gnsq Documentation

Release 1.0.1

Trevor Olson

Apr 26, 2019

Contents

CHAPTER 1

gnsq

A gevent based python client for NSQ distributed messaging platform.

Features include:

- Free software: BSD license
- Documentation: https://gnsq.readthedocs.org
- Battle tested on billions and billions of messages *</sagan>*
- Based on gevent for fast concurrent networking
- Fast and flexible signals with Blinker
- Automatic nsqlookupd discovery and back-off
- Support for TLS, DEFLATE, and Snappy
- Full HTTP clients for both nsqd and nsqlookupd

1.1 Installation

At the command line:

```
$ easy_install gnsq
```

Or even better, if you have virtualenvwrapper installed:

```
$ mkvirtualenv gnsq
$ pip install gnsq
```

Currently there is support for Python 2.7+, Python 3.4+ and PyPy.

1.2 Usage

First make sure nsq is installed and running. Next create a producer and publish some messages to your topic:

```
import gnsq

producer = gnsq.Producer('localhost:4150')

producer.publish('topic', 'hello gevent!')
producer.publish('topic', 'hello nsq!')
```

Then create a Consumer to consume messages from your topic:

```
consumer = gnsq.Consumer('topic', 'channel', 'localhost:4150')

@consumer.on_message.connect
def handler(consumer, message):
    print 'got message:', message.body

consumer.start()
```

1.3 Compatibility

For **NSQ 1.0** and later, use the major version 1 (`1.x.y`) of gnsq.

For **NSQ 0.3.8** and earlier, use the major version 0 (`0.x.y`) of the library.

The recommended way to set your requirements in your *setup.py* or *requirements.txt* is:

```
# NSQ 1.x.y
gnsq>=1.0.0

# NSQ 0.x.y
gnsq<1.0.0
```

1.4 Dependencies

Optional snappy support depends on the *python-snappy* package which in turn depends on libsnappy:

```
# Debian
$ sudo apt-get install libsnappy-dev

# Or OS X
$ brew install snappy

# And then install python-snappy
$ pip install python-snappy
```

1.5 Contributing

Feedback, issues, and contributions are always gratefully welcomed. See the contributing guide for details on how to help and setup a development environment.

CHAPTER 2

Contents

2.1 Consumer: high-level message reader

class gnsq.**Consumer**(*topic*, *channel*, *nsqd_tcp_addresses=[]*, *lookupd_http_addresses=[]*, *name=None*, *message_handler=None*, *max_tries=5*, *max_in_flight=1*, *requeue_delay=0*, *lookupd_poll_interval=60*, *lookupd_poll_jitter=0.3*, *low_ready_idle_timeout=10*, *max_backoff_duration=128*, *backoff_on_requeue=True*, ***kwargs*)

High level NSQ consumer.

A Consumer will connect to the nsqd tcp addresses or poll the provided nsqlookupd http addresses for the configured topic and send signals to message handlers connected to the *on_message* signal or provided by `message_handler`.

Messages will automatically be finished when the message handle returns unless *message.enable_async()* is called. If an exception occurs or `NSQRequeueMessage` is raised, the message will be requeued.

The Consumer will handle backing off of failed messages up to a configurable `max_interval` as well as automatically reconnecting to dropped connections.

Example usage:

```
from gnsq import Consumer

consumer = gnsq.Consumer('topic', 'channel', 'localhost:4150')

@consumer.on_message.connect
def handler(consumer, message):
    print 'got message:', message.body

consumer.start()
```

