
DogWhistle Documentation

Release 0.6.0

IST Research

Sep 28, 2017

Contents

1	Usage	1
1.1	Analyze	1
1.2	Local Configuration	4
1.3	Datadog Configuration	7
1.4	Wrapping Up	9
2	API Reference	11
3	License	13
4	Change Log	15
4.1	0.5.0	15
4.2	0.4.0	15
4.3	0.3.0	15
4.4	0.2.0	15
4.5	0.1.2	16
4.6	0.1.1	16
4.7	0.1.0	16
	Python Module Index	17

This section outlines how to use the `dog_whistle` library in your project.

Analyze

The first step you need to do is to download the `dog_whistle` library from pip.

```
pip install dog-whistle -U
```

Navigate to the root directory of your project or code base, and run the following commands in the python shell:

```
$ python
>>> from dog_whistle import dw_analyze
>>> dw_analyze('./')
# it will spit out a bunch of text, like below:

Valid Lines
-----
./crawling/distributed_scheduler.py
  111 : self.logger.error("Could not connect to Zookeeper")
  117 : self.logger.error("Could not ping Zookeeper")
  131 : self.logger.info("Zookeeper config changed", extra=loaded_config)
  189 : self.logger.info("Lost config from Zookeeper", extra=extras)
  303 : self.logger.error("Could not reach out to get public ip")
  381 : self.logger.info("Closing Spider", {'spiderid':self.spider.name})
  383 : self.logger.warning("Clearing crawl queues")

Invalid Lines
-----

<<<<<<<<< YOU MUST FIX THESE BEFORE USING THE DOGWHISTLE LIBRARY >>>>>>>>>

./crawling/spiders/wandering_spider.py
   87 : self._logger.info("Did not find any more links" + str(value))
```

Auto-Generated Template Settings

```
-----

dw_dict = {
    'name': '<my_project>',
    'tags': [
        # high level tags that everything in your app will have
        'item:descriptor'
    ],
    'metrics': {
        # By default, everything is a counter using the concatenated log string
        # the 'counters' key is NOT required, it is shown here for illustration
        'counters': [
            # datadog metrics that will use ++
            ("Could not connect to Zookeeper", "could_not_connect_to_zookeeper"),
            ("Could not ping Zookeeper", "could_not_ping_zookeeper"),
            ("Zookeeper config changed", "zookeeper_config_changed"),
            ("Lost config from Zookeeper", "lost_config_from_zookeeper"),
        ],
        # datadog metrics that have a predefined value like `51`
        # These metrics override any 'counter' with the same key,
        # and are shown here for illustration purposes only
        'gauges': [

            ("Zookeeper config changed", "zookeeper_config_changed", "<extras.key.
↪path>"),
            ("Lost config from Zookeeper", "lost_config_from_zookeeper", "<extras.key.
↪path>"),
        ]
    },
    'options': {
        # use statsd for local testing, see docs
        'statsd_host': 'localhost',
        'statsd_port': 8125,
        'local': True,
    },
}

Ensure the above dictionary is passed into `dw_config()`

>>>
```

Lets break down each of these sections in more detail.

Valid Lines

This section outlines the valid logger lines that dog whistle has detected will work without issue. They should represent 'static' messages, and not include any variable substitution within the log message itself.

Invalid Lines

These lines were noted to have some kind of variable substitution inside them, and need to be corrected before being used by the dog whistle module. All variables *should* be enclosed in the `extras` dictionary anyways, and this helps standardize our logging practices.

Warning: You **must** correct these lines before using the dog whistle in production, otherwise our datadog metrics will not be consistent

Please note that the `dw_analyze()` method may not pick everything up, and it is important to test and look through your code to find any problematic lines. Multiline log states, or more complex logging may not be picked up.

Auto-Generated Template Settings

By analyzing only your valid lines, the dog whistle library dumps out a dictionary object that you will need to tweak and use later. At a bare minimum, it requires the following key:

- **name** - the name of your overall project
- **options** - the options to be passed to configure the statsd or datadog setup

For example:

```
{
  'name': 'cool project',
  'options': {
    'statsd_host': 'localhost',
    'statsd_port': 8125,
    'local': True,
  }
}
```

This configures all log messages to be counters, tied to the `cool project` namespace, and configured to use a local statsd host.

