metlog-node Documentation

Release 0.1

Rob Miller, Victor Ng

July 01, 2016

Contents

metlog-node is a Node.js 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-node library you are currently reading about is a client library meant to be used by Node.js-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-node library, the `api/config` config class which exposes a factory function to generate a configured client, and the various `api/senders` classes, one of which must be provided to the factory function and which handles the actual delivery of the message to the router component.

The MetlogClient must be instantiated with the factory functions. The raw class definition is not exposed through the public api.

Folks new to using Metlog will probably find Metlog Configuration a good place to get started.

CHAPTER 1

Welcome to metlog-node's documentation!

Contents:

1.1 Getting Started

There are two primary components with which users of the metlog-node library should be aware. The first is the `api/config` clientFromJsonConfig factory function.

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-node, 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.

1.2 Metlog Configuration

To assist with getting a working Metlog set up, metlog-node provides a `api/config` module which will take declarative configuration info in JSON format and use it to configure a MetlogClient instance.

1.2.1 JSON format

The *clientFromJsonConfig* function of the config module is used to create a MetlogClient instance.

A minimal configuration that will instantiate a working Metlog client may look like this

```
var metlog = require('metlog');
var METLOG_CONF = {
    'sender': {'factory': 'metlog/Senders:udpSenderFactory',
               'hosts': '192.168.20.2',
               'ports': 5565},
};
var jsonConfig = JSON.stringify(METLOG_CONF);
var log_client = metlog.clientFromJsonConfig(jsonConfig);
```

There are several optional parameters you may use to specialize the metlog-node client. 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".

disabledTimers 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 *disabledTimers* 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.

filters You can configure client side filters to restrict messages from going to the server.

The following snippet demonstrates settings all optional parameters in the metlog client

```
var config = {
    'sender': {'factory': './example/config_imports:makeMockSender' },
    'logger': 'test',
    'severity': metlog.SEVERITY.INFORMATIONAL,
    'disabledTimers': ['disabled_timer_name'],
    'filters': [['./example/config_imports:payloadIsFilterProvider' , {'payload': 'nay!'}]],
    'plugins': {'showLogger': {'provider': './example/config_imports:showLoggerProvider',
                               'label': 'some-label-thing' }}
};
var jsonConfig = JSON.stringify(config);
var client = metlog.clientFromJsonConfig(jsonConfig);
```

You can find more runnable code samples at http://github.com/mozilla-services/metlog-node/ in the examples subdirectory.

1.3 Using the metlog client

Basic usage of the metlog client for timing functions is straight forward. Just decorate your function with client.timer and you'll get timing events

```
var name = 'decorator';
var minWait = 40;  // in milliseconds
var sleeper = function() {
    block(minWait);
};

// wrap it
sleeper = client.timer(sleeper, name)

// call it
sleeper();
```

```
```

There are several options you can set in the timer like timestamp, logger name, severity, sampling rate and a fields dictionary. None of those are required to get started though.

To send an increment event, you can call the incr() method. The only required field is the name of the increment event

```
var name = "some_name";
client.incr(name);
```

You may optionally set an options dictionary to increment by values other than 1 and you can also change the sample_rate. The following snippets increments by 2, but only samples half the time on the client side.

```
var name = "some_name";
client.incr(name, {'count': 2}, 0.5);
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search