Parameters

- **topic** – specifies the desired NSQ topic
- **channel** – specifies the desired NSQ channel
- **nsqd_tcp_addresses** – a sequence of string addresses of the nsqd instances this consumer should connect to
- **lookupd_http_addresses** – a sequence of string addresses of the nsqlookupd instances this consumer should query for producers of the specified topic
- **name** – a string that is used for logging messages (defaults to `'gnsq.consumer.{topic}.{channel}'`)
- **message_handler** – the callable that will be executed for each message received
- **max_tries** – the maximum number of attempts the consumer will make to process a message after which messages will be automatically discarded
- **max_in_flight** – the maximum number of messages this consumer will pipeline for processing. this value will be divided evenly amongst the configured/discovered nsqd producers
- **requeue_delay** – the default delay to use when requeueing a failed message
- **lookupd_poll_interval** – the amount of time in seconds between querying all of the supplied nsqlookupd instances. A random amount of time based on this value will be initially introduced in order to add jitter when multiple consumers are running
- **lookupd_poll_jitter** – the maximum fractional amount of jitter to add to the lookupd poll loop. This helps evenly distribute requests even if multiple consumers restart at the same time.
- **low_ready_idle_timeout** – the amount of time in seconds to wait for a message from a producer when in a state where RDY counts are re-distributed (ie. *max_in_flight* < *num_producers*)
- **max_backoff_duration** – the maximum time we will allow a backoff state to last in seconds. If zero, backoff wil not occur
- **backoff_on_requeue** – if `False`, backoff will only occur on exception
- ****kwargs** – passed to *NsqdTCPClient* initialization

close()
Immediately close all connections and stop workers.

is_running
Check if consumer is currently running.

is_starved
Evaluate whether any of the connections are starved.

This property should be used by message handlers to reliably identify when to process a batch of messages.

join(*timeout=None*, *raise_error=False*)
Block until all connections have closed and workers stopped.

on_auth
Emitted after a connection is successfully authenticated.

The signal sender is the consumer and the `conn` and parsed `response` are sent as arguments.

on_close
Emitted after *close()*.

The signal sender is the consumer.

on_error
Emitted when an error is received.

The signal sender is the consumer and the `error` is sent as an argument.

on_exception
Emitted when an exception is caught while handling a message.

The signal sender is the consumer and the `message` and `error` are sent as arguments.

on_finish
Emitted after a message is successfully finished.

The signal sender is the consumer and the `message_id` is sent as an argument.

on_giving_up
Emitted after a giving up on a message.

Emitted when a message has exceeded the maximum number of attempts (`max_tries`) and will no longer be requeued. This is useful to perform tasks such as writing to disk, collecting statistics etc. The signal sender is the consumer and the `message` is sent as an argument.

on_message
Emitted when a message is received.

The signal sender is the consumer and the `message` is sent as an argument. The `message_handler` param is connected to this signal.

on_requeue
Emitted after a message is requeued.

The signal sender is the consumer and the `message_id` and `timeout` are sent as arguments.

on_response
Emitted when a response is received.

The signal sender is the consumer and the `response` is sent as an argument.

start(*block=True*)
Start discovering and listing to connections.

2.2 Producer: high-level message writer

class gnsq.**Producer**(*nsqd_tcp_addresses=[]*, *max_backoff_duration=128*, ***kwargs*)
High level NSQ producer.

A Producer will connect to the nsqd tcp addresses and support async publishing (`PUB` & `MPUB` & `DPUB`) of messages to *nsqd* over the TCP protocol.

Example publishing a message:

```
from gnsq import Producer

producer = Producer('localhost:4150')
producer.start()
producer.publish('topic', b'hello world')
```

Parameters

- **nsqd_tcp_addresses** – a sequence of string addresses of the nsqd instances this consumer should connect to
- **max_backoff_duration** – the maximum time we will allow a backoff state to last in seconds. If zero, backoff wil not occur
- ****kwargs** – passed to *NsqdTCPClient* initialization

close()
Immediately close all connections and stop workers.

is_running
Check if the producer is currently running.

join(*timeout=None*, *raise_error=False*)
Block until all connections have closed and workers stopped.

multipublish(*topic*, *messages*, *block=True*, *timeout=None*, *raise_error=True*)
Publish an iterable of messages to the given topic.

Parameters

- **topic** – the topic to publish to
- **messages** – iterable of bytestrings to publish
- **block** – wait for a connection to become available before publishing the message. If block is *False* and no connections are available, `NSQNoConnections` is raised
- **timeout** – if timeout is a positive number, it blocks at most `timeout` seconds before raising `NSQNoConnections`
- **raise_error** – if `True`, it blocks until a response is received from the nsqd server, and any error response is raised. Otherwise an `AsyncResult` is returned

on_auth
Emitted after a connection is successfully authenticated.

