Go-based File Monitoring

Jump Cloud #2

Benjamin Fuller

Jeremy Alvis

Nate Caroe

Nick Masching

2014/06/17

Introduction

Client Description

Jump Cloud provides API and Software designed to manage and configure servers. This allows for centralized server management. In turn, server management is simplified and users can focus on solving server problems rather than infrastructure. The process is secure and allows management and tracking of users on the system, ensuring the accounts are being used as prescribed. Scripts or Commands can be scheduled and run on any or all the machines through the Jump Cloud. The process is fully audited, meaning any job can be reviewed, as well as who has administrator access, what the physical capacities of servers are, server load, and various other duties.

Product Vision

JumpCloud implements their own agent for managing servers. This agent can notify the user when different events occur. This project will implement a file notification system that can be set to watch specific files or directories to report any changes to the user. By watching a log file, for example, the user can be notified of any errors as soon as they are written to the file. This also provides a layer of security to the server by catching changes early and allowing the user to take appropriate action.

Requirements

Functional Requirements

The file monitoring system must allow the user to specify files/directories to watch, and flags to look for.

- User can specify any number of files or directories to watch and the specific flags to look for.
- This system must track the files and upon changes, reports these changes to the system
- Creates a way of showing the user what was changed inside the file.
- Sends the result to the agent to be dealt with accordingly.

There are is also a few implementation specifications that must be met in order to work with the client API:

- It must implement the interface util.Service:
    - Upon Stop it must clear the lists of EventListeners.

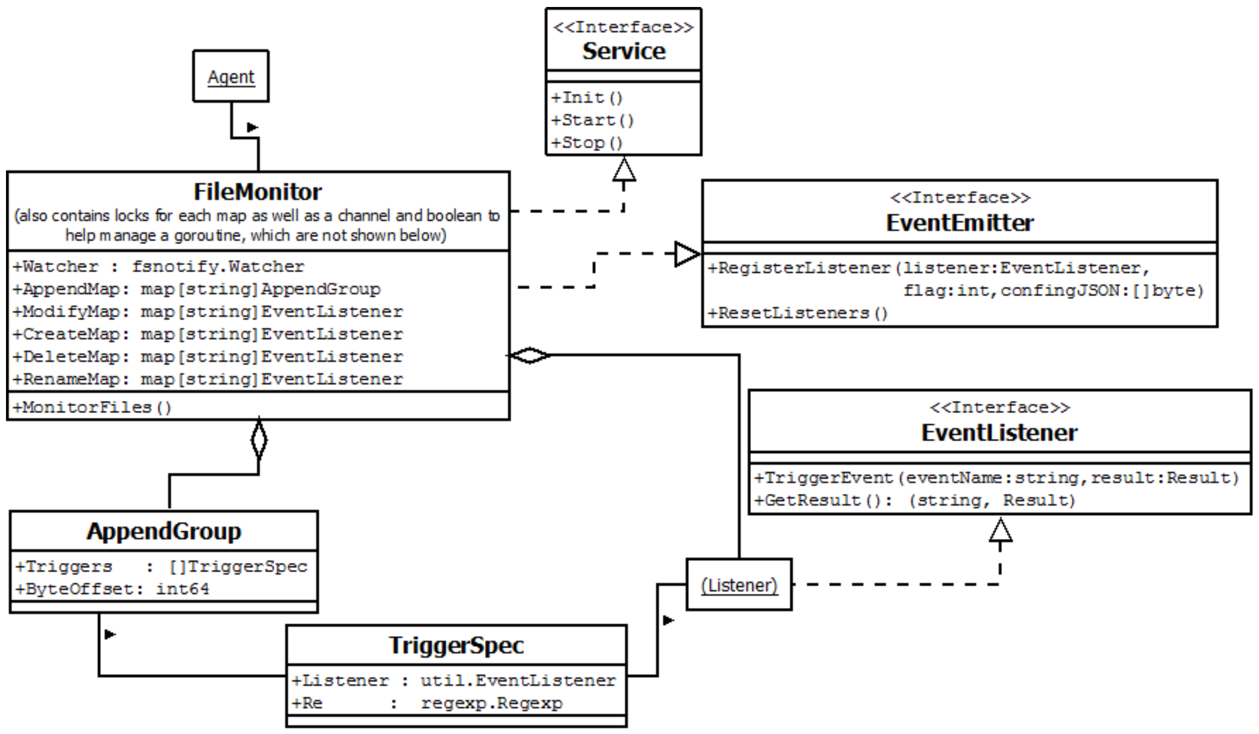
- Upon Start it must initialize the lists of EventListeners.
- Init will only be called one time and can be used to initialize data structures.

