grok_exporter Configuration

This page describes the `grok_exporter` configuration file. The configuration is in YAML format. An example configuration can be found in example/config.yml. The path to the configuration file is passed as a command line parameter when starting `grok_exporter`:

```
grok_exporter -config ./example/config.yml
```

Updating from Config Version 2

Configuration files are versioned. The `config_version` is specified in the `global` section of the configuration file, see global Section below.

The versions allow for backwards compatibility: If an incompatible change in the config format is introduced, we increment the version number. New `grok_exporter` versions support old configuration versions.

`grok_exporter` 1.0.0.RC3 introduced configuration version 3. While version 2 is still supported, we recommend updating to version 3. You can use the `-showconfig` command line option to convert the configuration automatically and print the result to the console:

```
grok_exporter -config /path/to/config_v2.yml -showconfig
```

The main difference between configuration version 2 and configuration version 3 is that the new version introduces an `imports` section for importing `grok_patterns` and `metrics` from external configuration files. The old `grok` section was renamed to `grok_patterns` and now contains just a plain list of grok patterns. There's also a minor change: `poll_interval_seconds` was renamed to `poll_interval`. The format is described in How to Configure Durations below.

The old documentation for configuration version 2 can be found in CONFIG_v2.md.

Overall Structure

The `grok_exporter` configuration file consists of six main sections:

```
global:
    # Config version
input:
    # How to read log lines (file or stdin).
imports:
    # External configuration files for grok patterns and for metrics.
grok_patterns:
    # Grok patterns.
metrics:
    # How to map Grok fields to Prometheus metrics.
server:
    # How to expose the metrics via HTTP(S).
```

The following shows the configuration options for each of these sections.

global Section

The `global` section is as follows:

```
global:
    config_version: 3
    retention_check_interval: 53s
```

The `config_version` specifies the version of the config file format. Specifying the `config_version` is mandatory, it has to be included in every configuration file. The current `config_version` is `3`.

The config file format is versioned independently of the `grok_exporter` program. When a new version of `grok_exporter` does not introduce incompatible changes to the config file, the `config_version` will remain the same.

The following table shows which `grok_exporter` version uses which `config_version`:

grok_exporter	config_version
≤ 0.1.4	1

grok_exporter	config_version
0.2.X, 1.0.0.RC1, 1.0.0.RC2	2
≥ 1.0.0.RC3	3

The `retention_check_interval` is the interval at which `grok_exporter` checks for expired metrics. By default, metrics don't expire so this is relevant only if `retention` is configured explicitly with a metric. The `retention_check_interval` is optional, the value defaults to `53s`. The default value is reasonable for production and should not be changed. This property is intended to be used in tests, where you might not want to wait 53 seconds until an expired metric is cleaned up. The format is described in How to Configure Durations below.

Input Section

`grok_exporter` supports three input types: `file`, `stdin`, and `webhook`. The following three sections describe the input types respectively:

File Input Type

The configuration for the `file` input type is as follows:

Example 1:

```
input:
    type: file
    path: /var/logdir1/*.log
    readall: false
    fail_on_missing_logfile: true
    poll_interval: 5s # should NOT be needed in most cases, see below
```

Example 2:

```
input:
    type: file
    paths:
    - /var/logdir1/*.log
    - /var/logdir2/*.log
    readall: false
    fail_on_missing_logfile: true
    poll_interval: 5s # should NOT be needed in most cases, see below
```

The `path` is the path to the log file. `path` is used if you want to monitor a single path. If you want to monitor a list of paths, use `paths` instead, as in example 2 above. Glob patterns are supported on the file level, but not on the directory level. If you want to monitor multiple logfiles, see also restricting a metric to specific log files and pre-defined label variables below.

The `readall` flag defines if `grok_exporter` starts reading from the beginning or the end of the file. True means we read the whole file, false means we start at the end of the file and read only new lines. True is good for debugging, because we process all available log lines. False is good for production, because we avoid to process lines multiple times when `grok_exporter` is restarted. The default value for `readall` is `false`.

