JS File Storage API

File Storage API is an interface for working with the file system. It provides a set of functions to perform various operations with files and folders directly from a **JavaScript** node.

To use the File Storage API, you need to import the built-in module `builtIn/FS`:

```
import * as fs from "builtIn/FS";
```

Object descriptions

Stats

▼ Description

```
class Stats {
    isFile(): boolean; // Check if the object is a file
    isDirectory(): boolean; // Check if the object is a directory
    isSymbolicLink(): boolean; // Check if the object is a symbolic
link

    mode: number; // A bitfield describing the file type and access
mode
    size: number; // File size in bytes
    atime: Date; // Time of last access to the object
    mtime: Date; // Time of last modification of the object
    ctime: Date; // Time of last status change of the object
    birthtime: Date; // Time of creation of the object
}
```

The `Stats` object provides information about a file system object.

FileHandle

▼ Description

```
class FileHandle {
    valueOf(): number; // Value of the file descriptor
}
```

The `FileHandle` object is a file descriptor.

Dirent

▼ Description

```
class Dirent {
    isFile(): boolean; // Check if the object is a regular file
    isDirectory(): boolean; // Check if the object is a directory
    isSymbolicLink(): boolean; // Check if the object is a symbolic
link
    name: string; // Name of the object
}
```

The `Dirent` object is a representation of a directory entry, which can be a file or a subdirectory.

File storage functions

▼ appendFileSync

`appendFileSync(file, data, options)`

- `file`: The full name of the file or a file descriptor. Accepts values of type `string` and `FileHandle`. Required parameter.
- `data`: The data. Accepts values of type `string`, `ArrayBuffer`, and `ArrayBufferView`. Required parameter.
- `options` (optional parameter):
  - writeBOM: Writes BOM. Accepts a value of type `boolean`.
  - `encoding`: File encoding. Default: **'utf8'**.
  - flag: A file system flag. Default: **'a'**.

Appends data to a file, creating the file if it does not yet exist. Returns `undefined`.

▼ closeSync

`closeSync(fd)`

- `fd`: A file descriptor. Required parameter.

Closes the file descriptor. Returns `undefined`.

▼ copyFileSync

`copyFileSync(src, dest, mode)`

- `src`: A path to the copy source file. Accepts a `string` value. Required parameter.
- `dest`: A path to the copy destination file. Accepts a `string` value. Required parameter.
- `mode`: An optional parameter that defines the behavior of the copy operation. Accepts a `number` value. Set by a wildcard (e.g., `constants.COPYFILE_EXCL | constants.COPYFILE_FICLONE`). The wildcard can consist of a combination of the following values:
  - `constants.COPYFILE_EXCL`: The copy operation will fail if `dest` already exists.
  - `constants.COPYFILE_FICLONE`: To use the copy-on-write technique, where no physical data transfer is performed. If not supported, a regular copy operation will be performed.
  - `constants.COPYFILE_FICLONE_FORCE`: To use the copy-on-write technique. If not supported, the copy will fail with an error.

Copies `src` to `dest`. By default, `dest` is overwritten if it already exists. Returns `undefined`.

▼ existsSync

existsSync(path)

- `path`: A path; accepts a `string` value. Required parameter.

Returns `true` if the path exists, `false` otherwise.

▼ fstatSync

`fstatSync(fd)`

- `fd`: A file descriptor. Accepts a value of type `FileHandle`. Required parameter.

Returns a Stats object for the file descriptor.

▼ ftruncateSync

```
ftruncateSync(fd, len)
```

- `fd` : A file descriptor. Accepts a value of type `FileHandle` . Required parameter.
- `len` : An optional parameter to specify the size in bytes. Accepts a `number` value.

Sets the length of a regular file with file descriptor `fd` to `len` bytes. If the file was longer before this operation, the truncated data is lost. If the file was shorter, it is extended; the added part is filled with null bytes. Returns `undefined` .

▼ lstatSync

```
lstatSync(path, options)
```

- `path` : A path. Accepts a `string` value. Required parameter.
- `options` (optional parameter):
  - `throwIfNoEntry` : Accepts a `boolean` value. Defines whether an exception will be thrown if the path does not exist in the file system. Default: `true` .

Returns a Stats object for `path` . If `path` is a path to a symbolic link, the function returns information about the link.

