Suricate Documentation

Release 0.1

Marco Buttu

Jul 14, 2020

Contents

Preface

GUIs, RFI checkers, receiver monitors, backends, meteo services, they all need to know some antenna parameters like the current azimuth and elevation, device temperatures, configurations, LO frequencies and so forth. This documentation shows how to easily get them by means of an application called Suricate, composted by three packages: *Application Programming Interface*, *Control system monitor*, and *Alarms notification*. There is one chapter for each package and the first one, *Application Programming Interface*, explains how to use the HTTP API to send commands to the control system and to get all antenna parameters. If you are a system administrator in charge of installing Suricate, please read the *System Administrator Guide*.

CHAPTER 1

Contents

1.1 Application Programming Interface

This chapter explains how to send commands to the control system and how to get the antenna parameters by means of an HTTP API. You can build your clients in whatever programming language you want and make it running on any operating system.

1.1.1 How to get the antenna parameters

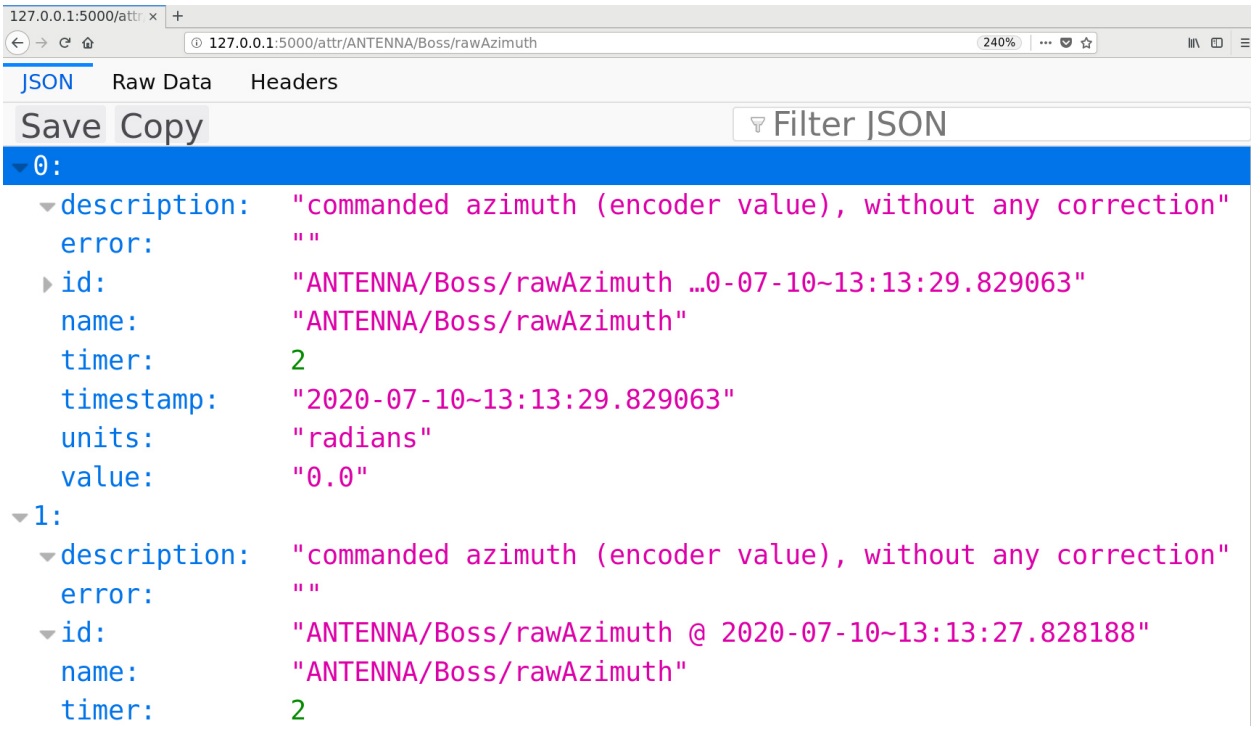
Suricate is configured by means of a configuration file. There are three templates, one for each telescope, and every template contains all parameters you can get for that telescope. Have a look at the first 20 lines of the `SRT configuration file`:

```
COMPONENTS:

  ANTENNA/Boss:
    startup_delay: 10
    container: AntennaBossContainer
    properties:
    - name: rawAzimuth
      timer: 2.0
      units: radians
      description: "commanded azimuth (encoder value), without any correction"
    - name: rawElevation
      timer: 2.0
      units: radians
      description: "commanded elevation (encoder value), without any correction"
    - name: observedAzimuth
      timer: 3.0
      units: radians
      description: "current azimuth with refraction and pointing model applied"
    - name: observedElevation
      timer: 3.0
```

Basically there is a list of *components*, and every component has some parameters (*properties* or *methods*) that you can ask for. They are identified by the lable `name`, it means, regarding to previous lines, you can get the following `ANTENNA/Boss` parameters: `rawAzimuth`, `rawElevation`, `observedAzimuth`, `observedElevation`.

