Striemann Documentation

Release 0.5

Made.com

Feb 11, 2019

Contents:

1 API **1**
- 1.1 striemann.metrics module . . . 1
- 1.2 striemann.test.fakes module . . . 15
- 1.3 striemann.test.expectsmatcher module . . . 16

2 How to contribute **17**
- 2.1 Run the tests . . . 17
- 2.2 Use Black for formatting . . . 17

3 Installation **19**

4 Basic Usage **21**

5 0.6 - 2018-05-31 **23**
- 5.1 Deprecated . . . 23

6 0.5 - 2018-04-27 **25**
- 6.1 Breaking changes . . . 25

7 0.4 - 2017-11-14 **27**
- 7.1 Fixes . . . 27
- 7.2 Deprecated . . . 27

8 [0.3.1] **29**
- 8.1 Fixes . . . 29

9 [0.3] **31**
- 9.1 Fixes . . . 31

10 [0.2] **33**
- 10.1 Fixes . . . 33

11 Indices and tables **35**

Python Module Index **37**

CHAPTER 1

API

1.1 striemann.metrics module

This is the main module for the package.

class `striemann.metrics.`**`Metrics`**(*transport*, *source=None*)

Buffers metrics and forwards them to a `Transport`

Parameters

- **`transport`** (*`Transport`*) – The transport used to send metrics.
- **`source`** (*`Optional[str]`*) – If provided, this value will be added to all outbound metrics as the *source* attribute. The value may still be overridden on a per-metric basis.

Examples

```
>>> from striemann.metrics import InMemoryTransport, Metrics
>>> import pprint
>>>
>>> pp = pprint.PrettyPrinter(indent=4)
>>>
>>> transport = InMemoryTransport()
>>> metrics = Metrics(transport)
>>>
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.flush()
>>> print(transport.last_batch)
[{'tags': [], 'attributes': {}, 'service': 'metrics written', 'metric_f': 1}]
```

`recordGauge`(*service_name*, *value*, *ttl=None*, *tags=None*, ***kwargs*)

Record a single scalar value, eg. Queue Depth, Current Uptime, Disk Free

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

In the simplest case, we just want to record a single number. Each time we record a Gauge, we replace the previous value.

```
>>> metrics.recordGauge("Customers in restaurant", 10)
>>> metrics.recordGauge("Customers in restaurant", 8)
>>> metrics.flush()
>>> print(transport.last_batch)
[{'tags': [], 'attributes': {}, 'service': 'Customers in restaurant', 'metric_
↪f': 8}]
```

We might want to segregate our metrics by some other attribute so so that we can aggregate and drill-down.

```
>>> metrics.recordGauge("Customers in restaurant", 6, hair="brown")
>>> metrics.recordGauge("Customers in restaurant", 2, hair="blonde")
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'hair': 'blonde'},
        'metric_f': 2,
        'service': 'Customers in restaurant',
        'tags': []},
    {   'attributes': {'hair': 'brown'},
        'metric_f': 6,
        'service': 'Customers in restaurant',
        'tags': []}]
```

incrementCounter(*service_name*, *value=1*, *ttl=None*, *tags=None*, ***kwargs*)

Record an increase in a value, eg. Cache Hits, Files Written

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Counters are useful when we don't know an absolute value, but we want to record that something happened.

```
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.incrementCounter("Burgers sold", value=2)
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[{'attributes': {}, 'metric_f': 4, 'service': 'Burgers sold', 'tags': []}]
```

Counters reset after each flush.

```
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[{'attributes': {}, 'metric_f': 1, 'service': 'Burgers sold', 'tags': []}]
```

Counters can have tags and attributes associated with them. Each unique set of tags and attributes is flushed as a separate metric.

```
>>> metrics.incrementCounter("Burgers sold", tags=["drive-thru"], name=
↪"cheeseburger")
>>> metrics.incrementCounter("Burgers sold", name="cheeseburger")
>>> metrics.incrementCounter("Burgers sold", value=2, name="whopper")
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'name': 'cheeseburger'},
        'metric_f': 1,
        'service': 'Burgers sold',
        'tags': ['drive-thru']},
    {   'attributes': {'name': 'cheeseburger'},
        'metric_f': 1,
        'service': 'Burgers sold',
        'tags': []},
    {   'attributes': {'name': 'whopper'},
        'metric_f': 2,
        'service': 'Burgers sold',
        'tags': []}]
```

decrementCounter(*service_name*, *value=1*, *ttl=None*, *tags=None*, ***kwargs*)

