Saberx

Release 1.0

Jul 28, 2020

Contents:

SaberX is a trigger based monitoring, alerting and action execution system which can be used for self healing. SaberX watches for specific events in your system and fires its trigger when any such event happens. In reply to the firing of any such trigger, you can execute an action, like sending alert to you alert management system or any command to heal your system.

A very simple example would be waching your apache server and making sure its accessable at port 80. If its not, then you can configure saberx to fire a trigger for this. When such a trigger gets fired, you may send a curl request call to your alert manager to raise a alert and at the same time restart your apache server.

SaberX provides many more such triggers like filetrigger (watching over files), Process trigger (watching over processes), CPUTrigger (watching over CPU), memory trigger (watching over memory) and the already described TCP trigger (watching over ports).

Currently SaberX only supports Linux.

View SaberX on Github

CHAPTER 1

Getting started with SaberX

Following section shows how to quickly get started with SaberX with an simple example.

1.1 Installing SaberX

SaberX can be simply installed using following steps:

- Clone/download the master branch.
- Enter into the repo
- Run `sudo python3 setup.py install`
- Verify installation using `saberx --help`

It is to be noted that the `setuptools` pulls dependencies from `pypi.org`. If you want to use your own custom registry url for building dependencies then you can try one of the below mentioned ways.

Create a file called `.pydistutils.cfg` under your home directory with the below content:

```
[easy_install]
index-url = https://your-custom.url
```

Once this file has been created you can continue with the normal installation procedure mentioned above. The registry url provided by you will be used rather than PyPi.

If you want to develop SaberX then you can also install SaberX in development mode with your custom registry url. In this case you do not require the `.pydistutils.cfg` file. Just use the below mentioned command for installing SaberX.

```
sudo python3 setup.py develop --index-url=https://your-custom.url
```

This is will install SaberX in development mode. You can make changes to source on the fly and when you run SaberX, changes will be reflected, you will not have to build SaberX again and again.

1.2 Setting up a simple Trigger and action

Lets setup a simple trigger like the once metioned in the above example. We will be setting up a trigger for Apache web server. The trigger will check whether the server is accessable (accepting connections) at port 80. If not, the trigger will be fired, and as a response to this trigger we will restart Apache.

Open **/etc/saberx/saberx.yaml** and paste the following in it:

```
actiongroups:
- groupname: grp1
  actions:
  - actionname: action_1
    trigger:
      type: TCP_TRIGGER
      check: tcp_fail
      host: 127.0.0.1
      port: 80
      attempts: 3
      threshold: 2
    execute:
    - "systemctl restart apache2"
```

Open **/etc/saberx/saberx.conf** and paste the following:

```
[DEFAULT]

action_plan = /etc/saberx/saberx.yaml
lock_dir = /run
sleep_period = 5
```

Note

```
Note: The above example assumes that you have Apache web server installed and␣
↪configured to receive connections at port 80
It also assumes that Apache is being managed by systemd, which is the default case␣
↪with most debian based systems.
```

Make sure the server is up and running.

Now start saberx by just typing `saberx`. Optionally you can start saberx using `saberx &` to push it to background. Alternatively you can create a systemd service file for saberx (more on that later). The user with which saberx is being run should have permission to restart Apache2.

In order to simulate an issue (refusal of connections at port 80), we will intentionally stop apache2:

```
sudo systemctl stop apache2
```

This will cause Apache2 to stop listening and port 80 and start rejecting connections. It will not take apache more than 5 seconds (since that is the sleep time we have configured and can be reduced) to detect that Apache is refusing connections at port 80 and it will fire the TCPTrigger we have defined. Once that happens, the action we have provided will get excuted, thereby restarting Apache.

1.3 Understanding the config

```
actiongroups:
- groupname: grp1
  actions:
  - actionname: action_1
    trigger:
      type: TCP_TRIGGER
      check: tcp_fail
      host: 127.0.0.1
      port: 80
      attempts: 3
      threshold: 2
    execute:
    - "systemctl restart apache2"
```

The above is the Trigger and action configuration. It contains only one action group: grp1 and grp 1 contains only one action: action_1. There can be multiple action groups and each group can contain multiple triggers. More on this later.

The type of the Trigger we used above is `TCP_TRIGGER`. This triiger is only used to check tcp connections to given host, port.

`check` is set to `tcp_fail`. This essectially means that the trigger will get fired when saberx fails to open a tcp connection to the give host, port.

