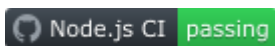

hot-shots

A Node.js client for Etsy's StatsD server, Datadog's DogStatsD server, and InfluxDB's Telegraf StatsD server.

This project was originally a fork off of node-statsd. This project includes all changes in the latest node-statsd and many additional changes, including: * TypeScript types * Telegraf support * events * child clients * tcp protocol support * uds (Unix domain socket) protocol support * raw stream protocol support * mock mode * asyncTimer * asyncDistTimer * much more, including many bug fixes

hot-shots supports Node 10.x and higher.



Usage

All initialization parameters are optional.

Parameters (specified as one object passed into hot-shots):

- **host**: The host to send stats to, if not set, the constructor tries to retrieve it from the `DD_AGENT_HOST` environment variable, **default**: `'undefined'` which as per UDP/data-gram socket docs results in `127.0.0.1` or `::1` being used.
- **port**: The port to send stats to, if not set, the constructor tries to retrieve it from the `DD_DOGSTATSD_PORT` environment variable, **default**: `8125`
- **prefix**: What to prefix each stat name with **default**: `''`. Note prefix separator must be specified explicitly if desired (e.g. `my_prefix.`).
- **suffix**: What to suffix each stat name with **default**: `''`. Note suffix separator must be specified explicitly if desired (e.g. `.my_suffix`).
- **tagPrefix**: Prefix tag list with character **default**: `'#'`. Note does not work with `telegraf` option.
- **tagSeparator**: Separate tags with character **default**: `','`. Note does not work with `telegraf` option.
- **globalize**: Expose this StatsD instance globally. **default**: `false`
- **cacheDns**: Caches dns lookup to *host* for *cacheDnsTtl*, only used when protocol is `udp`, **default**: `false`
- **cacheDnsTtl**: time-to-live of dns lookups in milliseconds, when *cacheDns* is enabled. **default**: `60000`
- **mock**: Create a mock StatsD instance, sending no stats to the server and allowing data to be read from `mockBuffer`. Note that `mockBuffer` will keep growing, so only use for testing or clear out periodically. **default**: `false`

-
- **globalTags**: Tags that will be added to every metric. Can be either an object or list of tags. **default**: `{}`. The following *Datadog* tags are appended to **globalTags** from the corresponding environment variable if the latter is set:
 - `dd.internal.entity_id` from `DD_ENTITY_ID` (docs)
 - `env` from `DD_ENV` (docs)
 - `service` from `DD_SERVICE` (docs)
 - `version` from `DD_VERSION` (docs)
 - **maxBufferSize**: If larger than 0, metrics will be buffered and only sent when the string length is greater than the size. **default**: 0
 - **bufferFlushInterval**: If buffering is in use, this is the time in ms to always flush any buffered metrics. **default**: 1000
 - **telegraf**: Use Telegraf's StatsD line protocol, which is slightly different than the rest **default**: **false**
 - **sampleRate**: Sends only a sample of data to StatsD for all StatsD methods. Can be overridden at the method level. **default**: 1
 - **errorHandler**: A function with one argument. It is called to handle various errors. **default**: `none`, errors are thrown/logger to console
 - **useDefaultRoute**: Use the default interface on a Linux system. Useful when running in containers
 - **protocol**: Use `tcp` option for TCP protocol, or `uds` for the Unix Domain Socket protocol or `stream` for the raw stream. Defaults to `udp` otherwise.
 - **path**: Used only when the protocol is `uds`. Defaults to `/var/run/datadog/dsd.socket`.
 - **stream**: Reference to a stream instance. Used only when the protocol is `stream`.
 - **tcpGracefulErrorHandling**: Used only when the protocol is `tcp`. Boolean indicating whether to handle socket errors gracefully. Defaults to true.
 - **tcpGracefulRestartRateLimit**: Used only when the protocol is `tcp`. Time (ms) between re-creating the socket. Defaults to 1000.
 - **udsGracefulErrorHandling**: Used only when the protocol is `uds`. Boolean indicating whether to handle socket errors gracefully. Defaults to true.
 - **udsGracefulRestartRateLimit**: Used only when the protocol is `uds`. Time (ms) between re-creating the socket. Defaults to 1000.
 - **closingFlushInterval**: Before closing, StatsD will check for inflight messages. Time (ms) between each check. Defaults to 50.
 - **udpSocketOptions**: Used only when the protocol is `uds`. Specify the options passed into `dgram.createSocket()`. Defaults to `{ type: 'udp4' }`
-