▼ mkdirSync

```
mkdirSync(path, options, mode)
```

- `path` : A path. Accepts a `string` value. Required parameter.
- `options` (optional parameter):
  - `recursive` : Specifies whether parent directories should be created. Accepts a `boolean` value. Default: `false` .
  - `mode` : Not used on Windows. Default: `0o777` .

Creates a directory. Returns `undefined` .

▼ openSync

```
openSync(path, flags, mode)
```

- `path` : A path. Accepts a `string` value. Required parameter.
- flags: File system flags to determine the file opening mode. Optional parameter. Default: — `r` .
- `mode` : Sets the file access permissions when the file is created. On Windows, affects the "Read-only" file attribute. Default: `0o666` . Accepts `number` and `string` values. More details on Wikipedia.

Returns an object representing the file descriptor.

▼ readdirSync

`readdirSync(path, options)`

- `path` : A path; accepts a value of type `string` . Required parameter.
- `options` (optional parameter):
  - `withFileTypes` : A parameter that determines the type of the function's return value. Default: `false` .

Reads the contents of a directory. If `withFileTypes` is `false` , returns an array of strings. If `true` , returns an array of `Dirent` objects.

▼ readFileSync

`readFileSync(path, options)`

- `path` : A path or a file descriptor. Accepts values of type `string` and `FileHandle` . Required parameter.
- `options` (optional parameter):
  - `encoding` : File encoding. Default: `'utf8'` .
  - flag: A file system flag. Default: `'r'` .

Returns the contents of the `path` file. If the `encoding` option is specified, the function returns a string; otherwise, it returns an `ArrayBuffer` .

▼ readSync

`readSync(fd, buffer, offset, length, position)`

- `fd` : A file descriptor. Accepts a value of type `FileHandle` .
- `buffer` : The buffer where data is written. Accepts values of type `ArrayBuffer` and `ArrayBufferView` . Required parameter.
- `offset` : The position in `buffer` to start writing data. Optional parameter. Accepts a `number` value.
- `length` : The number of bytes to read. Optional parameter. Accepts a `number` value.
- `position` : Specifies where to begin reading from in the file. If `position` is `null` or `-1` , data will be read from the current file position, and the file position will be updated. If `position` is an integer, the current file position is not changed. Optional parameter. Accepts a `number` value.

Reads a specified range of bytes from the file into the buffer. Returns the number of bytes read.

▼ realpathSync

`realpathSync(path)`

- `path` : The path; accepts a `string` value. Required parameter.

Returns the full path.

renameSync

`renameSync(oldPath, newPath)`

- `oldPath` : The old path. Accepts a value of type `string` . Required parameter.
- `newPath` : The new path. Accepts a value of type `string` . Required parameter.

Renames the file from `oldPath` to `newPath` . Returns `undefined` .

rmdirSync

`rmdirSync(path, options)`

- `path` : The path to the directory. Accepts a value of type `string` . Required parameter.
- `options` (optional parameter):
  - `recursive` : Accepts a `boolean` value. If `true` , performs recursive directory deletion. Default: `false` .

Deletes a directory. Returns `undefined` .

rmSync

`rmSync(path, options)`

- `path` : A path; accepts a value of type `string` . Required parameter.
- `options` (optional parameter):
  - `force` : Accepts a value of type `boolean` . When `true` , exceptions will be ignored if the path does not exist. Default: `false` .
  - `recursive` : Accepts a value of type `boolean` . If `true` , performs recursive directory deletion. Default: `false` .

Deletes files and directories. Returns `undefined` .

statSync

`statSync(path, options)`

- `path` : The path; accepts a value of type `string` . Required parameter.
- `options` (optional parameter):
  - `throwIfNoEntry` : Accepts a value of type `boolean` . Defines whether an exception will be thrown if the path does not exist in the file system. Default: `true` .

Returns a Stats object for `path` .

▼ truncateSync

`truncateSync(path, len)`

- `path` : The path. Accepts a value of type `string` . Required parameter.
- `len` : An optional parameter that specifies the size in bytes. Accepts a value of type `number` .

Sets the length of the regular file named `path` to `len` bytes. If the file was longer before this operation, the truncated data is lost. If the file was shorter, it is extended—the added part is filled with null bytes. Returns `undefined` .

▼ unlinkSync

`function unlinkSync(path)`

- `path` : The path. Accepts a value of type string. Required parameter.

Deletes a file. Returns `undefined` .