If `fail_on_missing_logfile` is true, `grok_exporter` will not start if the `path` is not found. This is the default value, and it should be used in most cases because a missing logfile is likely a configuration error. However, in some scenarios you might want `grok_exporter` to start successfully even if the logfile is not found, because you know the file will be created later. In that case, set `fail_on_missing_logfile: false`.

On `poll_interval`: You probably don't need this. The internal implementation of `grok_exporter`'s file input is based on the operating system's file system notification mechanism, which is `inotify` on Linux, `kevent` on BSD (or macOS), and `ReadDirectoryChangesW` on Windows. These tools will inform `grok_exporter` as soon as a new log line is written to the log file, and let `grok_exporter` sleep as long as the log file doesn't change. There is no need for configuring a poll interval. However, there is one combination where the above notifications don't work: If the logging application keeps the logfile open and the underlying file system is NTFS (see #17). For this specific case you can configure a `poll_interval`. This will disable file system notifications and instead check the log file periodically. The format is described in How to Configure Durations below.

Stdin Input Type

The configuration for the `stdin` input type does not have any additional parameters:

```
input:
    type: stdin
```

This is useful if you want to pipe log data to the `grok_exporter` command. For example if you want to monitor the output of `journalctl`:

```
journalctl -f | grok_exporter -config config.yml
```

Note that `grok_exporter` terminates as soon as it finishes reading from `stdin`. That means, if you run `cat sample.log | grok_exporter -config config.yml`, the exporter will terminate as soon as `sample.log` is processed. In order to keep `grok_exporter` running, always use a command that keeps the output open, like `tail -f -n +1 sample.log | grok_exporter -config config.yml`.

Webhook Input Type

The `grok_exporter` is capable of receive log entries from webhook sources. It supports webhook reception in various formats... plain-text or JSON, single entries or bulk entries.

The following input configuration example which demonstrates how to configure grok_exporter to receive HTTP webhooks from the Logstash HTTP Output Plugin configured in `json_batch` mode, which allows the transmission of multiple json log entries in a single webhook.

```
input:

    type: webhook

    # HTTP Path to POST the webhook
    # Default is `/webhook`
    webhook_path: /webhook

    # HTTP Body POST Format
    # text_single: Webhook POST body is a single plain text log entry
    # text_bulk: Webhook POST body contains multiple plain text log entries
    #   separated by webhook_text_bulk_separator (default: \n\n)
    # json_single: Webhook POST body is a single json log entry.  Log entry
    #   text is selected from the value of a json key determined by
    #   webhook_json_selector.
    # json_bulk: Webhook POST body contains multiple json log entries.  The
    #   POST body envelope must be a json array "[ <entry>, <entry> ]".  Log
    #   entry text is selected from the value of a json key determined by
    #   webhook_json_selector.
    # json_lines: Webhook POST body contains multiple json log entries, with
    #   newline-separated log lines holding an individual json object. JSON
    #   object itself may not contain newlines. For example:
    #   example:
    #       { app="foo", stage="prod", log="example log message" }
    #       { app="bar", stage="dev", log="another line" }
    #   Log entry text is selected from the value of a json key determined
    #   by webhook_json_selector.
    # Default is `text_single`
    webhook_format: json_bulk

    # JSON Path Selector
    # Within an json log entry, text is selected from the value of this json selector
    #   Example ".path.to.element"
    # Default is `.message`
    webhook_json_selector: .message

    # Bulk Text Separator
    # Separator for text_bulk log entries
    # Default is `\n\n`
    webhook_text_bulk_separator: "\n\n"
```

This configuration example may be found in the examples directory here.

