Flask-MQTT Documentation

Release 1.1.1

Stefan Lehmann

May 04, 2023

Contents

Flask-MQTT is a Flask extension meant to facilitate the integration of a MQTT client into your web application. Basically it is a thin wrapper around the paho-mqtt package to simplify MQTT integration in a Flask application. MQTT is a machine-to-machine (M2M)/"Internet of Things" (IoT) protocol which is designed as a lightweight publish/subscribe messaging transport. It comes very handy when trying to connect multiple IoT devices with each other or monitor and control these devices from one or multiple clients.

CHAPTER 1

Limitations

1.1 Multiple workers

Flask-MQTT is currently not suitable for the use with multiple worker instances. So if you use a WSGI server like *gevent* or *gunicorn* make sure you only have one worker instance.

Flask-MQTT was developed to provide an easy-to-setup solution for interacting with IoT devices. A typical scenario would be a Raspberry Pi running a mosquitto mqtt server combined with a Flask webserver.

1.2 Reloader

Make sure to disable Flasks autoreloader. If activated it spawns two instances of a Flask application. This leads to the same problems as multiple workers. To prevent Flask-MQTT from running code twice it is necessary to deactivate the automatic reloader.

1.3 Type annotations

This package uses type annotations so it needs Python 3.6 or Python 2.7/3.x with the typing package installed.

CHAPTER 2

Content

2.1 Configuration

The following configuration keys exist for Flask-MQTT. Flask-MQTT loads these values from your main Flask config.

2.1.1 Configuration Keys

MQTT_CLIENT_ID	the unique client id string used when connecting to the broker. If client_id is zero length or None, then one will be randomly generated.
MQTT_BROKER_URL	The broker URL that should be used for the connection. Defaults to `localhost`. Example: • `mybroker.com`
MQTT_BROKER_PORT	The broker port that should be used for the connection. Defaults to 1883. • MQTT: `1883` • MQTT encrypted (SSL): `8883`
MQTT_USERNAME	The username used for authentication. If none is provided authentication is disabled. Defaults to `None`.
MQTT_PASSWORD	The password used for authentication. Defaults to `None`. Only needed if a username is provided.
MQTT_KEEPALIVE	Maximum period in seconds between communications with the broker. If no other messages are being exchanged, this controls the rate at which the client will send ping messages to the broker. Defaults to 60 seconds.
MQTT_TLS_ENABLED	Enable TLS for the connection to the MQTT broker. Use the following config keys to configure TLS.
MQTT_TLS_CA_CERTS	A string path to the Certificate Authority certificate files that are to be treated as trusted by this client. Required.
MQTT_TLS_CERTFILE	String pointing to the PEM encoded client certificate. Defaults to None.
MQTT_TLS_KEYFILE	String pointing to the PEM encoded client private key. Defaults to None.
MQTT_TLS_CERT_REQS	Defines the certificate requirements that the client imposes on the broker. By default this is ssl.CERT_REQUIRED, which means that the broker must provide a certificate. See the ssl pydoc for more information on this parameter. Defaults to ssl.CERT_REQUIRED.
MQTT_TLS_VERSION	Specifies the version of the SSL/TLS protocol to be used. By default TLS v1 is used. Previous versions (all versions beginning with SSL) are possible but not recommended due to possible security problems. Defaults to ssl.PROTOCOL_TLSv1.
MQTT_TLS_CIPHERS	A string specifying which encryption ciphers are allowable for this connection, or None to use the defaults. See the ssl pydoc for more information. Defaults to None.
MQTT_TLS_INSECURE	Configure verification of the server hostname in the server certificate. Defaults to False. Do not use this function in a real system. Setting value to True means there is no point using encryption.
MQTT_LAST_WILL_TOPIC	The topic that the will message should be published on. If not set no will message will be sent on disconnecting the client.
MQTT_LAST_WILL_MESSAGE	The message to send as a will. If not given, or set to None a zero length message will be used as the will. Passing an int or float will result in the payload being converted to a string representing that number. If you wish to send a true int/float, use struct.pack() to create the payload you require.
MQTT_LAST_WILL_QOS	The quality of service level to use for the will. Defaults to 0.
MQTT_LAST_WILL_RETAIN	If set to true, the will message will be set as the "last known good"/retained message for the topic. Defaults to False.
MQTT_TRANSPORT	set to "websockets" to send MQTT over WebSockets. Leave at the default of "tcp" to use raw TCP.

