readme.md for @tempfix/watcher

The file system watcher that strives for perfection, with no native dependencies and optional rename detection support.

Features

- **Reliable**: This library aims to handle all issues that may possibly arise when dealing with the file system, including some the most popular alternatives don't handle, like EMFILE errors.
- **Rename detection**: This library can optionally detect when files and directories are renamed, which allows you to provide a better experience to your users in some cases.
- **Performant**: Native recursive watching is used when available (macOS and Windows), and it's efficiently manually performed otherwise.
- **No native dependencies**: Native dependencies can be painful to work with, this library uses 0 of them.
- **No bloat**: Many alternative watchers ship with potentially useless and expensive features, like support for globbing, this library aims to be much leaner while still exposing the right abstractions that allow you to use globbing if you want to.
- **TypeScript-ready**: This library is written in TypeScript, so types aren't an afterthought but come with the library.

Comparison

You are probably currently using one of the following alternatives for file system watching, here's how they compare against Watcher:

- `fs.watch`: Node's built-in `fs.watch` function is essentially garbage and you never want to use it directly.
  - Cons:
    - Recursive watching is not supported under Linux, so if you need to support Linux at all you are out of luck already.

    - Even if you only need to support macOS or Windows, where native recursive watching is provided, the events provided by `fs.watch` are completely useless as they tell you nothing about what actually happened in the file system, so you'll have to poll the file system on your own anyway.
    - There are many things that `fs.watch` doesn't take care of, for example watching non-existent paths is just not supported and EMFILE errors are not handled.
- `chokidar`: this is the most popular file system watcher available, while it may be good enough in some cases it's not perfect.
    - Cons:
        - It requires a native dependency for efficient recursive watching under macOS, and native dependencies can be a pain to work with.
        - It doesn't watch recursively efficiently under Windows, Watcher on the other hand is built upon Node's native recursive watching capabilities for Windows.
        - It can't detect renames.
        - If you don't need features like globbing then chokidar will bloat your app bundles unnecessarely.
        - EMFILE errors are not handled properly, so if you are watching enough files chokidar will eventually just give up on them.
        - It's not very actively maintened, Watcher on the other hand strives for having 0 bugs, if you can find some we'll fix them ASAP.
    - Pros:
        - It supports handling symlinks.
        - It has some built-in support for handling temporary files written to disk while perfoming an atomic write, although ignoring them in Watcher is pretty trivial too, you can ignore them via the `ignore` option.
        - It can more reliably watch network attached paths, although that will lead to performance issues when watching ~lots of files.
        - It's more battle tested, although Watcher has a more comprehensive test suite and is used in production too (for example in Notable, which was using `chokidar` before).
- `node-watch`: in some ways this library is similar to Watcher, but much less mature.
    - Cons:
        - No initial events can be emitted when starting watching.
        - Only the "update" or "remove" events are emitted, which tell you nothing about whether each event refers to a file or a directory, or whether a file got added or modified.
        - "add" and "unlink" events are not provided in some cases, like for files inside an added/deleted folder.
        - Watching non-existent paths is not supported.
        - It can't detect renames.
- `nsfw`: this is a lesser known but pretty good watcher, although it comes with some major drawbacks.
    - Cons:

    - It's based on native dependencies, which can be a pain to work with, especially considering that prebuild binaries are not provided so you have to build them yourself.
    - It's not very customizable, so for example instructing the watcher to ignore some paths is not possible.
    - Everything being native makes it more difficult to contribute a PR or a test to it.
    - It's not very actively maintained.
  - Pros:
    - It adds next to 0 overhead to the rest of your app, as the watching is performed in a separate process and events are emitted in batches.
- "`perfection`": if there was a "perfect" file system watcher, it would compare like this against Watcher (i.e. this is pretty much what's currently missing in Watcher):
  - Pros:
    - It would support symlinks, Watcher doesn't handle them just yet.
    - It would watch all parent directories of the watched roots, for unlink detection when those parents get unlinked, Watcher currently also watches only up-to 1 level parents, which is more than what most other watchers do though.
    - It would provide some simple and efficient APIs for adding and removing paths to watch from/to a watcher instance, Watcher currently only has some internal APIs that could be used for that but they are not production-ready yet, although closing a watcher and making a new one with the updated paths to watch works well enough in most cases.
    - It would add next to 0 overhead to the rest of your app, currenly Watcher adds some overhead to your app, but if that's significant for your use cases we would consider that to be a bug. You could potentially already spawn a separate process and do the file system watching there yourself too.
    - Potentially there are some more edge cases that should be handled too, if you know about them or can find any bug in Watcher just open an issue and we'll fix it ASAP.

