

API for your everyday file system manipulations, much more convenient than fs or fs-extra. You will especially appreciate it as a scripting/tooling library and for your build pipelines.

## Table of Contents

Key Concepts

Getting Started

**API:**

append

copy

createReadStream

createWriteStream

cwd

dir

exists

file

find

inspect

inspectTree

list

move

path

read

remove

rename

symlink

tmpDir

write

---

## Key Concepts

### Why not use more than one CWD?

You can create many fs-jetpack objects with different internal working directories (which are independent from `process.cwd()`) and work on directories in a little more object-oriented manner.

```
1 const src = jetpack.cwd("path/to/source");
2 const dest = jetpack.cwd("/some/different/path/to/destination");
3 src.copy("foo.txt", dest.path("bar.txt"));
```

### JSON is a first class citizen

You can write JavaScript object directly to disk and it will be transformed into JSON automatically.

```
1 const obj = { greet: "Hello World!" };
2 jetpack.write("file.json", obj);
```

Then you can get your object back just by telling `read` method that it's a JSON.

```
1 const obj = jetpack.read("file.json", "json");
```

### Automatic handling of ENOENT errors

Everyone who has a lot to do with file system probably is sick of seeing error “*ENOENT, no such file or directory*”. Fs-jetpack tries to recover from this.

- For write/creation operations, if any of parent directories doesn't exist jetpack will just create them as well (like `mkdir -p` works).
- For read/inspect operations, if file or directory doesn't exist `undefined` is returned instead of throwing.

### Sync & async harmony

API has the same set of synchronous and asynchronous methods. All async methods are promise based (no callbacks).

Commonly used naming convention in node.js world has been flipped in this API, so no `method()` (async) and `methodSync()` naming. Here the convention is `methodAsync()` and `method()` (sync). I know this looks wrong to you, but bear with me. Thanks to that, you always know how fs-jetpack method behaves, just by looking at the name: **If you don't see the word “Async”, this**

---

**method returns value immediately, if you do, promise is returned.** Standard node.js naming can't give you this clarity.

```
1 // Synchronous call
2 const data = jetpack.read('file.txt');
3 console.log(data);
4
5 // Asynchronous call
6 const data = await jetpack.readAsync('file.txt');
7 console.log(data);
```

### All API methods cooperate nicely with each other

Let's say you want to create folder structure as demonstrated in comment below. Piece of cake!

```
1 // .
2 // |- greets
3 //   |- greet.txt
4 //   |- greet.json
5 // |- greets-il8n
6 //   |- polish.txt
7
8 jetpack
9   .dir("greets")
10   .file("greet.txt", { content: "Hello world!" })
11   .file("greet.json", { content: { greet: "Hello world!" } })
12   .cwd("..")
13   .dir("greets-il8n")
14   .file("polish.txt", { content: "Witaj świecie!" });
```

Need to copy whole directory of files, but first perform some transformations on each file?

```
1 const src = jetpack.cwd("path/to/source/folder");
2 const dst = jetpack.cwd("path/to/destination");
3
4 src.find({ matching: "*" }).forEach((path) => {
5   const content = src.read(path);
6   const transformedContent = transformTheFileHoweverYouWant(content);
7   dst.write(path, transformedContent);
8 });
```

Need to delete all temporary and log files inside `my_folder` tree?

```
1 jetpack.find("my_folder", { matching: ["*.tmp", "*.log"] }).forEach(
  jetpack.remove);
```

Need to perform temporary data transformations?

---

---

```
1 const dir = jetpack.tmpDir();
2 dir.write("data.txt", myData);
3 // Perform some operations on the data and when you're done
4 // and don't need the folder any longer just call...
5 dir.remove();
```

## Getting Started

### Installation

```
1 npm install fs-jetpack
```

Import to your code:

```
1 const jetpack = require("fs-jetpack");
```

### Usage with TypeScript

Starting from v2.1.0 fs-jetpack is TypeScript compatible. But for backwards compatibility purposes all types and interfaces are reachable through special path `fs-jetpack/types`.

```
1 // Import fs-jetpack into TypeScript code (the jetpack typings will be
   loaded as well).
2 import * as jetpack from "fs-jetpack";
3
4 // Import one of jetpack's interfaces to cast it on a variable
   declaration.
5 import { InspectResult } from "fs-jetpack/types";
6 let result: InspectResult = jetpack.inspect("foo");
```

### Upgrading to New Version

This API is considered stable and all breaking changes to it are done as completely last resort. It also uses “better safe than sorry” approach to bumping major version number. So in 99.9% of cases you can upgrade to latest version with no worries, because all major version bumps so far, were due to edge case behaviour changes.

