FSBroker

FSBroker is a Go library that acts as an intelligent translation layer on top of fsnotify, converting raw file system events into reliable, high-level actions like `Create`, `Write`, `Rename`, and `Remove`.

Operating systems emit file system events inconsistently and often obscurely. For example:

- A simple file rename might appear as a `CREATE` followed by a `RENAME` on one OS, but `REMOVE` then `CREATE` on another.
- Moving a file to the trash might look like a `RENAME` event on some systems.
- Rapid saves can flood your application with `WRITE` events.
- System-specific files (`.DS_Store`, `thumbs.db`) generate unwanted noise.

`fsnotify` provides the raw events, but leaves the complex interpretation up to you. FSBroker handles this complexity by:

- **Batching:** Grouping events that occur close together in time.
- **Pattern Recognition:** Analyzing sequences of raw events (like `CREATE`+`RENAME`) within a time window.
- **Deduplication:** Filtering out redundant events (e.g., multiple `WRITE`s).
- **Contextual Analysis:** Considering platform specifics (Windows vs. macOS vs. Linux) and file metadata.

This allows FSBroker to reliably report the *actual* user actions happening on the filesystem.

Features

- **Accurate Event Interpretation:** Translates complex, platform-specific `fsnotify` event sequences into clear `Create`, `Write`, `Rename`, and `Remove` actions.
- **Intelligent Rename/Move Detection:** Correctly identifies file/directory renames and moves, even across different watched directories, providing both old and new paths.
- **Reliable Deletion Detection:** Differentiates between hard deletes (permanent removal) and soft deletes (moves to trash or to unwatched directories), emitting the appropriate `Remove` event.
- **Write Event Deduplication:** Consolidates bursts of `WRITE` events (e.g., rapid saves) into a single `Write` event.
- **macOS File Clear Handling:** Detects file clearing operations on macOS (which don't always emit `WRITE`) by analyzing `CHMOD` events and file size.
- **Noise Reduction:** Automatically ignores irrelevant events (like `WRITE` on directories in Windows) and common system files (e.g., `.DS_Store`, `thumbs.db`). Option to ignore all hidden files.
- **Automatic Recursive Watching:** Automatically adds newly created subdirectories within watched paths to the watcher.
- **Custom Event Filtering:** Allows pre-filtering of raw `fsnotify` events based on path or type *before* processing.
- **Optional Chmod Events:** Provides an option (`EmitChmod`) to receive raw `CHMOD` events if needed.

Supported Platforms

FSBroker is designed and tested to work on the following operating systems:

- **Windows**
- **macOS**
- **Linux**

While manually tested across these platforms, contributions towards better multi-platform testing are welcome (see Missing Features).

Changelog

- (New v1.0.0) First complete release.
- (New v0.1.8) Update fsnotify to v1.9.0
- (New v0.1.7) Update fsnotify to v1.9.0
- (New v0.1.6) Added the option to ignore hidden files.
- (New v0.1.5) Fix more bugs, added the option to emit chmod events (defaults to false).
- (New v0.1.4) Fix a bug where multiple consecutive file creations would not be detected on Windows.
- (New v0.1.4) Introduce the ability to interpret file moves/renames as deletes in cases where its appropriate.
- (New v0.1.3) Fix a problem where fsnotify would not detect file changes on macOS if the file was cleared.

Installation

To install FS Broker, use `go get`:

```
go get github.com/helshabini/fsbroker
```

Usage

Here is an example of how to use FS Broker:

```
package main

import (
    "log"
    "time"

    "github.com/helshabini/fsbroker"
)

func main() {
    config := fsbroker.DefaultFSConfig()
    broker, err := fsbroker.NewFSBroker(config)
    if err != nil {
        log.Fatalf("error creating FS Broker: %v", err)
    }
    defer broker.Stop()
```

```
    if err := broker.AddRecursiveWatch("watch"); err != nil {
        log.Printf("error adding watch: %v", err)
    }

    broker.Start()

    for {
        select {
        case event := <-broker.Next():
            log.Printf("fs event has occurred: type=%s, path=%s,
timestamp=%s, properties=%v",
                event.Type.String(), event.Path,
event.Timestamp.Format(time.RFC3339), event.Properties)
        case err := <-broker.Error():
            log.Printf("an error has occurred: %v", err)
        }
    }
}
```

