Tagalog Documentation

Release 0.3.1

Government Digital Service

July 09, 2014

Welcome. This is the documentation for Tagalog, a set of simple tools for the command line which aim to help system operators manage line-oriented logging data. Tagalog includes tools that

- tag each log line with an accurate timestamp
- convert log lines into Logstash-compatible JSON documents
- ship log lines over the network for further processing
- compute statistics over log streams by sending data to Statsd

This document aims to give a general introduction to Tagalog.

CHAPTER 1

Documentation index

1.1 Tagalog commands

The following page documents the Tagalog command-line utilities.

1.1.1 `logtag`

`logtag` accepts lines of log input on STDIN, applies a filter chain to the log data, and then prints the resulting log entries to STDOUT.

By default, `logtag` has a filter chain of `init_txt,add_timestamp,add_source_host`. This means that it will treat incoming log data as plain text, and will add a timestamp and hostname data to the log entries.

To use `logtag` in its default configuration, simply pass data on STDIN, calling `logtag` without arguments:

```
$ program_to_log | logtag
{"@timestamp": "2013-05-10T16:06:53.441811Z", "@source_host": "lynx.local", "@message": "my log entry
{"@timestamp": "2013-05-10T16:06:53.442090Z", "@source_host": "lynx.local", "@message": "another one"
```

To tag each log entry with “foo” and parse Lograge format logs into the `@fields` field:

```
$ program_to_log | logtag --filters-append add_tags:foo,parse_lograge
{"@timestamp": "2013-05-10T16:08:26.999239Z", ..., "@message": "a log with=some key=value pairs", "@t
{"@timestamp": "2013-05-10T16:08:26.999505Z", ..., "@message": "another=one", "@tags": ["foo"], "@fie
```

If your program emits JSON objects, you might want to use a different filter chain that parses the incoming data as JSON:

```
$ program_emitting_json | logtag --filters init_json,add_timestamp
{"age": 67, "name": "Bob Dole", "@timestamp": "2013-05-10T16:13:49.558039Z"}
{"age": 43, "name": "John Doe", "@timestamp": "2013-05-10T16:13:49.558452Z"}
```

1.1.2 `logship`

`logship` works just like `logtag`, but instead of printing the results to STDOUT, it ships the resulting log entries to one or more “shippers.” Shippers include:

- `redis`: a shipper that pushes log entries onto a Redis list.
- `statsd`: a shipper that submits data to Statsd on the basis of log contents.
- `stdout`: a shipper that simply prints to STDOUT. Mainly used for debugging.

To ship log data to a Redis server, you might invoke `logship` as follows:

```
$ program_to_log | logship --shipper redis,redis://localhost:6379
```

You can configure filters with the `--filters-append` and `--filters` options, just as you can with `logtag`:

```
$ program_emitting_json | logship --filters init_json,add_timestamp --shipper redis,redis://localhost
```

The Redis shipper can take multiple server addresses. It will round-robin between the list of servers. If one server is down, or out-of-memory, or otherwise unable to accept log entries, `logship` will attempt to submit the log entry to one of the other servers.

```
$ program_to_log | logship --shipper redis,redis://redis-1.local:6379,redis://redis-2.local:6379
```

For further shipper documentation, see the API documentation for the `tagalog.shipper` package and its submodules (e.g. `tagalog.shipper.redis`, `tagalog.shipper.statsd`).

1.1.3 `logtext`

`logtext` accepts a stream of newline-delimited Logstash-format JSON documents on STDIN, and prints the log messages on STDOUT. If the log entries have a timestamp, the lines will be prefixed with the timestamp:

```
$ echo '{"@message":"hello"}\n{"@message":"world"}' | logtext
hello
world
```

1.1.4 `logstamp`

`logstamp` accepts lines of input on STDIN, and emits each line prefixed with a high-precision ISO8601 timestamp on STDOUT:

```
$ echo "hello\nworld" | logstamp
2013-05-13T08:59:50.750691Z hello
2013-05-13T08:59:50.751132Z world
```

1.2 tagalog Package

1.2.1 `filters` Module

The `filters` Module contains the definitions for the filters supported by `logtag` and `logship`, as well as a number of functions used to construct the filter chains.