Kafka Input Type

The `grok_exporter` is also capable of consuming log entries from Kafka. Currently, only plain-text encoded messages are supported.

```
input:
  type: kafka
  # Version corresponding to the kafka cluster
  kafka_version: 2.1.0

  # The list of the Kafka brokers part of the Kafka cluster.  Please note that you need an inst
  kafka_brokers:
    - localhost:9092

  # The list of Kafka topics to consume from.
  kafka_topics:
    - grok_exporter_test

  # The assignor to use, which can be either range, roundrobin, sticky (range by default)
  kafka_partition_assignor: range

  # The name of the consumer group to register as on the broker.  If not specified, the default
  kafka_consumer_group_name: grok_exporter

  # Indicates if the exporter should start consuming as of the most recent messages in the topi
  kafka_consume_from_oldest: false
```

This configuration example may be found in the examples directory here.

imports Section

The imports section is used to load `grok_patterns` and `metrics` from external configuration files. This is optional, the configuration can be defined directly in the configuration file, as described in the grok_patterns Section and the metrics Section below.

Example:

```
imports:
- type: grok_patterns
  dir: ./logstash-patterns-core/patterns
- type: metrics
  file: /etc/grok_exporter/metrics.d/*.yaml
  defaults:
    path: /var/log/syslog/*
    retention: 2h30m0s
    buckets: [0, 1, 2, 3]
    quantiles: {0.5: 0.05, 0.9: 0.02, 0.99: 0.002}
    labels:
      logfile: '{{base .logfile}}'
```

The `type` can either be `grok_patterns` or `metrics`. Each import can either specify a `file` or a `dir`. The `file` is either a path to a config file, or a Glob pattern matching multiple config files. The `dir` is a directory, all files in that directory will be imported.

grok_patterns import type

The grok_exporter releases contain a `patterns/` directory with the pre-defined grok patterns from github.com/logstash-patterns-core. If you want to use them, configure an import for this directory. See the grok_patterns Section below for more information on the `grok_patterns`.

metrics import type

The external `metrics` configuration files are YAML files containing a list of metrics definitions. The contents is the same as in the metrics Section.

When importing metrics from external files, you can specify some default values in the `imports` section. If an imported metric configuration does not contain that value, the default from the `imports` is used. You can specify defaults for the following values:

- `path`, `paths`
- `retention`
- `buckets`
- `quantiles`
- `labels` (will be merged with the labels defined in the imported metrics)

The meaning of these values is defined in the metrics Section below.

grok_patterns Section

As described in the metrics Section below, each metric uses a regular expression to match log lines. Regular expressions quickly become complex and hard to read. Grok patterns are a way to break down regular expression into smaller snippets to improve readability.

The `grok_patterns` section configures these Grok patterns as a list of `name regular-expression-snippet` pairs. The regular expression snippets may themselves reference Grok patterns with the `%{name}` syntax.

An example of a `grok_patterns` section is as follows:

```
grok_patterns:
  - 'EXIM_MESSAGE [a-zA-Z ]*'
  - 'EXIM_SENDER_ADDRESS F=<%{EMAILADDRESS}>'
```

See the metrics Section below for more examples of Grok patterns and how to use them.

The `grok_patterns` section is optional. If you want to use plain regular expressions, you don't need to define Grok patterns.

The `grok_exporter` distribution includes a directory of pre-defined Grok patterns. These are taken from github.com/logstash-patterns-core. This directory can be imported as defined in the imports Section above.

Metrics Section

Metric Types Overview

The metrics section contains a list of metric definitions, specifying how log lines are mapped to Prometheus metrics. Four metric types are supported:

- Counter
- Gauge
- Histogram
- Summary

Example Log Lines

To exemplify the different metrics configurations, we use the following example log lines:

```
30.07.2016 14:37:03 alice 1.5
30.07.2016 14:37:33 alice 2.5
30.07.2016 14:43:02 bob 2.5
30.07.2016 14:45:59 alice 2.5
```

The following is a simple counter counting the number of log lines containing `alice`.

```
metrics:
    - type: counter
      name: alice_occurrences_total
      help: number of log lines containing alice
      match: 'alice'
      labels:
          logfile: '{{base .logfile}}'
```

Match

The `match` is a regular expression. In the simple example above, `alice` is a regular expression matching the string *alice*.

