metlog-py Documentation

Release 0.10.0

Rob Miller

December 06, 2013

Contents

This project has been superceded by *heka-py https://github.com/mozilla-services/heka-py* and *hekad https://github.com/mozilla-services/heka*.

This repository is no longer being actively maintained.

metlog-py is a Python client for the "Metlog" system of application logging and metrics gathering developed by the Mozilla Services team. The Metlog system is meant to make life easier for application developers with regard to generating and sending logging and analytics data to various destinations. It achieves this goal (we hope!) by separating the concerns of message generation from those of message delivery and analysis. Front end application code no longer has to deal directly with separate back end client libraries, or even know what back end data storage and processing tools are in use. Instead, a message is labeled with a type (and possibly other metadata) and handed to the Metlog system, which then handles ultimate message delivery.

The Metlog system consists of three pieces:

generator This is the application that will be generating the data that is to be sent into the system.

router This is the initial recipient of the messages that the generator will be sending. Typically, a metlog router deserializes the messages it receives, examines them, and decides based on the message metadata or contents which endpoint(s) to which the message should be delivered.

endpoints Different types of messages lend themselves to different types of presentation, processing, and analytics. The router has the ability to deliver messages of various types to destinations that are appropriate for handling those message types. For example, simple log messages might be output to a log file, while counter timer info is delivered to a statsd server, and Python exception information is sent to a Sentry server.

The metlog-py library you are currently reading about is a client library meant to be used by Python-based generator applications. It provides a means for those apps to insert messages into the system for delivery to the router and, ultimately, one or more endpoints.

More information about how Mozilla Services is using Metlog (including what is being used for a router and what endpoints are in use / planning to be used) can be found on the relevant spec page.

There are two primary components to the metlog-py library, the *MetlogClient* client class which exposes the primary metlog client API, and the various *Senders* classes, one of which must be provided to the MetlogClient and which handles the actual delivery of the message to the router component.

The MetlogClient can be instantiated directly, but metlog-py also provides some utility functions that will parse config files and set up a configured client instance for you. Folks new to using Metlog will probably find *Metlog Configuration* a good place to get started.

CHAPTER 1

Contents

1.1 Getting Started

There are two primary components with which users of the metlog-py library should be aware. The first is the *MetlogClient* client class. The MetlogClient exposes the Metlog API, and is generally your main point of interaction with the Metlog system. The client doesn't do very much, however; it just provides convenience methods for constructing messages of various types and then passes the messages along. Actual message delivery is handled by a *sender*. Without a properly configured sender, a MetlogClient is useless.

The first question you're likely to ask when using metlog-py, then, will probably be "How the heck do I get my hands on a properly configured client / sender pair?" You could read the source and instantiate and configure these objects yourself, but for your convenience we've provided a *Metlog Configuration* module that simplifies this process considerably. The config module provides utility functions that allow you pass in a declarative representation of the settings you'd like for your client and sender objects, and it will create and configure them for you based on the provided specifications.

1.1.1 Configuration formats

As described in detail on the *Metlog Configuration* page, you can create a MetlogClient from either an INI-file style text configuration (using *client_from_text_config* or *client_from_stream_config*) or a dictionary (using *client_from_dict_config*). The text version is simply parsed and turned into a dictionary internally, so the following text config:

```
[metlog]
logger = myapp
severity = 4
disabled_timers = foo
                  bar
sender_class = metlog.senders.zmq.ZmqPubSender
sender_bindstrs = tcp://127.0.0.1:5565
sender_queue_length = 5000
global_disabled_decorators = incr_count
```

Will be converted into this dictionary config:

```
{'logger': 'myapp',
 'severity': 4,
 'disabled_timers': ['foo', 'bar'],
'sender': {'class': 'metlog.senders.zmq.ZmqPubSender',
           'bindstrs': 'tcp://127.0.0.1:5565',
           'queue_length': 5000,
           },
'global': {'disabled_decorators': ['incr_count']
           },
}
```

Let's ignore the details of the config options for now (again, see *Metlog Configuration* for the nitty-gritty) and just assume that you have a working client configuration. How might you actually use this within your code to generate a client and make that client avaialable for use? A couple of mechanisms are described below.

