DogDtatsD Collector

Release 0.1.0

Aug 23, 2019

Contents

CHAPTER 1

Overview

docs	
tests	
package	

`dogstatsd-collector` is a library to make it easy to collect DataDog-style StatsD counters and histograms with tags and control when they are flushed. It gives you a drop-in wrapper for the DogStatsD library for counters and histograms and allows you to defer flushing the metrics until you choose to. This capability enables you to collect StatsD metrics at arbitrary granularity, for example on a per-web request or per-job basis (instead of the per-flush interval basis).

Counters and histograms are tracked separately for each metric series (unique set of tag key-value pairs) and a single metric is emitted for each series when the collector is flushed. You don't have to think about tracking your metric series separately; you just use the `DogstatsdCollector` object as you would the normal `DogStatsD` object, and flush when you're ready; the library will take care of emitting all the series for you.

- Free software: BSD 3-Clause License

1.1 Installation

```
pip install dogstatsd-collector
```

1.2 Example Usage

Imagine you want to track a distribution of the number of queries issued by requests to your webapp, and tag them by which database is queried and which verb is used. You collect the following metrics as you issue your queries:

```
collector = DogstatsdCollector(dogstatsd)
...
collector.histogram('query', tags=['database:master','verb:insert'])
collector.histogram('query', tags=['database:master','verb:update'])
collector.histogram('query', tags=['database:master','verb:update'])
collector.histogram('query', tags=['database:replica','verb:select'])
collector.histogram('query', tags=['database:replica','verb:select'])
```

Then, at the end of your web request, when you flush the collector, the following metrics will be pushed to `DogStatsD` (shown in DogStatsD datagram format):

```
collector.flush()
# query:1|h|#database:master,verb:insert
# query:2|h|#database:master,verb:update
# query:2|h|#database:replica,verb:select
```

1.2.1 Base Tags

The collector object also supports specifying a set of base tags, which will be included on every metric that gets emitted.

```
base_tags = ['mytag:myvalue']
collector = DogstatsdCollector(dogstatsd, base_tags=base_tags)
collector.histogram('query', tags=['database:master','verb:insert'])
collector.histogram('query', tags=['database:master','verb:update'])
collector.flush()
# query:1|h|#database:master,verb:insert,mytag:myvalue
# query:1|h|#database:master,verb:update,mytag:myvalue
```

1.3 Motivation

The StatsD model is to run an agent on each server/container in your infrastructure and periodically flush aggregations at a regular interval to a centralized location. This model scales very well because the volume of metrics sent to the centralized location grows very slowly even as you scale your application; each StatsD agent calculates aggregations to flush to the backend instead of every datapoint, so the storage volume is quite low even for a large application with lots of volume.

A drawback to this model is that you don't have much control of the granularity that your metrics represent. When your aggregations reach the centralized location (DataDog in this case), you only know the counts or distributions within the flush interval. You can't represent any other *execution granularity* beyond "across X seconds" (where X is the flush interval). This limitation precludes you from easily representings metrics on a "per-request" basis, for example.

The purpose of this library is to make it simple to control when your StatsD metrics are emitted so that you can defer emission of the metrics until a point you determine. This allows you to represent a finer granularity than "across X seconds" such as "across a web request" or "across a cron job." It also preserves metric tags by emitting each series independently when the collector is flushed, which ensures you don't lose any of the benefit of tagging your metrics (such as aggregating/slicing in DataDog).

1.4 Patterns

The `DogstatsdCollector` object is a singleton that provides a similar interface as the `DogStatsD` increment and histogram methods. As you invoke these methods, you collect counters and histograms for each series (determined by any tags you include). After calling `flush()`, each series is separately emitted as a StatsD metric.

1.4.1 Simple Request Metrics

You can collect various metrics over a request and emit them at the end of the request to get per-request granularity.