For instance, you can get the `rawAzimuth` parameter by pointing a browser to `http://192.168.200.203:5000/attr/ANTENNA/Boss/rawAzimuth`.



Note: In this image you see the browser pointing to localhost. That is because the control system was running locally, over a simulator. The proper IP address, at the Sardinia Radio Telescope, is `192.168.200.203`.

In all likelihood you want to get the parameters programmatically. Let's see how to do it by using Python:

```
>>> target = 'ANTENNA/Boss/rawAzimuth'
>>> import requests
>>> r = requests.get('http://192.168.200.203:5000/attr/%s' % target)
>>> r.json()
[
  {
    'description': 'commanded azimuth (encoder value),...',
    'timestamp': '2020-07-10~13:28:57.842001',
    'timer': 2.0,
    'value': '1.3248',
    'error': '',
    'units': 'radians',
    'id': 'ANTENNA/Boss/rawAzimuth @ 2020-07-10~13:28:57.842001',
    'name': 'ANTENNA/Boss/rawAzimuth'
  }
  {
    'description': 'commanded azimuth (encoder value),...',
    'timestamp': '2020-07-10~13:28:56.428177',
    'timer': 2.0,
```

(continues on next page)

(continued from previous page)

```
    'value': '1.3492',
    'error': '',
    'units': 'radians',
    'id': 'ANTENNA/Boss/rawAzimuth @ 2020-07-10~13:28:56.428177',
    'name': 'ANTENNA/Boss/rawAzimuth'
  }
 ...
]
```

From now on in this documentation we do not write anymore the base URL `http://192.168.200.203:5000`. That is just for the sake of simplicity, you still need to write it in your code. You can use the following URLs:

- `/attr/SYSTEM/Component/name`: last ten data dictionaries for the attribute `name`. Each dictionary has eight items (`description`, `timestamp`, `timer`, `value`, and so forth). In case of failure, the `error` field describes the issue and the `value` field is not reliable.
- `/attr/SYSTEM/Component/name/N`: last `N` data dictionaries for attribute `name`.
- `/attr/SYSTEM/Component/name/from/x`: all data dictionaries for attribute `name`, starting from timestamp `x` until now. The timestamp must have the following format: `YYYY-MM-DD~HH-mm-SS.f`. For instance, here is an example: `2020-07-10~13:28:56.428177`
- `/attr/SYSTEM/Component/name/from/x/to/y`: all data dictionaries for attribute `name`, starting from timestamp `x` until timestamp `y`.

For instance, in previous example we had `ANTENNA/Boss/rawAzimuth` for `SYSTEM/Component/name`. The system was `ANTENNA`, che component was `Boss` and the attribute name was `rawAzimuth`.

The attributes information is retrieved from a persistent data base. If you want to build a client that requires real time data, the best way is to use a Redis client, as explained in chapter *Control system monitor*.

Summarizing, `http://192.168.200.203:5000` as base URL, and:

1. `/attr/SYSTEM/Component/name`: last 10
2. `/attr/SYSTEM/Component/name/N`: last N
3. `/attr/SYSTEM/Component/name/from/x`: from timestamp x
4. `/attr/SYSTEM/Component/name/from/x/to/y`: from timestamp x to y.