▼ writeFileSync

`writeFileSync(path, data, options)`

- `path` : A path or a file descriptor. Accepts values of type `string` and `FileHandle` . Required parameter.
- `data` : The file content. Accepts values of type `ArrayBuffer` , `ArrayBufferView` . Required parameter.
- `options` (an optional parameter):
  - `encoding` : File encoding. Default: `'utf8'` .
  - flag: A file system flag. Default: `'r'` .
  - `writeBOM` : writes BOM. Accepts a value of type `boolean` .
  - `mode` : Affects the file access permissions. Accepts a value of type `number` . More details.

Writes data to a file, creating a new one if it does not exist. Returns `undefined` .

▼ writeSync

`writeSync(fd, buffer, offset, length, position)`

- `fd` : A file descriptor (`FileHandle`). Represents an open file to which data will be written.
- `buffer` : The data buffer from which bytes will be written to the file. Required parameter. Can be an `ArrayBuffer` or `ArrayBufferView` object (e.g., `Uint8Array` , `Uint16Array` , etc.).
- `offset` : The position within `buffer` from which to start reading data for writing. Optional parameter. If not specified, the beginning of the buffer (`0`) is used by

default. Accepts a value of type `number`.
- `length`: The number of bytes to write from `buffer`. Optional parameter. If not specified, all bytes from the `offset` position to the end of the buffer are written. Accepts a `number` value.
- `position`: The position in the file where writing will begin. Optional parameter. Accepts a `number` value. If the value is `null` or the parameter is not specified, writing occurs at the current position. Negative values indicate the position relative to the end of the file.

`writeSync(fd, string, position, encoding)`

- `fd`: A file descriptor (`FileHandle`). Represents an open file to which data will be written.
- `string`: A string of text that will be written to the file. This parameter is required if this function signature is used. Accepts a value of type `string`.
- `position`: The position in the file where writing will begin. Optional parameter. Accepts a `number` value. If the value is `null` or the parameter is not specified, writing occurs at the current position. Negative values indicate the position relative to the end of the file.
- `encoding`: Defines the encoding of the string to be written to the file. Optional parameter. Accepts a value of type `Encoding`. If no value is specified, the default encoding UTF-8 is used. List of supported encodings

Writes `buffer` to the specified file.

File system flags

- `a`: open file for appending. The file is created if it does not exist.
- `ax`: same as `a`, but using this flag will cause an error if the path exists.
- `a+`: open file for reading and appending. The file is created if it does not exist.
- `ax+`: same as `a+`, but using this flag will cause an error if the path exists.
- `as`: open file for reading and appending in synchronous mode. The file is created if it does not exist. This flag disables system data caching on write. Data will be written directly to the disk insdtead of the system cache.
- `as+`: open file for appending in synchronous mode. The file is created if it does not exist. Disables system data caching on write. Data is written directly to the disk and not to the system cache.
- `r`: open file for reading. Using this flag will cause an error if the file does not exist.
- `r+`: open file for reading and writing. Using this flag will cause an error if the file does not exist.
- `rs+`: open file for reading and writing in synchronous mode. Disables system data caching on write. Data is written directly to the disk and not to the system cache.
- `w`: open file for writing. The file is created (if it does not exist), or its content is truncated (if it exists).

- `wx` : same as `w` , but will return an error if the path exists.
- `w+` : open file for reading and writing. The file is created (if it does not exist), or its content is truncated (if it exists).
- `wx+` : same as `w+` , but will return an error if the path exists.

List of supported encodings

- `latin1` or `binary` — the **ISO 8859-1 Latin 1** encoding which uses the first 256 Unicode code points.
- `utf-8` or `utf8` — the **UTF-8** encoding.
- `utf16le` , `utf-16le` , `ucs2` , or `ucs-2` — the **UTF-16 Llittle-endian** encoding.
- `cp<codepage>` — a single-byte code page.
- `iso-8859-<number>` .