2.2 Usage

2.2.1 Connect to a broker

To connect to a broker you only need to initialize the *Flask-MQTT* extension with your *Flask application*. You can do this by directly passing the Flask application object on object creation.

```
from flask import Flask
from flask_mqtt import Mqtt

app = Flask(__name__)
mqtt = Mqtt(app)
```

The *Flask-MQTT* extension supports the factory pattern so you can instantiate a `Mqtt` object without an app object. Use the `init_app()` function inside the factory function for initialization.

```
from flask import Flask
from flask_mqtt import Mqtt

mqtt = Mqtt()

def create_app():
    app = Flask(__name__)
    mqtt.init_app(app)
```

2.2.2 Configure the MQTT client

The configuration of the MQTT client is done via configuration variables as it is common for Flask extension.

```
from flask import Flask
from flask_mqtt import Mqtt


app = Flask(__name__)
app.config['MQTT_BROKER_URL'] = 'broker.hivemq.com'  # use the free broker from HIVEMQ
app.config['MQTT_BROKER_PORT'] = 1883  # default port for non-tls connection
app.config['MQTT_USERNAME'] = ''  # set the username here if you need authentication␣
↪for the broker
app.config['MQTT_PASSWORD'] = ''  # set the password here if the broker demands␣
↪authentication
app.config['MQTT_KEEPALIVE'] = 5  # set the time interval for sending a ping to the␣
↪broker to 5 seconds
app.config['MQTT_TLS_ENABLED'] = False  # set TLS to disabled for testing purposes

mqtt = Mqtt()
```

All available configuration variables are listed in the configuration section.

2.2.3 Subscribe to a topic

To subscribe to a topic simply use *`flask_mqtt.Mqtt.subscribe()`*.

```
mqtt.subscribe('home/mytopic')
```

If you want to subscribe to a topic right from the start make sure to wait with the subscription until the client is connected to the broker. Use the *flask_mqtt.Mqtt.on_connect()* decorator for this.

```
@mqtt.on_connect()
def handle_connect(client, userdata, flags, rc):
    mqtt.subscribe('home/mytopic')
```

To handle the subscribed messages you can define a handling function by using the *flask_mqtt.Mqtt.on_message()* decorator.

```
@mqtt.on_message()
def handle_mqtt_message(client, userdata, message):
    data = dict(
        topic=message.topic,
        payload=message.payload.decode()
    )
```

To unsubscribe use *flask_mqtt.Mqtt.unsubscribe()*.

```
mqtt.unsubscribe('home/mytopic')
```

Or if you want to unsubscribe all topics use *flask_mqtt.Mqtt.unsubscribe_all()*.

```
mqtt.unsubscribe_all()
```

2.2.4 Publish a message

Publishing a message is easy. Just use the *flask_mqtt.Mqtt.publish()* method here.

```
mqtt.publish('home/mytopic', 'hello world')
```

2.2.5 Logging

To enable logging there exists the *flask_mqtt.Mqtt.on_log()* decorator. The level variable gives the severity of the message and will be one of these:

flask_mqtt.MQTT_LOG_INFO	0x01
flask_mqtt.MQTT_LOG_NOTICE	0x02
flask_mqtt.MQTT_LOG_WARNING	0x04
flask_mqtt.MQTT_LOG_ERR	0x08
flask_mqtt.MQTT_LOG_DEBUG	0x10

```
@mqtt.on_log()
def handle_logging(client, userdata, level, buf):
    if level == MQTT_LOG_ERR:
        print('Error: {}'.format(buf))
```

