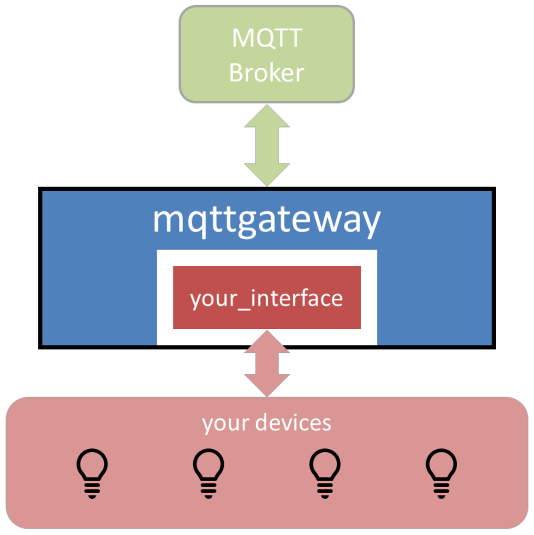
mqttgateway Documentation

Release 1.0.0

Paolo Taddonio

Nov 12, 2018

Contents



`mqttgateway` is a python wrapper to build consistent gateways to an MQTT network.

CHAPTER 1

What it does:

- it deals with all the boilerplate code to manage an MQTT connection, load configuration and other data files, and create log handlers,
- it encapsulates the interface in a class that needs only 2 methods `__init__` and `loop`,
- it creates an intuitive messaging abstraction layer between the wrapper and the interface,
- it can isolate the syntax and keywords of the MQTT network from the internal ones of the interface.

CHAPTER 2

Who is it for:

Developers of MQTT networks in a domestic environment looking to adopt a definitive syntax for their MQTT messages and to build gateways with their devices that are not MQTT enabled.

CHAPTER 3

Available interfaces

Check the existing fully developped interfaces. Their names usually follows the pattern **<interface_name>2mqtt**, for example musiccast2mqtt.

This library comes with 2 interfaces:

- **dummy**: to test the environment and use as a template;
- **entry**: example used for the tutorial.

CHAPTER 4

Links

- **Documentation** on readthedocs.
- **Source** on github.
- **Distribution** on pypi.

CHAPTER 5

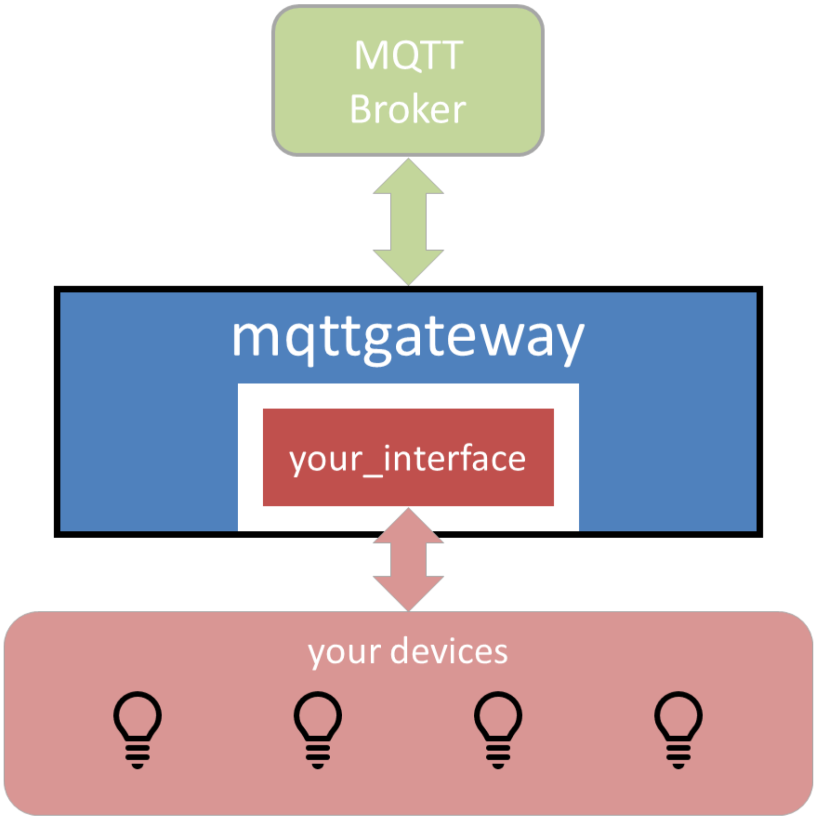
Contents

5.1 Overview

5.1.1 Objective

When setting up an IoT eco-system with a lot of different devices, it becomes quickly difficult to have them talking to each other. A few choices need to be made to solve this problem. This project assumes that one of those choices has been made: using MQTT as the messaging transport. This project then intends to help in the next set of choices to make: defining a messaging model and expressing it in an MQTT syntax to be shared by all devices.

This model is implemented as a python library aimed at facilitating coding the gateways between devices that do not support natively MQTT communication and the MQTT network. These gateways can then run as services on machines connected to these devices via whatever interface is available: serial, bluetooth, TCP, or else.



5.1.2 Concepts

This project has two parts:

1. The definition of the messaging model. It is an abstraction layer that defines a message not only by destination and content but by a few attributes adapted to domestic IoT environments.
2. The implementation of this model through a python library. The library takes care of formatting and translating back and forth the messages between their MQTT syntax and their internal representation.

For more information, go to *Concepts*.

5.1.3 Usage

This project is provided with the core library, and an example interface (the **dummy** interface) that does not interface with anything but shows how the system works. Once installed, running the application `dummy2mqtt` allows to test the basic configuration and show how it is reacting to incoming MQTT messages, for example.

Developers can then write their own interface by using the **dummy** interface as a template, or following the tutorial alongside the theoretical interface **entry**.

End users will download already developped interfaces, for which this library will simply be a dependency.

For a complete guide on how to develop an interface, go to *Tutorial*.

5.1.4 Installation

The installation can be done with pip, on both Linux and Windows systems. The only dependency is the paho.mqtt library.

For the full installation guide, go to *Installation*.

5.2 Installation

5.2.1 Download

Get the library from the PyPi repository with the pip command, preferrably using the --user option:

```
pip install --user mqttgateway
```

Alternatively use the *bare* pip command if you have administrator rights or if you are in a virtual environment.

```
pip install mqttgateway
```

Running pip also installs an executable file (exe in Windows or executable python script in Linux) called dummy2mqtt. It launches the demo interface **dummy** with the default configuration. Its location *should* be **UPDATE NEEDED HERE** on Windows and **UPDATE NEEDED HERE** on Linux. If not, search for the file manually.

Also, those same locations *should* be already defined in the **PATH** environment variable and therefore the executable *should* launch from any working directory. If not, the variable will have to be updated manually, or the executable needs to be specified with its real path.

5.2.2 Configuration

A configuration file is needed for each interface. In the library, the default interface dummy has its own configuration file dummy2mqtt.conf inside the package folder dummy.

The configuration file has a standard INI syntax, with sections identified by [SECTION] and options within sections identified by option:value. Comments are identified with a starting character #.