Note: If you have multiple modules or components to your project, you can use dot notation to namespace them like `my_project.component1`

Further configuration can be refined via the `metrics` key, allow you to specify custom mappings of *log messages* to *keys*.

```
'counters': [
  ("Could not connect to Zookeeper", "zookeeper.connection.error"),
  ("Could not ping Zookeeper", "zookeeper.connection.ping"),
  ("Zookeeper config changed", "zookeeper.config_changed"),
  ("Lost config from Zookeeper", "lost_config_from_zookeeper"),
],
```

Normally, the dog whistle sanitizes the log message into a lowercase/underscore form. However, we also provide the ability to custom map log messages to key naming conventions.

In the above example, we see a `counters` mapping being applied to three of the four log messages. Instead of using the default (shown on line 4), it will use the custom key.

The same can be said for gauges:

```
'gauges': [
  ("Zookeeper config changed", "zookeeper_config_changed", "buried.key.here"),
  ("Lost config from Zookeeper", "zookeeper.connection.problem", "num_tries"),
]
```

The dog whistle library automatically detects `extras` being passed into the log method, and adds lines here to recommend you use a gauge incase you are tracking a particular value in question via your `extras` dictionary.

Here, we dig into the extras dictionary using dot notation to try to find the value we are looking for. If no value is found, it is not sent.

Multiple gauges can also be mapped to a single log statement within your extras dictionary. Just supply a **list** of different keys you would like to send to Datadog like so:

```
'gauges': [
    ("Counter Stats Dump", [
        ("counter_dump_num_values", "counter.total_values"),
        ("counter_dump_num_connections", "counter.connections"),
        ("counter_dump_lifetime_users", "lifetime_users"),
    ])
]
```

Where your extras dictionary in your log statement might look like the following:

```
extras = {
    'counter': {
        'total_values': 5,
        'connections': 12
    },
    'lifetime_users': 511
}
logger.info("Counter Stats Dump", extras)
```

Here, we supply a normal dictionary to be logged to the logger instance, but DogWhistle is able to pick up and parse the single log statement into multiple gauges to be sent to Datadog.

Lastly, tags are something that will always be included in your datadog stats. Here, you can specify a unique descriptor or other item to identify your process from the rest of the group. These tags are optional, but are helpful.

Setting `allow_extra_tags=True` in your configuration will allow you to add additional tags on a per-message basis:

```
logger.info("this is a test",extra={'tags': ['my:tag', 'my:othertag']})
```

Note: Tags are not sent while dog whistle is in local configuration mode.

Local Configuration

Setup

Now that you have an idea about your configuration, you need to integrate dog whistle into every python process or application you run. You will need to get the settings dictionary with your proper configuration into your application somehow. This guide does not cover the various ways of including the dictionary, however it is advised that you use either a settings file, environment variables, or some other way to avoid hard coding critical settings into your source code.

Once you have figured that out, at a **single** point within your application, add the following lines of code:

```
from dog_whistle import dw_config, dw_callback
settings = {} # your settings here
dw_config(settings)
```

This will configure your dog whistle library to be ready to send metrics, the next step is to add a LogFactory callback like so:

```
logger = LogFactory.get_instance() # your normal LogFactory setup can go here
logger.register_callback('>=INFO', dw_callback)
```

Note: You will need `scutis==1.2.0` or above in order to use the callback feature in your project. Please update your requirements appropriately!

This will allow the dog whistle library to integrate and monitor every call the LogFactory logger creates. The callback system is much more advanced than what is described here, but this gives us the ability to monitor all log messages actually written by logger, anything ignored by the logger will also be ignored by this callback.

Testing

Let's test our configuration using a simple `statsd` + `graphite` host. Here, we are going to use Docker to pull a container that allows us to view our new metrics to check naming conventions, typos, and other things.

If you are running your script locally, use the following:

```
$ docker run -p 80:80 -p 2003-2004:2003-2004 -p 2023-2024:2023-2024 -p 8125:8125/udp -
  -p 8126:8126 hopsoft/graphite-statsd
```

```
'options': {
    # use statsd for local testing, see docs
    'statsd_host': 'localhost',
    'statsd_port': 8125,
    'local': True,
}
```

If you are using Docker Compose, we recommend instead using the following settings:

```
statsd:
  image: hopsoft/graphite-statsd
  ports:
    - "80:80"
    - "2003-2004:2003-2004"
    - "2023-2024:2023-2024"
    - "8125:8125/udp"
    - "8126:8126"
```

Then set the following in your options for your script:

```
'options': {
    # use statsd for local testing, see docs
    'statsd_host': os.getenv('DATADOG_STATSD_HOST', 'statsd'),
    'statsd_port': int(os.getenv('DATADOG_STATSD_PORT', 8125)),
    'local': os.getenv('DATADOG_LOCAL', 'False') == 'True',
},
```

and use an environment variable in your application:

```
apl:
  image: example/cool-app:prod
```

This will allow you to toggle your Datadog configuration on in local testing, but leaving it off will use production settings.