Regular expressions quickly become hard to read. Grok patterns are pre-defined regular expression snippets that you can use in your `match` patterns. For example, a complete `match` pattern for the log lines above looks like this:

```
%{DATE} %{TIME} %{USER} %{NUMBER}
```

The actual regular expression snippets referenced by `DATE`, `TIME`, `USER`, and `NUMBER` are defined in github.com/logstash-patterns-core.

Labels

One of the main features of Prometheus is its multi-dimensional data model: A Prometheus metric can be further partitioned using labels.

In order to define a label, you need two steps. First: Define a Grok field name in the `match:` pattern. Second: Add label template under `labels:` .

1. *Define Grok field names.* In Grok, each field, like `%{USER}` , can be given a name, like `%{USER:user}` . The name `user` can then be used in label templates.
2. *Define label templates.* Each metric type supports `labels` , which is a map of name/template pairs. The name will be used in Prometheus as the label name. The template is a Go template that may contain references to Grok fields, like `{{.user}}` .

Example: In order to define a label `user` for the example log lines above, use the following fragment:

```
match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
labels:
    user: '{{.user}}'
```

The `match` stores whatever matches the `%{USER}` pattern under the Grok field name `user` . The label defines a Prometheus label `user` with the value of the Grok field `user` as its content.

This simple example shows a one-to-one mapping of a Grok field to a Prometheus label. However, the label definition is pretty flexible: You can combine multiple Grok fields in one label, and you can define constant labels that don't use Grok fields at all.

Pre-Defined Label Variables

Two pre-defined label variables, that are independent of Grok patterns are defined, namely:

- `logfile` : Which contains the full path of the log file the line was read from (for input type `file`).
- `extra` : Which contains the entire JSON object parsed from the input (for input type `webhook` , with format= `json_*`).

logfile

The `logfile` variable is always present for input type `file` , and contains the full path to the log file the line was read from. You can use it like this:

```
match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
labels:
    user: '{{.user}}'
    logfile: '{{.logfile}}'
```

If you don't want the full path but only the file name, you can use the `base` template function, see next section.

extra

The `extra` variable is always present for input type `webhook` with format being either `json_single`, `json_lines` or `json_bulk`. It contains the entire JSON object that was parsed. You can use it like this:

```
match: 'Login occured'
labels:
    user: '{{ index .extra "user" }}'
    ip: '{{ index .extra "ip" }}'
```

With the incoming log object being:

```
{"message": "Login occured", "user": "Skeen", "ip": "1.1.1.1"}'
```

Label Template Functions

Label values are defined as Go templates. `grok_exporter` supports the following template functions: `gsub`, `base`, `add`, `subtract`, `multiply`, `divide`.

For example, let's assume we have the match from above:

```
match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
```

We apply this pattern to the first log line of our example:

```
30.07.2016 14:37:03 alice 1.5
```

This results in three variables that are available for defining labels:

- `user` has the value `alice`
- `val` has the value `1.5`
- `logfile` has the value `/tmp/example/example.log`, if that's where the logfile was located.

The following examples show how to use these fields as label values using the Go template language:

- `'{{.user}}' -> alice`
- `'user {{.user}} with number {{.val}}.' -> user alice with number 1.5.`
- `'{{gsub .user "ali" "beatri"}}' -> beatrice`
- `'{{multiply .val 1000}}' -> 1500`
- `'{{if eq .user "alice"}}1{{else}}0{{end}}' -> 1`
- `'{{base .logfile}}' -> example.log`
- `'{{gsub .logfile "/tmp/" ""}} -> example/example.log`

The syntax of the `gsub` function is `{{gsub input pattern replacement}}`. The pattern and replacement are is similar to Elastic's mutate filter's gsub (derived from Ruby's String.gsub()), except that you need to double-escape backslashes (\\ instead of \). A more complex example (including capture groups) can be found in this comment.

