cnaas-nms

Release 0.1.0

Johan Marcusson, Kristofer Hallin

Jan 12, 2023

CONTENTS

Campus Network-as-a-Service - Network Management System (CNaaS-NMS) is a software to automate the management of campus networks, developed as part of the SUNET CNaaS service.

CHAPTER

ONE

CONTENTS

1.1 Tutorials

Tutorials to help you get started with using and understanding CNaaS

1.1.1 Beginner tutorial for CNaaS-NMS

CNaaS-NMS is a hybrid infrastructure-as-code (IaC) and API driven automation system. The IaC part is managed by textfiles (YAML, Jinja2) in a version controlled repository (Git), and the API part is managed using a JSON REST-like API. In this beginner tutorial we will learn how to manage both parts.

Repositories

There are three main repositories used by each CNaaS-NMS installation:

- templates
- settings
- etc

These repositories store configuration used to manage the nework. Changes to the network is mostly done by updating configuration files in any of these repositories.

REST API

The API runs on the container called cnaas/api and is accessible via HTTPS on port 443. The API is used to get information on running jobs and device status, as well as allowing minor changes to the edge of the network that should be able to be performed by an IT-helpdesk for example, like changing port VLANs on access switches.

Runtime environment

The components of CNaaS-NMS are executed in separate Docker containers. A docker-compose file is used to set up the entire collection of containers in one step.

Network setup

Each container exposes one or several listening ports to the outside, some ports are only used between different containers (like the databases) and some ports needs to be accessible from the world outside the docker host.

External listening ports:

- API exposes port tcp/443 for HTTPS interface to the API
- DHCPd exposes port udp/69 for DHCP requests during ZTP of managed switches
- HTTP exposes port tcp/80 for ZTP config and firmware downloads from managed switches

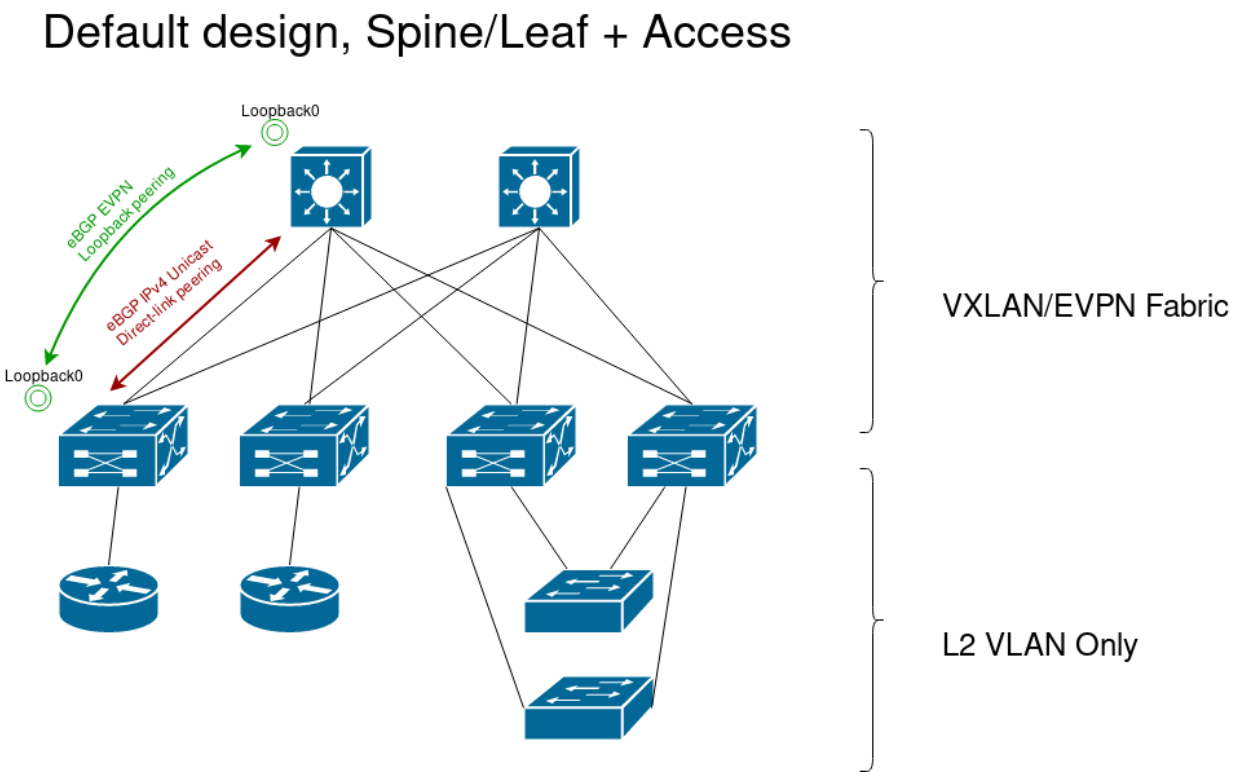
The API container also needs access to the managed network devices using the prefered mangement interface for each platform, like SSH or Netconf.