A filter is a generator function which takes an iterable as its first argument, and optionally takes additional configuration arguments. It returns a generator yielding the filtered log lines.

`tagalog.filters.`**`add_fields`**(*iterable*, ***kw_fields*)

Add fields to each item in `iterable`. Each key=value pair provided is merged into the `@fields` object, which will be created if required.

```
>>> data_in = [{'@message': 'one message'}, {'@message': 'another message'}]
>>> data_out = add_fields(data_in, foo='bar', baz='qux')
>>> [x for x in data_out]
[{'@fields': {'foo': 'bar', 'baz': 'qux'}, '@message': 'one message'},
 {'@fields': {'foo': 'bar', 'baz': 'qux'}, '@message': 'another message'}]
```

`tagalog.filters.`**`add_source_host`**(*iterable*, *override=False*)

Add the FQDN of the current machine to the `@source_host` field of each item in `iterable`.

By default, existing `@source_host` fields will not be overwritten. This behaviour can be toggled with the `override` argument.

```
>>> data_in = [{'@message': 'one message'}, {'@message': 'another message'}]
>>> data_out = add_source_host(data_in)
>>> [x for x in data_out]
[{'@source_host': 'lynx.local', '@message': 'one message'},
 {'@source_host': 'lynx.local', '@message': 'another message'}]
```

`tagalog.filters.`**`add_tags`**(*iterable*, **taglist*)

Add tags to each item in `iterable`. Each tag is added to the `@tags` array, which will be created if required.

```
>>> data_in = [{'@message': 'one message'}, {'@message': 'another message'}]
>>> data_out = add_tags(data_in, 'foo', 'bar')
>>> [x for x in data_out]
[{'@message': 'one message', '@tags': ['foo', 'bar']},
 {'@message': 'another message', '@tags': ['foo', 'bar']}]
```

`tagalog.filters.`**`add_timestamp`**(*iterable*, *override=False*)

Compute an accurate timestamp for each item in `iterable`, adding an accurate timestamp to each one when received. The timestamp is a usecond-precision ISO8601 string added to the `@timestamp` field.

By default, existing `@timestamp` fields will not be overwritten. This behaviour can be toggled with the `override` argument.

```
>>> data_in = [{'@message': 'one message'}, {'@message': 'another message'}]
>>> data_out = add_timestamp(data_in)
>>> [x for x in data_out]
[{'@timestamp': '2013-05-13T10:37:56.766743Z', '@message': 'one message'},
 {'@timestamp': '2013-05-13T10:37:56.767185Z', '@message': 'another message'}]
```

`tagalog.filters.`**`build`**(*description*)

Build a filter chain from a filter description string

`tagalog.filters.`**`get`**(*name*, *args=*[ ])

Get a filter function from a filter name and a list of unparsed arguments

`tagalog.filters.`**`init_json`**(*iterable*)

Read lines of JSON text from `iterable` and parse each line as a JSON object. Yield dicts for each line that successfully parses as a JSON object. Unparseable events will be skipped and raise a warning.

```
>>> data_in = ['{"@message": "one message"}', '{"@message": "another message"}']
>>> data_out = init_json(data_in)
>>> [x for x in data_out]
[{u'@message': u'one message'}, {u'@message': u'another message'}]
```

`tagalog.filters.`**`init_txt`**(*iterable*)

Read lines of text from `iterable` and yield dicts with the line data stored in the `@message` field.

```
>>> data_in = ["hello\n", "world\n"]
>>> data_out = init_txt(data_in)
>>> [x for x in data_out]
[{'@message': 'hello'}, {'@message': 'world'}]
```

`tagalog.filters.`**`parse_lograge`**(*iterable*)

