Ansible with Arista devices Documentation

Release 0.9

T

Oct 21, 2019

Contents:

CHAPTER 1

Overview

1.1 About Ansible modules for Arista EOS automation

Arista EOS modules are part of the core modules list of ansible and do not require any additional 3rd part modules to be installed on your server. They are maintained by Ansible Network Team and provides options to manage configuration and get status as well.

A complete list of available eos module is available on Ansible documentation website

1.2 Available How-To:

Do not forget to install requirements if you want to run tests

- All **ansible** content is under ansible-content folder.

List of documents:

- Manage `eos` configuration with ansible: eos_config module
- Getting status of Arista devices with Ansible: eos_command module
- Jinja2 & YML structures: templating engine

1.3 Installation & Requirements

Please refer to installation before running content of this repository

1.4 Ask question or report issue

Please open an issue on Github this is the fastest way to get an answer.

1.5 Contribute

Contributing pull requests are gladly welcomed for this repository. If you are planning a big change, please start a discussion first to make sure we'll be able to merge it.

1.6 License

Project is published under BSD License.

CHAPTER 2

Documentation structure

2.1 Overview

2.1.1 About Ansible modules for Arista EOS automation

Arista EOS modules are part of the core modules list of ansible and do not require any additional 3rd part modules to be installed on your server. They are maintained by Ansible Network Team and provides options to manage configuration and get status as well.

A complete list of available eos module is available on Ansible documentation website

2.1.2 Available How-To:

Do not forget to install requirements if you want to run tests

- All **ansible** content is under ansible-content folder.

List of documents:

- Manage `eos` configuration with ansible: eos_config module
- Getting status of Arista devices with Ansible: eos_command module
- Jinja2 & YML structures: templating engine

2.1.3 Installation & Requirements

Please refer to installation before running content of this repository

2.1.4 Ask question or report issue

Please open an issue on Github this is the fastest way to get an answer.

2.1.5 Contribute

Contributing pull requests are gladly welcomed for this repository. If you are planning a big change, please start a discussion first to make sure we'll be able to merge it.

2.1.6 License

Project is published under BSD License.

2.2 Installation and Lab Environment

2.2.1 Requirements

Repository requires to install some requirements to be consumed:

- Docker daemon
- **docker-topo** script
- cEOS-LAB image
- ansible software

In the meantime, it is recommended to run this repository in a virtual-environment. To start such environment, use following commands:

```
$ python3 -m pip install virtualenv
$ python3 -m virtualenv ansible_training
$ cd ansible_training
$ source bin/activate
```

2.2.2 Docker and docker-topo

Docker installation is platform specific and you should use following links:

- macOS installation
- Ubuntu installation
- Centos installation

Then, install `docker-topo` from `pip`:

```
$ python3 -m pip install git+https://github.com/networkop/docker-topo.git
```

2.2.3 Get cEOS-LAB image

With your Arista login, go to download page and download cEOS-LAB image on your laptop. Then, add ceos images to docker:

```
$ docker import cEOS-lab.tar.xz ceosimage:latest
```

2.2.4 Run docker topology

Once docker-topo is installed, run the docker topology with following commands:

```
# Clone repository locally
$ git clone https://github.com/titom73/ansible-arista-module-howto.git

# Enter repository
$ cd ansible-arista-module-howto/

# Build docker topology
$ docker-topo --create ansible-demo-topology.yaml
```

2.2.5 Install requirements

Install ansible with `pip`:

```
$ pip install -r requirements.txt
```

Then, check ansible version (version might have changed until we release this document):

```
$ ansible --version
ansible 2.7.8
  config file = /Users/tgrimonet/Projects/ansible-demo/ansible.cfg
  configured module search path = ['/Users/tgrimonet/.ansible/plugins/modules', '/usr/
↪share/ansible/plugins/modules']
  ansible python module location = /Users/tgrimonet/.venv/ansible-demo/lib/python3.7/
↪site-packages/ansible
  executable location = /Users/tgrimonet/.venv/ansible-demo/bin/ansible
  python version = 3.7.2 (default, Jan 13 2019, 12:50:01) [Clang 10.0.0 (clang-1000.
↪11.45.5)]
```

2.3 Collecting status from Arista EOS devices

Ansible provides 2 different modules to collect information from EOS devices:

- `eos_facts`: Collect facts from devices such has Hardware, EOS version, configuration.
- `eos_command`: Collect result of any command you send to a device.

