metrics-clojure Documentation

Release 2.10.0

Steve Losh

Jan 25, 2024

CONTENTS

metrics-clojure is a thin Clojure façade around Coda Hale's wonderful metrics library.

metrics-clojure is a thin wrapper around metrics, so if you don't know what any of the words in this documentation mean you should probably read the metrics documentation and/or watch the talk.

- Source (Git): http://github.com/clj-commons/metrics-clojure/
- Documentation: http://metrics-clojure.rtfd.org/
- Issues: http://github.com/clj-commons/metrics-clojure/issues/
- License: MIT/X11

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

Add this to your `project.clj`'s dependencies:

`[metrics-clojure "2.10.0"]`

That's it.

1.2 Gauges

Gauges are used to measure the instantaneous value of something.

They're often useful for things that aren't events, like "users currently in the database". If you can perform a query of some kind at any time to get a value, it's probably something suitable for a gauge.

Examples of metrics you might want to track with a gauge:

- Number of users in your database.

TODO: More examples.

1.2.1 Creating

Create your gauge using `gauge-fn`:: but you have to pass it a function, not just a body:

```
(require '[metrics.core :refer [new-registry]])
(require '[metrics.gauges :refer [gauge-fn gauge]])

(def reg (new-registry))
(def files-open
  (gauge-fn reg "files-open"
        #(return-number-of-files-open ...)))
```

Once a gauge has been registered, a call to `(gauge reg "files-open")` will return the existing gauge.

You can also use the `defgauge` macro to create a gauge and bind it to a var in one concise, easy step:

```
(require '[metrics.gauges :refer [defgauge]])

(defgauge reg files-open
```

(continues on next page)

(continued from previous page)

```
  (fn []
    (return-number-of-files-open ...)))
```

`defgauge` takes a function like `gauge-fn`.

All the `def[metric]` macros do some *magic* to the metric title to make it easier to define.

1.2.2 Writing

With gauges there is no writing. Gauges execute the form(s) (or function) you passed when creating them every time they're read. You don't need to do anything else.

1.2.3 Reading

There's only one way to get data from a gauge.

value

You can read the value of a gauge at any time with `value`:

```
(require '[metrics.gauges :refer [value]])

(value files-open)
```

Or if you haven't held a reference to `files-open`, you can do the following:

```
(value (gauge reg "files-open"))
```

1.3 Counters

Counters are integer values you can increment and decrement.

They can be useful for tracking things that you don't have a particular list of, but you do control when they're added/removed/opened/closed.

For example: it's not necessarily very easy to get a list of all the open HTTP connections for a web server, but it's often easy to wrap some "middleware" that increments the "open requests" counter when a request comes in and decrements it when the response goes out.

They can also be useful for tracking simple "total" values like "total requests served in this app's entire lifetime".

Examples of metrics you might want to track with a counter:

- Number of http requests currently being processed.
- Total number of requests/responses received/sent.

TODO: More examples.

1.3.1 Creating

Create your counter:

```
(require '[metrics.core :refer [new-registry]])
(require '[metrics.counters :refer [counter]])

(def reg (new-registry))
(def users-connected (counter reg "users-connected"))
```

The `counter` function is idempotent, which means that you don't need to keep a local reference to the counter. Once a counter has been registered, a call to `(counter reg "users-connected")` will return the existing counter.

You can also use the `defcounter` macro to create a counter and bind it to a var in one concise, easy step:

```
(require '[metrics.counters :refer [defcounter]])

(defcounter reg users-connected)
```

All the `def[metric]` macros do some *magic* to the metric title to make it easier to define.

1.3.2 Writing

Once you have a counter you can increment it and decrement it.

inc!

Increment counters with `inc!`. You can pass a number to increment it by, or omit it to increment by 1:

```
(require '[metrics.counters :refer [inc!]])

(inc! users-connected)
(inc! users-connected 2)
```

Or if you haven't held a reference to `users-connected`, you can do the following:

```
(inc! (counter reg "users-connected"))
(inc! (counter reg "users-connected") 2)
```

dec!