1.1.2 Example 1: Use your framework

It is of course possible to simply define or load your configuration up at module scope and then use it to create a client that is then available to the rest of the module, like so:

```
from metlog.config import client_from_stream_config

with open('/path/to/config.ini', 'r') as inifile:
    metlogger = client_from_stream_config(inifile, 'metlog')

metlogger.metlog('msgtype', payload='Message payload')
```

However, this is considered by many (including the Metlog authors) to be A Bad Idea®. Instead of creating import time side effects, you can use your application's bootstrap code to initialize your client and make it available as needed. How exactly this should work varies from application to application. For instance, the Pyramid web framework expects a *main* function to be defined at the root of your application's package. This might contain the following code:

```
from metlog.config import client_from_stream_config
from pyramid.config import Configurator

def main(global_config, **settings):
    """ This function returns a Pyramid WSGI application.
    """
    config = Configurator(settings=settings)
    with open('/path/to/config.ini', 'r') as metlog_ini:
        metlog_client = client_from_stream_config(metlog_ini, 'metlog')
        config.registry['metlog'] = metlog_client
    config.add_static_view('static', 'static', cache_max_age=3600)
    config.add_route('home', '/')
    config.scan()
    return config.make_wsgi_app()
```

Then the MetlogClient instance will be available on Pyramid's "registry", which is always available through the request object or a library call. With a bit more code you could put your Metlog configuration info into Pyramid's default config file and then extract it from the *settings* values passed in to the *main* function. The idea is that you make use of whatever infrastructure or patterns that your application and/or framework provide and cooperate with those to create and make available a client for logging and metrics-gathering needs.

1.1.3 Example 2: Module scope, if you must

Despite the fact that some consider it to be an anti-pattern, there are those who are quite fond of the *import logging; logger = logging.getLogger('foo')* idiom that the stdlib logging package provides for making a logger available at module scope. We recommend that you consider not doing so and instead making your client available through some application- or framework-specific mechanism, but if you really want to stick to your guns then there's a bit of convenience that metlog-py provides.

The short version is that where you would have done this:

```
from logging import getLogger
logger = getLogger('myapp')
```

Instead you'd do the following:

```
from metlog.holder import get_client
metlogger = get_client('myapp')
```

Every time throughout your application's process, a call to *get_client('myapp')* will return the same MetlogClient instance. At this point, however, the client in question is still not usable, because it doesn't have a working sender. Again, the recommendation is that somewhere in your application code you use one of the config functions to initialize the client, which might look like this:

```
from metlog.config import client_from_stream_config
from metlog.holder import get_client
metlogger = get_client('myapp')

def some_init_function():
    with open('/path/to/metlog.ini', 'r') as metlog_ini:
        client_from_stream_config(metlog_ini, 'metlog', metlogger)
```

Note that the *metlogger* client was passed in to the *client_from_stream_config* call, which causes the configuration to be applied to that client rather than a new client being created.

If you *really* want to do all of your initialization at module scope, you can pass a config dict to the *get_client* function. This is a minimal working configuration that will cause all Metlog output to be sent to stdout:

```
from metlog.holder import get_client
metlog_config = {'sender': {'class': 'metlog.senders.dev.StdOutSender'}}
metlogger = get_client('myapp', metlog_config)
```

1.2 Metlog Configuration

To assist with getting a working Metlog set up, metlog-py provides a *Config* module which will take declarative configuration info in either ini file or python dictionary format and use it to configure a MetlogClient instance. Even if you choose not to use these configuration helpers, this document provides a good overview of the configurable options provided by default by the *MetlogClient* client class.

The config module will accept configuration data either in ini format (as a text value or a stream input) or as a Python dictionary value. This document will first describe the supported ini file format, followed by the corresponding dictionary format to which the ini format is ultimately converted behind the scenes.