- It must implement the interface util.EventEmitter.
- When a relevant event occurs on the system it must call the TriggerEvent on all registered EventListeners.
- The Result object used in the call to TriggerEvent must contain:
    - The name of the file.
    - The complete text that matched the regular expression if it is an append result.
    - A map of the named capture groups and their matched values if it is an append result.

Non-Functional Requirements

- Code must be written in Go.
- System needs to integrate with existing JumpCloud API.
- System must be lightweight (passive monitoring).
- System must work on all operating systems.

System Architecture


Agent
<<Interface>>
Service
+Init()
+Start()
+Stop()
FileMonitor
(also contains locks for each map as well as a channel and boolean to help manage a goroutine, which are not shown below)
+Watcher : fsnotify.Watcher
+AppendMap: map[string]AppendGroup
+ModifyMap: map[string]EventListener
+CreateMap: map[string]EventListener
+DeleteMap: map[string]EventListener
+RenameMap: map[string]EventListener
+MonitorFiles()
<<Interface>>
EventEmitter
+RegisterListener(listener:EventListener, flag:int,confingJSON:[]byte)
+ResetListeners()
<<Interface>>
EventListener
+TriggerEvent(eventName:string,result:Result)
+GetResult(): (string, Result)
AppendGroup
+Triggers : []TriggerSpec
+ByteOffset: int64
(Listener)
TriggerSpec
+Listener : util.EventListener
+Re : regexp.Regexp

File Monitor is both a Service, and an event emitter. The service is initialized and started by the system(Jump Cloud Agent). Users of FileMonitor register listeners using the Event Emitter function, and File Monitor sends events as needed using the listener’s trigger event interface. Registering listeners and emitting events is done synchronously using goroutines(Simple threads).

Technical Design

We had trouble with the fsnotify tool, which has numerous bugs. This meant that it didn’t behave as expected or as advertised on certain operating systems and configurations. Fsnotify has a setting that allows its users to track only certain types of events from a file, but this is not currently supported by Windows. As a result, we couldn’t simply watch a folder for files being created in it, because on Windows we would still receive all the events that happen inside that folder, whether or not they were file creation events.

Rename events were tricky because of the way in which fsnotify sends events differs across operating systems. Additionally, rename events are sent as two separate events: an event that contains the old filename, and an event with the new name, but both events are only sent in the case where the folder the file is in is being watched. In the case where only the file is being watched, fsnotify only sends one event on a rename, which is the event with the old file name. This can lead to losing track of where the file is. Fsnotify also does not support watching a file or folder that does not exist. The only way to watch a file or folder that does not exist is to watch the folder it is in and wait for a create event from that folder that contains the specified filename. Because of fsnotify handling (or lack thereof) of watching for rename events, create events, and watching for files that don’t exists, we ended up watching the folder that contains the file we were interesting, instead of watching the file itself, and then filtering out events that we weren’t watching for. This allows complete cross-platform support while still maintaining reasonable efficiency.

Because our project is a plugin to a larger whole, the interface that is used by this plugin needs to comply with many other plugins to the jump cloud agent. This means rather than having a variety of methods from every plugin added to the interface, there is only one function to register listeners. This is done by using a combination of a enumeration set by the plugin, as well as a json object. This allows the number of arguments and type of said arguments to fluxuate given the requirements given for a certain event type. This same type of pattern is followed when returning results to the listener. Rather than creating some type of object that contains all the fields any of the plugins could need, a json object and an enumeration are given to the listener, which will then interpret the result based on the enumeration passed into it. This simplifies the interface between components and allows one simple shared interface for all plugins to the Jump Cloud Agent.