`host` is the host we are monitoring and `port` is the respective port we are openning the TCP connection to.

`attempts` indicate the number of attempts we are going to make. Here saberx tries to make 3 attempts at openning a tcp connection port 80. `threshold` is the minimum number of times the tcp connection should fail to fire the trigger. So. in the above example if saberx fails to open TCP connection to 127.0.0.1:80 twice out of the three times it will try, then the trigger will get fired.

Under `execute` we give a list of commands to be executed when a trigger is fired. It can be any linux commnad or script that needs to be executed. It is to be noted that if any command in the list of command fails or throws error or exit code if non 0, the rest of the commands after that is ignored.

```
[DEFAULT]

action_plan = /etc/saberx/saberx.yaml
lock_dir = /run
sleep_period = 5
```

This is the main conf file. It contains path to the yaml file containg the actions and triggers.

`action_plan` is the path to the yaml file containing actions and triggers (the one mentioned above)

`lock_dir` is the directory where saberx stores a lock file. This file acts as a lock making sure the next run of saberx takes place only after the previous run has ended and all old threads are gone.

`sleep_period` is the amount of time (in seconds) saberx will wait before initiating the next run.

CHAPTER 2

How SaberX works

This setion describes how SaberX works, what it does and how it does and how to configure it properly.

2.1 Actions and Groups

As already mentioned above, in the yaml file (which we will refer as the **action yaml**), we can provide a list of groups.

Each group comprises of a list of actions.

Each action comprises of a **trigger** and a **execute** section.

The important and interesting thing to be notes here is SaberX evaluates all the groups concurrently. It spawns a thread for each group. So two actions in two different group will be executed concurrently. However, inside the same group, the actions are executed synchronously.

Lets take an example:

```
actiongroups:
- groupname: grp1
  actions:
  - actionname: action_1
    trigger:
      type: TCP_TRIGGER
      check: tcp_fail
      host: 127.0.0.1
      port: 80
      attempts: 3
      threshold: 1
    execute:
    - "command 1"
    - "command 2"
  - actionname: action_2
    trigger:
      type: TCP_TRIGGER
```

(continues on next page)

(continued from previous page)

```
      check: tcp_fail
      host: 127.0.0.1
      port: 443
      attempts: 3
      threshold: 1
    execute:
    - "command 1"
    - "command 2"
- groupname: grp2
  actions:
  - actionname: action_1
    trigger:
      type: PROCESS_TRIGGER
      check: cmdline
      regex: "nginx"
      count: 1
      operation: '>='
    execute:
    - "command 1"
```

In the above action yaml, actiongroups grp1 and grp2 will be run concurrently. SaberX will spawn two threads and allocate one group to each thread. This would mean both the TCP triggers will be evaluated concurrently on each run with the process trigger (more on process trigger below). However both tcp triggers will be evaluated synchronously. That is first saberx will evaluate the tcp trigger for port 80 and then for port 443.

If the above seems to be confusing, the here is a small description of the control flow for the above config.

At the start of each run, SaberX will spawn a thread for each group. In this case there will be two threads in total. The thread responsible for a group, lets consider the first group, will first evaluate grp1, once it is done with grp1 (if trigger is fired it will run commands or simply pass), it will try to evaluate the second trigger (the one for port 443).

The second thread responsible for grp2 will also be excuting concurrently along with grp1 and hence the action contained with in grp2 (and hence the trigger and execute sections) will be evaluated concurrently to the actions present in grp1.

Here another important thing worth noting is, if in the same group, one action's execute section fails, that action is marked as a failure and all the remaining actions in that group will be ignored.

For example, in the above scanario, if command 1 in action_1 in grp_1 fails (throws exception, returns non 0 exit code), then action_1 will be marked as a failure and action_2 in grp1 will be skipped in that run. However this wont affect any action in grp2.

In short, if you want your actions to be evaluated concurrently with no **dependency** between them, consider putting them in separate groups. If you want your actions to be synchronous, then put them in same group.

2.2 Execute section

Each trigger should have an execute section. This section/key contains a list of commands to be executed if the corresponding trigger is fired. Lets take an example:

```
actiongroups:
- groupname: grp1
  actions:
  - actionname: action_1
    trigger:
```

(continues on next page)

(continued from previous page)

```
    type: TCP_TRIGGER
    check: tcp_fail
    host: 127.0.0.1
    port: 80
    attempts: 3
    threshold: 2
  execute:
  - "command 1"
  - "command 2"
  - "command 3"
  - "command 4"
```

In this example, if the above trigger fails, that is saberx is unable to open connection to 127.0.0.1:80, then SaberX will try to execute all the 4 commands one after the other synchronously.