Record an decrease in a value, eg. Cache Hits, Files Written

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Occasionally, we want to record a decrease in a value.

```
>>> metrics.incrementCounter("Customers waiting")
>>> metrics.incrementCounter("Customers waiting")
>>>
```

(continues on next page)

(continued from previous page)

```
>>> metrics.decrementCounter("Customers waiting")
>>>
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[{'attributes': {}, 'metric_f': 1, 'service': 'Customers waiting', 'tags': []}
↪]
```

recordRange (*service_name*, *value*, *ttl=None*, *tags=[]*, ***kwargs*)

Record statistics about a range of values

Ranges are useful when we care about a metric in aggregate rather than recording each individual event. When flushed, a Range sends the minimum, maximum, and mean of each recorded metric, and the count of values recorded.

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Ranges are useful when we want to know the distribution of a value. They're used by the *time()* method internally.

```
>>> metrics.recordRange("Customer height", 163)
>>> metrics.recordRange("Customer height", 185)
>>> metrics.recordRange("Customer height", 134)
>>> metrics.recordRange("Customer height", 158)
>>> metrics.recordRange("Customer height", 170)
>>>
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {},
        'metric_f': -185,
        'service': 'Customer height.min',
        'tags': []},
    {   'attributes': {},
        'metric_f': -134,
        'service': 'Customer height.max',
        'tags': []},
    {   'attributes': {},
        'metric_f': -162.0,
        'service': 'Customer height.mean',
        'tags': []},
    {   'attributes': {},
        'metric_f': 5,
        'service': 'Customer height.count',
        'tags': []}]
```

Ranges respect tags an attributes when aggregating their values.

```
>>> metrics.recordRange("Customer height", 163, sex='female')
>>> metrics.recordRange("Customer height", 185, sex='male')
>>> metrics.recordRange("Customer height", 134, tags='child', sex='male')
>>> metrics.recordRange("Customer height", 158, sex='female')
>>> metrics.recordRange("Customer height", 170, sex='male')
>>>
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'sex': 'female'},
        'metric_f': 158,
        'service': 'Customer height.min',
        'tags': []},
    {   'attributes': {'sex': 'female'},
        'metric_f': 163,
        'service': 'Customer height.max',
        'tags': []},
    {   'attributes': {'sex': 'female'},
        'metric_f': 160.5,
        'service': 'Customer height.mean',
        'tags': []},
    {   'attributes': {'sex': 'female'},
        'metric_f': 2,
        'service': 'Customer height.count',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 170,
        'service': 'Customer height.min',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 185,
        'service': 'Customer height.max',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 177.5,
        'service': 'Customer height.mean',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 2,
        'service': 'Customer height.count',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 134,
        'service': 'Customer height.min',
        'tags': 'child'},
    {   'attributes': {'sex': 'male'},
        'metric_f': 134,
        'service': 'Customer height.max',
        'tags': 'child'},
    {   'attributes': {'sex': 'male'},
        'metric_f': 134.0,
        'service': 'Customer height.mean',
        'tags': 'child'},
    {   'attributes': {'sex': 'male'},
        'metric_f': 1,
        'service': 'Customer height.count',
        'tags': 'child'}]
```

time(*service_name*, *ttl=None*, *tags=None*, ***kwargs*)

Record the time taken for an operation.

The time method returns a context manager that can be used for timing an operation. The timer uses the *default timer<https://docs.python.org/2/library/timeit.html#timeit.default_timer>_* for the operating system.

Under the hood, the time method uses a `Range` to record its values.

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Since timers use a `striemann.metrics.Range` to record their values they send a summary of all the values recorded since the last flush.

```
>>> import time
>>> with metrics.time("Burger Cooking Time"):
>>>    time.sleep(1)
>>> with metrics.time("Burger Cooking Time"):
>>>    time.sleep(5)
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {},
        'metric_f': 1.0011436779996075,
        'service': 'Burger cooking time.min',
        'tags': []},
    {   'attributes': {},
        'metric_f': 5.00513941600002,
        'service': 'Burger cooking time.max',
        'tags': []},
    {   'attributes': {},
        'metric_f': 3.0031415469998137,
        'service': 'Burger cooking time.mean',
        'tags': []},
    {   'attributes': {},
        'metric_f': 2,
        'service': 'Burger cooking time.count',
        'tags': []}]
```