1.1.2 How to execute a command

To execute a command `foo`, use the following `POST` request:

```
POST /cmd/foo
```

For instance, that is the case for `getTpi` command:

```
>>> import requests
>>> r = requests.post('http://192.168.200.203:5000/cmd/getTpi')
```

The response is a json dictionary containing some information about the command:

```
>>> r.json()
{
  'delivered': True,
  'complete': False,
```

(continues on next page)

(continued from previous page)

```
  'success': False,
  'command': 'getTpi',
  'result': 'unknown',
  'stime': '2020-07-10~14:26:21.442051'
  'etime': '2020-07-10~14:26:21.442051',
  'seconds': 0.0,
  'id': 'getTpi_2020-07-10~14:26:21.442051',
}
```

The fields have the following meaning:

- `command`: the command sent to the system.
- `stime`: starting execution time.
- `etime`: ending execution time.
- `delivered`: has the command been delived to the scheduler? In case it is `False`, maybe the redis queue was not running. See *System Administrator Guide* for more details.
- `complete`: has the command terminated its execution?
- `success`: the boolean `success` returned by the scheduler. In case it is `False`, the command has not been executed properly.
- `result`: is the `answer` returned by the scheduler.
- `seconds`: seconds of execution for the command.
- `id`: the database `id` field.

The request is *no blocking*, it means that the server returns immediately a response and put the request in a queue, to be dispatched as soon as possible. It also means that the fields of the response tell you only the status of the execution at time zero. That is not enough: the following section explains how to know the status of the command.

1.1.3 How to get the command status

When you execute a command, the response gives you the command `id`:

```
>>> import requests
>>> r = requests.post('http://192.168.200.203:5000/cmd/getTpi')
>>> response = r.json()
>>> response
{
  'complete': False,
      ...
  'seconds': 0.0,
  'id': 'getTpi_2020-07-10~14:26:21.442051',
}
```

Use this `id` to ask for the status of the command. You have to perform a GET request instead of a POST one:

```
>>> id = response['id']
>>> r = requests.get('http://192.168.200.203:5000/cmd/%s' % id)
>>> r.json()
{
  'delivered': True,
  'complete': False,
```

(continues on next page)

(continued from previous page)

```
  'success': False,
  'command': 'getTpi',
  'result': 'unknown'
  'stime': '2020-07-10~14:26:21.442051'
  'etime': '2020-07-10~14:26:21.442051'
  'seconds': 0.0
  'id': 'getTpi_2020-07-10~14:26:21.442051',
}
>>> r = requests.get('http://192.168.200.203:5000/cmd/%s' % id)
>>> r.json()
{
  'delivered': True,
  'complete': True,
  'success': True,
  'command': 'getTpi',
  'result': 'getTpi\\\n00) 2508.475000\n01) 2506.97500'
  'stime': '2020-07-10~14:26:21.442051'
  'etime': '2020-07-10~14:26:21.831193',
  'seconds': 0.389142,
  'id': 'getTpi_2020-07-10~14:26:21.442051',
}
```

1.2 Control system monitor

There are two ways to get the antenna parameters: by using the HTTP REST API, as explained in chapter *Application Programming Interface*, and by using a Redis client, as discussed in this chapter. The Redis client allows you to get the parameters as quickly as possible, while the HTTP API has a lower time resolution but allows you to look at the history of the attribute, asking for the last `N` values, for all values after datetime `x` or between datetime `x` and datetime `y`.

Note: Suricate is composed by a monitor, a database filler and a server. The *monitor* collects all attributes information from the control system and stores them on a Redis database. The *database filler* reads the attributes from Redis and stores them on a persistent database. When the user performs a request by means of the HTTP REST API, the *server* executes a query to the persistent database and returns the response. The user, as explained in this chapter, can also read the attributes information by reading directly from the Redis in memory database.

If you want to get the antenna parameters from the Redis database, you need a Redis client installed on your machine.

1.2.1 Install a Redis client

In this section we will briefly see how to install one of them for Python and C, but if you do not use these languages, visit the official Redis webpage to get the right client for your operating system and programming language.

Python

The most used Python client is called redis-py. To install it by `pip`, simply:

```
$ pip install redis
```

If you do not have `pip`, then download the source files from the redis-py webpage, extract them, move to the source file directory and execute:

```
$ python setup.py install
```

C Programming Language

The official C client is available on the Hiredis GitHub page. Clone it on your machine:

```
$ git clone https://github.com/redis/hiredis.git
```

The installation steps depend of your operating system and compiler. For instance, on Linux CentOS:

```
$ cd hiredis/
$ make
$ sudo make install
```

1.2.2 Use your client to get the antenna parameters

To get the antenna parameters you have to connect your client to the Redis server. Server IP and port are `192.168.200.203` and `6379`. That is not enough, because you need to understand how your client works. These instructions should be provided by your Redis client documentation. Let's see two examples, using Python and C. Please read the *How to use the Python client* section also if you want to use the C programming language, because the Python examples show you all information (the *How to use the C client* section is just a short summary).