There are four sections:

1. [MQTT] defines the MQTT parameters, most importantly the IP address of the broker under the option host. The address of the MQTT broker should be provided in the same format as expected by the **paho.mqtt** library, usually a raw IP address (192.168.1.55 for example) or an address like *test.mosquitto.org*. The default port is 1883, if it is different it can also be indicated in the configuration file under the option port. Authentication is not available at this stage.
2. [LOG] defines the different logging options. The library can log to the console, to a file, send emails or just send the logs to the standard error output. By default it logs to the console.
3. [INTERFACE] is the section reserved to the actual interface using this library. Any number of options can be inserted here and will be made available to the interface code through a dictionary initialised with all the option:value pairs.
4. [CONFIG] is a section reserved to the library to store information about the configuration loading process. It is not visible in the template files but it is created at runtime.

For more details about the .conf file, its defaults and the command line arguments, go to *Configuration*.

5.2.3 Launch

If `pip` installed correctly the executable files, just launch it from anywhere:

```
dummy2mqtt
```

By default, the process will log messages from all levels into the console. It should start printing a banner message to indicate the application has started, then a list of the full configuration used.

If the MQTT connection is successful it should say so as well as displaying the topics to which the application has subscribed.

5.2.4 First Run

After the start-up phase, the **dummy** interface logs (at a DEBUG level) any MQTT messages it receives. It also emits a unique message every 30 seconds. Start your favourite MQTT monitor app (I use the excellent mqtt-spy). Connect to your MQTT broker and subscribe to the topic:

```
home/+/dummy/+/+/+/C
```

You should see the messages arriving every 30 seconds in the MQTT monitor, as well as in the log.

Publish now a message from the MQTT monitor:

```
topic: home/lighting/dummy/office/undefined/me/C
payload: LIGHT_ON
```

You should see in the log that the message has been received by the gateway, and that it has been processed correctly, meaning that even if it does not do anything, the translation methods have worked.

5.2.5 The mapping data

The mapping data is an optional feature that allows to map some or every keyword in the MQTT vocabulary into the equivalent keyword in the interface. This mapping is a very simple one-to-one relationship between keywords of each characteristic, and its use is only to isolate the internal code from any changes in the MQTT vocabulary. For the **dummy** interface, the mapping data is provided by the text file `dummy_map.json`. It's just there as an example, as, once again, the **dummy** interface really doesn't do anything, and it is disabledby default. Note that the map file also contains the *root* of the MQTT messages and the topics that the interface should subscribe to.

For more details on the mapping data, go to *Concepts*.

5.2.6 Deploying a gateway

The objective of developing a gateway is to ultimately deploy it in a production environment. To install a gateway as a service on a linux machine, go to *Configuration*.

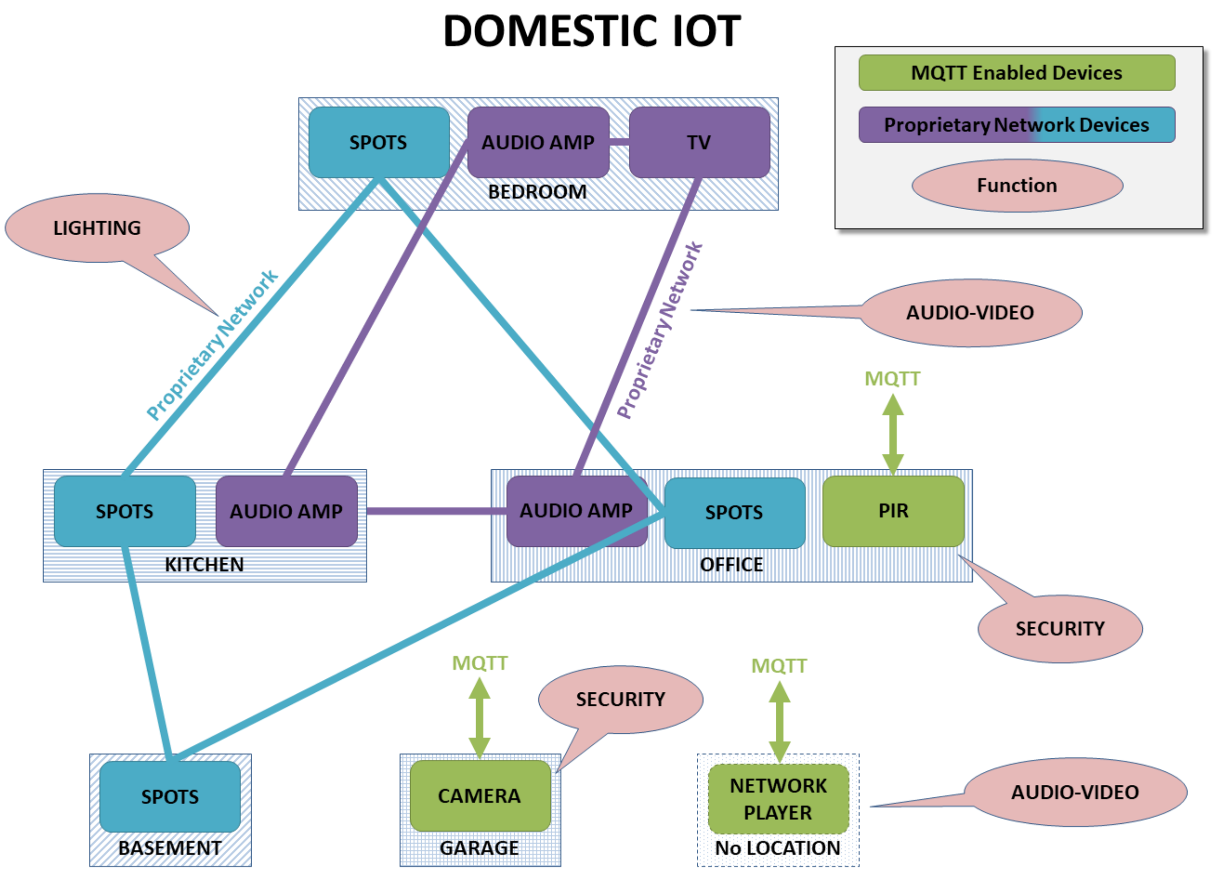
5.3 Concepts

5.3.1 The message model

The primary use case for this project is a domestic environment with multiple devices of any type: lights, audio video components, security devices, heating, air conditioning, controllers, keypads, etc… For many (good) reasons, MQTT has been selected as the communication protocol. But only a few, if any, devices are MQTT enabled. For those that are not, there is a need to develop ad-hoc gateways to *bridge* whatever interface they use natively (serial for example) to one that is MQTT based. Even for those devices that communicate natively through MQTT, there is a need to agree on a syntax that makes the exchange of messages coherent.

Example

In the example below, a smart home has some lighting connected in four different rooms through a proprietary network, four audio-video devices connected through another proprietary network, and some other devices that are already MQTT-enabled, but which still need to speak a common language.



One of the objectives of this project is not only to define a common MQTT syntax, but also to make it as *intuitive* as possible. Ideally, a human should be able to write an MQTT message off-hand and operate successfully any device in the network.