The signal sender is the consumer and the `conn` and parsed `response` are sent as arguments.

on_close
Emitted after *close()*.

The signal sender is the consumer.

on_error
Emitted when an error is received.

The signal sender is the consumer and the `error` is sent as an argument.

on_response
Emitted when a response is received.

The signal sender is the consumer and the ‘ ‘ is sent as an argument.

publish(*topic*, *data*, *defer=None*, *block=True*, *timeout=None*, *raise_error=True*)
Publish a message to the given topic.

Parameters

- **topic** – the topic to publish to
- **data** – bytestring data to publish
- **defer** – duration in milliseconds to defer before publishing (requires nsq 0.3.6)

- **block** – wait for a connection to become available before publishing the message. If block is *False* and no connections are available, `NSQNoConnections` is raised
- **timeout** – if timeout is a positive number, it blocks at most `timeout` seconds before raising `NSQNoConnections`
- **raise_error** – if `True`, it blocks until a response is received from the nsqd server, and any error response is raised. Otherwise an `AsyncResult` is returned

start()
Start discovering and listing to connections.

2.3 Nsqd clients

class gnsq.**NsqdHTTPClient**(*host='localhost'*, *port=4151*, ***kwargs*)
Low level http client for nsqd.

Parameters

- **host** – nsqd host address (default: localhost)
- **port** – nsqd http port (default: 4151)
- **useragent** – useragent sent to nsqd (default: `<client_library_name>/<version>`)
- **connection_class** – override the http connection class

create_channel(*topic*, *channel*)
Create a channel for an existing topic.

create_topic(*topic*)
Create a topic.

delete_channel(*topic*, *channel*)
Delete an existing channel for an existing topic.

delete_topic(*topic*)
Delete a topic.

empty_channel(*topic*, *channel*)
Empty all the queued messages for an existing channel.

empty_topic(*topic*)
Empty all the queued messages for an existing topic.

classmethod from_url(*url*, ***kwargs*)
Create a client from a url.

info()
Returns version information.

multipublish(*topic*, *messages*, *binary=False*)
Publish an iterable of messages to the given topic over http.

Parameters

- **topic** – the topic to publish to
- **messages** – iterable of bytestrings to publish
- **binary** – enable binary mode. defaults to False (requires nsq 1.0.0)

By default multipublish expects messages to be delimited by `"\n"`, use the binary flag to enable binary mode where the POST body is expected to be in the following wire protocol format.

pause_channel(*topic*, *channel*)
Pause message flow to consumers of an existing channel.

Messages will queue at channel.

pause_topic(*topic*)
Pause message flow to all channels on an existing topic.

Messages will queue at topic.

ping()
Monitoring endpoint.

Returns should return `"OK"`, otherwise raises an exception.

publish(*topic*, *data*, *defer=None*)
Publish a message to the given topic over http.

Parameters

- **topic** – the topic to publish to
- **data** – bytestring data to publish
- **defer** – duration in millisconds to defer before publishing (requires nsq 0.3.6)

stats(*topic=None*, *channel=None*, *text=False*)
Return internal instrumented statistics.

Parameters

- **topic** – (optional) filter to topic
- **channel** – (optional) filter to channel
- **text** – return the stats as a string (default: `False`)

unpause_channel(*topic*, *channel*)
Resume message flow to consumers of an existing, paused, channel.

unpause_topic(*topic*)
Resume message flow to channels of an existing, paused, topic.

class gnsq.**NsqdTCPClient**(*address='127.0.0.1'*, *port=4150*, *timeout=60.0*, *client_id=None*, *hostname=None*, *heartbeat_interval=30*, *output_buffer_size=16384*, *output_buffer_timeout=250*, *tls_v1=False*, *tls_options=None*, *snappy=False*, *deflate=False*, *deflate_level=6*, *sample_rate=0*, *auth_secret=None*, *user_agent='gnsq/1.0.1'*)
Low level object representing a TCP connection to nsqd.