Timers respect tags and attributes when aggregating.

```
>>> with metrics.time("Burger Cooking Time", type="whopper"):
>>>    time.sleep(1)
>>> with metrics.time("Burger Cooking Time", type="cheeseburger"):
>>>    time.sleep(5)
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'type': 'whopper'},
        'metric_f': 1.001190301999486,
```

(continues on next page)

(continued from previous page)

```
        'service': 'Burger cooking time.min',
        'tags': []},
    {   'attributes': {'type': 'whopper'},
        'metric_f': 1.001190301999486,
        'service': 'Burger cooking time.max',
        'tags': []},
    {   'attributes': {'type': 'whopper'},
        'metric_f': 1.001190301999486,
        'service': 'Burger cooking time.mean',
        'tags': []},
    {   'attributes': {'type': 'whopper'},
        'metric_f': 1,
        'service': 'Burger cooking time.count',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 5.005140869999195,
        'service': 'Burger cooking time.min',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 5.005140869999195,
        'service': 'Burger cooking time.max',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 5.005140869999195,
        'service': 'Burger cooking time.mean',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 1,
        'service': 'Burger cooking time.count',
        'tags': []}]
```

class `striemann.metrics.`**`MetricId`**

Bases: `tuple`

Used to uniquely identify a metric

`attributes`

(Dict[str, Any]) – Arbitrary key-value pairs associated with the metric

`name`

(str) – The 'service' of the metric.

`tags`

(List[str]) – A list of string tags associated with the metric.

class `striemann.metrics.`**`Recorder`**

Bases: `object`

Collects metrics and flushes them to a transport

`send`(*metric*, *value*, *transport*, *suffix=''*)

class `striemann.metrics.`**`LogTransport`**

Bases: `striemann.metrics.Transport`

Simple Transport that sprints metrics to the log. Useful for development environments

`flush`(*is_closing*)

Send all buffered metrics

Parameters **is_closing** (*bool*) – True if the transport should tear down any resources, eg. Sockets or file handles.

send_event(*event*)
Buffer a single event for sending.

class striemann.metrics.**InMemoryTransport**
Bases: striemann.metrics.Transport

Dummy transport that keeps a copy of the last flushed batch of events. This is used to store the data for the stats endpoints.

flush(*is_closing*)
Send all buffered metrics

Parameters **is_closing** (*bool*) – True if the transport should tear down any resources, eg. Sockets or file handles.

send_event(*event*)
Buffer a single event for sending.

class striemann.metrics.**RiemannTransport**(*host='localhost', port='5555', timeout=5*)
Bases: striemann.metrics.Transport

Transport that sends metrics to a Riemann server.

flush(*is_closing*)
Send all buffered metrics

Parameters **is_closing** (*bool*) – True if the transport should tear down any resources, eg. Sockets or file handles.

is_connected()
Check whether the transport is connected.

send_event(*event*)
Buffer a single event for sending.

class striemann.metrics.**CompositeTransport**(**args*)
Bases: striemann.metrics.Transport

Transport that wraps two or more transports and forwards events to all of them.

flush(*is_closing*)
Send all buffered metrics

Parameters **is_closing** (*bool*) – True if the transport should tear down any resources, eg. Sockets or file handles.

send_event(*event*)
Buffer a single event for sending.

class striemann.metrics.**Gauge**(*source*)
Bases: *striemann.metrics.Recorder*

Gauges record scalar values at a single point in time, eg. queue size, active sessions, and forward only the latest value.

flush(*transport*)

record(*service_name, value, ttl, tags, attributes*)

class striemann.metrics.**Range**(*source*)
Bases: *striemann.metrics.Recorder*

Summarys record the range of a value across a set of datapoints, eg response time, items cleared from cache, and forward aggregated metrics to describe that range.

flush(*transport*)

record(*service_name*, *value*, *ttl=None*, *tags=[]*, *attributes={}*)

class `striemann.metrics.`**Counter**(*source*)
Bases: *striemann.metrics.Recorder*

Counters record incrementing or decrementing values, eg. Events Processed, error count, cache hits.

flush(*transport*)

record(*service_name*, *value*, *ttl*, *tags*, *attributes*)

class `striemann.metrics.`**Timer**(*service_name*, *ttl*, *tags*, *attributes*, *histogram*)
Bases: `object`

Timers provide a context manager that times an operation and records a gauge with the elapsed time.

class `striemann.metrics.`**Metrics**(*transport*, *source=None*)
Bases: `object`

Buffers metrics and forwards them to a `Transport`

Parameters

- **transport** (*Transport*) – The transport used to send metrics.
- **source** (*Optional[str]*) – If provided, this value will be added to all outbound metrics as the *source* attribute. The value may still be overridden on a per-metric basis.

