Datadog API Documentation

Release 1.0.0

Datadog, Inc.

March 25, 2016

The *dogapi* module provides DogHttpApi - a simple wrapper around DataDog's HTTP API - and *DogStatsApi* - a tool for collecting metrics in high performance applications.

Installation

The module can be downloaded from PyPI and installed in one step with easy_install:

```
>>> sudo easy_install dogapi
```

Or with pip:

```
>>> sudo pip install dogapi
```

To install from source, download a distribution and run:

```
>>> sudo python setup.py install
```

If you use virtualenv you do not need to use sudo.

CHAPTER 2

DogHttpApi

DogHttpApi is a Python client library for DataDog's HTTP API.

class `dogapi.http.`**`DogHttpApi`**(*api_key=None*, *application_key=None*, *api_version='v1'*, *api_host=None*, *timeout=2*, *max_timeouts=3*, *backoff_period=300*, *swallow=True*, *use_ec2_instance_id=False*, *json_responses=False*)

A high-level client for interacting with the Datadog API.

By default, service calls to the simple client silently swallow any exceptions before they escape from the library. To disable this behavior, simply set the *swallow* attribute of your `Datadog` instance to *False*.

The default timeout is 2 seconds, but that can be changed by setting the client's *timeout* attribute.

`add_tags`(*host_id*[, *tag1*, *tag2*, *...*])

Add one or more tags to a host.

```
>>> dog_http_api.add_tags(host_id, ['env:test'])
>>> dog_http_api.add_tags(host_id, ['env:test', 'database'])
```

`alert`(*query*, *name=None*, *message=None*, *silenced=False*, *notify_no_data=None*, *timeout_h=None*)

Create a new metric alert for the given *query*. If *name* is unset, the alert will be given a name based on the query. The *message* will accompany any notifications sent for the alert and can contain the same '@' notation as events to alert individual users. The *silenced* flag controls whether or not notifications are sent for alert state changes.

```
>>> dog_http_api.alert("sum(last_1d):sum:system.net.bytes_rcvd{host:host0} > 100")
```

`all_tags`(*source=None*)

Get a list of tags for your org and their member hosts.

```
>>> dog_http_api.all_tags()
[ { 'tag1': [ 'host1', 'host2', ... ] }, ... ]
```

`cancel_downtime`(*downtime_id*)

Cancel the downtime identified by *downtime_id*

`change_tags`(*host_id*[, *tag1*, *tag2*, *...*])

Replace a host's tags with one or more new tags.

```
>>> dog_http_api.change_tags(host_id, ['env:test'])
>>> dog_http_api.change_tags(host_id, ['env:test', 'database'])
```

`comment`(*handle*, *message*, *comment_id=None*, *related_event_id=None*)

Post a comment *message* as the user with *handle*. Edit a comment by including it's *comment_id*. Reply to a related event by setting the *related_event_id*.

```
>>> dog_http_api.comment("matt", "Hey! Something strange is going on.")
```

create_dashboard(*title*, *description*, *graphs*, *template_variables=None*)
Create a new dashboard with the given *title*, *description* and *graphs*.

See the dashboard API documentation for the dashboard data format.

create_screenboard(*description*)
Create a new Screenboard.

See the screenboard API documentation for the Screenboard *description* format.

dashboard(*dash_id*)
Return the dashboard with the given id.

See the dashboard API documentation for the dashboard data format.

dashboards()
Return all of your dashboards.

See the dashboard API documentation for the dashboard data format.

delete_alert(*alert_id*)
Delete the metric alert identified by *alert_id*.

```
>>> dog_http_api.delete_alert(1234)
```

delete_comment(*comment_id*)
Delete a comment with the given *comment_id*.

```
>>> dog_http_api.delete_comment('1234')
```

delete_dashboard(*dash_id*)
Delete the dashboard with the given *dash_id*.

```
>>> dog_http_api.delete_dashboard(dash_id)
```

delete_monitor(*monitor_id*)
Delete the monitor identified by *monitor_id*.

```
>>> dog_http_api.delete_monitor(1234)
```

delete_screenboard(*board_id*)
Delete the Screenboard with the given id.

detach_tags(*host_id*, *source=None*)
Remove all tags from a host.

```
>>> dog_http_api.detach_tags(123)
```

event(**args*, ***kwargs*)
Post an event.