Parameters

- **address** – the host or ip address of the nsqd
- **port** – the nsqd tcp port to connect to
- **timeout** – the timeout for read/write operations (in seconds)
- **client_id** – an identifier used to disambiguate this client (defaults to the first part of the hostname)
- **hostname** – the hostname where the client is deployed (defaults to the clients hostname)

- **heartbeat_interval** – the amount of time in seconds to negotiate with the connected producers to send heartbeats (requires nsqd 0.2.19+)
- **output_buffer_size** – size of the buffer (in bytes) used by nsqd for buffering writes to this connection
- **output_buffer_timeout** – timeout (in ms) used by nsqd before flushing buffered writes (set to 0 to disable). Warning: configuring clients with an extremely low (< 25ms) output_buffer_timeout has a significant effect on nsqd CPU usage (particularly with > 50 clients connected).
- **tls_v1** – enable TLS v1 encryption (requires nsqd 0.2.22+)
- **tls_options** – dictionary of options to pass to ssl.wrap_socket()
- **snappy** – enable Snappy stream compression (requires nsqd 0.2.23+)
- **deflate** – enable deflate stream compression (requires nsqd 0.2.23+)
- **deflate_level** – configure the deflate compression level for this connection (requires nsqd 0.2.23+)
- **sample_rate** – take only a sample of the messages being sent to the client. Not setting this or setting it to 0 will ensure you get all the messages destined for the client. Sample rate can be greater than 0 or less than 100 and the client will receive that percentage of the message traffic. (requires nsqd 0.2.25+)
- **auth_secret** – a string passed when using nsq auth (requires nsqd 0.2.29+)
- **user_agent** – a string identifying the agent for this client in the spirit of HTTP (default: `<client_library_name>/<version>`) (requires nsqd 0.2.25+)

auth()
Send authorization secret to nsqd.

close()
Indicate no more messages should be sent.

close_stream()
Close the underlying socket.

connect()
Initialize connection to the nsqd.

finish(*message_id*)
Finish a message (indicate successful processing).

identify()
Update client metadata on the server and negotiate features.

Returns nsqd response data if there was feature negotiation, otherwise `None`

is_connected
Check if the client is currently connected.

is_starved
Evaluate whether the connection is starved.

This property should be used by message handlers to reliably identify when to process a batch of messages.

listen()
Listen to incoming responses until the connection closes.

multipublish(*topic*, *messages*)
Publish an iterable of messages to the given topic over http.

Parameters

- **topic** – the topic to publish to
- **messages** – iterable of bytestrings to publish

nop()
Send no-op to nsqd. Used to keep connection alive.

on_auth
Emitted after the connection is successfully authenticated.

The signal sender is the connection and the parsed `response` is sent as arguments.

on_close
Emitted after *close_stream()*.

Sent after the connection socket has closed. The signal sender is the connection.

on_error
Emitted when an error frame is received.

The signal sender is the connection and the `error` is sent as an argument.

on_finish
Emitted after *finish()*.

Sent after a message owned by this connection is successfully finished. The signal sender is the connection and the `message_id` is sent as an argument.

on_message
Emitted when a message frame is received.

The signal sender is the connection and the `message` is sent as an argument.

on_requeue
Emitted after *requeue()*.

Sent after a message owned by this connection is requeued. The signal sender is the connection and the `message_id`, `timeout` and `backoff` flag are sent as arguments.

on_response
Emitted when a response frame is received.

The signal sender is the connection and the `response` is sent as an argument.

publish(*topic*, *data*, *defer=None*)
Publish a message to the given topic over tcp.

Parameters

- **topic** – the topic to publish to
- **data** – bytestring data to publish
- **defer** – duration in milliseconds to defer before publishing (requires nsq 0.3.6)

read_response()
Read an individual response from nsqd.

