pyEOS Documentation

Release 1

David Barroso

January 22, 2015

Contents

CHAPTER 1

Tutorials

1.1 First Steps

1.1.1 Connecting to a device

First you have to connect to a device:

```
>>> from pyEOS import EOS
>>> device = EOS(hostname='10.48.71.3', username='admin', password='pa55w0rd', use_ssl=False)
>>> device.open()
```

1.1.2 Running show commands

Once you are connected you can execute any show command you want. All show commands are supported by default and will return the same output as the eAPI without modifying anything. For example:

```
>>> lldp = device.show_lldp_neighbors()
>>> print lldp
{u'tablesDrops': 0, u'_meta': {u'execStartTime': 1418311629.19, u'execDuration': 0.0096}, u'tablesAge
```

Let's explain what you are seeing there. You got a dictionary with different keys, the interesting ones are **_meta** and **lldpNeighbors**. The key *_meta* gives you some useful information to find out if your query is hogging the device and the key **lldpNeighbors** is a list of dictionaries giving you information about your LLDP neighbors. We can easily walk through them:

```
>>> for neighbor in lldp['lldpNeighbors']:
...       print "%s is connected on port %s. Remote port is %s" % (neighbor['neighborDevice'], neighbo
...
lom-bjg61-r1-1 is connected on port Management1. Remote port is 6
```

As I explained before, any show command is supported out of the box (with all of its suboptions). Let's try to check the status of the interfaces with **show interfaces description**:

```
>>> interfaces = device.show_interfaces_description()
>>> for interface, status in interfaces['interfaceDescriptions'].iteritems():
...     print "%s is %s" % (interface, status['interfaceStatus'])
...
Ethernet8 is down
Ethernet9 is down
Ethernet2 is down
Ethernet3 is down
```

```
Ethernet1 is down
Ethernet6 is down
Ethernet7 is down
Ethernet4 is down
Ethernet5 is down
Ethernet52/1 is down
Ethernet52/3 is down
Ethernet52/2 is down
Ethernet52/4 is down
Ethernet34 is down
Ethernet22 is down
Ethernet50/4 is down
Ethernet50/3 is down
Ethernet50/2 is down
Ethernet50/1 is down
Ethernet51/4 is down
Ethernet51/2 is down
Ethernet51/3 is down
Ethernet51/1 is down
Ethernet38 is down
Ethernet39 is down
Ethernet18 is down
Ethernet19 is down
Ethernet32 is down
Ethernet15 is down
Ethernet16 is down
Ethernet31 is down
Ethernet49/1 is down
Ethernet37 is down
Ethernet49/3 is down
Ethernet35 is down
Ethernet10 is down
Ethernet14 is down
Ethernet49/2 is down
Ethernet33 is down
Ethernet49/4 is down
Ethernet30 is down
Management1 is up
Ethernet17 is down
Ethernet48 is down
Ethernet47 is down
Ethernet36 is down
Ethernet45 is down
Ethernet44 is down
Ethernet43 is down
Ethernet42 is down
Ethernet41 is down
Ethernet40 is down
Ethernet29 is down
Ethernet28 is down
Ethernet11 is down
Ethernet12 is down
Ethernet46 is down
Ethernet21 is down
Ethernet20 is down
Ethernet23 is down
Ethernet13 is down
Ethernet25 is down
```

```
Ethernet24 is down
Ethernet27 is down
Ethernet26 is down
```

Now let's try with the command **show ip route**:

```
>>> routes = device.show_ip_route()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pyEOS/eos.py", line 19, in wrapper
    return self.run_commands(cmd, **kwargs)[1]
  File "pyEOS/eos.py", line 63, in run_commands
    raise exceptions.CommandUnconverted(error)
pyEOS.exceptions.CommandUnconverted: CLI command 2 of 2 'show ip route' failed: unconverted command
```