Management network

CNaaS expects to use a separate VLAN for ZTP boot (probably untagged/VLAN1) and additional tagged management VLANs for use when the switch has booted up with the CNaaS managed configuration. DHCP requests on the ZTP VLAN must be forwarded to the docker host using DHCP relay (ip helper). For the management VLANs static IP-addresses are automatically assigned by CNaaS. For access switches one management VLAN is created for each pair of distribution switches (called one management domain). Distribution and core switches are managed via loopback interfaces.

VXLAN/EVPN Fabric

CNaaS is currently focused on building a campus network using a VXLAN/EVPN overlay, but it might work in other general cases as well. This section will discuss the network architecture of the proposed VXLAN/EVPN fabric layer consisting of core and distribution layer switches. In our case a core switch is the same as a spine switch would be when talking about a normal datacenter architecture, and a distribution switch in our design is basically the same as a leaf switch except that we will put a layer of access switches that does not participate in the VXLAN/EVPN fabric under the leaf layer.



The top layer consists of the core (or spine) switches, each of the core switches will connect to all of the distribution switches forming a full mesh between core and distribution. Core switches does not have any other connections, no connections between two core switches and no external connections.

The distribution switches (leaf switches) will have the VXLAN VTEPs (Virtual Tunnel Endpoint). This is where traffic gets encapsulated into VXLAN packets to be sent over the fabric. The distribution switches is also where routing between different end user IP networks will happen.

All links between core and distrubiton switches are routed links with /31 linknets.

The access switches does not participate in the VXLAN fabric and instead use only 802.1Q VLAN tagging to separate different traffic. The access switches are connected to the distribution layer using LACP link aggregation to connect to two different distribution switches, and the distribution layer swiches will use EVPN ESI to connect a pair of distribution switches and make them act as a single device when talking to the access switch.

1.1.2 Change workflow in CNaaS-NMS

Changing the active configuration of devices managed by CNaaS involves several steps. It might seem complicated if you just need to make a single small change, but it's a very powerful tool for keeping the entire network synchronized to your ideal configuration state. Hopefully our web frontend will help make things easier by providing a workflow system for common tasks.

The are five basic steps to commiting a change to the network:

1. Update git repository and commit/push
2. Ask CNaaS to pull latest git repo change (refresh)
3. Do a dry_run to get a diff for all affected devices
4. Look through and verify diff outputs
5. If everything looks good, commit configuration to devices

Client facing interfaces on access switches can be changed in a web interface without the need for using Git to make it easier for helpdesk/servicedesk/NOC etc to make smaller changes. If a change only impacts one device and has a very low "change impact score" (more on this later) you can also simplify the workflow above and skip the steps of manual verification of the diff and manual push to commit configuration to devices. In this case, the workflow could look like this:

1. Update port setting in web frontend and click "save"
2. CNaaS does dry run and calculates "change impact score". If score is less than 10 a live run will automatically be triggered to run immediately after.

Change impact score

The change impact score is an attempt from the CNaaS-NMS software to estimate how risky a particular change (sync) operation might be. It's currently very primitive and has no deeper intelligence, it just calculates number of lines changed and does text pattern matching to try and identify important lines in the configuration. The score calculated for a particular sync job is between 1 and 100 where 1 signifies a very small risk and 100 a very big risk. Configuration lines relating to the description of an interface has a modifier to make it have a very low score, and configuration lines that will remove an IP-address from an interface for example will get a much higher score. Exactly how the score is calculated and what text patterns are being searched for can be found in the file changescore.py in the source code.