2.3.1 Collecting facts from devices.

This module collects all device facts from remote device. All keys found on device are prepend with `ansible_net_`. In addition of](https://docs.ansible.com/ansible/latest/reference_appendices/common_return_values.html#common-return-values) collecting by Ansible, this module also collects following keys:

- All IPv4 addresses configured on the device
- All IPv6 addresses configured on the device
- The current active config from the device
- All file system names available on the device
- The fully qualified domain name of the device

- The list of fact subsets collected from the device
- The configured hostname of the device
- The image file the device is running
- A hash of all interfaces running on the system
- The available free memory on the remote device in Mb
- The total memory on the remote device in Mb
- The model name returned from the device
- The list of LLDP neighbors from the remote device
- The serial number of the remote device
- The operating system version running on the remote device

Collect all facts and display some variables

Below is a playbook example that grab all facts, register them into a variable named `facts` and then display output

```
---
- name: Run commands on remote LAB devices
  hosts: lab
  connection: network_cli
  gather_facts: false
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Collect all facts from device
      eos_facts:
        provider: "{{arista_credentials}}"
        gather_subset:
          - all
      register: facts

    - name: Display result
      debug:
        msg: "Model is {{facts.ansible_facts.ansible_net_model}} and it is running {
↪{facts.ansible_facts.ansible_net_version}}"
```

Output of this playbook is quite easy to:

```
$ ansible-playbook pb.get.commands.yaml --limit ceos1
[...]
TASK [Display result] ********
ok: [ceos01] => {
    "msg": "Model is cEOS and it is running 4.21.1.1F"
}
[...]
```

Save device configuration from facts

To save configuration, we can collect only configuration fact from device and save output to a file like in playbook below:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: false
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Create output directory (if missing)
      file:
        path: "{{playbook_dir}}/config"
        state: directory

    - name: Collect all facts from device
      eos_facts:
        provider: "{{arista_credentials}}"
        gather_subset:
          - config
      register: facts

    - name: copy collected configuration in configuration/text directory
      copy:
        content: "{{ facts.ansible_facts.ansible_net_config }}"
        dest: "{{ playbook_dir }}/config/{{ inventory_hostname }}.conf"
```

Playbook will create a folder named `config` if it is missing and then save config files:

```
$ ansible-playbook pb.get.commands.yaml

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos01]
ok: [ceos02]


TASK [Create output directory (if missing)] ********
ok: [ceos01]
ok: [ceos02]


TASK [Collect all facts from device] ********
ok: [ceos01]
ok: [ceos02]


TASK [copy collected configuration in configuration/text directory] ********
changed: [ceos01]
changed: [ceos02]

PLAY RECAP ********
ceos01                     : ok=4    changed=1    unreachable=0    failed=0
ceos02                     : ok=4    changed=1    unreachable=0    failed=0
```

And from your shell perspective:

```
$ tree -L 2
.
├── ansible.cfg
├── authentication.yaml
├── config
│   ├── ceos1.conf
│   └── ceos2.conf
├── group_vars
├── host_vars
│   ├── ceos1
│   └── ceos2
```

2.3.2 Collecting result of commands

Collect full command result:

Another module to collect status of EOS devices is `eos_command` module. It gives you a method to run commandon any Arista device and save result for further tasks.

Next playbook will monitor status of BGP peers:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: false
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Collect BGP Status
      eos_command:
        commands:
          - enable
          - show ip bgp summary
        provider: "{{arista_credentials}}"
      register: bgp_status

    - name: Display result
      debug:
        var: bgp_status
```

And the result is (truncated):

```
$ ansible-playbook pb.collect.status.bgp.yaml

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]
ok: [ceos2]

TASK [Collect BGP Status] ********
ok: [ceos2]
ok: [ceos1]
```

(continues on next page)

(continued from previous page)

```
TASK [Display result] ********
ok: [ceos1] => {
    "bgp_status": {
        "changed": false,
        "failed": false,
        "stdout": [
            {},
            {
                "vrfs": {
                    "default": {
                        "asn": 65001,
[...]
PLAY RECAP ********
ceos1                      : ok=3    changed=0    unreachable=0    failed=0
ceos2                      : ok=3    changed=0    unreachable=0    failed=0
```