---

## API

### **append(path, data, [options])**

asynchronous: **appendAsync(path, data, [options])**

Appends given data to the end of file. If file or any parent directory doesn't exist it will be created.

#### **arguments:**

**path** the path to file.

**data** data to append (can be `String` or `Buffer`).

**options** (optional) `Object` with possible fields:

- **mode** if the file doesn't exist yet, will be created with given mode. Value could be number (eg. `0o700`) or string (eg. `'700'`).

#### **returns:**

Nothing.

### **copy(from, to, [options])**

asynchronous: **copyAsync(from, to, [options])**

Copies given file or directory (with everything inside).

#### **arguments:**

**from** path to location you want to copy.

**to** path to destination location, where the copy should be placed.

**options** (optional) additional options for customization. Is an `Object` with possible fields:

- **overwrite** (default: **false**) Whether to overwrite destination path when it already exists. Can be `Boolean` or `Function`. If **false**, an error will be thrown if it already exists. If **true**, the overwrite will be performed (for directories, this overwrite consists of a recursive merge - i.e. only files that already exist in the destination directory will be overwritten). If a function was provided, every time there is a file conflict while copying the function will be invoked with inspect objects of both: source and destination file and overwrites the file only if **true** has been returned from the function (see example below). In async mode, the overwrite function can also return a promise, so you can perform multi step processes to determine if file should be overwritten or not (see example below).
- **matching** if defined will actually copy **only** items matching any of specified glob patterns and omit everything else (all possible globs are described further in this readme).

- 
- `ignoreCase` (default **false**) whether or not case should be ignored when processing glob patterns passed through the `matching` option.

**returns:**

Nothing.

**examples:**

```
1 // Copies a file (and replaces it if one already exists in 'foo'
  directory)
2 jetpack.copy("file.txt", "foo/file.txt", { overwrite: true });
3
4 // Copies files from folder foo_1 to foo_final, but overwrites in
5 // foo_final only files which are newer in foo_1.
6 jetpack.copy("foo_1", "foo_final", {
7   overwrite: (srcInspectData, destInspectData) => {
8     return srcInspectData.modifyTime > destInspectData.modifyTime;
9   }
10 });
11
12 // Asynchronously copies files from folder foo_1 to foo_final,
13 // but overwrites only files containing "John Doe" string.
14 jetpack.copyAsync("foo_1", "foo_final", {
15   overwrite: (srcInspectData, destInspectData) => {
16     return jetpack.readAsync(srcInspectData.absolutePath).then(data =>
17       {
18         return data.includes("John Doe");
19       }
20     ));
21
22 // Copies only '.md' files from 'foo' (and subdirectories of 'foo') to
23 // 'bar'.
24 jetpack.copy("foo", "bar", { matching: "*.md" });
25 // Copies only '.md' and '.txt' files from 'foo' (and subdirectories of
26 // 'foo') to 'bar'.
27 jetpack.copy("foo", "bar", { matching: ["*.md", "*.txt"] });
28
29 // You can filter previous matches by defining negated pattern further
30 // in the order:
31 // Copies only '.md' files from 'foo' (and subdirectories of 'foo') to
32 // 'bar'
33 // but will skip file '!top-secret.md'.
34 jetpack.copy("foo", "bar", { matching: ["*.md", "!top-secret.md"] });
35 // Copies only '.md' files from 'foo' (and subdirectories of 'foo') to
36 // 'bar'
37 // but will skip all files in 'foo/top-secret' directory.
38 jetpack.copy("foo", "bar", { matching: ["*.md", "!top-secret/**/*"] });
39
40 // All patterns are anchored to directory you want to copy, not to CWD.
41 // So in this example directory 'dir1/dir2/images' will be copied
```

---

```
37 // to 'copied-dir2/images'
38 jetpack.copy("dir1/dir2", "copied-dir2", {
39   matching: "images/**"
40 });
```

### **createReadStream(path, [options])**

Just an alias to vanilla `fs.createReadStream`.

### **createWriteStream(path, [options])**

Just an alias to vanilla `fs.createWriteStream`.

### **cwd([path...])**

Returns Current Working Directory (CWD) for this instance of jetpack, or creates new jetpack object with given path as its internal CWD.