1.2 Howtos

1.2.1 Update settings for devices

Clone your settings repository in to a local directory. In this example we will use the CNaaS provided example setting repository from github:

```
git clone https://github.com/SUNET/cnaas-nms-settings
cd cnaas-nms-settings
vim access/base_system.yml
<do some changes, save file>
git commit -a -m "Updated setting for XYZ for access devices"
git push
```

Tell the NMS API to fetch latest updates from the settings repo and try a sync to devices with dry_run to preview changes:

```
curl https://localhost/api/v1.0/repository/settings -d '{"action": "refresh"}' -X PUT␣
↪-H "Content-Type: application/json"
curl https://localhost/api/v1.0/device_syncto -d '{"hostname": "ex2300-top", "dry_run
↪": true}' -X POST -H "Content-Type: application/json"
curl https://localhost/api/v1.0/jobs?per_page=1&sort=-id
```

The API call to device_syncto will start a job running in the background on the API server. To show the progress/output of the job run the last command (/job) until you get a finished result.

1.2.2 Zero-touch provisioning of access switch

Power on the switch with a blank configuration and wait for it to boot, during boot with a blank configuration it should ask for a DHCP on the native/untagged VLAN. This DHCP request should then be forwarded to the server running the CNaaS-NMS dhcpd container. The DHCP server will offer a lease to known vendor devices and in the DHCP offer give them a path for a configuration file to download. After the DHCP lease has been accepted the DHCP server will also trigger the creation of a new device in the CNaaS-NMS database which will save information on what MAC address the DHCP request was sent from, what IP address was offered to the device, and set the state of the device to DHCP_BOOT. The device should then download the configuration file specified from the DHCP server, this configuration adds a dummy user account that will be used to further configure the device. It will also ask the device to continue using DHCP to aquire an IP address. When the device applies this new configuration a new DHCP request will be sent to the DHCP server, and the DHCP server will at that point trigger the CNaaS-NMS API to schedule a job to scan or "discover" the new device. If the API can successfully reach the device it's serial number and other data will be saved into the database and the state of the device will be updated to "DISCOVERED". Now an operator can choose to continue to provision the device.

List new devices that has booted using ZTP and reached the DISCOVERED state:

```
curl https://localhost/api/v1.0/devices?filter[state]=DISCOVERED
```

Example output:


```
{
  "status": "success",
  "data": {
    "devices": [
      {
        "id": 45,
        "hostname": "mac-B8C253EA5D52",
        "site_id": null,
        "description": null,
        "management_ip": null,
        "dhcp_ip": "192.168.0.240",
        "infra_ip": null,
        "oob_ip": null,
        "serial": "JW0218490737",
        "ztp_mac": "B8C253EA5D52",
        "platform": "junos",
        "vendor": "Juniper",
        "model": "EX2300-48P",
        "os_version": "18.4R1-S2.4",
        "synchronized": false,
        "state": "DISCOVERED",
        "device_type": "UNKNOWN",
        "confhash": null,
        "last_seen": "2019-09-20 10:33:44.265137",
        "port": null
      }
    ]
  }
}
```


If the device serial/MAC matches with a device you want to provision, call the API to initialize the device with a specified hostname and device type:

```
curl https://localhost/api/v1.0/device_init/45 -d '{"hostname": "ex2300-top", "device_
↪type": "ACCESS"}' -X POST -H "Content-Type: application/json"
```

Check job status to see progress, there should be two jobs running after each other, step1 and step2:

```
curl https://localhost/api/v1.0/jobs?per_page=1&sort=-id
```

The first job will send a new base configuration to the device, this will move the device management to a tagged VLAN and set a static IP address amongst other things. The device will now move to state INIT. After this change the TCP connection to the device will get disconnected (since the IP was changed), this is expected. A new job (step2) will be scheduled to run one minute later, this step2 job will try to log in to the device using the new IP address and verify that the device accepted the new configuration. If everything looks OK the device will move to the state MANAGED. If you have any plugins registered they will execute the "new_managed_device" hook that can be used to add the device to monitoring systems etc at this point.