The arithmetic functions `add`, `subtract`, `multiply`, and `divide` are straightforward. These functions may not be useful for label values, but they can be useful as the `value:` in gauge, histogram, or summary metrics. For example, they could be used to convert milliseconds to seconds.

Conditionals like `'{{if eq .user "alice"}}1{{else}}0{{end}}` are described in the Go template documentation. For example, they can be used to define boolean metrics, i.e. gauge metrics with a value of `1` or `0`. Another example can be found in this comment.

The `base` function is like Golang's path.Base(). If you want something other than either the full path or the file name, use `gsub`.

Restricting a Metric to Specific Log Files

In the `input` section above, we showed that you can monitor multiple logfiles. By default, all metrics are applied to all log files. If you want to restrict a metric to specific log files, you can specify either a `path` or a list of `paths`:

```
  - type: counter
    name: alice_occurrences_total
    help: number of log lines containing alice
    match: 'alice'
    paths: /tmp/example/*.log
    labels:
        logfile: '{{base .logfile}}'
```

In the example, the `alice_occurrences_total` would only be applied to files matching `/tmp/example/*.log` and not to other files. If you have only one single path, you can use `path` as an alternative to `paths`. Note that `path` and `paths` are Glob patterns, which is not the same as Grok patterns or regular expressions.

Expiring Old Labels

By default, metrics are kept forever. However, sometimes you might want metrics with old labels to expire. There are two ways to do this in `grok_exporter`:

`delete_labels`

As of version 0.2.2, `grok_exporter` supports `delete_match` and `delete_labels` configuration:

```
  delete_match: '%{DATE} %{TIME} %{USER:user} logged out'
  delete_labels:
      user: '{{.user}}'
```

Without `delete_match` and `delete_labels`, all labels are kept forever (until `grok_exporter` is restarted). However, it might sometimes be desirable to explicitly remove metrics with specific labels. For example, if a service shuts down, it might be desirable to remove metrics labeled with that service name.

Using `delete_match` you can define a regular expression that will trigger removal of metrics. For example, `delete_match` could match a shutdown message in a log file.

Using `delete_labels` you can restrict which labels are deleted if a line matches `delete_match`. If no `delete_labels` are specified, all labels for the given metric are deleted. If `delete_labels` are specified, only those metrics are deleted where the label values are equal to the delete label values.

`retention`

As of version 0.2.3, `grok_exporter` supports `retention` configuration for metrics:

```
metrics:
    - type: ...
      name: retention_example
      help: ...
      match: ...
      labels:
          ...
      retention: 2h30m
```

The example above means that if label values for the metrics named `retention_example` have not been observed for 2 hours and 30 minutes, the `retention_example` metrics with these label values will be removed. For the format of the `retention` value, see How to Configure Durations below. Note that `grok_exporter` checks the `retention` every 53 seconds by default, so it may take 53 seconds until the metric is actually removed after the retention time is reached, see `retention_check_interval` above.

Counter Metric Type

The counter metric counts the number of matching log lines.

```
metrics:
    - type: counter
      name: grok_example_lines_total
      help: Example counter metric with labels.
      match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
      value: '{{.val}}'
      labels:
          user: '{{.user}}'
```

The configuration is as follows:

- `type` is `counter`.

- `name` is the name of the metric. Metric names are described in the [Prometheus data model documentation].
- `help` is a comment describing the metric.
- `match` is the Grok expression. See the Grok documentation for more info.
- `value` is an optional Go template for the value to be monitored. The template must evaluate to a valid positive number. The template may use to Grok fields from the `match` patterns, like the label templates described above.
- `labels` is an optional map of name/template pairs, as described above.