Omce again, it is to be noted over here that if any of the commands fail, the rest of the command will be ignored in that run.

SaberX marks the execution of a command as failure if, the command throws an error/exception or it returns a non 0 return code.

2.3 What happens in SaberX run

Before SaberX initiates a run, it tries to aquire a lock. It does so by trying to create a lock file in the configured lock directory (present in /etc/saberx/saberx.conf).

If SaberX fails to create the file, the run fails. If a lock file is already present, it means the previous run is not yet finished, so it does nothing and waits for the next turn.

If it succeeds in creating the lock file, then the run begins. SaberX first parses the action yaml, extarcts all the groups. Following that it spawns one thread for each group. Each thread evaluates the actions of the concerned groups. Whenever a triiger in any action is fired, it executes the commands present in that action's execute section.

Once all the threads have done their job, the lock file is deleted and SaberX waits for the next run. If SaberX is unable to delete the lock file it throws error and exits.

CHAPTER 3

Triggers

SaberX provides the following 5 triggers as of now:

- TCP_TRGIGGER
- PROCESS_TRIGGER
- CPU_TRIGGER
- MEMORY_TRIGGER
- FILE_TRIGGER

3.1 TCP_TRIGGER

TCP_TRIGGER watches for tcp connection to a given host and port. It gets triggered when it succeeds/fails in creating normal/ssl connection to a given host and port.

Example:

```
- actionname: action_1
  trigger:
    type: TCP_TRIGGER
    check: tcp_fail
    host: 127.0.0.1
    port: 8899
    attempts: 3
    threshold: 1
  execute:
  - "command 1"
```

- `type` tells what kind of trigger it is. It is mandatory for all triggers.
- `check` denotes what we want to check. If we want to fire our trigger on tcp failure then we have to set this to `tcp_failure`. If we want it to fire on tcp connect, then we have to set this to `tcp_connect`
- `host` tells the host to connect to. Can be hostname or IP address. Default is `127.0.0.1`.

- `port` tell the port of the host to connect to. Default is `80`.
- `ssl` need to be set to True we want to create TCP connection with SSL or else False. Default is `False`.
- `timeout` is the time in seconds after which saberx will give up trying to establish the connection and report as failure. Default is 5.
- `attempts` is the number of times saberx should try to establish a connection to the given host, port. Default is 3.
- `threshold` is the minimum number of success or failure saberx must enounter in order to report the same.

3.2 PROCESS_TRIGGER

Process trigger watches for processes with given name/regex or commandline arguments matcing given regex. If saberx finds a process with the matching conditionals, then it fires this trigger based on certain conditions. For example you can instruct saberx to fire process trigger if there are more than (or less than or equal to) 5 (or any number) process with the name "nginx" running in the system.

Example:

```
- actionname: action_2
  trigger:
    type: PROCESS_TRIGGER
    check: cmdline
    regex: "k.* start"
    count: 1
    operation: '>='
  execute:
  - "command 1"
```

- `type` tells what kind of trigger it is. It is mandatory for all triggers.
- `check` can be set to either `name` or `cmdline`. If set to name then saberx will look for name and if set to cmdline then it will look out for processes with arguments matching the given regex.
- `regex` is the regex pattern to match against the process name or command line arguments.
- `count` can be any integer. SaberX checks if the number of desired procsses in the system are greater than or less than or equal to (as configured) `count` then the trigger is fired. Default is `1`.
- `operation` can be anything among `<, >, <=, >=, =`. This is how SaberX will compare the number of desired processes against the provided `count` in order to fire the trigger. Default is =.

In the above example, the trigger will be fired if the number of processes in the system having command line matching the given regex is greater than or equal to 1.

3.3 CPU_TRIGGER

CPU trigger watches over the loadaverage of the system. If the loadaverage (1, 5, 15) is more, less or equal (as desired) than the configured value, this trigger will get fired.

Example:

```
- actionname: action_3
  trigger:
    type: CPU_TRIGGER
```

(continues on next page)

(continued from previous page)

```
    check: loadaverage
    threshold:
    - 10.0
    - 10.0
    - 10.0
    operation: '>'
  execute:
  - "command 1"
```

The above trigger will get fired if last 1, 5 and 15 min load average is greater than 10.0.