Filter results using loop

Because Output is very verbose, we can reduce a bit to focus only on status of any configured peers by using `with_dict` loop management:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: false
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Collect BGP Status
      eos_command:
        commands:
          - enable
          - show ip bgp summary
        provider: "{{arista_credentials}}"
      register: bgp_status

    - name: Display result
      debug:
        msg: "Peering with {{item.key}} is {{item.value.peerState}}"
      with_dict: "{{bgp_status.stdout[1].vrfs.default.peers}}"
```

And output is quite shorter than previous try:

```
$ ansible-playbook pb.collect.status.bgp.yaml

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]
ok: [ceos2]

TASK [Collect BGP Status] ********
```

(continues on next page)

(continued from previous page)

```
ok: [ceos2]
ok: [ceos1]

TASK [Display result] ********
ok: [ceos1] => (item={'key': '10.0.0.2', 'value': {'msgSent': 33, 'inMsgQueue': 0,
↪'prefixReceived': 0, 'upDownTime': 1552640949.496501, 'version': 4, 'msgReceived':␣
↪32, 'prefixAccepted': 0, 'peerState': 'Established', 'outMsgQueue': 0,
↪'underMaintenance': False, 'asn': 65002}}) => {
    "msg": "Peering with 10.0.0.2 is Established"
}
ok: [ceos2] => (item={'key': '10.0.0.1', 'value': {'msgSent': 32, 'inMsgQueue': 0,
↪'prefixReceived': 0, 'upDownTime': 1552640948.972306, 'version': 4, 'msgReceived':␣
↪32, 'prefixAccepted': 0, 'peerState': 'Established', 'outMsgQueue': 0,
↪'underMaintenance': False, 'asn': 65001}}) => {
    "msg": "Peering with 10.0.0.1 is Established"
}

PLAY RECAP ********
ceos1                      : ok=3    changed=0    unreachable=0    failed=0
ceos2                      : ok=3    changed=0    unreachable=0    failed=0
```

Wait for a specific result

Waiting for a specific result is useful to set a task to fail or success depending on a change propagating in the network.

In this example, we are going to put one device in maintenance mode. To do that, we use a playbook to configure device in maintenance mode using *eos_config* and then waiting for result of this change with command `show maintenance`

```
---
- name: Run commands on remote LAB devices
  hosts: ceos1
  connection: local
  gather_facts: false
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure device hostname from lines
      eos_config:
        provider: "{{arista_credentials}}"
        lines:
          - quiesce
        parents:
          - maintenance
          - unit System

    - name: Wait for device to change to maintenance mode (10sec)
      eos_command:
        provider: "{{arista_credentials}}"
        commands: show maintenance
        wait_for: result[0]['units']['System']['state'] eq 'underMaintenance'
        interval: 2
        retries: 5
```

In case of these 10 seconds (5 tries with 2 seconds in between) are not enough, then, task fails:

```
$ ansible-playbook pb.collect.waitfor.maintenance.yaml

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]

TASK [Configure device hostname from lines] ********
ok: [ceos1]

TASK [Wait for device to change to maintenance mode (10sec)] ********
fatal: [ceos1]: FAILED! => {"changed": false, "failed_conditions": ["result[0]['units
↪']['System']['state'] eq 'underMaintenance'"], "msg": "One or more conditional␣
↪statements have not been satisf
ied"}

PLAY RECAP ********
ceos1                      : ok=2    changed=0    unreachable=0    failed=1
```

But when device changes to `underMaintenance`, then, task is a success:

```
$ ansible-playbook pb.collect.waitfor.maintenance.yaml

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]

TASK [Configure device hostname from lines] ********
ok: [ceos1]

TASK [Wait for device to change to maintenance mode (10sec)] ********
ok: [ceos1]