**Note:** fs-jetpack never changes value of `process.cwd()`, the CWD we are talking about here is internal value inside every jetpack instance.

#### **arguments:**

`path...` (optional) path (or many path parts) to become new CWD. Could be absolute, or relative. If relative path given new CWD will be resolved basing on current CWD of this jetpack instance.

#### **returns:**

If `path` not specified, returns CWD path of this jetpack object. For main instance of fs-jetpack it is always `process.cwd()`.

If `path` specified, returns new jetpack object (totally the same thing as main jetpack). The new object resolves paths according to its internal CWD, not the global one (`process.cwd()`).

#### **examples:**

```
1 // Let's assume that process.cwd() outputs...
2 console.log(process.cwd()); // '/one/two/three'
3 // jetpack.cwd() will always return the same value as process.cwd()
4 console.log(jetpack.cwd()); // '/one/two/three'
5
6 // Now let's create new CWD context...
7 const jetParent = jetpack.cwd("..");
8 console.log(jetParent.cwd()); // '/one/two'
9 // ...and use this new context.
10 jetParent.dir("four"); // we just created directory '/one/two/four'
```

---

```
11
12 // One CWD context can be used to create next CWD context.
13 const jetParentParent = jetParent.cwd("../");
14 console.log(jetParentParent.cwd()); // '/one'
15
16 // When many parameters specified they are treated as parts of path to
  resolve
17 const sillyCwd = jetpack.cwd("a", "b", "c");
18 console.log(sillyCwd.cwd()); // '/one/two/three/a/b/c'
```

## **dir(path, [criteria])**

asynchronous: **dirAsync(path, [criteria])**

Ensures that directory on given path exists and meets given criteria. If any criterium is not met it will be after this call. If any parent directory in `path` doesn't exist it will be created (like `mkdir -p`).

If the given path already exists but is not a directory, an error will be thrown.

### **arguments:**

`path` path to directory to examine.

`criteria` (optional) criteria to be met by the directory. Is an `Object` with possible fields:

- `empty` (default: **false**) whether directory should be empty (no other files or directories inside). If set to **true** and directory contains any files or subdirectories all of them will be deleted.
- `mode` ensures directory has specified mode. If not set and directory already exists, current mode will be preserved. Value could be number (eg. `0o700`) or string (eg. `'700'`).

### **returns:**

New CWD context with directory specified in `path` as CWD (see docs of `cwd()` method for explanation).

### **examples:**

```
1 // Creates directory if doesn't exist
2 jetpack.dir("new-dir");
3
4 // Makes sure directory mode is 0700 and that it's empty
5 jetpack.dir("empty-dir", { empty: true, mode: "700" });
6
7 // Because dir returns new CWD context pointing to just
8 // created directory you can create dir chains.
9 jetpack
10   .dir("main-dir") // creates 'main-dir'
11   .dir("sub-dir"); // creates 'main-dir/sub-dir'
```



---

## **exists(path)**

asynchronous: **existsAsync(path)**

Checks whether something exists on given `path`. This method returns values more specific than `true`/`false` to protect from errors like “I was expecting directory, but it was a file”.

### **returns:**

- `false` if path doesn't exist.
- `"dir"` if path is a directory.
- `"file"` if path is a file.
- `"other"` if none of the above.

## **file(path, [criteria])**

asynchronous: **fileAsync(path, [criteria])**

Ensures that file exists and meets given criteria. If any criterium is not met it will be after this call. If any parent directory in `path` doesn't exist it will be created (like `mkdir -p`).

### **arguments:**

`path` path to file to examine.

`criteria` (optional) criteria to be met by the file. Is an `Object` with possible fields:

- `content` sets file content. Can be `String`, `Buffer`, `Object` or `Array`. If `Object` or `Array` given to this parameter data will be written as JSON.
- `jsonIndent` (defaults to 2) if writing JSON data this tells how many spaces should one indentation have.
- `mode` ensures file has specified mode. If not set and file already exists, current mode will be preserved. Value could be number (eg. `0o700`) or string (eg. `'700'`).

### **returns:**

Jetpack object you called this method on (self).

### **examples:**

```
1 // Creates file if doesn't exist
2 jetpack.file("something.txt");
3
4 // Creates file with mode '777' and content 'Hello World!'
5 jetpack.file("hello.txt", { mode: "777", content: "Hello World!" });
```

---

## **find([path], searchOptions)**

asynchronous: **findAsync([path], searchOptions)**

Finds in directory specified by `path` all files fulfilling `searchOptions`. Returned paths are relative to current CWD of jetpack instance.