Examples

```
>>> from striemann.metrics import InMemoryTransport, Metrics
>>> import pprint
>>>
>>> pp = pprint.PrettyPrinter(indent=4)
>>>
>>> transport = InMemoryTransport()
>>> metrics = Metrics(transport)
>>>
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.flush()
>>> print(transport.last_batch)
[{'tags': [], 'attributes': {}, 'service': 'metrics written', 'metric_f': 1}]
```

decrementCounter(*service_name*, *value=1*, *ttl=None*, *tags=None*, ***kwargs*)
Record an decrease in a value, eg. Cache Hits, Files Written

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Occasionally, we want to record a decrease in a value.

```
>>> metrics.incrementCounter("Customers waiting")
>>> metrics.incrementCounter("Customers waiting")
>>>
>>> metrics.decrementCounter("Customers waiting")
>>>
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[{'attributes': {}, 'metric_f': 1, 'service': 'Customers waiting', 'tags': []}
↪]
```

flush (*is_closing=False*)

Flush all metrics to the underlying transport.

Parameters **is_closing** (*bool*) – True if the transport should be shut down.

incrementCounter (*service_name*, *value=1*, *ttl=None*, *tags=None*, ***kwargs*)

Record an increase in a value, eg. Cache Hits, Files Written

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Counters are useful when we don't know an absolute value, but we want to record that something happened.

```
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.incrementCounter("Burgers sold", value=2)
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[{'attributes': {}, 'metric_f': 4, 'service': 'Burgers sold', 'tags': []}]
```

Counters reset after each flush.

```
>>> metrics.incrementCounter("Burgers sold")
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[{'attributes': {}, 'metric_f': 1, 'service': 'Burgers sold', 'tags': []}]
```

Counters can have tags and attributes associated with them. Each unique set of tags and attributes is flushed as a separate metric.

```
>>> metrics.incrementCounter("Burgers sold", tags=["drive-thru"], name=
↪"cheeseburger")
>>> metrics.incrementCounter("Burgers sold", name="cheeseburger")
```

(continues on next page)

(continued from previous page)

```
>>> metrics.incrementCounter("Burgers sold", value=2, name="whopper")
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'name': 'cheeseburger'},
        'metric_f': 1,
        'service': 'Burgers sold',
        'tags': ['drive-thru']},
    {   'attributes': {'name': 'cheeseburger'},
        'metric_f': 1,
        'service': 'Burgers sold',
        'tags': []},
    {   'attributes': {'name': 'whopper'},
        'metric_f': 2,
        'service': 'Burgers sold',
        'tags': []}]
```

recordGauge(*service_name*, *value*, *ttl=None*, *tags=None*, ***kwargs*)

Record a single scalar value, eg. Queue Depth, Current Uptime, Disk Free

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

In the simplest case, we just want to record a single number. Each time we record a Gauge, we replace the previous value.

```
>>> metrics.recordGauge("Customers in restaurant", 10)
>>> metrics.recordGauge("Customers in restaurant", 8)
>>> metrics.flush()
>>> print(transport.last_batch)
[{'tags': [], 'attributes': {}, 'service': 'Customers in restaurant', 'metric_
↪f': 8}]
```

We might want to segregate our metrics by some other attribute so so that we can aggregate and drill-down.

```
>>> metrics.recordGauge("Customers in restaurant", 6, hair="brown")
>>> metrics.recordGauge("Customers in restaurant", 2, hair="blonde")
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'hair': 'blonde'},
        'metric_f': 2,
        'service': 'Customers in restaurant',
        'tags': []},
    {   'attributes': {'hair': 'brown'},
        'metric_f': 6,
        'service': 'Customers in restaurant',
        'tags': []}]
```

recordRange(*service_name*, *value*, *ttl=None*, *tags=[]*, ***kwargs*)

Record statistics about a range of values

Ranges are useful when we care about a metric in aggregate rather than recording each individual event. When flushed, a Range sends the minimum, maximum, and mean of each recorded metric, and the count of values recorded.