Something happened here. That command has not been converted to JSON yet so the command failed as the API will always to fetch JSON by default. You can tell the API to detect this problem automatically for you and fix it:

```
>>> routes = device.show_ip_route(auto_format=True)
>>> print routes['output']
Codes: C - connected, S - static, K - kernel,
       O - OSPF, IA - OSPF inter area, E1 - OSPF external type 1,
       E2 - OSPF external type 2, N1 - OSPF NSSA external type 1,
       N2 - OSPF NSSA external type2, B I - iBGP, B E - eBGP,
       R - RIP, I - ISIS, A B - BGP Aggregate, A O - OSPF Summary,
       NG - Nexthop Group Static Route

Gateway of last resort:
 S      0.0.0.0/0 [1/0] via 10.48.68.1, Management1

 C      10.48.68.0/22 is directly connected, Management1
```

Or you can also explictly ask for text output:

```
>>> lldp = device.show_lldp_neighbors(format='text')
>>> print lldp['output']
Last table change time   : 105 days, 1:02:36 ago
Number of table inserts  : 1
Number of table deletes  : 0
Number of table drops    : 0
Number of table age-outs : 0

Port       Neighbor Device ID            Neighbor Port ID          TTL
Ma1        lom-bjg61-r1-1                6                         120
```

1.1.3 Running arbitrary commands

You can also run a list of commands. They can be any command you want:

```
>>> cmds = ['ping 8.8.8.8', 'traceroute 8.8.8.8']
>>> output = device.run_commands(cmds)
>>> print output[1]['messages']
[u'PING 8.8.8.8 (8.8.8.8) 72(100) bytes of data.\n80 bytes from 8.8.8.8: icmp_req=1 ttl=60 time=1.31
>>> print output[1]['messages'][0]
PING 8.8.8.8 (8.8.8.8) 72(100) bytes of data.
80 bytes from 8.8.8.8: icmp_req=1 ttl=60 time=1.31 ms
80 bytes from 8.8.8.8: icmp_req=2 ttl=60 time=1.08 ms
80 bytes from 8.8.8.8: icmp_req=3 ttl=60 time=0.783 ms
```

```
80 bytes from 8.8.8.8: icmp_req=4 ttl=60 time=0.722 ms
80 bytes from 8.8.8.8: icmp_req=5 ttl=60 time=0.724 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4ms
rtt min/avg/max/mdev = 0.722/0.925/1.313/0.237 ms, ipg/ewma 1.163/1.105 ms

>>> print output[2]['messages'][0]
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
 1  10.48.68.1 (10.48.68.1)  0.377 ms  0.351 ms  0.334 ms
 2  80.239.169.193 (80.239.169.193)  1.206 ms  1.181 ms  1.167 ms
 3  62.115.45.14 (62.115.45.14)  48.143 ms  48.117 ms  48.042 ms
 4  72.14.239.241 (72.14.239.241)  1.596 ms 64.233.175.10 (64.233.175.10)  1.303 ms 72.14.239.239 (72
 5  8.8.8.8 (8.8.8.8)  1.551 ms  1.538 ms  0.954 ms
```

On the previous example *output[0]* will contain the result of executing the command **enable**, *output[1]* the result of the **ping** command and finally *output[2]*, the result of the **traceroute**. *Note: If you don't specify the command enable as the first command on the list it is added automatically by this API.*

Here is another example with only one command:

```
>>> output = device.run_commands(['dir /all'])
>>> print output[1]['messages'][0]
Directory of flash:/

       -r-x   358580934           Aug 12 12:23  .boot-image.swi
       drwx        4096            Feb 5 02:31  .extensions
       -rwx   306459060            Feb 5 02:29  EOS-4.11.6.swi
       -rwx   358580934           Aug 12 12:15  EOS-4.14.1F.swi
       -rwx          27           Aug 12 12:21  boot-config
       drwx        4096           Aug 12 12:25  debug
       drwx        4096           Dec 11 13:39  persist
       drwx        4096            Feb 5 02:36  schedule
       -rwx        2199           Dec 11 13:39  startup-config
       -rwx           0            Aug 4 12:31  zerotouch-config

1691504640 bytes total (664113152 bytes free)
```

It can be any command supported on the CLI. You could even reload the device, install extensions or upgrade firmware:

```
>>> output = device.run_commands(['reload now'])
>>> print output[1]
{u'_meta': {u'execStartTime': 1418314151.77, u'execDuration': 1.2613}}
```