You can visit `localhost:80` to view your Graphite dashboard. On the Tree on the left hand side, navigate to `Metrics/stats`. You should see your project name as a folder, and you can click on the individual metric to get it to show up in the graph.

Datadog Configuration

This section outlines how to test and connect DogWhistle to Datadog.

Local Datadog Agent

Here, we assume you have a local [Datadog Agent](#) running on your local machine. Please refer to the official documentation for troubleshooting connecting your machine to Datadog.

At this point you should have a Datadog Agent installed and running successfully on your machine. Refer to your datadog configuration file to get the location of the statsd host that Datadog is running. For example, on unix based systems the log location is:

```
dogstatsd_log_file: /var/log/datadog/dogstatsd.log
```

This log is important to help debug any errors that may occur. One of the first problems you may encounter is that the agent may be shut down due to the following error.

```
Traceback (most recent call last):
  File "/opt/datadog-agent/agent/dogstatsd.py", line 396, in run
    self.server.start()
  File "/opt/datadog-agent/agent/dogstatsd.py", line 327, in start
    self.socket.bind(self.address)
  File "/opt/datadog-agent/embedded/lib/python2.7/socket.py", line 228, in meth
    return getattr(self._sock,name)(*args)
error: [Errno 48] Address already in use
```

This is caused by running both our Docker based Statsd container, and the Datadog Agent. Shut down the docker container and then restart the agent for it to boot up successfully.

If you are using Docker, the following extra configuration is required in your Datadog Agent conf file.

```
# Required for Docker
non_local_traffic: yes
```

The following python snippet allows us to test if DogWhistle and your Datadog Agent are working correctly

```
# imports
import logging
from dog_whistle import dw_config, dw_callback
from scutils.log_factory import LogFactory

# show all application logs
logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.DEBUG)

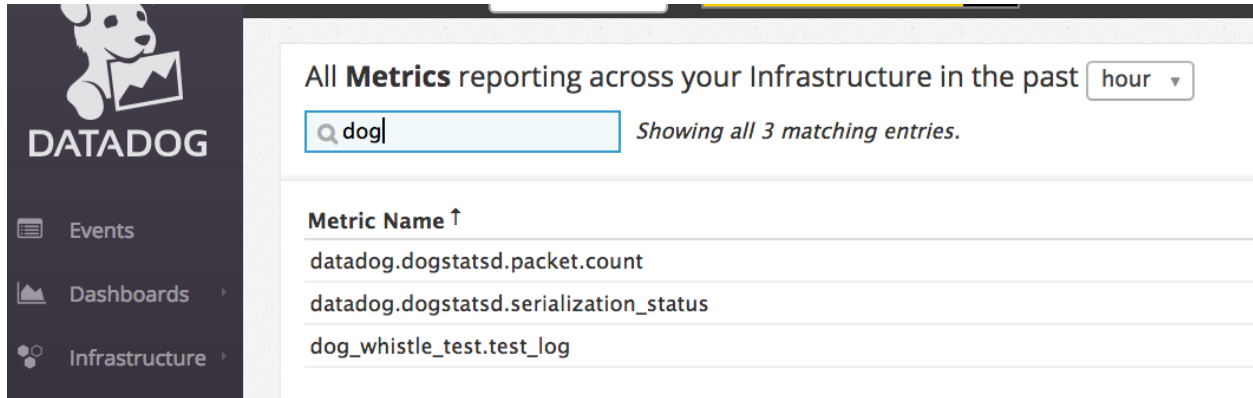
# our settings
DW_SETTINGS = {
    'name': 'dog-whistle_test',
    'options': {
        'statsd_host': "localhost",
        'statsd_port': 8125,
        'local': True,
    },
}

# configure dog whistle and log factory
```

```
dw_config(DW_SETTINGS)
logger = LogFactory.get_instance()
logger.register_callback('>=INFO', dw_callback)

logger.info("test log")
```

While running this you should see a number of successful `DEBUG` or `INFO` level messages come through your console. Within a few minutes, visit the Datadog [Metrics Summary](#) page, and you should see your new metric appear like so.



If your log messages begin to show up, you are now ready to enable DogWhistle locally to fully test your integration! For a more thorough test of DogWhistle, you can use the `example.py` script located within the base of this repo.