Decrement counters with `dec!`. You can pass a number to decrement it by, or omit it to decrement by 1:

```
(require '[metrics.counters :refer [dec!]])

(dec! users-connected)
(dec! users-connected 2)
```

Or if you haven't held a reference to `users-connected`, you can do the following:

```
(dec! (counter reg "users-connected"))
(dec! (counter reg "users-connected") 2)
```

1.3.3 Reading

There's only one way to get data from a counter.

value

You can get the current value of a counter with `value`:

```
(require '[metrics.counters :refer [value]])

(value users-connected)
```

Or if you haven't held a reference to `users-connected`, you can do the following:

```
(value (counter reg "users-connected"))
```

The counter will be created and return the default value if it hasn't been registered before.

1.4 Meters

Meters are metrics that let you "mark" when an event happens and tell you how often it occurs.

Meters are used for events where the only thing you care about is "this event happened".

If you need to record a value along with this you probably want a histogram. For example: "a user performed a search" could be tracked with a meter, but "a user performed a search *and got N results*" would need a histogram.

Meters can tell you things like:

> Over the past five minutes, an average of 6,500 searches were performed each second.

Examples of metrics you might want to track with a meter:

- A user logged in.
- A POST request was received.

TODO: More examples.

1.4.1 Creating

Create your meter:

```
(require '[metrics.core :refer [new-registry]])
(require '[metrics.meters :refer [meter]])

(def reg (new-registry))
(def files-served (meter reg "files-served"))
```

The `meter` function is idempotent, which means that you don't need to keep a local reference to the meter. Once a meter has been registered, a call to `(meter reg "files-served")` will return the existing meter.

You can also use the `defmeter` macro to create a meter and bind it to a var in one concise, easy step:

```
(require '[metrics.meters :refer (defmeter)])

(defmeter reg files-served)
```

All the `def[metric]` macros do some *magic* to the metric title to make it easier to define.

1.4.2 Writing

Once you've got a meter you can mark occurrences of events.

mark!

Mark the meter every time the event happens with `mark!`:

```
(require '[metrics.meters :refer [mark!]])

(mark! files-served)
```

Or if you haven't held a reference to `files-served`, you can do the following:

```
(mark! (meter reg "files-served"))
```

1.4.3 Reading

There are a few functions you can use to retrieve the rate at which the metered events occur.

rates

You can get a map containing the mean rates of the event considering the last one, five, and fifteen minute periods with `rates`:

```
(require '[metrics.meters :refer (rates)])

(rates files-served)
=> { 1 100.0,
     5 120.0,
    15 76.0}
```

In this example the event happened approximately 100 times per second during the last one minute period, 120 times per second in the last five minute period, and 76 times per second in the last fifteen minute period.

rate-one

If you only care about the rate of events during the last minute you can use `rate-one`:

```
(require '[metrics.meters :refer [rate-one]])

(rate-one files-served)
=> 100.0
```

Or if you haven't held a reference to `files-served`, you can do the following:

```
(rate-one (meter reg "files-served"))
=> 100.0
```

rate-five

If you only care about the rate of events during the last five minutes you can use `rate-five`:

```
(require '[metrics.meters :refer [rate-five]])

(rate-five files-served)
=> 120.0
```

Or if you haven't held a reference to `files-served`, you can do the following:

```
(rate-five (meter reg "files-served"))
=> 120.0
```

rate-fifteen

If you only care about the rate of events during the last fifteen minutes you can use `rate-fifteen`:

```
(require '[metrics.meters :refer [rate-fifteen]])

(rate-fifteen files-served)
=> 76.0
```

Or if you haven't held a reference to `files-served`, you can do the following:

```
(rate-fifteen (meter reg "files-served"))
=> 76.0
```

rate-mean

If you really want the mean rate of events over the lifetime of the meter (hint: you probably don't) you can use `rate-mean`:

```
(require '[metrics.meters :refer [rate-mean]])

(rate-mean files-served)
=> 204.123
```

Or if you haven't held a reference to `files-served`, you can do the following:

```
(rate-mean (meter reg "files-served"))
=> 204.123
```

1.5 Histograms

Histograms are used to record the distribution of a piece of data over time.

They're used when you have a type of data for which the following are true:

- There are distinct "events" for this type of data, such as "user performs a search and we return N results".
- Each event has a numeric value (the "N results" in our example).
- Comparisons of these numeric values are meaningful.

For example: HTTP status codes do *not* fit this because comparisons between the numeric values are not meaningful. The fact that 404 happens to be less than 500 doesn't tell you anything.

Contrast this with something like "search results returned": one value being less than the other tells you something meaningful about the data.

Histograms can tell you things like:

> 75% of all searches returned 100 or fewer results, while 95% got 200 or fewer.