2.2.6 Interact with SocketIO

Flask-MQTT plays nicely with the Flask-SocketIO extension. Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. So it is ideal for displaying live data, state

changes or alarms that get in via MQTT. Have a look at the example to see Flask-MQTT and Flask-SocketIO play together. The example provides a small publish/subscribe client using Flask-SocketIO to insantly show subscribed messages and publish messages.

```
"""

A small Test application to show how to use Flask-MQTT.

"""

import eventlet
import json
from flask import Flask, render_template
from flask_mqtt import Mqtt
from flask_socketio import SocketIO
from flask_bootstrap import Bootstrap

eventlet.monkey_patch()

app = Flask(__name__)
app.config['SECRET'] = 'my secret key'
app.config['TEMPLATES_AUTO_RELOAD'] = True
app.config['MQTT_BROKER_URL'] = 'broker.hivemq.com'
app.config['MQTT_BROKER_PORT'] = 1883
app.config['MQTT_USERNAME'] = ''
app.config['MQTT_PASSWORD'] = ''
app.config['MQTT_KEEPALIVE'] = 5
app.config['MQTT_TLS_ENABLED'] = False

# Parameters for SSL enabled
# app.config['MQTT_BROKER_PORT'] = 8883
# app.config['MQTT_TLS_ENABLED'] = True
# app.config['MQTT_TLS_INSECURE'] = True
# app.config['MQTT_TLS_CA_CERTS'] = 'ca.crt'

mqtt = Mqtt(app)
socketio = SocketIO(app)
bootstrap = Bootstrap(app)


@app.route('/')
def index():
    return render_template('index.html')


@socketio.on('publish')
def handle_publish(json_str):
    data = json.loads(json_str)
    mqtt.publish(data['topic'], data['message'])


@socketio.on('subscribe')
def handle_subscribe(json_str):
    data = json.loads(json_str)
    mqtt.subscribe(data['topic'])


@socketio.on('unsubscribe_all')
```

(continues on next page)

(continued from previous page)

```
def handle_unsubscribe_all():
    mqtt.unsubscribe_all()


@mqtt.on_message()
def handle_mqtt_message(client, userdata, message):
    data = dict(
        topic=message.topic,
        payload=message.payload.decode()
    )
    socketio.emit('mqtt_message', data=data)


@mqtt.on_log()
def handle_logging(client, userdata, level, buf):
    print(level, buf)


if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5000, use_reloader=False, debug=True)
```

2.3 Testing

For testing use the command `setup.py test`. You will need a broker like mosquitto running on your localhost, port 1883 to run the integration tests.

2.4 API Documentation

Flask-MQTT Package.

author Stefan Lehmann <stlm@posteo.de>

license MIT, see license file or https://opensource.org/licenses/MIT

class flask_mqtt.**Mqtt**(*app: flask.app.Flask = None*, *connect_async: bool = False*, *mqtt_logging: bool = False*, *config_prefix: str = 'MQTT'*)

Bases: `object`

Main Mqtt class.

Parameters

- **app** – flask application object
- **connect_async** – if True then connect_aync will be used to connect to MQTT broker
- **mqtt_logging** – if True then messages from MQTT client will be logged

init_app(*app: flask.app.Flask*, *config_prefix: str = 'MQTT'*) → None

Init the Flask-MQTT addon.

on_connect() → Callable

Decorator.

Decorator to handle the event when the broker responds to a connection request. Only the last decorated function will be called.

on_disconnect() → Callable
Decorator.

Decorator to handle the event when client disconnects from broker. Only the last decorated function will be called.

on_log() → Callable
Decorate a callback function to handle MQTT logging.

Example Usage:

```
@mqtt.on_log()
def handle_logging(client, userdata, level, buf):
    print(client, userdata, level, buf)
```

on_message() → Callable
Decorator.

Decorator to handle all messages that have been subscribed and that are not handled via the *on_message* decorator.

Note: Unlike as written in the paho mqtt documentation this callback will not be called if there exists an topic-specific callback added by the *on_topic* decorator.

Example Usage::

```
@mqtt.on_message()
def handle_messages(client, userdata, message):
    print('Received message on topic {}: {}'
          .format(message.topic, message.payload.decode()))
```

on_publish() → Callable
Decorator.

