

# FILE SYSTEM SCRIPT INTERFACE

**View online:** <https://www.construct.net/en/make-games/manuals/construct-3/scripting/scripting-reference/plugin-interfaces/file-system>

The `IFileSystemObjectType` interface derives from `IObjectClass` to add APIs specific to the [File System plugin](#). Refer to the plugin documentation for details about permissions and known folders.

Note this class derives from the object class interface, not the instance interface. Typically it is used through `runtime.objects.FileSystem`.

## Common types

The following types are used across multiple APIs and so are described once here.

### Accept types

File open and save pickers allow specifying file type filters, also known as the accept types. Each accept type is specified by an object with the following properties:

- `description` : a description of this file type that may be shown to the user.
- `accept` : an object whose keys are MIME types, and values are an array of the corresponding file extensions, including the dot.

Here is an example for an accept type for common image file formats:

```
{
    description: "Images",
    accept: {
        "image/*": [".png", ".jpg", ".webp", ".avif"],
    }
}
```

### Start in locations

An initial folder location can be specified using a string which may be one of the following values: `"default"`, `"desktop"`, `"documents"`, `"downloads"`, `"music"`, `"pictures"`, `"videos"`.

## File System events

---

`"drop"`

Fired when the user drags and drops files in to the window. The event object has the property `files` which has an array of `File` objects representing the dropped files.

## File System APIs

---

### **addEventListener(eventName, callback)**

### **removeEventListener(eventName, callback)**

Add or remove a callback function for a File System event. See *File System events* above for more information.

---

### **isSupported**

A read-only boolean indicating if file system features are supported. This depends on support for the File System Access API in the browser. If false then none of the file system APIs will work.

---

### **desktopFeaturesSupported**

A read-only boolean indicating whether desktop-specific features are available. This is only the case when running after using a supported desktop export option. When true, the known folder pickers like `"<documents>"` can be used (providing they are available - use the `hasPickerTag()` method to check), as well as the desktop-specific methods like `runFile()` and `shellOpen()`.

---

### **hasPickerTag(pickerTag)**

Return a boolean indicating if a given picker tag has been remembered from a previous session, or if a known folder tag is available. In this case the picker tag can still be referred to for file system operations, such as to read a previously chosen file, without having to show a picker again.

---

### **async showSaveFilePicker(opts)**

Show a save file picker allowing the user to choose a file on their local system to save to. The `opts` parameter is an object which uses the following properties to specify options:

- `pickerTag` (required): a string of a tag to identify the chosen folder.
  
- `acceptTypes` : an array of objects used to filter the listed files. See *Accept types* above for more information.
  
- `showAcceptAll` : a boolean indicating whether to show an accept type that shows all kinds of files.

- `suggestedName` : a string used as the initial filename choice to save to in the picker.
- `id` : a string used as an extra identifier for remembering the picker settings, such as the last viewed folder.
- `startIn` : a string specifying the initial folder to show (when not remembered based on the `id`). See *Start in locations above for more information*.

The method returns a Promise that resolves with an array of strings for the selected file names, which in the case of a save file picker, will contain a single entry.

---

### **async showOpenFilePicker(opts)**

Show an open file picker allowing the user to choose one or multiple files on their local system to open. The `opts` parameter is an object which uses the following properties to specify options:

- `pickerTag` (required): a string of a tag to identify the chosen folder.
- `acceptTypes` : an array of objects used to filter the listed files. See *Accept types above* for more information.
- `showAcceptAll` : a boolean indicating whether to show an accept type that shows all kinds of files.
- `multiple` : a boolean indicating whether to allow the user to select multiple files.
- `id` : a string used as an extra identifier for remembering the picker settings, such as the last viewed folder.
- `startIn` : a string specifying the initial folder to show (when not remembered based on the `id`). See *Start in locations above for more information*.

The method returns a Promise that resolves with an array of strings for the selected file names.

---

### **async showFolderPicker(opts)**

Show a folder picker allowing the user to choose a folder on their local system. The `opts` parameter is an object which uses the following properties to specify options:

- `pickerTag` (required): a string of a tag to identify the chosen folder.
- `mode` : a string of either `"read"` or `"readwrite"` to specify whether read-only or read-write permission is to be granted for the chosen folder, which also affects the permission

prompt shown to the user.

- `id` : a string used as an extra identifier for remembering the picker settings, such as the last viewed folder.
- `startIn` : a string specifying the initial folder to show (when not remembered based on the `id`). See *Start in locations above for more information*.

The method returns a Promise that resolves with an array of strings for the selected folder names, which in the case of a folder picker, will contain a single entry.