1.2.1 ini format

The primary *MetlogClient* configuration should be provided in a *metlog* section of the provided ini file text. (Note that the actual name of the section is passed in to the config parsing function, so it can be any legal ini file section name, but for illustration purposes these documents will assume that the section name is *metlog*.) A sample *metlog* section might look like this:

```
[metlog]
logger = myapp
severity = 4
disabled_timers = foo
                  bar
sender_class = metlog.senders.zmq.ZmqPubSender
sender_bindstrs = tcp://127.0.0.1:5565
sender_queue_length = 5000
global_disabled_decorators = incr_count
```

Of all of these settings, only *sender_class* is strictly required. A detailed description of each option follows:

logger Each metlog message that goes out contains a *logger* value, which is simply a string token meant to identify the source of the message, usually the name of the application that is running. This can be specified separately for each message that is sent, but the client supports a default value which will be used for all messages that don't explicitly override. The *logger* config option specifies this default value. This value isn't strictly required, but if it is omitted '' (i.e. the empty string) will be used, so it is strongly suggested that a value be set.

severity Similarly, each metlog message specifies a *severity* value corresponding to the integer severity values defined by RFC 3164. And, again, while each message can set its own severity value, if one is omitted the client's default value will be used. If no default is specified here, the default default (how meta!) will be 6, "Informational".

disabled_timers Metlog natively supports "timer" behavior, which will calculate the amount of elapsed time taken by an operation and send that data along as a message to the back end. Each timer has a string token identifier. Because the act of calculating code performance actually impacts code performance, it is sometimes desirable to be able to activate and deactivate timers on a case by case basis. The *disabled_timers* value specifies a set of timer ids for which the client should NOT actually generate messages. Metlog will attempt to minimize the run-time impact of disabled timers, so the price paid for having deactivated timers will be very small. Note that the various timer ids should be newline separated.

sender_class This should be a Python dotted notation reference to a class (or factory function) for a Metlog "sender" object. A sender needs to provide a *send_message(msg)* method, which is responsible for serializing the message and passing it along to the router / back end / output mechanism / etc. metlog-py provides some development senders, but the main one it provides for intended production use makes use of ZeroMQ (using the pub/sub pattern) to broadcast the messages to any configured listeners.

sender_* As you might guess, different types of senders can require different configuration values. Any config options other than *sender_class* that start with *sender_* will be passed to the sender factory as keyword arguments, where the argument name is the option name minus the *sender_* component and the value is the specified value. In the example above, the ZeroMQ bind string and the queue length will be passed to the ZmqPubSender constructor.

global_* Any configuration value prefaced with *global_* represents an option that is global to all Metlog clients process-wide and not just the client being configured presently.

In addition to the main *metlog* section, any other config sections that start with *metlog_* (or whatever section name is specified) will be considered to be related to the metlog installation. Only specific variations of these are supported, however. The first of these is configuration for MetlogClient `api/filters`. Here is an example of such a configuration:

```
[metlog_filter_sev_max]
provider = metlog.filters.severity_max_provider
severity = 4

[metlog_filter_type_whitelist]
provider = metlog.filters.type_whitelist_provider
types = timer
        oldstyle
```

Each *metlog_filter_** section must contain a *provider* entry, which is a dotted name specifying a filter provider function. The rest of the options in that section will be converted into configuration parameters. The provider function will be called and passed the configuration parameters, returning a filter function that will be added to the client's filters. The filters will be applied in the order they are specified. In this case a "severity max" filter will be applied, so that only messages with a severity of 4 (i.e. "warning") or lower will actually be passed in to the sender. Additionally a "type whitelist" will be applied, so that only messages of type "timer" and "oldstyle" will be delivered.

1.2.2 plugins

Metlog allows you to bind new extensions onto the client through a plugin mechanism.

Each plugin must have a configuration section name with a prefix of *metlog_plugin_*. Configuration is parsed into a dictionary, passed into a configurator and then the resulting plugin method is bound to the client.

Each configuration section for a plugin must contain at least one option with the name *provider*. This is a dotted name for a function which will be used to configure a plugin. The return value for the provider is a configured method which will then be bound into the Metlog client.

Each plugin extension method has a canonical name that is bound to the metlog client as a method name. The suffix that follows the *metlog_plugin_* prefix is used only to distinguish logical sections for each plugin within the configuration file.

