sprockets.mixins.metrics

Release 4.2

Jul 21, 2021

Contents

Adjust counter and timer metrics in StatsD using the same API.

The mix-in is configured through the `tornado.web.Application` settings property using a key defined by the specific mix-in.

CHAPTER 1

Deprecation Notice

This library is deprecated. It will not receive features or bug fixes. Please switch to either sprockets-influxdb or sprockets-statsd.

CHAPTER 2

Documentation

https://sprocketsmixinsmetrics.readthedocs.io

CHAPTER 3

Statsd Mixin

The following snippet configures the StatsD mix-in from common environment variables. This simple handler will emit a timer metric that identifies each call to the `get` method as well as a separate metric for the database query.

```
import os

from sprockets.mixins import mediatype
from sprockets.mixins.metrics import statsd
from tornado import web
import queries

def make_application():
    application = web.Application([
        web.url(r'/', MyHandler),
    ], **settings)

    statsd.install({'namespace': 'my-application',
                    'host': os.environ.get('STATSD_HOST', '127.0.0.1'),
                    'port': os.environ.get('STATSD_PORT', '8125')})
    return application

class MyHandler(statsd.StatsdMixin,
                mediatype.ContentMixin,
                web.RequestHandler):

    def initialize(self):
        super(MyHandler, self).initialize()
        self.db = queries.TornadoSession(os.environ['MY_PGSQL_DSN'])

    async def get(self, obj_id):
        with self.execution_timer('dbquery', 'get'):
            result = await self.db.query('SELECT * FROM foo WHERE id=%s',
                                         obj_id)
        self.send_response(result)
```

3.1 Settings

namespace The namespace for the measurements

host The Statsd host

port The Statsd port

prepend_metric_type Optional flag to prepend bucket path with the StatsD metric type

CHAPTER 4

Development Quickstart

```
$ python3.4 -mvenv env
$ . ./env/bin/activate
(env)$ env/bin/pip install -r requires/development.txt
(env)$ nosetests
test_metrics_with_buffer_not_flush (tests.InfluxDbTests) ... ok
test_that_cached_db_connection_is_used (tests.InfluxDbTests) ... ok
test_that_counter_is_tracked (tests.InfluxDbTests) ... ok
test_that_execution_timer_is_tracked (tests.InfluxDbTests) ... ok
test_that_http_method_call_details_are_recorded (tests.InfluxDbTests) ... ok
test_that_metric_tag_is_tracked (tests.InfluxDbTests) ... ok
test_that_add_metric_tag_is_ignored (tests.StatsdMethodTimingTests) ... ok
test_that_cached_socket_is_used (tests.StatsdMethodTimingTests) ... ok
test_that_counter_accepts_increment_value (tests.StatsdMethodTimingTests) ... ok
test_that_counter_increment_defaults_to_one (tests.StatsdMethodTimingTests) ... ok
test_that_default_prefix_is_stored (tests.StatsdMethodTimingTests) ... ok
test_that_execution_timer_records_time_spent (tests.StatsdMethodTimingTests) ... ok
test_that_http_method_call_is_recorded (tests.StatsdMethodTimingTests) ... ok

----------------------------------------------------------------------
Ran 13 tests in 3.572s

OK
(env)$ ./setup.py build_sphinx -q
running build_sphinx
(env)$ open build/sphinx/html/index.html
```

CHAPTER 5

License

5.1 Reference Documentation

This library defines mix-ins that record application metrics. Each mix-in implements the same interface:

class sprockets.mixins.metrics.**Mixin**

SETTINGS_KEY
Key in `self.application.settings` that contains this particular mix-in's configuration data.

record_timing(*duration*, **path*)

Parameters

- **duration** (*float*) – number of seconds to record
- **path** – timing path to record

```
self.record_timing(self.request.request_time(), 'request', 'lookup')
```

increase_counter(**path*, *amount=1*)

Parameters

- **path** – counter path to increment
- **amount** (*int*) – value to increase the counter by

```
self.increase_counter('db', 'query', 'foo')
```

execution_timer(**path*)

Parameters **path** – timing path to record

This method returns a context manager that records a timing metric to the specified path.

```
with self.execution_timer('db', 'query', 'foo'):
    rows = yield self.session.query('SELECT * FROM foo')
```

5.1.1 Statsd Implementation

class sprockets.mixins.metrics.statsd.**StatsdMixin**

Mix this class in to record metrics to a Statsd server.

execution_timer(**path*)

Record the time it takes to perform an arbitrary code block.