Examples

Using the File Storage API

Getting a list of files

```
import * as fs from "builtIn/FS";

let path = "."; // Get the contents of the current directory
let dirents = fs.readdirSync(path, { withFileTypes: true }); // Get the
contents of the path directory as an array of Dirent objects
dirents.sort((e1, e2) => (+e2.isDirectory() - +e1.isDirectory()) ||
(e2.name > e1.name ? -1 : e1.name == e2.name)); // Sort the array of
Dirent objects
for (let i = 0, l = dirents.length; i < l; i++) {
    let dirent = dirents[i];
    let isDir = dirent.isDirectory(),
        name = dirent.name;
    let entry_info = isDir ? `${name}: folder` : `${name}: file`; // Check
if it is a directory or a file
    if (dirent.isSymbolicLink()) // Check if it is a symbolic link
        entry_info += ", symbolic link";
    if (!isDir) {
        let stat = fs.statSync(`${path}/${name}`); // Get file information
as a Stat object
        entry_info += `, size in bytes: ${stat.size}`;
    }
    console.log(entry_info);
}
```

Copying a file from the user folder to the Megaladata folder

```
fs.copyFileSync("/user/test.mgp", "/user/megaladata/test.mgp");
```

Moving a file from the user folder to the Megaladata folder

```
fs.renameSync("/user/test.mgp", "/user/megaladata/test.mgp");
```

Deleting the test.mgp file

```
fs.unlinkSync("/user/test.mgp")
```

Getting a list of Dirent objects

```
fs.readdirSync("/user/test.mgp", {withFileTypes: true})
```

Truncating a file to the first 10 bytes

```
fs.truncateSync("/user/megaladata/test.mgp", 10);
```

Copying a folder with its contents

```
// Import the module for working with the file system
import * as fs from "builtIn/FS";

// Recursive function to copy a folder and its contents

function copyFolderRecursive(srcPath, dstPath) {
    // Read the contents of the source folder (using withFileTypes to get
information about the item type)
    for (const entry of fs.readdirSync(srcPath, { withFileTypes: true }))
{
        // The name of the current item (file or folder)
        let name = entry.name;
        // Full path to the item in the source folder
        let src = srcPath + "/" + name;
        // Full path to the item in the destination folder
        let dst = dstPath + "/" + name;

        if (entry.isFile()) {
            // If the item is a file, copy it to the destination folder
            fs.copyFileSync(src, dst);
        } else if (entry.isDirectory()) {
            // If the item is a folder, create it in the destination
folder
            fs.mkdirSync(dst);
            // Recursively copy the contents of this folder
            copyFolderRecursive(src, dst);
        }
    }
}

// Main function to copy a folder
function copyFolder(srcPath, dstPath) {
    // Check if the destination folder exists
    if (!fs.existsSync(dstPath)) {
        // If the destination folder does not exist, create it
        fs.mkdirSync(dstPath);
    }
    // Start the recursive copy of the source folder's contents to the
destination folder
```

```
    copyFolderRecursive(srcPath, dstPath);
}

// Example usage: copy the "/user/data" folder to "/user/megaladata"
copyFolder("/user/data", "/user/megaladata");
```

Writing data to a file

```
import * as fs from "builtIn/FS";

function str2ab(str) {
  let buf = new ArrayBuffer(str.length * 2); // Creates a buffer of 2
bytes for each character of the string
  let bufView = new Uint16Array(buf); // Creates a Uint16Array view for
the ArrayBuffer
  for (let i = 0, strLen = str.length; i < strLen; i++)
    bufView[i] = str.charCodeAt(i); // Writes the character code of each
character to the buffer
  return buf;
}

var bom_utf16 = new ArrayBuffer(2); // Creates a 2-byte ArrayBuffer
var bufView = new Uint16Array(bom_utf16); // Creates a Uint16Array to work
with the buffer
bufView[0] = 0xFEFF; // Writes BOM (0xFEFF) to the buffer

let buf = str2ab("987"); // Converts the string "987" to an ArrayBuffer

let fd = fs.openSync("new.txt", "w"); // Opens a file for writing. Creates
it if the file does not exist
try {
try {
  fs.writeSync(fd, bom_utf16); // Writes BOM to the beginning of the file
  fs.writeSync(fd, buf, 2, 2); // Writes part of the buffer starting from
the 2nd byte, 2 bytes (the file will contain the string "8")
  fs.writeSync(fd, "222", null, "utf16le"); // Writes the string "222" in
UTF-16LE encoding (the file will contain the string "8222")
  fs.writeSync(fd, "33", -4, "utf16le"); // Writes the string "33" in UTF-
16LE encoding, starting from the -4 position (the file will contain the
string "8233"
} finally {
  fs.closeSync(fd); // Closes the file
}
```

Appending data with a specified encoding

```
fs.appendFileSync('data.txt', 'Hello, world!', 'utf8');
```

Determining the absolute path

```
const realPath = fs.realpathSync(relativeFilePath); // relativeFilePath
contains the relative path
```

Read on: JS Debug Console