You are looking at documentation for an older release. Not what you want? See the current release documentation (../current/index.html).

aggregate

- Version: 2.5.2
- Released on: March 25, 2017
- Changelog (https://github.com/logstash-plugins/logstash-filter-aggregate/blob/master/CHANGELOG.md#252)

Installation

For plugins not bundled by default, it is easy to install by running `bin/logstash-plugin install logstash-filter-aggregate`. See *Working with plugins* (working-with-plugins.html) for more details.

Getting Help

For questions about the plugin, open a topic in the Discuss (http://discuss.elastic.co) forums. For bugs or feature requests, open an issue in Github (https://github.com/elastic/logstash). For the list of Elastic supported plugins, please consult the Elastic Support Matrix (https://www.elastic.co/support/matrix#show_logstash_plugins).

Description

The aim of this filter is to aggregate information available among several events (typically log lines) belonging to a same task, and finally push aggregated information into final task event.

You should be very careful to set Logstash filter workers to 1 (`-w 1` flag) for this filter to work correctly otherwise events may be processed out of sequence and unexpected results will occur.

Example #1

- with these given logs :

```
INFO - 12345 - TASK_START - start
INFO - 12345 - SQL - sqlQuery1 - 12
INFO - 12345 - SQL - sqlQuery2 - 34
INFO - 12345 - TASK_END - end
```

- you can aggregate "sql duration" for the whole task with this configuration :

```
filter {
  grok {
    match => [ "message", "%{LOGLEVEL:loglevel} - %{NOTSPACE:taskid} - %{NOTSPACE:l
ogger} - %{WORD:label}( - %{INT:duration:int})?" ]
  }

  if [logger] == "TASK_START" {
    aggregate {
      task_id => "%{taskid}"
      code => "map['sql_duration'] = 0"
      map_action => "create"
    }
  }

  if [logger] == "SQL" {
    aggregate {
      task_id => "%{taskid}"
      code => "map['sql_duration'] += event.get('duration')"
      map_action => "update"
    }
  }

  if [logger] == "TASK_END" {
    aggregate {
      task_id => "%{taskid}"
      code => "event.set('sql_duration', map['sql_duration'])"
      map_action => "update"
      end_of_task => true
      timeout => 120
    }
  }
}
```

- the final event then looks like :

```
{
       "message" => "INFO - 12345 - TASK_END - end message",
  "sql_duration" => 46
}
```

the field `sql_duration` is added and contains the sum of all sql queries durations.

Example #2 : no start event

- If you have the same logs than example #1, but without a start log :

```
 INFO - 12345 - SQL - sqlQuery1 - 12
 INFO - 12345 - SQL - sqlQuery2 - 34
 INFO - 12345 - TASK_END - end
```

- you can also aggregate "sql duration" with a slightly different configuration :

```
 filter {
   grok {
     match => [ "message", "%{LOGLEVEL:loglevel} - %{NOTSPACE:taskid} - %{NOTSPACE:l
ogger} - %{WORD:label}( - %{INT:duration:int})?" ]
   }

   if [logger] == "SQL" {
     aggregate {
       task_id => "%{taskid}"
       code => "map['sql_duration'] ||= 0 ; map['sql_duration'] += event.get('durati
on')"
     }
   }

   if [logger] == "TASK_END" {
     aggregate {
       task_id => "%{taskid}"
       code => "event.set('sql_duration', map['sql_duration'])"
       end_of_task => true
       timeout => 120
     }
   }
 }
```

- the final event is exactly the same than example #1
- the key point is the "||=" ruby operator. It allows to initialize *sql_duration* map entry to 0 only if this map entry is not already initialized

Example #3 : no end event

Third use case: You have no specific end event.

A typical case is aggregating or tracking user behaviour. We can track a user by its ID through the events, however once the user stops interacting, the events stop coming in. There is no specific event indicating the end of the user's interaction.

In this case, we can enable the option *push_map_as_event_on_timeout* to enable pushing the aggregation map as a new event when a timeout occurs. In addition, we can enable *timeout_code* to execute code on the populated timeout event. We can also add *timeout_task_id_field* so we can correlate the task_id, which in this case would be the user's ID.