Parameters **path** – elements of the metric path to record

This method returns a context manager that records the amount of time spent inside of the context and submits a timing metric to the specified *path* using (*record_timing()*).

increase_counter(**path*, ***kwargs*)

Increase a counter.

This method increases a counter within the application's namespace. Each element of *path* is converted to a string and normalized before joining the elements by periods. The normalization process is little more than replacing periods with dashes.

Parameters

- **path** – elements of the metric path to incr
- **amount** (*int*) – amount to increase the counter by. If omitted, the counter is increased by one.

on_finish()

Records the time taken to process the request.

This method records the amount of time taken to process the request (as reported by `request_time()`) under the path defined by the class's module, it's name, the request method, and the status code. The *record_timing()* method is used to send the metric, so the configured namespace is used as well.

record_timing(*duration*, **path*)
Record a timing.

This method records a timing to the application's namespace followed by a calculated path. Each element of *path* is converted to a string and normalized before joining the elements by periods. The normalization process is little more than replacing periods with dashes.

Parameters

- **duration** (*float*) – timing to record in seconds
- **path** – elements of the metric path to record

class sprockets.mixins.metrics.statsd.**StatsDCollector**(*host*, *port*, *protocol='udp'*, *namespace='sprockets'*, *prepend_metric_type=True*)

Collects and submits stats to StatsD.

This class should be constructed using the *install()* function. When installed, it is attached to the Application instance for your web application.

Parameters

- **host** (*str*) – The StatsD host
- **port** (*str*) – The StatsD port
- **protocol** (*str*) – The StatsD protocol. May be either udp or tcp.
- **namespace** (*str*) – The StatsD bucket to write metrics into.
- **prepend_metric_type** (*bool*) – Optional flag to prepend bucket path with the StatsD metric type

close()
Gracefully close the socket.

send(*path*, *value*, *metric_type*)
Send a metric to Statsd.

Parameters

- **path** (*list*) – The metric path to record
- **value** (*mixed*) – The value to record
- **metric_type** (*str*) – The metric type

5.1.2 Application Functions

Before you can use the mixin, you have to install the client by calling the install function on your application instance.

sprockets.mixins.metrics.statsd.**install**(*application*, ***kwargs*)
Call this to install StatsD for the Tornado application.

Parameters

- **application** (*tornado.web.Application*) – the application to install the collector into.
- **kwargs** – keyword parameters to pass to the *StatsDCollector* initializer.

Returns True if the client was installed successfully, or False otherwise.

- **host The StatsD host. If host is not specified, the** `STATSD_HOST` environment variable, or default *127.0.0.1*, will be pass into the *StatsDCollector*.
- **port The StatsD port. If port is not specified, the** `STATSD_PORT` environment variable, or default *8125*, will be pass into the *StatsDCollector*.
- **namespace** The StatsD bucket to write metrics into.

sprockets.mixins.metrics.statsd.**get_client**(*application*)
Fetch the statsd client if it is installed.

Return type *StatsDCollector*

5.1.3 Testing Helpers

So who actually tests that their metrics are emitted as they expect?

Usually the answer is *no one*. Why is that? The `testing` module contains some helper that make testing a little easier.

class sprockets.mixins.metrics.testing.**FakeStatsdServer**(*iol*, *protocol='udp'*)
Implements something resembling a statsd server.

Parameters

- **iol** (*tornado.ioloop.IOLoop*) – the loop to attach to
- **protocol** (*str*) – The StatsD protocol. May be either `udp` or `tcp`.

Create an instance of this class in your asynchronous test case attached to the IOLoop and configure your application to send metrics to it. The received datagrams are available in the `datagrams` attribute for validation in your tests.

sockaddr
The socket address that the server is listening on. This is a tuple returned from `socket.socket.getsockname()`.

datagrams
A list of datagrams that have been received by the server.

find_metrics(*prefix*, *metric_type*)
Yields captured datagrams that start with *prefix*.

Parameters

- **prefix** (*str*) – the metric prefix to search for
- **metric_type** (*str*) – the statsd metric type (e.g., 'ms', 'c')

Returns yields (path, value, metric_type) tuples for each captured metric that matches

Raises **AssertionError** – if no metrics match.

handle_stream(*stream*, *address*)
Override to handle a new *.IOStream* from an incoming connection.