You will want to make sure your final application can hit the local Datadog Agent, if running as a normal process use

```
'statsd_host': "localhost",
```

within your options. If using Docker on Linux, set your configuration to

```
environment:
- DATADOG_LOCAL=True
- DATADOG_STATSD_HOST=172.17.0.1 # linux only
```

If using Docker on Mac OS, set your configuration to

```
environment:
- DATADOG_LOCAL=True
- DATADOG_STATSD_HOST=docker.for.mac.localhost # Mac only
```

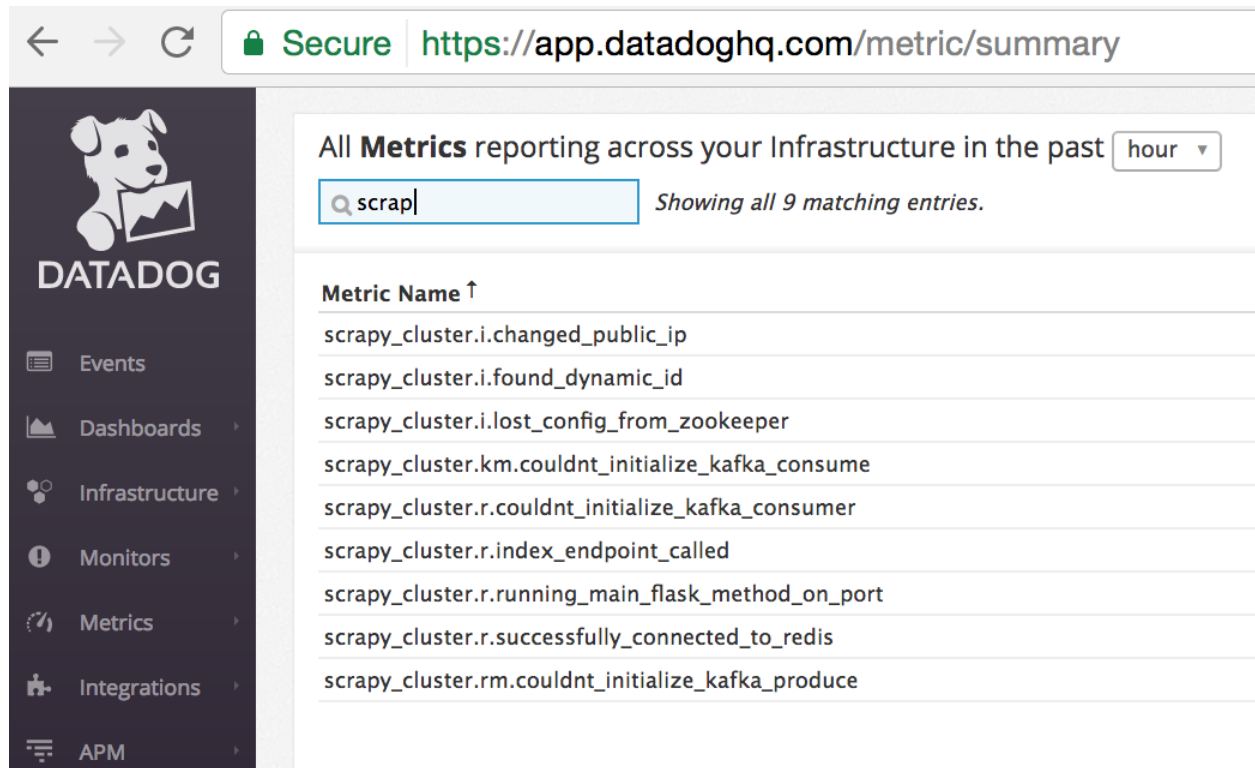
to ensure your container can hit the local Datadog Agent.

Once your application is configured, you can see the metrics in your [Metrics Summary](#) like below, and begin building custom dashboards, graphs, and alerts based on your logging.

DataDog Statsd Configuration

To enable DataDog integration simply set the `local` option to `False`, and ensure that `statsd_host` is pointing to a dd-agent. For instance:

```
environment:
- DATADOG_LOCAL=False
- DATADOG_STATSD_HOST=172.17.0.1
```



The screenshot shows the Datadog web application interface. On the left is a dark sidebar with the Datadog logo and navigation links: Events, Dashboards, Infrastructure, Monitors, Metrics, Integrations, and APM. The main content area is titled 'All Metrics reporting across your Infrastructure in the past' with a dropdown menu set to 'hour'. A search bar contains the text 'scrap', and a message indicates 'Showing all 9 matching entries.' Below this is a table of metrics.