Attempt to parse each dict or dict-like object in `iterable` as if it were in lograge format, (e.g. "status=200 path=/users/login time=125ms"), adding key-value pairs to '@fields' for each matching item.

```
>>> data_in = [{'@message': 'path=/foo/bar status=200 time=0.060'},
...            {'@message': 'path=/baz/qux status=503 time=1.651'}]
>>> data_out = parse_lograge(data_in)
>>> [x for x in data_out]
[{'@fields': {'status': '200', 'path': '/foo/bar', 'time': '0.060'}, '@message': 'path=/foo/bar
 {'@fields': {'status': '503', 'path': '/baz/qux', 'time': '1.651'}, '@message': 'path=/baz/qux
```

tagalog.filters.**pipeline**(**functions*)

Construct a filter pipeline from a list of filter functions

Given a list of functions taking an iterable as their only argument, and which return a generator, this function will return a single function with the same signature, which applies each function in turn for each item in the iterable, yielding the results.

That is, given filter functions `a`, `b`, and `c`, `pipeline(a, b, c)` will return a function which yields `c(b(a(x)))` for each item `x` in the iterable passed to it. For example:

```
>>> def a(iterable):
...     for item in iterable:
...         yield item + ':a'
...
>>> def b(iterable):
...     for item in iterable:
...         yield item + ':b'
...
>>> pipe = pipeline(a, b)
>>> data_in = ["foo", "bar"]
>>> data_out = pipe(data_in)
>>> [x for x in data_out]
['foo:a:b', 'bar:a:b']
```

1.2.2 `io` Module

tagalog.io.**lines**(*fp*)

Read lines of UTF-8 from the file-like object given in `fp`, making sure that when reading from STDIN, reads are at most line-buffered.

UTF-8 decoding errors are handled silently. Invalid characters are replaced by U+FFFD REPLACEMENT CHARACTER.

Line endings are normalised to newlines by Python's universal newlines feature.

Returns an iterator yielding lines.

1.2.3 Subpackages

shipper Package

`shipper` Package

class tagalog.shipper.**NullShipper**

Bases: `tagalog.shipper.ishipper.IShipper`

ship(*msg*)

tagalog.shipper.**build_null**(**args*, ***kwargs*) `tagalog.shipper.`**`build_redis`**(**args*, ***kwargs*)

`tagalog.shipper.`**`build_shipper`**(*description*)
: Takes a command-line description of a shipper and build the relevant shipper from it

`tagalog.shipper.`**`build_statsd_counter`**(**args*, ***kwargs*)

`tagalog.shipper.`**`build_statsd_timer`**(**args*, ***kwargs*)

`tagalog.shipper.`**`build_stdout`**(**args*, ***kwargs*)

`tagalog.shipper.`**`get_shipper`**(*name*)

`tagalog.shipper.`**`parse_shipper`**(*description*)

`tagalog.shipper.`**`register_shipper`**(*name*, *constructor*)

`tagalog.shipper.`**`str2bool`**(*s*)

`tagalog.shipper.`**`unregister_shipper`**(*name*)

`formatter` Module

`tagalog.shipper.formatter.`**`elasticsearch_bulk_decorate`**(*bulk_index*, *bulk_type*, *msg*)
: Decorates the msg with elasticsearch bulk format and adds index and message type

`tagalog.shipper.formatter.`**`format_as_elasticsearch_bulk_json`**(*bulk_index*, *bulk_type*, *msg*)

`tagalog.shipper.formatter.`**`format_as_json`**(*msg*)

`ishipper` Module

class `tagalog.shipper.ishipper.`**`IShipper`**
: Bases: `object`

  Abstract class representing a log shipper. Log shippers should implement the following methods:

  `ship`(*message*)

`redis` Module

class `tagalog.shipper.redis.`**`RedisShipper`**(*urls*, *key='logs'*, *bulk=False*, *bulk_index='logs'*, *bulk_type='message'*)
: Bases: `tagalog.shipper.ishipper.IShipper`

  `ship`(*msg*)

class `tagalog.shipper.redis.`**`ResilientStrictRedis`**(*host='localhost'*, *port=6379*, *db=0*, *password=None*, *socket_timeout=None*, *connection_pool=None*, *charset='utf-8'*, *errors='strict'*, *decode_responses=False*, *unix_socket_path=None*)
: Bases: `redis.client.StrictRedis`