Returns tuple of the frame type and the processed data.

ready(*count*)
Indicate you are ready to receive `count` messages.

requeue(*message_id*, *timeout=0*, *backoff=True*)
Re-queue a message (indicate failure to process).

subscribe(*topic*, *channel*)
Subscribe to a nsq *topic* and *channel*.

touch(*message_id*)
Reset the timeout for an in-flight message.

class gnsq.**Nsqd**(*address='127.0.0.1'*, *tcp_port=4150*, *http_port=4151*, ***kwargs*)
Use *NsqdTCPClient* or *NsqdHTTPClient* instead.

Deprecated since version 1.0.0.

multipublish_http(*topic*, *messages*, ***kwargs*)
Use *NsqdHTTPClient.multipublish()* instead.

Deprecated since version 1.0.0.

multipublish_tcp(*topic*, *messages*, ***kwargs*)
Use *NsqdTCPClient.multipublish()* instead.

Deprecated since version 1.0.0.

publish_http(*topic*, *data*, ***kwargs*)
Use *NsqdHTTPClient.publish()* instead.

Deprecated since version 1.0.0.

publish_tcp(*topic*, *data*, ***kwargs*)
Use *NsqdTCPClient.publish()* instead.

Deprecated since version 1.0.0.

2.4 Nsqlookupd client

class gnsq.**LookupdClient**(*host='localhost'*, *port=4161*, ***kwargs*)
Low level http client for nsqlookupd.

Parameters

- **host** – nsqlookupd host address (default: localhost)
- **port** – nsqlookupd http port (default: 4161)
- **useragent** – useragent sent to nsqlookupd (default: `<client_library_name>/<version>`)
- **connection_class** – override the http connection class

channels(*topic*)
Returns all known channels of a topic.

create_channel(*topic*, *channel*)
Add a channel to nsqlookupd's registry.

create_topic(*topic*)
Add a topic to nsqlookupd's registry.

delete_channel(*topic*, *channel*)
Deletes an existing channel of an existing topic.

delete_topic(*topic*)
Deletes an existing topic.

classmethod from_url(*url*, ***kwargs*)
Create a client from a url.

info()
Returns version information.

lookup(*topic*)
Returns producers for a topic.

nodes()
Returns all known nsqd.

ping()
Monitoring endpoint.

Returns should return *"OK"*, otherwise raises an exception.

tombstone_topic(*topic*, *node*)
Tombstones a specific producer of an existing topic.

topics()
Returns all known topics.

class gnsq.**Lookupd**(*address='http://localhost:4161/'*, ***kwargs*)
Use *LookupdClient* instead.

Deprecated since version 1.0.0.

base_url
Use `LookupdClient.address` instead.

Deprecated since version 1.0.0.

tombstone_topic_producer(*topic*, *node*)
Use *LookupdClient.tombstone_topic()* instead.

Deprecated since version 1.0.0.

2.5 NSQ Message

class gnsq.**Message**(*timestamp*, *attempts*, *id*, *body*)
A class representing a message received from nsqd.

enable_async()
Enables asynchronous processing for this message.

Consumer will not automatically respond to the message upon return of `handle_message()`.

finish()
Respond to nsqd that you've processed this message successfully (or would like to silently discard it).

has_responded()
Returns whether or not this message has been responded to.

is_async()
Returns whether or not asynchronous processing has been enabled.

on_finish
Emitted after *finish()*.

The signal sender is the message instance.

on_requeue
Emitted after *requeue()*.

The signal sender is the message instance and sends the `timeout` and a `backoff` flag as arguments.

on_touch
Emitted after *touch()*.

The signal sender is the message instance.

requeue(*time_ms=0*, *backoff=True*)
Respond to nsqd that you've failed to process this message successfully (and would like it to be requeued).

touch()
Respond to nsqd that you need more time to process the message.

2.6 Signals

Both *Consumer* and *NsqdTCPClient* classes expose various signals provided by the Blinker library.