Parameters

- **title** (*string*) – title for the new event
- **text** (*string*) – event message
- **date_happened** (*integer*) – when the event occurred. if unset defaults to the current time. (POSIX timestamp)
- **handle** (*string*) – user to post the event as. defaults to owner of the application key used to submit.

- **priority** (*string*) – priority to post the event as. ("normal" or "low", defaults to "normal")
- **related_event_id** (*id*) – post event as a child of the given event
- **tags** (*list of strings*) – tags to post the event with
- **host** (*list of strings*) – host to post the event with
- **device_name** (*list of strings*) – device_name to post the event with

Returns new event id

Return type integer

get_alert (*alert_id*)
Get the details for the metric alert identified by *alert_id*.

```
>>> dog_http_api.get_alert(1234)
```

get_all_alerts ()
Get the details for all metric alerts.

```
>>> dog_http_api.get_all_alert()
```

get_all_downtimes (*current_only=False*)
List all scheduled downtimes.

get_all_monitors (*group_states=None*, *tags=None*)
Get the details for all monitors. If *include_state* is set to True then the response will include the state of each active group in the alert.

group_states is optionally a list of statuses chosen from "all", "ok", "warn", "alert", "no data". For example, if you want only the failing groups then you would set it to ['alert', 'warn']. If no value is given then no group states will be returned.

tags is optionally a list of scope tags that will be used to filter the list of monitors returned. If no value is given, then all monitors, regardless of scope, will be returned.

```
>>> dog_http_api.get_all_monitors(group_states=['alert'], tags=['host:myhost'])
```

get_all_screenboards ()
Get the Screenboard with the given id.

See the screenboard API documentation for the data format.

get_downtime (*downtime_id*)
Get the downtime identified by *downtime_id*

get_event (*id*)
Get details for an individual event with the given *id*.

See the event API documentation for the event data format.

```
>>> dog_http_api.get_event("event-1")
{
  "id": "event-1",
  "title": "my first event",
  "priority": "normal",
  "handle": "alq@datadoghq.com",
  "date_happened": 1313769783,
  "source": "nagios",
  "alert_type": "ok",
  "is_aggregate": true,
          "children": [
            {
              "id": "event-100",
              "date_happened": 123459833,
              "alert_type": "error"
            }, ...
          ]
        }
```

get_monitor(*monitor_id*, *group_states=None*)
Get the details for the monitor identified by *monitor_id*.

group_states is optionally a list of statuses chosen from "all", "ok", "warn", "alert", "no data". For example, if you want only the failing groups then you would set it to ['alert', 'warn']. If no value is given then no group states will be returned.

```
>>> dog_http_api.get_monitor(1234, group_states=['all'])
```

get_screenboard(*board_id*)
Get the Screenboard with the given id.

See the screenboard API documentation for the Screenboard *description* format.

graph_snapshot(*metric_query*, *start*, *end*, *event_query=None*)
Take a snapshot of a graph, returning the full url to the snapshot. Values for *start* and *end* are given in seconds since the epoch. An optional event query can be provided to overlay events on the graph.

```
>>> end = int(time.time())
>>> start = end - 60 * 60
>>> dog_http_api.snapshot("system.load.1{*}", start, end)
```

graph_snapshot_from_def(*graph_def*, *start*, *end*)
Take a snapshot of a graph from a graph definition, returning the full url to the snapshot. Values for *start* and *end* are given in seconds since the epoch.

```
>>> end = int(time.time())
>>> start = end - 60 * 60
>>> graph_def = json.dumps({
    "requests": [{
        "q": "system.load.1{*}"
    }, {
        "q": "system.load.5{*}"
    }],
    "viz": "timeseries",
    "events": [{
        "q": "*"
    }]
})
>>> dog_http_api.snapshot(graph_def, start, end)
```