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Ranges are useful when we want to know the distribution of a value. They're used by the `time()` method internally.

```
>>> metrics.recordRange("Customer height", 163)
>>> metrics.recordRange("Customer height", 185)
>>> metrics.recordRange("Customer height", 134)
>>> metrics.recordRange("Customer height", 158)
>>> metrics.recordRange("Customer height", 170)
>>>
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {},
        'metric_f': -185,
        'service': 'Customer height.min',
        'tags': []},
    {   'attributes': {},
        'metric_f': -134,
        'service': 'Customer height.max',
        'tags': []},
    {   'attributes': {},
        'metric_f': -162.0,
        'service': 'Customer height.mean',
        'tags': []},
    {   'attributes': {},
        'metric_f': 5,
        'service': 'Customer height.count',
        'tags': []}]
```

Ranges respect tags an attributes when aggregating their values.

```
>>> metrics.recordRange("Customer height", 163, sex='female')
>>> metrics.recordRange("Customer height", 185, sex='male')
>>> metrics.recordRange("Customer height", 134, tags='child', sex='male')
>>> metrics.recordRange("Customer height", 158, sex='female')
>>> metrics.recordRange("Customer height", 170, sex='male')
>>>
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
```

(continues on next page)

(continued from previous page)

```
[   {   'attributes': {'sex': 'female'},
        'metric_f': 158,
        'service': 'Customer height.min',
        'tags': []},
    {   'attributes': {'sex': 'female'},
        'metric_f': 163,
        'service': 'Customer height.max',
        'tags': []},
    {   'attributes': {'sex': 'female'},
        'metric_f': 160.5,
        'service': 'Customer height.mean',
        'tags': []},
    {   'attributes': {'sex': 'female'},
        'metric_f': 2,
        'service': 'Customer height.count',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 170,
        'service': 'Customer height.min',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 185,
        'service': 'Customer height.max',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 177.5,
        'service': 'Customer height.mean',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 2,
        'service': 'Customer height.count',
        'tags': []},
    {   'attributes': {'sex': 'male'},
        'metric_f': 134,
        'service': 'Customer height.min',
        'tags': 'child'},
    {   'attributes': {'sex': 'male'},
        'metric_f': 134,
        'service': 'Customer height.max',
        'tags': 'child'},
    {   'attributes': {'sex': 'male'},
        'metric_f': 134.0,
        'service': 'Customer height.mean',
        'tags': 'child'},
    {   'attributes': {'sex': 'male'},
        'metric_f': 1,
        'service': 'Customer height.count',
        'tags': 'child'}]
```

time(*service_name*, *ttl=None*, *tags=None*, ***kwargs*)

Record the time taken for an operation.

The time method returns a context manager that can be used for timing an operation. The timer uses the *default timer<https://docs.python.org/2/library/timeit.html#timeit.default_timer>_* for the operating system.

Under the hood, the time method uses a `Range` to record its values.

Parameters

- **service_name** (*str*) – The name of the recorded metric.
- **value** (*SupportsFloat*) – The numeric value to record.
- **ttl** (*Optional[int]*) – An optional time-to-live for the metric, measured in seconds.
- **tags** (*Optional[List[str]]*) – A list of strings to associate with the metric.
- ****kwargs** (*Any*) – Additional key-value pairs to associate with the metric.

Examples

Since timers use a *striemann.metrics.Range* to record their values they send a summary of all the values recorded since the last flush.

```
>>> import time
>>> with metrics.time("Burger Cooking Time"):
>>>    time.sleep(1)
>>> with metrics.time("Burger Cooking Time"):
>>>    time.sleep(5)
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {},
        'metric_f': 1.0011436779996075,
        'service': 'Burger cooking time.min',
        'tags': []},
    {   'attributes': {},
        'metric_f': 5.00513941600002,
        'service': 'Burger cooking time.max',
        'tags': []},
    {   'attributes': {},
        'metric_f': 3.0031415469998137,
        'service': 'Burger cooking time.mean',
        'tags': []},
    {   'attributes': {},
        'metric_f': 2,
        'service': 'Burger cooking time.count',
        'tags': []}]
```