2.6.1 Subscribing to signals

To subscribe to a signal, you can use the `connect()` method of a signal. The first argument is the function that should be called when the signal is emitted, the optional second argument specifies a sender. To unsubscribe from a signal, you can use the `disconnect()` method.

```
def error_handler(consumer, error):
    print 'Got an error:', error

consumer.on_error.connect(error_handler)
```

You can also easily subscribe to signals by using `connect()` as a decorator:

```
@consumer.on_giving_up.connect
def handle_giving_up(consumer, message):
    print 'Giving up on:', message.id
```

2.7 Contrib modules

Patterns and best practices for gnsq made code.

2.7.1 Batching messages

class gnsq.contrib.batch.**BatchHandler**(*batch_size*, *handle_batch=None*, *handle_message=None*, *handle_batch_error=None*, *handle_message_error=None*, *timeout=10*, *spawn=<bound method Greenlet.spawn of <class 'gevent._greenlet.Greenlet'>>*)
Batch message handler for gnsq.

It is recommended to use a max inflight greater than the batch size.

Example usage:

```
>>> consumer = Consumer('topic', 'worker', max_in_flight=16)
>>> consumer.on_message.connect(BatchHandler(8, my_handler), weak=False)
```

handle_batch(*messages*)
Handle a batch message.

Processes a batch of messages. You must provide a *handle_batch()* function to the constructor or override this method.

Raising an exception in *handle_batch()* will cause all messages in the batch to be requeued.

handle_batch_error(*error*, *messages*, *batch*)
Handle an exception processsing a batch of messages.

This may be overridden or passed into the constructor.

handle_message(*message*)
Handle a single message.

Over ride this to provide some processing and an individual message. The result of this function is what is passed to *handle_batch()*. This may be overridden or passed into the constructor. By default it simply returns the message.

Raising an exception in *handle_message()* will cause that message to be requeued and excluded from the batch.

handle_message_error(*error*, *message*)
Handle an exception processesing an individual message.

This may be overridden or passed into the constructor.

2.7.2 Giveup handlers

class gnsq.contrib.giveup.**LogGiveupHandler**(*log=<built-in method write of _io.TextIOWrapper object>*, *newline='n'*)
Log messages on giveup.

Writes the message body to the log. This can be customized by subclassing and implementing `format_message()`. Assuming messages do not requeued using the *to_nsq* utility.

Example usage:

```
>>> fp = open('topic.__BURY__.log', 'w')
>>> consumer.on_giving_up.connect(
...     LogGiveupHandler(fp.write), weak=False)
```

class gnsq.contrib.giveup.**JSONLogGiveupHandler**(*log=<built-in method write of _io.TextIOWrapper object>*, *newline='n'*)
Log messages as json on giveup.

Works like *LogGiveupHandler* but serializes the message details as json before writing to the log.

Example usage:

```
>>> fp = open('topic.__BURY__.log', 'w')
>>> consumer.on_giving_up.connect(
...     JSONLogGiveupHandler(fp.write), weak=False)
```

class gnsq.contrib.giveup.**NsqdGiveupHandler**(*topic*, *nsqd_hosts=['localhost']*, *nsqd_class=<class 'gnsq.nsqd.NsqdHTTPClient'>*)

Send messages by to nsq on giveup.

Forwards the message body to the given topic where it can be inspected and requeued. This can be customized by subclassing and implementing `format_message()`. Messages can be requeued with the *nsq_to_nsq* utility.

Example usage:

```
>>> giveup_handler = NsqdGiveupHandler('topic.__BURY__')
>>> consumer.on_giving_up.connect(giveup_handler)
```

2.7.3 Concurrency

class gnsq.contrib.queue.**QueueHandler**

Iterator like api for gnsq.

Example usage:

```
>>> queue = QueueHandler()
>>> consumer = Consumer('topic', 'worker', max_in_flight=16)
>>> consumer.on_message.connect(queue)
>>> consumer.start(block=False)
>>> for message in queue:
...     print(message.body)
...     message.finish()
```

Or give it to a pool:

```
>>> gevent.pool.Pool().map(queue, my_handler)
```

Parameters **maxsize** – maximum number of messages that can be queued. If less than or equal to zero or None, the queue size is infinite.

empty

Return `True` if the queue is empty, `False` otherwise.

full

Return `True` if the queue is full, `False` otherwise.