host_tags(*host_id*, *source=None*, *by_source=False*)
Get a list of tags for the specified host by name or id.

```
>>> dog_http_api.host_tags('web.example.com')
['web', 'env:production']
>>> dog_http_api.host_tags(1234)
['database', 'env:test']
```

invite(*emails*)
Send an invite to join datadog to each of the email addresses in the *emails* list. If *emails* is a string, it will

be wrapped in a list and sent. Returns a list of email addresses for which an email was sent.

```
>>> dog_http_api.invite(["user1@mydomain.com", "user2@mydomain.com"])
```

metric(*name*, *points*, *host=None*, *device=None*, *tags=None*, *metric_type='gauge'*)
Submit a point or series of *points* to the metric API, optionally specifying a *host* or *device*. Points can either be a value, a tuple of POSIX timestamps and a value, or a list of timestamp value pairs.

```
>>> dog_http_api.metric('my.value', 123.4, host="my.custom.host")
>>> dog_http_api.metric('my.pair', (1317652676, 15), device="eth0")
>>> dog_http_api.metric('my.series', [(1317652676, 15), (1317652800, 16)])
```

metrics(*metrics*)
Submit a list of *metrics* with 1 or more data points to the metric API. Each metric is a dictionary that includes the fields metric_name, points and optionally, host and device to scope the metric.

```
>>> dog_http_api.metrics([{'metric':'my.metric', 'points':[(1317652676, 15)]}])
```

monitor(*mtype*, *query*, *name=None*, *message=None*, *options=None*)
Create a new monitor of type *mtype* with the given *name* and *query*. The *message* will accompany any notifications sent for the alert and can contain the same '@' notation as events to alert individual users. The *options* argument is a dictionary of settings for the monitor. See the Datadog API documentation for a break down of available options.

```
>>> dog_http_api.monitor("metric alert", "sum(last_1d):sum:system.net.bytes_rcvd{host:host0}
```

mute_alerts()
Mute all alerts.

```
>>> dog_http_api.mute_alerts()
```

mute_monitor(*monitor_id*, *scope=None*, *end=None*)
Mute the monitor identified by *monitor_id*. If a *scope* is given your mute will just apply to that scope. You can give an *end* argument that is a POSIX timestamp of when the mute should stop.

```
>>> dog_http_api.mute_monitor(1234, scope='env:staging')
```

mute_monitors()
Mute all monitors.

```
>>> dog_http_api.mute_monitors()
```

schedule_downtime(*scope*, *start=None*, *end=None*, *message=None*)
Schedule downtime over *scope* from *start* to *end*, where *start* and *end* are POSIX timestamps. If *start* is omitted, the downtime will begin immediately. If *end* is omitted, the downtime will continue until cancelled.

search(*query*)
Search datadog for hosts and metrics by name. The search *query* can be faceted to limit the results (e.g. `"hosts:foo"`, or `"metrics:bar"`) or un-faceted, which will return results of all types (e.g. `"baz"`). Return a dictionary mapping each queried facet to a list of name strings.

```
>>> dog_http_api.search("cassandra")
{ "results": {
    "hosts": ["cassandraHostA", "cassandraHostB", ...],
    "metrics": ["cassandra.load", "cassandra.requests", ...]
  }
}
```

share_screenboard(*board_id*)

Share the screenboard with given id. Returns the structure:

```
>>> dog_http_api.share_screenboard(1234)
{
    "board_id": 1234,
    "public_url": "https://p.datadoghq.com/sb/48f234f"
}
```

snapshot_status(*snapshot_url*)

Returns the status code of snapshot. Can be used to know when the snapshot is ready for download.