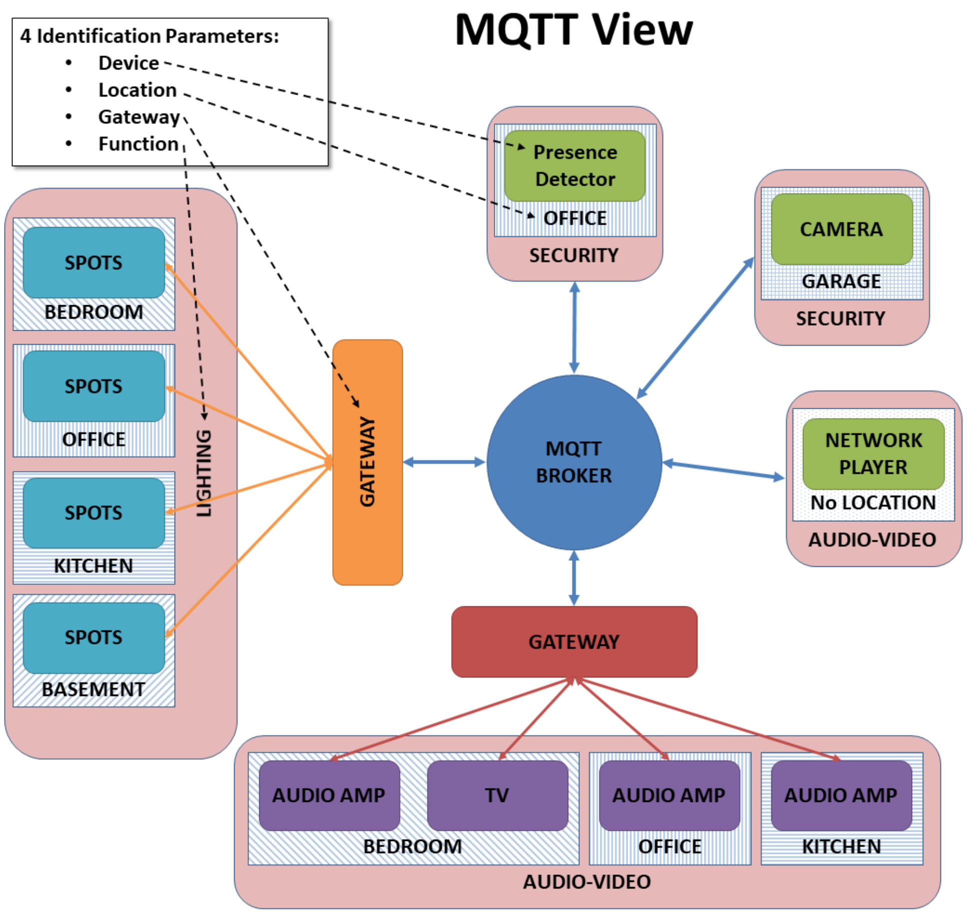
Message Addressing

The first step of any message is to define its destination. A flexible addressing model should allow for a heuristic approach based on a combination of characteristics of the recipient, on top of the standard deterministic approach (e.g. a unique device id). Four characteristics are usually considered:

- the **function** of the device: lighting, security, audio-video, etc;
- its **location**;

- its **gateway**: which application is managing that device, if any;
- the name of the **device**.

In our example, the MQTT point of view shows how those four characteristics, or just a subset, can define all the devices in the network.



Some considerations about those four characteristics:

- not all four characteristics need to be provided to address succesfully a device;
- the **device** name can be generic (e.g. `spotlight`) or specific and unique within the network (e.g. `lightid1224`); in the generic case, obviously other characteristics are needed to address the device.
- any device can have more than one value for each characteristics, particularly the **function** and **device** ones (it is probable that the **gateway** and the **location** characteristics are unique for a given device);
- the **location** is important and probably the most intuitive characteristic of all; preferably it should represent the place where the device operates and not where it is physically located (e.g. an audio amplifier might be in the basement but it powers speakers in the living room; the location should be the living room); but the location might even not be defined (e.g. to address the security system of the whole house, or an audio network player that can broadcast to different *channels* or *zones*).
- the **gateway** is the most deterministic characteristic (alongside a unique device id); this should be the chosen route for fast and unambiguous messaging.
- the **function** is another important intuitive characteristic; not only it helps in addressing devices (combined with a location for example), but it also clarifies ambiguous commands (e.g. `POWER_ON` with `lighting` or with

`audiovideo` means different things). However things can get more complicated if a device has more than one function; this should be allowed, it is up to the gateway to make sure any ambiguity is resolved from the other characteristics.

Those four characteristics should ensure that the messaging model is flexible enough to be heuristic or deterministic. A gateway will decide how flexible it wants to be. If it has enough *processing bandwidth*, it can decide to subscribe to all **lighting** messages for example, and then parse all messages received to check if they are actually addressed to it. Or it can subscribe only to messages addressed specifically to itself (through the gateway name for example), restricting access only to the senders that know the name of that gateway.

Message Content

The content of a message in the context of domestic IoT can be modelled in many different ways. This project splits it into 3 *characteristics*:

- a **type** with 2 possible values: *command* for messages that are requiring an action to be performed, or *status* for messages that only broadcast a state;
- an **action** that indicates what to do or what the status is referring to;
- a set of **arguments** that might complete the **action** characteristic.

The key characteristic here is the **action**, a string representing the *what* to do, with the optional **arguments** helping to define by *how much* for example. It can be `POWER_ON` and `POWER_OFF` on their own for example (no argument), or `SET_POWER` with the argument `power:ON` or `power:OFF`, or both. The interface decides what actions it recognises, the more the better.

Message Source

The sender, which can be a device or another gateway for example, is an optional characteristic in our message model. It can be very useful in answering status requests in a targeted way, for example.

5.3.2 Bridging MQTT and the interface

There are therefore a total of 8 characteristics in our message model:

- **function**,
- **gateway**,
- **location**,
- **device**,
- **type**,
- **action**,
- **argument** of action,
- **sender**.

They are all strings except **type** which can only have 2 predefined values. They are all the fields that can appear in a MQTT message, either in the topic or in the payload. They are all attributes of the internal message class that is used to exchange messages between the library and the interface being developed. They are all the characteristics available to the developer to code its interface.

The internal message class

The internal message class `internalMsg` defines the message objects stored in the lists that are shared by the library and the interface. There is a list for incoming messages and a list for outgoing messages. At its essence, the library simply parses MQTT messages into internal ones, and back. The library therefore defines the MQTT syntax by the way it converts the messages.

The conversion process

The conversion process happens inside the class `msgMap` with the methods `MQTT2Internal()` and `Internal2MQTT()`. These methods achieve 2 things:

- define the syntax of the MQTT messages in the way the various characteristics are positioned within the MQTT topic and payload;
- if mapping is enabled, map the keywords for every characteristic between the MQTT *vocabulary* and the internal one; this is done via dictionaries initialised by a *mapping file*.

The MQTT syntax

The library currently defines the MQTT syntax as follows. The topic is structured like this:

```
root/function/gateway/location/device/source/type
```

where `root` can be anything the developer wants (`home` for example) and `type` can be only `C` or `S`.

The payload is simply the action alone if there are no arguments:

```
action_name
```

or the action with the arguments all in a JSON string like this:

```
{"action":"action_name","arg1":"value1","arg2":"value2",...}
```

where the first `action` key is written as is and the other argument keys can be chosen by the developer and will be simply copied in the **argument** dictionary.

This syntax is defined within the 2 methods doing the conversions. The rest of the library is agnostic to the MQTT syntax. Therefore one only needs to change these 2 methods to change the syntax. However in that case, all the devices and other gateways obviously have to adopt the same new syntax.