Metric Name ↑
scrapy_cluster.i.changed_public_ip
scrapy_cluster.i.found_dynamic_id
scrapy_cluster.i.lost_config_from_zookeeper
scrapy_cluster.km.couldnt_initialize_kafka_consume
scrapy_cluster.r.couldnt_initialize_kafka_consumer
scrapy_cluster.r.index_endpoint_called
scrapy_cluster.r.running_main_flask_method_on_port
scrapy_cluster.r.successfully_connected_to_redis
scrapy_cluster.rm.couldnt_initialize_kafka_produce

Wrapping Up

Once you have your ideal Datadog integration set up, you can build custom dashboards like the one shown below:

Be sure to add the following to your projects requirements.txt!

```
dog-whistle==X.X
scutis==1.3.0dev0 # required for Python 2/3 compatibility
```

Where X.X is the current version of dog-whistle on pypi

Note: Version numbers are subject to change!



`dog_whistle._ddify(message, prepend=True)`

Datadogify and normalizes a log message into a datadog key

Parameters

- **message** (*str*) – The message to concatenate
- **prepend** (*bool*) – prepend the application name to the metric

Returns the final datadog string result

`dog_whistle._gauge(name, value, tags)`

Increments a gauge

Parameters

- **name** (*str*) – The name of the stats
- **value** (*int*) – The value of the gauge
- **tag** (*list*) – A list of tags

`dog_whistle._get_config()`

Returns the current configuration of the module

`dog_whistle._get_value(item, key)`

Grabs a nested value within a dict

Parameters

- **item** (*dict*) – the dictionary
- **key** (*str*) – the nested key to find

Returns the value if found, otherwise None

`dog_whistle._increment(name, tags)`

Increments a counter

Parameters

- **name** (*str*) – The name of the stats
- **tag** (*list*) – A list of tags

`dog_whistle._reset()`

Resets the module configuration to the defaults

`dog_whistle.dw_analyze(path)`

Used to analyze a project structure and output the recommended settings dictionary to be used when used in practice. Run this method, then add the resulting output to your project

Parameters **path** (*str*) – The folder path to analyze

`dog_whistle.dw_callback(message, extras)`

The actual callback method passed to the logger

Parameters

- **message** (*str*) – The log message
- **extras** (*dict*) – The extras dictionary from the logger

`dog_whistle.dw_config(settings)`

Set up the datadog callback integration

Parameters **settings** (*dict*) – The settings dict containing the *dw_analyze()* configuration

Raises `Exception` if configuration is missing

CHAPTER 3

License

The MIT License (MIT)

Copyright (c) 2017 IST Research

Permission **is** hereby granted, free of charge, to **any** person obtaining a copy of this software **and** associated documentation files (the "**Software**"), to deal **in** the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, **and/or** sell copies of the Software, **and** to permit persons to whom the Software **is** furnished to do so, subject to the following conditions:

The above copyright notice **and** this permission notice shall be included **in all** copies **or** substantial portions of the Software.

THE SOFTWARE IS PROVIDED "**AS IS**", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 4

Change Log

This page serves to document any changes made between releases.

0.5.0

Date: 09/28/17

- Remove requirement for API keys when not in local mode

0.4.0

Date: 07/26/17

- Strip trailing periods
- Allow custom tags per message

0.3.0

Date: 05/18/17

- Added multi gauge support
- Improved regexes for dw_analyze

0.2.0

Date: 04/25/17

- Added Python3 Support
- Added Changelog

0.1.2

Date: 04/20/17

- Finished Datadog Agent Support documentation

0.1.1

Date: 02/08/17

- Corrected Datadog vs Statsd integration

0.1.0

Date: 02/02/17

- Initial Release

Welcome to DogWhistle's documentation! This project is designed to make [Datadog](#) and [Logfactory](#) integration go as smooth as possible, with minimal code overhead.

d

`dog_whistle`, [11](#)

Symbols

[_ddify\(\)](#) (in module [dog_whistle](#)), [11](#)
[_gauge\(\)](#) (in module [dog_whistle](#)), [11](#)
[_get_config\(\)](#) (in module [dog_whistle](#)), [11](#)
[_get_value\(\)](#) (in module [dog_whistle](#)), [11](#)
[_increment\(\)](#) (in module [dog_whistle](#)), [11](#)
[_reset\(\)](#) (in module [dog_whistle](#)), [12](#)

D

[dog_whistle](#) (module), [11](#)
[dw_analyze\(\)](#) (in module [dog_whistle](#)), [12](#)
[dw_callback\(\)](#) (in module [dog_whistle](#)), [12](#)
[dw_config\(\)](#) (in module [dog_whistle](#)), [12](#)