# NAPALM Documentation

***Release 3***

**David Barroso**

**May 23, 2023**

# Contents


**1 Supported Network Operating Systems: 3**
- 1.1 extras . . . 3

**2 Selecting the right driver 5**

**3 Documentation 7**
- 3.1 Installation . . . 7
- 3.2 Tutorials . . . 9
- 3.3 Validating deployments . . . 25
- 3.4 Supported Devices . . . 30
- 3.5 Command Line Tool . . . 38
- 3.6 NetworkDriver . . . 42
- 3.7 YANG . . . 69
- 3.8 napalm-logs . . . 69
- 3.9 Integrations . . . 70
- 3.10 Contributing . . . 77
- 3.11 Development . . . 80
- 3.12 Hackathons . . . 85

**Index 89**

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is a Python library that implements a set of functions to interact with different network device Operating Systems using a unified API.

NAPALM supports several methods to connect to the devices, to manipulate configurations or to retrieve data.

CHAPTER 1

# Supported Network Operating Systems:

- Arista EOS
- Cisco IOS
- Cisco IOS-XR
- Cisco NX-OS
- Juniper JunOS

## 1.1 extras

In addition to the core drivers napalm also supports community driven drivers. You can find more information about them here: *Community Drivers*

CHAPTER 2

# Selecting the right driver

You can select the driver you need by doing the following:

```
>>> from napalm import get_network_driver
>>> get_network_driver('eos')
<class napalm.eos.eos.EOSDriver at 0x10ebad6d0>
>>> get_network_driver('iosxr_netconf')
<class napalm.iosxr_netconf.iosxr_netconf.IOSXRNETCONFDriver at 0x10ad170f0>
>>> get_network_driver('iosxr')
<class napalm.iosxr.iosxr.IOSXRDriver at 0x10ec90050>
>>> get_network_driver('junos')
<class napalm.junos.junos.JunOSDriver at 0x10f8f61f0>
>>> get_network_driver('nxos')
<class napalm.nxos.nxos.NXOSDriver at 0x10f9304c8>
>>> get_network_driver('ios')
<class napalm.ios.ios.IOSDriver at 0x10f9b0738>
```

CHAPTER 3

# Documentation

## 3.1 Installation

### 3.1.1 Full installation

You can install napalm with pip:

```
pip install napalm
```

That will install all the core drivers currently available.

**Note:** Beginning with release 4.0.0 and later, NAPALM offers support for Python 3.7+ only.

**Note:** Beginning with release 3.0.0 and later, NAPALM offers support for Python 3.6+ only.

### 3.1.2 OS Package Managers

Some execution environments offer napalm through a system-level package manager. Installing with pip outside of a user profile or virtualenv/venv is inadvisable in these cases.

#### FreeBSD

```
pkg install net-mgmt/py-napalm
```

This will install napalm and all drivers and dependencies for the default version(s) of python. To install for a specific version, python X.Y, if supported:

```
pkg install pyXY-napalm
```

### 3.1.3 Dependencies

Although dependencies for the transport libraries are solved by `pip`, on some operating systems there are some particular requirements:

#### napalm-ios dependencies

#### Ubuntu and Debian

```
sudo apt-get install -y --force-yes libssl-dev libffi-dev python-dev python-cffi
```

#### RedHat and CentOS

```
sudo yum install -y python-pip gcc openssl openssl-devel libffi-devel python-devel
```

#### napalm-iosxr dependencies

#### Ubuntu and Debian

```
sudo apt-get install -y --force-yes libssl-dev libffi-dev python-dev python-cffi
```

#### RedHat and CentOS

```
sudo yum install -y python-pip gcc openssl openssl-devel libffi-devel python-devel
```

#### napalm-junos dependencies

#### Ubuntu and Debian

```
sudo apt-get install -y --force-yes libxslt1-dev libssl-dev libffi-dev python-dev␣
↪python-cffi
```

#### RedHat and CentOS

```
sudo yum install -y python-pip python-devel libxml2-devel libxslt-devel gcc openssl␣
↪openssl-devel libffi-devel
```

## 3.2 Tutorials

### 3.2.1 Outline

This tutorial gets you up-and-running quickly with NAPALM in a local virtual environment so you can see it in action in under an hour. We'll cover the following:

1. Installing the required tools
2. Creating a virtual lab with an Arista device
3. Manually applying configuration to the device using NAPALM
4. Driving NAPALM through Python code

---

**Note:** This tutorial does not cover fully automated configuration management (e.g., using NAPALM in conjunction with Ansible, Chef, Salt, etc.). We hope that tutorials for these tools will be contributed soon so that you can evaluate the options for your particular environment.

---

### 3.2.2 Installation

#### Tools

You'll need a few tools:

- Python
- pip: The PyPA recommended tool for installing Python packages
- VirtualBox: a software virtualization tool
- Vagrant: a command line utility for managing the lifecycle of virtual machines

As the focus of this tutorial is NAPALM, we don't even scratch the surface of these tools. If you're not familiar with them, please do some research[1] as they will be an important part of your development/ops toolkit.

#### Install

Install NAPALM with pip:

```
pip install napalm
```

### 3.2.3 Setting up the lab

We'll set up a lab using VirtualBox and Vagrant, with a virtual Arista device, and get some sample files for the following steps.

#### Working directory

Create a directory for your files anywhere on your machine.

[1] Vagrant's getting started guide is worth reading and working through.

### Arista vEOS

The Arista EOS image can be downloaded for free from the Arista site.

Create an account at https://www.arista.com/en/user-registration, and go to https://www.arista.com/en/support/software-download.

Download the latest "vEOS-lab-<version>-virtualbox.box" listed in the vEOS folder at the bottom of the page.

Add it to your vagrant box list, changing the <*version*>:

```
$ vagrant box add --name vEOS-lab-<version>-virtualbox ~/Downloads/vEOS-lab-<version>-
↪virtualbox.box
$ vagrant box list
vEOS-lab-quickstart (virtualbox, 0)
```

You can delete the downloaded .box file once you have added it, as `vagrant box add` copies downloaded file to a designated directory (e.g., for Mac OS X and Linux: `~/.vagrant.d/boxes`, Windows: `C:/Users/USERNAME/.vagrant.d/boxes`).

### Starting Vagrant

Create a file named `Vagrantfile` (no file extension) in your working directory with the following content (replace VEOS_BOX by your downloaded EOS version):

```
# Vagrantfile for the quickstart tutorial

# Script configuration:
#
# Arista vEOS box.
# Please change this to match your installed version
# (use `vagrant box list` to see what you have installed).
VEOS_BOX = "vEOS-lab-4.15.5M-virtualbox"

Vagrant.configure(2) do |config|

  config.vm.define "base" do |base|
    # This box will be downloaded and added automatically if you don't
    # have it already.
    base.vm.box = "hashicorp/precise64"
    base.vm.network :forwarded_port, guest: 22, host: 12200, id: 'ssh'
    base.vm.network "private_network", virtualbox__intnet: "link_1", ip: "10.0.1.100"
    base.vm.network "private_network", virtualbox__intnet: "link_2", ip: "10.0.2.100"
    base.vm.provision "shell", inline: "apt-get update; apt-get install lldpd -y"
  end

  config.vm.define "eos" do |eos|
    eos.vm.box = VEOS_BOX
    eos.vm.network :forwarded_port, guest: 22, host: 12201, id: 'ssh'
    eos.vm.network :forwarded_port, guest: 443, host: 12443, id: 'https'
    eos.vm.network "private_network", virtualbox__intnet: "link_1", ip: "169.254.1.11
↪", auto_config: false
    eos.vm.network "private_network", virtualbox__intnet: "link_2", ip: "169.254.1.11
↪", auto_config: false
  end

end
```

The above content is also available on GitHub.

This Vagrantfile creates a base box and a vEOS box when you call `vagrant up`:

```
$ vagrant up --provider virtualbox
... [output omitted] ...

$ vagrant status
Current machine states:
base                      running (virtualbox)
eos                       running (virtualbox)
```

You may see some errors when the eos box is getting created[1].

#### Troubleshooting

- After running `vagrant up`, ensure that you can ssh to the box with `vagrant ssh eos`.
- If you receive the warning "eos: Warning: Remote connection disconnect. Retrying...", see this StackOverflow post.

#### Sample files

There are some sample Arista vEOS configuration files on GitHub. You can download them to your machine by copying them from GitHub, or using the commands below:

```
$ for f in new_good.conf merge_good.conf merge_typo.conf; do
$   wget https://raw.githubusercontent.com/napalm-automation/napalm/master/docs/
↪tutorials/sample_configs/$f
$ done
```

(Note: please open a GitHub issue if these URLs are invalid.)

### 3.2.4 Programming samples

NAPALM tries to provide a common interface and mechanisms to push configuration and retrieve state data from network devices. This method is very useful in combination with tools like Ansible, which in turn allows you to manage a set of devices independent of their network OS.

---

**Note:** These samples assume you have set up your virtual lab (see *Setting up the lab*), and that the 'eos' box is accessible via point 12443 on your machine. You should also have the sample configuration files saved locally.

---

Now that you have installed NAPALM (see *Installation*) and set up your virtual lab, you can try running some sample scripts to demonstrate NAPALM in action. You can run each of the scripts below by either pulling the files from the GitHub repository, or you can copy the content to a local script (e.g., `sample_napalm_script.py`) and run it.

For people new to Python:

- the script name should not conflict with any existing module or package. For example, don't call the script `napalm.py`.

[1] Currently, `vagrant up` with the eos box prints some warnings: "No guest additions were detected on the base box for this VM! Guest additions are required for forwarded ports, shared folders, host only networking, and more. If SSH fails on this machine, please install the guest additions and repackage the box to continue. This is not an error message; everything may continue to work properly, in which case you may ignore this message." This is not a reassuring message, but everything still seems to work correctly.

- run a Python script with `$ python your_script_name.py`.

### Load/Replace configuration

Create a file called `load_replace.py` in a folder with the following content:

```
# Sample script to demonstrate loading a config for a device.
#
# Note: this script is as simple as possible: it assumes that you have
# followed the lab setup in the quickstart tutorial, and so hardcodes
# the device IP and password.  You should also have the
# 'new_good.conf' configuration saved to disk.
from __future__ import print_function

import napalm
import sys
import os


def main(config_file):
    """Load a config for the device."""

    if not (os.path.exists(config_file) and os.path.isfile(config_file)):
        msg = "Missing or invalid config file {0}".format(config_file)
        raise ValueError(msg)

    print("Loading config file {0}.".format(config_file))

    # Use the appropriate network driver to connect to the device:
    driver = napalm.get_network_driver("eos")

    # Connect:
    device = driver(
        hostname="127.0.0.1",
        username="vagrant",
        password="vagrant",
        optional_args={"port": 12443},
    )

    print("Opening ...")
    device.open()

    print("Loading replacement candidate ...")
    device.load_replace_candidate(filename=config_file)

    # Note that the changes have not been applied yet. Before applying
    # the configuration you can check the changes:
    print("\nDiff:")
    print(device.compare_config())

    # You can commit or discard the candidate changes.
    try:
        choice = raw_input("\nWould you like to commit these changes? [yN]: ")
    except NameError:
        choice = input("\nWould you like to commit these changes? [yN]: ")
    if choice == "y":
        print("Committing ...")
```

(continues on next page)

(continued from previous page)

```
        device.commit_config()
    else:
        print("Discarding ...")
        device.discard_config()

    # close the session with the device.
    device.close()
    print("Done.")


if __name__ == "__main__":
    if len(sys.argv) < 2:
        print('Please supply the full path to "new_good.conf"')
        sys.exit(1)
    config_file = sys.argv[1]
    main(config_file)
```

Run the script, passing the path to the `new_good.conf` file as an argument:

```
python load_replace.py ../sample_configs/new_good.conf
```

### 3.2.5 Changing the Configuration

NAPALM tries to provide a common interface and mechanisms to push configuration and retrieve state data from network devices. This method is very useful in combination with tools like Ansible, which in turn allows you to manage a set of devices independent of their network OS.

#### Connecting to the Device

Use the appropriate network driver to connect to the device:

```
>>> from napalm import get_network_driver
>>> driver = get_network_driver('eos')
>>> device = driver('192.168.76.10', 'dbarroso', 'this_is_not_a_secure_password')
>>> device.open()
```

Configurations can be replaced entirely or merged into the existing device config. You can load configuration either from a string or from a file.

#### Replacing the Configuration

To replace the configuration do the following:

```
>>> device.load_replace_candidate(filename='test/unit/eos/new_good.conf')
```

Note that the changes have not been applied yet. Before applying the configuration you can check the changes:

```
>>> print(device.compare_config())
+ hostname pyeos-unittest-changed
- hostname pyeos-unittest
router bgp 65000
   vrf test
```

(continues on next page)

(continued from previous page)

```
     + neighbor 1.1.1.2 maximum-routes 12000
     + neighbor 1.1.1.2 remote-as 1
     - neighbor 1.1.1.1 remote-as 1
     - neighbor 1.1.1.1 maximum-routes 12000
   vrf test2
     + neighbor 2.2.2.3 remote-as 2
     + neighbor 2.2.2.3 maximum-routes 12000
     - neighbor 2.2.2.2 remote-as 2
     - neighbor 2.2.2.2 maximum-routes 12000
interface Ethernet2
+ description ble
- description bla
```

If you are happy with the changes you can commit them:

```
>>> device.commit_config()
```

On the contrary, if you don't want the changes you can discard them:

```
>>> device.discard_config()
```

### Merging Configuration

Merging configuration is similar, but you need to load the configuration with the merge method:

```
>>> device.load_merge_candidate(config='hostname test\ninterface␣
↪Ethernet2\ndescription bla')
>>> print(device.compare_config())
configure
hostname test
interface Ethernet2
description bla
end
```

If you are happy with the changes you can commit them:

```
>>> device.commit_config()
```

On the contrary, if you don't want the changes you can discard them:

```
>>> device.discard_config()
```

### Committing the Configuration with a Required Confirmation

For certain platforms, you can also commit the configuration and set a revert timer. If you do not confirm the commit, by executing confirm_commit(), before the revert timer expires, then the configuration will be automatically rolled back to its previous state (and the candidate configuration will be discarded):

```
# Load new candidate config
>>> device.load_replace_candidate(filename=filename)

# Look at the pending changes
>>> print(device.compare_config())
```

(continues on next page)

(continued from previous page)

```
@@ -5,6 +5,8 @@
transceiver qsfp default-mode 4x10G
!
hostname arista9-napalm
+!
+ntp server 130.126.24.24
!
spanning-tree mode rapid-pvst
!

# Commit the changes with a 300 second revert timer.
device.commit_config(revert_in=300)

# You can now use the has_pending_commit() method to check for an in-process commit-
↪confirm
>>> device.has_pending_commit()
True

# To confirm the commit (i.e. ensure the change is permanently committed).
>>> device.confirm_commit()

# At this point there should be no pending commits.
>>> device.has_pending_commit()
False
```

### Immediately Canceling a Pending Commit-Confirm

Alternatively, to immediately cancel a pending commit_config with the revert timer set, you can execute the rollback() method:

```
>>> device.load_replace_candidate(filename=filename)
>>> device.commit_config(revert_in=300)
>>> device.has_pending_commit()
True

>>> device.rollback()
>>> device.has_pending_commit()
False

# At this point, our change would have been rolled-back (the change in this case␣
↪added an 'ntp server').
>>> output = device.get_config()["running"]
>>> "ntp" in output
False
```

### Allowing the Revert Timer to Expire

Finally, you can cancel a pending commit-confirm by letting the revert timer expire:

```
>>> device.load_replace_candidate(filename=filename)
>>> device.commit_config(revert_in=60)
>>> device.has_pending_commit()
True
```

(continues on next page)

(continued from previous page)

```
# Sleeping 80 seconds
>>> time.sleep(80)

# The device has automatically rolled-back the config to its previous state.
>>> device.has_pending_commit()
False
```

#### Rollback Changes

If for some reason you committed the changes and you want to rollback:

```
>>> device.rollback()
```

#### Disconnecting

To close the session with the device just do:

```
>>> device.close()
```

### 3.2.6 Context Manager

In the previous tutorial we used the methods `open()` to connect to the device and `close()` to disconnect. Using those methods are useful if you want to do complex or asynchronous code. However, for most situations you should try to stick with the context manager. It handles opening and closing the session automatically and it's the pythonic way:

```

```

```
>>> from napalm import get_network_driver
>>> driver = get_network_driver('eos')
>>> with driver('localhost', 'vagrant', 'vagrant', optional_args={'port': 12443}) as␣
↪device:
...     print(device.get_facts())
...     print(device.get_interfaces_counters())
...
{'os_version': u'4.15.2.1F-2759627.41521F', 'uptime': 2010, 'interface_list': [u
↪'Ethernet1', u'Ethernet2', u'Management1'], 'vendor': u'Arista', 'serial_number': u'
↪', 'model': u'vEOS', 'hostname': u'NEWHOSTNAME', 'fqdn': u'NEWHOSTNAME'}
{u'Ethernet2': {'tx_multicast_packets': 1028, 'tx_discards': 0, 'tx_octets': 130744,
↪'tx_errors': 0, 'rx_octets': 0, 'tx_unicast_packets': 0, 'rx_errors': 0, 'tx_
↪broadcast_packets': 0, 'rx_multicast_packets': 0, 'rx_broadcast_packets': 0, 'rx_
↪discards': 0, 'rx_unicast_packets': 0}, u'Management1': {'tx_multicast_packets': 0,
↪'tx_discards': 0, 'tx_octets': 99664, 'tx_errors': 0, 'rx_octets': 105000, 'tx_
↪unicast_packets': 773, 'rx_errors': 0, 'tx_broadcast_packets': 0, 'rx_multicast_
↪packets': 0, 'rx_broadcast_packets': 0, 'rx_discards': 0, 'rx_unicast_packets': 0},␣
↪u'Ethernet1': {'tx_multicast_packets': 1027, 'tx_discards': 0, 'tx_octets': 130077,
↪'tx_errors': 0, 'rx_octets': 0, 'tx_unicast_packets': 0, 'rx_errors': 0, 'tx_
↪broadcast_packets': 0, 'rx_multicast_packets': 0, 'rx_broadcast_packets': 0, 'rx_
↪discards': 0, 'rx_unicast_packets': 0}}
```

### 3.2.7 Extend Driver

Occassionally you may have a need that does not fit within one of Napalm's methods, nor will support ever be expected. As an example, if you wanted to build a parser to filter your unique banner and return structured data from it, you could extend the driver. The positive side effect is that tools such as Salt, Ansible, and Netbox implicitly have access to these methods.