---

### **async writeFile(opts)**

Write data to a previously chosen file. The `opts` parameter is an object which uses the following properties to specify options:

- `pickerTag` (required): a string of the picker tag used to choose the file or folder.
- `data` (required): a string or `ArrayBuffer` with the data to be written. Strings are written as text with UTF-8 encoding.
- `FolderPath` : when the picker tag refers to a folder, the relative path to the file inside that folder, e.g. "subfolder/file.txt".
- `mode` : a string of either `"overwrite"` or `"append"` for the file write mode. The default is to overwrite, replacing the entire file contents with the provided data. Append mode is only supported when writing text (passing a string for `data` ).
- `fileTag` : a file tag used by triggers in the event system.

The method returns a Promise that resolves when the write has completed.

---

### **async readFile(opts)**

Read data from a previously chosen file. The `opts` parameter is an object which uses the following properties to specify options:

- `pickerTag` (required): a string of the picker tag used to choose the file or folder.
- `mode` (required): a string of either `"text"` or `"binary"` to specify the type of the data returned. In text mode the returned Promise will resolve with a string, and in binary mode it will resolve with an `ArrayBuffer`.
- `FolderPath` : when the picker tag refers to a folder, the relative path to the file inside that folder, e.g. "subfolder/file.txt".

- `fileTag` : a file tag used by triggers in the event system.

The method returns a Promise that resolves with either a string when `mode` is `"text"`, or an ArrayBuffer when `mode` is `"binary"`.

### **async createFolder(pickerTag, folderPath, fileTag)**

Only applies to folder pickers. Create a subfolder within a previously picked folder.

`folderPath` specifies the folder to create. This can refer to multiple subfolders which will all be created, e.g. "subfolder/otherfolder". `fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves when the folder has been created.

### **async copyFile(pickerTag, srcFolderPath, destFolderPath, fileTag)**

Only applies to folder pickers. Copy a file within a previously picked folder. `srcFolderPath` specifies an existing file to copy, e.g. "subfolder/file1.txt". `destFolderPath` specifies where to create a copy; this will either create a new file if it doesn't exist, or overwrite an existing file if it already exists. `fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves when the file has been copied.

### **async moveFile(pickerTag, srcFolderPath, destFolderPath, fileTag)**

Only applies to folder pickers. Move a file within a previously picked folder. The file can be moved from one subfolder to another, or if it stays in the same folder, it renames the file.

`srcFolderPath` specifies the file to move, e.g. "subfolder/file1.txt". `destFolderPath` specifies where to move (or rename) the file to. `fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves when the file has been moved.

### **async delete(pickerTag, folderPath, isRecursive, fileTag)**

Only applies to folder pickers. Delete a file or folder within a previously picked folder.

`folderPath` specifies the file or folder to delete, e.g. "subfolder/file1.txt" or "subfolder".

`isRecursive` applies only when Folder path identifies a folder: if true it will delete all files and folders within the identified folder as well as the folder itself, but if false it will only successfully delete the folder if it is already empty. `fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves when the deletion has completed. Note that deleting a folder recursively may take a long time if it contains a large number of files/folders.

### **async listContent(pickerTag, folderPath, isRecursive, fileTag)**

Only applies to folder pickers. Retrieves a list of all files and folders within a previously picked folder. `folderPath` identifies a subfolder to list the contents for, or can be left empty to list the contents of the originally picked folder. If `isRecursive` is true it will also list all files and folders through subfolders of the specified folder.

Recursive listing may be slow with a very large folder that contains thousands of files/subfolders.

`fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves upon completion with an object with the following properties:

- `files` : an array of strings with file names in the specified folder.
- `folders` : an array of strings with folder names in the specified folder.

When `isRecursive` was true, the listed file and folder names may include slashes, such as `subfolder/file.txt`; for cross-platform consistency these always use forwards slashes, even on Windows.

### **async shellOpen(pickerTag, filePath, fileTag)**

Only supported in desktop exports when `desktopFeaturesSupported` is true. This action requests that the operating system shell (the component that handles the system user interface) opens the specified file. Typically this launches the default app associated with the file type and then opens the file in it, similar to double-clicking the file in the system file explorer. `fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves when the operation has completed.

### **async runFile(pickerTag, filePath, args, fileTag)**

Only supported in desktop exports when `desktopFeaturesSupported` is true. This action attempts to execute the provided file path. Command-line arguments can also be optionally provided as the string `args`. This can be used to run another program. Uniquely for this action the folder picker parameter can be left empty, in which case the provided file path is run directly; this can be useful to run system executables like "cmd.exe" on Windows.

`fileTag` is optional and used by triggers in the event system. Returns a Promise that resolves when the operation has completed.