You can also apply your own filters to events:

```
broker.Filter = func(action *FSAction) bool {
    return action.Type != fsbroker.Remove // Filters out any event that is
not Remove
}
```

or

```
broker.Filter = func(action *FSAction) bool {
    return action.Path == "/some/excluded/path" // Filters out any event
which is related to this path
}
```

API

`DefaultFSConfig() *FSConfig`

Creates a default FS Config instance with these default settings:

- Timeout: 300 * time.Millisecond
- IgnoreSysFiles: true
- IgnoreHiddenFiles: true
- EmitChmod: false

`NewFSBroker(config *FSConfig) (*FSBroker, error)`

Creates a new FS Broker instance.

- `config`: FS Config instance.

`(*FSBroker) AddRecursiveWatch(path string) error`

Adds a recursive watch on the specified path. This will:

- Watch the specified directory and all its subdirectories
- Automatically add watches for newly created subdirectories
- Add all existing files and directories to the watch map
- `path`: The directory path to watch.

`(*FSBroker) AddWatch(path string) error`

Adds a watch on a specific file or directory. This will:

- Watch the specified path
- Add all existing files in the directory to the watch map (if path is a directory)
- Not automatically watch new subdirectories (use AddRecursiveWatch for that)
- `path`: The file or directory path to watch.

`(*FSBroker) RemoveWatch(path string)`

Removes a watch on a file or directory. This will:

- Remove the watch from fsnotify
- Remove the path and all its subpaths from the watch map
- `path`: The file or directory path to stop watching.

`(*FSBroker) Start()`

Starts the FS Broker to begin monitoring file system events. This will:

- Start the event processing loop
- Begin listening for fsnotify events
- Process and emit events through the Next() channel

`(*FSBroker) Stop()`

Stops the FS Broker. This will:

- Stop the event processing loop
- Close the fsnotify watcher
- Clean up all resources

`(*FSBroker) Next() <-chan *FSAction`

Returns a channel that receives processed file system events. Each event is an `FSAction` that represents a high-level file system operation (Create, Write, Rename, or Remove).

`(*FSBroker) Error() <-chan error`

Returns a channel that receives errors that occur during file system watching or event processing.

`(*FSBroker) Filter func(*FSAction) bool`

A function that can be set to filter events before they are emitted through the Next() channel. Return `true` to allow the event to be emitted, `false` to filter it out.

Debugging

FSBroker uses the standard `log/slog` package for logging. By default, only messages with `INFO` level or higher are shown.

To enable detailed debug logging (using `slog.LevelDebug`), compile or test the package with the `fsbroker_debug` build tag. This enables verbose logging of internal event processing steps, which can be helpful for troubleshooting.

Examples:

```
# Run tests with debug logging enabled
go test -v -tags fsbroker_debug ./...

# Build your application with debug logging enabled
go build -tags fsbroker_debug -o myapp main.go
```

When the tag is *not* provided, the `slog` level defaults to `INFO`, and the `slog.Debug` calls in the code have minimal performance impact.

Missing features:

Here is a list of features I would like to add in the future, please feel free to submit pull requests:

- Testing on different operating systems

Currently, existing automated tests may give a false negative due to concurrency issues with the way tests are performed. Any contribution to the testing methodology is welcomed.

FAQ

- How does fsbroker work and what does it offer as opposed to fsnotify?

  fsbroker depends on fsnotify as an underlying file system event watcher. However, fsnotify expects you to make sense of the events it emits according to your platform and your use case. Unfortunately, because of how messy file system events are in various operating systems, it is diffucilt to make sense of them.

It works by accumliating events emitted from fsnotify in an event queue, then process them in batches to try and make sense of what actually happened on the file system.

- Does fsbroker have any overhead on top of fsnotify?

  Yes. To be able to accurately make sense of what happened on the file system, fsbroker needs to keep an in-memory map of your watched directories. We've tried to keep it as small as possible, but it still grows as large as your number of files being watched.

  In addition, fsbroker may perform some additional file checks and syscalls to verify file attributes or existance on disk.

  In my view, these are checks/overheads you would have had to deal with in your code anyway. They are also fairly minimal. So I consider it a very good trade-off.

- How accurate is fsbroker?

  It is not a 100%. But good enough for most use cases, and certainly far better than what fsnotify offers. The accuracy is directly propotional to the timeout interval you set in fsbroker configuration. The more you wait for batching events together, the better your accuracy will be.