The mapping data

By default, when the mapping option is disabled, the keywords used in the MQTT messages are simply copied in the internal class. So, for example, if the **function** in the MQTT message is `lighting`, then the attribute `function` in the `internalMsg` will also be `lighting`. If for any reason a keyword has to change on either *side*, it has to be reflected on the other one, which is unfortunate. For example, let's assume a location name in the MQTT vocabulary is `basement` and that is what is used in the internal code of the interface to start with. For some reason the name in the MQTT vocabulary needs to be changed to `lowergroundfloor`. In order for the interface to recognise this new keyword, a mapping can be introduced that links the keyword `lowergroundfloor` in the MQTT messages to `basement` in the internal representation of messages. This mapping is defined in a separate JSON file, and the code does not need to be modified.

The mapping option can be enabled (it is off by default) in the configuration file, in which case the location of the JSON file is required. All the keyword characteristics (except **type**) can (but do not have to) be mapped in that file:

function, **gateway**, **location**, **device**, **sender**, **action**, **argument keys** and **argument values**. To give more flexibility, there are 3 mapping options available for each characteristic that need to be specified:

- `none`: the keywords are left unchanged, so there is no need to provide the mapping data for that characteristic;
- `strict`: the conversion of the keywords go through the provided map, and any missing keyword raises an exception; the message with that keyword is probably ignored;
- `loose`: the conversion of the keywords go through the provided map, but missing keywords do not raise any error but are passed unchanged.

The mapping between internal keywords and MQTT ones is a one-to-many relationship for each characteristic. For each internal keyword there can be more than one MQTT keyword, even if there will have to be one which has *priority* in order to define without ambiguity the conversion from internal to MQTT keyword. In practice, this MQTT keyword will be the first one in the list provided in the mapping (see below) and the other keywords of that list can be considered *aliases*. Going back to the example above, for the unique internal location keyword `basement`, we could define a list of MQTT keywords as `["lowergroundfloor", "basement"]`, so that `basement` in internal code gets converted to `lowergroundfloor` in MQTT (as it is the new *official* keyword) but `basement` in MQTT is still accepted as a keyword that gets *converted* to `basement` in internal messages.

In practice, the mapping data is provided by a JSON formatted file. The JSON schema `mqtt_map_schema.json` is available in the `gateway` package. New JSON mapping files can be tested against this schema (I use the online validation tool https://www.jsonschemavalidator.net/) The mapping file also contains the topics to subscribe to and the root token for all the topics. These values override the ones found in the configuration file if the mapping feature is enabled.

5.4 Tutorial

Let's go through a practical example, with a very simple protocol.

5.4.1 The Need

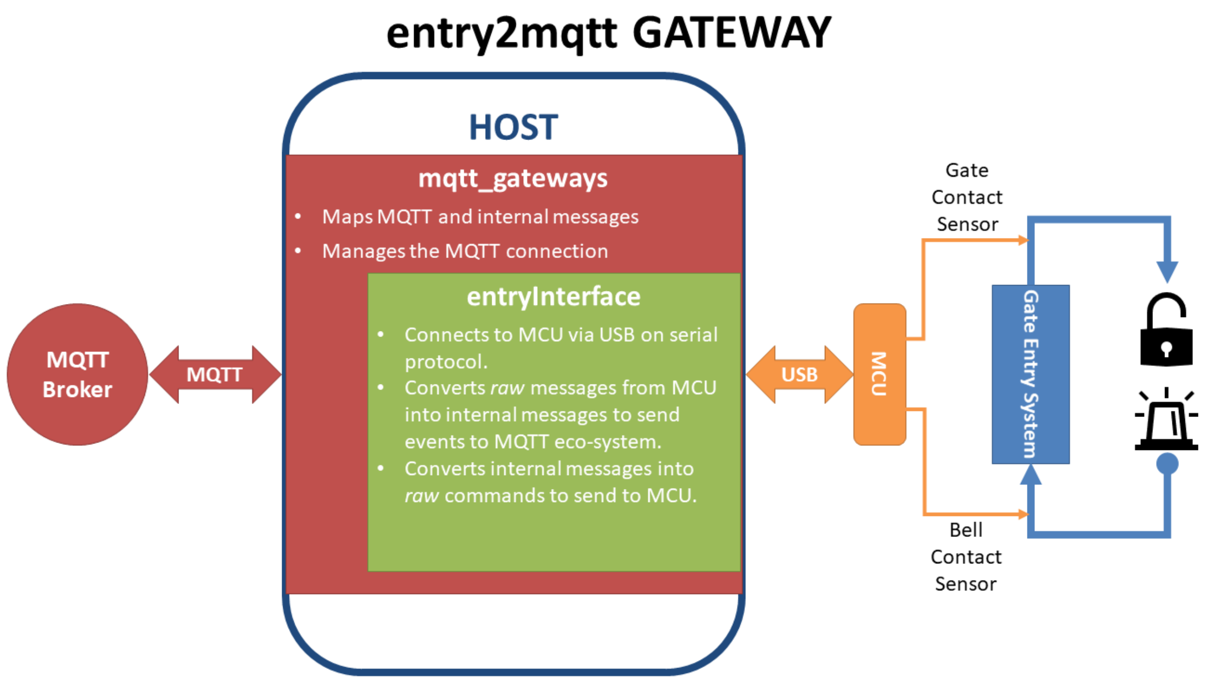
The gate of the house has an entry system, or intercom. Visitors push the bell button, and if all goes well after a brief conversation someone in the house let them in by pushing a gate release button. Residents have a code to let themselves in: they enter the code and the system releases the gate.

It would be nice to receive messages about these events, so that other events can be triggered (like switching on lights by night). It would also be nice to trigger the gate release independently of the entry system.

5.4.2 The Solution

We assume the entry system exposes the electrical contacts that operate the bell and the gate. A micro-controller (an Arduino for example), can sense the electrical contacts going HIGH or LOW and can communicate these levels to a computer through a serial port. The micro-controller can also be told to switch ON or OFF a relay to release the gate. We will call `Entry System` the combination of the actual entry system with the micro-controller physical interface.

Note: a computer with the right sensors like a Raspberry Pi could sense directly the electrical contacts without being shielded by another board. However this use-case suits the tutorial, and is probably more reliable in the long run.


entry2mqtt GATEWAY
HOST
mqtt_gateways
• Maps MQTT and internal messages
• Manages the MQTT connection
entryInterface
• Connects to MCU via USB on serial protocol.
• Converts *raw* messages from MCU into internal messages to send events to MQTT eco-system.
• Converts internal messages into *raw* commands to send to MCU.
MQTT Broker
MQTT
USB
MCU
Gate Contact Sensor
Gate Entry System
Bell Contact Sensor

5.4.3 Implementation

The micro-controller is programmed to communicate with very simple messages for each event: each message is a pair of digits (in ASCII), the first indicating which contact the message is about and the second indicating its state. With 2 contacts (the bell and the gate), and 2 states (ON and OFF), there are only 4 messages to deal with: `10`, `11`, `20` and `21`. More precisely, the micro-controller:

- sends a message when a contact goes ON (`11` or `21`) and another one when it goes off (`10` or `20`);
- can also receive and process messages; in our case only the one triggering the gate release makes sense (let's say it is the *message* `21`); we will assume that the micro-controller turns the gate release OFF automatically after 3 seconds, for security, so there is no need to use the gate release OFF message (`20`); similarly, there is no need to process the messages `11` or `10` as there is no need to operate the bell via MQTT.

The next step is therefore to code the interface for the computer connected to the micro-controller. Let's call the interface **entry**. This will be the label used in all the names in the project (packages, modules, folders, class, configuration and mapping files).