How to use the Python client

Let's see how to get the `rawAzimuth` from a Python shell:

```
>>> from redis import StrictRedis # Import the redis client
>>> r = StrictRedis(host='192.168.200.203', port=6379)  # Connect to server
>>> r.hgetall('ANTENNA/Boss/rawAzimuth')  # Ask for the rawAzimuth parameter
{
  'units': 'radians', 'timestamp': '2019-12-18 12:52:04.206445',
  'description': 'raw azimuth (encoder value), without any correction',
  'value': '0.602314332058', 'error': '', 'timer': '2.0'
}
```

The format is the same as you see in previous chapter: `SYSTEM/Component/name`. The result of the request contains the attribute value, its units, description, timer and timestamp. In case of error there is an error message, and the `value` is an empty string:

```
>>> r.hgetall('ANTENNA/Boss/rawAzimuth')
{
  'units': 'radians', 'timestamp': '2019-12-18 12:55:13.197819',
  'description': 'raw azimuth (encoder value), without any correction',
  'value': '', 'error': 'component ANTENNA/Boss not available',
  'timer': '2.0'
}
```

Here is how to get a particular field:

```
>>> result = r.hgetall('ANTENNA/Boss/rawAzimuth')
>>> result['units']
'radians'
>>> result['value']
'0.599527371772'
>>> result['description']
'raw azimuth (encoder value), without any correction'
```

Another way is to use `hget()`, giving the field name as a second argument:

```
>>> r.hget('ANTENNA/Boss/rawAzimuth', 'value')
'0.598923921358'
```

Note: The Python `hgetall()` and `hget()` methods execute the Redis calls HGETALL and HGET.

To know the status of the components (`available` or `unavailable`) use the key `components`:

```
>>> r.hgetall('components')
{
  'MANAGEMENT/Gavino': 'available', 'WEATHERSTATION/WeatherStation': 'available',
  'RECEIVERS/SRTKBandMFReceiver': 'available', 'RECEIVERS/SRT7GHzReceiver': 'available
↪',
  'ANTENNA/Boss': 'available', 'RECEIVERS/Boss': 'available'
}
```

Some attributes are sequences, but Suricate saves them as strings. For instance, have a look at the current `LO` value:

```
>>> lo = r.hget('RECEIVERS/Boss/LO', 'value')
>>> lo
'(5850.0, 5850.0)'
```

To get a tuple you do not need to parse the string. Just use `literal_eval` from the standard library `ast` module:

```
>>> from ast import literal_eval
>>> literal_eval(lo)
(5850.0, 5850.0)
```

How to use the C client

To understand how to get the `rawAzimuth` parameter look at the most important lines of `example.c` source file:

```
1     reply = redisCommand(c,"HGETALL ANTENNA/Boss/rawAzimuth");
2     int i;
3     printf("\nHGETALL ANTENNA/Boss/rawAzimuth\n");
4     for(i=0; i<reply->elements; i++) {
5         if(i%2 == 0) {
6             printf("* %s: ", reply->element[i]->str);
7         }
8         else {
9             printf("%s\n", reply->element[i]->str);
10        }
11    }
12    freeReplyObject(reply);
13
```

(continues on next page)

(continued from previous page)

```
    reply = redisCommand(c,"HGET ANTENNA/Boss/rawAzimuth value");
    printf("\nHGET ANTENNA/Boss/rawAzimuth value: %s\n", reply->str);
    freeReplyObject(reply);
```

If you compile `example.c` and execute the program, then you get the following output:

```
$ gcc example.c -I /usr/local/include/hiredis/ -lhiredis  # Linux CentOS
$ ./a.out

HGETALL ANTENNA/Boss/rawAzimuth
* description: raw azimuth (encoder value), without any correction
* error:
* units: radians
* timer: 2.0
* timestamp: 2019-12-18 16:19:50.042722
* value: 0.469729642021

HGET ANTENNA/Boss/rawAzimuth value: 0.469729642021
```

For more information about the Redis C client please scroll down the C client website page.