### **arguments:**

`path` (optional, defaults to `'.'`) path to start search in (all subdirectories will be searched).

`searchOptions` is an `Object` with possible fields:

- `matching` (default `"*"`) glob patterns of files you want to find (all possible globs are described further in this readme).
- `filter` (default `undefined`) function that is called on each matched path with inspect object of that path as an argument. Return `true` or `false` to indicate whether given path should stay on list or should be filtered out (see example below).
- `files` (default `true`) whether or not should search for files.
- `directories` (default `false`) whether or not should search for directories.
- `recursive` (default `true`) whether the whole directory tree should be searched recursively, or only one-level of given directory (excluding it's subdirectories).
- `ignoreCase` (`false` otherwise) whether or not case should be ignored when processing glob patterns passed through the `matching` option.

### **returns:**

`Array` of found paths.

### **examples:**

```
1 // Finds all files inside 'foo' directory and its subdirectories
2 jetpack.find("foo");
3
4 // Finds all files which has 2015 in the name
5 jetpack.find("my-work", { matching: "*2015*" });
6
7 // Finds all '.txt' files inside 'foo/bar' directory and its
  subdirectories
8 jetpack.find("foo", { matching: "bar/**/*.txt" });
9 // Finds all '.txt' files inside 'foo/bar' directory WITHOUT
  subdirectories
10 jetpack.find("foo", { matching: "bar/*.txt" });
11
12 // Finds all '.txt' files that were modified after 2019-01-01
13 const borderDate = new Date("2019-01-01")
14 jetpack.find("foo", {
15   matching: "*.txt",
16   filter: (inspectObj) => {
```

---

```
17     return inspectObj.modifyTime > borderDate
18   }
19 });
20
21 // Finds all '.js' files inside 'my-project' but excluding those in '
  vendor' subtree.
22 jetpack.find("my-project", { matching: ["*.js", "!vendor/**/*"] });
23
24 // Looks for all directories named 'foo' (and will omit all files named
  'foo').
25 jetpack.find("my-work", { matching: ["foo"], files: false, directories:
  true });
26
27 // Finds all '.txt' files inside 'foo' directory WITHOUT subdirectories
28 jetpack.find("foo", { matching: "/*.txt" });
29 // This line does the same as the above, but has better performance
30 // (skips looking in subdirectories)
31 jetpack.find("foo", { matching: "*.txt", recursive: false });
32
33 // Path parameter might be omitted and CWD is used as path in that case
  .
34 const myStuffDir = jetpack.cwd("my-stuff");
35 myStuffDir.find({ matching: ["*.md"] });
36
37 // You can chain find() with different jetpack methods for more power.
38 // For example lets delete all '.tmp' files inside 'foo' directory
39 jetpack
40   .find("foo", {
41     matching: "*.tmp"
42   })
43   .forEach(jetpack.remove);
```

## inspect(path, [options])

asynchronous: **inspectAsync(path, [options])**

Inspects given path (replacement for `fs.stat`). Returned object by default contains only very basic, not platform-dependent properties (so you have something e.g. your unit tests can rely on), you can enable more properties through options object.

### arguments:

**path** path to inspect.

**options** (optional). Possible values:

- **checksum** if specified will return checksum of inspected file. Possible values are strings `'md5'`, `'sha1'`, `'sha256'` or `'sha512'`. If given path is a directory this field is ignored.
- **mode** (default **false**) if set to **true** will add file mode (unix file permissions) value.

- 
- **times** (default **false**) if set to **true** will add `atime`, `mtime` and `ctime` fields (here called `accessTime`, `modifyTime`, `changeTime` and `birthTime`).
  - **absolutePath** (default **false**) if set to **true** will add absolute path to this resource.
  - **symlinks** (default **'report'**) if a given path is a symlink by default **inspect** will report that symlink (not follow it). You can flip this behaviour by setting this option to **'follow'**.

**returns:** `undefined` if given path doesn't exist.

Otherwise `Object` of structure:

```
1 {
2   name: "my_dir",
3   type: "file", // possible values: "file", "dir", "symlink"
4   size: 123, // size in bytes, this is returned only for files
5   // if checksum option was specified:
6   md5: '900150983cd24fb0d6963f7d28e17f72',
7   // if mode option was set to true:
8   mode: 33204,
9   // if times option was set to true:
10  accessTime: [object Date],
11  modifyTime: [object Date],
12  changeTime: [object Date],
13  birthTime: [object Date]
14 }
```

## **inspectTree(path, [options])**

asynchronous: **inspectTreeAsync(path, [options])**

Calls `inspect` recursively on given path so it creates tree of all directories and sub-directories inside it.