If the numeric value you're recording is the amount of time taken to do something, you probably want a timer instead of a histogram.

Examples of metrics you might want to track with a histogram:

- Search results returned ("99% of searches returned 300 or fewer results").
- Response body size ("75% of responses were 30kb or smaller").

TODO: More examples.

1.5.1 Creating

Create your histogram:

```
(require '[metrics.core :refer [new-registry]])
(require '[metrics.histograms :refer (histogram)])

(def reg (new-registry))

(def search-results-returned
  (histogram reg "search-results-returned"))
```

The `histogram` function is idempotent, which means that you don't need to keep a local reference to the histogram. Once a histogram has been registered, a call to `(histogram reg "search-results-returned")` will return the existing histogram.

You can also use the `defhistogram` macro to create a histogram and bind it to a var in one concise, easy step:

```
(require '[metrics.histograms :refer [defhistogram]])

(defhistogram reg search-results-returned)
```

All the `def[metric]` macros do some *magic* to the metric title to make it easier to define.

1.5.2 Writing

Once you've got a histogram you can update it with the numeric values of events as they occur.

update!

Update the histogram when you have a new value to record with `update!`:

```
(require '[metrics.histograms :refer [update!]])

(update! search-results-returned 10)
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(update! (histogram reg "search-results-returned") 10)
```

1.5.3 Reading

The data of a histogram metrics can be retrived in a bunch of different ways.

percentiles

The function you'll usually want to use to pull data from a histogram is `percentiles`:

```
(require '[metrics.histograms :refer [percentiles]])

(percentiles search-results-returned)
=> { 0.75   180
     0.95   299
     0.99   300
     0.999  340
     1.0   1345 }
```

This returns a map of the percentiles you probably care about. The keys are the percentiles (doubles between 0 and 1 inclusive) and the values are the maximum value for that percentile. In this example:

- 75% of searches returned 180 or fewer results.
- 95% of searches returned 299 or fewer results.
- ... etc.

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(percentiles (histogram reg "search-results-returned"))
=> { 0.75   180
     0.95   299
     0.99   300
     0.999  340
     1.0   1345 }
```

If you want a different set of percentiles just pass them as a sequence:

```
(require '[metrics.histograms :refer [percentiles]])

(percentiles search-results-returned [0.50 0.75])
=> { 0.50 100
     0.75 180 }
```

number-recorded

To get the number of data points recorded over the entire lifetime of this histogram:

```
(require '[metrics.histograms :refer [number-recorded]])

(number-recorded search-results-returned)
=> 12882
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(number-recorded (histogram reg "search-results-returned"))
=> 12882
```

smallest

To get the smallest data point recorded over the entire lifetime of this histogram:

```
(require '[metrics.histograms :refer [smallest]])

(smallest search-results-returned)
=> 4
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(smallest (histogram reg "search-results-returned"))
=> 4
```

largest

To get the largest data point recorded over the entire lifetime of this histogram:

```
(require '[metrics.histograms :refer [largest]])

(largest search-results-returned)
=> 1345
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(largest (histogram reg "search-results-returned"))
=> 1345
```

mean

To get the mean of the data points recorded over the entire lifetime of this histogram:

```
(require '[metrics.histograms :refer [mean]])

(mean search-results-returned)
=> 233.12
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(mean (histogram reg "search-results-returned"))
=> 233.12
```

std-dev

To get the standard deviation of the data points recorded over the entire lifetime of this histogram:

```
(require '[metrics.histograms :refer [std-dev]])

(std-dev search-results-returned)
=> 80.2
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(std-dev (histogram reg "search-results-returned"))
=> 80.2
```

sample

You can get the current sample points the histogram is using with `sample`, but you almost *certainly* don't care about this. If you use it make sure you know what you're doing.