Public and subscribe

We saw how to ask for antenna parameters in a *request-response* manner. Using that pattern you have to take care of the result, because you can get the same value for different requests. For intance, if you look at the `SRT configuration file` you see that `rawAzimuth` has a sampling time of 2 seconds. That is why if you ask for the parameter to fast, for instance, once per second, you get the same result for different requests:

```
>>> import time
>>> import redis
>>> r = redis.StrictRedis(host='192.168.200.203', port=6379)
...     print(r.hgetall('ANTENNA/Boss/rawAzimuth')['timestamp'])
...     time.sleep(0.9)  # 900ms
...
2019-12-20 12:11:48.443357
2019-12-20 12:11:49.842924
2019-12-20 12:11:49.842924
2019-12-20 12:11:50.446206
2019-12-20 12:11:50.446206
```

As you can see by looking at these timestamps, the second result is the same as the third, and the forth is the same as the fifth. It is not a big issue, because you can discard the result in case its timestamp is the same as the previous one. But there is another way: the *public-subscribe* pattern. In that case you subscribe to a channel and wait for new data. Let'see how to do it by examples, using the Python client.

As a first step we create a *pubsub* object and subscribe it to the `ANTENNA/Boss/rawAzimuth` channel:

```
>>> import redis
>>> r = redis.StrictRedis(host='192.168.200.203', port=6379)
>>> pubsub = r.pubsub()
>>> pubsub.subscribe('ANTENNA/Boss/rawAzimuth')
```

Now we are ready to get the messages. The first one is a kind of header that tell us which channel we are listening from. Its `type` is `subscribe`:

```
>>> pubsub.get_message()
{
  u'pattern': None, u'type': 'subscribe',
  u'channel': 'ANTENNA/Boss/rawAzimuth', u'data': 1L
}
```

From now on the `type` of the messages is `message`, and that means they contain the antenna parameter:

```
>>> pubsub.get_message()
{
  u'pattern': None, u'type': 'message', u'channel': 'ANTENNA/Boss/rawAzimuth',
  u'data': '{
    "description": "raw azimuth (encoder value), without any correction",
    "error": "", "units": "radians", "timestamp": "2019-12-20 14:43:16.262544",
    "value": "3.97375753112", "timer": "2.0"
  }'
}
```

The `pubsub` has a method `listen()` that waits for new messages. In fact, as you can see in the following example, we do not get the same message on different responses, as appened in the *request-response* pattern:

```
... for item in pubsub.listen():
...     if item['type'] == 'message':
...         data = json.loads(item['data'])
...         timestamp = data.get('timestamp')
...         value = data.get('value')
...         print(timestamp, value)
...
(u'2019-12-20 15:04:22.148343', u'4.07524413623')
(u'2019-12-20 15:04:24.043550', u'4.07527853477')
(u'2019-12-20 15:04:26.465494', u'4.07530617372')
(u'2019-12-20 15:04:28.843325', u'4.07533925004')
(u'2019-12-20 15:04:30.956359', u'4.07536964412')
```

Note: The antenna parameter is stored as a json string in the `data` field of the item. I used `json.loads()` in order to convert the json string to a Python dictionary.

You can subscribe to more than one channel at the same time:

```
... pubsub = r.pubsub()
... pubsub.subscribe(
...       'ANTENNA/Boss/rawAzimuth',
...       'ANTENNA/Boss/rawElevation'
... )
... for item in pubsub.listen():
...     if item['type'] == 'message':
...         channel = item['channel']
...         data = json.loads(item['data'])
...         value = data.get('value')
...         print(channel, value)
...
('ANTENNA/Boss/rawAzimuth', u'0.758804232371')
('ANTENNA/Boss/rawElevation', u'0.659474554184')
('ANTENNA/Boss/rawAzimuth', u'0.758810651617')
('ANTENNA/Boss/rawElevation', u'0.659490653912')
```

(continues on next page)

(continued from previous page)

```
('ANTENNA/Boss/rawElevation', u'0.659506753743')
                        ...
```

You can also use the *glob* syntax. It means you can use a * to listen from more than one channel. For insance, in the following case we are listening to all channels starting with `ANTENNA/Boss`:

```
... pubsub = r.pubsub()
... pubsub.psubscribe('ANTENNA/Boss/*')
... for item in pubsub.listen():
...     if item['type'] == 'pmessage':
...         channel = item['channel']
...         data = json.loads(item['data'])
...         value = data.get('value')
...         print(channel, value)
...
('ANTENNA/Boss/observedDeclination', u'0.952583014116')
('ANTENNA/Boss/observedAzimuth', u'0.800302875968')
('ANTENNA/Boss/rawElevation', u'0.668070294866')
('ANTENNA/Boss/observedElevation', u'0.666206037338')
('ANTENNA/Boss/rawAzimuth', u'0.762179742186')
('ANTENNA/Boss/observedRightAscension', u'0.929348150783')
('ANTENNA/Boss/observedGalLongitude', u'2.53064537516')
('ANTENNA/Boss/observedGalLatitude', u'-0.0212982297497')
('ANTENNA/Boss/status', u'MNG_OK')
('ANTENNA/Boss/observedAzimuth', u'0.800308445826')
                         ...
```

Note: In the last example (glob syntax) we subcribed to the channels using `pubsub.psubscribe()` and not `pubsub.subscribe()`, and we waited for the type `pmessage`, not `message`.