- `type` tells what kind of trigger it is. It is mandatory for all triggers.
- `check` as of now can only be `loadaverage`
- `threshold` is a list of thresholds for 1, 5 and 15 min load average. must be `float`
- `operation` is the operation to be performed in order to compare current loadaverage with the thresholds. This can be set to either of `<, >, <=, >=, =`. Default is `>`.

3.4 MEMORY_TRIGGER

MEMORY_TRIGER watches over the memory of the system and fires the trigger if a given metric (used, free, available) of the given type of memory (swap or virtual) breaches the given threshold.

Example:

```
- actionname: action_4
  trigger:
    type: MEMORY_TRIGGER
    check: virtual
    attr: used
    threshold: 5368709120.0
    operation: '>'
  execute:
  - "command 1"
```

The above trigger gets fired when used virtual memory in the system goes above 5368709120.0.

- `type` tells what kind of trigger it is. It is mandatory for all triggers.
- `check` can be either `virtual` or `swap`. It denotes the type of memory to check.
- `attr` can be either of `used, free or available`. Default is `used`.
- `threshold` is the breach value. Must be `float`.
- `operation` is the operation to be performed in order to compare current memory metric with the threshold. This can be set to either of `<, >, <=, >=, =`. Default is `>`

3.5 FILE_TRIGGER

FILE_TRIGGER is fired when a certain condtion is met in a file. For example this trigger can be configured such that if the last 10 lines of a log file has a certain text (pattern given by a regex), then this trigger will get fired. It can also be made to fire if a certain file is present, empty.

Example:

```
- actionname: action_5
  trigger:
    type: FILE_TRIGGER
    check: regex
    path: "/var/log/apache2/error.log"
    regex: ".*act[a-z]{2}ns"
    limit: 10
    position: head
  execute:
  - "command 1"
```

The above trigger gets fired when the apache2 error log file given by the path param has something matching the given regex in the first 10 lines.

- `type` tells what kind of trigger it is. It is mandatory for all triggers.
- `check` can be wither of `empty, present, regex`. Seting it to empty fires the trigger when the file is empty, present fires it when the file is present. Setting it regex will search for the regex inside the file along with other params.
- `path` is the path of the file resourse. Must be abosulute path.
- `regex` is the regex (pattern) to search in the file.
- `limit` is the limit for the number of lines (from bottom or top) to search the regex in. Must be as integer. Default is `50`.
- `position` denotes whether to search for the given regex in the file from head or tail. Value can be either of `head` or `tail`.

For all of the above mentioned triggers, `negate` param can be used. It simply negates the status of the trigger. By default its False. For example in case of file trigger, if type if `present` and the file is absent, trigger status will be false. However if `negate` is set to true, then it will fire the trugger since the status will not become true.

CHAPTER 4

Running SaberX

SaberX can be ran easily by just typing the following:

`sudo saberx` or `sudo saberx &`

In the above saberx is run with superuser priviledges. However if all the actions/commands that you want saberx to perform can be done by a normal user, then saberx can be run with that user.

The preferred method to run saberx on Debian based linux systems would be by creating a service file for it.

CHAPTER 5

SaberX Code Documentation

5.1 driver : Main entry point for SaberX

5.2 threadexecuter : Module for spawning threads and executing groups.

class saberx.executers.threaddriver.**ThreadExecuter**(***kwargs*)

Class for spawning and managing threads for executing groups

spawn_workers(*lock_file*)

** Method to spawn threads**

This method is used to spawn new threads to execute groups. Each thread calls the __worker fuction as target with a given group.

Parameters **lock_file** (*string*) – Path to lock file

Returns; bool: Threads spawned and executed successfully or not.

5.3 groupexecuter : Module for executing a group of actions.

class saberx.executers.groupexecuter.**GroupExecuter**

Class for handling executing of a group of actions

static execute_group(***kwargs*)

Method for executing a group of actions

This method takes a group of actions. It then ieterates over thoses group actions and executes them one by one using the required actionexecuter module.

It is important to be noted here that actions in a group are executed synchronously, and if one action in the pipeline fails, ie, triggered but command executions fails due to some excpetion or error, the entire pipeline after the failed action is ignored.

If you dont wont the above dependency between your actions, it is advised to place the actions in different groups. Groups have no such dependencies are executed concurrently.