### **arguments:**

**path** the starting path to inspect.

**options** (optional). Possible values:

- **checksum** if specified will also calculate checksum of every item in the tree. Possible values are strings **'md5'**, **'sha1'**, **'sha256'** or **'sha512'**. Checksums for directories are calculated as checksum of all children's checksums plus their filenames (see example below).
- **times** (default **false**) if set to **true** will add `atime`, `mtime` and `ctime` fields (here called `accessTime`, `modifyTime` and `changeTime`) to each tree node.
- **relativePath** if set to **true** every tree node will have relative path anchored to root inspected folder.
- **symlinks** (default **'report'**) if a given path is a symlink by default **inspectTree** will report that symlink (not follow it). You can flip this behaviour by setting this option to **'follow'**.

---

**returns:**

`undefined` if given path doesn't exist. Otherwise tree of inspect objects like:

```
1 {
2   name: 'my_dir',
3   type: 'dir',
4   size: 123, // this is combined size of all items in this directory
5   relativePath: '.',
6   md5: '11c68d9ad988ff4d98768193ab66a646',
7   // checksum of this directory was calculated as:
8   // md5(child[0].name + child[0].md5 + child[1].name + child[1].md5)
9   children: [
10    {
11      name: 'empty',
12      type: 'dir',
13      size: 0,
14      relativePath: './dir',
15      md5: 'd41d8cd98f00b204e9800998ecf8427e',
16      children: []
17    }, {
18      name: 'file.txt',
19      type: 'file',
20      size: 123,
21      relativePath: './file.txt',
22      md5: '900150983cd24fb0d6963f7d28e17f72'
23    }
24  ]
25 }
```

**list([path])**

asynchronous: **listAsync(path)**

Lists the contents of directory. Equivalent of `fs.readdir`.

**arguments:**

`path` (optional) path to directory you would like to list. If not specified defaults to CWD.

**returns:**

Array of file names inside given path, or `undefined` if given path doesn't exist.

**move(from, to, [options])**

asynchronous: **moveAsync(from, to, [options])**

Moves given path to new location.

---

**arguments:**

**from** path to directory or file you want to move.

**to** path where the thing should be moved.

**options** (optional) additional options for customization. Is an **Object** with possible fields:

- **overwrite** (default: **false**) Whether to overwrite destination path when it already exists. If **true**, the overwrite will be performed.

**returns:**

Nothing.

**path(parts...)**

Returns path resolved to internal CWD of this jetpack object.

**arguments:**

**parts** strings to join and resolve as path (as many as you like).

**returns:**

Resolved path as string.

**examples:**

```
1 jetpack.cwd(); // if it returns '/one/two'
2 jetpack.path(); // this will return the same '/one/two'
3 jetpack.path("three"); // this will return '/one/two/three'
4 jetpack.path("../four"); // this will return '/one/four'
```

**read(path, [returnAs])**

asynchronous: **readAsync(path, [returnAs])**

Reads content of file.

**arguments:**

**path** path to file.

**returnAs** (optional) how the content of file should be returned. Is a string with possible values:

- **'utf8'** (default) content will be returned as UTF-8 String.
- **'buffer'** content will be returned as a Buffer.
- **'json'** content will be returned as parsed JSON object.
- **'jsonWithDates'** content will be returned as parsed JSON object, and date strings in ISO format will be automatically turned into Date objects.

---

**returns:**

File content in specified format, or `undefined` if file doesn't exist.

**remove([path])**

asynchronous: **removeAsync([path])**

Deletes given path, no matter what it is (file, directory or non-empty directory). If path already doesn't exist terminates gracefully without throwing, so you can use it as 'ensure path doesn't exist'.

**arguments:**

`path` (optional) path to file or directory you want to remove. If not specified the remove action will be performed on current working directory (CWD).

**returns:**

Nothing.

**examples:**

```
1 // Deletes file
2 jetpack.remove("my_work/notes.txt");
3
4 // Deletes directory "important_stuff" and everything inside
5 jetpack.remove("my_work/important_stuff");
6
7 // Remove can be called with no parameters and will default to CWD then
8 // In this example folder 'my_work' will cease to exist.
9 const myStuffDir = jetpack.cwd("my_stuff");
10 myStuffDir.remove();
```

**rename(path, newName, [options])**

asynchronous: **renameAsync(path, newName, [options])**

Renames given file or directory.