The get_driver method, is simply looking for a *custom_napalm.<os>* driver first, and then fail to the normal napalm driver.

```
try:
    module = importlib.import_module("custom_" + module_install_name)
except ImportError:
    module = importlib.import_module(module_install_name)
```

#### Extending a Driver

By simply adding custom_napalm folder with an *__init__.py* and an *<os>.py* (e.g. *ios.py*) with class built to inherit the os class, you can expose all of the napalm methods, and your custom ones. This may sound like a lot, but this is here is a simple example of how to inherit the OS driver and all the requirements.

```
from napalm.ios.ios import IOSDriver
class CustomIOSDriver(IOSDriver):
    """Custom NAPALM Cisco IOS Handler."""
    def get_my_custom_method(self):
        pass
```

Sample python path custom_napalm directory.:

```
custom_napalm/
├── __init__.py
└── ios.py
```

#### Creating a Custom Method

Bulding on the previous example, we can create a a simple parse to return what our custom enviornment is looking for.

```
def get_my_banner(self):
    command = 'show banner motd'
    output = self._send_command(command)

    return_vars = {}
    for line in output.splitlines():
        split_line = line.split()
        if "Site:" == split_line[0]:
            return_vars["site"] = split_line[1]
        elif "Device:" == split_line[0]:
            return_vars["device"] = split_line[1]
        elif "Floor:" == split_line[0]:
            return_vars["floor"] = split_line[1]
        elif "Room:" == split_line[0]:
            return_vars["room"] = split_line[1]
    return return_vars
```

Which can build.

```
>>> import napalm
>>> ios_device='10.1.100.49'
>>> ios_user='ntc'
>>> ios_password='ntc123'
>>> driver = napalm.get_network_driver('ios')
>>> device = driver(ios_device, ios_user, ios_password)
>>> device.open()
>>> device.get_my_banner()
{'device': u'NYC-SW01', 'room': u'1004', 'site': u'NYC', 'floor': u'10'}
```

#### Custom Driver Notes

Please note that since there is no base class *get_my_banner* method, if you attempt to access this method from an os that is not supporting, then it will fail ungracefully. To alleviate that, you can raise *NotImplementedError* methods in other os's. It is up to the user to be able to support their own environment.

```
def get_my_banner(self):
    raise NotImplementedError
```

This feature is meant to allow for maximum amount of flexibility, but it is up to the user to ensure they do not run into namespace issues, and follow best practices.

### 3.2.8 Wrapping up

You've now tried the main pieces of NAPALM:

- using NAPALM to get, set, and diff the configuration of a device manually
- driving NAPALM using Python

#### Shutting down

Shut down the Vagrant virtual boxes. You can recreate them later using `vagrant up` if needed.:

```
$ vagrant destroy -f
```

#### Next Steps

There are many possible steps you could take next:

- create Vagrant boxes for other devices
- explore using configuration management tools (Ansible, Chef, Salt, etc.)

Thanks for trying NAPALM! Please contribute to this documentation and help grow the NAPALM community!

### 3.2.9 napalm-ansible

Collection of ansible modules that use napalm to retrieve data or modify configuration on networking devices.

### Modules

The following modules are currently available:

- napalm_get_facts
- napalm_install_config
- napalm_validate

### Install

To install, clone napalm-ansible into your ansible module path. This will depend on your own setup and contents of your ansible.cfg file which tells ansible where to look for modules. For more in-depth explanation, see the Ansible Docs.

If your ansible.cfg looks like:

```
[defaults]
library = ~/workspace/napalm-ansible
```

Then you can do the following:

```
cd ~/workspace

git clone
```

If your ansible.cfg looks like:

```
[defaults]
library = ~/workspace/napalm-ansible
```

Then you can do the following:

```
cd ~/workspace

git clone https://github.com/napalm-automation/napalm-ansible.git



user@hostname:~/workspace ls -la
total 12
drwxrwxr-x 3 user user 4096 Feb 26 12:51 .
drwxr-xr-x 7 user user 4096 Feb 26 12:49 ..
drwxrwxr-x 5 user user 4096 Feb 26 12:51 napalm-ansible
```

From here you would add your playbook(s) for your project, for example:

```
mkdir ansible-playbooks

user@hostname:~/workspace ls -la
total 12
drwxrwxr-x 3 user user 4096 Feb 26 12:51 .
drwxr-xr-x 7 user user 4096 Feb 26 12:49 ..
drwxrwxr-x 5 user user 4096 Feb 26 12:51 napalm-ansible
drwxrwxr-x 5 user user 4096 Feb 26 12:53 ansible-playbooks
```

### Dependencies

napalm 1.00.0 or later

### Examples

Example to retrieve facts from a device:

```
- name: get facts from device
  napalm_get_facts:
    hostname={{ inventory_hostname }}
    username={{ user }}
    dev_os={{ os }}
    password={{ passwd }}
    filter='facts,interfaces,bgp_neighbors'
  register: result

- name: print data
  debug: var=result
```

Example to install config on a device:

```
- assemble:
    src=../compiled/{{ inventory_hostname }}/
    dest=../compiled/{{ inventory_hostname }}/running.conf

 - napalm_install_config:
    hostname={{ inventory_hostname }}
    username={{ user }}
    dev_os={{ os }}
    password={{ passwd }}
    config_file=../compiled/{{ inventory_hostname }}/running.conf
    commit_changes={{ commit_changes }}
    replace_config={{ replace_config }}
    get_diffs=True
    diff_file=../compiled/{{ inventory_hostname }}/diff
```

Example to get compliance report:

```
- name: GET VALIDATION REPORT
  napalm_validate:
    username: "{{ un }}"
    password: "{{ pwd }}"
    hostname: "{{ inventory_hostname }}"
    dev_os: "{{ dev_os }}"
    validation_file: validate.yml
```

### A More Detailed Example

It's very often we come to these tools needing to know how to run before we can walk. Please review the Ansible Documentation as this will answer some basic questions. It is also advised to have some kind of yaml linter or syntax checker available.

Non parameterized example with comments to get you started:

```
- name: Test Inventory #The Task Name
  hosts: cisco         #This will be in your ansible inventory file
  connection: local    #Required
  gather_facts: no     #Do not gather facts

  tasks:                                        #Begin Tasks
    - name: get facts from device               #Task Name
      napalm_get_facts:                         #Call the napalm module, in this case␣
↪napal_get_facts
        optional_args: {'secret': password}  #The enable password for Cisco IOS
        hostname: "{{ inventory_hostname }}" #This is a parameter and is derived from␣
↪your ansible inventory file
        username: 'user'                        #The username to ssh with
        dev_os: 'ios'                           #The hardware operating system
        password: 'password'                    #The line level password
        filter: 'facts'                         #The list of items you want to retrieve.␣
↪The filter keyword is _inclusive_ of what you want
      register: result                          #Ansible function for collecting output

    - name: print results                       #Task Name
      debug: msg="{{ result }}"                 #Display the collected output
```

Keeping with our example dir at the beginning of the Readme, we now have this layout:

```
user@host ~/workspace/ansible-playbooks
08:16 $ ls -la
total 32
drwxrwxr-x 3 user user 4096 Feb 26 07:24 .
drwxrwxr-x 8 user user 4096 Feb 25 16:32 ..
-rw-rw-r-- 1 user user  404 Feb 26 07:24 inventory.yaml
```

You would run this playbook like as:

```
cd ~/workspace

ansible-playbook ansible-playbooks/inventory.yaml
```

And it should produce output similar to this:

```
PLAY [Push config to switch group.] ********************************************

TASK [get facts from device] ***************************************************
ok: [192.168.0.11]

TASK [print results]␣
↪*********************************************************************
ok: [192.168.0.11] => {
    "msg": {
        "ansible_facts": {
            "facts": {
                "fqdn": "router1.not set",
                "hostname": "router1",
                "interface_list": [
                    "FastEthernet0/0",
                    "GigabitEthernet1/0",
                    "GigabitEthernet2/0",
                    "GigabitEthernet3/0",
```

(continues on next page)

(continued from previous page)

```
                    "GigabitEthernet4/0",
                    "POS5/0",
                    "POS6/0"
                ],
                "model": "7206VXR",
                "os_version": "7200 Software (C7200-ADVENTERPRISEK9-M), Version 15.
↪2(4)S7, RELEASE SOFTWARE (fc4)",
                "serial_number": "0123456789",
                "uptime": 420,
                "vendor": "Cisco"
            }
        },
        "changed": false
    }
}

PLAY RECAP *********************************************************************
192.168.0.11               : ok=2    changed=0    unreachable=0    failed=0
```

### 3.2.10 Unit tests: Mock driver

A mock driver is a software that imitates the response pattern of another system. It is meant to do nothing but returns the same predictable result, usually of the cases in a testing environment.

A driver *mock* can mock all actions done by a common napalm driver. It can be used for unit tests, either to test napalm itself or inside external projects making use of napalm.

#### Overview

For any action, the `mock` driver will use a file matching a specific pattern to return its content as a result.

Each of these files will be located inside a directory specified at the driver initialization. Their names depend on the entire call name made to the driver, and about their order in the call stack.

#### Replacing a standard driver by a `mock`

Get the driver in napalm:

```
>>> import napalm
>>> driver = napalm.get_network_driver('mock')
```

And instantiate it with any host and credentials:

```
device = driver(
    hostname='foo', username='user', password='pass',
    optional_args={'path': path_to_results}
)
```

Like other drivers, `mock` takes optional arguments:

- `path` - Optional directory where results files are located (defaults to the current directory).

Open the driver:

```
>>> device.open()
```

A user should now be able to call any function of a standard driver:

```
>>> device.get_network_instances()
```

But should get an error because no mocked data is yet written:

```
NotImplementedError: You can provide mocked data in get_network_instances.1
```

### Mocked data

We will use `/tmp/mock` as an example of a directory that will contain our mocked data. Define a device using this path:

```
>>> with driver('foo', 'user', 'pass', optional_args={'path': '/tmp/mock'}) as device:
```

### Mock a single call

In order to be able to call, for example, `device.get_interfaces()`, a mocked data is needed.

To build the file name that the driver will look for, take the function name (`get_interfaces`) and suffix it with the place of this call in the device call stack.

---

**Note:** `device.open()` counts as a command. Each following order of call will start at 1.

---

Here, `get_interfaces` is the first call made to `device` after `open()`, so the mocked data need to be put in `/tmp/mock/get_interfaces.1`:

```
{
    "Ethernet1/1": {
        "is_up": true, "is_enabled": true, "description": "",
        "last_flapped": 1478175306.5162635, "speed": 10000,
        "mac_address": "FF:FF:FF:FF:FF:FF"
    },
    "Ethernet1/2": {
        "is_up": true, "is_enabled": true, "description": "",
        "last_flapped": 1492172106.5163276, "speed": 10000,
        "mac_address": "FF:FF:FF:FF:FF:FF"
    }
}
```

The content is the wanted result of `get_interfaces` in JSON, exactly as another driver would return it.

### Mock multiple iterative calls

If `/tmp/mock/get_interfaces.1` was defined and used, for any other call on the same device, the number of calls needs to be incremented.

For example, to call `device.get_interfaces_ip()` after `device.get_interfaces()`, the file `/tmp/mock/get_interfaces_ip.2` needs to be defined:

```
{
    "Ethernet1/1": {
        "ipv6": {"2001:DB8::": {"prefix_length": 64}}
    }
}
```

### Mock a CLI call

`device.cli(commands)` calls are a bit different to mock, as a suffix corresponding to the command applied to the device needs to be added. As before, the data mocked file will start by `cli` and the number of calls done before (here, `cli.1`). Then, the same process needs to be applied to each command.

Each command needs to be sanitized: any special character (‘‘ -,./‘‘, etc.) needs to be replaced by _. Add the index of this command as it is sent to `device.cli()`. Each file then will contain the raw wanted output of its associated command.

### Example

Example with 2 commands, `show interface Ethernet 1/1` and `show interface Ethernet 1/2`.

To define the mocked data, create a file `/tmp/mock/cli.1.show_interface_Ethernet_1_1.0`:

```
Ethernet1/1 is up
admin state is up, Dedicated Interface
```

And a file `/tmp/mock/cli.1.show_interface_Ethernet_1_2.1`:

```
Ethernet1/2 is up
admin state is up, Dedicated Interface
```

And now they can be called:

```
>>> device.cli(["show interface Ethernet 1/1", "show interface Ethernet 1/2"])
```

### Mock an error

The *mock* driver can raise an exception during a call, to simulate an error. An error definition is actually a json composed of 3 keys:

- *exception*: the exception type that will be raised
- *args* and *kwargs*: parameters sent to the exception constructor

For example, to raise the exception *ConnectionClosedException* when calling `device.get_interfaces()`, the file `/tmp/mock/get_interfaces.1` needs to be defined:

```
{
    "exception": "napalm.base.exceptions.ConnectionClosedException",
    "args": [
        "Connection closed."
    ],
    "kwargs": {}
}
```

Now calling *get_interfaces()* for the 1st time will raise an exception:

```
>>> device.get_interfaces()
ConnectionClosedException: Connection closed
```

As before, mock will depend on the number of calls. If a second file `/tmp/mock/get_interfaces.2` was defined and filled with some expected data (not an exception), retrying *get_interfaces()* will run correctly if the first exception was caught.

## 3.3 Validating deployments

Let's say you just deployed a few devices and you want to validate your deployment. To do that, you can write a YAML file describing the state you expect your devices to be in and tell napalm to retrieve the state of the device and build a compliance report for you.

As always, with napalm, doing this is very easy even across multiple vendors : )

---

**Note:** Note that this is meant to validate **state**, meaning live data from the device, not the configuration. Because that something is configured doesn't mean it looks as you want.