5.4.4 The interface

The interface will be a Python package called `entry2mqtt`. Let's create it in a new folder `entry` with an empty module `__init__.py`. In order not to start from scratch, let's use the `dummy` interface as a template. Copy the files `dummy_start.py` and `dummy_interface.py` from the `dummy` package into the `entry` package, and change all the `dummy` instances into `entry` (in the name of the file as well as inside the file). The actual interface code has to be in the class `entryInterface` within the module `entry_interface.py`. It needs to have at least a constructor `__init__` and a method called `loop`.

The constructor

The constructor receives 3 arguments: a dictionary of parameters and two message lists, one for incoming messages and the other one for outgoing ones.

The dictionary of parameters is loaded with whatever we put in the configuration file in the `[INTERFACE]` section. It's up to us to decide what we put in there. Here we probably only need a *port* name in order to open the serial port. We will create the configuration file later, but for now we will assume that there will be an option `port:what_ever_it_is` in the `[INTERFACE]` section, so we can retrieve it in our code.

The constructor will generally need to keep the message lists locally so that the `loop` method can access them, so they will be assigned to local members.

Finally, the constructor will have to initialise the serial communication.

Starting from the template copied above, the only thing to add is the opening of the serial port. Add at the top of the module:

```
import serial
```

(you need to have the PySerial library in your environment), and add the following line inside the constructor:

```
self._ser = serial.Serial(port=port, baudrate=9600, timeout=0.01)
```

The `port` variable is already defined in the template (check the code). The `baudrate` has to be the same as the one set by the micro-controller. Finally the `timeout` is fundamental. It has to be short enough so that the main loop is not delayed too much. Without timeout, all the serial exchanges will be blocking, which can not work in our *mono-thread* process.

Note: It is obviously possible to use *natively* multiple threads for the library to avoid the blocking calls issues. Indeed, the paho library is already doing so for its part. However this is not the case for now even if it might be implemented in the future.

The `loop` method

This method is called periodically by the main loop to let our interface do whatever it needs to do.

The `loop` method should deal with the incoming messages first, process them, then *read* its own connected device for events, process them and stack in the outgoing list whatever message needs to be sent, if there are any.

Use the code in the template to read the incoming messages list and add the following code to deal with the case where the message is a command to open the gate:

```
if msg.action == 'GATE_OPEN':
    try:
        self._ser.write('21')
    except serial.SerialException:
        self._logger.info('Problem writing to the serial interface')
```

Always try to catch any exception that should not disrupt the whole application. Most of them should not be fatal.

Then read the serial interface to see if there are any events:

```
try:
    data = self._ser.read(2)
except serial.SerialException:
    self._logger.info('Problem reading the serial interface')
    return
if len(data) < 2:
    return
```

If there is an event, convert it into an internal message and add it to the outgoing message list:

```
if data[0] == '1':
    device = 'Bell'
    if data[1] == '0':
        action = 'BELL_OFF'
    elif data[1] == '1':
        action = 'BELL_ON'
    else:
        self._logger.info('Unexpected code from Entry System')
        return
elif data[0] == '2':
    device = 'Gate'
    if data[1] == '0':
        action = 'GATE_CLOSE'
    elif data[1] == '1':
        action = 'GATE_OPEN'
    else:
        self._logger.info('Unexpected code from Entry System')
        return
msg = internalMsg(iscmd=False, # it is a status message
                  function='Security',
                  gateway='entry2mqtt',
                  location='gate_entry',
                  device=device,
                  action=action)
self._msgl_out.append(msg)
```

Finally, let's send a command to switch on the light in case the gate was opened:

```
if data == '21':
    msg = internalMsg(iscmd=True,
                      function='Lighting',
                      location='gate_entry',
                      action='LIGHT_ON')
    self._msgl_out.append(msg)
```

That's it for the basic logic.

Other coding strategies

The resulting code is as simple as it can be. There are clearly more advanced *coding strategies* to make the code more *elegant* and ultimately easier to mantain and upgrade.

For example, the class can be defined as a subclass of the Serial class, as this would reflect well what it actually is, i.e. a higher level serial interface to a specific device.

Another possibility is to code the conversion of the messages from the serial interface into internal messages through lookup tables (dictionaries) instead of nested *ifs*.

There are always better ways to code, but it is important to note that, as the loop is supposed to run fast and is the piece of code that will run forever, it is worth investing some time on how to make that part more efficient.

The map file

The mapping feature is disabled by default. This means that all the keywords introduced earlier in the code (the commands GATE_OPEN, GATE_CLOSE, BELL_ON and BELL_OFF, as well as the location gate_entry and the function identifiers Security``and ``Lighting) will all be passed on **as is** to the MQTT messages, with exactly the same spelling and the same capitalised letters, if any. This might be acceptable if there are only a few devices and gateways in the MQTT network and the *vocabulary* stays quite small. But if the network grows and evolves, inevitably changes will happen and it becomes inpractical to have to change the code any time an identifier in the MQTT vocabulary had to change. That is where the mapping feature steps in.

The mapping feature can be enabled in the configuration file, in which case a file location for the map needs to be provided:

```
...
mapping: on
mapfilename: /the/path/to/your/mapfile/filename.json
```

The map file location option is subject to the various *rules* for file paths used in this library. In a nutshell, if the path is absolute there is no ambiguity, if it is relative the library will try the path starting from the configuration file directory first, then try the current working directory of the process, and finally the directory of the launching script.

The mapping file is a JSON formatted file with 2 objects (the root of the MQTT messages and a list of topics to subscribe to) and up to 8 dictionaries, 1 for each characteristic that can potentially be mapped. For each characteristic, a maptype needs to be provided (it can be either none, loose or strict) and then an actual map, if the maptype is not none.

For our interface, we assume we want to map all the data, as shown in the table:

Table 1: Data to map for the entry gateway

Characteritic	MQTT Keyword	Interface Keyword
function	security	Security
function	lighting	Lighting
gateway	entry2mqtt	entry2mqtt
location	frontgarden	gate_entry
device	gate	Gate
device	bell	Bell
action	gate_open	GATE_OPEN
action	bell_off	BELL_OFF
action	bell_on	BELL_ON
action	light_off	LIGHT_OFF
action	light_on	LIGHT_ON
action	gate_close	GATE_CLOSE

The map file would then look like this:

```
{
    "root": "home",
    "topics": ["home/security/+/frontgarden/+/+/C",
               "home/+/entry2mqtt/+/+/+/C",
               "home/+/+/+/entrysystem/+/C"],
    "function": {
```

(continues on next page)

(continued from previous page)

```
        "map": {"security": "Security", "lighting": "Lighting"},
        "maptype": "strict"
    },
    "gateway": {
        "map": {"entry2mqtt": "entry2mqtt"},
        "maptype": "strict"
    },
    "location": {
        "map": {"frontgarden": "gate_entry"},
        "maptype": "strict"
    },
    "device": {
        "map": {"gate": "Gate", "bell": "Bell"},
        "maptype": "strict"
    },
    "sender": {"maptype": "none"},
    "action": {
        "map": {"gate_open": "GATE_OPEN",
                "bell_off": "BELL_OFF",
                "bell_on": "BELL_ON",
                "light_off": "LIGHT_OFF",
                "light_on": "LIGHT_ON",
                "gate_close": "GATE_CLOSE"
        },
        "maptype": "strict"
    },
    "argkey": {"maptype": "none"},
    "argvalue": {"maptype": "none"}
}
```

Save it in a file named `entry_map.json`.