`Queue(None)` is never full.

get

Remove and return an item from the queue.

If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

get_nowait

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the `Empty` exception.

peek
Return an item from the queue without removing it.

If optional args *block* is true and *timeout* is `None` (the default), block if necessary until an item is available. If *timeout* is a positive number, it blocks at most *timeout* seconds and raises the `Empty` exception if no item was available within that time. Otherwise (*block* is false), return an item if one is immediately available, else raise the `Empty` exception (*timeout* is ignored in that case).

peek_nowait
Return an item from the queue without blocking.

Only return an item if one is immediately available. Otherwise raise the `Empty` exception.

qsize
Return the size of the queue.

class gnsq.contrib.queue.**ChannelHandler**
Iterator like api for gnsq.

Like *QueueHandler* with a `maxsize` of `1`.

2.7.4 Error logging

class gnsq.contrib.sentry.**SentryExceptionHandler**(*client*)
Log gnsq exceptions to sentry.

Example usage:

```
>>> from raven import Sentry
>>> sentry = Sentry()
>>> consumer.on_exception.connect(
...     SentryExceptionHandler(sentry), weak=False)
```

2.8 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.8.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/wtolson/gnsq/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

gnsq could always use more documentation, whether as part of the official gnsq docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/wtolson/gnsq/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.8.2 Get Started!

Ready to contribute? Here's how to set up *gnsq* for local development.

1. Fork the *gnsq* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/gnsq.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper and libsnappy installed, this is how you set up your fork for local development:

```
$ mkvirtualenv gnsq
$ cd gnsq/
$ pip install -r requirements.dev.txt -r requirements.docs.txt
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 gnsq tests
$ pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6 and 2.7. Check https://travis-ci.org/wtolson/gnsq/pull_requests and make sure that the tests pass for all supported Python versions.

2.8.4 Tips

To run a subset of tests:

```
$ pytest tests/test_basic.py
```

2.9 Credits

2.9.1 Development Lead

- Trevor Olson <trevor@heytrevor.com>

2.9.2 Contributors

None yet. Why not be the first?

2.10 Upgrading to Newer Releases

This section of the documentation enumerates all the changes in gnsq from release to release and how you can change your code to have a painless updating experience.

Use the **pip** command to upgrade your existing Flask installation by providing the `--upgrade` parameter:

```
$ pip install --upgrade gnsq
```

2.10.1 Version 1.0.0

While there are no breaking changes in version 1.0.0, much of the interface has been deprecated to both simplify the api and bring it into better compliance with the recommended naming schemes for nsq clients. Existing code should work as is and deprecation warnings will be emitted for any code paths that need to be changed.

Deprecated Reader

The main interface has been renamed from `Reader` to *`Consumer`*. The api remains largely the same and can be swapped out directly in most cases.

Async messages

The `async` flag has been removed from the *`Consumer`*. Instead *`messages`* has a *`message.enable_async()`* method that may be used to indicate that a message will be handled asynchronous.

Max concurrency

The `max_concurrency` parameter has been removed from *`Consumer`*. If you wish to replicate this behavior, you should use the `gnsq.contrib.QueueHandler` in conjunction with a worker pool:

```
from gevent.pool import Pool
from gnsq import Consumer
from gnsq.contrib.queue import QueueHandler

MAX_CONCURRENCY = 4

# Create your consumer as usual
consumer = Consumer(
    'topic', 'worker', 'localhost:4150', max_in_flight=16)

# Connect a queue handler to the on message signal
queue = QueueHandler()
consumer.on_message.connect(queue)

# Start your consumer without blocking or in a separate greenlet
consumer.start(block=False)

# If you want to limit your concurrency to a single greenlet, simply loop
# over the queue in a for loop, or you can use a worker pool to distribute
# the work.
pool = Pool(MAX_CONCURRENCY)
results = pool.imap_unordered(queue, my_handler)