This method may be a coroutine; if so any exceptions it raises asynchronously will be logged. Accepting of incoming connections will not be blocked by this coroutine.

If this *TCPServer* is configured for SSL, `handle_stream` may be called before the SSL handshake has completed. Use *.SSLIOStream.wait_for_handshake* if you need to verify the client's certificate or use NPN/ALPN.

Changed in version 4.2: Added the option for this method to be a coroutine.

5.2 Configuration

sprockets.mixins.metrics has the ability to be configured via environment variables.

The following environment variables are recognized:

Name	Description	Required	Default Value
STATSD_HOST	The StatsD host to connect to	No	127.0.0.1
STATSD_PORT	The port on which StatsD is listening	No	8125
STATSD_PROTOCOL	The transport-layer protocol to use	No	8125

5.3 Examples

5.3.1 Sending metrics to StatsD

This simple application emits metrics to port 8125 on localhost. The mix-in is configured by passing a `sprockets.mixins.metrics.statsd` key into the application settings as shown below.

```
def make_application():
    """
    Create a application configured to send metrics.

    Metrics will be sent to localhost:8125 namespaced with
    ``webapps``. Run netcat or a similar listener then run this
    example. HTTP GETs will result in a metric like::

        webapps.SimpleHandler.GET.204:255.24497032165527|ms

    """
    settings = {}
    application = web.Application([web.url('/', SimpleHandler)], **settings)
    statsd.install(application, **{'namespace': 'testing'})
    return application
```

The request handler is simple. In fact, there is nothing of interest there except that it uses `StatsdMixin` as a base class.

```
class SimpleHandler(statsd.StatsdMixin, web.RequestHandler):
    """
    Simply emits a timing metric around the method call.

    The metric namespace and StatsD endpoint are configured in
    the application settings object so there is nothing to do in
    a request handler.

    """

    async def get(self):
        await asyncio.sleep(0.25)
        self.set_status(204)
        self.finish()

    def post(self):
```

(continues on next page)

(continued from previous page)

```
        """Example of increasing a counter."""
        self.increase_counter('request', 'path')
        self.set_status(204)
```

5.4 How to Contribute

Do you want to contribute fixes or improvements?

> **AWesome!** *Thank you very much, and let's get started.*

5.4.1 Set up a development environment

The first thing that you need is a development environment so that you can run the test suite, update the documentation, and everything else that is involved in contributing. The easiest way to do that is to create a virtual environment for your endevours:

```
$ python3.4 -mvenv env
```

Don't worry about writing code against previous versions of Python unless you you don't have a choice. That is why we run our tests through tox. If you don't have a choice, then install virtualenv to create the environment instead. The next step is to install the development tools that this project uses. These are listed in *requires/development.txt*:

```
$ env/bin/pip install -qr requires/development.txt
```

At this point, you will have everything that you need to develop at your disposal. *setup.py* is the swiss-army knife in your development tool chest. It provides the following commands:

./setup.py nosetests Run the test suite using nose and generate a nice coverage report.

./setup.py build_sphinx Generate the documentation using sphinx.

./setup.py flake8 Run flake8 over the code and report style violations.

If any of the preceding commands give you problems, then you will have to fix them **before** your pull request will be accepted.

5.4.2 Running Tests

The easiest (and quickest) way to run the test suite is to use the *nosetests* command. It will run the test suite against the currently installed python version and report not only the test result but the test coverage as well:

```
$ ./setup.py nosetests
running nosetests
running egg_info
writing sprockets.mixins.metrics.egg-info/PKG-INFO
writing top-level names to sprockets.mixins.metrics.egg-info/top_level.txt
writing dependency_links to sprockets.mixins.metrics.egg-info/dependency_links.txt
writing namespace_packages to sprockets.mixins.metrics.egg-info/namespace_packages.txt
reading manifest file 'sprockets.mixins.metrics.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
writing manifest file 'sprockets.mixins.metrics.egg-info/SOURCES.txt'
test_that_cached_socket_is_used (tests.StatsdMethodTimingTests) ... ok
```

(continues on next page)

(continued from previous page)

```
test_that_counter_accepts_increment_value (tests.StatsdMethodTimingTests) ... ok
test_that_counter_increment_defaults_to_one (tests.StatsdMethodTimingTests) ... ok
test_that_default_prefix_is_stored (tests.StatsdMethodTimingTests) ... ok
test_that_execution_timer_records_time_spent (tests.StatsdMethodTimingTests) ... ok
test_that_http_method_call_is_recorded (tests.StatsdMethodTimingTests) ... ok

----------------------------------------------------------------------
Ran 6 tests in 1.080s

OK
```