Output for the example log lines above:

```
# HELP grok_example_lines_total Example counter metric with labels.
# TYPE grok_example_lines_total counter
grok_example_lines_total{user="alice"} 3
grok_example_lines_total{user="bob"} 1
```

Gauge Metric Type

The gauge metric is used to monitor values that are logged with each matching log line.

```
metrics:
    - type: gauge
      name: grok_example_values
      help: Example gauge metric with labels.
      match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
      value: '{{.val}}'
      cumulative: false
      labels:
          user: '{{.user}}'
```

The configuration is as follows:

- `type` is `gauge`.
- `name`, `help`, `match`, and `labels` have the same meaning as for `counter` metrics.
- `value` is a Go template for the value to be monitored. The template must evaluate to a valid number. The template may use to Grok fields from the `match` patterns, like the label templates described above.
- `cumulative` is optional. By default, the last observed value is measured. With `cumulative: true`, the sum of all observed values is measured.

Output for the example log lines above::

```
# HELP grok_example_values Example gauge metric with labels.
# TYPE grok_example_values gauge
```

```
grok_example_values{user="alice"} 6.5
grok_example_values{user="bob"} 2.5
```

Histogram Metric Type

Like `gauge` metrics, the histogram metric monitors values that are logged with each matching log line. However, instead of just summing up the values, histograms count the observed values in configurable buckets.

```
  - type: histogram
    name: grok_example_values
    help: Example histogram metric with labels.
    match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
    value: '{{.val}}'
    buckets: [1, 2, 3]
    labels:
        user: '{{.user}}'
```

The configuration is as follows:

- `type` is `histogram`.
- `name`, `help`, `match`, `labels`, and `value` have the same meaning as for `gauge` metrics.
- `buckets` configure the categories to be observed. In the example, we have 4 buckets: One for values < 1, one for values < 2, one for values < 3, and one for all values (i.e. < infinity). Buckets

762 lines (580 sloc) | 33.9 KB

Output for the example log lines above::

```
# HELP grok_example_values Example histogram metric with labels.
# TYPE grok_example_values histogram
grok_example_values_bucket{user="alice",le="1"} 0
grok_example_values_bucket{user="alice",le="2"} 1
grok_example_values_bucket{user="alice",le="3"} 3
grok_example_values_bucket{user="alice",le="+Inf"} 3
grok_example_values_sum{user="alice"} 6.5
grok_example_values_count{user="alice"} 3
grok_example_values_bucket{user="bob",le="1"} 0
grok_example_values_bucket{user="bob",le="2"} 0
grok_example_values_bucket{user="bob",le="3"} 1
grok_example_values_bucket{user="bob",le="+Inf"} 1
grok_example_values_sum{user="bob"} 2.5
grok_example_values_count{user="bob"} 1
```

Summary Metric Type

Like `gauge` and `histogram` metrics, the summary metric monitors values that are logged with each matching log line. Summaries measure configurable φ quantiles, like the median (φ=0.5) or the 95% quantile (φ=0.95). See histograms and summaries for more info.

```
metrics:
   - type: summary
     name: grok_example_values
     help: Summary metric with labels.
     match: '%{DATE} %{TIME} %{USER:user} %{NUMBER:val}'
     value: '{{.val}}'
     quantiles: {0.5: 0.05, 0.9: 0.01, 0.99: 0.001}
     max_age: 10m
     labels:
         user: '{{.user}}'
```

The configuration is as follows:

- `type` is `summary`.
- `name`, `help`, `match`, `labels`, and `value` have the same meaning as for `gauge` metrics.
- `quantiles` is a list of quantiles to be observed. `grok_exporter` does not provide exact values for the quantiles, but only estimations. For each quantile, you also specify an uncertainty that is tolerated for the estimation. In the example, we measure the median (0.5 quantile) with uncertainty 5%, the 90% quantile with uncertainty 1%, and the 99% quantile with uncertainty 0.1%. `quantiles` is optional, the default value is `{0.5: 0.05, 0.9: 0.01, 0.99: 0.001}`.
- `max_age` is a summary sliding window. By default, summaries represent a sliding time window of 10 minutes, i.e. if you observe a 0.5 quantile (median) of *x*, the value *x* represents the median within the last 10 minutes. The time window is moved forward every 2 minutes.