Figure 1 outlines a general flow of how file monitoring works along with fsnotify. The system is started by the agent and begins to register listeners. A watch is added to the folder and fsnotify uses a go routine to listen to the OS in regards to the file. When an event to the file occurs, an event is sent out to an event channel. The file monitoring system is also running a go routine which waits for the event channel to be populated. When an event is received, we format the event into a JSON object and call the listener's TriggerEvent method. This method sends our result back to the agent.

Figure 1. General Flowchart

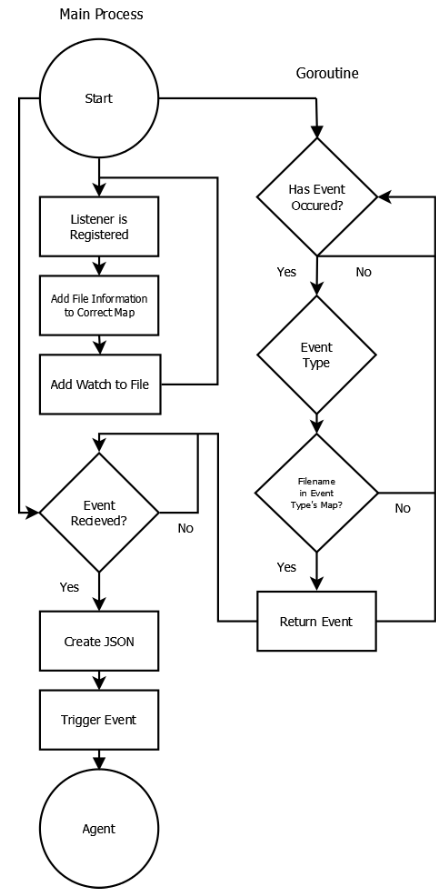
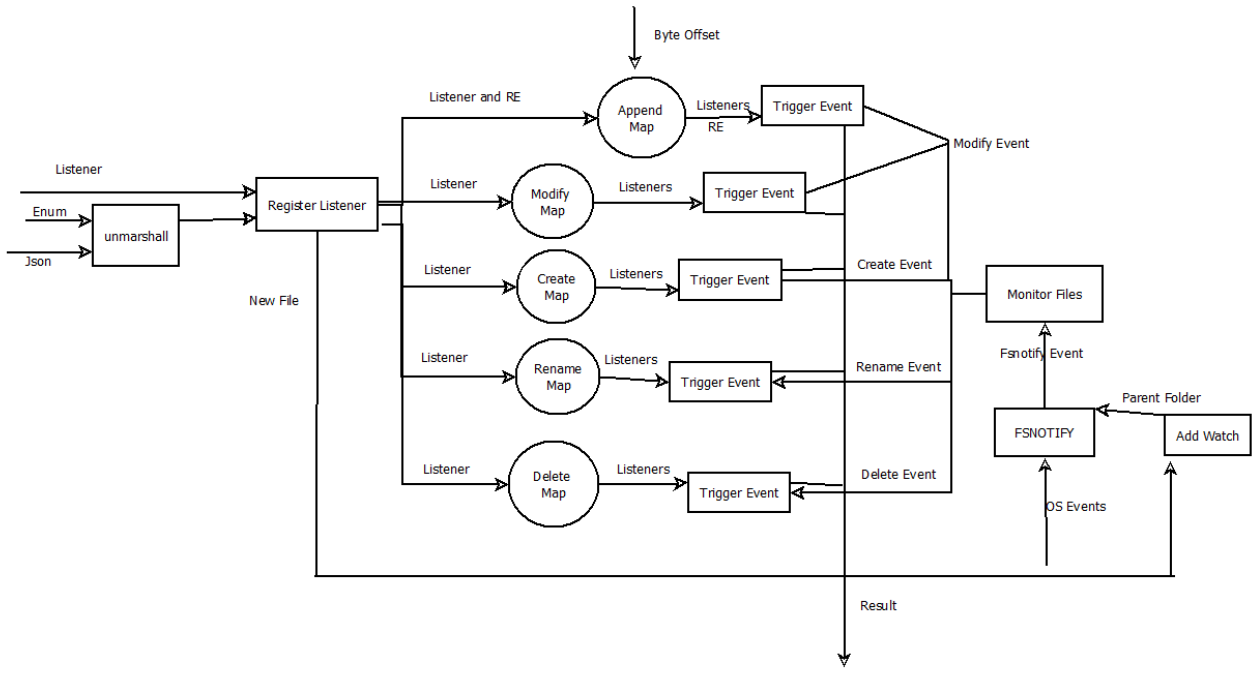