1.3 Alarms notification

Todo: Write a package for alarms management. It has to push the active alarms to the clients by using SSE.

1.4 System Administrator Guide

Preface

A system administrator in charge of installing, configuring, starting and stopping Suricate should read this chapter.

1.4.1 Installation

Suricate requires Redis. Additional requirements are listed in the Suricate's *setup.py* file: we do not care about them because they will be automatically installed during the Suricate installation.

Note: Suricate works with Python 2.7, the same version of the latest ACS framework. Python 2.7 is not supported anymore and maybe at some point we will not be able to find the Python 2.7 Suricate dependencies online. That is why we put all of them on a private DISCOS GitHub repository.

Redis

In this section we will see how to install and configure Redis on Linux CentOS. If that is not your case, get the source code from the Redis website and follow the documentation.

On Linux CentOS:

```
sudo yum install redis
```

To bind Redis server to all ports, open */etc/redis.conf* and change the line `bind 127.0.0.1` to `bind 0.0.0.0`. Change also `protected-mode` from `yes` to `no`. At this point:

```
$ sudo chkconfig redis on
$ sudo service redis restart
```

Start the Redis queue. It is used to send the commands to DISCOS:

```
$ rqworker -P ~/suricate/suricate discos-api
```

Remote configuration

In case Suricate has not been installed on the machine running the manager, you need to export the manager reference. On the Suricate machine, open */discos-sw/config/misc/bash_profile* and write:

```
MNG_IP=192.168.200.203
export MANAGER_REFERENCE=corbaloc::$MNG_IP:3000/Manager
```

Upload your public SSH key to the manager host:

```
$ ssh-keygen -t dsa
$ scp .ssh/id_dsa.pub discos@discos-manager:~
```

Go to the manager host and add your public SSH key:

```
$ ssh discos@discos-manager
$ cat id_dsa.pub >> .ssh/authorized_keys
$ rm id_dsa.pub
$ logout
```

Now login to the manager host via SSH and answer `yes`:

```
$ ssh discos@discos-manager
          ...
Are you sure you want to continue connecting (yes/no)?
```

Note: In the configuration file need to set the `RUN_ON_MANAGER_HOST: False`. Next section explains how to create a configuration file.

You are now ready to install and use Suricate.

Install Suricate

To install Suricate clone the repository and use `pip`:

```
$ sudo ln -s /alma/ACS-FEB2017/Python/bin/python /bin/python
$ sudo ln -s /alma/ACS-FEB2017/Python/bin/pip /bin/pip
$ git clone https://github.com/marco-buttu/suricate.git
$ cd suricate
$ sudo -u discos pip install .
$ sudo cp startup/suricate-server /etc/rc.d/init.d/
$ sudo chkconfig --add suricate-server
$ sudo chkconfig suricate-server on
```

At this point Suricate is a startup service. Before starting we need to configure it. To install the SRT configuration:

```
$ suricate-config -t srt
```

This command copies the SRT configuration to *~/.suricate/config/config.yaml*. If you want to add or change some antenna parameters, change that file.

Create the database

Todo: All these steps must be deployed automatically. To be done.

Create the database tables:

```
$ cd suricate/suricate
$ source .flaskenv
$ flask db init
```

Every time a table changes:

```
$ flask db migrate -m "Task table"
$ flask db upgrade
```

Run Suricate

You are ready to start Suricate:

```
$ sudo service suricate-server start
```

To know its status and stop it:

```
$ sudo service suricate-server status
suricate-server is running
$ sudo service suricate-server stop
$ sudo service suricate-server status
suricate-server is NOT running
```

To uninstall Suricate:

```
$ sudo pip uninstall suricate
```

1.4.2 Logging

There are three log files you have to take care of:

- *~/.suricate/logs/suricate.log*: user log file, with main information
- *~/.suricate/logs/apscheduler.log*: apscheduler debug file
- */tmp/suricate_service_dbg.log*: service log file