Timers respect tags and attributes when aggregating.

```
>>> with metrics.time("Burger Cooking Time", type="whopper"):
>>>    time.sleep(1)
>>> with metrics.time("Burger Cooking Time", type="cheeseburger"):
>>>    time.sleep(5)
>>> metrics.flush()
>>> pp.pprint(transport.last_batch)
[   {   'attributes': {'type': 'whopper'},
        'metric_f': 1.001190301999486,
        'service': 'Burger cooking time.min',
        'tags': []},
    {   'attributes': {'type': 'whopper'},
        'metric_f': 1.001190301999486,
        'service': 'Burger cooking time.max',
        'tags': []},
    {   'attributes': {'type': 'whopper'},
```

(continues on next page)

(continued from previous page)

```
        'metric_f': 1.001190301999486,
        'service': 'Burger cooking time.mean',
        'tags': []},
    {   'attributes': {'type': 'whopper'},
        'metric_f': 1,
        'service': 'Burger cooking time.count',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 5.005140869999195,
        'service': 'Burger cooking time.min',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 5.005140869999195,
        'service': 'Burger cooking time.max',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 5.005140869999195,
        'service': 'Burger cooking time.mean',
        'tags': []},
    {   'attributes': {'type': 'cheeseburger'},
        'metric_f': 1,
        'service': 'Burger cooking time.count',
        'tags': []}]
```

1.2 striemann.test.fakes module

In-memory fake replacements for objects in striemann.metrics.

For use in tests.

`striemann.test.fakes.`**`metric_id`**(*service_name*, *tags=None*, *fields=None*)

Helper function for creating instances of `MetricId`

class `striemann.test.fakes.`**`FakeTimer`**(*service_name*, *tags*, *attributes*, *metrics*)

Fake implementation of the context manager `time()`

class `striemann.test.fakes.`**`FakeMetrics`**

Fake implementation of `Metrics`

Examples

```
>>> import expects
>>> from striemann.test.expectsmatcher import contain_metric
>>> metrics = FakeMetrics()
>>> metrics.incrementCounter('Burgers sold')
>>>
>>> assert (metric_id('Burgers sold'), 1) in metrics
```

```
>>> metrics.recordGauge('Hunger level', 10)
>>> expect(metrics).to(contain_metric('Hunger level'))
```

`flush`(*is_closing=False*)

`incrementCounter`(*servicename*, *value=1*, *tags=[]*, ***kwargs*)

metrics()

recordGauge(*service_name*, *value*, *tags=[]*, ***kwargs*)

recordRange(*service_name*, *value*, *tags=[]*, ***kwargs*)

time(*service_name*, *tags=[]*, ***kwargs*)

1.3 striemann.test.expectsmatcher module

An expects matcher for asserting metrics are recorded.

https://pypi.python.org/pypi/expects

class striemann.test.expectsmatcher.**contain_metric**(*service_name*, *tags=[]*, *value=None*, ***kwargs*)

CHAPTER 2

How to contribute

Striemann is licensed under the MIT license welcomes contributors.

2.1 Run the tests

A `tox.ini` file is provided to run the tests with different versions of Python.

To run the tests:

1. Install tox
2. Run `tox` from the root folder of the repository

2.2 Use Black for formatting

Striemann uses the Black formatter. Pull requests that are not correctly formatted will be rejected. Striemann provides a developer friendly interface for sending metrics to the Riemann monitoring system. It's heavily inspired by statsd. It aims to provide a strongly opinionated way for developers to record metrics from their applications.

CHAPTER 3

Installation

Striemann is available on the cheese shop.

```
pip install striemann
```

Documentation is available on Read the Docs

CHAPTER 4

Basic Usage

```
from striemann import RiemannTransport, Metrics

# Transports are responsible for sending metrics to an endpoint
transport = RiemannTransport("localhost", 5555)

# the Metrics class is the entrypoint for the library
metrics = Metrics(transport)

# Counters keep track of how often a thing happens.
# They send the sum of their metrics when flushed.
metrics.incrementCounter("Burgers sold")
metrics.incrementCounter("Burgers sold")

metrics.incrementCounter("Days without an incident", value=2)
metrics.decrementCounter("Days without an incident", value=2)

# Gauges track a single value. They send the most recent value
# when flushed.
metrics.recordGauge("Awesomeness", value=10)
metrics.recordGauge("Awesomeness", value=100)

# Timers record how long it takes a thing to happen.
# They send the min, max, mean, and count of their recorded values when flushed
with metrics.time("Do a slow thing"):
    time.sleep(5)


# periodically you should flush metrics
metrics.flush()
```

CHAPTER 5

0.6 - 2018-05-31

Added documentation on Read the Docs.