A few comments on this *suggested* mapping:

- most of these keyword mappings only change the case or even nothing; this is for illustration purposes anyway, but in general it might still be good discipline to list all the keywords in a mapping to have in one view what the interface can deal with or not. Then if one day some MQTT keyword needs to change, everything is ready to do so.
- an important choice to make is the `maptype` for each characteristic. If it is set to `strict`, it will enable to filter messages quite early in the process and alleviate the code of further testing. In our example, even if the `gateway` map has only one item, which is even the same on both sides, setting the `maptype` to `strict` ensures that **only** that keyword is accepted, and any other one is discarded. This is obviously very different from setting the `maptype` to `none`, in which case that only keyword would still be accepted and left unchanged, but so would any other keyword.

5.4.5 Wrapping it all up

Once the interface is defined, all is left to do is to create the launch script and the configuration file. Those 2 steps are easy using the templates.

Copy the **dummy** project launch script `dummy_start.py` and paste it into the `entry` directory. Change every instance of `dummy` into `entry``. If all the naming steps have been respected, the script `entry_start.py` just created should work.

To create the configuration file, copy the configuration file `dummy2mqtt.conf` from the `dummy` package and paste it in the folder `entry` with the name `entry2mqtt.conf`. Edit the file and enter the `port` option under the

`[INTERFACE]` section:

```
[INTERFACE]
port=/dev/ttyACM0
```

Obviously input whatever is the correct name of the port, the one shown is generally the one to use on a Raspberry Pi for the USB serial connection. If you are on Windows, your port should be something like `COM3`.

If you went through the *installation* process, then the MQTT parameters should already be set up, otherwise do so. Other parameters can be left as they are. Check the *configuration* guide for more details.

5.4.6 Launch

To launch the gateway, just run the launcher script directly from its directory:

```
python entry_start
```

Done!

5.5 Configuration

Note: Coming soon!

In the meantime, the default configuration, which is in the python module `configuration.py` inside the `gateway` package, is well documented and is a good starting point to understand the various options.

```
''' The default configuration settings in a single string constant.

.. reviewed 29May2018

.. any changes in CONFIG should be reflected in the dummy2mqtt.conf file.

Use this declaration as a template configuration file.

The configuration loader :mod:`mqttgateway.utils.load_config` only
considers sections and options that are already present in this string
and disregard anything else.  If a section or option is found in a configuration
file but is not listed here, it will not be taken into account.  Only
sections and options already here are taken into account when found in a configuration
file, and then they overwrite the default values defined here.

The only exception is the ``[INTERFACE]`` section which is reserved to the␣
↪configuration
parameters needed by the interface being implemented.  The section itself is defined␣
↪here
but no options are present as those are defined by the developper, and those are the␣
↪only
options that the loader will take into account.

'''

CONFIG = '''
# ----------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
[CONFIG]
# Reserved section used by the loader to store the location where
#   the configuration settings are coming from, or to store
#   the error if there was one.

# ------------------------------------------------------------------------------
[INTERFACE]
# Section for whatever options are needed by the gateway interface
#   being developed. All these options will be written in a
#   dictionary and passed to the interface.

# ------------------------------------------------------------------------------
[MQTT]
# The parameters to connect to the MQTT broker
host: 127.0.0.1
port: 1883
keepalive: 60

# The client id can be provided here; if left empty it defaults to the application␣
↪name
clientid:

# This is the timeout of the 'loop()' call in the MQTT library
timeout: 0.01

# Mapping option. By default it is off.
mapping: off

# Map file: there needs to be a mapping file if the <mapping> option is on.
#   If the <mapfilename> option is left blank, the mapping option is turned
#   off, whatever the value of the <mapping> option.
#   To use the default name and path, use a dot <.> for this option.
#   The default name used is <*application_name*.map>.
#   See below for other instructions on file names and paths.
mapfilename: .

# The 'root' keyword for all MQTT messages.
#   Only necessary if <mapping> is off, disregarded otherwise
#   as the keyword should then be found in the mapping file.
root: home

# The topics to subscribe to, separated by a comma.
#   Only necessary if <mapping> is off, disregarded otherwise
#   as the topics should then be found in the mapping file.
topics: home/dummyfunction/#, home/+/dummy/#

# ------------------------------------------------------------------------------
[LOG]
# Logs: all WARN level logs and above are sent to stderr or equivalent.
#   3 more log outputs can be set up: console, rotating files and email.
# Log levels: indicate what log levels are required for each log output.
#   Levels are indicated with the following strings (from the logging module):
#   CRITICAL, ERROR, WARN or WARNING, INFO and DEBUG; use NONE if unused.

# Console level: these are the logs directed to stdout.  Usually only used for␣
↪testing.
consolelevel: NONE
```

(continues on next page)

(continued from previous page)

```
# Log file: file location if logs to file is required.
#   Leave this option blank to not enable a log file.
#   Use a dot <.> to use the default name and path.
#   The default name used is <*application_name*.log>.
#   Make sure the process will have the rights to write in this file.
#   See below for other instructions on file names and paths.
logfilename:

# File level: level for logs directed to the file named by the <logfilename> option.
#   If that option is blank, there is not file log whatever value is given to the␣
↪option
#   <filelevel> (there is no default file).
filelevel: INFO

# Number of files required for the rotating files. Default is 3.
filenum:3

# Maximum size of each log file, in KB. Default is 50'000.
filesize: 50000

# Email credentials; leave empty if not required.
#   All CRITICAL level logs are sent to this email, if defined.
#   For now there is no authentication.
emailhost:
# for example: emailhost: 127.0.0.1
emailport:
# for example: emailport: 25
emailaddress:
# for example: address: me@example.com

#------------------------------------------------------------------------------
# Note on file paths and names:
#   - the default name is 'application_name' +
#                           default extension (.log, .map, ... etc);
#   - the default directories are (1) the configuration file location, (2) the
#     current working directory, (3) the application directory, which
#     'should' be the location of the launching script;
#   - empty file paths have different meaning depending where they are used;
#     best to avoid;
#   - file paths can be directory only (ends with a '/') and are appended with
#     the default name;
#   - file paths can be absolute or relative; absolute start with a '/' and
#     relative are prepended with the default directory;
#   - file paths can be file only (no '/' whatsoever) and are prepended with
#     the default directory;
#   - use forward slashes '/' in any case, even for Windows systems, it should
#     work;
#   - however for Windows systems, use of the drive letter might be an issue
#     and has not been tested.
#------------------------------------------------------------------------------

'''
```

5.6 Project Decription

Note: Coming soon!

5.7 mqttgateway package

5.7.1 Warning

As of 24 May 2018, most of the docstrings are obsolete. They will be updated gradually as soon as possible.

5.7.2 Subpackages

mqttgateway.dummy package

Submodules

mqttgateway.dummy.dummy_start module

Launcher script for the **dummy** interface.

Use this as a template. If the name conventions have been respected, just change all occurrences of `dummy` into the name of your interface.

`mqttgateway.dummy.dummy_start.`**`main`**`()`
The entry point for the application

mqttgateway.dummy.dummy_interface module

The **dummy** interface class definition. Use it as a template.

This module defines the class *`dummyInterface`* that will be instantiated by the main module in the `gateway` package.

`class` `mqttgateway.dummy.dummy_interface.`**`dummyInterface`**`(`*params*, *msglist_in*, *msglist_out*`)`

Bases: `object`

Doesn't do anything but provides a template.