Reloading the device did not give that much feedback but this is what I got on an SSH session I had open:

```
arista-7150s-64-2#
Broadcast message from root@arista-7150s-64-2
    (unknown) at 16:09 ...

The system is going down for reboot NOW!
Connection to 10.48.71.3 closed by remote host.
Connection to 10.48.71.3 closed.
```

1.1.4 Managing Configuration

You can easily get the configuration and print it:

```
>>> device.load_running_config()
>>> print device.running_config.to_string()
transceiver qsfp default-mode 4x10G
queue-monitor length update-interval 5000000
hostname eapi-lab
spanning-tree mode mstp
aaa authorization exec default local
no aaa root
username admin privilege 15 role network-admin secret 5 $1$7uXjRZfX$pOFGCCKivNwqIDYFIYbze0
vrf definition mgmtVRF
  rd 65000:65000
interface Ethernet1
interface Ethernet2
   description "whatever"
   shutdown
   no switchport
   ip address 10.0.0.1/31
interface Ethernet3
  description "whatever"
interface Ethernet4
interface Ethernet5
interface Ethernet6
interface Ethernet7
interface Ethernet8
interface Ethernet9
interface Ethernet10
interface Ethernet11
interface Ethernet12
interface Ethernet13
interface Ethernet14
interface Ethernet15
interface Ethernet16
interface Ethernet17
interface Ethernet18
interface Ethernet19
interface Ethernet20
interface Ethernet21
interface Ethernet22
interface Ethernet23
interface Ethernet24
interface Ethernet25
interface Ethernet26
interface Ethernet27
interface Ethernet28
interface Ethernet29
interface Ethernet30
interface Ethernet31
interface Ethernet32
interface Ethernet33
interface Ethernet34
interface Ethernet35
interface Ethernet36
interface Ethernet37
interface Ethernet38
interface Ethernet39
interface Ethernet40
interface Ethernet41
interface Ethernet42
```

```
interface Ethernet43
interface Ethernet44
interface Ethernet45
interface Ethernet46
interface Ethernet47
interface Ethernet48
interface Ethernet49/1
interface Ethernet49/2
interface Ethernet49/3
interface Ethernet49/4
interface Ethernet50/1
interface Ethernet50/2
interface Ethernet50/3
interface Ethernet50/4
interface Ethernet51/1
interface Ethernet51/2
interface Ethernet51/3
interface Ethernet51/4
interface Ethernet52/1
interface Ethernet52/2
interface Ethernet52/3
interface Ethernet52/4
interface Management1
  ip address 10.48.71.3/22
ip route 0.0.0.0/0 10.48.68.1
ip route vrf mgmtVRF 0.0.0.0/0 10.48.68.1
no ip routing
no ip routing vrf mgmtVRF
management api http-commands
  no protocol https
  protocol http
  no shutdown
end
```

Or just check an interface configuration:

```
>>> print device.running_config['interface Ethernet2']
[u'description "whatever"', u'shutdown', u'no switchport', u'ip address 10.0.0.1/31']
```

You can also read configuration from a file, compare the running config with the candidate config:

```
>>> device.load_candidate_config('tests/config.txt')
>>> print device.compare_config()
+ hostname NEWHOSTNAME
- hostname eapi-lab
interface Ethernet1
  + description "whatever"
interface Ethernet2
  - shutdown
```

You can commit the configuration if you are happy:

```
>>> device.commit()
[{u'_meta': {u'execStartTime': 1418660581.91, u'execDuration': 0.00144815444946}}, {u'messages': [u"e
>>> print device.compare_config()

>>>
```

And even rollback if you regret it:

```
>>> device.rollback()
[{u'_meta': {u'execStartTime': 1418660622.75, u'execDuration': 0.00146913528442}}, {u'messages': [u"e
>>> print device.compare_config()
+ hostname NEWHOSTNAME
- hostname eapi-lab
interface Ethernet1
  + description "whatever"
interface Ethernet2
  - shutdown
```

1.1.5 Facts

The API also supports gathering some facts:

```
 device.get_facts()
>>> print facts['serial_number']
JPE14023449
>>> print facts['system_mac_address']
00:1c:73:42:86:b7
>>> print facts['uptime']
1418646951.73
>>> print facts['model_name']
DCS-7150S-64-CL-F
>>> print facts['version']
4.14.5F
```