StatsD methods

All StatsD methods other than `event`, `close`, and `check` have the same API: * `name`: Stat name **required** * `value`: Stat value **required** except in `increment/decrement` where it defaults to 1/-1 respectively * `sampleRate`: Sends only a sample of data to StatsD **default**: 1 * `tags`: The tags to add to metrics. Can be either an object { `tag`: "value" } or an array of tags. **default**: [] * `callback`: The callback to execute once the metric has been sent or buffered

If an array is specified as the `name` parameter each item in that array will be sent along with the specified value.

close The close method has the following API:

- `callback`: The callback to execute once close is complete. All other calls to statsd will fail once this is called.

event The event method has the following API:

- `title`: Event title **required**
- `text`: Event description **default** is title
- `options`: Options for the event
 - `date_happened` Assign a timestamp to the event **default** is now
 - `hostname` Assign a hostname to the event.
 - `aggregation_key` Assign an aggregation key to the event, to group it with some others.
 - `priority` Can be 'normal' or 'low' **default**: normal
 - `source_type_name` Assign a source type to the event.
 - `alert_type` Can be 'error', 'warning', 'info' or 'success' **default**: info
- `tags`: The tags to add to metrics. Can be either an object { `tag`: "value" } or an array of tags. **default**: []
- `callback`: The callback to execute once the metric has been sent.

check The check method has the following API:

- `name`: Check name **required**
- `status`: Check status **required**
- `options`: Options for the check
 - `date_happened` Assign a timestamp to the check **default** is now

-
- `hostname` Assign a hostname to the check.
 - `message` Assign a message to the check.
 - `tags`: The tags to add to metrics. Can be either an object { `tag: "value"` } or an array of tags. **default**: []
 - `callback`: The callback to execute once the metric has been sent.

```
1  var StatsD = require('hot-shots'),
2      client = new StatsD({
3      port: 8020,
4      globalTags: { env: process.env.NODE_ENV },
5      errorHandler: errorHandler,
6  });
7
8  // Increment: Increments a stat by a value (default is 1)
9  client.increment('my_counter');
10
11 // Decrement: Decrements a stat by a value (default is -1)
12 client.decrement('my_counter');
13
14 // Histogram: send data for histogram stat (DataDog and Telegraf only
15 // )
16 client.histogram('my_histogram', 42);
17
18 // Distribution: Tracks the statistical distribution of a set of
19 // values across your infrastructure.
20 // (DataDog v6)
21 client.distribution('my_distribution', 42);
22
23 // Gauge: Gauge a stat by a specified amount
24 client.gauge('my_gauge', 123.45);
25
26 // Set: Counts unique occurrences of a stat (alias of unique)
27 client.set('my_unique', 'foobar');
28 client.unique('my_unique', 'foobarbaz');
29
30 // Event: sends the titled event (DataDog only)
31 client.event('my_title', 'description');
32
33 // Check: sends a service check (DataDog only)
34 client.check('service.up', client.CHECKS.OK, { hostname: 'host-1' },
35             ['foo', 'bar'])
36
37 // Incrementing multiple items
38 client.increment(['these', 'are', 'different', 'stats']);
39
40 // Incrementing with tags
41 client.increment('my_counter', ['foo', 'bar']);
42
43 // Sampling, this will sample 25% of the time the StatsD Daemon will
```

```
    compensate for sampling
41 client.increment('my_counter', 1, 0.25);
42
43 // Tags, this will add user-defined tags to the data
44 // (DataDog and Telegraf only)
45 client.histogram('my_histogram', 42, ['foo', 'bar']);
46
47 // Using the callback. This is the same format for the callback
48 // with all non-close calls
49 client.set(['foo', 'bar'], 42, function(error, bytes){
50     //this only gets called once after all messages have been sent
51     if(error){
52         console.error('Oh noes! There was an error:', error);
53     } else {
54         console.log('Successfully sent', bytes, 'bytes');
55     }
56 });
57
58 // Timing: sends a timing command with the specified milliseconds
59 client.timing('response_time', 42);
60
61 // Timing: also accepts a Date object of which the difference is
    calculated
62 client.timing('response_time', new Date());
63
64 // Timer: Returns a function that you call to record how long the
    first
65 // parameter takes to execute (in milliseconds) and then sends that
    value
66 // using 'client.timing'.
67 // The parameters after the first one (in this case 'fn')
68 // match those in 'client.timing'.
69 var fn = function(a, b) { return a + b };
70 client.timer(fn, 'fn_execution_time')(2, 2);
71
72 // Async timer: Similar to timer above, but you instead pass in a
    function
73 // that returns a Promise. And then it returns a Promise that will
    record the timing.
74 var fn = function () { return new Promise(function (resolve, reject)
    { setTimeout(resolve, n); }); };
75 var instrumented = statsd.asyncTimer(fn, 'fn_execution_time');
76 instrumented().then(function() {
77     console.log('Code run and metric sent');
78 });
79
80 // Async timer: Similar to asyncTimer above, but it instead emits a
    distribution.
81 var fn = function () { return new Promise(function (resolve, reject)
    { setTimeout(resolve, n); }); };
82 var instrumented = statsd.asyncDistTimer(fn, 'fn_execution_time');
```

```

83   instrumented().then(function() {
84       console.log('Code run and metric sent');
85   });
86
87   // Sampling, tags and callback are optional and could be used in any
      combination (DataDog and Telegraf only)
88   client.histogram('my_histogram', 42, 0.25); // 25% Sample Rate
89   client.histogram('my_histogram', 42, { tag: 'value' }); // User-
      defined tag
90   client.histogram('my_histogram', 42, ['tag:value']); // Tags as an
      array
91   client.histogram('my_histogram', 42, next); // Callback
92   client.histogram('my_histogram', 42, 0.25, ['tag']);
93   client.histogram('my_histogram', 42, 0.25, next);
94   client.histogram('my_histogram', 42, { tag: 'value' }, next);
95   client.histogram('my_histogram', 42, 0.25, { tag: 'value' }, next);
96
97   // Use a child client to add more context to the client.
98   // Clients can be nested.
99   var childClient = client.childClient({
100       prefix: 'additionalPrefix.',
101       suffix: '.additionalSuffix',
102       globalTags: { globalTag1: 'forAllMetricsFromChildClient' }
103   });
104   childClient.increment('my_counter_with_more_tags');
105
106   // Close statsd. This will ensure all stats are sent and stop statsd
107   // from doing anything more.
108   client.close(function(err) {
109       console.log('The close did not work quite right: ', err);
110   });