PLAY RECAP ********
ceos1                      : ok=3    changed=0    unreachable=0    failed=0
```

2.4 Configure Arista EOS with Ansible

`eos_config` is a core module managed by Ansible network team. As this module is part of the core, there is no need to install additional Ansible module with `ansible-galaxy`

As `eos_*` modules are part of core engine, we can use ansible options to run tests and capture configuration changes when running playbooks:

- **`--check` option**: When ansible-playbook is executed with `--check` it will not make any changes on remote systems. Modules will report what changes they would have made rather than making them.
- **`--diff` option**: When this flag is supplied and the module supports this, Ansible will report back the changes made or, if used with –check, the changes that would have been made.

  These options are generally called **dry-run** mode.

2.4.1 Apply lines of configurations to devices.

Example playbooks: 'pb.config.lines.simple.yaml'

Basic lines of configuration

To push lines of configuraiton to device, a small playbook should be like this:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure device hostname from lines
      eos_config:
        provider: "{{arista_credentials}}"
        lines:
          - "hostname MyHostname"
          - "ntp server 1.1.1.1"
```

In this playbook, we use an authentication provider for ARISTA products as described in the home page

When applying this playbook, output is like below:

using `--diff` option display changes applied by tasks

```
$ ansible-playbook pb.config.lines.simple.yaml --diff

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]
ok: [ceos2]

TASK [Configure device hostname from lines] ********
--- system:/running-config
+++ session:/********_1551450161-session-config
@@ -2,7 +2,9 @@
 !
 transceiver qsfp default-mode 4x10G
 !
-hostname CEOS2
+hostname MyHostname
+!
+ntp server 1.1.1.1
 !
 spanning-tree mode mstp
 !

changed: [ceos2]
--- system:/running-config
+++ session:/********_1551450161-session-config
@@ -2,7 +2,7 @@
```

(continues on next page)

(continued from previous page)

```
 !
 transceiver qsfp default-mode 4x10G
 !
-hostname CEOS1
+hostname MyHostname
 !
 ntp server 1.1.1.1
 !

changed: [ceos1]

PLAY RECAP ********
ceos1                      : ok=2    changed=1    unreachable=0    failed=0
ceos2                      : ok=2    changed=1    unreachable=0    failed=0
```

Dynamic lines of configuration

Even if it is not the easiest way to play with, you can include Jinja2 variables in your configuration lines. These Jinja2 variables shall be defined previously in your `host_vars` and / or `group_vars`

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure device hostname from lines
      eos_config:
        provider: "{{arista_credentials}}"
        lines:
          - "hostname {{inventory_hostname}}-ansible"
          - "ntp server 1.1.1.1"
```

Below is output of a change when running this specific playbooK with `--diff`

```
TASK [Configure device hostname from lines] ********
--- system:/running-config
+++ session:/********_1551450491-session-config
@@ -2,7 +2,7 @@
 !
 transceiver qsfp default-mode 4x10G
 !
-hostname MyHostname
+hostname ceos2-automation
 !
 ntp server 1.1.1.1
 !
```

Lines of configuration within a block

Assuming we want to configure an interface, we have to first enter to this block and then push configuration like this:

Arista interace configuration:

```
!
interface Ethernet 1
  description My Wonderful description
!
```

To achieve this configuration with ansible module, syntax should be like this:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure interface description
      eos_config:
        provider: "{{arista_credentials}}"
        lines:
          - description My Wonderful description
        parents: interface Ethernet 1
        replace: block
```

With a `--diff`, we can that description is applied to interface `Ethernet 1`

```
[...]
changed: [ceos1]
--- system:/running-config
+++ session:/********_1551451836-session-config
@@ -13,6 +13,7 @@
 !
 interface Ethernet1
+   description My Wonderful description
 !
 interface Ethernet2
 !
[...]
PLAY RECAP ***
ceos1                      : ok=4    changed=1    unreachable=0    failed=0
ceos2                      : ok=4    changed=1    unreachable=0    failed=0
```

And the result is quite simple:

```
ceos1-automation#show interfaces description
Interface      Status         Protocol    Description
Et1            up             up          My Wonderful description
Et2            up             up
```

2.4.2 Apply configuration file to device

When playing with jinja2 templates, it is easier to generate a file and then push it to devices instead of applying line by line.

`eos_config` support a mechanism to push a config files from your server to remote devices. This file can be either

a plain text file or a JINJA2 template. In case of a template, rendering will be done first by ansible and then result will be applied on devices.