Decorator to handle all messages that have been published by the client.

Example Usage::

```
@mqtt.on_publish()
def handle_publish(client, userdata, mid):
    print('Published message with mid {}.'
          .format(mid))
```

on_subscribe() → Callable
Decorate a callback function to handle subscritions.

Usage::

```
@mqtt.on_subscribe()
def handle_subscribe(client, userdata, mid, granted_qos):
    print('Subscription id {} granted with qos {}.'
          .format(mid, granted_qos))
```

on_topic(*topic: str*) → Callable
Decorator.

Decorator to add a callback function that is called when a certain topic has been published. The callback function is expected to have the following form: *handle_topic(client, userdata, message)*

Parameters **topic** – a string specifying the subscription topic to subscribe to

The topic still needs to be subscribed via mqtt.subscribe() before the callback function can be used to handle a certain topic. This way it is possible to subscribe and unsubscribe during runtime.

Example usage::

```
app = Flask(__name__)
mqtt = Mqtt(app)
mqtt.subscribe('home/mytopic')

@mqtt.on_topic('home/mytopic')
def handle_mytopic(client, userdata, message):
    print('Received message on topic {}: {}'
          .format(message.topic, message.payload.decode()))
```

on_unsubscribe() → Callable
Decorate a callback funtion to handle unsubscribtions.

Usage::

```
@mqtt.unsubscribe()
def handle_unsubscribe(client, userdata, mid)
    print('Unsubscribed from topic (id: {})'
          .format(mid)')
```

publish(*topic: str*, *payload: Optional[bytes] = None*, *qos: int = 0*, *retain: bool = False*) → Tuple[int, int]
Send a message to the broker.

Parameters

- **topic** – the topic that the message should be published on
- **payload** – the actual message to send. If not given, or set to None a zero length message will be used. Passing an int or float will result in the payload being converted to a string representing that number. If you wish to send a true int/float, use struct.pack() to create the payload you require.
- **qos** – the quality of service level to use
- **retain** – if set to True, the message will be set as the "last known good"/retained message for the topic

Returns Returns a tuple (result, mid), where result is MQTT_ERR_SUCCESS to indicate success or MQTT_ERR_NO_CONN if the client is not currently connected. mid is the message ID for the publish request.

subscribe(*topic*, *qos: int = 0*) → Tuple[int, int]
Subscribe to a certain topic.

Parameters

- **topic** – a string specifying the subscription topic to subscribe to.
- **qos** – the desired quality of service level for the subscription. Defaults to 0.

Return type (int, int)

Result (result, mid)

A topic is a UTF-8 string, which is used by the broker to filter messages for each connected client. A topic consists of one or more topic levels. Each topic level is separated by a forward slash (topic level separator).

The function returns a tuple (result, mid), where result is MQTT_ERR_SUCCESS to indicate success or (MQTT_ERR_NO_CONN, None) if the client is not currently connected. mid is the message ID for the

subscribe request. The mid value can be used to track the subscribe request by checking against the mid argument in the on_subscribe() callback if it is defined.

Topic example: *myhome/groundfloor/livingroom/temperature*

unsubscribe(*topic: str*) → Optional[Tuple[int, int]]
Unsubscribe from a single topic.

Parameters **topic** – a single string that is the subscription topic to unsubscribe from

Return type (int, int)

Result (result, mid)

Returns a tuple (result, mid), where result is MQTT_ERR_SUCCESS to indicate success or (MQTT_ERR_NO_CONN, None) if the client is not currently connected. mid is the message ID for the unsubscribe request. The mid value can be used to track the unsubscribe request by checking against the mid argument in the on_unsubscribe() callback if it is defined.

unsubscribe_all() → None
Unsubscribe from all topics.

Returns True if all topics are unsubscribed from self.topics, otherwise False

class flask_mqtt.**TopicQos**(*topic*, *qos*)
Bases: tuple

Container for topic + qos

qos
Alias for field number 1

topic
Alias for field number 0

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index