The minimum requirement for the interface class is to define 2 public methods:

- the constructor `__init__`,
- the `loop` method.

Parameters

- **`params`** (*`dictionary of strings`*) – contains all the options from the configuration file This dictionary is initialised by the `[INTERFACE]` section in the configuration file. All the options in that section generate an entry in the dictionary. Use this to pass parameters from the configuration file to the interface, for example the name of a port, or the speed of a serial communication.

- **(** (*msglist_out*) – class:MsgList): list of incoming messages (internal representation)
- **(** – class:MsgList): list of outgoing messages (internal representation)

loop()

The method called periodically by the main loop.

Place here your code to interact with your system.

Module contents

A dummy gateway to test the installation setup, the loading of the configuration files, and the basic operation of the core application.

mqttgateway.entry package

Submodules

mqttgateway.entry.entry_start module

mqttgateway.entry.entry_interface module

Module contents

Package of the **entry** gateway

mqttgateway.gateway package

Submodules

mqttgateway.gateway.configuration module

The default configuration settings in a single string constant.

Use this declaration as a template configuration file.

The configuration loader *mqttgateway.utils.load_config* only considers sections and options that are already present in this string and disregard anything else. If a section or option is found in a configuration file but is not listed here, it will not be taken into account. Only sections and options already here are taken into account when found in a configuration file, and then they overwrite the default values defined here.

The only exception is the `[INTERFACE]` section which is reserved to the configuration parameters needed by the interface being implemented. The section itself is defined here but no options are present as those are defined by the developper, and those are the only options that the loader will take into account.

mqttgateway.gateway.mqtt_client module

This is a child class of the MQTT client in the paho library.

It includes the management of reconnection when using only the loop() method (which is not included natively in the paho library).

Notes on MQTT behaviour:

- if not connected, the *loop* and *publish* methods will not do anything, but raise no errors either.
- the *loop* method handles always only one message per call.

exception `mqttgateway.gateway.mqtt_client.`**`connectionError`**(*msg=None*)

Bases: *`mqttgateway.utils.throttled_exception.ThrottledException`*

Base Exception class for this module, inherits from ThrottledException

class `mqttgateway.gateway.mqtt_client.`**`mgClient`**(*host='localhost', port=1883, keepalive=60, client_id='', on_msg_func=None, topics=None, userdata=None*)

Bases: `paho.mqtt.client.Client`

Class representing the MQTT connection. `mg` means `MqttGateway`.

Note: The MQTT paho library sets quite a few attributes in the Client class. They all start with an underscore. Be careful not to overwrite them.

Parameters

- **`on_msg_func`** (*`function`*) – takes an MQTT message as argument and is called during on_message().
- **`topics`** (*`list of strings`*) – e.g.['home/audiovideo/#', 'home/lighting/#'].

`lag_end`()

Function to inhibit the connection test during the lag.

There is the possibility of a race condition when testing the connection state too soon after requesting a connection. This happens if the `on_connect()` call-back is not called fast enough and the main loop tests the connection state before that call-back has set the state to `connected`. As a consequence the automatic reconnection feature gets triggered while a connection is already under way, and the connection process gets jammed with the broker. That's why we need to leave a little lag before testing the connection.

`connect`()

Sets up the 'lag' feature on top of the parent method.

`reconnect`()

Sets up the 'lag' feature on top of the parent method.

`loop`(*timeout*)

Implements automatic reconnection on top of the parent loop method.

The use of the method/attribute `lag_test()` is to avoid having to test the lag forever once the connection is established. Once the lag is finished, this method gets replaced by a simple lambda, which hopefully is much faster than calling the time library and doing a comparison. Probably a case of premature optimisation though...

mqttgateway.gateway.mqtt_map module

This module is the bridge between the internal and the MQTT representation of messages.

As a reminder, we define the MQTT syntax as follows:

- topic: `root/function/gateway/location/device/sender/type-{C or S}`
- payload: action or status, in plain text or in a json string e.g. `{key1:value1,key2:value2,..}`

class `mqttgateway.gateway.mqtt_map.`**`internalMsg`**(*iscmd=False*, *function=None*, *gateway=None*, *location=None*, *device=None*, *sender=None*, *action=None*, *arguments=None*)

Bases: `object`

Defines all the characteristics of an internal message.

Behaviour of `None`: even if it could be interesting to differentiate between a characteristic set to `None` and one set to an empty string (an empty string could still be mapped for example), currently they are considered the same, i.e. a non existent value. Therefore `None` values are converted always to empty strings.

Parameters

- **`iscmd`** (*`bool`*) – Indicates if the message is a command (True) or a status (False), optional
- **`function`** (*`string`*) – internal representation of function, optional
- **`gateway`** (*`string`*) – internal representation of gateway, optional
- **`location`** (*`string`*) – internal representation of location, optional
- **`device`** (*`string`*) – internal representation of device, optional
- **`sender`** (*`string`*) – internal representation of sender, optional
- **`action`** (*`string`*) – internal representation of action, optional
- **`arguments`** (*`dictionary of strings`*) – all values should be assumed to be strings, optional

`copy`()

Creates a copy of the message.

`str`()

Stringifies a class instance.

`reply`(*response*, *reason*)

Formats the message to be sent as a reply to an existing command

This method is supposed to be used with an existing message that has been received. Using this method for all replies guarantees a consistent syntax for replies.

Parameters

- **`response`** (*`string`*) – code or abbreviation for response, e.g. ``OK```or ``ERROR``
- **`reason`** (*`string`*) – longer description of the response

class `mqttgateway.gateway.mqtt_map.`**`MsgList`**

Bases: `Queue.Queue`, `object`

Implementation of a Queue list just in case its needed.

The methods are called `push` and `pull` in order to differentiate them from the *usual* names (put, get, append, pop, . . .). TODO: implement maxsize and timeout.

`push`(*item*)

Equivalent to self._list.append(item)

`pull`()

Equivalent to self._list.pop(0)

class `mqttgateway.gateway.mqtt_map.`**`mappedTokens`**(*function*, *gateway*, *location*, *device*, *sender*, *action*, *argkey*, *argvalue*)

Bases: `tuple`

Tokens representing a message that can be mapped.

`action`
Alias for field number 5

`argkey`
Alias for field number 6

`argvalue`
Alias for field number 7

`device`
Alias for field number 3

`function`
Alias for field number 0

`gateway`
Alias for field number 1

`location`
Alias for field number 2

`sender`
Alias for field number 4

`mqttgateway.gateway.mqtt_map.`**`NO_MAP = {'action': {'maptype': 'none'}, 'argkey': {'maptyp`**
Default map, with no mapping at all.

`class` `mqttgateway.gateway.mqtt_map.`**`msgMap`**(*jsondict=None*)
Bases: `object`

Contains the mapping data and the conversion methods.

The mapping data is read from a JSON style dictionary. To access the maps use: mqtt_token = maps.*field*.i2m(internal_token) Example: mqtt_token = maps.gateway.i2m(internal_token)

Parameters **`jsondict`** (*`dictionary`*) – contains the map data in the agreed format; if None, the NO_MAP structure is used.

`class tokenMap`(*maptype*, *mapdict=None*)
Bases: `object`

Represents the mapping for a given token or characteristic.

Each instantiation of this class represent the mapping for a given token, and contains the type of mapping, the mapping dictionary if available, and the methods to convert the keywords back and forth between MQTT and internal representation.