- Given these logs:

```
INFO - 12345 - Clicked One
INFO - 12345 - Clicked Two
INFO - 12345 - Clicked Three
```

- You can aggregate the amount of clicks the user did like this:

```
filter {
  grok {
    match => [ "message", "%{LOGLEVEL:loglevel} - %{NOTSPACE:user_id} - %{GREEDYDAT
A:msg_text}" ]
  }

  aggregate {
    task_id => "%{user_id}"
    code => "map['clicks'] ||= 0; map['clicks'] += 1;"
    push_map_as_event_on_timeout => true
    timeout_task_id_field => "user_id"
    timeout => 600 # 10 minutes timeout
    timeout_tags => ['_aggregatetimeout']
    timeout_code => "event.set('several_clicks', event.get('clicks') > 1)"
  }
}
```

- After ten minutes, this will yield an event like:

```
{
  "user_id": "12345",
  "clicks": 3,
  "several_clicks": true,
    "tags": [
       "_aggregatetimeout"
    ]
}
```

Example #4 : no end event and tasks come one after the other

Fourth use case : like example #3, you have no specific end event, but also, tasks come one after the other. That is to say : tasks are not interlaced. All task1 events come, then all task2 events come, ... In that case, you don't want to wait task timeout to flush aggregation map. * A typical case is aggregating results from jdbc input plugin. * Given that you have this SQL query : `SELECT country_name, town_name FROM town`* Using jdbc input plugin, you get these 3 events from :

```
  { "country_name": "France", "town_name": "Paris" }
  { "country_name": "France", "town_name": "Marseille" }
  { "country_name": "USA", "town_name": "New-York" }
```

- And you would like these 2 result events to push them into elasticsearch :

```
  { "country_name": "France", "towns": [ {"town_name": "Paris"}, {"town_name": "Mars
eille"} ] }
  { "country_name": "USA", "towns": [ {"town_name": "New-York"} ] }
```

- You can do that using `push_previous_map_as_event` aggregate plugin option :

```
 filter {
   aggregate {
     task_id => "%{country_name}"
     code => "
       map['country_name'] = event.get('country_name')
       map['towns'] ||= []
       map['towns'] << {'town_name' => event.get('town_name')}
       event.cancel()
     "
     push_previous_map_as_event => true
     timeout => 3
   }
 }
```

- The key point is that each time aggregate plugin detects a new `country_name`, it pushes previous aggregate map as a new Logstash event, and then creates a new empty map for the next country
- When 5s timeout comes, the last aggregate map is pushed as a new event
- Finally, initial events (which are not aggregated) are dropped because useless (thanks to `event.cancel()`)

How it works

- the filter needs a "task_id" to correlate events (log lines) of a same task
- at the task beggining, filter creates a map, attached to task_id
- for each event, you can execute code using *event* and *map* (for instance, copy an event field to map)

- in the final event, you can execute a last code (for instance, add map data to final event)
- after the final event, the map attached to task is deleted (thanks to `end_of_task => true`)
- an aggregate map is tied to one task_id value which is tied to one task_id pattern. So if you have 2 filters with different task_id patterns, even if you have same task_id value, they won't share the same aggregate map.
- in one filter configuration, it is recommanded to define a timeout option to protect the feature against unterminated tasks. It tells the filter to delete expired maps
- if no timeout is defined, by default, all maps older than 1800 seconds are automatically deleted
- all timeout options have to be defined in only one aggregate filter per task_id pattern. Timeout options are : timeout, timeout_code, push_map_as_event_on_timeout, push_previous_map_as_event, timeout_task_id_field, timeout_tags
- if `code` execution raises an exception, the error is logged and event is tagged *_aggregateexception*

Use Cases

- extract some cool metrics from task logs and push them into task final log event (like in example #1 and #2)
- extract error information in any task log line, and push it in final task event (to get a final event with all error information if any)
- extract all back-end calls as a list, and push this list in final task event (to get a task profile)
- extract all http headers logged in several lines to push this list in final task event (complete http request info)
- for every back-end call, collect call details available on several lines, analyse it and finally tag final back-end call log line (error, timeout, business-warning, ...)
- Finally, task id can be any correlation id matching your need : it can be a session id, a file path, ...

Synopsis

This plugin supports the following configuration options:

Required configuration options:

```
aggregate {
    code => ...
    task_id => ...
}
```

Available configuration options:

Setting	Input type	Required
`add_field` (plugins-filters-aggregate.html#plugins-filters-aggregate-add_field)	hash (configuration-file-structure.html#hash)	No
`add_tag` (plugins-filters-aggregate.html#plugins-filters-aggregate-add_tag)	array (configuration-file-structure.html#array)	No
`aggregate_maps_path` (plugins-filters-aggregate.html#plugins-filters-aggregate-aggregate_maps_path)	string (configuration-file-structure.html#string)	No
`code` (plugins-filters-aggregate.html#plugins-filters-aggregate-code)	string (configuration-file-structure.html#string)	Yes
`enable_metric` (plugins-filters-aggregate.html#plugins-filters-aggregate-enable_metric)	boolean (configuration-file-structure.html#boolean)	No
`end_of_task` (plugins-filters-aggregate.html#plugins-filters-aggregate-end_of_task)	boolean (configuration-file-structure.html#boolean)	No
`id` (plugins-filters-aggregate.html#plugins-filters-aggregate-id)	string (configuration-file-structure.html#string)	No
`map_action` (plugins-filters-aggregate.html#plugins-filters-aggregate-map_action)	string (configuration-file-structure.html#string)	No
`periodic_flush` (plugins-filters-aggregate.html#plugins-filters-aggregate-periodic_flush)	boolean (configuration-file-structure.html#boolean)	No