That's the quick way to run tests. The slightly longer way is to run the tox utility. It will run the test suite against all of the supported python versions in parallel. This is essentially what Travis-CI will do when you issue a pull request anyway:

```
$ env/bin/tox
GLOB sdist-make: /Users/daves/Source/platform/sprockets.mixins.metrics/setup.py
py27 create: /Users/daves/Source/platform/sprockets.mixins.metrics/build/tox/py27
py27 installdeps: -rrequires/testing.txt

________________________ >8 ________________________________________________

  py27: commands succeeded
  py34: commands succeeded
  py35: commands succeeded
SKIPPED:  pypy: InterpreterNotFound: pypy
  congratulations :)
```

This is what you want to see. Now you can make your modifications and keep the tests passing. If you see the "missing interpreter" errors, that means that you do not have all of the interpreters installed.

5.4.3 Submitting a Pull Request

Once you have made your modifications, gotten all of the tests to pass, and added any necessary documentation, it is time to contribute back for posterity. You've probably already cloned this repository and created a new branch. If you haven't, then checkout what you have as a branch and roll back *master* to where you found it. Then push your repository up to github and issue a pull request. Describe your changes in the request, if Travis isn't too annoyed someone will review it, and eventually merge it back.

5.5 Release History

5.5.1 4.2.0 (21-Jul-2021)

- Library deprecated – use sprockets-statsd if you need StatsD support

5.5.2 4.1.0 (04-Sep-2019)

- Add configuration documentation
- Exclude Tornado >6 (as-yet-unreleased version)
- Add *sprockets.mixins.metrics.statsd.get_client()* function

- Add *sprockets.mixins.metrics.statsd.StatsDCollector.close()* method

5.5.3 4.0.0 (06-Feb-2019)

- Add support for Tornado 5
- Remove support for Tornado < 5
- Remove support for Python < 3.7
- Remove InfluxDB support (use sprockets-influxdb)

5.5.4 3.1.1 (07-Aug-2018)

- Fixed bad formatted TCP StatsD messages by appending a newline

5.5.5 3.1.0 (20-Jul-2018)

- Add TCP support to StatsD

5.5.6 3.0.4 (31-Jan-2018)

- Loosen Tornado pin to include 4.4.

5.5.7 3.0.3 (24-Mar-2017)

- Fix retrival of status code.

5.5.8 3.0.2 (12-Dec-2016)

- Fix influxdb test that fails intermittently.

5.5.9 3.0.1 (12-Dec-2016)

- Add README.rst to MANIFEST.in

5.5.10 3.0.0 (12-Dec-2016)

- Add install usage pattern for using mixin within Tornado app
- Strip down statsd mixin adding a collector class to do metric recording
- Add path prefix for the metric type, eg. counters, timers, etc
- Add configuration parameters to enable/disable metric type prefix

5.5.11 2.1.1 (9-Aug-2016)

- Fix InfluxDB URL creation from environment variables

5.5.12 2.1.0 (9-Aug-2016)

- Add authentication environment variables for InfluxDB

5.5.13 2.0.1 (21-Mar-2016)

- Make it possible to call methods (e.g., `set_metric_tag()`) during the Tornado request handler initialization phase.

5.5.14 2.0.0 (11-Mar-2016)

- Rework InfluxDB buffering to use a periodic callback instead of flushing the buffer upon request.

5.5.15 1.1.1 (9-Mar-2016)

- Fix packaging woes part deux.

5.5.16 1.1.0 (9-Mar-2016)

- Update InfluxDB mixin to buffer measurements across requests based on a max time and/or length.

5.5.17 1.0.1 (1-Feb-2016)

- Fix packaging woes.

5.5.18 1.0.0 (1-Feb-2016)

- Remove extraneous quotes from InfluxDB tag values.
- Convert HTTP status code from value to a tag in the InfluxDB mix-in.

5.5.19 0.9.0 (27-Jan-2016)

- Add `sprockets.mixins.metrics.StatsdMixin`
- Add *sprockets.mixins.metrics.testing.FakeStatsdServer*
- Add `sprockets.mixins.metrics.testing.FakeInfluxHandler`
- Add `sprockets.mixins.metrics.InfluxDBMixin`
- Add `sprockets.mixins.metrics.influxdb.InfluxDBConnection`

Index