```

DogStatsD and Telegraf functionality

Some of the functionality mentioned above is specific to DogStatsD or Telegraf. They will not do anything if you are using the regular statsd client.

- * globalTags parameter- DogStatsD or Telegraf
- * tags parameter- DogStatsD or Telegraf
- * telegraf parameter- Telegraf
- * uds option in protocol parameter- DogStatsD
- * histogram method- DogStatsD or Telegraf
- * event method- DogStatsD
- * check method- DogStatsD

Errors

As usual, callbacks will have an error as their first parameter. You can have an error in both the message and close callbacks.

If the optional callback is not given, an error is thrown in some cases and a `console.log` message is used in others. An error will only be explicitly thrown when there is a missing callback or if it is some potential configuration issue to be fixed.

If you would like to ensure all errors are caught, specify an `errorHandler` in your root client. This will catch errors in socket setup, sending of messages, and closing of the socket. If you specify an `errorHandler` and a callback, the callback will take precedence.

```
1 // Using errorHandler
2 var client = new StatsD({
3   errorHandler: function (error) {
4     console.log("Socket errors caught here: ", error);
5   }
6 })
```

Congestion error

If you get an error like `Error sending hot-shots message: Error: congestion` with an error code of 1, it is probably because you are sending large volumes of metrics to a single agent/server. This error only arises when using the UDS protocol and means that packages are being dropped. Take a look at the Datadog docs for some tips on tuning your connection.

Unix domain socket support

The 'uds' option as the protocol is to support Unix Domain Sockets for Datadog. It has the following limitations: - It only works where 'node-gyp' works. If you don't know what this is, this is probably fine for you. If you had an troubles with libraries that you 'node-gyp' before, you will have problems here as well. - It does not work on Windows

The above will cause the underlying library that is used, `unix-dgram`, to not install properly. Given the library is listed as an `optionalDependency`, and how it's used in the codebase, this install failure will not cause any problems. It only means that you can't use the uds feature.

Migrating from node-statsd

You should only need to do one thing: change `node-statsd` to `hot-shots` in all requires.

You can check the detailed change log for what has changed since the last release of `node-statsd`.

Submitting changes

Thanks for considering making any updates to this project! This project is entirely community-driven, and so your changes are important. Here are the steps to take in your fork:

1. Run “npm install”
2. Add your changes in your fork as well as any new tests needed
3. Run “npm test”
4. Update README.md with any needed documentation
5. If you have made any API changes, update types.d.ts
6. Push your changes and create the PR

When you’ve done all this we’re happy to try to get this merged in right away.

Package versioning and security

Versions will attempt to follow semantic versioning, with major changes only coming in major versions.

npm publishing is possible by one person, bdeitte, who has two-factor authentication enabled for publishes. Publishes only contain one additional library, unix-dgram.

Name

Why is this project named hot-shots? Because:

1. It’s impossible to find another statsd name on npm
2. It’s the name of a dumb movie
3. No good reason

License

hot-shots is licensed under the MIT license.