Setting	Input type	Required
`push_map_as_event_on_timeout`(plugins-filters-aggregate.html#plugins-filters-aggregate-push_map_as_event_on_timeout)	boolean (configuration-file-structure.html#boolean)	No
`push_previous_map_as_event`(plugins-filters-aggregate.html#plugins-filters-aggregate-push_previous_map_as_event)	boolean (configuration-file-structure.html#boolean)	No
`remove_field` (plugins-filters-aggregate.html#plugins-filters-aggregate-remove_field)	array (configuration-file-structure.html#array)	No
`remove_tag` (plugins-filters-aggregate.html#plugins-filters-aggregate-remove_tag)	array (configuration-file-structure.html#array)	No
`task_id` (plugins-filters-aggregate.html#plugins-filters-aggregate-task_id)	string (configuration-file-structure.html#string)	Yes
`timeout` (plugins-filters-aggregate.html#plugins-filters-aggregate-timeout)	number (configuration-file-structure.html#number)	No
`timeout_code` (plugins-filters-aggregate.html#plugins-filters-aggregate-timeout_code)	string (configuration-file-structure.html#string)	No
`timeout_tags` (plugins-filters-aggregate.html#plugins-filters-aggregate-timeout_tags)	array (configuration-file-structure.html#array)	No
`timeout_task_id_field`(plugins-filters-aggregate.html#plugins-filters-aggregate-timeout_task_id_field)	string (configuration-file-structure.html#string)	No

Details

add_field

- Value type is hash (configuration-file-structure.html#hash)
- Default value is `{}`

If this filter is successful, add any arbitrary fields to this event. Field names can be dynamic and include parts of the event using the `%{field}`.

Example:

```
filter {
  aggregate {
    add_field => { "foo_%{somefield}" => "Hello world, from %{host}" }
  }
}
```

```
# You can also add multiple fields at once:
filter {
  aggregate {
    add_field => {
      "foo_%{somefield}" => "Hello world, from %{host}"
      "new_field" => "new_static_value"
    }
  }
}
```

If the event has field `"somefield" == "hello"` this filter, on success, would add field `foo_hello` if it is present, with the value above and the `%{host}` piece replaced with that value from the event. The second example would also add a hardcoded field.

add_tag

- Value type is array (configuration-file-structure.html#array)
- Default value is `[]`

If this filter is successful, add arbitrary tags to the event. Tags can be dynamic and include parts of the event using the `%{field}` syntax.

Example:

```
filter {
  aggregate {
    add_tag => [ "foo_%{somefield}" ]
  }
}
```

```
# You can also add multiple tags at once:
filter {
  aggregate {
    add_tag => [ "foo_%{somefield}", "taggedy_tag"]
  }
}
```

If the event has field `"somefield" == "hello"` this filter, on success, would add a tag `foo_hello` (and the second example would of course add a `taggedy_tag` tag).

aggregate_maps_path

- Value type is string (configuration-file-structure.html#string)
- There is no default value for this setting.

The path to file where aggregate maps are stored when Logstash stops and are loaded from when Logstash starts.

If not defined, aggregate maps will not be stored at Logstash stop and will be lost. Must be defined in only one aggregate filter (as aggregate maps are global).

Example:

```
filter {
  aggregate {
    aggregate_maps_path => "/path/to/.aggregate_maps"
  }
}
```

code

- This is a required setting.
- Value type is string (configuration-file-structure.html#string)
- There is no default value for this setting.

The code to execute to update map, using current event.

Or on the contrary, the code to execute to update event, using current map.

You will have a *map* variable and an *event* variable available (that is the event itself).

Example:

```
filter {
  aggregate {
    code => "map['sql_duration'] += event.get('duration')"
  }
}
```

enable_metric

- Value type is boolean (configuration-file-structure.html#boolean)
- Default value is `true`

Disable or enable metric logging for this specific plugin instance by default we record all the metrics we can, but you can disable metrics collection for a specific plugin.

`end_of_task`

- Value type is boolean (configuration-file-structure.html#boolean)
- Default value is `false`

Tell the filter that task is ended, and therefore, to delete aggregate map after code execution.

`id`

- Value type is string (configuration-file-structure.html#string)
- There is no default value for this setting.

Add a unique `ID` to the plugin configuration. If no ID is specified, Logstash will generate one. It is strongly recommended to set this ID in your configuration. This is particulary useful when you have two or more plugins of the same type, for example, if you have 2 grok filters. Adding a named ID in this case will help in monitoring Logstash when using the monitoring APIs.

```
output {
 stdout {
   id => "my_plugin_id"
 }
}
```

`map_action`

- Value type is string (configuration-file-structure.html#string)
- Default value is `"create_or_update"`

Tell the filter what to do with aggregate map.