5.4 actionexecuter : Module for executing an action

class saberx.executers.actionexecuter.**ActionExecuter**

class for handling execution of a given action. This class mostly comrises of status function.

static execute_action(***kwargs*)

Method to execute a given action

This method executes a given action. It fires the associated trigger using the required trigger handler if trigger is successfull, executes the desired commands.

The layout of a action will be as follows:

action_name: string trigger:

type: TCP_TRIGGER check: tcp_connect | tcp_fail host: host_name port: port negate: true | false attemp: number threshold: number ssl: true | false

execute: - command1 - command2

Parameters **kwargs** – Object containing action, thread lock and logger

Returns Success or failure for this action

Return type bool

5.5 shellexecuter : Module for executing shell commands

class saberx.sabercore.shellexecutor.**ShellExecutor**(***kwargs*)

Class for executing shell commands

execute_shell_list()

Method for executing a list of shell commands

This method executes a list of shell commands.

It is to be notes that, if a single command fails, the remaning commands adter the failed command will be ignored.

Returns status of execution of the commands.

Return type bool

execute_shell_single(*command*)

Method for executing a single shell command

This method executes a single shell command

Parameters **command** (*string*) – command tp be exeuted

Returns status of the command execution, output of the command, return code

Return type bool, output, proc_exit_code

5.6 tcptrigger : Module for firing tcp trigger

class saberx.sabercore.triggers.tcptrigger.**TCPTrigger**(***kwargs*)
Method for initialing memory trigger

fire_trigger()
Method to fire the trigger

This method first sanitises the parameters, calls tcp handler to evaluate the trigger conditions and returns trigger status

Returns Trigger fired or not

Return type bool

sanitise()
Method to check validity of the params

Returns params are proper or not

Return type bool

5.7 tcphandler : Module for evaluating tcptrigger trigger.

class saberx.sabercore.triggers.tcphandler.**TCPHandler**
Class containing TCP handler methods

static check_connection(***kwargs*)
Method to evaluate the TCP trigger

This method evaulates the TCP trigger. Calls the desired methods to check tcp (normal or ssl) to a host.

Parameters **kwargs** (*dict*) – dict containing host, port, ssl, timeout, attempts, threshold, check_type

Returns status, error if any

Return type bool, error

static check_tcp(***kwargs*)
Check tcp connection to a host

This method tries to open a tcp connection to a host.

Parameters **kwargs** (*dict*) – dict containing host, port and timeout

Returns Whether the tcp connection could be established or not.

Return type bool

static check_tcp_ssl(***kwargs*)
Check tcp connection with ssl to a host

This method tries to open a ssl tcp connection to a host.

Parameters **kwargs** (*dict*) – dict containing host, port and timeout

Returns Whether the tcp connection could be established or not.

Return type bool

5.8 cputrigger : Module for firing CPU trigger

class saberx.sabercore.triggers.cputrigger.**CPUTrigger**(***kwargs*)

Class for creating CPU trigger

fire_trigger()

Method to fire the trigger

This method first sanitises the parameters, calls cpu handler to evaluate the trigger conditions and returns trigger status

Returns Trigger fired or not

Return type bool

sanitise()

Method to check validity of the params

Returns params are proper or not

Return type bool

5.9 cpuhandler : Module for evaluating trigger conditions

class saberx.sabercore.triggers.cpuhandler.**CPUHandler**

Class for evaluatingcputrigger params

static check_loadavg(***kwargs*)

Method to check loadaverage

This method evaluates the trigger param and descied whether the trigger is fired or not.

Parameters **kwargs** (*object*) – Object containing thresholds and operation

Returns status of the trigger and error string if any

Return type bool, string

5.10 memorytrigger : Module for firing memory trigger

class saberx.sabercore.triggers.memorytrigger.**MemoryTrigger**(***kwargs*)

Class for creating memory trigger

fire_trigger()

Method to fire the trigger

This method first sanitises the parameters, calls memory handler to evaluate the trigger conditions and returns trigger status

Returns Trigger fired or not

Return type bool

sanitise()

Method to check validity of the params

Returns params are proper or not

Return type bool

5.11 memoryhandler : Module for performing the memory trigger operation

class saberx.sabercore.triggers.memoryhandler.**MemoryHandler**

Class for handling memory trigger operation

static check_mem(***kwargs*)

Method to perform the memory operation

This method accpets the trigger attributes, performs the specified operation and returns the trigger status.