Output for the example log lines above::

```
# HELP grok_example_values Example summary metric with labels.
# TYPE grok_example_values summary
grok_example_values{user="alice",quantile="0.5"} 2.5
grok_example_values{user="alice",quantile="0.9"} 2.5
grok_example_values{user="alice",quantile="0.99"} 2.5
grok_example_values_sum{user="alice"} 6.5
grok_example_values_count{user="alice"} 3
grok_example_values{user="bob",quantile="0.5"} 2.5
grok_example_values{user="bob",quantile="0.9"} 2.5
grok_example_values{user="bob",quantile="0.99"} 2.5
grok_example_values_sum{user="bob"} 2.5
grok_example_values_count{user="bob"} 1
```

Server Section

The server section configures the HTTP(S) server for exposing the metrics:

```
server:
    protocol: https
    host: localhost
    port: 9144
    path: /metrics
    cert: /path/to/cert
    key: /path/to/key
    client_ca: /path/to/client_ca
    client_auth: RequireAndVerifyClientCert
```

- `protocol` can be `http` or `https`. Default is `http`.
- `host` can be a hostname or an IP address. If host is specified, `grok_exporter` will listen on the network interface with the given address. If host is omitted, `grok_exporter` will listen on all available network interfaces. If `host` is set to `[::]`, `grok_exporter` will listen on all IPV6 addresses.
- `port` is the TCP port to be used. Default is `9144`.
- `path` is the path where the metrics are exposed. Default is `/metrics`, i.e. by default metrics will be exported on http://localhost:9144/metrics.
- `cert` is the path to the SSL certificate file for protocol `https`. It is optional. If omitted, a hard-coded default certificate will be used.
- `key` is the path to the SSL key file for protocol `https`. It is optional. If omitted, a hard-coded default key will be used.
- `client_ca` is the CA certificate used for client authentication. It is optional. If omitted, `grok_exporter` will not validate client certificates.
- `client_auth` is the policy used for client authentication. It can only be used together with `client_ca`. It is optional. The default is `RequireAndVerifyClientCert`, meaning if you specify a `client_ca`, you want to allow only clients with a valid certificate. Golang's tls.ClientAuthType documentation contains a list of valid values: `NoClientCert`, `RequestClientCert`, `RequireAnyClientCert`, `VerifyClientCertIfGiven`, and `RequireAndVerifyClientCert`.

Example commands for creating SSL test certificates:

The following will generate `server.crt` and `server.key`:

```
openssl req -x509 -sha256 -subj "/CN=localhost" -nodes -days 365 -newkey rsa:2048 -keyout
server.key -out server.crt
```

These can be configured in `grok_exporter` as follows:

```
server:
    protocol: https
    cert: ./server.crt
    key: ./server.key
```

The following will generate `client.crt` and `client.key`:

```
openssl req -x509 -sha256 -subj "/CN=localhost" -nodes -days 365 -newkey rsa:2048 -keyout
client.key -out client.crt
```

The `client.crt` can be configured in `grok_exporter` as follows:

```
server:
  protocol: https
  cert: ./server.crt
  key: ./server.key
  client_ca: ./client.crt
  client_auth: RequireAndVerifyClientCert
```

The `client.key` can be used to authenticate client requests. With `curl`, you can test this as follows:

```
curl --cacert server.crt --cert client.crt --key client.key https://localhost:9144/metrics
```

How to Configure Durations

`grok_exporter` uses the format from golang's time.ParseDuration() for configuring time intervals. Some examples are:

- `2h30m` : 2 hours and 30 minutes
- `100ms` : 100 milliseconds
- `1m30s` : 1 minute and 30 seconds
- `5m` : 5 minutes

Give feedback