Figure 2 is a more in-depth look of how our system works. A listener is registered along with an EventType enumeration and a JSON string which contains the necessary information to be watching for. Depending on the enum, the listener is placed inside the appropriate map. When fsnotify returns an event, we pull the correct listener from the map and call its TriggerEvent method.

Figure 2. Detailed Process Flow



Design and Implementation Decision

1. We used fsnotify because it was written by the Go team and easily allowed us to track files without writing a low level operating system tracker ourselves. Writing it ourselves would have taken much longer and would have required much more knowledge of the system calls needed to track files and their changes for each operating system. Unfortunately fsnotify is still under development, which limited our usability of the resource.

2. Because our system needs to be able to continue to monitoring files while registering listeners and while other processes and services are running, the monitoring of files is done in a goroutine (simple thread). This decouples entire processes concurrent with registering listeners and allows it not to miss events. The goroutine also pulls from a channel, which “alerts” processes when the channel has something in it. This means the goroutine is not wasting processing cycles checking if there are objects in the channel, but rather passively receiving them.

3. We used a multiple maps to couple the “watches” from fsnotify to the resources we needed to be able to trigger events as needed. We have one map for each of the use cases that file monitoring allows. When fsnotify triggers an event (deleting, renaming, modifying, or creating of a file), it sends an event to our system containing the filename. This filename is used as the key of the map and is then used to access the listeners as well as any other resources need to determine if an event needs to be sent from our system, as well as create the result that is sent to the listener. This allows shared resources for multiple listeners on the same file, rather than needlessly having copies of resources that are the same.

4. We used json for our results because it allows for a lot of flexibility in what and how much data our results contain. This also allows other modules to use the same “result” without having to create some struct with multiple unnecessary fields. The use of json was consistent with a request from the client.

5. Go provides some helpful libraries that were used in the project. Golang provided regular expression, json, file access and file path manipulation libraries that we could easily leverage to create our code. This allowed us to focus on more important details of our project.

6. Because maps in the Go Language are not thread safe, a mutex had to be created for each one. This is then locked before and unlocked after accessing the data inside the map. This keeps the maps threadsafe, and allows adding listeners to the map while still monitoring without risking corrupting the map.

Results

Our goal was to monitor files and send events based on user specified criteria. We were able to achieve this using a tool called fsnotify made by the GO language development team. Fsnotify is currently under development, which caused some limitations, but we were able to work around them. The project should be easily extensible to allow more types of events to be monitored with minimal additional coding.

The current implementation works on Windows and Linux operating systems. We were unable to test using OS X and BSD, but given that fsnotify is written to work across all major operating systems, we can be fairly confident that our implementation works on these as well.

Current features include tracking files to monitor create, delete, rename, and modify events. In addition, a user can choose to look for text appended to a file that matches a regular expression. This implementation can also track the existence of rolling log files.

We began this project by only tracking the individual files, with an option to watch a directory. We found that rename events were impossible to track using this method and chose the broader approach of watching the whole folder. The resulting behavior is that fsnotify will report all events related to that folder, but we filter out and deal only with events that relate to specified files.

For the future, we recommend an approach that watches the individual files while solving the problem we had with renames. This way, fsnotify only reports events that directly relate with the files we are trying to monitor. This approach was too hard to implement in our project partially to due fsnotify still being under development.