5.1 Deprecated

- FakeMetrics is now a list. Fakemetrics.metrics will be removed in 1.0 release.

CHAPTER 6

0.5 - 2018-04-27

6.1 Breaking changes

- Gauges no longer record min/mean/max/count Once we actually started using the library in anger, it became apparent that for most gauges, the min/max/mean params aren't helpful.

  We've decided to drop that feature from `recordGauge` replacing it with a new `recordSummary` method. The `time` method has been rewritten to use a Summary rather than a Gauge.
- Made the "counters", and "gauges" properties of the Metrics class private.
- Made the state of Counter, and Gauge private.

CHAPTER 7

0.4 - 2017-11-14

7.1 Fixes

- Fix issue where we get stuck always 'Failed to flush metrics to riemann'

7.2 Deprecated

- RiemannTransport.is_connected() should no longer be needed

CHAPTER 8

[0.3.1]

8.1 Fixes

- We now reconnect if there is an exception raised during `flush`

CHAPTER 9

[0.3]

9.1 Fixes

- Added missing TTL parameter to `time` method.

CHAPTER 10

[0.2]

10.1 Fixes

- TTL attributes are no longer coerced to strings

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

striemann.metrics, 7
striemann.test.expectsmatcher, 16
striemann.test.fakes, 15

Index

A

attributes (striemann.metrics.MetricId attribute), 7

C

CompositeTransport (class in striemann.metrics), 8
contain_metric (class in striemann.test.expectsmatcher), 16
Counter (class in striemann.metrics), 9

D

decrementCounter() (striemann.metrics.Metrics method), 3, 9

F

FakeMetrics (class in striemann.test.fakes), 15
FakeTimer (class in striemann.test.fakes), 15
flush() (striemann.metrics.CompositeTransport method), 8
flush() (striemann.metrics.Counter method), 9
flush() (striemann.metrics.Gauge method), 8
flush() (striemann.metrics.InMemoryTransport method), 8
flush() (striemann.metrics.LogTransport method), 7
flush() (striemann.metrics.Metrics method), 10
flush() (striemann.metrics.Range method), 9
flush() (striemann.metrics.RiemannTransport method), 8
flush() (striemann.test.fakes.FakeMetrics method), 15

G

Gauge (class in striemann.metrics), 8

I

incrementCounter() (striemann.metrics.Metrics method), 2, 10
incrementCounter() (striemann.test.fakes.FakeMetrics method), 15
InMemoryTransport (class in striemann.metrics), 8
is_connected() (striemann.metrics.RiemannTransport method), 8

L

LogTransport (class in striemann.metrics), 7

M

metric_id() (in module striemann.test.fakes), 15
MetricId (class in striemann.metrics), 7
Metrics (class in striemann.metrics), 1, 9
metrics() (striemann.test.fakes.FakeMetrics method), 15

N

name (striemann.metrics.MetricId attribute), 7

R

Range (class in striemann.metrics), 8
record() (striemann.metrics.Counter method), 9
record() (striemann.metrics.Gauge method), 8
record() (striemann.metrics.Range method), 9
Recorder (class in striemann.metrics), 7
recordGauge() (striemann.metrics.Metrics method), 1, 11
recordGauge() (striemann.test.fakes.FakeMetrics method), 16
recordRange() (striemann.metrics.Metrics method), 4, 11
recordRange() (striemann.test.fakes.FakeMetrics method), 16
RiemannTransport (class in striemann.metrics), 8

S

send() (striemann.metrics.Recorder method), 7
send_event() (striemann.metrics.CompositeTransport method), 8
send_event() (striemann.metrics.InMemoryTransport method), 8
send_event() (striemann.metrics.LogTransport method), 8
send_event() (striemann.metrics.RiemannTransport method), 8
striemann.metrics (module), 7
striemann.test.expectsmatcher (module), 16
striemann.test.fakes (module), 15

T

tags (striemann.metrics.MetricId attribute), 7
time() (striemann.metrics.Metrics method), 5, 13
time() (striemann.test.fakes.FakeMetrics method), 16
Timer (class in striemann.metrics), 9