- Why is the accuracy of event detection not a 100%?

  There are two main reasons

  - fsbroker depends on accumilating a bunch of fsnotify events togther to try and make sense of what actually happened. Because of the nature of this process, there has to be a cutoff interval. Some user actions may force fsnotify to emit multiple events in succession providing a certain pattern, which fsbroker uses to make its decision. If the action the user took results in a pattern that is divided between two queue frames, the result may not be accurate.
  - Many software products handle files in very convulted ways, such as writing to shadow files, burst writing, and shadow deletion. We have no control over how other software decides to handle the file system. However, we tried to pick the most general and common patterns to detect and emit.

- What patterns does fsbroker detect and what events does it emit?

  Here are the main general behaviours which fsbroker applies to all platforms.

  - fsnotify REMOVE events for a directory or file invalidates all previous events for that path. So, fsbroker will ignore all events preceeding a REMOVE event for a given path within the event queue interval window.
  - fsnotify WRITE events on a file are all ignored except the last one within the event queue interval window. This offers WRITE event deduplication. Saving you from when users burst save the file multiple times.
  - fsnotify WRITE events on directories are always ignored. They occur only on Windows when any watched directory (except the root watch directory) contents are modified (e.g. file created or file removed). These events are completely ignored by fsbroker, because the file created/removed event is already captured.

- fsnotify CHMOD events will be forwarded as is if the user elects to set `EmitChmod` to `true` in broker configuration. They will not be deduplicated.
- fsbroker will enrich events with additional event information.

Here are also common fsnotify patterns that fsbroker detects and handles as follows:

User Action	Windows Pattern	macOS Pattern	Linux Pattern	Event Emitted	Notes
Create empty file within watched directory	- CREATE (file path)	- CREATE (file path)	- CREATE (file path)	Create	
Create non-empty file within watched directory	- CREATE (file path) - WRITE (file path)	- CREATE (file path) - WRITE (file path)	- CREATE (file path) - WRITE (file path) - WRITE (file path)	Create	
Clear file within a watched directory	- WRITE (file path)	-	- WRITE (file path)	Write	fsnotify doesn't emit a WRITE event when a file is cleared on macOS, instead we use the CHMOD event and check the file's existance and size to know it is a WRITE event
Modify file within a watched directory	- WRITE (file path)	- WRITE (file path)	- WRITE (file path)	Write	
Rename file inplace within a watched directory	- RENAME (old file path) - CREATE (new file path)	- CREATE (new file path) - RENAME (old file path)	- RENAME (old file path) - CREATE (new file path)	Rename	

User Action	Windows Pattern	macOS Pattern	Linux Pattern	Event Emitted	Notes
Move file from a watched directory to another watched directory	- REMOVE (old file path) - CREATE (new file path)	- CREATE (new file path) - RENAME (old file path)	- RENAME (old file path) - CREATE (new file path)	Rename	
Move file from an unwatched directory to a watched directory	-	-	-	Create	Similar to creating a non-empty file
Soft delete file (move to trash) or Move file from a watched directory to an unwatched directory	- REMOVE (file path)	- RENAME (file path)	- RENAME (file path)	Remove	
Hard delete file (permenantly)	- REMOVE (file path)	- REMOVE (file path)	- REMOVE (file path)	Remove	
Create a directory	- CREATE (dir path)	- CREATE (dir path)	- CREATE (dir path)	Create	
Rename directory inplace within a watched directory	- RENAME (old dir path) - CREATE (new dir path)	- CREATE (new dir path) - RENAME (old dir path)	- RENAME (old dir path) - CREATE (new dir path) - RENAME (old dir path)	Rename	

User Action	Windows Pattern	macOS Pattern	Linux Pattern	Event Emitted	Notes
Move directory from a watched directory to another watched directory	- REMOVE (old dir path) - CREATE (new dir path)	- CREATE (new dir path) - RENAME (old dir path)	- RENAME (old dir path) - CREATE (new dir path)	Rename	Note that operating systems do not raise events for any files already existing within the moved directory
Move directory from an unwatched directory to a watched directory	-	-	-	Create	Similar to creating a directory. Note that operating systems do not raise events for any files already existing within the moved directory
Soft delete directory (move to trash) or Move directory from a watched directory to an unwatched directory	- REMOVE (dir path)	- RENAME (dir path)	- RENAME (dir path)	Remove	Note that no additional remove events are emitted for each directory or file within the deleted directory except on windows, additional remove events are emitted once the trash is emptied
Hard delete directory (permenantly)	- REMOVE (dir path)	- REMOVE (file path)	- REMOVE (file path)	Remove	Note that additional remove events are emitted for each directory or file within the deleted directory