Apply a config file

To apply a plain text config, following playbook should be used:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure interface description
      eos_config:
        provider: "{{arista_credentials}}"
        src: inputs/generic-config.cfg
```

Config file can be a complete Arista configuration or just a snippet of the configuration you want to update.

In this example, **inputs/generic-config.cfg** has following content:

```
interface Ethernet2
   description Description from generic configuration block
!
```

And running playbook **pb.config.file.yaml** with `--diff` option shows all changes:

```
$ ansible-playbook pb.config.file.yaml --diff

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]
ok: [ceos2]

TASK [Configure interface description] ********
--- system:/running-config
+++ session:/********_1551711841-session-config
@@ -13,6 +13,7 @@
 interface Ethernet1
 !
 interface Ethernet2
+   description Description from generic configuration block
 !
  no ip routing
 !

changed: [ceos2]
--- system:/running-config
+++ session:/********_1551711841-session-config
@@ -13,6 +13,7 @@
 interface Ethernet1
 !
```

(continues on next page)

(continued from previous page)

```
 interface Ethernet2
+   description Description from generic configuration block
 !
 no ip routing
 !

changed: [ceos1]

PLAY RECAP ********
ceos1                      : ok=2    changed=1    unreachable=0    failed=0
ceos2                      : ok=2    changed=1    unreachable=0    failed=0
```

Apply a template

`eos_config` also supports a JINJA2 template as an input file. In this scenario, ansible will run template rendering locally and then will push result to devices. It means that your configuration will have some content specific per device and / or collected in a previous task.

> This section will not describe JINJA2 syntax. A specific page is available with some hints about jinja2 syntax and YAML structures.

Following template creates a basic SNMP configuration with generic fields. Only thing is `chassis-id` will be set with `inventory_hostname` defined in our `inventory.ini` file.

```
snmp-server chassis-id {{inventory_hostname}}
snmp-server contact demo@acme.com
snmp-server location "cEOS Virtual lab"
```

Playbook is very close to playbook in Apply a config file

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure SNMP information with template
      eos_config:
        provider: "{{arista_credentials}}"
        src: "inputs/template-config.j2"
```

The main difference is `src` field where you put a template file instead of a plan text configuration.

As you can see in extract below, content is built on a per device approach:

```
$ ansible-playbook pb.config.template.yaml --diff

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]
ok: [ceos2]
```

(continues on next page)

(continued from previous page)

```
TASK [Configure SNMP information with template] ********
--- system:/running-config
+++ session:/********_1551713015-session-config
@@ -3,6 +3,10 @@
 transceiver qsfp default-mode 4x10G
 !
 hostname ceos1
+!
+snmp-server chassis-id ceos1
+snmp-  server contact demo@acme.com
+snmp-server location "cEOS Virtual lab"
 !
 spanning-tree mode mstp
 !

changed: [ceos1]
--- system:/running-config
+++ session:/********_1551713015-session-config
@@ -3,6 +3,10 @@
 transceiver qsfp default-mode 4x10G
 !
 hostname ceos2
+!
+snmp-server chassis-id ceos2
+snmp-server contact demo@acme.com
+snmp-server location "cEOS Virtual lab"
 !
 spanning-tree mode mstp
 !

changed: [ceos2]

PLAY RECAP ********
ceos1                      : ok=2    changed=1    unreachable=0    failed=0
ceos2                      : ok=2    changed=1    unreachable=0    failed=0
```

2.4.3 Intended configuration approach

Pushing intended configuration is probably the best approach in an automation workflow: your device configuration does not deviate from your expectations as complete configuration is pushed by ansible.

To work in this approach, you need to build complete configuration using either a complete template or by assembling rendered templates. Once it is done, you can push and replace configuration with following syntax:

```
  tasks:
   - name: Load a config in an intended way
     eos_config:
       provider: "{{arista_credentials}}"
       src: "{{ lookup('file', 'inputs/{{inventory_hostname}}-master.cfg') }}"
       replace: 'config'
```

This approach can also be applied with template:

```
  tasks:
    - name: Intended configuration management
```

(continues on next page)

(continued from previous page)

```
      eos_config:
        provider: "{{arista_credentials}}"
        src: "device-configuration.j2"
        replace: "config"
```

2.4.4 And finally, how to save config

`eos_config` module can automatically save config to the **startup_config**. It provides 3 different options to do that. The `save` action is started by either **save** or **save_when** keywords:

- **save**: automatically save the running-config to startup-config after any execution of the task.
- **save_when**: gives an option to select when ansible save running-config to startup-config:
  - `always`: Equal to **save** keyword
  - `modified`: The running-config will only be copied to the startup-config if it has changed since the last save to startup-config
  - `changed`: (ansible >= 2.5) The running-config will only be copied to the startup-config if the task has made a change
  - `never`: Well, running will never be copied to startup-config. Can be useful for validation purpose

So the last playbook should be like this one:

```
---
- name: Run commands on remote LAB devices
  hosts: all
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"

  tasks:
    - name: Configure SNMP information with template
      eos_config:
        provider: "{{arista_credentials}}"
        src: "inputs/template-config.j2"
        save_when: changed
```

2.4.5 Get diff between running and intended configuration

In cas we generate complete configuration, it is nice to capture the deviation between our target (generated configuration) and running configuration on devices.

So to do that, `eos_config` has a special option named `diff_against: intended` and then we defined what we named **intended** configuration.

```
---
- name: Run commands on remote LAB devices
  hosts: ceos1
  connection: local
  gather_facts: no
  pre_tasks:
    - include_vars: "authentication.yaml"
```

(continues on next page)

(continued from previous page)

```
  tasks:
    - name: diff the running config against a master config
      eos_config:
        provider: "{{arista_credentials}}"
        diff_against: intended
        intended_config: "{{ lookup('file', 'inputs/{{inventory_hostname}}-master.cfg
↪') }}"
```

Output of the **pb.config.intended.yaml** playbook is the following:

```
$ ansible-playbook pb.configure.intended.yaml --check --diff

PLAY [Run commands on remote LAB devices] ********

TASK [include_vars] ********
ok: [ceos1]

TASK [diff the running config against a master config] ********
--- before
+++ after
@@ -1,16 +1,10 @@
 transceiver qsfp default-mode 4x10G
-hostname ceos1
-snmp-server chassis-id ceos1
-snmp-server contact demo@acme.com
-snmp-server location "cEOS Virtual lab"
+hostname CEOS1-TEST
 spanning-tree mode mstp
 no aaa root
 username ******** privilege 15 secret sha512 $6$j80NtRkV0CMlgXPS$a0.
↪JbwuO1NMvIthS4eu6dEMHIV9gNGRRFf5SI6qNu5g4I3zxinnVrSMyj8EkQ1V/x7ORAWwe5CpYmgQME2jad1
 interface Ethernet1
 interface Ethernet2
-   description My Wonderful description
-   switchport trunk allowed vlan 12
-   switchport mode trunk
 no ip routing
 management api http-commands
    no shutdown

changed: [ceos1]

PLAY RECAP ********
ceos1                      : ok=2    changed=1    unreachable=0    failed=0
```

As you can see, we have a complete view of the configuration deviation between running and intended.

Note: this module requires to be run with –check flag.

2.5 YAML Structures in Jinja2

Data structure to use YAML with JINJA2 templates. YAML keeps data stored as a map containing keys and values associated to those keys.

2.5.1 Basic

Generic key allocation

YAML input data structure

```
---
time_zone: Europe/Paris
hostname: myDevice
comment: "This device is a fake one"
```

JINJA2 to consume YAML data structure

```
{% if time_zone is defined %}
clock timezone  {{ time_zone }};
{% endif %}
```

Structured key allocation

YAML will consider lines prefixed with more spaces than parent key are contained inside it

```
---
routing_policy:
  communities:
    myCommunity: 99:1
    tenant2: 99:2
```

Data structure similar to this YAML representation is :

```
routing_policy = {
    "communities": {
        "myCommunity": "99:1",
        "tenant2": "99:2"
    }
}
```

2.5.2 List Management

YAML List

Assuming following data structure in YAML

```
---
ntp_servers:
  - 8.8.8.8
  - 4.4.4.4
```

Access data from JINJA2 can be done with the following code:

```
{% for ntp_server in ntp_servers %}
ntp server {{ ntp_server }};
{% endfor %}
```

Data structure similar to this YAML representation is :

```
ntp_servers = ['8.8.8.8', '4.4.4.4']
```

YAML Dictionary

Assuming following data structure in YAML

```
---
vlans:
  10: descr1
  20: descr2
  30: descr3
```