`"create"`: create the map, and execute the code only if map wasn't created before

`"update"`: doesn't create the map, and execute the code only if map was created before

`"create_or_update"`: create the map if it wasn't created before, execute the code in all cases

`periodic_flush`

- Value type is boolean (configuration-file-structure.html#boolean)
- Default value is `false`

Call the filter flush method at regular interval. Optional.

push_map_as_event_on_timeout

edit (https://github.com/logstash-plugins/logstash-filter-aggregate/edit/master/lib/logstash/filters/aggregate.rb)

- Value type is boolean (configuration-file-structure.html#boolean)
- Default value is `false`

When this option is enabled, each time a task timeout is detected, it pushes task aggregation map as a new Logstash event. This enables to detect and process task timeouts in Logstash, but also to manage tasks that have no explicit end event.

push_previous_map_as_event

edit (https://github.com/logstash-plugins/logstash-filter-aggregate/edit/master/lib/logstash/filters/aggregate.rb)

- Value type is boolean (configuration-file-structure.html#boolean)
- Default value is `false`

When this option is enabled, each time aggregate plugin detects a new task id, it pushes previous aggregate map as a new Logstash event, and then creates a new empty map for the next task.

this option works fine only if tasks come one after the other. It means : all task1 events, then all task2 events, etc...

remove_field

edit (https://github.com/logstash-plugins/logstash-filter-aggregate/edit/master/lib/logstash/filters/aggregate.rb)

- Value type is array (configuration-file-structure.html#array)
- Default value is `[]`

If this filter is successful, remove arbitrary fields from this event. Example:

```
filter {
  aggregate {
    remove_field => [ "foo_%{somefield}" ]
  }
}
```

```
# You can also remove multiple fields at once:
filter {
  aggregate {
    remove_field => [ "foo_%{somefield}", "my_extraneous_field" ]
  }
}
```

If the event has field `"somefield" == "hello"` this filter, on success, would remove the field with name `foo_hello` if it is present. The second example would remove an additional, non-dynamic field.

`remove_tag`

- Value type is array (configuration-file-structure.html#array)
- Default value is `[]`

If this filter is successful, remove arbitrary tags from the event. Tags can be dynamic and include parts of the event using the `%{field}` syntax.

Example:

```
filter {
  aggregate {
    remove_tag => [ "foo_%{somefield}" ]
  }
}
```

```
# You can also remove multiple tags at once:
filter {
  aggregate {
    remove_tag => [ "foo_%{somefield}", "sad_unwanted_tag"]
  }
}
```

If the event has field `"somefield" == "hello"` this filter, on success, would remove the tag `foo_hello` if it is present. The second example would remove a sad, unwanted tag as well.

`task_id`

- This is a required setting.
- Value type is string (configuration-file-structure.html#string)
- There is no default value for this setting.

CONFIG OPTIONS # ## # The expression defining task ID to correlate logs.

This value must uniquely identify the task.

Example:

```
filter {
  aggregate {
    task_id => "%{type}%{my_task_id}"
  }
}
```

timeout

- Value type is number (configuration-file-structure.html#number)
- There is no default value for this setting.

The amount of seconds after a task "end event" can be considered lost.

When timeout occurs for a task, The task "map" is evicted.

Timeout can be defined for each "task_id" pattern.

If no timeout is defined, default timeout will be applied : 1800 seconds.

timeout_code

- Value type is string (configuration-file-structure.html#string)
- There is no default value for this setting.

The code to execute to complete timeout generated event, when `'push_map_as_event_on_timeout'` or `'push_previous_map_as_event'` is set to true. The code block will have access to the newly generated timeout event that is pre-populated with the aggregation map.

If `'timeout_task_id_field'` is set, the event is also populated with the task_id value

Example:

```
filter {
  aggregate {
    timeout_code => "event.set('state', 'timeout')"
  }
}
```

timeout_tags

- Value type is array (configuration-file-structure.html#array)
- Default value is `[]`

Defines tags to add when a timeout event is generated and yield

Example:

```
filter {
  aggregate {
    timeout_tags => ["aggregate_timeout']
  }
}
```

`timeout_task_id_field`

- Value type is string (configuration-file-structure.html#string)
- There is no default value for this setting.

This option indicates the timeout generated event's field for the "task_id" value. The task id will then be set into the timeout event. This can help correlate which tasks have been timed out.

For example, with option `timeout_task_id_field => "my_id"`,when timeout task id is `"12345"`, the generated timeout event will contain `'my_id' => '12345'`.

By default, if this option is not set, task id value won't be set into timeout generated event.

« age (plugins-filters-age.html) alter » (plugins-filters-alter.html)

Top Videos

On this page