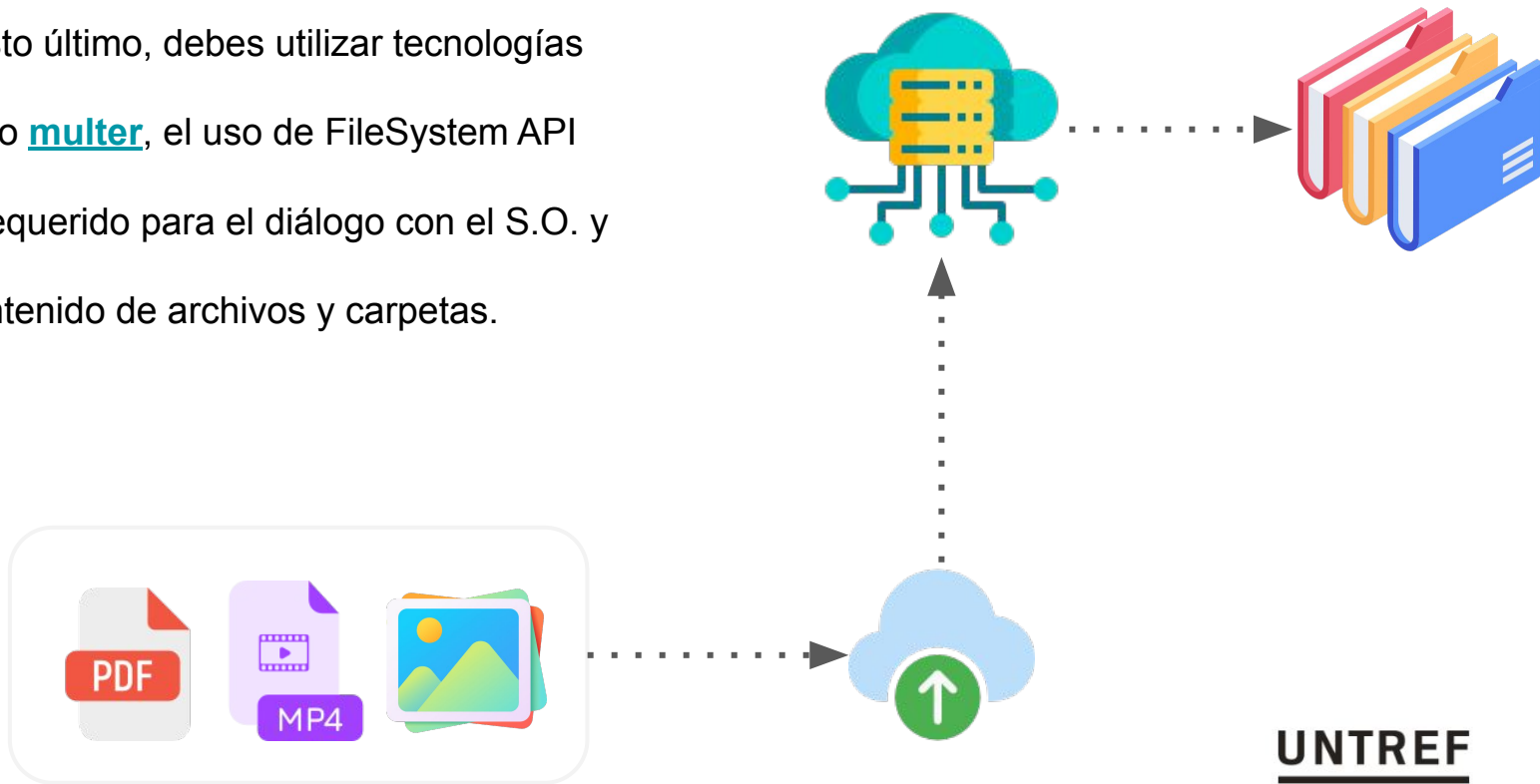
# Implementaciones de FileSystem API

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.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Trabajo con Directorios / Carpetas

# Trabajo con Directorios / Carpetas

¿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.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Trabajo con Directorios / Carpetas

## Crear un directorio:

El método **.mkdir()** nos permite realizar esta tarea.