In Django:

```
from datadog.dogstatsd.base import DogStatsd
from dogstatsd_collector import DogstatsdCollector

# Middleware
class MetricsMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
        self.dogstatsd = DogStatsd()

    def __call__(self, request):
        request.metrics = DogstatsdCollector(self.dogstatsd)
        response = self.get_response(request)
        request.metrics.flush()

        return response

# Inside a view
def my_view(request):
    # Do some stuff...
    request.metrics.increment('my.count')
    request.metrics.histogram('my.time', 0.5)
    return HttpResponse('ok')
```

In Flask:

```
from datadog.dogstatsd.base import DogStatsd
from dogstatsd_collector import DogstatsdCollector

from flask import Flask
from flask import request

app = Flask(__name__)
dogstatsd = DogStatsd()

@app.before_request
def init_metrics():
    request.metrics = DogstatsdCollector(dogstatsd)

@app.after_request
def flush_metrics():
    request.metrics.flush()

@app.route('/')
def my_view():
```

(continues on next page)

(continued from previous page)

```
    # Do some stuff...
    request.metrics.increment('my.count')
    request.metrics.histogram('my.time', 0.5)
    return 'ok'
```

1.4.2 Celery Task Metrics

Same as above, but over a Celery task.

```
from datadog.dogstatsd.base import DogStatsd
from dogstatsd_collector import DogstatsdCollector

from celery import Celery
from celery import current_task
from celery.signals import task_prerun
from celery.signals import task_postrun

app = Celery('tasks', broker='pyamqp://guest@localhost//')

dogstatsd = DogStatsd()

@task_prerun.connect
def init_metrics(task_id, task, *args, **kwargs):
    task.request.metrics = DogstatsdCollector(dogstatsd)

@task_postrun.connect
def flush_metrics(task_id, task, *args, **kwargs):
    task.request.metrics.flush()

@app.task
def my_task():
    # Do some stuff...
    current_task.request.metrics.increment('my.count')
    current_task.request.metrics.histogram('my.time', 0.5)
```

1.4.3 Metrics Within a Function

Emit a set of metrics for a particular function you execute.

```
from datadog.dogstatsd.base import DogStatsd
from dogstatsd_collector import DogstatsdCollector

dogstatsd = DogStatsd()

def do_stuff(metrics):
    # Do some stuff...
    metrics.increment('my.count')
    metrics.histogram('my.time', 0.5)

metrics = DogstatsdCollector(dogstatsd)
do_stuff(metrics)
metrics.flush()
```

1.5 Thread Safety

The `DogstatsdCollector` singleton is **not threadsafe.** Do not share a single `DogstatsdCollector` object among multiple threads.

1.6 More Documentation

Full documentation can be found on ReadTheDocs:

https://dogstatsd-collector.readthedocs.io/

1.7 Development

To run the all tests run:

```
tox
```

CHAPTER 2

Installation

At the command line:

```
pip install dogstatsd-collector
```

CHAPTER 3

Reference

3.1 dogstatsd_collector

class dogstatsd_collector.**DogstatsdCollector**(*dogstatsd*, *base_tags=None*)

A singleton for collecting DogStatsD-style metrics with tags. Collects metrics in-memory and then emits them when flush() is called. Each series (metric and all combination of tag key-value pairs) is emitted separately.

Parameters

- **dogstatsd** (*datadog.dogstatsd.base.DogStatsD*) – The DogStatsD object to use for emitting metrics.
- **base_tags** (*list*) – A list of tags to be included on every metric emitted from the collector. Should be of the form ['tag:value', ...]

SUPPORTED_DOGSTATSD_METRICS = ['histogram', 'increment']

The DogStatsD metrics supported by the collector.

flush()

Flush all metrics, emitting each metric once per series (combination of tag key-value pairs).

histogram(*metric*, *value*, *tags=None*)

Track a DogStatsD histogram metric.

increment(*metric*, *value=1*, *tags=None*)

Track a DogStatsD counter metric.

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

4.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.2 Documentation improvements

We could always use more documentation, whether as part of the official docs, in docstrings, or even on the web in blog posts, articles, and such.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/roverdotcom/dogstatsd-collector/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Code contributions are always welcome :)

4.4 Development

To set up *dogstatsd-collector* for local development:

1. Fork dogstatsd-collector (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/dogstatsd-collector.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

4.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

4.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 5

Authors

- Alex Landau - https://www.rover.com/

CHAPTER 6

Changelog

6.1 0.0.2 (2019-08-14)

- Add base_tags optional kwarg to support tags added to all metrics that get flushed.

6.2 0.0.1 (2019-05-02)

- First release on PyPI.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index