Facts also include interface details:

```
>>> print facts['interfaces'].keys()
[u'Ethernet8', u'Ethernet9', u'Ethernet2', u'Ethernet3', u'Ethernet1', u'Ethernet6', u'Ethernet7', u'
>>> print facts['interfaces']['Ethernet2']
{u'interfaceStatistics': {u'inBitsRate': 0.0, u'updateInterval': 300.0, u'outBitsRate': 0.0, u'outPkt
>>> print facts['interfaces']['Ethernet2']['description']
"whatever"
>>> print facts['interfaces']['Ethernet2']['forwardingModel']
routed
>>> print facts['interfaces']['Ethernet2']['interfaceAddress'][0]['primaryIp']['address']
10.0.0.1
>>> print facts['interfaces']['Ethernet2']['interfaceAddress'][0]['primaryIp']['maskLen']
31
```

CHAPTER 2

Classes

2.1 EOS

class pyEOS.eos.**EOS**(*hostname*, *username*, *password*, *use_ssl=True*)

Represents a device running EOS.

The object will contain the following interesting attributes:

- **running_config** - The configuration retrieved from the device using the method load_running_config
- **candidate_config** - The configuration we desire for the device. Can be populated using the method load_candidate_config

Parameters

- **hostname** – IP or FQDN of the device you want to connect to
- **username** – Username
- **password** – Password
- **use_ssl** – If set you True we will connect to the eAPI using https, otherwise http will be used

close()

Dummy, method. Today it does not do anything but it would be interesting to use it to fake closing a connection.

compare_config()

Returns A string showing the difference between the running_config and the candidate_config. The running_config is loaded automatically just before doing the comparison so there is no neeed for you to do it.

get_config(*format='json'*)

Parameters **format** – Either 'json' or 'text'

Returns The running configuration of the device.

load_candidate_config(*filename=None*, *config=None*)

Populates the attribute candidate_config with the desired configuration. You can populate it from a file or from a string. If you send both a filename and a string containing the configuration, the file takes precedence.

Parameters

- **filename** – Path to the file containing the desired configuration. By default is None.

- **config** – String containing the desired configuration.

load_running_config()
Populates the attribute running_config with the running configuration of the device.

open()
Opens the connection with the device.

replace_config(*config=None*)
Applies the configuration changes on the device. You can either commit the changes on the candidate_config attribute or you can send the desired configuration as a string. Note that the current configuration of the device is replaced with the new configuration.

Parameters **config** – String containing the desired configuration. If set to None the candidate_config will be used

rollback()
If used after a commit, the configuration will be reverted to the previous state.

run_commands(*commands*, *version=1*, *auto_format=False*, *format='json'*, *timestamps=True*)
This method will run as many commands as you want. The 'enable' command will be prepended automatically so you don't have to worry about that.

Parameters

- **commands** – List of commands you want to run
- **version** – Version of the eAPI you want to connect to. By default is 1.
- **auto_format** – If set to True API calls not supporting returning JSON messages will be converted automatically to text. By default is False.
- **format** – Format you want to get; 'json' or 'text'. By default is json. This will trigger a CommandUnconverted exception if set to 'json' and auto_format is set to False. It will return text if set to 'json' but auto_format is set to True.
- **timestamps** – This will return some useful information like when was the command executed and how long it took.

2.2 EOSConf

class pyEOS.config.**EOSConf**(*name*)
You will probably not have to bother that much about this module yourself as it is usually easier to parse the configuration from a file and then use the "load_config" methods on the EOS class to get this object populated. However, if you understand how the eAPI handles the configuration in JSON mode you should be able to manipulate it in the same way.

Parameters **name** – Name of the configuration

compare_config(*other*)
This method will compare the self object with the other object. The other object will be the target of the comparison.

Parameters **other** – Configuration object you want to do the comparison with.

Returns A string representation of the changes between the self object and other.

load_config(*filename=None*, *config=None*)
Reads the configuration from a file or from a string and loads the object. If you send both a filename and a string containing the configuration, the file takes precedence.

Parameters

- **filename** – Path to the file containing the desired configuration. By default is None.
- **config** – String containing the desired configuration.

`to_string()`

Returns A string representation of the configuration.

C

E

G

L

O

R

T