An example best demonstrates what can be expected. To load the dummy plugin, you need a *metlog_plugin_dummy* section as well as some configuration parameters. Here's an example

```
[metlog_plugin_dummysection]
provider=metlog.tests.plugin.config_plugin
port=8080
host=localhost
```

Once you obtain a reference to a client, you can access the new method.

```
from metlog.holder import CLIENT_HOLDER
client = CLIENT_HOLDER.get_client('your_app_name')
client.dummy('some', 'ignored', 'arguments', 42)
```

1.2.3 dictionary format

When using the *client_from_text_config* or *client_from_stream_config* functions of the config module to parse an ini format configuration, metlog-py simply converts these values to a dictionary which is then passed to *client_from_dict_config*. If you choose to not use the specified ini format, you can parse configuration yourself and call *client_from_dict_config* directly. The configuration specified in the "ini format" section above would be converted to the following dictionary:

```
{'logger': 'myapp',
 'severity': 4,
 'disabled_timers': ['foo', 'bar'],
 'sender': {'class': 'metlog.senders.zmq.ZmqPubSender',
            'bindstrs': 'tcp://127.0.0.1:5565',
            'queue_length': 5000,
  },
 'global': {'disabled_decorators': ['incr_count']},
 'filters': [('metlog.filters.severity_max',
              {'severity': 4},
              ),
             ('metlog.filters.type_whitelist',
              {'types': ['timer', 'oldstyle']},
              ),
 ],
 }
```

To manually load a Metlog client with plugins, the *client_from_dict_config* function allows you to pass in a list of plugin configurations using the *plugins* dict key, used in the same fashion as *filters* in the example directly above.

The configuration specified in the "plugins" section above would be converted into the following dictionary, where the key will be the name of the method bound to the client:

```
{'dummy': ('metlog.tests.plugin:config_plugin',
           {'port': 8080,
            'host': 'localhost'
            },
)
}
```

1.2.4 Debugging your configuration

You may find yourself with a metlog client which is not behaving in a manner that you expect. Metlog provides a deepcopy of the configuration that was used when the client was instantiated for debugging purposes.

The following code shows how you can verify that the configuration used is actually what you expect it to be

```
cfg = {'logger': 'addons-marketplace-dev',
       'sender': {'class': 'metlog.senders.UdpSender',
       'host': ['logstash1', 'logstash2'],
       'port': '5566'}}
client = client_from_dict_config(cfg)
assert client._config == json.dumps(cfg)
```

1.3 Config

This module provides helpers to handle MetlogClient configuration details.

metlog.config.**_convert**(*value*)

Converts a config value. Numeric integer strings are converted to integer values. 'True-ish' string values are converted to boolean True, 'False-ish' to boolean False. Any alphanumeric (plus underscore) value enclosed within ${dollar_sign_curly_braces} is assumed to represent an environment variable, and will be converted to the corresponding value provided by os.environ.

metlog.config.**client_from_dict_config**(*config*, *client=None*, *clear_global=False*)

Configure a metlog client, fully configured w/ sender and plugins.

Parameters

- **config** – Configuration dictionary.
- **client** – MetlogClient instance to configure. If None, one will be created.
- **clear_global** – If True, delete any existing global config on the CLIENT_HOLDER before applying new config.

The configuration dict supports the following values:

logger Metlog client default logger value.

severity Metlog client default severity value.

disabled_timers Sequence of string tokens identifying timers that are to be deactivated.

filters Sequence of 2-tuples *(filter_provider, config)*. Each *filter_provider* is a dotted name referring to a function which, when called and passed the associated *config* dict as kwargs, will return a usable MetlogClient filter function.

plugins Nested dictionary containing plugin configuration. Keys are the plugin names (i.e. the name the method will be given when attached to the client). Values are 2-tuples *(plugin_provider, config)*. Each *plugin_provider* is a dotted name referring to a function which, when called and passed the associated *config*, will return the usable plugin method.

sender Nested dictionary containing sender configuration.

global Dictionary to be applied to CLIENT_HOLDER's *global_config* storage. New config will overwrite any conflicting values, but will not delete other config entries. To delete, calling code should call the function with *clear_global* set to True.

All of the configuration values are optional, but failure to include a sender may result in a non-functional Metlog client. Any unrecognized keys will be ignored.

Note that any top level config values starting with *sender_* will be added to the *sender* config dictionary, overwriting any values that may already be set.

The sender configuration supports the following values:

class (required) Dotted name identifying the sender class to instantiate.

args Sequence of non-keyword args to pass to sender constructor.