---

### 3.3.1 Documentation

Writing validators files that can be interpreted by napalm is very easy. You have to start by telling napalm how to retrieve that piece of information by using as key the name of the getter and then write the desired state using the same format the getter would retrieve it. For example:

```
---
- get_facts:
    os_version: 7.0(3)I2(2d)
    interface_list:
      _mode: strict
      list:
        - Vlan5
        - Vlan100
    hostname: n9k2

- get_environment:
    memory:
      used_ram: '<15.0'
      available_ram: '10.0<->20.0'
    cpu:
      0/RP0/CPU0
        '%usage': '<15.0'

- get_bgp_neighbors:
    global:
      router_id: 192.0.2.2
      peers:
        _mode: strict
        192.0.2.2:
          is_enabled: true
          address_family:
```

(continues on next page)

(continued from previous page)

```
            ipv4:
              sent_prefixes: 5
              received_prefixes: '<10'
            ipv6:
              sent_prefixes: 2
              received_prefixes: '<5'

- get_interfaces_ip:
    Ethernet2/1:
      ipv4:
        192.0.2.1:
          prefix_length: 30

- ping:
    _name: ping_google
    _kwargs:
      destination: 8.8.8.8
      source: 192.168.1.1
    success:
      packet_loss: 0
    _mode: strict

- ping:
    _name: something_else
    _kwargs:
      destination: 10.8.2.8
      source: 192.168.1.1
    success:
      packet_loss: 0
    _mode: strict
```

A few notes:

- You don't have to validate the entire state of the device, you might want to validate certain information only. For example, with the getter `get_interfaces_ip` we are only validating that the interface `Ethernet2/1` has the IP address `192.0.2.1/30`. If there are other interfaces or if that same interface has more IP's, it's ok.
- You can also have a more strict validation. For example, if we go to `get_bgp_neighbors`, we want to validate there that the `default` vrf has *only* the BGP neighbor `192.0.2.2`. We do that by specifying at that level `_mode: strict`. Note that the strict mode is specific to a level (you can add it to as many levels as you want). So, going back the the example, we are validating that only that BGP neighbor is present on that vrf but we are not validating that other vrfs don't exist. We are not validating all the data inside the BGP neighbor either, we are only validating the ones we specified.
- Lists of objects to be validated require an extra key `list`. You can see an example with the `get_facts` getter. Lists can be strict as well. In this case, we want to make sure the device has only those two interfaces.
- We can also use comparison on the conditions of numerical validate. For example, if you want to validate there that the `cpu``and ``memory` into `get_environment` are `15%` or less. We can use writing comparison operators such as `<15.0` or `>10.0` in this case, or range with the operator syntax of `<->` such as `10.0<->20.0` or `10<->20`.
- Some methods require extra arguments, for example `ping`. You can pass arguments to those methods using the magic keyword `_kwargs`. In addition, an optional keyword `_name` can be specified to override the name in the report. Useful for having a more descriptive report or for getters than can be run multiple times

### 3.3.2 Example

Let's say we have two devices, one running `eos` and another one running `junos`. A typical script could start like this:

```
from napalm import get_network_driver
import pprint

eos_driver = get_network_driver("eos")
eos_config = {
    "hostname": "localhost",
    "username": "vagrant",
    "password": "vagrant",
    "optional_args": {"port": 12443},
}

junos_driver = get_network_driver("junos")
junos_config = {
    "hostname": "localhost",
    "username": "vagrant",
    "password": "",
    "optional_args": {"port": 12203},
}
```

Now, let's validate that the devices are running a specific version and that the management IP is the one we expect. Let's start by writing the validator files.

- `validate-eos.yml`:

```
---
- get_facts:
    os_version: 4.17

- get_interfaces_ip:
    Management1:
        ipv4:
            10.0.2.14:
                prefix_length: 24
            _mode: strict
```

- `validate-junos.yml`:

```
---
- get_facts:
    os_version: 12.1X47

- get_interfaces_ip:
    ge-0/0/0.0:
        ipv4:
            10.0.2.15:
                prefix_length: 24
            _mode: strict
```

**Note:** You can use regular expressions to validate values.

As you can see we are validating that the OS running is the one we want and that the management interfaces have only the IP we expect it to have. Now we can validate the devices like this:

```
>>> with eos_driver(**eos_config) as eos:
...     pprint.pprint(eos.compliance_report("validate-eos.yml"))
...
{u'complies': False,
 u'skipped': [],
 'get_facts': {u'complies': False,
               u'extra': [],
               u'missing': [],
               u'present': {'os_version': {u'actual_value': u'4.15.2.1F-2759627.41521F
↪',
                                           u'complies': False,
                                           u'nested': False}}},
 'get_interfaces_ip': {u'complies': True,
                       u'extra': [],
                       u'missing': [],
                       u'present': {'Management1': {u'complies': True,
                                                    u'nested': True}}}}
```

Let's take a look first to the report. The first thing we have to note is the first key `complies` which is telling us that overall, the device is not compliant. Now we can dig in on the rest of the report. The `get_interfaces_ip` part seems to be complying just fine, however, the `get_facts` is complaining about something. If we keep digging we will see that the `os_version` key we were looking for is present but it's not complying as its actual value is not the one we specified; it is `4.15.2.1F-2759627.41521F`.

Now let's do the same for junos:

```
>>> with junos_driver(**junos_config) as junos:
...     pprint.pprint(junos.compliance_report("validate-junos.yml"))
...
{u'complies': True,
 u'skipped': [],
 'get_facts': {u'complies': True,
               u'extra': [],
               u'missing': [],
               u'present': {'os_version': {u'complies': True,
                                           u'nested': False}}},
 'get_interfaces_ip': {u'complies': True,
                       u'extra': [],
                       u'missing': [],
                       u'present': {'ge-0/0/0.0': {u'complies': True,
                                                   u'nested': True}}}}
```

This is great, this device is fully compliant. We can check the outer `complies` key is set to `True`. However, let's see what happens if someone adds and extra IP to `ge-0/0/0.0`:

```
>>> with junos_driver(**junos_config) as junos:
...     pprint.pprint(junos.compliance_report("validate-junos.yml"))
...
{u'complies': False,
 u'skipped': [],
 'get_facts': {u'complies': True,
               u'extra': [],
               u'missing': [],
               u'present': {'os_version': {u'complies': True,
                                           u'nested': False}}},
 'get_interfaces_ip': {u'complies': False,
                       u'extra': [],
```

(continues on next page)

(continued from previous page)

```
                      u'missing': [],
                      u'present': {'ge-0/0/0.0': {u'complies': False,
                                                  u'diff': {u'complies': False,
                                                            u'extra': [],
                                                            u'missing': [],
                                                            u'present': {'ipv4': {u
↪'complies': False,
                                                                                 u
↪'diff': {u'complies': False,
                                                                                   ␣
↪       u'extra': [u'172.20.0.1'],
                                                                                   ␣
↪       u'missing': [],
                                                                                   ␣
↪       u'present': {'10.0.2.15': {u'complies': True,
                                                                                   ␣
↪                                  u'nested': True}}},
                                                                                 u
↪'nested': True}}},
                                                  u'nested': True}}}}
```

After adding the extra IP it seems the device is not compliant anymore. Let's see what happened:

- Outer `complies` key is telling us something is wrong.
- `get_facts` is fine.
- `get_interfaces_ip` is telling us something interesting. Note that is saying that `ge-0/0/0.0` has indeed the IPv4 address `10.0.2.15` as noted by being `present` and with the inner `complies` set to `True`. However, it's telling us that there is an `extra` IP `172.20.0.1`.

The output might be a bit complex for humans but it's predictable and very easy to parse so it's great if you want to integrate it with your documentation/reports by using simple `jinja2` templates.

#### Skipped tasks

In cases where a method is not implemented, the validation will be skipped and the result will not count towards the result. The report will let you know a method wasn't executed in the following manner:

```
...
"skipped": [ "method_not_implemented", ],
"method_not_implemented": {
    "reason": "NotImplemented",
    "skipped": True,
    }
...
```

`skipped` will report the list of methods that were skipped. For details about the reason you can dig into the method's report.

### 3.3.3 CLI & Ansible

If you prefer, you can also make use of the validate functionality via the CLI with the command `cl_napalm_validate` or with ansible plugin. You can find more information about them here:

- CLI - https://github.com/napalm-automation/napalm/pull/168

- Ansible - https://github.com/napalm-automation/napalm-ansible/blob/master/napalm_ansible/modules/napalm_validate.py

### 3.3.4 Why this and what's next

As mentioned in the introduction, this is interesting to validate state. You could, for example, very easily check that your BGP neighbors are configured and that the state is up. It becomes even more interesting if you can build the validator file from data from your inventory. That way you could deploy your network and verify it matches your expectations all the time without human intervention.

Something else you could do is write the validation file manually prior to a maintenance based on some gathered data from the network and on your expectations. You could, then, perform your changes and use this tool to verify the state of the network is exactly the one you wanted. No more forgetting things or writing one-offs scripts to validate deployments.

## 3.4 Supported Devices

### 3.4.1 General support matrix

| _ | EOS | Junos | IOS-XR (NETCONF) | IOS-XR (XML-Agent) | NX-OS | NX-OS SSH | IOS |
|---|---|---|---|---|---|---|---|
| **Driver Name** | eos | junos | iosxr_netconf | iosxr | nxos | nxos_ssh | ios |
| **Structured data** | Yes | Yes | Yes | No | Yes | No | No |
| **Minimum version** | 4.15.0F | 12.1 | 7.0 | 5.1.0 | 6.1[1] | 6.3.2 | 12.4(20)T |
| **Backend library** | pyeapi | junos-eznc | ncclient | pyIOSXR | pynxos | netmiko | netmiko |
| **Caveats** | *EOS* | | *IOS-XR (NETCONF)* | | *NXOS* | *NXOS* | *IOS* |

**Warning:** Please, make sure you understand the caveats for your particular platforms before using the library.

### 3.4.2 Configuration support matrix

| _ | EOS | Junos | IOS-XR (NETCONF) | IOS-XR (XML-Agent) | NX-OS | IOS |
|---|---|---|---|---|---|---|
| **Config. replace** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Config. merge** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Commit Confirm** | Yes | Yes | No | No | No | Yes |
| **Compare config** | Yes | Yes | Yes | Yes[2] | Yes[4] | Yes |
| **Atomic Changes** | Yes | Yes | Yes | Yes | Yes/No[5] | Yes/No[5] |
| **Rollback** | Yes[3] | Yes | Yes | Yes | Yes/No[5] | Yes |

[1] NX-API support on the Nexus 5k, 6k and 7k families was introduced in version 7.2

**Warning:** Before building a workflow to deploy configuration it is important you understand what the table above means; what are atomic changes and which devices support it, what does replacing or merging configuration mean, etc. The key to success is to test your workflow and to try to break things on a lab first.

### 3.4.3 Getters support matrix

**Note:** The following table is built automatically. Every time there is a release of a supported driver a built is triggered. The result of the tests are aggregated on the following table.

### 3.4.4 Other methods

| _ | EOS | Junos | IOS-XR (NETCONF) | IOS-XR | NX-OS | IOS |
|---|---|---|---|---|---|---|
| **load_template** | | | | | | |
| **ping** | | | | | | |
| **traceroute** | | | | | | |

### 3.4.5 Available configuration templates

- `set_hostname` (JunOS, IOS-XR, IOS) - Configures the hostname of the device.
- `set_ntp_peers` (JunOS, IOS-XR, EOS, NXOS, IOS) - Configures NTP peers of the device.
- `delete_ntp_peers` (JunOS, IOS-XR, EOS, NXOS, IOS): Removes NTP peers from device's configuration.
- `set_probes` (JunOS, IOS-XR): Configures RPM/SLA probes.
- `schedule_probes` (IOS-XR): On Cisco devices, after defining the SLA probes, it is mandatory to schedule them. Defined also for JunOS as empty template, for consistency reasons.
- `delete_probes` (JunOS, IOS-XR): Removes RPM/SLA probes.

### 3.4.6 Caveats

#### EOS

##### Minimum Version

To be able to support the `compare_config`, `load_merge_candidate` or `load_replace_candidate` methods you will require to run at least EOS version *4.15.0F*.

The ssh driver for the config session timers requires you to run at least EOS 4.18.0F.

[2] Hand-crafted by the API as the device doesn't support the feature.
[4] For merges, the diff is very simplistic. See caveats.
[5] No for merges. See caveats.
[3] Not supported but emulated. Check caveats.

### Multi-line/HEREDOC

EOS configuration is loaded via `pyeapi.eapilib.Node.run_commands()`, which by itself cannot handle multi-line commands such as `banner motd`. The helper function `EOSDriver._load_config()` will attempt to detect HEREDOC commands in the input configuration and convert them into a dictionary that eAPI understands

### Rollback

The rollback feature is supported only when committing from the API. In reality, what the API does during the commit operation is as follows:

```
copy startup-config flash:rollback-0
```

And the rollback does:

```
configure replace flash:rollback-0
```

This means that the rollback will be fine as long as you only use this library. If you are going to do changes outside this API don't forget to mark your last rollback point manually.

### IOS

### Prerequisites

IOS has no native API to play with, that's the reason why we used the Netmiko library to interact with it. Having Netmiko installed in your working box is a prerequisite.

Check napalm-ios/requirements.txt for Netmiko version requirement

Full ios driver support requires configuration rollback on error:

```
Cisco IOS requirements for 'Configuration Rollback Confirmed Change' feature.
12.2(33)SRC
12.2(33)SB
12.4(20)T
12.2(33)SXI
```

Downgraded ios driver support (i.e. no auto rollback on configuration error for replace operation):

```
Cisco IOS requirements for 'Configuration Replace and Configuration Rollback' feature.
12.3(7)T
12.2(25)S
12.3(14)T
12.2(27)SBC
12.2(31)SB2
12.2(33)SRA
12.2(33)SXH
12.2(33)SB
```

Note, to disable auto rollback you must add the *auto_rollback_on_error=False* optional argument.

### Archive

IOSDriver requires that the *archive* functionality be enabled to perform auto-rollback on error. Make sure it's enabled and set to a local filesystem (for example 'flash:' or 'bootflash:':

```
archive
  path flash:archive
  write-memory
```

### Configuration file

- IOS requires config file to begin with a *version* eg. *15.0* and *end* marker at the end of the file. Otherwise IOS will reject *configure replace* operation.
- For the diff to work properly, indentation of your candidate file has to exactly match the indentation in the running config.
- Finish blocks with *!* as with the running config, otherwise, some IOS version might not be able to generate the diff properly.

### Self-Signed Certificate (and the hidden tab character)

Cisco IOS adds a tab character into the self-signed certificate. This exists on the quit line:

```
crypto pki certificate chain TP-self-signed-1429897839
 certificate self-signed 01
  3082022B 30820194 A0030201 02020101 300D0609 2A864886 F70D0101 05050030
  ...
  ...
  ...
  9353BD17 C345E1D7 71AFD125 D23D7940 2DECBE8E 46553314 396ACC63 34839EF7
  3C056A00 7E129168 F0CD3692 F53C62
    quit
```

The quit line reads as follows:

```
>>> for char in line:
...   print("{}: {}".format(repr(char), ord(char)))
...
' ': 32     # space
' ': 32     # space
'\t': 9     # tab
'q': 113
'u': 117
'i': 105
't': 116
'\n': 10
```

This implies that you will not generally be able to copy-and-paste the self-signed certificate. As when you copy-and-paste it, the tab character gets converted to spaces.

You will need to transfer the config file directly from the device (for example, SCP the config file) or you will need to manually construct the quit line exactly right.

Cisco IOS is very particular about the self-signed certificate and will reject replace operations with an invalid certificate. Cisco IOS will also reject replace operations that are missing a certificate (when the current configuration has a self-signed certificate).

### Banner

IOS requires that the banner use the ETX character (ASCII 3). This looks like a cntl-C in the file, but as a single character. It is NOT a separate '^' + 'C' character, but an ASCII3 character:

```
banner motd ^C
    my banner test
^C

>>> etx_char = chr(3)
>>> with open("my_config.conf", "a") as f:
...   f.write("banner motd {}\n".format(etx_char))
...   f.write("my banner test\n")
...   f.write("{}\n".format(etx_char))
...
>>> quit()
```

Configure replace operations will reject a file with a banner unless it uses the ASCII character. Note, this likely also implies you cannot just copy-and-paste what you see on the screen.

In vim insert, you can also type <ctrl>+V, release only the V, then type C

### File Operation Prompts

By default IOS will prompt for confirmation on file operations. These prompts need to be disabled before the NAPALM-ios driver performs any such operation on the device. This can be controlled using the *auto_file_prompt* optional argument:

- *auto_file_prompt=True* (default): NAPALM will automatically add *file prompt quiet* to the device configuration before performing file operations, and un-configure it again afterwards. If the device already had the command in its configuration then it will be silently removed as a result, and this change will not show up in the output of *compare_config()*.
- *auto_file_prompt=False*: Disable the above automated behaviour. The managed device must have *file prompt quiet* in its running-config already, otherwise a *CommandErrorException* will be raised when file operations are attempted.

### SCP File Transfers

The NAPALM-ios driver requires SCP to be enabled on the managed device. SCP server functionality is disabled in IOS by default, and is configured using *ip scp server enable*.

If an operation requiring a file transfer is attempted, but the necessary configuration is not present, a *CommandErrorException* will be raised.

### Notes

- The NAPALM-ios driver supports all Netmiko arguments as either standard arguments (hostname, username, password, timeout) or as optional_args (everything else).

### NXOS

#### Notes on configuration replacement

Config files aren't aren't normal config files but special "checkpoint" files. That's because on NXOS the only way to replace a config without reboot is to rollback to a checkpoint (which could be a file). These files explicitly list a lot of normally implicit config lines, some of them starting with `!#`. The `!#` part isn't necessary for the rollback to work, but leaving these lines out can cause erratic behavior. See the "Known gotchas" section below.

#### Prerequisites

Your device must be running NXOS 6.1. The features `nxapi` server `scp-server` must be enabled. On the device and any checkpoint file you push, you must have the lines:

```
feature scp-server
feature nxapi
```

#### Getting a base checkpoint file

An example of a checkpoint file can be seen in `test/unit/nxos/new_good.conf`. You can get a checkpoint file representing your device's current config by running the `_get_checkpoint_file()` function in the `napalm.nxos` driver:

```
device.open()
checkpoint = device._get_checkpoint_file()
print(checkpoint)
device.close()
```

#### Known gotchas

- Leaving out a `shutdown` or `no shutdown` line will cause the switch to toggle the up/down state of an interface, depending on it's current state.
- `!#switchport trunk allowed vlan 1-4094` is required even if the switchport is in `switchport mode access`. However if `!#switchport trunk allowed vlan 1-4094` is included with `no switchport`, the configuration replacement will fail.
- Vlans are listed vertically. For example `vlan 1, 10, 20, 30` will fail. To succeed, you need:

```
vlan 1
vlan 10
vlan 20
vlan 30
```

#### Diffs

Diffs for config replacement are a list of commands that would be needed to take the device from it's current state to the desired config state. See `test/unit/nxos/new_good.diff` as an example.