# Consume the results from the pool
for result in results:
    pass
```

Deprecated Nsqd

The *`Nsqd`* client has been split into two classes, corresponding to the tcp and http APIs. The new classes are *`NsqdTCPClient`* and *`NsqdHTTPClient`* respectively.

The methods *publish_tcp*, *publish_http*, *multipublish_tcp*, and *multipublish_http* have been removed from the new classes.

Deprecated Lookupd

The *Lookupd* class has been replaced by *LookupdClient*. *LookupdClient* can be constructed using the `host` and `port` or by passing the url to *LookupdClient.from_url()* instead.

The method *tombstone_topic_producer()* has been renamed to *tombstone_topic()*.

2.11 History

2.11.1 1.0.1 (2019-04-24)

- Fix long description in packaging

2.11.2 1.0.0 (2019-04-24)

- Drop support for python 2.6 and python 3.3, add support for python 3.7
- Drop support for nsq < 1.0.0
- Handle changing connections during redistribute ready
- Add create topic and create channel to LookupdClient
- Add pause and unpause topic to NsqdHTTPClient
- Add ability to filter NsqdHTTPClient stats by topic/channel
- Add text format for NsqdHTTPClient stats
- Add binary multipublish over http
- Add queue handler to the contrib package
- Add Producer class, a high level tcp message writer
- Fixed detecting if consumer is starved
- Optimizations to better distribute ready state among the nsqd connections
- Detect starved consumers when batching messages
- [DEPRECATED] *Nsqd* is deprecated. Use *NsqdTCPClient* or *NsqdHTTPClient* instead. See *Version 1.0.0* for more information.
- [DEPRECATED] *Lookupd* is deprecated. Use *LookupdClient* instead. See *Version 1.0.0* for more information.
- [DEPRECATED] `Reader` is deprecated. Use *Consumer* instead. See *Version 1.0.0* for more information.

2.11.3 0.4.0 (2017-06-13)

- #13 - Allow use with nsq v1.0.0 (thanks @daroot)
- Add contrib package with utilities.

2.11.4 0.3.3 (2016-09-25)

- #11 - Make sure all socket data is sent.
- #5 - Add support for DPUB (defered publish).

2.11.5 0.3.2 (2016-04-10)

- Add support for Python 3 and PyPy.
- #7 - Fix undeclared variable in compression socket.

2.11.6 0.3.1 (2015-11-06)

- Fix negative in flight causing not throttling after backoff.

2.11.7 0.3.0 (2015-06-14)

- Fix extra backoff success/failures during backoff period.
- Fix case where handle_backoff is never called.
- Add backoff parameter to message.requeue().
- Allow overriding backoff on NSQRequeueMessage error.
- Handle connection failures while starting/completing backoff.

2.11.8 0.2.3 (2015-02-16)

- Remove disconnected nsqd messages from the worker queue.
- #4 - Fix crash in Reader.random_ready_conn (thanks @ianpreston).

2.11.9 0.2.2 (2015-01-12)

- Allow finishing and requeuing in sync handlers.

2.11.10 0.2.1 (2015-01-12)

- Topics and channels are now valid to 64 characters.
- Ephemeral topics are now valid.
- Adjustable backoff behavior.

2.11.11 0.2.0 (2014-08-03)

- Warn on connection failure.
- Add extra requires for snappy.
- Add support for nsq auth protocol.

2.11.12 0.1.4 (2014-07-24)

- Preemptively update ready count.
- Dependency and contributing documentation.
- Support for nsq back to 0.2.24.

2.11.13 0.1.3 (2014-07-08)

- Block as expected on start, even if already started.
- Raise runtime error if starting the reader without a message handler.
- Add on_close signal to the reader.
- Allow upgrading to tls+snappy or tls+deflate.

2.11.14 0.1.2 (2014-07-08)

- Flush delfate buffer for each message.

2.11.15 0.1.1 (2014-07-07)

- Fix packaging stream submodule.
- Send queued messages before closing socket.
- Continue to read from socket on EAGAIN

2.11.16 0.1.0 (2014-07-07)

- First release on PyPI.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index