Here is a decision tree to help visualize this nonsense:

```
Event received from fsnotify
│
├── Windows
│   │
│   ├── Is it a CREATE event?
│   │   ├── Is there a RENAME event for the same Id?
│   │   │   ├── Yes → RENAME event (internal rename/move)
│   │   │   └── No → CREATE event
│   │   │
```

```
   │    │    └── For a file: Is it followed by WRITE event(s)?
   │    │        └── Yes → Still CREATE event (non-empty file creation)
   │    │
   │    ├── Is it a WRITE event?
   │    │    ├── Is it for a directory?
   │    │    │    └── Yes → Ignore event (Windows directory write noise)
   │    │    │
   │    │    ├── Was there a CREATE event for this path just before?
   │    │    │    └── Yes → Part of CREATE event (non-empty file)
   │    │    │
   │    │    └── Is it the most recent WRITE for this path?
   │    │        ├── Yes → WRITE event
   │    │        └── No → Ignore (deduplicate)
   │    │
   │    ├── Is it a REMOVE event?
   │    │    └── Is there a CREATE event with the same Id?
   │    │        ├── Yes → RENAME event (internal rename/move)
   │    │        └── No → REMOVE event (hard delete) / Invalidate all previous
   events for this path
   │    │
   │    ├── Is it a RENAME event?
   │    │    └── Yes → REMOVE event (Windows treats renames as removes)
   │    │
   │    └── Is it a CHMOD event?
   │        └── Yes → Emit if EmitChmod is true
   │
   ├── macOS
   │    ├── Is it a CREATE event?
   │    │    ├── Is there a RENAME event for the same Id?
   │    │    │    ├── Yes → RENAME event (internal rename/move)
   │    │    │    └── No → CREATE event
   │    │    │
   │    │    └── For a file: Is it followed by WRITE event(s)?
   │    │        └── Yes → Still CREATE event (non-empty file creation)
   │    │
   │    ├── Is it a WRITE event?
   │    │    ├── Was there a CREATE event for this path just before?
   │    │    │    └── Yes → Part of CREATE event (non-empty file)
   │    │    │
   │    │    └── Is it the most recent WRITE for this path?
   │    │        ├── Yes → WRITE event
   │    │        └── No → Ignore (deduplicate)
   │    │
   │    ├── Is it a REMOVE event?
   │    │    ├── Yes → REMOVE event (hard delete)
   │    │    │
   │    │    └── Invalidate all previous events for this path
   │    │
   │    ├── Is it a RENAME event?
   │    │    └── Yes → REMOVE event (soft delete/move to trash)
   │    │
   │    └── Is it a CHMOD event?
   │        └── Yes
   │            ├── Is it a file that became zero bytes?
```

```
│           │   └── Yes → WRITE event (macOS file clear)
│           │
│           └── No → Emit if EmitChmod is true
│
└── Linux
    ├── Is it a CREATE event?
    │   ├── Is there a RENAME event for the same Id?
    │   │   ├── Yes → RENAME event (internal rename/move)
    │   │   └── No → CREATE event
    │   │
    │   └── For a file: Is it followed by WRITE event(s)?
    │       └── Yes → Still CREATE event (non-empty file creation)
    │
    ├── Is it a WRITE event?
    │   ├── Was there a CREATE event for this path just before?
    │   │   └── Yes → Part of CREATE event (non-empty file)
    │   │
    │   └── Is it the most recent WRITE for this path?
    │       ├── Yes → WRITE event
    │       └── No → Ignore (deduplicate)
    │
    ├── Is it a REMOVE event?
    │   ├── Yes → REMOVE event (hard delete)
    │   │
    │   └── Invalidate all previous events for this path
    │
    ├── Is it a RENAME event?
    │   └── Yes → REMOVE event (soft delete/move)
    │
    └── Is it a CHMOD event?
        └── Yes → Emit if EmitChmod is true
```

License

This project is licensed under the BSD-3-Clause License.