```
(require '[metrics.histograms :refer [sample]])

(sample search-results-returned)
=> [12 2232 234 122]
```

Or if you haven't held a reference to `search-results-returned`, you can do the following:

```
(sample (histogram reg "search-results-returned"))
=> [12 2232 234 122]
```

1.6 Timers

Timers record the time it takes to do things. They're a bit like histograms where the value being recorded is time.

Timers should be a fairly intuitive concept. They can tell you things like:

> 75% of all searches took 0.5 seconds or less. 95% of all searches took 1.0 seconds or less.

Timers also track the rate of the timed events, so it's like they have a meter metric built-in for convenience.

1.6.1 Creating

Create your timer:

```
(require '[metrics.core :refer [new-registry]])
(require '[metrics.timers :refer [timer]])

(def image-processing-time (timer "image-processing-time"))
```

The `timer` function is idempotent, which means that you don't need to keep a local reference to the timer. Once a timer has been registered, a call to `(timer reg "image-processing-time")` will return the existing timer.

You can also use the `deftimer` macro to create a timer and bind it to a var in one concise, easy step:

```
(require '[metrics.timers :refer [deftimer]])

(deftimer image-processing-time)
```

All the `def[metric]` macros do some *magic* to the metric title to make it easier to define.

1.6.2 Writing

Once you have a timer you can record times to it in three different ways.

`time!`

You can record the time it takes to evaluate one or more expressions with the `time!` macro:

```
(require '[metrics.timers :refer [time!]])

(time! image-processing-time
       (process-image-part-1 ...)
       (process-image-part-2 ...))
```

Or if you haven't held a reference to `image-processing-time`, you can do the following:

```
(time! (timer reg "image-processing-time")
       (process-image-part-1 ...)
       (process-image-part-2 ...))
```

time-fn!

`time!` is a macro. If you need a function instead (e.g.: for `map`'ing over a list), you can use `time-fn!`, but you'll need to pass it a function instead of a body:

```
(require '[metrics.timers :refer [time-fn!]])

(time-fn! image-processing-time
          (fn []
            (process-image-part-1 ...)
            (process-image-part-2 ...)))
```

start/stop

You can also use the start and stop functions in `metrics.timers`, assuming you hang onto the Timer$Context instance that is returned.:

```
(require '[metrics.timers :as tmr])

(tmr/deftimer my-tmr)

(let [a (tmr/start my-tmr)
      b (tmr/start my-tmr)
      c (tmr/start my-tmr)]
    (Thread/sleep 1000)
    (println (tmr/stop c))
    (println (tmr/stop b))
    (println (tmr/stop a))
#_ => 1000266000 ; nanoseconds this instance ran for.
      1000726000
      1000908000
      nil
```

1.6.3 Reading

percentiles

You can use `percentiles` to find the percentage of actions that take less than or equal to a certain amount of time:

```
(require '[metrics.timers :refer (percentiles)])

(percentiles image-processing-time)
=> { 0.75  232.00
     0.95  240.23
     0.99  280.01
```

(continues on next page)

(continued from previous page)

```
    0.999 400.232
    1.0   903.1 }
```

This returns a map of the percentiles you probably care about. The keys are the percentiles (doubles between 0 and 1 inclusive) and the values are the maximum time taken for that percentile. In this example:

- 75% of images were processed in 232 nanoseconds or less
- 95% of images were processed in 240 nanoseconds or less
- … etc

If you want a different set of percentiles just pass them as a sequence:

```
(require '[metrics.timers :refer [percentiles]])

(percentiles image-processing-time [0.50 0.75])
=> { 0.50 182.11
     0.75 232.00 }
```

number-recorded

To get the number of data points recorded over the entire lifetime of this timers:

```
(require '[metrics.timers :refer [number-recorded]])

(number-recorded image-processing-time)
=> 12882
```

smallest

To get the smallest data point recorded over the entire lifetime of this timer:

```
(require '[metrics.timers :refer [smallest]])

(smallest image-processing-time)
=> 80.66
```

largest

To get the largest data point recorded over the entire lifetime of this timer:

```
(require '[metrics.timers :refer [largest]])

(largest image-processing-time)
=> 903.1
```

mean

To get the mean of the data points recorded over the entire lifetime of this timer:

```
(require '[metrics.timers :refer [mean]])

(mean image-processing-time)
=> 433.12
```

std-dev

To get the standard deviation of the data points recorded over the entire lifetime of this timer:

```
(require '[metrics.histograms :only [std-dev]])

(std-dev image-processing-time)
=> 300.51
```

sample

You can get the current sample points the timer is using with `sample`, but you almost *certainly* don't care about this. If you use it make sure you know what you're doing.