Example usage:

>> snap = dog_http_api.snapshot(metric_query, start, end) >> snapshot_url = snap['snapshot_url'] >> while snapshot_status(snapshot_url) != 200: >> time.sleep(1) >> img = urllib.urlopen(snapshot_url)

stream(*start*, *end*, *priority=None*, *sources=None*, *tags=None*)

Get the events that occurred between the *start* and *end* POSIX timestamps, optional filtered by *priority* ("low" or "normal"), *sources* and *tags*.

See the event API documentation for the event data format.

```
>>> dog_http_api.stream(1313769783, 131378000, sources=["nagios"])
{ "events": [
    {
      "id": "event-1",
      "title": "my first event",
      "priority": "normal",
      "handle": "alq@datadoghq.com",
      "date_happened": 1313769783,
      "source": "nagios",
      "alert_type": "ok",
      "is_aggregate": true,
      "children": [
        {
          "id": "event-100",
          "date_happened": 123459833,
          "alert_type": "error"
        }, ...
      ]
    }, ...
  ]
}
```

unmute_alerts()

Unmute all alerts.

```
>>> dog_http_api.unmute_alerts()
```

unmute_monitor(*monitor_id*, *scope=None*)

Unmute the monitor identified by *monitor_id*. If a *scope* is given your unmute will just apply to that scope.

```
>>> dog_http_api.unmute_monitors(1234, scope='env:staging')
```

unmute_monitors()

Unmute all monitors.

```
>>> dog_http_api.unmute_monitors()
```

update_alert(*alert_id*, *query*, *name=None*, *message=None*, *silenced=False*, *notify_no_data=None*, *timeout_h=None*, *silenced_timeout_ts=None*)

Update the metric alert identified by *alert_id* with the given *query*. If *name* is unset, the alert will be given a name based on the query. The *message* will accompany any notifications sent for the alert and can contain the same '@' notation as events to alert individual users. The *silenced* flag controls whether or not notifications are sent for alert state changes.

```
>>> dog_http_api.update_alert(1234, "sum(last_1d):sum:system.net.bytes_rcvd{host:host0} > 10
```

update_dashboard(*dash_id*, *title*, *description*, *graphs*, *template_variables=None*)

Update the dashboard whose id is *dash_id*, replacing it's *title*, *description* and *graphs*. Return the dashboard with the given id.

See the dashboard API documentation for the dashboard data format.

update_downtime(*downtime_id*, *scope=None*, *start=None*, *end=None*, *message=None*)

Update downtime parameters.

update_monitor(*monitor_id*, *query*, *name=None*, *message=None*, *options=None*)

Update the monitor identified by *monitor_id* with the given *query*. The *message* will accompany any notifications sent for the alert and can contain the same '@' notation as events to alert individual users. The *options* argument is a dictionary of settings for the monitor. See the Datadog API documentation for a break down of available options.

```
>>> dog_http_api.update_monitor(1234, "sum(last_1d):sum:system.net.bytes_rcvd{host:host0} >
```

update_screenboard(*board_id*, *description*)

Update the Screenboard with the given id.

See the screenboard API documentation for the Screenboard *description* format.

CHAPTER 3

DogStatsApi

DogStatsApi is a tool for collecting application metrics without hindering performance. It collects metrics in the application thread with very little overhead and allows flushing metrics in process, in a thread or in a greenlet, depending on your application's needs.

class `dogapi.stats.`**`DogStatsApi`**

`start`(*api_key=api_key*, *flush_interval=10*, *flush_in_thread=True*, *flush_in_greenlet=False*, *disabled=False*)

Begin flushing metrics with your account's *api_key* every *flush_interval* seconds. By default, metrics will be flushed in a seperate thread.

```
>>> dog_stats_api.start(api_key='my_api_key')
```