Install

```
npm install --save watcher
```

Options

The following options are provided, you can use them to customize watching to your needs:

- `debounce`: amount of milliseconds to debounce event emission for.
  - by default this is set to `300`.

    - the higher this is the more duplicate events will be ignored automatically.
- `depth`: maximum depth to watch files at.
    - by default this is set to `20`.
    - this is useful for avoiding watching directories that are absurdly deep, that would probably waste resources.
- `limit`: maximum number of paths to prod.
    - by default this is set to `10_000_000`.
    - this is useful as a safe guard in cases where for example the user decided to watch `/`, perhaps by mistake.
- `ignore`: optional function (or regex) that if returns `true` for a path it will cause that path and all its descendants to not be watched at all.
    - by default this is not set, so all paths are watched.
    - setting an `ignore` function can be very important for performance, you should probably ignore folders like `.git` and temporary files like those used when writing atomically to disk.
    - if you need globbing you'll just have to match the path passed to `ignore` against a glob with a globbing library of your choosing.
- `ignoreInitial`: whether events for the initial scan should be ignored or not.
    - by default this is set to `false`, so initial events are emitted.
- `native`: whether to use the native recursive watcher if available and needed.
    - by default this is set to `true`.
    - the native recursive watcher is only available under macOS and Windows.
    - when the native recursive watcher is used the `depth` option is ignored.
    - setting it to `false` can have a positive performance impact if you want to watch recursively a potentially very deep directory with a low `depth` value.
- `persistent`: whether to keep the Node process running as long as the watcher is not closed.
    - by default this is set to `false`.
- `pollingInterval`: polling is used as a last resort measure when watching non-existent paths inside non-existent directories, this controls how often polling is performed, in milliseconds.
    - by default this is set to `3000`.
    - you can set it to a lower value to make the app detect events much more quickly, but don't set it too low if you are watching many paths that require polling as polling is expensive.
- `pollingTimeout`: sometimes polling will fail, for example if there are too many file descriptors currently open, usually eventually polling will succeed after a few tries though, this controls the amount of milliseconds the library should keep retrying for.
    - by default this is set to `20000`.
- `recursive`: whether to watch recursively or not.
    - by default this is set to `false`.
    - this is supported under all OS'.
    - this is implemented natively by Node itself under macOS and Windows.
- `renameDetection`: whether the library should attempt to detect renames and emit `rename`/`renameDir` events.

    - by default this is set to `false`.
    - rename detection may cause a delayed event emission, because the library may have to wait some more time for it.
    - if disabled, the raw underlying `add`/`addDir` and `unlink`/`unlinkDir` events will be emitted instead after a rename.
    - if enabled, the library will check if each pair of `add`/`unlink` or `addDir`/`unlinkDir` events are actually `rename` or `renameDir` events respectively, so it will wait for both of those events to be emitted.
    - rename detection is fairly reliable, but it is fundamentally dependent on how long the file system takes to emit the underlying raw events, if it takes longer than the set rename timeout the app won't detect the rename and will instead emit the underlying raw events.
- `renameTimeout`: amount of milliseconds to wait for a potential `rename`/`renameDir` event to be detected.
    - by default this is set to `1250`.
    - the higher this value is the more reliably renames will be detected, but don't set this too high, or the emission of some events could be delayed by that amount.
    - the higher this value is the longer the library will take to emit `add`/`addDir`/`unlink`/`unlinkDir` events.

Usage

Watcher returns an `EventEmitter` instance, so all the methods inherited from that are supported, and the API is largely event-driven.

The following events are emitted:

- Watcher events:
    - `error`: Emitted whenever an error occurs.
    - `ready`: Emitted after the Watcher has finished instantiating itself. No events are emitted before this events, expect potentially for the `error` event.
    - `close`: Emitted when the watcher gets explicitly closed and all its watching operations are stopped. No further events will be emitted after this event.
    - `all`: Emitted right before a file system event is about to get emitted.
- File system events:
    - `add`: Emitted when a new file is added.
    - `addDir`: Emitted when a new directory is added.
    - `change`: Emitted when an existing file gets changed, maybe its content changed, maybe its metadata changed.
    - `rename`: Emitted when a file gets renamed. This is only emitted when `renameDetection` is enabled.
    - `renameDir`: Emitted when a directory gets renamed. This is only emitted when `renameDetection` is enabled.
    - `unlink`: Emitted when a file gets removed from the watched tree.

- `unlinkDir`: Emitted when a directory gets removed from the watched tree.