```
(require '[metrics.timers :refer [sample]])

(sample image-processing-time)
=> [803.234 102.223 ...]
```

TODO: Rates

1.7 Metric Names

In all the examples we've used a string to name the metric. The `metrics` library actually names metrics with three-part names, sometimes called "group", "type", and "metric name".

In Java and Scala those names are usually set to the package and class where the metric is being used.

In Clojure you usually won't have a meaningful class name to record, and the package name would be a pain to find, so metrics-clojure uses "default" and "default" for those parts. This results in a name like "default.default.my-metric".

If you want to specify something other than "default" you can pass a collection of three strings instead of a single string:

```
(require '[metrics.core :refer [new-registry]])
(require '[metrics.timers :refer [timer]])

(def reg (new-registry))

(def response-time
  (timer reg ["webservice" "views" "response-time"]))
```

This will result in a name like "webservice.views.response-time".

1.7.1 Title Desugaring by the `def[metric]` Macros

All of the `def[metric]` macros (*defcounter*, *defgauge*, *defhistogram*, *defmeter*, and *deftimer*) take a title as their first argument.

All of the macros will use this title as a symbol when binding it to a var, and will convert it to a string to use as the name of the metric.

For example, this:

```
(defmeter post-requests)
```

is equivalent to:

```
(def post-requests (meter reg "post-requests"))
```

If you want to define a metric with a name outside of the default group/type, you can use a vector of three strings and/or symbols instead of a bare symbol. The last entry in the vector will be used as the symbol for the var (and will be coerced if necessary). For example, all of these:

```
(defcounter reg ["mysite.http" api post-requests])
(defcounter reg ["mysite.http" "api" post-requests])
(defcounter reg ["mysite.http" "api" "post-requests"])
```

are equivalent to:

```
(def post-requests (meter reg ["mysite.http" "api" "post-requests"]))
```

If you need more control than this (e.g.: if you want a var named differently than the last segment of the metric name) you should use the normal creation methods. These macros are only a bit of syntactic sugar to reduce typing.

1.8 Removing Metrics

You can remove metrics as long as you know their names:

```
(require '[metrics.core :refer [remove-metric]])

(remove-metric "files-served")
```

You can use the sequence form of *metric names* here too, of course:

```
(remove-metric ["webservice" "views" "response-time"])
```

You can also remove a bunch of metrics by using a predicate:

```
(remove-metrics #(= "webservice" (first %)))
```

1.9 Side Effects and Agents

Pretty much everything metrics-clojure does causes side effects. If you're recording metrics inside of a `dosync` they may be recorded multiple times if the transaction is restarted.

This may or may not be what you want. If you're recording how long it takes to do something, and you do it twice, then it makes sense to record it each time. If you're recording how many responses get sent to a user, then you probably don't want to overcount them.

metrics-clojure doesn't try to decide for you. It leaves it up to you to handle the issue.

If you don't want to record something multiple times, an agent may be a good way to handle things:

```
(def thing-given-to-user (agent (counter reg "thing-given-to-user")))

(dosync
   (send thing-given-to-user inc!)
   (make-thing ...))
```

This works because the recording functions for counters, meters and histograms return the metric object.

This will *not* work with timers! `time!` returns the result of the work, so the agent's value will be replaced by that and it won't work more than once.

In practice this shouldn't be a problem though, because if you're timing work you'll probably want to time it every time it happens, right?

1.10 Reporting

Once you've started tracking some metrics there are several ways you can read their values.

More reporting methods will be added in the future. Pull requests are welcome.

1.10.1 Console

metrics-clojure supports reporting metrics through the console (on standard error):

```
(require '[metrics.reporters.console :as console])

(def CR (console/reporter reg {}))
(console/start CR 10)
```

This will tell `metrics` to print all metrics to the console every ten seconds.

Optional arguments to console/reporter are:

- :stream
- :locale
- :clock
- :rate-unit
- :duration-unit
- :filter

1.10.2 CSV Reporting

metrics-clojure supports reporting metrics into csv files (one file per metric):

```
(require '[metrics.reporters.csv :as csv])

(def CR (csv/reporter reg "/tmp/csv_reporter" {}))
(csv/start CR 1)
```

This will tell `metrics` to append the most recent value or each metric (every second), to a file named after the metric, in `/tmp/csv_reporter`. The directory name is required. The directory (and parents) will be created if they doesn't exist, it will throw an exception if it is not writable, or if the given path is not a directory.

To use this reporter, you may need to sanitize your metric names to ensure that they are valid filenames for your system.

Optional arguments to csv/reporter are:

- :locale
- :rate-unit
- :duration-unit
- :filter

1.10.3 JMX and `jvisualvm`

metrics-clojure also supports JMX reporting, since it's built into `metrics` itself.:

```
(require '[metrics.reporters.jmx :as jmx])

(def JR (jmx/reporter reg {}))
(jmx/start JR)
```

This will tell `metrics` to make all metrics available via `JMX` under `metrics` domain.

Once this is done, you can open `jvisualvm` (which you probably already have as it's included in many Java distributions), connect to a process, and view metrics in the MBeans tab.

Optional arguments to jmx/reporter are:

- :domain
- :rate-unit
- :duration-unit
- :filter

Note that there are options available to the JMX reporter that are not visible through the Clojure interface. I'm not that familiar with JMX, and didn't know how to handle things like ObjectNameFactory.

See https://github.com/dropwizard/metrics/blob/master/metrics-core/src/main/java/com/codahale/metrics/JmxReporter.java for the original code.

1.11 Aggregation, Graphing, and Historical Values

metrics-clojure is a wrapper around metrics, and metrics only stores the current value of any given metric. It doesn't keep the values at previous times around (aside from the one/five/fifteen minute windows for timers and meters).

Not storing historical data is important because it allows metrics to be used in production without worrying about memory usage growing out of control.

In practice you'll want to not only view these instantaneous values but also their values over time. For this you'll need to turn to external utilities.

You can implement a solution yourself as we'll show in the example below, or use an existing utility that metrics-clojure provides support for.

1.11.1 Sending Metrics to Graphite

Note: You must include `metrics-clojure-graphite` in your project.clj.

metrics-clojure supports aggregating metrics to graphite:

```
(require '[metrics.reporters.graphite :as graphite])
(import '[java.util.concurrent TimeUnit])
(import '[com.codahale.metrics MetricFilter])

(def GR (graphite/reporter {:host "your.graphite.host"
                            :prefix "my-api.common.prefix"
                            :rate-unit TimeUnit/SECONDS
                            :duration-unit TimeUnit/MILLISECONDS
                            :filter MetricFilter/ALL}))
(graphite/start GR 10)
```

This will tell `metrics` to aggregate all metrics to graphite every ten seconds.

Optional arguments to graphite/reporter are:

- :graphite
- :host
- :port
- :prefix
- :clock
- :rate-unit
- :duration-unit
- :filter

Note: The :graphite argument allows an existing `com.codahale.metrics.graphite.Graphite` instance to be specified. If specified, this instance will be used rather than a new instance constructed using the :host and :port arguments. This is useful if you need to supply a specially-constructed Graphite instance.

1.11.2 Sending Metrics to Ganglia

Note: You must include `metrics-clojure-ganglia` in your project.clj.

I don't have a ganglia server to test against, so while this compiles, and should work, it still needs testing.

metrics-clojure supports aggregating metrics to ganglia:

```
(require '[metrics.reporters.ganglia :as ganglia])
(import '[java.util.concurrent TimeUnit])
(import '[com.codahale.metrics MetricFilter])

(def ganglia (... your ganglia GMetric config here ...))
(def GR (ganglia/reporter ganglia
                          {:rate-unit TimeUnit/SECONDS
                           :duration-unit TimeUnit/MILLISECONDS
                           :filter MetricFilter/ALL}))
(ganglia/start GR 1)
```

This will tell `metrics` to aggregate all metrics to ganglia every minute.

Optional arguments to ganglia/reporter are:

- :rate-unit
- :duration-unit
- :filter

1.11.3 Sending Metrics to Riemann

Note: You must include `metrics-clojure-riemann` in your project.clj.

metrics-clojure supports aggregating metrics to riemann:

```
(require '[metrics.reporters.riemann :as riemann])
(import '[java.util.concurrent TimeUnit])
(import '[com.codahale.metrics MetricFilter])

(def riemann-client (riemann/make-riemann "localhost" 5555))
(def RR (riemann/reporter riemann-client
                          {:rate-unit TimeUnit/SECONDS
                           :duration-unit TimeUnit/MILLISECONDS
                           :filter MetricFilter/ALL}))
(riemann/start RR 1)
```

This will tell `metrics` to aggregate all metrics to Riemann every minute.

Optional arguments to riemann/reporter are:

- :clock - Clock instance to use for time
- :prefix - String to prefix all metrics with
- :rate-unit - TimeUnit to convert rates to
- :duration-unit - TimeUnit to convert durations to
- :filter - MetricFilter for filtering reported metrics

- :ttl - Time to live for reported metrics
- :separator - Separator between metric name components
- :host-name - Override source host name
- :tags - collection of tags to attach to event

1.11.4 Implementing a Simple Graphing Server

TODO

1.12 Extras for Ring

metrics-clojure contains some extra glue code you can use with your Ring apps.

1.12.1 Installation

The extra Ring-related functionality is in a separate `metrics-clojure-ring` library so you don't have to install it unless you want it.

To install it, add this to your `project.clj`'s dependencies:

```
[metrics-clojure-ring "2.10.0"]
```

Note: the versions of metrics-clojure and metrics-clojure-ring should always be the same.

1.12.2 Exposing Metrics as JSON

You can expose your app's metrics as JSON by using the `expose-metrics-as-json` middleware:

```
(require '[metrics.ring.expose :refer [expose-metrics-as-json]])

(def app (expose-metrics-as-json app))
```

This will add a `/metrics/` URL that will show all the metrics for the app. The trailing slash is required.

This middleware works great with Noir, too.

If you want to use a different endpoint you can pass it as a parameter:

```
(require '[metrics.ring.expose :refer [expose-metrics-as-json]])

(def app (expose-metrics-as-json app "/admin/stats/"))
```

Using compojure:

```
(def app
  (-> (routes home-routes app-routes)
      (wrap-base-url)
      (expose-metrics-as-json)))
```

WARNING: this URL will not be protected by a username or password in any way (yet), so if you have sensitive metrics you might want to think twice about using it (or protect it yourself).

JSON Format

Here's an example of the JSON format:

```
{
    "default.default.sample-metric": {
        "type": "counter",
        "value": 10
    }
}
```

The JSON is an object that maps metric names to their data.

Each metric object will have a `type` attribute. The rest of the attributes will depend on the type of metric.

TODO: Document each individual type.

1.12.3 Instrumenting a Ring App

You can add some common metrics by using the `instrument` middleware:

```
(require '[metrics.ring.instrument :refer [instrument]])

(def app (instrument app))
```

Using compojure:

```
(def app
  (-> (routes home-routes app-routes)
      (wrap-base-url)
      (instrument)))
```

This will add a number of metrics, listed below.

This middleware works great with Noir, too.

ring.requests.active

A counter tracking the number of currently open requests.

ring.requests.rate

A meter measuring the rate of all incoming requests.

ring.requests.rate.*

Six separate meters (ending in `GET`, `POST`, etc) measuring the rate of incoming requests of a given type.

ring.requests-scheme.rate.*

Two separate meters (ending in `http` or `https`) measuring the rate of incoming requests of a given type.

ring.responses.rate

A meter measuring the rate of all outgoing responses.

ring.responses.rate.*

Four separate meters (ending in `2xx`, `3xx`, etc) measuring the rate of outgoing responses of a given type.

ring.handling-time.*

Six separate timers (ending in `GET`, `POST`, etc) measuring the time taken to handle incoming requests of a given type.

1.12.4 Dynamic Instrumentation

instrument-by allows more dynamic collection of metrics at the cost of slightly more complex API. It accepts a *metrics-prefix* function which takes a request and returns a prefix (as a vector of strings). All the metrics described above will be collected for each prefix returned by this function.

For example, if you use *instrument-by* with the *uri-prefix* function, you'll end up with a set of metrics for each distinct URI (not including query parameters) handled by your app.

This is useful if you want to break your metrics down by endpoint. The example from above adapted to collect metrics per endpoint would be as follows:

```
(def app
  (-> (routes home-routes app-routes)
      (wrap-base-url)
      (instrument-by uri-prefix)))
```

1.12.5 Troubleshooting

If you're using these extras alongside Noir you'll need to be running the latest version of Noir to avoid dependency issues.

If you're getting errors about overriding stuff in Jackson this is the problem.

1.13 Extras for JVM

metrics-clojure contains some functions for instrumenting JVM metrics

1.13.1 Installation

The extra JVM-related functionality is in a separate `metrics-clojure-jvm` library so its installation is optional.

To install it, add this to your `project.clj`'s dependencies:

```
[metrics-clojure-jvm "2.10.0"]
```

1.13.2 Instrumenting the JVM

The simplest way to add JVM metrics to your application is to simply call the `instrument-jvm` function in your code:

```
(require '[metrics.jvm.core :refer [instrument-jvm]])