The mapping dictionary passed as argument has the internal keywords as keys and as value a list of corresponding MQTT keywords. Only the first of the list will be used for the reverse dictionary, the other MQTT keywords being 'aliases'.

Parameters

- **`maptype`** (*`string`*) – type of map, should be either 'strict'. 'loose' or 'none'
- **`mapdict`** (*`dictionary`*) – dictionary representing the mapping

`m2i`(*mqtt_token*)
Generic method converting an MQTT token into an internal characteristic.

`i2m`(*internal_token*)
Generic method converting an internal characteristic into an MQTT token.

sender()
Getter for the `_sender` attribute.

mqtt2internal(*mqtt_msg*)
Converts the MQTT message into an internal one.

Parameters **mqtt_msg** (*mqtt.MQTTMessage*) – a MQTT message.

Returns the conversion of the MQTT message

Return type internalMsg object

Raises `ValueError` – in case of bad MQTT syntax or unrecognised map elements

internal2mqtt(*internal_msg*)
Converts an internal message into a MQTT one.

Parameters **internal_msg** (*internalMsg*) – the message to convert

Returns a full MQTT message where topic syntax is `root/function/gateway/location/device/sender/{C or S}` and payload syntax is either a plain action or a JSON string.

Return type a MQTTMessage object

Raises `ValueError` – in case a token conversion fails

`mqttgateway.gateway.mqtt_map.`**test**()
docstring

`mqttgateway.gateway.mqtt_map.`**reverse**()

mqttgateway.gateway.start_gateway module

Defines the function that starts the gateway.

`mqttgateway.gateway.start_gateway.`**startgateway**(*gateway_interface*)
Entry point.

Module contents

The package representing the *core* of the application.

There are 4 modules:

- `mqtt_client.py` defines the child class of the official MQTT Client class of the paho library;
- `mqtt_map.py` defines the classes `internalMsg` and `msgMap`;
- `start_gateway.py` which contains the script for the application initialisation and main loop.
- `configuration.py` which contains the default configuration as a string.

This package uses the logger help provided by *mqttgateway.utils.app_properties*. The `Properties` object should have been already initialised by the application using this library. However, if that is not the case, the initialisation is done here with default parameters. As a consequence, the main application should initialise `Properties` before importing anything from this package.

mqttgateway.utils package

Submodules

mqttgateway.utils.app_properties module

Application wide properties.

This module is an alternative to a singleton. It keeps some application variables in the module namespace making them effectively global. At startup, the module should be imported straight away so that it creates an AppProperties object called Properties that is global, and initialised with mostly empty values, except the `init` attribute which is essentially the constructor.

class `mqttgateway.utils.app_properties.`**`AppProperties`**(*name*, *directories*, *config_file_path*, *root_logger*, *init*, *get_path*, *get_logger*)

Bases: `tuple`

`config_file_path`
Alias for field number 2

`directories`
Alias for field number 1

`get_logger`
Alias for field number 6

`get_path`
Alias for field number 5

`init`
Alias for field number 4

`name`
Alias for field number 0

`root_logger`
Alias for field number 3

mqttgateway.utils.init_logger module

Function to initialise a logger with pre-defined handlers.

`mqttgateway.utils.init_logger.`**`initlogger`**(*logger*, *logfiledata*, *emaildata*)

Initialise the logging environment for the application. from cgi import logfile

The logger passed as parameter should be sent by the 'root' module if hierarchical logging is the objective. The logger is then initialised with the following handlers:

- the standard 'Stream' handler will always log level WARN and above;
- a rotating file handler, with fixed parameters (max 50kB, 3 rollover files); the level for this handler is DEBUG if the parameter 'log_debug' is True, INFO otherwise; the file name for this log is given by the log_filepath parameter which is used as is; an error message is logged in the standard handler if there was a problem creating the file;
- an email handler with the level set to CRITICAL;

Args: logger: the actual logger object to be initialised; logfiledata (tuple): 3 elements tuple made of

[0] = logfilepath (string): the log file path, if None, file logging is disabled; [1] = filelevel (string): the level of log to be sent to the file, or NONE; [2] = consolelevel (string): the level of log to be sent to the console (stdout), or NONE.

emaildata (tuple): 4 elements tuple; no email logging if either of first 3 values invalid [0] = host (string), [1] = port (int), [2] = address (string), ,is enabled; [3] = app_name (string).

Returns: Nothing

Raises: any IOErrors thrown by file handling methods are caught.

mqttgateway.utils.load_config module

Function to facilitate the loading of configuration parameters.

Based on ConfigParser.

`mqttgateway.utils.load_config.`**`loadconfig`**`(`*cfg_dflt_string*, *cfg_filepath*`)`

The configuration is loaded with the help of the python library ConfigParser and the following convention.

The default configuration is represented by the string passed as argument `cfg_dflt_string`. This string is expected to have all the necessary sections and options that the application will need, with their default values. All options need to be listed there, even the ones that HAVE to be updated and have no default value. This default configuration is declared as a constant string, and is also the template for the actual configuration file.

The function 'loads' this default configuration, then checks if the configuration file is available, and if found it grabs only the values from the file that are also present in the default configuration. Anything else in the file is not considered, except for the [INTERFACE] section (see below). The result is a configuration object with all the necessary fields, updated by the file values if present, or with the default values if not. The application can therefore call all fields without index errors.

The exception to the above process in the [INTERFACE] section. The options of this section will be loaded 'as is' in the Config object. This section can be used to define ad-hoc options that are not in the default configuration.

Finally, the function updates the option 'location' in the section [CONFIG] with the full path of the configuration file used, just in case it is needed later. It also 'logs' the error in the 'error' option of the same section, if any OS exception occurred while opening or reading the configuration file.

Parameters

- **`cfg_dflt_string`** (*`string`*) – represents the default configuration.
- **`cfg_filepath`** (*`string`*) – the path of the configuration file; it is used 'as is' and if it is relative there is no guarantee of where it will actually point. . . Better send an absolute path then.

Returns object loaded with the parameters.

Return type ConfigParser.RawConfigParser object

mqttgateway.utils.throttled_exception module

An exception class that throttles events in case an error is triggered too often.

exception `mqttgateway.utils.throttled_exception.`**`ThrottledException`**(*msg=None, throttlelag=10, module_name=None*)

Bases: `exceptions.Exception`

Exception class base to throttle events

This exception can be used as a base class instead of `Exception`. It adds a counter and a timer that allow to silence the error for a while if desired. Only after a given period a trigger is set to `True` to indicate that a number of errors have happened and it is time to report them.

It creates 2 members:

- `trigger` is a boolean set to True after the requested lag;
- `report` is a string giving some more information on top of the latest message.

The code using these exceptions can test the member `trigger` and decide to silence the error until it is True. At any point one can still decide to use these exceptions as normal ones and ignore the `trigger` and `report` members.

Usage:

```
try:
    some statements that might raise your own exception derived from␣
↪ThrottledException
except YourExceptionError as err:
    if err.trigger:
        log(err.report)
```

Parameters

- **msg** (*string*) – the error message, as for usual exceptions, optional
- **throttlelag** (*int*) – the lag time in seconds while errors should be throttled, defaults to 10 seconds
- **module_name** (*string*) – the calling module to give extra information, optional

Module contents

Utilities package

5.7.3 Submodules

5.7.4 Module contents

Root package for the **mqttgateway** project.

CHAPTER 6

Indices and tables

- genindex
- modindex

Python Module Index

Index