<kwargs> All remaining key-value pairs in the sender config dict will be passed as keyword arguments to the sender constructor.

`metlog.config.`**`client_from_stream_config`**(*stream*, *section*, *client=None*, *clear_global=False*)

Extract configuration data in INI format from a stream object (e.g. a file object) and use it to generate a Metlog client. Config values will be sent through the *_convert* function for possible type conversion.

Parameters

- **stream** – Stream object containing config information.
- **section** – INI file section containing the configuration we care about.
- **client** – MetlogClient instance to configure. If None, one will be created.

Note that all sender config options should be prefaced by "**sender_**", e.g. "sender_class" should specify the dotted name of the sender class to use. Similarly all extension method settings should be prefaced by "**extensions_**". Any values prefaced by "**global_**" will be added to the global config dictionary.

`metlog.config.`**`client_from_text_config`**(*text*, *section*, *client=None*, *clear_global=False*)

Extract configuration data in INI format from provided text and use it to configure a Metlog client. Text is converted to a stream and passed on to *client_from_stream_config*.

Parameters

- **text** – INI text containing config information.
- **section** – INI file section containing the configuration we care about.
- **client** – MetlogClient instance to configure. If None, one will be created.

`metlog.config.`**`dict_from_stream_config`**(*stream*, *section*)

Parses configuration from a stream and converts it to a dictionary suitable for passing to *client_from_dict_config*.

Parameters

- **stream** – Stream object containing config information.

- **section** – INI file section containing the configuration we care about.

metlog.config.**nest_prefixes**(*config_dict*, *prefixes=None*, *separator='_'*)

Iterates through the *config_dict* keys, looking for any starting w/ one of a specific set of prefixes, moving those into a single nested dictionary keyed by the prefix value.

Parameters

- **config_dict** – Dictionary to mutate. Will also be returned.
- **prefixes** – Sequence of prefixes to look for in *config_dict* keys.
- **separator** – String which separates prefix values from the rest of the key.

1.4 MetlogClient

class metlog.client.**MetlogClient**(*sender*, *logger*, *severity=6*, *disabled_timers=None*, *filters=None*)

Client class encapsulating metlog API, and providing storage for default values for various metlog call settings.

add_method(*method*, *override=False*)

Add a custom method to the MetlogClient instance.

Parameters

- **method** – Callable that will be used as the method.
- **override** – Set this to the method name you want to override. False indicates no override will occur.

critical(*msg*, **args*, ***kwargs*)

Log a CRITICAL level message

debug(*msg*, **args*, ***kwargs*)

Log a DEBUG level message

env_version = '0.8'

error(*msg*, **args*, ***kwargs*)

Log an ERROR level message

exception(*msg*, *exc_info=True*, **args*, ***kwargs*)

Log an ALERT level message

incr(*name*, *count=1*, *logger=None*, *severity=None*, *fields=None*, *rate=1.0*)

Sends an 'increment counter' message.

Parameters

- **name** – String label for the counter.
- **count** – Integer amount by which to increment the counter.
- **logger** – String token identifying the message generator.
- **severity** – Numerical code (0-7) for msg severity, per RFC 5424.
- **fields** – Arbitrary key/value pairs for add'l metadata.

info(*msg*, **args*, ***kwargs*)

Log an INFO level message

`is_active`
Is this client ready to transmit messages? For now we assume that if the default sender (i.e. *NoSendSender*) has been replaced then we're good to go.

`metlog` (*type*, *logger=None*, *severity=None*, *payload=''*, *fields=None*)
Create a single message and pass it to the sender for delivery.

Parameters

- **type** – String token identifying the type of message payload.
- **logger** – String token identifying the message generator.
- **severity** – Numerical code (0-7) for msg severity, per RFC 5424.
- **payload** – Actual message contents.
- **fields** – Arbitrary key/value pairs for add'l metadata.

`send_message` (*msg*)
Apply any filters and, if required, pass message along to the sender for delivery.

`setup` (*sender=None*, *logger=''*, *severity=6*, *disabled_timers=None*, *filters=None*)

Parameters

- **sender** – A sender object used for actual message delivery.
- **logger** – Default *logger* value for all sent messages.
- **severity** – Default *severity* value for all sent messages.
- **disabled_timers** – Sequence of string tokens identifying timers that should be deactivated.
- **filters** – A sequence of filter callables.