### Notes on configuration merging

Merges are currently implemented by simply applying the the merge config line by line. This doesn't use the checkpoint/rollback functionality. As a result, merges are **not atomic**.

### Diffs

Diffs for merges are simply the lines in the merge candidate config. Netutils is used for creating the merge diff between the candidate and running configurations. One caveat of using netutils diff of configurations is that the diff is performed offline and not online in the device.

Example assuming that the device config contains:

```
interface loopback0
  ip address 10.1.4.4/32
  ip router ospf 100 area 0.0.0.1
```

Then what you will get with the diff:

```
candidate_cfg = """
interface loopback0
  ip address 10.1.4.5/32
  ip router ospf 100 area 0.0.0.1
"""

nxos1.load_merge_candidate(config=candidate_cfg)

print(nxos1.compare_config())
interface loopback0
  ip address 10.1.4.5/32
```

## IOS-XR (NETCONF)

### Minimum IOS-XR OS Version

Only devices running IOS-XR 7.0 or later are supported by NAPALM and the IOS-XR NETCONF driver.

### Device management using XML Configuration

Using `iosxr_netconf` and a `config_encoding="xml"` for NAPALM configuration operations is entirely experimental. There is a very good chance XML configurations will not work properly and that only small subsections of the configuration will be configurable using merge operations.

### Device management using CLI Configuration

All configuration methods (`load_merge_candidate`, `load_replace_candidate`, `get_config`, `compare_config`) support configuration encoded in XML and CLI (unstructured) format. This can be specified by using the `config_encoding` optional_args argument and setting it to either `cli` or `xml` (`cli` is the default value).

#### Retrieving device environment

In IOS-XR 64-bit devices that support an administration mode, the proper operation of `get_environment` requires that the `iosxr_netconf` driver session is authenticated against a username defined in that administration mode.

### 3.4.7 Optional arguments

NAPALM supports passing certain optional arguments to some drivers. To do that you have to pass a dictionary via the `optional_args` parameter when creating the object:

```
>>> from napalm import get_network_driver
>>> driver = get_network_driver('eos')
>>> optional_args = {'my_optional_arg1': 'my_value1', 'my_optional_arg2': 'my_value2'}
>>> device = driver('192.168.76.10', 'dbarroso', 'this_is_not_a_secure_password',␣
↪optional_args=optional_args)
>>> device.open()
```

#### List of supported optional arguments

- `allow_agent` (ios, iosxr, nxos_ssh) - Paramiko argument, enable connecting to the SSH agent (default: `False`).
- `alt_host_keys` (ios, iosxr, nxos_ssh) - If `True`, host keys will be loaded from the file specified in `alt_key_file`.
- `alt_key_file` (ios, iosxr, nxos_ssh) - SSH host key file to use (if `alt_host_keys` is `True`).
- `auto_probe` (junos) - A timeout in seconds, for which to probe the device. Probing determines if the device accepts remote connections. If *auto_probe* is set to `0`, no probing will be done. (default: `0`).
- `auto_rollback_on_error` (ios) - Disable automatic rollback (certain versions of IOS support configure replace, but not rollback on error) (default: `True`).
- `config_lock` (iosxr_netconf, iosxr, junos) - Lock the config during open() (default: `False`).
- `lock_disable` (junos) - Disable all configuration locking for management by an external system (default: `False`).
- `config_private` (junos) - Configure private, no DB locking (default: `False`).
- `canonical_int` (ios) - Convert operational interface's returned name to canonical name (fully expanded name) (default: `False`).
- `dest_file_system` (ios) - Destination file system for SCP transfers (default: `flash:`).
- `enable_password` (eos) - Password required to enter privileged exec (enable) (default: `''`).
- `force_no_enable` (ios, nxos_ssh) - Do not automatically enter enable-mode on connect (default: `False`).
- `global_delay_factor` (ios, nxos_ssh) - Allow for additional delay in command execution (default: `1`).
- `ignore_warning` (junos) - Allows to set *ignore_warning* when loading configuration to avoid exceptions via junos-pyez. (default: `False`).
- `keepalive` (iosxr, junos) - SSH keepalive interval, in seconds (default: `30` seconds).
- `key_file` (ios, iosxr_netconf, iosxr, junos, nxos_ssh) - Path to a private key file. (default: `False`).
- `port` (eos, ios, iosxr_netconf, iosxr, junos, nxos, nxos_ssh) - Allows you to specify a port other than the default.
- `secret` (ios, nxos_ssh) - Password required to enter privileged exec (enable) (default: `''`).

- `ssh_config_file` (ios, iosxr, junos, nxos_ssh) - File name of OpenSSH configuration file.
- `ssh_strict` (ios, iosxr, nxos_ssh) - Automatically reject unknown SSH host keys (default: `False`, which means unknown SSH host keys will be accepted).
- `ssl_verify` (nxos) - Requests argument, enable the SSL certificates verification. See requests ssl-cert-verification for valid values (default: `None` equivalent to `False`).
- `transport` (eos, ios, nxos) - Protocol to connect with (see *The transport argument* for more information).
- `use_keys` (ios, iosxr, nxos_ssh) - Paramiko argument, enable searching for discoverable private key files in `~/.ssh/` (default: `False`).
- `eos_autoComplete` (eos) - Allows to set *autoComplete* when running commands. (default: `None` equivalent to `False`)
- `config_encoding` (iosxr_netconf) - Set encoding to either `"xml"` or `"cli"` for configuration load methods. (default: `"cli"`)
- `eos_fn0039_config` (eos) - Transform old style configuration to the new style, available beginning with EOS release 4.23.0, as per FN 0039. Beware that enabling this option will change the configuration you're loading through NAPALM. Default: `False` (won't change your configuration commands). .. versionadded:: 3.0.1
- `force_cfg_session_invalid` (eos) - Force the config_session to be cleared in case of issues, like *discard_config* failure. (default: `False`)

### The transport argument

Certain drivers support providing an alternate transport in the `optional_args`, overriding the default protocol to connect with. Allowed transports are therefore device/library dependant:

| _ | EOS | NXOS | IOS |
|---|---|---|---|
| **Default** | `https` | `https` | `ssh` |
| **Supported** | `http`, `https`, `ssh` | `http`, `https` | `telnet`, `ssh` |

## 3.5 Command Line Tool

NAPALM ships with a very simple cli tool so you can use napalm straight from the CLI. It's use is quite simple and you can see the help with `--help`:

```
$ napalm --help
usage: napalm [-h] [--user USER] [--password PASSWORD] --vendor VENDOR
              [--optional_args OPTIONAL_ARGS] [--debug]
              hostname {configure,call,validate} ...

Command line tool to handle configuration on devices using NAPALM.The script
will print the diff on the screen

positional arguments:
  hostname              Host where you want to deploy the configuration.

optional arguments:
  -h, --help            show this help message and exit
  --user USER, -u USER  User for authenticating to the host. Default: user
                        running the script.
```

(continues on next page)

(continued from previous page)

```
  --password PASSWORD, -p PASSWORD
                        Password for authenticating to the host.If you do not
                        provide a password in the CLI you will be prompted.
  --vendor VENDOR, -v VENDOR
                        Host Operating System.
  --optional_args OPTIONAL_ARGS, -o OPTIONAL_ARGS
                        String with comma separated key=value pairs passed via
                        optional_args to the driver.
  --debug               Enables debug mode; more verbosity.

actions:
  {configure,call,validate}
    configure           Perform a configuration operation
    call                Call a napalm method
    validate            Validate configuration/state

Automate all the things!!!
```

You can mostly do three things:

1. Configure the device (dry-run with diff supported)
2. Call any method (like `get_interfaces` or `ping`)
3. Validate configuration/state

Let's see a few examples:

```
# napalm --user vagrant --password vagrant --vendor eos --optional_args "port=12443"␣
↪localhost configure new_config.txt --strategy merge --dry-run
@@ -8,7 +8,7 @@
 !
 transceiver qsfp default-mode 4x10G
 !
-hostname myhost
+hostname a-new-hostname
 !
 spanning-tree mode mstp
 !
@@ -20,6 +20,7 @@
 username vagrant privilege 15 role network-admin secret 5 $1$gxUZF/4Q
↪$FoUvji7hq0HpJGxc67PJM0
 !
 interface Ethernet1
+   description "TBD"
 !
 interface Ethernet2
 !
$ napalm --user vagrant --password vagrant --vendor eos --optional_args "port=12443"␣
↪localhost call get_interfaces
{
    "Ethernet2": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1502731278.4344141,
        "is_up": true,
        "mac_address": "08:00:27:3D:83:34",
        "speed": 0
```

(continues on next page)

(continued from previous page)

```
    },
    "Management1": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1502731294.598835,
        "is_up": true,
        "mac_address": "08:00:27:7D:44:C1",
        "speed": 1000
    },
    "Ethernet1": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1502731278.4342606,
        "is_up": true,
        "mac_address": "08:00:27:E6:4C:E9",
        "speed": 0
    }
}
$ napalm --user vagrant --password vagrant --vendor eos --optional_args "port=12443"␣
↪localhost call ping --method-kwargs "destination='127.0.0.1'"
{
    "success": {
        "packet_loss": 0,
        "rtt_stddev": 0.011,
        "rtt_min": 0.005,
        "results": [
            {
                "rtt": 0.035,
                "ip_address": "127.0.0.1"
            },
            {
                "rtt": 0.008,
                "ip_address": "127.0.0.1"
            },
            {
                "rtt": 0.006,
                "ip_address": "127.0.0.1"
            },
            {
                "rtt": 0.005,
                "ip_address": "127.0.0.1"
            },
            {
                "rtt": 0.007,
                "ip_address": "127.0.0.1"
            }
        ],
        "rtt_avg": 0.012,
        "rtt_max": 0.035,
        "probes_sent": 5
    }
}
$ napalm --user vagrant --password vagrant --vendor eos --optional_args "port=12443"␣
↪localhost call cli --method-kwargs "commands=['show  version']"
{
    "show  version": "Arista vEOS\nHardware version:    \nSerial number:
↪\nSystem MAC address:  0800.2761.b6ba\n\nSoftware image version: 4.15.2.
↪1F\nArchitecture:           i386\nInternal build version: 4.15.2.1F-2759621.
↪41521F\nInternal build ID:      8404cfa4-04c4-4008-838b-faf3f77ef6b8\n\nUptime:
↪           19 hours and 46 minutes\nTotal memory:           1897596 kB\nFree
↪memory:            117196 kB\n\n"
```

(continues on next page)

(continued from previous page)

```
}
```

### 3.5.1 Debug Mode

The debugging mode is also quite useful and it's recommended you use it to report and issue.:

```
$ napalm --debug --user vagrant --password vagrant --vendor eos --optional_args
↪"port=12443" localhost configure new_config.txt --strategy merge --dry-run
2017-08-15 15:14:23,527 - napalm - DEBUG - Starting napalm's debugging tool
2017-08-15 15:14:23,527 - napalm - DEBUG - Gathering napalm packages
2017-08-15 15:14:23,541 - napalm - DEBUG - napalm-ansible==0.7.0
2017-08-15 15:14:23,542 - napalm - DEBUG - napalm==2.0.0
2017-08-15 15:14:23,542 - napalm - DEBUG - get_network_driver - Calling with args: (
↪'eos',), {}
2017-08-15 15:14:23,551 - napalm - DEBUG - get_network_driver - Successful
2017-08-15 15:14:23,551 - napalm - DEBUG - __init__ - Calling with args: (<class
↪'napalm.eos.eos.EOSDriver'>, 'localhost', 'vagrant'), {'password': u'*******',
↪'optional_args': {u'port': 12443}, 'timeout': 60}
2017-08-15 15:14:23,551 - napalm - DEBUG - __init__ - Successful
2017-08-15 15:14:23,551 - napalm - DEBUG - pre_connection_tests - Calling with args: (
↪<napalm.eos.eos.EOSDriver object at 0x105d58bd0>,), {}
2017-08-15 15:14:23,551 - napalm - DEBUG - open - Calling with args: (<napalm.eos.eos.
↪EOSDriver object at 0x105d58bd0>,), {}
2017-08-15 15:14:23,586 - napalm - DEBUG - open - Successful
2017-08-15 15:14:23,586 - napalm - DEBUG - connection_tests - Calling with args: (
↪<napalm.eos.eos.EOSDriver object at 0x105d58bd0>,), {}
2017-08-15 15:14:23,587 - napalm - DEBUG - get_facts - Calling with args: (<napalm.
↪eos.eos.EOSDriver object at 0x105d58bd0>,), {}
2017-08-15 15:14:23,622 - napalm - DEBUG - Gathered facts:
{
    "os_version": "4.15.2.1F-2759627.41521F",
    "uptime": 71636,
    "interface_list": [
        "Ethernet1",
        "Ethernet2",
        "Management1"
    ],
    "vendor": "Arista",
    "serial_number": "",
    "model": "vEOS",
    "hostname": "myhost",
    "fqdn": "myhost"
}
{
    "os_version": "4.15.2.1F-2759627.41521F",
    "uptime": 71636,
    "interface_list": [
        "Ethernet1",
        "Ethernet2",
        "Management1"
    ],
    "vendor": "Arista",
    "serial_number": "",
    "model": "vEOS",
    "hostname": "myhost",
```

(continues on next page)

(continued from previous page)

```
    "fqdn": "myhost"
}
2017-08-15 15:14:23,622 - napalm - DEBUG - get_facts - Successful
2017-08-15 15:14:23,622 - napalm - DEBUG - load_merge_candidate - Calling with args: (
↪<napalm.eos.eos.EOSDriver object at 0x105d58bd0>,), {'filename': 'new_config.txt'}
2017-08-15 15:14:23,894 - napalm - ERROR - load_merge_candidate - Failed: Error␣
↪[1000]: CLI command 3 of 5 'hostname a_new-hostname' failed: could not run command␣
↪[ Host name is invalid. Host name must contain only alphanumeric characters, '.'␣
↪and '-'.
It must begin and end with an alphanumeric character.]

================ Traceback =================

Traceback (most recent call last):
  File "/Users/dbarroso/.virtualenvs/napalm/bin/napalm", line 11, in <module>
    load_entry_point('napalm', 'console_scripts', 'napalm')()
  File "/Users/dbarroso/workspace/napalm/napalm/napalm.base/clitools/cl_napalm.py",␣
↪line 285, in main
    run_tests(args)
  File "/Users/dbarroso/workspace/napalm/napalm/napalm.base/clitools/cl_napalm.py",␣
↪line 270, in run_tests
    configuration_change(device, args.config_file, args.strategy, args.dry_run)
  File "/Users/dbarroso/workspace/napalm/napalm/napalm.base/clitools/cl_napalm.py",␣
↪line 224, in configuration_change
    strategy_method(device, filename=config_file)
  File "/Users/dbarroso/workspace/napalm/napalm/napalm.base/clitools/cl_napalm.py",␣
↪line 27, in wrapper
    r = func(*args, **kwargs)
  File "/Users/dbarroso/workspace/napalm/napalm/napalm.base/clitools/cl_napalm.py",␣
↪line 202, in call_load_merge_candidate
    return device.load_merge_candidate(*args, **kwargs)
  File "/Users/dbarroso/workspace/napalm/napalm-eos/napalm.eos/eos.py", line 176, in␣
↪load_merge_candidate
    self._load_config(filename, config, False)
  File "/Users/dbarroso/workspace/napalm/napalm-eos/napalm.eos/eos.py", line 168, in _
↪load_config
    raise MergeConfigException(e.message)
napalm.base.exceptions.MergeConfigException: Error [1000]: CLI command 3 of 5
↪'hostname a_new-hostname' failed: could not run command [ Host name is invalid.␣
↪Host name must contain only alphanumeric characters, '.' and '-'.
It must begin and end with an alphanumeric character.]
```

## 3.6 NetworkDriver

**class** napalm.base.base.**NetworkDriver**(*hostname: str*, *username: str*, *password: str*, *timeout: int = 60*, *optional_args: Dict[KT, VT] = None*)

Bases: object

This is the base class you have to inherit from when writing your own Network Driver to manage any device. You will, in addition, have to override all the methods specified on this class. Make sure you follow the guidelines for every method and that you return the correct data.

**Parameters**

- **hostname** – IP or FQDN of the device you want to connect to.

- **username** – Username you want to use
- **password** – Password
- **timeout** – Time in seconds to wait for the device to respond.
- **optional_args** – Pass additional arguments to underlying driver

**Returns**

**cli**(*commands: List[str], encoding: str = 'text'*) → Dict[str, Union[str, Dict[str, Any]]]
Will execute a list of commands and return the output in a dictionary format.