(instrument-jvm metric-registry)
```

This will add a number of metrics, listed below.

jvm.attribute

A set of gauges for the JVM name, vendor and uptime.

jvm.memory

A set of gauges for JVM memory usage, include stats on heap vs non-heap memory, plus GC-specific memory pools.

jvm.file

A gauge for the ratio of used to total file descriptors.

jvm.gc

A set of gauges for the counts and elapsed times of garbage collection.

jvm.thread

A set of gauges for the number of threads in their various states and deadlock detection.

If you want to add the individual gauge/metric sets that codahale's metrics-jvm library provides, then the following functions are available:

```
(require '[metrics.jvm.core :as jvm])

(register-jvm-attribute-gauge-set metric-registry)

(register-memory-usage-gauge-set metric-registry)

(register-file-descriptor-ratio-gauge-set metric-registry)

(register-garbage-collector-metric-set metric-registry)

(register-thread-state-gauge-set metric-registry)
```

These functions also take an optional second argument should you wish to override the metric prefix, e.g.:

```
(register-jvm-attribute-gauge-set metric-registry ["my" "preferred" "prefix"])
```

1.14 HealthChecks

`metrics-clojure-health` will allow you to define and run healthcheck.

1.14.1 Install

`metrics-clojure-health` is provided as a separate package from `metrics.core`. To install add the following to your `project.clj`:

```
[metrics-clojure-health "2.10.0"]
```

1.14.2 Example

An example using the default HealthCheckRegistry, (which is different from the default-registry).:

```
(require '[metrics.health.core :as health])
(defhealthcheck "second-check" (fn [] (let [now (.getSeconds (java.util.Date.))]
                                        (if (< now 30)
                                           (health/healthy "%d is less than 30!" now)
                                           (health/unhealthy "%d is more than 30!"␣
↪now)))))

(health/check second-check)
#_ => #<Result Result{isHealthy=true, message=3 is less than 30!}>

(health/check-all)
#_ => {:healthy [#<UnmodifiableEntry foo=Result{isHealthy=true, message=6 is less than␣
↪30!}>]}
```

1.15 Contributing

Want to contribute? Cool. Send a pull request.

Run the test cases first with `lein test` to make sure you're not crazy.

Fixing a bug? Add a test case in `test/`.

Adding a feature? Add some documentation in `docs/`.

1.16 Changelog

As of version `1.0.0` metrics-clojure is stable. New features will still be added, but backwards-incompatible changes should be very rare.

We use semantic versioning to decide how to number versions, so you know what's going to happen when you upgrade.

1.16.1 2.3.0

- Updated graphite reporter api to bring it in line with the console reporter.
- Updated docs for console reporter and graphite reporter.

1.16.2 2.0.0

Metrics Registries

metrics-clojure 1.x maintained a metrics registry in a dynamic var. This approach makes the library a little easier for beginners but also much harder to use in more sophisticated cases, e.g. in concurrent applications or those that use a Component-like approach to program structure.

As such, *metrics-clojure* 2.0+ makes metrics registry a required explicit argument to most functions in the API:

```
(require '[metrics.meters :as meters])

;; with 1.x
(meters/rate-mean)

(meters/mark! 10)
;; with 2.0
(let [m (meters/meter ["test" "meters" "test-rate-mean-update-multiple"])]
  (meters/rate-mean m)
  (meters/mark! m 10))
```

The library maintains a default registry in *metrics.core/default-registry* which tries to keep the *1.x* API as functional as possible but using your own registry is encouraged.

To instantiate a registry, use *metrics.core/new-registry*:

```
(require '[metrics.core :as mtr])

(mtr/new-registry)
```

See GH #19 for discussion.

defgauge Restricted to Functions Only

In *metrics-clojure* 1.x, *metrics.gauges/defgauge* could accept a function or a bunch of forms (body). In 2.0, it only accepts a function. This is in part due to the new API structure but also make the API more straightforward and works much better with explicit registry management now advocated by the library.

Nanoseconds Precision in Timers

Metrics 3.0 uses nanoseconds precision in timers.

Upgrade to Metrics 3.0

Metrics 3.0 is now used internally by the library.

Clojure 1.3 No Longer Supported

Clojure 1.3 is no longer supported by the library.

1.16.3 1.0.1

Fixes compilation of `defhistogram`, `defcounter` and friends.

1.16.4 1.0.0

Initial stable release.