**arguments:**

`path` path to thing you want to change name of.

`newName` new name for this thing (not full path, just a name).

`options` (optional) additional options for customization. Is an `Object` with possible fields:

- `overwrite` (default: **false**) Whether to overwrite destination path when it already exists. If **true**, the overwrite will be performed.

---

**returns:**

Nothing.

**examples:**

```
1 // The file "my_work/important.md" will be renamed to "my_work/very_important.md"
2 jetpack.rename("my_work/important.txt", "very_important.md");
```

**symlink(symlinkValue, path)**

asynchronous: **symlinkAsync(symlinkValue, path)**

Creates symbolic link.

**arguments:**

**symlinkValue** path where symbolic link should point.

**path** path where symbolic link should be put.

**returns:**

Nothing.

**tmpDir([options])**

asynchronous: **tmpDirAsync([options])**

Creates temporary directory with random, unique name.

**arguments:**

**options** (optional) **Object** with possible fields:

- **prefix** prefix to be added to created random directory name. Defaults to none.
- **basePath** the path where temporary directory should be created. Defaults to <https://nodejs.org/api/os.html#os.tmpdir>

**returns:**

New CWD context with temporary directory specified in **path** as CWD (see docs of **cwd()** method for explanation).

**examples:**

```
1 // Creates temporary directory, e.g. /tmp/90ed0f0f4a0ba3b1433c5b51ad8fc76b
2 // You can interact with this directory by returned CWD context.
3 const dirContext = jetpack.tmpDir();
4
```



---

```
5 // Creates temporary directory with a prefix, e.g. /tmp/
   foo_90ed0f0f4a0ba3b1433c5b51ad8fc76b
6 jetpack.tmpDir({ prefix: "foo_" });
7
8 // Creates temporary directory on given path, e.g. /some/other/path/90
   ed0f0f4a0ba3b1433c5b51ad8fc76b
9 jetpack.tmpDir({ basePath: "/some/other/path" });
10
11 // Creates temporary directory on jetpack.cwd() path
12 jetpack.tmpDir({ basePath: "." });
13
14 // The method returns new jetpack context, so you can easily clean your
15 // temp files after you're done.
16 const dir = jetpack.tmpDir();
17 dir.write("foo.txt", data);
18 // ...and when you're done using the dir...
19 dir.remove();
```

## **write(path, data, [options])**

asynchronous: **writeAsync(path, data, [options])**

Writes data to file. If any parent directory in `path` doesn't exist it will be created (like `mkdir -p`).

### **arguments:**

`path` path to file.

`data` data to be written. This could be `String`, `Buffer`, `Object` or `Array` (if last two used, the data will be outputted into file as JSON).

`options` (optional) `Object` with possible fields:

- `mode` file will be created with given mode. Value could be number (eg. `0o700`) or string (eg. `'700'`).
- `atomic` (default **false**) if set to **true** the file will be written using strategy which is much more resistant to data loss. The trick is very simple, read this to get the concept.
- `jsonIndent` (defaults to 2) if writing JSON data this tells how many spaces should one indentation have.

### **returns:**

Nothing.

## **Matching patterns**

API methods `copy` and `find` have `matching` option. Those are all the possible tokens to use there:

- 
- `*` - Matches 0 or more characters in a single path portion.
  - `?` - Matches 1 character.
  - `!` - Used as the first character in pattern will invert the matching logic (match everything what **is not** matched by tokens further in this pattern).
  - `[...]` - Matches a range of characters, similar to a RegExp range. If the first character of the range is `!` or `^` then it matches any character not in the range.
  - `@(pattern|pat*|pat?ern)` - Matches exactly one of the patterns provided.
  - `+(pattern|pat*|pat?ern)` - Matches one or more occurrences of the patterns provided.
  - `?(pattern|pat*|pat?ern)` - Matches zero or one occurrence of the patterns provided.
  - `*(pattern|pat*|pat?ern)` - Matches zero or more occurrences of the patterns provided.
  - `!(pattern|pat*|pat?ern)` - Matches anything that does not match any of the patterns provided.
  - `**` - If a “globstar” is alone in a path portion, then it matches zero or more directories and sub-directories.

*(explanation borrowed from glob which is using the same matching library as this project)*