Example:

```
{
    u'show version and haiku':  u'''Hostname: re0.edge01.arn01
                                    Model: mx480
                                    Junos: 13.3R6.5

                                           Help me, Obi-Wan
                                           I just saw Episode Two
                                           You're my only hope
                                ''',
    u'show chassis fan'     :   u'''
        Item                Status  RPM     Measurement
        Top Rear Fan        OK      3840    Spinning at intermediate-speed
        Bottom Rear Fan     OK      3840    Spinning at intermediate-speed
        Top Middle Fan      OK      3900    Spinning at intermediate-speed
        Bottom Middle Fan   OK      3840    Spinning at intermediate-speed
        Top Front Fan       OK      3810    Spinning at intermediate-speed
        Bottom Front Fan    OK      3840    Spinning at intermediate-speed'''
}
```

**close**() → None
Closes the connection to the device.

**commit_config**(*message: str = ''*, *revert_in: Optional[int] = None*) → None
Commits the changes requested by the method load_replace_candidate or load_merge_candidate.

NAPALM drivers that support 'commit confirm' should cause self.has_pending_commit to return True when a 'commit confirm' is in progress.

Implementations should raise an exception if commit_config is called multiple times while a 'commit confirm' is pending.

**Parameters**

- **message** (*str*) – Optional - configuration session commit message
- **revert_in** (*int | None*) – Optional - number of seconds before the configuration will be reverted

**compare_config**() → str

**Returns** A string showing the difference between the running configuration and the candidate configuration. The running_config is loaded automatically just before doing the comparison so there is no need for you to do it.

**compliance_report**(*validation_file: Optional[str] = None*, *validation_source: Optional[str] = None*) → napalm.base.models.ReportResult
Return a compliance report.

Verify that the device complies with the given validation file and writes a compliance report file. See https://napalm.readthedocs.io/en/latest/validate/index.html.

**Parameters**

- **validation_file** – Path to the file containing compliance definition. Default is None.
- **validation_source** – Dictionary containing compliance rules.

**Raises**

- **ValidationException** – File is not valid.
- **NotImplementedError** – Method not implemented.

**confirm_commit**() → None

Confirm the changes requested via commit_config when commit_confirm=True.

Should cause self.has_pending_commit to return False when done.

**connection_tests**() → None

This is a helper function used by the cli tool cl_napalm_show_tech. Drivers can override this method to do some tests, show information, enable debugging, etc. before a connection with the device has been successful.

**discard_config**() → None

Discards the configuration loaded into the candidate.

**get_arp_table**(*vrf: str = ''*) → List[napalm.base.models.ARPTableDict]

**Returns a list of dictionaries having the following set of keys:**

- interface (string)
- mac (string)
- ip (string)
- age (float)

'vrf' of null-string will default to all VRFs. Specific 'vrf' will return the ARP table entries for that VRFs (including potentially 'default' or 'global').

In all cases the same data structure is returned and no reference to the VRF that was used is included in the output.

Example:

```
[
    {
        'interface' : 'MgmtEth0/RSP0/CPU0/0',
        'mac'       : '5C:5E:AB:DA:3C:F0',
        'ip'        : '172.17.17.1',
        'age'       : 1454496274.84
    },
    {
        'interface' : 'MgmtEth0/RSP0/CPU0/0',
        'mac'       : '5C:5E:AB:DA:3C:FF',
        'ip'        : '172.17.17.2',
        'age'       : 1435641582.49
    }
]
```

**get_bgp_config**(*group: str = ''*, *neighbor: str = ''*) → napalm.base.models.BPGConfigGroupDict

Returns a dictionary containing the BGP configuration. Can return either the whole config, either the config only for a group or neighbor.

**Parameters**

- **group** – Returns the configuration of a specific BGP group.
- **neighbor** – Returns the configuration of a specific BGP neighbor.

Main dictionary keys represent the group name and the values represent a dictionary having the keys below. A default group named "_" will contain information regarding global settings and any neighbors that are not members of a group.

- type (string)
- description (string)
- apply_groups (string list)
- multihop_ttl (int)
- multipath (True/False)
- local_address (string)
- local_as (int)
- remote_as (int)
- import_policy (string)
- export_policy (string)
- remove_private_as (True/False)
- prefix_limit (dictionary)
- neighbors (dictionary)

Neighbors is a dictionary of dictionaries with the following keys:

- description (string)
- import_policy (string)
- export_policy (string)
- local_address (string)
- local_as (int)
- remote_as (int)
- authentication_key (string)
- prefix_limit (dictionary)
- route_reflector_client (True/False)
- nhs (True/False)

The inner dictionary prefix_limit has the same structure for both layers:

```
{
    [FAMILY_NAME]: {
        [FAMILY_TYPE]: {
            'limit': [LIMIT],
```

(continues on next page)

(continued from previous page)

```
            ... other options
        }
    }
}
```

Example:

```
{
    'PEERS-GROUP-NAME':{
        'type'              : u'external',
        'description'       : u'Here we should have a nice description',
        'apply_groups'      : [u'BGP-PREFIX-LIMIT'],
        'import_policy'     : u'PUBLIC-PEER-IN',
        'export_policy'     : u'PUBLIC-PEER-OUT',
        'remove_private_as' : True,
        'multipath'         : True,
        'multihop_ttl'      : 30,
        'neighbors'         : {
            '192.168.0.1': {
                'description'   : 'Facebook [CDN]',
                'prefix_limit'  : {
                    'inet': {
                        'unicast': {
                            'limit': 100,
                            'teardown': {
                                'threshold' : 95,
                                'timeout'   : 5
                            }
                        }
                    }
                }
                'remote_as'             : 32934,
                'route_reflector_client': False,
                'nhs'                   : True
            },
            '172.17.17.1': {
                'description'   : 'Twitter [CDN]',
                'prefix_limit'  : {
                    'inet': {
                        'unicast': {
                            'limit': 500,
                            'no-validate': 'IMPORT-FLOW-ROUTES'
                        }
                    }
                }
                'remote_as'              : 13414
                'route_reflector_client': False,
                'nhs'                   : False
            }
        }
    }
}
```

**get_bgp_neighbors**() → Dict[str, napalm.base.models.BGPStateNeighborsPerVRFDict]

Returns a dictionary of dictionaries. The keys for the first dictionary will be the vrf (global if no vrf). The inner dictionary will contain the following data for each vrf:

- router_id
- **peers - another dictionary of dictionaries. Outer keys are the IPs of the neighbors. The inner keys are:**
  - local_as (int)
  - remote_as (int)
  - remote_id - peer router id
  - is_up (True/False)
  - is_enabled (True/False)
  - description (string)
  - uptime (int in seconds)
  - **address_family (dictionary) - A dictionary of address families available for the neighbor. So far it can be '**
    - received_prefixes (int)
    - accepted_prefixes (int)
    - sent_prefixes (int)

Note, if is_up is False and uptime has a positive value then this indicates the uptime of the last active BGP session.

Example:

```
{
  "global": {
    "router_id": "10.0.1.1",
    "peers": {
      "10.0.0.2": {
        "local_as": 65000,
        "remote_as": 65000,
        "remote_id": "10.0.1.2",
        "is_up": True,
        "is_enabled": True,
        "description": "internal-2",
        "uptime": 4838400,
        "address_family": {
          "ipv4": {
            "sent_prefixes": 637213,
            "accepted_prefixes": 3142,
            "received_prefixes": 3142
          },
          "ipv6": {
            "sent_prefixes": 36714,
            "accepted_prefixes": 148,
            "received_prefixes": 148
          }
        }
      }
    }
  }
}
```

**get_bgp_neighbors_detail**(*neighbor_address: str = ''*) → Dict[str, napalm.base.models.PeerDetailsDict]

Returns a detailed view of the BGP neighbors as a dictionary of lists.

**Parameters** **neighbor_address** – Retuns the statistics for a spcific BGP neighbor.

Returns a dictionary of dictionaries. The keys for the first dictionary will be the vrf (global if no vrf). The keys of the inner dictionary represent the AS number of the neighbors. Leaf dictionaries contain the following fields:

- up (True/False)
- local_as (int)
- remote_as (int)
- router_id (string)
- local_address (string)
- routing_table (string)
- local_address_configured (True/False)
- local_port (int)
- remote_address (string)
- remote_port (int)
- multihop (True/False)
- multipath (True/False)
- remove_private_as (True/False)
- import_policy (string)
- export_policy (string)
- input_messages (int)
- output_messages (int)
- input_updates (int)
- output_updates (int)
- messages_queued_out (int)
- connection_state (string)
- previous_connection_state (string)
- last_event (string)
- suppress_4byte_as (True/False)
- local_as_prepend (True/False)
- holdtime (int)
- configured_holdtime (int)
- keepalive (int)
- configured_keepalive (int)
- active_prefix_count (int)
- received_prefix_count (int)

- accepted_prefix_count (int)
- suppressed_prefix_count (int)
- advertised_prefix_count (int)
- flap_count (int)

Example:

```
{
    'global': {
        8121: [
            {
                'up'                          : True,
                'local_as'                    : 13335,
                'remote_as'                   : 8121,
                'local_address'               : u'172.101.76.1',
                'local_address_configured'    : True,
                'local_port'                  : 179,
                'routing_table'               : u'inet.0',
                'remote_address'              : u'192.247.78.0',
                'remote_port'                 : 58380,
                'multihop'                    : False,
                'multipath'                   : True,
                'remove_private_as'           : True,
                'import_policy'               : u'4-NTT-TRANSIT-IN',
                'export_policy'               : u'4-NTT-TRANSIT-OUT',
                'input_messages'              : 123,
                'output_messages'             : 13,
                'input_updates'               : 123,
                'output_updates'              : 5,
                'messages_queued_out'         : 23,
                'connection_state'            : u'Established',
                'previous_connection_state'   : u'EstabSync',
                'last_event'                  : u'RecvKeepAlive',
                'suppress_4byte_as'           : False,
                'local_as_prepend'            : False,
                'holdtime'                    : 90,
                'configured_holdtime'         : 90,
                'keepalive'                   : 30,
                'configured_keepalive'        : 30,
                'active_prefix_count'         : 132808,
                'received_prefix_count'       : 566739,
                'accepted_prefix_count'       : 566479,
                'suppressed_prefix_count'     : 0,
                'advertised_prefix_count'     : 0,
                'flap_count'                  : 27
            }
        ]
    }
}
```

**get_config**(*retrieve: str = 'all', full: bool = False, sanitized: bool = False*) → napalm.base.models.ConfigDict

Return the configuration of a device.

**Parameters**

- **retrieve** (*string*) – Which configuration type you want to populate, default is all of them. The rest will be set to "".

- **full** (*bool*) – Retrieve all the configuration. For instance, on ios, "sh run all".
- **sanitized** (*bool*) – Remove secret data. Default: `False`.

**Returns**

- running(string) - Representation of the native running configuration
- candidate(string) - Representation of the native candidate configuration. If the device doesnt differentiate between running and startup configuration this will an empty string
- startup(string) - Representation of the native startup configuration. If the device doesnt differentiate between running and startup configuration this will an empty string

**Return type** The object returned is a dictionary with a key for each configuration store

**get_environment**() → napalm.base.models.EnvironmentDict

Returns a dictionary where:

- **fans is a dictionary of dictionaries where the key is the location and the values:**
  - status (True/False) - True if it's ok, false if it's broken
- **temperature is a dict of dictionaries where the key is the location and the values:**
  - temperature (float) - Temperature in celsius the sensor is reporting.
  - is_alert (True/False) - True if the temperature is above the alert threshold
  - is_critical (True/False) - True if the temp is above the critical threshold
- **power is a dictionary of dictionaries where the key is the PSU id and the values:**
  - status (True/False) - True if it's ok, false if it's broken
  - capacity (float) - Capacity in W that the power supply can support
  - output (float) - Watts drawn by the system
- **cpu is a dictionary of dictionaries where the key is the ID and the values**
  - %usage
- **memory is a dictionary with:**
  - available_ram (int) - Total amount of RAM installed in the device
  - used_ram (int) - RAM in use in the device

**get_facts**() → napalm.base.models.FactsDict

**Returns a dictionary containing the following information:**

- uptime - Uptime of the device in seconds.
- vendor - Manufacturer of the device.
- model - Device model.
- hostname - Hostname of the device
- fqdn - Fqdn of the device
- os_version - String with the OS version running on the device.
- serial_number - Serial number of the device
- interface_list - List of the interfaces of the device

Example:

```
{
'uptime': 151005.57332897186,
'vendor': u'Arista',
'os_version': u'4.14.3-2329074.gaatlantarel',
'serial_number': u'SN0123A34AS',
'model': u'vEOS',
'hostname': u'eos-router',
'fqdn': u'eos-router',
'interface_list': [u'Ethernet2', u'Management1', u'Ethernet1', u'Ethernet3']
}
```

**get_firewall_policies**() → Dict[str, List[napalm.base.models.FirewallPolicyDict]]

Returns a dictionary of lists of dictionaries where the first key is an unique policy name and the inner dictionary contains the following keys:

- position (int)
- packet_hits (int)
- byte_hits (int)
- id (text_type)
- enabled (bool)
- schedule (text_type)
- log (text_type)
- l3_src (text_type)
- l3_dst (text_type)
- service (text_type)
- src_zone (text_type)
- dst_zone (text_type)
- action (text_type)

Example:

```
{
    'policy_name': [{
        'position': 1,
        'packet_hits': 200,
        'byte_hits': 83883,
        'id': '230',
        'enabled': True,
        'schedule': 'Always',
        'log': 'all',
        'l3_src': 'any',
        'l3_dst': 'any',
        'service': 'HTTP',
        'src_zone': 'port2',
        'dst_zone': 'port3',
        'action': 'Permit'
    }]
}
```

**get_interfaces**() → Dict[str, napalm.base.models.InterfaceDict]

Returns a dictionary of dictionaries. The keys for the first dictionary will be the interfaces in the devices. The inner dictionary will containing the following data for each interface:

- is_up (True/False)
- is_enabled (True/False)
- description (string)
- last_flapped (float in seconds)
- speed (float in Mbit)
- MTU (in Bytes)
- mac_address (string)

Example:

```
{
u'Management1':
    {
    'is_up': False,
    'is_enabled': False,
    'description': '',
    'last_flapped': -1.0,
    'speed': 1000.0,
    'mtu': 1500,
    'mac_address': 'FA:16:3E:57:33:61',
    },
u'Ethernet1':
    {
    'is_up': True,
    'is_enabled': True,
    'description': 'foo',
    'last_flapped': 1429978575.1554043,
    'speed': 1000.0,
    'mtu': 1500,
    'mac_address': 'FA:16:3E:57:33:62',
    },
u'Ethernet2':
    {
    'is_up': True,
    'is_enabled': True,
    'description': 'bla',
    'last_flapped': 1429978575.1555667,
    'speed': 1000.0,
    'mtu': 1500,
    'mac_address': 'FA:16:3E:57:33:63',
    },
u'Ethernet3':
    {
    'is_up': False,
    'is_enabled': True,
    'description': 'bar',
    'last_flapped': -1.0,
    'speed': 1000.0,
    'mtu': 1500,
    'mac_address': 'FA:16:3E:57:33:64',
```

(continues on next page)

(continued from previous page)

```
    }
}
```

**get_interfaces_counters**() → Dict[str, napalm.base.models.InterfaceCounterDict]

Returns a dictionary of dictionaries where the first key is an interface name and the inner dictionary contains the following keys:

- tx_errors (int)
- rx_errors (int)
- tx_discards (int)
- rx_discards (int)
- tx_octets (int)
- rx_octets (int)
- tx_unicast_packets (int)
- rx_unicast_packets (int)
- tx_multicast_packets (int)
- rx_multicast_packets (int)
- tx_broadcast_packets (int)
- rx_broadcast_packets (int)

Example:

```
{
    u'Ethernet2': {
        'tx_multicast_packets': 699,
        'tx_discards': 0,
        'tx_octets': 88577,
        'tx_errors': 0,
        'rx_octets': 0,
        'tx_unicast_packets': 0,
        'rx_errors': 0,
        'tx_broadcast_packets': 0,
        'rx_multicast_packets': 0,
        'rx_broadcast_packets': 0,
        'rx_discards': 0,
        'rx_unicast_packets': 0
    },
    u'Management1': {
         'tx_multicast_packets': 0,
         'tx_discards': 0,
         'tx_octets': 159159,
         'tx_errors': 0,
         'rx_octets': 167644,
         'tx_unicast_packets': 1241,
         'rx_errors': 0,
         'tx_broadcast_packets': 0,
         'rx_multicast_packets': 0,
         'rx_broadcast_packets': 80,
         'rx_discards': 0,
         'rx_unicast_packets': 0
```

(continues on next page)

(continued from previous page)

```
    },
    u'Ethernet1': {
         'tx_multicast_packets': 293,
         'tx_discards': 0,
         'tx_octets': 38639,
         'tx_errors': 0,
         'rx_octets': 0,
         'tx_unicast_packets': 0,
         'rx_errors': 0,
         'tx_broadcast_packets': 0,
         'rx_multicast_packets': 0,
         'rx_broadcast_packets': 0,
         'rx_discards': 0,
         'rx_unicast_packets': 0
    }
}
```

**get_interfaces_ip**() → Dict[str, napalm.base.models.InterfacesIPDict]

Returns all configured IP addresses on all interfaces as a dictionary of dictionaries. Keys of the main dictionary represent the name of the interface. Values of the main dictionary represent are dictionaries that may consist of two keys 'ipv4' and 'ipv6' (one, both or none) which are themselves dictionaries with the IP addresses as keys. Each IP Address dictionary has the following keys:

- prefix_length (int)

Example:

```
{
    u'FastEthernet8': {
        u'ipv4': {
            u'10.66.43.169': {
                'prefix_length': 22
            }
        }
    },
    u'Loopback555': {
        u'ipv4': {
            u'192.168.1.1': {
                'prefix_length': 24
            }
        },
        u'ipv6': {
            u'1::1': {
                'prefix_length': 64
            },
            u'2001:DB8:1::1': {
                'prefix_length': 64
            },
            u'2::': {
                'prefix_length': 64
            },
            u'FE80::3': {
                'prefix_length': u'N/A'
            }
        }
    },
    u'Tunnel0': {
```

(continues on next page)

(continued from previous page)

```
        u'ipv4': {
            u'10.63.100.9': {
                'prefix_length': 24
            }
        }
    }
}
```

**get_ipv6_neighbors_table**() → List[napalm.base.models.IPV6NeighborDict]

Get IPv6 neighbors table information.