`timer` (*name*, *logger=None*, *severity=None*, *fields=None*, *rate=1.0*)
Return a timer object that can be used as a context manager or a decorator, generating a metlog 'timer' message upon exit.

Parameters

- **name** – Required string label for the timer.
- **logger** – String token identifying the message generator.
- **severity** – Numerical code (0-7) for msg severity, per RFC 5424.
- **fields** – Arbitrary key/value pairs for add'l metadata.
- **rate** – Sample rate, btn 0 & 1, inclusive (i.e. .5 = 50%). Sample rate is enforced in this method, i.e. if a sample rate is used then some percentage of the timers will do nothing.

`timer_send` (*name*, *elapsed*, *logger=None*, *severity=None*, *fields=None*, *rate=1.0*)
Converts timing data into a metlog message for delivery.

Parameters

- **name** – Required string label for the timer.
- **elapsed** – Elapsed time of the timed event, in ms.
- **logger** – String token identifying the message generator.
- **severity** – Numerical code (0-7) for msg severity, per RFC 5424.
- **fields** – Arbitrary key/value pairs for add'l metadata.

- **rate** – Sample rate, btn 0 & 1, inclusive (i.e. .5 = 50%). Sample rate is *NOT* enforced in this method, i.e. all messages will be sent through to metlog, sample rate is purely informational at this point.

warn(*msg*, **args*, ***kwargs*)
Log a WARN level message

class metlog.client.**SEVERITY**
Put a namespace around RFC 3164 syslog messages

ALERT = 1

CRITICAL = 2

DEBUG = 7

EMERGENCY = 0

ERROR = 3

INFORMATIONAL = 6

NOTICE = 5

WARNING = 4

1.5 Senders

1.5.1 Development

class metlog.senders.dev.**DebugCaptureSender**(***kwargs*)
Capture up to 100 metlog messages in a circular buffer for inspection later. This is only for DEBUGGING. Do not use this for anything except development.

__weakref__
list of weak references to the object (if defined)

send_message(*msg*)
JSONify and append to the circular buffer.

class metlog.senders.dev.**FileSender**(*filepath*, **args*, ***kwargs*)
Emits messages to a filesystem file.

class metlog.senders.dev.**StdOutSender**(**args*, ***kwargs*)
Emits metlog messages to stdout.

class metlog.senders.dev.**StreamSender**(*stream*, *formatter=None*)
Emits messages to a provided stream object.

__init__(*stream*, *formatter=None*)

Parameters

- **stream** – Stream object to which the messages should be written.
- **formatter** – Optional callable (or dotted name identifier) that accepts a msg dictionary and returns a formatted string to be written to the stream.

__weakref__
list of weak references to the object (if defined)

default_formatter(*msg*)
Default formatter, just converts the message to 4-space-indented JSON.

send_message(*msg*)
Deliver message to the stream object.

1.5.2 ZeroMQ

class metlog.senders.zmq.**Pool**(*client_factory*, *size=10*, *livecheck=10*)
This is a threadsafe pool of 0mq clients.

Parameters

- **client_factory** – a factory function that creates Client instances
- **size** – The number of clients to create in the pool
- **livecheck** – The time in seconds to wait to ping the server from each client

__weakref__
list of weak references to the object (if defined)

send(*msg*)
Threadsafely send a single text message over a 0mq socket

start_reconnecting()
Start the background thread that handles pings to the server to synchronize the initial pub/sub

stop()
Shutdown the background reconnection thread

class metlog.senders.zmq.**ZmqHandshakePubSender**(*handshake_bind*, *connect_bind*, *handshake_timeout*, *pool_size=10*, *hwm=200*, *livecheck=10*, *debug_stderr=False*)
Sends metlog messages out via a ZeroMQ publisher socket.