Access data from JINJA2 can be done with the following code:

```
{% for vlan_id, descr in vlans.items() %}
vlan {{ vlan_id }}
  description {{ descr }}
{% endfor %}
```

Data structure similar to this YAML representation is :

```
vlans = {10: "descr1", 20: "descr2", 30: "descr3"}
```

YAML Maps

Assuming following data structure in YAML

```
---
interfaces:
  - name: "ge-0/0/0"
    descr: "Blah"
  - name: "ge-0/0/1"
    descr: "comment"
```

Access data from JINJA2 can be done with the following code:

```
{% for interface in interfaces %}
interface {{interface.name}} {
   description {{interface.descr}};
{% endfor %}
```

Data structure similar to this YAML representation is :

```
interfaces = [{"name": "ge-0/0/0", "descr": "Blah"},
              {"name": "ge-0/0/1", "descr": "comment"}]
```

2.5.3 Advanced Jinja2 syntax

Update variable in a Loop

Syntax below allows user to update value of a variable within a loop and access to it after in a different jinja2 block:

```
{% set ns = namespace (dev_state = "disable") %}

{% for portname, portlist in topo[inventory_hostname].iteritems() %}
  {% if portlist.state == "enable" %}
    {% set ns.dev_state = "enable" %}
  {% endif %}
{% endfor %}

status: {{ns.dev_state|default("enable")}}
```

Use IPaddr within Jinja2 template

Assuming following `yaml` definition:

```
id: 10
underlay:
  networks:
    loopbacks: 10.0.0.0/16
```

Jinja gives option to build IP address within `loopback` network with following syntax where `id` is the idth in the network:

```
loopback_ip: {{ underlay.networks.loopbacks | ipaddr(id) | ipaddr('address') }}
```

Another example:

`{{ tenant.interconnect_prefix | ipaddr(6)}}` combined with `interconnect_prefix: 172.25.10.16/28` resulst in `172.25.10.22`

Manage Jinja2 rendering indentation

Jinja2 can manage whitespace and tabular indentation with `lstrip_blocks` and `trim_blocks` options:

- `trim_blocks`: If this is set to `True` the first newline after a block is removed (block, not variable tag!). Defaults to `False`.
- `lstrip_blocks`: If this is set to `True` leading spaces and tabs are stripped from the start of a line to a block. Defaults to `False`.

To manage these options, just put this line in your template:

```
#jinja2: lstrip_blocks: "True (or False)", trim_blocks: "True (or False)"
...
```

Example

Using this template:

```
{% for host in groups['webservers'] %}
    {% if inventory_hostname in hostvars[host]['ansible_fqdn'] %}
{{ hostvars[host]['ansible_default_ipv4']['address'] }} {{ hostvars[host]['ansible_
↪fqdn'] }} {{ hostvars[host]['inventory_hostname'] }} MYSELF
    {% else %}
{{ hostvars[host]['ansible_default_ipv4']['address'] }} {{ hostvars[host]['ansible_
↪fqdn'] }} jcs-server{{ loop.index }} {{ hostvars[host]['inventory_hostname'] }}
    {% endif %}
{% endfor %}
```

lstrip_block=False

Rendering is similar to :

```
    172.16.25.1 spine1
                172.16.25.3 spine2
                172.16.25.4 spine3
```

lstrip_block=true

Rendering should be:

```
172.16.25.1 spine1
172.16.25.3 spine2
172.16.25.4 spine3
```

Loop management

Jinja2 has built-in option to manage loop information:

`loop.index`: The current iteration of the loop. (1 indexed) `loop.index0`: The current iteration of the loop. (0 indexed) `loop.revindex`: The number of iterations from the end of the loop (1 indexed) `loop.revindex0`: The number of iterations from the end of the loop (0 indexed) `loop.first`: True if first iteration. `loop.last`: True if last iteration. `loop.length`: The number of items in the sequence. `loop.cycle`: A helper function to cycle between a list of sequences. See the explanation below. `loop.depth`: Indicates how deep in deep in a recursive loop the rendering currently is. Starts at level 1 `loop.depth0`: Indicates how deep in deep in a recursive loop the rendering currently is. Starts at level 0

CHAPTER 3

Indices and tables

- search