If you're running a gevent server and want to flush metrics in a greenlet, set *flush_in_greenlet* to True. Be sure to import and monkey patch gevent before starting dog_stats_api.

```
>>> from gevent import monkey; monkey.patch_all()
>>> dog_stats_api.start(api_key='my_api_key', flush_in_greelet=True)
```

If you'd like to flush metrics in process, set *flush_in_thread* to False, though you'll have to call *flush* manually to post metrics to the server.

```
>>> dog_stats_api.start(api_key='my_api_key', flush_in_thread=False)
```

If for whatever reason, you need to disable metrics collection in a hurry, set *disabled* to True and metrics won't be collected or flushed.

```
>>> dog_stats_api.start(api_key='my_api_key', disabled=True)
```

`gauge`(*metric_name*, *value*, *timestamp=None*, *tags=None*, *sample_rate=1*, *host=None*)

Record the current *value* of a metric. They most recent value in a given flush interval will be recorded. Optionally, specify a set of tags to associate with the metric. This should be used for sum values such as total hard disk space, process uptime, total number of active users, or number of rows in a database table.

```
>>> dog_stats_api.gauge('process.uptime', time.time() - process_start_time)
>>> dog_stats_api.gauge('cache.bytes.free', cache.get_free_bytes(), tags=['version:1.0'])
```

`increment`(*metric_name*, *value=1*, *timestamp=None*, *tags=None*, *sample_rate=1*, *host=None*)

Increment the counter by the given *value*. Optionally, specify a list of *tags* to associate with the metric. This is useful for counting things such as incrementing a counter each time a page is requested.

```
>>> dog_stats_api.increment('home.page.hits')
>>> dog_stats_api.increment('bytes.processed', file.size())
```

histogram(*metric_name*, *value*, *timestamp=None*, *tags=None*, *sample_rate=1*, *host=None*)

Sample a histogram value. Histograms will produce metrics that describe the distribution of the recorded values, namely the minimum, maximum, average, count and the 75th, 85th, 95th and 99th percentiles. Optionally, specify a list of *tags* to associate with the metric.

```
>>> dog_stats_api.histogram('uploaded_file.size', uploaded_file.size())
```

timed(*metric_name*, *sample_rate=1*, *tags=None*, *host=None*)

A decorator that will track the distribution of a function's run time. Optionally specify a list of tags to associate with the metric.

```
@dog_stats_api.timed('user.query.time')
def get_user(user_id):
    # Do what you need to ...
    pass

# Is equivalent to ...
start = time.time()
try:
    get_user(user_id)
finally:
    dog_stats_api.histogram('user.query.time', time.time() - start)
```

flush(*timestamp=None*)

Flush and post all metrics to the server. Note that this is a blocking call, so it is likely not suitable for user facing processes. In those cases, it's probably best to flush in a thread or greenlet.

dogapi.**dog_stats_api**

A global `DogStatsApi` instance that is easily shared across an application's modules. Initialize this once in your application's set-up code and then other modules can import and use it without further configuration.

```
>>> from dogapi import dog_stats_api
>>> dog_stats_api.start(api_key='my_api_key')
```

Here's an example that put's it all together.

```
# Import the dog stats instance.
from dogapi import dog_stats_api as dog

# Begin flushing asynchronously with the given api key. After this is done
# once in your application, other modules can import and use dog_stats_api
# without any further configuration.
dog.start(api_key='my_api_key')


@dog.timed('home_page.render.time')
def render_home_page(user_id):
    """ Render the home page for the given user. """

    # Fetch the user from the cache or the database
    # and record metrics on our cache hits and misses.
    user = user_cache.get(user_id)
    if user:
        dog.increment('user_cache.hit')
    else:
        dog.increment('user_cache.miss')
        user = user_database.get(user_id)

    return render('home.html', user_id)
```

CHAPTER 4

Source

The DogApi source is freely available on Github. Check it out here.

CHAPTER 5

Get in Touch

If you'd like to suggest a feature or report a bug, please add an issue here. If you want to talk about DataDog in general, reach out at datadoghq.com.

Index