Basically it you have used `chokidar` in the past Watcher emits pretty much the same exact events, except that it can also emit `rename`/`renameDir` events, it doesn't provide `stats` objects but only paths, and in general it exposes a similar API surface, so switching from (or to) `chokidar` should be easy.

The following interface is provided:

```
type Roots = string[] | string;

type TargetEvent = 'add' | 'addDir' | 'change' | 'rename' | 'renameDir' | 'unlink' |
'unlinkDir';
type WatcherEvent = 'all' | 'close' | 'error' | 'ready';
type Event = TargetEvent | WatcherEvent;

type Options = {
  debounce?: number,
  depth?: number,
  ignore?: (( targetPath: Path ) => boolean) | RegExp,
  ignoreInitial?: boolean,
  native?: boolean,
  persistent?: boolean,
  pollingInterval?: number,
  pollingTimeout?: number,
  recursive?: boolean,
  renameDetection?: boolean,
  renameTimeout?: number
};

class Watcher {
  constructor ( roots: Roots, options?: Options, handler?: Handler ): this;
  on ( event: Event, handler: Function ): this;
  close (): void;
}
```

You would use the library like this:

```
import Watcher from 'watcher';

// Watching a single path
```

```
const watcher = new Watcher ( '/foo/bar' );

// Watching multiple paths
const watcher = new Watcher ( ['/foo/bar', '/baz/qux'] );

// Passing some options
const watcher = new Watcher ( '/foo/bar', { renameDetection: true } );

// Passing an "all" handler directly
const watcher = new Watcher ( '/foo/bar', {}, ( event, targetPath, targetPathNext ) => {} );

// Attaching the "all" handler manually
const watcher = new Watcher ( '/foo/bar' );
watcher.on ( 'all', ( event, targetPath, targetPathNext ) => { // This is what the library
does internally when you pass it a handler directly
  console.log ( event ); // => could be any target event: 'add', 'addDir', 'change', 'rename',
'renameDir', 'unlink' or 'unlinkDir'
  console.log ( targetPath ); // => the file system path where the event took place, this is
always provided
  console.log ( targetPathNext ); // => the file system path "targetPath" got renamed to, this
is only provided on 'rename'/'renameDir' events
});

// Listening to individual events manually
const watcher = new Watcher ( '/foo/bar' );

watcher.on ( 'error', error => {
  console.log ( error instanceof Error ); // => true, "Error" instances are always provided on
"error"
});
watcher.on ( 'ready', () => {
  // The app just finished instantiation and may soon emit some events
});
watcher.on ( 'close', () => {
  // The app just stopped watching and will not emit any further events
});
watcher.on ( 'all', ( event, targetPath, targetPathNext ) => {
  console.log ( event ); // => could be any target event: 'add', 'addDir', 'change', 'rename',
'renameDir', 'unlink' or 'unlinkDir'
```

```
  console.log ( targetPath ); // => the file system path where the event took place, this is always provided
  console.log ( targetPathNext ); // => the file system path "targetPath" got renamed to, this is only provided on 'rename'/'renameDir' events
});
watcher.on ( 'add', filePath => {
  console.log ( filePath ); // "filePath" just got created, or discovered by the watcher if this is an initial event
});
watcher.on ( 'addDir', directoryPath => {
  console.log ( directoryPath ); // "directoryPath" just got created, or discovered by the watcher if this is an initial event
});
watcher.on ( 'change', filePath => {
  console.log ( filePath ); // "filePath" just got modified
});
watcher.on ( 'rename', ( filePath, filePathNext ) => {
  console.log ( filePath, filePathNext ); // "filePath" got renamed to "filePathNext"
});
watcher.on ( 'renameDir', ( directoryPath, directoryPathNext ) => {
  console.log ( directoryPath, directoryPathNext ); // "directoryPath" got renamed to "directoryPathNext"
});
watcher.on ( 'unlink', filePath => {
  console.log ( filePath ); // "filePath" got deleted, or at least moved outside the watched tree
});
watcher.on ( 'unlinkDir', directoryPath => {
  console.log ( directoryPath ); // "directoryPath" got deleted, or at least moved outside the watched tree
});

// Closing the watcher once you are done with it
watcher.close ();

// Updating watched roots by closing a watcher and opening an updated one
watcher.close ();
watcher = new Watcher ( /* Updated options... */ );
```

Thanks

- `chokidar`: for providing me a largely good-enough file system watcher for a long time.
- `node-watch`: for providing a good base from with to make Watcher, and providing some good ideas for how to write good tests for it.

License

Revision #3
Created 2026-03-28 11:15:05 UTC by foss.global Team
Updated 2026-03-28 12:21:49 UTC by foss.global Team