Return a list of dictionaries having the following set of keys:

- interface (string)
- mac (string)
- ip (string)
- age (float) in seconds
- state (string)

For example:

```
[
    {
        'interface' : 'MgmtEth0/RSP0/CPU0/0',
        'mac'       : '5c:5e:ab:da:3c:f0',
        'ip'        : '2001:db8:1:1::1',
        'age'       : 1454496274.84,
        'state'     : 'REACH'
    },
    {
        'interface': 'MgmtEth0/RSP0/CPU0/0',
        'mac'       : '66:0e:94:96:e0:ff',
        'ip'        : '2001:db8:1:1::2',
        'age'       : 1435641582.49,
        'state'     : 'STALE'
    }
]
```

**get_lldp_neighbors**() → Dict[str, List[napalm.base.models.LLDPNeighborDict]]

**Returns a dictionary where the keys are local ports and the value is a list of dictionaries with the following informati**

- hostname
- port

Example:

```
{
u'Ethernet2':
    [
        {
        'hostname': u'junos-unittest',
        'port': u'520',
        }
```

(continues on next page)

(continued from previous page)

```
    ],
u'Ethernet3':
    [
        {
        'hostname': u'junos-unittest',
        'port': u'522',
        }
    ],
u'Ethernet1':
    [
        {
        'hostname': u'junos-unittest',
        'port': u'519',
        },
        {
        'hostname': u'ios-xrv-unittest',
        'port': u'Gi0/0/0/0',
        }
    ],
u'Management1':
    [
        {
        'hostname': u'junos-unittest',
        'port': u'508',
        }
    ]
}
```

**get_lldp_neighbors_detail**(*interface: str = ''*) → Dict[str, List[napalm.base.models.LLDPNeighborDetailDict]]

Returns a detailed view of the LLDP neighbors as a dictionary containing lists of dictionaries for each interface.

Empty entries are returned as an empty string (e.g. '') or list where applicable.

Inner dictionaries contain fields:

- parent_interface (string)
- remote_port (string)
- remote_port_description (string)
- remote_chassis_id (string)
- remote_system_name (string)
- remote_system_description (string)
- **remote_system_capab (list) with any of these values**
  - other
  - repeater
  - bridge
  - wlan-access-point
  - router
  - telephone

        - docsis-cable-device
        - station
    - remote_system_enabled_capab (list)

Example:

```
{
    'TenGigE0/0/0/8': [
        {
            'parent_interface': u'Bundle-Ether8',
            'remote_chassis_id': u'8c60.4f69.e96c',
            'remote_system_name': u'switch',
            'remote_port': u'Eth2/2/1',
            'remote_port_description': u'Ethernet2/2/1',
            'remote_system_description': u'''Cisco Nexus Operating System␣
↪(NX-OS)
                 Software 7.1(0)N1(1a)
                 TAC support: http://www.cisco.com/tac
                 Copyright (c) 2002-2015, Cisco Systems, Inc. All rights␣
↪reserved.''',
            'remote_system_capab': ['bridge', 'repeater'],
            'remote_system_enable_capab': ['bridge']
        }
    ]
}
```

**get_mac_address_table**() → List[napalm.base.models.MACAdressTable]

Returns a lists of dictionaries. Each dictionary represents an entry in the MAC Address Table, having the following keys:

- mac (string)
- interface (string)
- vlan (int)
- active (boolean)
- static (boolean)
- moves (int)
- last_move (float)

However, please note that not all vendors provide all these details. E.g.: field last_move is not available on JUNOS devices etc.

Example:

```
[
    {
        'mac'       : '00:1C:58:29:4A:71',
        'interface' : 'Ethernet47',
        'vlan'      : 100,
        'static'    : False,
        'active'    : True,
        'moves'     : 1,
        'last_move' : 1454417742.58
    },
    {
```

(continues on next page)

(continued from previous page)

```
        'mac'       : '00:1C:58:29:4A:C1',
        'interface' : 'xe-1/0/1',
        'vlan'       : 100,
        'static'    : False,
        'active'    : True,
        'moves'     : 2,
        'last_move' : 1453191948.11
    },
    {
        'mac'       : '00:1C:58:29:4A:C2',
        'interface' : 'ae7.900',
        'vlan'      : 900,
        'static'    : False,
        'active'    : True,
        'moves'     : None,
        'last_move' : None
    }
]
```

**get_network_instances**(*name: str = ''*) → Dict[str, napalm.base.models.NetworkInstanceDict]

Return a dictionary of network instances (VRFs) configured, including default/global

**Parameters** **name** (*string*) –

**Returns**

- **name (dict)**
  - name (unicode)
  - type (unicode)
  - **state (dict)**
    - route_distinguisher (unicode)
  - **interfaces (dict)**
    - **interface (dict)**
      - interface name: (dict)

**Return type** A dictionary of network instances in OC format

Example:

```
{
    u'MGMT': {
        u'name': u'MGMT',
        u'type': u'L3VRF',
        u'state': {
            u'route_distinguisher': u'123:456',
        },
        u'interfaces': {
            u'interface': {
                u'Management1': {}
            }
        }
    },
    u'default': {
        u'name': u'default',
```

(continues on next page)

(continued from previous page)

```
        u'type': u'DEFAULT_INSTANCE',
        u'state': {
            u'route_distinguisher': None,
        },
        u'interfaces: {
            u'interface': {
                u'Ethernet1': {}
                u'Ethernet2': {}
                u'Ethernet3': {}
                u'Ethernet4': {}
            }
        }
    }
}
```

**get_ntp_peers**() → Dict[str, napalm.base.models.NTPPeerDict]

Returns the NTP peers configuration as dictionary. The keys of the dictionary represent the IP Addresses of the peers. Inner dictionaries do not have yet any available keys.

Example:

```
{
    '192.168.0.1': {},
    '17.72.148.53': {},
    '37.187.56.220': {},
    '162.158.20.18': {}
}
```

**get_ntp_servers**() → Dict[str, napalm.base.models.NTPServerDict]

Returns the NTP servers configuration as dictionary. The keys of the dictionary represent the IP Addresses of the servers. Inner dictionaries do not have yet any available keys.

Example:

```
{
    '192.168.0.1': {},
    '17.72.148.53': {},
    '37.187.56.220': {},
    '162.158.20.18': {}
}
```

**get_ntp_stats**() → List[napalm.base.models.NTPStats]

Returns a list of NTP synchronization statistics.

- remote (string)
- referenceid (string)
- synchronized (True/False)
- stratum (int)
- type (string)
- when (string)
- hostpoll (int)
- reachability (int)
- delay (float)
- offset (float)
- jitter (float)

Example:

```
[
    {
        'remote'        : u'188.114.101.4',
        'referenceid'   : u'188.114.100.1',
        'synchronized'  : True,
        'stratum'       : 4,
        'type'          : u'-',
        'when'          : u'107',
        'hostpoll'      : 256,
        'reachability'  : 377,
        'delay'         : 164.228,
        'offset'        : -13.866,
        'jitter'        : 2.695
    }
]
```

**get_optics**() → Dict[str, napalm.base.models.OpticsDict]

Fetches the power usage on the various transceivers installed on the switch (in dbm), and returns a view that conforms with the openconfig model openconfig-platform-transceiver.yang

Returns a dictionary where the keys are as listed below:

- **intf_name (unicode)**
  - **physical_channels**
    - **channels (list of dicts)**
      - index (int)
      - **state**

        **input_power**

        instant (float)

        avg (float)

        min (float)

        max (float)

        **output_power**

        instant (float)

        avg (float)

        min (float)

        max (float)

        **laser_bias_current**

        instant (float)

        avg (float)

        min (float)

        max (float)

Example:

```
{
    'et1': {
        'physical_channels': {
```

(continues on next page)

(continued from previous page)

```
            'channel': [
                {
                    'index': 0,
                    'state': {
                        'input_power': {
                            'instant': 0.0,
                            'avg': 0.0,
                            'min': 0.0,
                            'max': 0.0,
                        },
                        'output_power': {
                            'instant': 0.0,
                            'avg': 0.0,
                            'min': 0.0,
                            'max': 0.0,
                        },
                        'laser_bias_current': {
                            'instant': 0.0,
                            'avg': 0.0,
                            'min': 0.0,
                            'max': 0.0,
                        },
                    }
                }
            ]
        }
    }
}
```

**get_probes_config**() → Dict[str, napalm.base.models.ProbeTestDict]

Returns a dictionary with the probes configured on the device. Probes can be either RPM on JunOS devices, either SLA on IOS-XR. Other vendors do not support probes. The keys of the main dictionary represent the name of the probes. Each probe consists on multiple tests, each test name being a key in the probe dictionary. A test has the following keys:

- probe_type (str)
- target (str)
- source (str)
- probe_count (int)
- test_interval (int)

Example:

```
{
    'probe1':{
        'test1': {
            'probe_type'   : 'icmp-ping',
            'target'       : '192.168.0.1',
            'source'       : '192.168.0.2',
            'probe_count'  : 13,
            'test_interval': 3
        },
        'test2': {
            'probe_type'   : 'http-ping',
            'target'       : '172.17.17.1',
            'source'       : '192.17.17.2',
            'probe_count'  : 5,
            'test_interval': 60
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

**get_probes_results**() → Dict[str, napalm.base.models.ProbeTestResultDict]

Returns a dictionary with the results of the probes. The keys of the main dictionary represent the name of the probes. Each probe consists on multiple tests, each test name being a key in the probe dictionary. A test has the following keys:

- target (str)
- source (str)
- probe_type (str)
- probe_count (int)
- rtt (float)
- round_trip_jitter (float)
- current_test_loss (float)
- current_test_min_delay (float)
- current_test_max_delay (float)
- current_test_avg_delay (float)
- last_test_min_delay (float)
- last_test_max_delay (float)
- last_test_avg_delay (float)
- global_test_min_delay (float)
- global_test_max_delay (float)
- global_test_avg_delay (float)

Example:

```
{
    'probe1':  {
        'test1': {
            'last_test_min_delay'    : 63.120,
            'global_test_min_delay' : 62.912,
            'current_test_avg_delay': 63.190,
            'global_test_max_delay' : 177.349,
            'current_test_max_delay': 63.302,
            'global_test_avg_delay' : 63.802,
            'last_test_avg_delay'    : 63.438,
            'last_test_max_delay'    : 65.356,
            'probe_type'             : 'icmp-ping',
            'rtt'                    : 63.138,
            'current_test_loss'      : 0,
            'round_trip_jitter'      : -59.0,
            'target'                 : '192.168.0.1',
            'source'                 : '192.168.0.2'
            'probe_count'            : 15,
            'current_test_min_delay': 63.138
        },
        'test2': {
            'last_test_min_delay'    : 176.384,
            'global_test_min_delay' : 169.226,
            'current_test_avg_delay': 177.098,
            'global_test_max_delay' : 292.628,
            'current_test_max_delay': 180.055,
            'global_test_avg_delay' : 177.959,
            'last_test_avg_delay'    : 177.178,
            'last_test_max_delay'    : 184.671,
```

(continues on next page)

(continued from previous page)

```
            'probe_type'            : 'icmp-ping',
            'rtt'                   : 176.449,
            'current_test_loss'     : 0,
            'round_trip_jitter'     : -34.0,
            'target'                : '172.17.17.1',
            'source'                : '172.17.17.2'
            'probe_count'           : 15,
            'current_test_min_delay': 176.402
        }
    }
}
```

**get_route_to**(*destination: str = ''*, *protocol: str = ''*, *longer: bool = False*) → Dict[str, napalm.base.models.RouteDict]

Returns a dictionary of dictionaries containing details of all available routes to a destination.

**Parameters**

- **destination** – The destination prefix to be used when filtering the routes.
- **protocol** (*optional*) – Retrieve the routes only for a specific protocol.
- **longer** (*optional*) – Retrieve more specific routes as well.

Each inner dictionary contains the following fields:

- protocol (string)
- current_active (True/False)
- last_active (True/False)
- age (int)
- next_hop (string)
- outgoing_interface (string)
- selected_next_hop (True/False)
- preference (int)
- inactive_reason (string)
- routing_table (string)
- protocol_attributes (dictionary)

protocol_attributes is a dictionary with protocol-specific information, as follows:

- **BGP**
  - local_as (int)
  - remote_as (int)
  - peer_id (string)
  - as_path (string)
  - communities (list)
  - local_preference (int)
  - preference2 (int)
  - metric (int)
  - metric2 (int)
- **ISIS:**
  - level (int)

Example:

```
{
    "1.0.0.0/24": [
        {
            "protocol"          : u"BGP",
            "inactive_reason"   : u"Local Preference",
            "last_active"       : False,
            "age"               : 105219,
            "next_hop"          : u"172.17.17.17",
```

(continues on next page)

(continued from previous page)

```
            "selected_next_hop" : True,
            "preference"        : 170,
            "current_active"    : False,
            "outgoing_interface": u"ae9.0",
            "routing_table"     : "inet.0",
            "protocol_attributes": {
                "local_as"          : 13335,
                "as_path"           : u"2914 8403 54113 I",
                "communities"       : [
                    u"2914:1234",
                    u"2914:5678",
                    u"8403:1717",
                    u"54113:9999"
                ],
                "preference2"       : -101,
                "remote_as"         : 2914,
                "local_preference"  : 100
            }
        }
    ]
}
```

**get_snmp_information**() → napalm.base.models.SNMPDict

Returns a dict of dicts containing SNMP configuration. Each inner dictionary contains these fields

- chassis_id (string)
- community (dictionary)
- contact (string)
- location (string)

'community' is a dictionary with community string specific information, as follows:

- acl (string) # acl number or name
- mode (string) # read-write (rw), read-only (ro)

Example:

```
{
    'chassis_id': u'Asset Tag 54670',
    'community': {
        u'private': {
            'acl': u'12',
            'mode': u'rw'
        },
        u'public': {
            'acl': u'11',
            'mode': u'ro'
        },
        u'public_named_acl': {
            'acl': u'ALLOW-SNMP-ACL',
            'mode': u'ro'
        },
        u'public_no_acl': {
            'acl': u'N/A',
            'mode': u'ro'
        }
    },
    'contact' : u'Joe Smith',
    'location': u'123 Anytown USA Rack 404'
}
```

**get_users**() → Dict[str, napalm.base.models.UsersDict]

Returns a dictionary with the configured users. The keys of the main dictionary represents the username. The values represent the details of the user, represented by the following keys:

- level (int)
- password (str)
- sshkeys (list)

The level is an integer between 0 and 15, where 0 is the lowest access and 15 represents full access to the device.

Example:

```
{
    'mircea': {
        'level': 15,
        'password': '$1$0P70xKPa$z46fewjo/10cBTckk6I/w/',
        'sshkeys': [
            'ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAADAQABAAABAQC4pFn+shPwTb2yELO4L7NtQrKOJXNeCl1je␣
↪␣
↪l9STXVaGnRAnuc2PXl35vnWmcUq6YbUEcgUTRzzXfmelJKuVJTJIlMXii7h2xkbQp0YZIEs4P␣
↪8ipwnRBAxFfk/ZcDsN3mjep4/
↪yjN56eorF5xs7zP9HbqbJ1dsqk1p3A/9LIL7l6YewLBCwJj6␣
↪D+fWSJ0/YW+7oH17Fk2HH+tw0L5PcWLHkwA4t60iXn16qDbIk/ze6jv2hDGdCdz7oYQeCE55C␣
↪CHOHMJWYfN3jcL4s0qv8/u6Ka1FVkV7iMmro7ChThoV/
↪5snI4Ljf2wKqgHH7TfNaCfpU0WvHA nTs8zhOrGScSrtb␣
↪mircea@master-roshi'
        ]
    }
}
```

**get_vlans**() → Dict[str, napalm.base.models.VlanDict]

**Return structure being spit balled is as follows.**

- **vlan_id (int)**
  - name (text_type)
  - interfaces (list)

Example:

```
{
    1: {
        "name": "default",
        "interfaces": ["GigabitEthernet0/0/1", "GigabitEthernet0/0/2"]
    },
    2: {
        "name": "vlan2",
        "interfaces": []
    }
}
```

**has_pending_commit**() → bool

:return Boolean indicating if a commit_config that needs confirmed is in process.

**is_alive**() → napalm.base.models.AliveDict

Returns a flag with the connection state. Depends on the nature of API used by each driver. The state does not reflect only on the connection status (when SSH), it must also take into consideration other parameters, e.g.: NETCONF session might not be usable, althought the underlying SSH session is still open etc.

**load_merge_candidate**(*filename: Optional[str] = None*, *config: Optional[str] = None*) → None

Populates the candidate 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.

If you use this method the existing configuration will be merged with the candidate configuration once you commit the changes. This method will not change the configuration by itself.

**Parameters**

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

**Raises** **MergeConfigException** – If there is an error on the configuration sent.

**load_replace_candidate**(*filename: Optional[str] = None*, *config: Optional[str] = None*) → None

Populates the candidate 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.

If you use this method the existing configuration will be replaced entirely by the candidate configuration once you commit the changes. This method will not change the configuration by itself.

**Parameters**

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

**Raises** **ReplaceConfigException** – If there is an error on the configuration sent.

**load_template**(*template_name: str*, *template_source: Optional[str] = None*, *template_path: Optional[str] = None*, ***template_vars*) → None

Will load a templated configuration on the device.

**Parameters**

- **cls** – Instance of the driver class.
- **template_name** – Identifies the template name.
- **template_source** (*optional*) – Custom config template rendered and loaded on device
- **template_path** (*optional*) – Absolute path to directory for the configuration templates
- **template_vars** – Dictionary with arguments to be used when the template is rendered.

**Raises**

- **DriverTemplateNotImplemented** – No template defined for the device type.
- **TemplateNotImplemented** – The template specified in template_name does not exist in the default path or in the custom path if any specified using parameter *template_path*.
- **TemplateRenderException** – The template could not be rendered. Either the template source does not have the right format, either the arguments in *template_vars* are not properly specified.

**open**() → None

Opens a connection to the device.

**ping**(*destination: str*, *source: str = ''*, *ttl: int = 255*, *timeout: int = 2*, *size: int = 100*, *count: int = 5*, *vrf: str = ''*, *source_interface: str = ''*) → napalm.base.models.PingResultDict

Executes ping on the device and returns a dictionary with the result

**Parameters**

- **destination** – Host or IP Address of the destination
- **source** (*optional*) – Source address of echo request
- **ttl** (*optional*) – Maximum number of hops
- **timeout** (*optional*) – Maximum seconds to wait after sending final packet
- **size** (*optional*) – Size of request (bytes)
- **count** (*optional*) – Number of ping request to send

- **vrf** (*optional*) – Use a specific VRF to execute the ping
- **source_interface** (*optional*) – Use an IP from a source interface as source address of echo request

Output dictionary has one of following keys:

- success
- error

In case of success, inner dictionary will have the following keys:

- probes_sent (int)
- packet_loss (int)
- rtt_min (float)
- rtt_max (float)
- rtt_avg (float)
- rtt_stddev (float)
- results (list)

'results' is a list of dictionaries with the following keys:

- ip_address (str)
- rtt (float)

Example:

```
{
    'success': {
        'probes_sent': 5,
        'packet_loss': 0,
        'rtt_min': 72.158,
        'rtt_max': 72.433,
        'rtt_avg': 72.268,
        'rtt_stddev': 0.094,
        'results': [
            {
                'ip_address': u'1.1.1.1',
                'rtt': 72.248
            },
            {
                'ip_address': '2.2.2.2',
                'rtt': 72.299
            }
        ]
    }
}

OR

{
    'error': 'unknown host 8.8.8.8'
}
```

**post_connection_tests**() → None

This is a helper function used by the cli tool cl_napalm_show_tech. Drivers can override this method to do some tests, show information, enable debugging, etc. after a connection with the device has been closed successfully.

**pre_connection_tests**() → None

This is a helper function used by the cli tool cl_napalm_show_tech. Drivers can override this method to do some tests, show information, enable debugging, etc. before a connection with the device is attempted.

**rollback**() → None

If changes were made, revert changes to the original state.

If commit confirm is in process, rollback changes and clear has_pending_commit.

**traceroute**(*destination: str*, *source: str = ''*, *ttl: int = 255*, *timeout: int = 2*, *vrf: str = ''*) → napalm.base.models.TracerouteResultDict

Executes traceroute on the device and returns a dictionary with the result.

**Parameters**

- **destination** – Host or IP Address of the destination
- **source** (*optional*) – Use a specific IP Address to execute the traceroute
- **ttl** (*optional*) – Maximum number of hops
- **timeout** (*optional*) – Number of seconds to wait for response
- **vrf** (*optional*) – Use a specific VRF to execute the traceroute

Output dictionary has one of the following keys:

- success
- error

In case of success, the keys of the dictionary represent the hop ID, while values are dictionaries containing the probes results:

- rtt (float)
- ip_address (str)
- host_name (str)

Example:

```
{
    'success': {
        1: {
            'probes': {
                1: {
                    'rtt': 1.123,
                    'ip_address': u'206.223.116.21',
                    'host_name': u'eqixsj-google-gige.google.com'
                },
                2: {
                    'rtt': 1.9100000000000001,
                    'ip_address': u'206.223.116.21',
                    'host_name': u'eqixsj-google-gige.google.com'
                },
                3: {
                    'rtt': 3.347,
                    'ip_address': u'198.32.176.31',
                    'host_name': u'core2-1-1-0.pao.net.google.com'}
                }
            },
            2: {
                'probes': {
                    1: {
                        'rtt': 1.586,
                        'ip_address': u'209.85.241.171',
                        'host_name': u'209.85.241.171'
                        },
                    2: {
                        'rtt': 1.6300000000000001,
                        'ip_address': u'209.85.241.171',
                        'host_name': u'209.85.241.171'
                    },
                    3: {
                        'rtt': 1.6480000000000001,
                        'ip_address': u'209.85.241.171',
                        'host_name': u'209.85.241.171'}
```

(continues on next page)

(continued from previous page)

```
                    }
                },
            3: {
                'probes': {
                    1: {
                        'rtt': 2.529,
                        'ip_address': u'216.239.49.123',
                        'host_name': u'216.239.49.123'},
                    2: {
                        'rtt': 2.474,
                        'ip_address': u'209.85.255.255',
                        'host_name': u'209.85.255.255'
                    },
                    3: {
                        'rtt': 7.813,
                        'ip_address': u'216.239.58.193',
                        'host_name': u'216.239.58.193'}
                    }
                },
            4: {
                'probes': {
                    1: {
                        'rtt': 1.361,
                        'ip_address': u'8.8.8.8',
                        'host_name': u'google-public-dns-a.google.com'
                    },
                    2: {
                        'rtt': 1.605,
                        'ip_address': u'8.8.8.8',
                        'host_name': u'google-public-dns-a.google.com'
                    },
                    3: {
                        'rtt': 0.989,
                        'ip_address': u'8.8.8.8',
                        'host_name': u'google-public-dns-a.google.com'}
                    }
                }
            }
        }

OR

{
    'error': 'unknown host 8.8.8.8'
}
```

## 3.7 YANG

For documentation regarding `YANG`, please visit napalm-yang documentation.

## 3.8 napalm-logs

For documentation regarding `napalm-logs`, please visit the dedicated readthedocs page.

## 3.9 Integrations

NAPALM can be integrated with automation frameworks such as Ansible or Salt. In order to use NAPALM with ansible you will need to install napalm-ansible.

### 3.9.1 napalm-ansible

Below are the ansible modules which are currently available for NAPALM

#### modules

Below are the ansible modules which are currently available for NAPALM

#### napalm_cli

Executes network device CLI commands and returns response using NAPALM

#### Parameters

#### Examples

```
- napalm_cli:
    hostname: "{{ inventory_hostname }}"
    username: "napalm"
    password: "napalm"
    dev_os: "eos"
    args:
        commands:
            - show version
            - show snmp chassis


- napalm_cli:
    provider: "{{ napalm_provider }}"
    args:
        commands:
            - show version
```

(continues on next page)

(continued from previous page)

```
            - show snmp chassis
```

### Return

### napalm_diff_yang

Create two YANG objects from dictionaries and runs mehtod

### Parameters

### Examples

```
- napalm_diff_yang:
    first: "{{ candidate.yang_model }}"
    second: "{{ running_config.yang_model }}"
    models:
     - models.openconfig_interfaces
  register: diff
```

### Return

### napalm_get_facts

Gathers facts from a network device via the Python module napalm

### Parameters

### Examples

```
- name: get facts from device
  napalm_get_facts:
    hostname: '{{ inventory_hostname }}'
    username: '{{ user }}'
    dev_os: '{{ os }}'
    password: '{{ passwd }}'
```

(continues on next page)

(continued from previous page)

```
    filter: ['facts']

  register: result


- name: print data

  debug:

    var: result


- name: Getters

  napalm_get_facts:

    provider: "{{ ios_provider }}"

    filter:

      - "lldp_neighbors_detail"

      - "interfaces"


- name: get facts from device

  napalm_get_facts:

    hostname: "{{ host }}"

    username: "{{ user }}"

    dev_os: "{{ os }}"

    password: "{{ password }}"

    optional_args:

      port: "{{ port }}"

    filter: ['facts', 'route_to', 'interfaces']

    args:

        route_to:

            protocol: static

            destination: 8.8.8.8
```

### Return

### napalm_install_config

This library will take the configuration from a file and load it into a device running any OS supported by napalm. The old configuration will be replaced or merged with the new one.

### Parameters

### Examples

```
- assemble:
    src: '../compiled/{{ inventory_hostname }}/'
    dest: '../compiled/{{ inventory_hostname }}/running.conf'


- name: Install Config and save diff
  napalm_install_config:
    hostname: '{{ inventory_hostname }}'
    username: '{{ user }}'
    dev_os: '{{ os }}'
    password: '{{ passwd }}'
    config_file: '../compiled/{{ inventory_hostname }}/running.conf'
    commit_changes: '{{ commit_changes }}'
    replace_config: '{{ replace_config }}'
    get_diffs: True
    diff_file: '../compiled/{{ inventory_hostname }}/diff'


- name: Install Config using Provider
  napalm_install_config:
    provider: "{{ ios_provider }}"
    config_file: '../compiled/{{ inventory_hostname }}/running.conf'
    commit_changes: '{{ commit_changes }}'
    replace_config: '{{ replace_config }}'
    get_diffs: True
```

(continues on next page)

(continued from previous page)

```
    diff_file: '../compiled/{{ inventory_hostname }}/diff'
```

### Return

### napalm_parse_yang

Parse configuration/state from a file or device and returns a dict that

### Parameters

### Examples

```
- name: Parse from device
  napalm_parse_yang:
    hostname: '{{ inventory_hostname }}'
    username: '{{ user }}'
    dev_os: '{{ os }}'
    password: '{{ passwd }}'
    mode: "config"
    profiles: ["eos"]
    models:
        - models.openconfig_interfaces
  register: running


- name: Parse from file
  napalm_parse_yang:
    file_path: "eos.config"
    mode: "config"
    profiles: ["eos"]
    models:
        - models.openconfig_interfaces
  register: config
```

### Return

### napalm_ping

This module logs into the device, issues a ping request, and returns the response

### Parameters

### Examples

```
- napalm_ping:
    hostname: "{{ inventory_hostname }}"
    username: "napalm"
    password: "napalm"
    dev_os: "eos"
    destination: 10.0.0.5
    vrf: MANAGEMENT
    count: 2


- napalm_ping:
    provider: "{{ napalm_provider }}"
    destination: 8.8.8.8
    count: 2
```

### Return

### napalm_translate_yang

Load a YANG object from a dict and translates the object to native

### Parameters

### Examples

```
- name: "Translate config"
  napalm_translate_yang:
```

(continues on next page)

(continued from previous page)

```
    data: "{{ interfaces.yang_model }}"

    profiles: ["eos"]

    models:

        - models.openconfig_interfaces

  register: config
```

### Return

### napalm_validate

Performs deployment validation via napalm.

### Parameters

### Examples

```
- name: GET VALIDATION REPORT

  napalm_validate:

    username: "{{ un }}"

    password: "{{ pwd }}"

    hostname: "{{ inventory_hostname }}"

    dev_os: "{{ dev_os }}"

    validation_file: validate.yml


- name: GET VALIDATION REPORT USING PROVIDER

  napalm_validate:

    provider: "{{ ios_provider }}"

    validation_file: validate.yml


# USING YANG

- name: Let's gather state of interfaces

  napalm_parse_yang:
```

(continues on next page)

(continued from previous page)

```
    dev_os: "{{ dev_os }}"
    hostname: "{{ hostname }}"
    username: "{{ username }}"
    password: "{{ password }}"
    mode: "state"
    optional_args:
        port: "{{ port }}"
    models:
        - models.openconfig_interfaces
  register: interfaces


- name: Check all interfaces are up
  napalm_validate:
    data: "{{ interfaces.yang_model }}"
    models:
        - models.openconfig_interfaces
    validation_file: "validate.yaml"
  register: report
```

#### Return

## 3.10 Contributing

Contributing is very easy and you can do it many ways; documentation, bugfixes, new features, etc. Any sort of contribution is useful.

### 3.10.1 How to Contribute

In order to speed up things we recommend you to follow the following rules when doing certain types of contributions. If something is not clear don't worry, just ask or send your contribution back and we will help you.

### 3.10.2 New Feature

New features are going to mostly be either a new method that is not yet defined or implementing a method already defined for a particular driver.

#### Proposing a new method

The best way to propose a new method is as follows to send a PR with the proposed method. That will probably spark some debate around the format. The PR will not only have to include the proposed method but some testing.

In addition, before merging we will want an implementation for any driver of your choice.

**For example:**

- get_config proposal - That particular example had an issue that some had raised as a reference but that's not mandatory. You can create an issue first but that's optional.
- get_config implementation for EOS - Before the PR was merged an implementation was provided as a proof of concept. This is mandatory. This PRs doesn't have to arrive at the same time as the previous one but it will be required. Note that the rules for "*Implementing an already defined method*" apply to this PR.

#### Implementing an already defined method

Adding an already defined method to a driver has three very simple steps:

1. Implement the code.
2. Add necessary mocked data.
3. Enable the test and ensure it passes (this step is no longer needed so ignore the `.travis.yaml` change on the example below).

Again get_config implementation for EOS is a good example.

### 3.10.3 Bugfixes

If you found a bug and know how to fix just contribute the bugfix. It might be interesting to provide a test to make sure we don't introduce the bug back in the future but this step is optional.

### 3.10.4 Documentation

The documentation is built using Sphinx and hosted on Read the Docs. The docs are kept in the `docs/` directory at the top of the source tree.

To make changes, it is preferable to set up a python virtual environment called `env` and activate it.

Next, install the documentation dependencies using `pip`:

```
pip install -r docs/requirements.txt
```

Make your changes and then check them for correctness by building them locally:

```
# in the docs directory
make html
```

**Tip:** If it is a simple change to a single page, you can use the "Edit on GitHub" button in the upper right hand corner of the page in question.

### 3.10.5 Proposing a new driver

This is a more complex process but completely doable. You can find more information here.

Please check *Community Drivers* to understand the process.

#### NAPALM Core Developers

NAPALM core developers have the responsibility to maintain the *Supported Devices*, hereby including:

- Identify and fix bugs
- Add new features and enhancements
- Ensure an uniform API
- Triage Issues
- Review and eventually Pull Requests
- Continuously improve the documentation
- Grant access to volunteers that propose new *Community Drivers*
- Provide a set of tools used in the core drivers to test and generate documentation to developers willing to write and maintain extra drivers

#### Who can be a core developer?

Anyone that is (git) committed to the project, proves skills to maintain a driver, and continuously identifies and fixes issues, or any applicable of the above. We are always looking into statistics and considering new members.

#### Community Drivers

In addition to the core drivers, maintained by the *NAPALM Core Developers*, the community can always provide additional drivers. We welcome any submission, and new drivers will be hosted under the napalm-automation-community on GitHub. They can be seen as *plugins* for the base NAPALM framework, which is why each driver is maintained under its own repository.

Once a new driver is added under the named organisation, the user proposing it takes the entire responsibility to maintain. That includes:

- Documentation
- Fix bugs
- Triage and manage issues
- Review and merge Pull Requests
- API compatibility with the core drivers

While everything is a best-effort of the volunteer, we highly encourage the community to avoid submitting drivers they are unable to maintain. We understand that personal and professional situations can change and we are not asking for an ever-lasting commitment but there should be some willingness to take some initial responsibility over it.

The core developers will provide you the tools and the methodologies to create a new driver, together with the associated documentation.

If there is high demand for a specific driver, and its maintainer proves they are able to commit to a continuous work on the long run, the driver can be promoted to be included into the core.

## 3.11 Development

Some useful information for development purposes.

### 3.11.1 Testing Framework

As NAPALM consists of multiple drivers and all of them have to provide similar functionality, we have developed a testing framework to provide a consistent test suite for all the drivers.

#### Features

The testing framework has the following features:

1. Same tests across all vendors. Tests defined in `napalm.base/test/getters.py` are shared across all drivers.
2. Multiple test cases per test.
3. Target of the test can be configured with environmental variables.
4. Expected output is compared against the actual output of the test result.
5. NotImplemented methods are skipped automatically.

#### Using the testing framework

To use the testing framework you have to implement two files in addition to the mocked data:

- `test/unit/test_getters.py` - Generic file with the same content as this file test_getters.py
- `test/unit/conftest.py` - Code specific to each driver with instructions on how to fake the driver. For example, conftest.py

#### Multiple test cases

To create test cases for your driver you have to create a folder named `test/unit/mocked_data/$name_of_test_function/$name_of_test_case`. For example:

- `test/unit/mocked_data/test_get_bgp_neighbors/no_peers/`
- `test/unit/mocked_data/test_get_bgp_neighbors/lots_of_peers/`

Each folder will have to contain it's own mocked data and expected result.

#### Target

By default, the tests are going to be run against mocked data but you can change that behavior with the following environmental variables:

- `NAPALM_TEST_MOCK` - 1 (default) for mocked data and 0 for connecting to a device.
- `NAPALM_HOSTNAME`
- `NAPALM_USERNAME`
- `NAPALM_PASSWORD`

- NAPALM_OPTIONAL_ARGS

### Mocking the open method

To mock data needed to connect to the device, ie, needed by the open method, just put the data in the folder test/unit/mocked_data/

### Examples

Multiple test cases:

```
(napalm)   napalm-eos git:(test_framework)  ls test/unit/mocked_data/test_get_bgp_
↪neighbors
lots_of_peers no_peers      normal
(napalm)   napalm-eos git:(test_framework)  py.test test/unit/test_getters.
↪py::TestGetter::test_get_bgp_neighbors
...
test/unit/test_getters.py::TestGetter::test_get_bgp_neighbors[lots_of_peers] <- ../
↪napalm/napalm.base/test/getters.py PASSED
test/unit/test_getters.py::TestGetter::test_get_bgp_neighbors[no_peers] <- ../napalm/
↪napalm.base/test/getters.py PASSED
test/unit/test_getters.py::TestGetter::test_get_bgp_neighbors[normal] <- ../napalm/
↪napalm.base/test/getters.py PASSED
```

Missing test cases:

```
(napalm)   napalm-eos git:(test_framework)  ls test/unit/mocked_data/test_get_bgp_
↪neighbors
ls: test/unit/mocked_data/test_get_bgp_neighbors: No such file or directory
(napalm)   napalm-eos git:(test_framework)  py.test test/unit/test_getters.
↪py::TestGetter::test_get_bgp_neighbors
...
test/unit/test_getters.py::TestGetter::test_get_bgp_neighbors[no_test_case_found] <- .
↪./napalm/napalm.base/test/getters.py FAILED

=============================================================== FAILURES␣
↪===============================================================
_____________________________________ TestGetter.test_get_bgp_neighbors[no_test_case_
↪found] ______________________________________

cls = <test_getters.TestGetter instance at 0x10ed5eb90>, test_case = 'no_test_case_
↪found'

    @functools.wraps(func)
    def wrapper(cls, test_case):
        cls.device.device.current_test = func.__name__
        cls.device.device.current_test_case = test_case

        try:
            # This is an ugly, ugly, ugly hack because some python objects don't load
            # as expected. For example, dicts where integers are strings
            result = json.loads(json.dumps(func(cls)))
        except IOError:
            if test_case == "no_test_case_found":
>               pytest.fail("No test case for '{}' found".format(func.__name__))
```

(continues on next page)

(continued from previous page)

```
E               Failed: No test case for 'test_get_bgp_neighbors' found

../napalm/napalm.base/test/getters.py:64: Failed
================================================== 1 failed in 0.12 seconds␣
↪==================================================
```

Method not implemented:

```
(napalm)   napalm-eos git:(test_framework)  py.test test/unit/test_getters.
↪py::TestGetter::test_get_probes_config
...
test/unit/test_getters.py::TestGetter::test_get_probes_config[no_test_case_found] <- .
↪./napalm/napalm.base/test/getters.py SKIPPED

================================================== 1 skipped in 0.09 seconds␣
↪==================================================
```

### 3.11.2 Testing Matrix

NAPALM leverages [Github Actions](https://docs.github.com/en/actions) to test and lint code on commits and pull requests. If you want to test prior to opening a pull request, you can use [nektos/act](https://github.com/nektos/act) and Docker to locally run the tests

```
$ act -j std_tests
[build/std_tests-4]   Start image=catthehacker/ubuntu:act-latest
[build/std_tests-3]   Start image=catthehacker/ubuntu:act-latest
[build/std_tests-1]   Start image=catthehacker/ubuntu:act-latest
[build/std_tests-2]   Start image=catthehacker/ubuntu:act-latest
[build/std_tests-5]   Start image=catthehacker/ubuntu:act-latest
[build/std_tests-4]     docker pull image=catthehacker/ubuntu:act-latest platform=␣
↪username= forcePull=true
[build/std_tests-1]     docker pull image=catthehacker/ubuntu:act-latest platform=␣
↪username= forcePull=true
[build/std_tests-3]     docker pull image=catthehacker/ubuntu:act-latest platform=␣
↪username= forcePull=true
[build/std_tests-5]     docker pull image=catthehacker/ubuntu:act-latest platform=␣
↪username= forcePull=true
[build/std_tests-2]     docker pull image=catthehacker/ubuntu:act-latest platform=␣
↪username= forcePull=true

...

| ----------------------------------------------------------------------
| TOTAL                                    9258   1836    80%
|
| ================= 619 passed, 80 skipped, 3 warnings in 19.97s =================
[build/std_tests-5]     Success - Main Run Tests
[build/std_tests-5]  Run Post Setup Python 3.11
[build/std_tests-5]     docker exec cmd=[node /var/run/act/actions/actions-setup-
↪python@v2/dist/cache-save/index.js] user= workdir=
[build/std_tests-5]     Success - Post Setup Python 3.11
[build/std_tests-5]   Job succeeded
```

### 3.11.3 Triaging Issues and Pull Requests

**Note:** This document serves mainly as a reference for the NAPALM maintainers, but the users are equally welcome to read this document and understand our process, and eventually suggest improvements.

We triage Issues and Pull Requests (PR) using GitHub features only:

- *Labels*
- *Milestone*
- *Projects*

#### Labels

#### Driver labels

Each platform supported by NAPALM has associated a label, e.g., `junos`, `eos`, `ios`, `iosxr_netconf`, `iosxr`, `vyos`, etc. It is mandatory that the maintainer to apply one or more of these labels.

##### `api change`

If the Issue would imply a change in the API, or the PR introduces changes in the API. By API change we refer to changes in the getters output structure, methods signature, or any core changes that must be uniformly introduced across all the drivers.

##### `awesome`

When someone adds or proposes something really awesome.

##### `base`

When base components are affected, e.g., *get_network_driver*, the validate functionality, or the testing framework.

##### `blocked`

Added in case we block the PR temporarly, or an Issue is currently blocked by other internal or external factors (PRs pending to be merged, other bugs to be solved a priori, etc.)

##### `bug`

Whenever the behaviour reported in the Issue is different than it should, or the PR kills a bug.

##### `cannot reproduce`

This refers to Issues only, and it is added when the maintainer(s) cannot reproduce the behaviour reported.

#### core

When any core components (drivers) are affected.

#### deprecation

Added only to PRs, when a API deprecation is introduced.

#### documentation

Can be added to both Issues and PRs, anything related to the documentation.

#### duplicate

Applicable to Issues only, to be added before closing a duplicate.

#### feature

When a new feature is introduced, or the user requests a new feature.

#### good first issue

While we want to encourage the community to contribute more and more frequent, many engineers are still afraid of complex tasks. This label marks simple fixes that new contributors can address. It is recommended that this label to be accompanied by an explanation and a pointer for the new contributors.

#### help wanted

This marks an Issue were we ask the community for help, or we need more details on a particular topic (e.g., outputs from different platforms, explanation, etc.) from any volunteer from the community.

Once we have all the details required, the maintainer has to remove this label even though it does not start working on it immediately.

#### high severity

Whenever a *bug* affects severely one or more features, making it basically unusable.

#### info needed

We add this label when we need more details and further explanation from the user that reports an Issue. Once we received everything needed, we can remove that label.

#### investigation

We need to investigate the problem further.

**new driver**

When we discuss the possibility to add a new core driver.

**new method**

When we discuss the possibility or implement a new method to one or more drivers. The method does not necessarily need to be a completely new one to NAPALM.

**vendor bug**

When the bug is casued by a vendor stupidity.

#### Milestone

The milestones are used to group the Issues and the Pull Requests from a different angle:

#### Version

The Issue will be solved, or the PR will be included in this release.

**APPROVED**

It means that we accept the Issue or the PR, but we don't have a schedule yet for when the Issue will be solved, or the PR will be included in a release.

**BLOCKED**

This groups the Issues or the PRs we could not accept for the reasons marked using the labels.

**DISCUSSION**

The Issue or the PR needs further discussion.

#### Projects

Any major change that may consist on several Pull Requests should be groupped into a GitHub Project.

## 3.12 Hackathons

### 3.12.1 Hackathon 2016

Welcome to the very first `NAPALM` hackathon ever. What you are about to see is a bunch of people doing the unthinkable; `writing code to manage the network`!!!

**Important:** I love the smell of `automation` in the morning.

### Introduction

During a weekend we will gather online to hack around napalm, fix existing issues, clean the codebase or just do whatever we want.

### Quick Information

| | |
|---|---|
| **Date** | 16, 17 and 18th of September |
| **Location** | *Here for more information* |
| **Slack** | `#napalm-hackathon-2016` in networkToCode |
| **Live Feed** | Youtube Live |
| **Recordings** | Youtube |

### Information

### Agenda

**Important:** All dates and times are in UTC

### Location

The hackathon will be held online. We will use slack as the main communications channel, github to coordinate the work and hangouts for live presentations, which will be posted online on a youtube channel within the hour.

Check this *link* for more information regarding the slack channel, hangouts, youtube, etc..

### IRL Gatherings

Apparently there is something called real life and people like to gather in groups, shocking, isn't it? We will gather in some locations just for the sake of feeling the warmth of other human beings and probably have beer after a long day of hacking. This is completely optional and unofficial, although, if you want to host a physical meetup I will be happy to announce it here.

### Known gatherings

- London, UK:

```
CloudFlare, 25 Lavington St, London SE1 0NZ
Contact: Mircea, @mirceaulinic (slack), tel: +447427735256
```

- New York, NY, United States:

```
Network to Code (WeWork location), 315 W. 36th, NY
Contact: Jason Edelman  (jason@networktocode.com) or jedelman8 on Slack and
↪Twitter
```

- San Francisco, CA, United states:

```
Contact ktbyers@twb-tech.com for details
```

- Krakow, Poland:

```
Contact elisa@bigwaveit.org for details
```

### Participating

#### Before the event

Feel free to navigate all the repos on the napalm automation organization and find issues you might want to work on.

1. If you find any feel free to comment on the issue to let the organizers know.
2. If you don't and you know what you would like to work on, please, create an issue with the description of what you want to achieve.
3. If you want to participate and you don't know what to do, feel free to ask on the slack channel. It is already available so go there and ask.

#### During the event

1. Make sure you are on slack. We will use that as our main communications channel. We will make all the announcements there and we will notify you there if we plan to make some announcement on the hangout.
2. If you can't attend the live videos don't sweat. They will be published on YouTube as soon as they are done so you don't miss anything.
3. If you are in New Zealand and think you live on a strange timezone and, thus, you can't participate, you are wrong. GitHub is great for asynchronous communications and I am sure there will always be someone around on slack to help you through any problem you might encounter, to help someone yourself or simply to wind up for a second and just talk about any random topic you want.

#### Mentors

---

**Important:** All levels of expertise are welcome so if you are new to github, python, napalm or even to networking, don't let that be on the way. We have some *mentors* to help you and plenty of tasks to get you started with python, github, and various other tools for testing code, making it look pretty, etc. So if you are looking for some free training this might be a good way to get it ;)

---

The community in networkToCode is pretty much great so if you have any problem just post it on the slack channel. Feel free to also ping any of the mentors if you didn't get any satisfactory answer.

Mentors' slack handles are:

- `dbarroso`

- mirceaulinic
- ggabriele

### Volunteering

Do you want to help out? Please, do it. We need mentors. If you know how to work with git, python, napalm, how to record a google hangout, you want to organize a physical gathering or just correct my spleling, please, ping dbarroso on slack.

### Presentations

- NAPALM Introduction (Slides)
- NAPALM Kickoff (Slides)

# Index


## C

cli() (*napalm.base.base.NetworkDriver method*), 43
close() (*napalm.base.base.NetworkDriver method*), 43
commit_config() (*napalm.base.base.NetworkDriver method*), 43
compare_config() (*napalm.base.base.NetworkDriver method*), 43
compliance_report() (*napalm.base.base.NetworkDriver method*), 43
confirm_commit() (*napalm.base.base.NetworkDriver method*), 44
connection_tests() (*napalm.base.base.NetworkDriver method*), 44

## D

discard_config() (*napalm.base.base.NetworkDriver method*), 44

## G

get_arp_table() (*napalm.base.base.NetworkDriver method*), 44
get_bgp_config() (*napalm.base.base.NetworkDriver method*), 44
get_bgp_neighbors() (*napalm.base.base.NetworkDriver method*), 46
get_bgp_neighbors_detail() (*napalm.base.base.NetworkDriver method*), 47
get_config() (*napalm.base.base.NetworkDriver method*), 49
get_environment() (*napalm.base.base.NetworkDriver method*), 50
get_facts() (*napalm.base.base.NetworkDriver method*), 50
get_firewall_policies() (*napalm.base.base.NetworkDriver method*), 51
get_interfaces() (*napalm.base.base.NetworkDriver method*), 51
get_interfaces_counters() (*napalm.base.base.NetworkDriver method*), 53
get_interfaces_ip() (*napalm.base.base.NetworkDriver method*), 54
get_ipv6_neighbors_table() (*napalm.base.base.NetworkDriver method*), 55
get_lldp_neighbors() (*napalm.base.base.NetworkDriver method*), 55
get_lldp_neighbors_detail() (*napalm.base.base.NetworkDriver method*), 56
get_mac_address_table() (*napalm.base.base.NetworkDriver method*), 57
get_network_instances() (*napalm.base.base.NetworkDriver method*), 58
get_ntp_peers() (*napalm.base.base.NetworkDriver method*), 59
get_ntp_servers() (*napalm.base.base.NetworkDriver method*), 59
get_ntp_stats() (*napalm.base.base.NetworkDriver method*), 59
get_optics() (*napalm.base.base.NetworkDriver method*), 60
get_probes_config() (*na-*

*palm.base.base.NetworkDriver method*), 61
get_probes_results() (*napalm.base.base.NetworkDriver method*), 62
get_route_to() (*napalm.base.base.NetworkDriver method*), 63
get_snmp_information() (*napalm.base.base.NetworkDriver method*), 64
get_users() (*napalm.base.base.NetworkDriver method*), 64
get_vlans() (*napalm.base.base.NetworkDriver method*), 65

## H

has_pending_commit() (*napalm.base.base.NetworkDriver method*), 65

## I

is_alive() (*napalm.base.base.NetworkDriver method*), 65

## L

load_merge_candidate() (*napalm.base.base.NetworkDriver method*), 65
load_replace_candidate() (*napalm.base.base.NetworkDriver method*), 66
load_template() (*napalm.base.base.NetworkDriver method*), 66

## N

NetworkDriver (*class in napalm.base.base*), 42

## O

open() (*napalm.base.base.NetworkDriver method*), 66

## P

ping() (*napalm.base.base.NetworkDriver method*), 66
post_connection_tests() (*napalm.base.base.NetworkDriver method*), 67
pre_connection_tests() (*napalm.base.base.NetworkDriver method*), 67

## R

rollback() (*napalm.base.base.NetworkDriver method*), 67

## T

traceroute() (*napalm.base.base.NetworkDriver method*), 68