Redirect all dropped messages to stderr

__init__(*handshake_bind*, *connect_bind*, *handshake_timeout*, *pool_size=10*, *hwm=200*, *livecheck=10*, *debug_stderr=False*)

Parameters

- **handshake_bind** – A single 0mq recognized endpoint URL. This should point to the endpoint for handshaking of connections
- **connect_bind** – A single 0mq recognized endpoint URL. This should point ot the endpoint for sending actual Metlog messages.
- **handshake_timeout** – Timeout in ms to wait for responses from the 0mq server on handshake
- **pool_size** – The number of connections we maintain to the 0mq backend
- **hwm** – High water mark. Set the maximum number of messages to queue before dropping messages in case of a slow reading 0mq server.
- **livecheck** – Polling interval in seconds between client.connect() calls
- **debug_stderr** – Boolean flag to send messages to stderr in addition to the actual 0mq socket

class `metlog.senders.zmq.`**`ZmqPubSender`**(*bindstrs*, *pool_size=10*, *queue_length=1000*, *livecheck=10*, *debug_stderr=False*)

Sends metlog messages out via a ZeroMQ publisher socket.

`__init__`(*bindstrs*, *pool_size=10*, *queue_length=1000*, *livecheck=10*, *debug_stderr=False*)

Parameters

- **bindstrs** – One or more URL strings which 0mq recognizes as an endpoint URL. Either a string or a list of strings is accepted.
- **pool_size** – The number of connections we maintain to the 0mq backend
- **livecheck** – Polling interval in seconds between client.connect() calls
- **debug_stderr** – Boolean flag to send messages to stderr in addition to the actual 0mq socket

class `metlog.senders.zmq.`**`ZmqSender`**

Base class for ZmqPubSender and ZmqHandshakePubSender

static **`__new__`**(**args*, ***kwargs*)

Just check that we have pyzmq installed

`__weakref__`

list of weak references to the object (if defined)

`send_message`(*msg*)

Serialize and send a message off to the metlog listener.

Parameters **msg** – Dictionary representing the message.

1.5.3 UDP

class `metlog.senders.udp.`**`UdpSender`**(*host*, *port*)

Sends metlog messages out via a UDP socket.

`__init__`(*host*, *port*)

Create UdpSender object.

Parameters

- **host** – A string or sequence of strings representing the hosts to which messages should be delivered.
- **port** – An integer or sequence of integers representing the ports to which the messages should be delivered. Will be zipped w/ the provided hosts to generate host/port pairs. If there are extra hosts, the last port in the sequence will be repeated for each extra host. If there are extra ports they will be truncated and ignored.

`__weakref__`

list of weak references to the object (if defined)

`send_message`(*msg*)

Serialize and send a message off to the metlog listener(s).

Parameters **msg** – Dictionary representing the message.

1.6 Decorators

1.6.1 base

This module contains a Metlog decorator base class and some additional helper code. The primary reason for these abstractions is 'deferred configuration'. Decorators are evaluated by Python at import time, but often the configuration needed for a Metlog client, which might negate (or change) the behavior of a Metlog decorator, isn't available until later, after some config parsing code has executed. This code provides a mechanism to have a function get wrapped in one way (or not at all) when the decorator is originally evaluated, but then to be wrapped differently once the config has loaded and the desired final behavior has been established.

`metlog.decorators.base.`**`MetlogDecorator`**

This is a base class for Metlog decorators, designed to support 'rebinding' of the actual decorator method once Metlog configuration has actually been loaded. The first time the decorated function is invoked, the *predicate* method will be called. If the result is True, then *metlog_call* (intended to be implemented by subclasses) will be used as the decorator. If the *predicate* returns False, then *_invoke* (which by default does nothing but call the wrapped function) will be used as the decorator.

1.6.2 util

`metlog.decorators.util.`**`return_fq_name`**(*func*, *klass=None*)

Resolve a fully qualified name for a function or method

1.6.3 stats

`metlog.decorators.stats.`**`incr_count`**

Lazily decorate any callable w/ a wrapper that will increment a metlog counter whenever the callable is invoked.

`metlog.decorators.stats.`**`timeit`**

Lazily decorate any callable with a metlog timer.

1.7 Exceptions

exception `metlog.exceptions.`**`EnvironmentNotFoundError`**(*varname*)

Raised when an environment variable is not found

exception `metlog.exceptions.`**`MethodNotFoundError`**(*msg=''*)

Raised when a method lookup fails

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index