  `execute_command`(**args*, ***options*)
  : Execute a command and return a parsed response

  `execution_attempts`

class `tagalog.shipper.redis.`**`RoundRobinConnectionPool`**(*patterns=None*, *max_connections_per_pattern=None*, *connection_class=<class 're-dis.connection.Connection'>*)

Bases: `object`

Round-robin Redis connection pool

`add_pattern`(*pattern*)

`all_connections`()
Returns a generator over all current connection objects

`disconnect`()
Disconnect all connections in the pool

`get_connection`(*command_name*, **keys*, ***options*)
Get a connection from the pool

`make_connection`()
Create a new connection

`purge`(*connection*)
Remove the connection from rotation

`release`(*connection*)
Releases the connection back to the pool

`remove_pattern`(*pattern*)

`shipper_error` Module

exception `tagalog.shipper.shipper_error.`**`ShipperError`**
Bases: `exceptions.Exception`

`statsd` Module

class `tagalog.shipper.statsd.`**`StatsdShipper`**(*metric*, *host='127.0.0.1'*, *port='8125'*)
Bases: `tagalog.shipper.ishipper.IShipper`

`ship`(*msg*)

`tagalog.shipper.statsd.`**`get_from_msg`**(*field*, *msg*)

`stdout` Module

class `tagalog.shipper.stdout.`**`StdoutShipper`**(*bulk=False*, *bulk_index='logs'*, *bulk_type='message'*)
Bases: `tagalog.shipper.ishipper.IShipper`

`ship`(*msg*)

CHAPTER 2

Why would I use Tagalog?

If you are managing a large infrastructure, you're probably generating a lot of log data. It's becoming common practice to send this data to a central location (an elasticsearch cluster, typically) for further analysis (perhaps using Kibana). At the Government Digital Service, we use Tagalog to tag, parse, and ship our logging data to a set of Redis servers, out of which it is then fetched by an elasticsearch cluster using the elasticsearch redis river.

There are other tools which do similar things, such as Logstash and, in particular, lumberjack. If Tagalog doesn't suit your needs, perhaps one of these will.

CHAPTER 3

What is logging data?

Most programs generate some kind of logging data. Your webserver generates access and error logs, while applications and system daemons will inform you of warnings and errors in their logs. Tagalog is broadly agnostic about what you log, but it will work best if each line of logging data can be considered a distinct, timestamped event. While Tagalog won't prevent you using multi-line log formats, it's generally much harder to search and analyse them, so we suggest that if possible, you try and use single-line log entries, ideally in a format that's both human-readable and machine-parseable, such as the Lograge format.

CHAPTER 4

What does Tagalog do?

Tagalog isn't a single tool. Rather, it's a set of tools which are designed to work together. Two of the more important ones are `logtag` and `logship`, and they both treat logging data in a similar manner. They both expect lines of logging data on their standard input [1], and then apply a series of filters to those log lines. Each of these filters typically performs quite a small task: examples include:

- `init_txt`: transforms a line of UTF-8 text into a JSON document with the message contents in a field named `@message`. This is typically the first filter applied.
- `add_timestamp`: adds a precise timestamp to a JSON document in a field named `@timestamp`.
- `add_tags`: adds user specified tags to an array in a field named `@tags`.
- `parse_lograge`: tries to parse any `key=value` pairs (Lograge format) in the `@message` field and adds them to a JSON document in a field named `@fields`.

`logtag` and `logship` both accept arguments that allow you to build a "filter chain": a series of filters which are applied to the incoming log data. At the end of the filter chain, `logtag` prints the results to its standard output, while `logship` (with some additional configuration) ships the log data across the network for further processing.

[1] Support for logging directly from files (log tailing) is being considered for a future release.

CHAPTER 5

Tagalog commands

The next step to using Tagalog is to introduce yourself to the *Tagalog commands*.

CHAPTER 6

Indices and tables

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