En este código de ejemplo, **fs.mkdir()** crea un nuevo directorio llamado "prueba" dentro de la ruta de nuestro proyecto. Si ocurre un error al crearlo, se muestra un mensaje en la consola.

```
FileSystem API

fs.mkdir('/prueba', (err) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Directorio creado con éxito');
  }
});
```

# Trabajo con Directorios / Carpetas

## Leer el contenido de un directorio:

El método `.readdir()` permite realizar una lectura del contenido especificado.

Como respuesta, nos **devuelve un array** de nombres de archivos y directorios, a través de la variable `'files'`. Si ocurre un error al leer el directorio, se muestra un mensaje en la consola.

```
FileSystem API

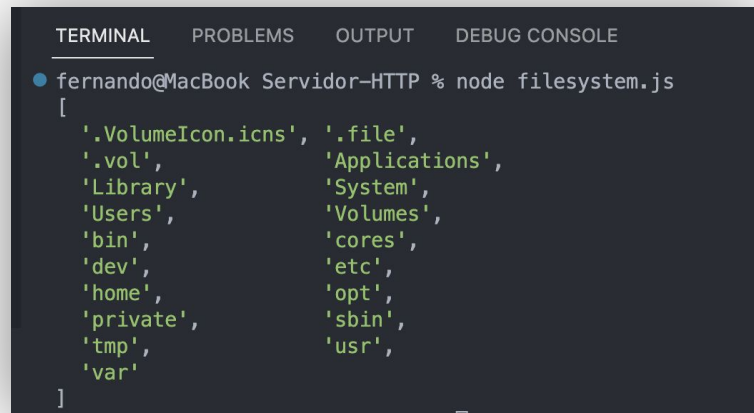
fs.readdir('/', (err, files) => {
  if (err) {
    console.error(err);
  } else {
    console.log(files);
  }
});
```



# Trabajo con Directorios / Carpetas

En el ejemplo de código anterior, establecimos como directorio base, el parámetro `'/'`. Esto leyó el punto raíz de la unidad de almacenamiento y devolvió todas las carpetas y archivos encontrados en éste, incluso aquellos ocultos.

En la imagen ilustrativa, identificamos archivos y directorios ocultos, con un `.` (*punto*) delante de su nombre.



```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE
fernando@MacBook Servidor-HTTP % node filesystem.js
[
  '.VolumeIcon.icns', '.file',
  '.vol',              'Applications',
  'Library',          'System',
  'Users',            'Volumes',
  'bin',              'cores',
  'dev',              'etc',
  'home',             'opt',
  'private',          'sbin',
  'tmp',              'usr',
  'var'
]
```

# Trabajo con Directorios / Carpetas

## Renombrar un directorio:

El método **.rename()** cambia el nombre del directorio especificado como parámetro.

Si ocurre un error al realizar este cambio, se muestra un mensaje en la consola.

```
FileSystem API

fs.rename('prueba', 'nueva-prueba', (err) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Directorio renombrado con éxito');
  }
});
```

# Trabajo con Directorios / Carpetas

## Eliminar un directorio:

El método `.rmdir()` elimina el nombre de un directorio del sistema de archivos del servidor.

Si ocurre un error al realizar este cambio, se muestra un mensaje en la consola.

```
FileSystem API

fs.rmdir('nueva-prueba', (err) => {
  if (err) {
    console.error(err);
  } else {
    console.log('Directorio eliminado con éxito');
  }
});
```

# Trabajo con Directorios / Carpetas

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.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Trabajo con Directorios / Carpetas

Y, si utilizas Windows, los nombres de los comandos son algo diferentes en la **Command Line**, pero puedes recurrir a [instalar PowerShell](#), una aplicación de Microsoft que cambia la estructura de archivos y directorios de Windows **a través de una línea de comandos diferente**, haciéndola parecida a la interfaz Linux.

Y lo mejor de todo, es que podrás unificar los comandos para manejar archivos y carpetas en Windows, tal como lo harías con Linux ó Mac.



# Trabajo con Directorios / Carpetas

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.



**UNTREF**

UNIVERSIDAD NACIONAL  
DE TRES DE FEBRERO

# Espacio de trabajo

# Espacio de trabajo

Tomar el ejemplo asincrónico de la explicación elaborada, y convertirlo a Promesas.

**Tiempo estimado: 20 minutos.** 



```
const questions = [ 'dudas', 'consultas', '🧐' ]
```



```
> node gracias.js
```