Parameters **kwargs** (*dict*) – Contains all check attributes

Returns trigger status - fire or not

Return type bool

static get_mem_type(*check*)

Method for getting metric type from trigger check

This method accepts the check type and returns the desired method to get the value of the metric specified in type.

Parameters **check** (*string*) – Type of check : virtial | swap

Returns psutl method to get virtual or swap memory values.

Return type psutil method

5.12 processtrigger : Module for firing process trigger

class saberx.sabercore.triggers.processtrigger.**ProcessTrigger**(***kwargs*)

Class for creating process trigger

fire_trigger()

Method to fire the trigger

This method first sanitises the parameters, calls process handler to evaluate the trigger conditions and returns trigger status

Returns Trigger fired or not

Return type bool

sanitise()

Method to check validity of the params

Returns params are proper or not

Return type bool

5.13 processhandler : Module for evaluating process trigger

class saberx.sabercore.triggers.processhandler.**ProcessHandler**

**Class for performing process trigger operation

static check_cmdline(*regex*)
Method for checking if a process exists by cmdline text

This method checks if there is any process whose cmdline arg matches the given pattern

Parameters **regex** (*string*) – String containing regex

Returns status, count, error if any

Return type bool, Integer, String

static check_cmdline_count(*regex*, *count*, *operator*)
Method to get the Number of regex filtered processes by cmd and perform the deired operation

Parameters

- **regex** (*string*) – Regex to filter processes
- **count** (*string*) – Threshold
- **operation** (*string*) – Desired operation

Returns Result, error if any

Return type bool, string

static check_name(*regex*)
Method for checking if a process exists by name

This method checks if there is any process whose name matches the given pattern

Parameters **regex** (*string*) – String containing regex

Returns status, count, error if any

Return type bool, Integer, String

static check_name_count(*regex*, *count*, *operator*)
Method to get the Number of regex filtered processes and perform the deired operation

Parameters

- **regex** (*string*) – Regex to filter processes
- **count** (*string*) – Threshold
- **operation** (*string*) – Desired operation

Returns Result, error if any

Return type bool, string

static get_cmdline_count(*regex*)
Method for getting count of process

This method returns the count of processes whose cmdline matches the given regex

Parameters **regex** (*string*) – String containing regex for filtering process cmdline

Returns status, count, error if any

Return type bool, Integer, String

static get_name_count(*regex*)
Method for getting count of process

This method returns the count of processes whose name matches the given regex

Parameters **regex** (*string*) – String containing regex for filtering process names

Returns status, count, error if any

Return type bool, Integer, String

5.14 filetrigger : Module for firing file trigger

class saberx.sabercore.triggers.filetrigger.**FileTrigger**(***kwargs*)

Class for creating file trigger

fire_trigger()

Method to fire the trigger

This method first sanitises the parameters, calls file handler to evaluate the trigger conditions and returns trigger status

Returns Trigger fired or not

Return type bool

sanitise()

This method must be implemented by child class

5.15 filehandler : Module to evaluate file trigger

class saberx.sabercore.triggers.filehandler.**FileHandler**

** Class containing file handling methods **

static is_empty(*path*)

Method for checking if given file is empty

Parameters **path** (*string*) – path to the file resource

Returns status, error if any

Return type bool, string

static is_present(*path*)

Method for checking if the given file exists

This method checks whether the given path is valid or not.

Parameters **path** (*string*) – path to the file resource

Returns status, error if any

Return type bool, string

static read_from_head(*path*, *regex*, *limit*)

Method to read a file from head and see if pattern exists

This method reads a given file and checks if the given pattern exists in the top 'n' lines where n is given as 'limits'.

Parameters

- **path** (*string*) – path to the file resource
- **regex** (*string*) – string representing the pattern to search for
- **limit** (*Integer*) – Number of lines to query

Returns status , error

Return type bool, string

static read_from_tail(*path*, *regex*, *lines*)

Method to read a file from end and see if pattern exists

This method reads a given file and checks if the given pattern exists in the last 'n' lines where n is given as 'lines'.

Parameters

- **path** (*string*) – path to the file resource
- **regex** (*string*) – string representing the pattern to search for
- **limit** (*Integer*) – Number of lines to query

Returns status , error

Return type bool, string

static search_keyword(***kwargs*)

Method to execute the file operation

Method to perform the reuired file operations to execute the trigger

Parameters **kwargs** (*dict*) – Dict containing path, pattern, limit and position to read the file from

Returns status, error if any

Return type bool, string

5.16 Indices and tables

- genindex
- modindex
- search

Index