To debug this process it can be helpful to tail the logs from the DHCPd container at the initial steps of the process, and also logs from the API container at later stages of the process. If the device gets stuck in the DHCP_BOOT process for example, it probably means the API can not log in to the device using the credentials and IP address saved in the database. The API will retry connecting to the device 3 times with increasing delay between each attempt. If you want to trigger more retries at a later point you can manually call the discover_device API call and send the MAC and DHCP IP of the device. New attempts to discover the device will also be made when the DHCP lease is renewed or reaquired.

1.2.3 Zero-touch provisioning of fabric switch

You can also provision a new switch to be part of the (EVPN/VXLAN) fabric, that is a switch with device_type DIST or CORE. Interfaces that connects between CORE and DIST devices should be configured as ifclass "fabric" on both ends. You can configure this in the settings repository via a device specific setting or via a model specific setting (model setting might be preferable for ZTP since you don't need to pre-provision new device hostnames in the settings repository).

To verify that interfaces are configured correctly and that LLDP neighbors are seen you can use the device_initcheck API call (see devices API reference):

```
curl https://localhost/api/v1.0/device_initcheck/45 -d '{"hostname": "dist3", "device_
↪type": "DIST"}' -X POST -H "Content-Type: application/json"
```

If all parameters are compatible you can start initialization:

```
curl https://localhost/api/v1.0/device_init/45 -d '{"hostname": "dist3", "device_type
↪": "DIST"}' -X POST -H "Content-Type: application/json"
```

If LLDP neighbors are not seen or are not of the expected type (DIST type expect neighbors of type CORE and vice versa) you can manually specify the neighbors you want to verify connectivity to, but make sure you know what you are doing and maybe set up console access if something goes wrong here:

```
curl https://localhost/api/v1.0/device_init/45 -d '{"hostname": "dist3", "device_type
↪": "DIST", "neighbors": ["dist1", "dist2"]}' -X POST -H "Content-Type: application/
↪json"
```

If you don't expect to see any LLDP neighbors at all and instead have pre-configured some kind of uplink interfaces via ifclass custom interfaces in settings for this device, you could also specify an empty list as neighbors and in this case init will continue even if no LLDP neighbors were detected. This is also very risky since you can't verify that interfaces are connected correctly before sending configuration and possibly losing connectivity to the device.

1.3 API Reference

1.3.1 Devices

The API is used to manage devices, interfaces and other objects used by the CNaaS NMS. All requests in the examples below are using 'curl'.

Show devices

A single device entry can be listed by device_id or hostname:

```
curl https://hostname/api/v1.0/device/9
```

This will return the entire device entry from the database:


```
{
    "status": "success",
    "data": {
        "devices": [
            {
                "id": 9,
                "hostname": "eosdist",
                "site_id": null,
                "description": null,
                "management_ip": "10.100.3.101",
                "dhcp_ip": null,
                "infra_ip": null,
                "oob_ip": null,
                "serial": null,
                "ztp_mac": "08002708a8be",
                "platform": "eos",
                "vendor": null,
                "model": null,
                "os_version": null,
                "synchronized": true,
                "state": "MANAGED",
                "device_type": "DIST",
                "confhash": null,
                "last_seen": "2019-02-27 10:30:23.338681",
                "port": null
            }
        ]
    }
}
```


To list all devices the following API call can be used:

```
curl https://hostname/api/v1.0/devices
```

You can also do filtering, ordering and limiting of results from the devices API:

```
curl "https://hostname/api/v1.0/devices?filter[hostname][contains]=eos&filter[device.
↪type]=dist&page=2&per_page=50&sort=-hostname"
```

This will filter the results like so:

- Only devices that has a hostname that contains the string "eos" will be returned
- Only devices that has type exactly matching "dist" will be returned
- A maximum of 50 results will be returned (per_page=50)
- The second page of results will be returned, since per_page is set to 50 this means items 51-100 (page=2)
- The results will be ordered based on the column hostname, in descending order. "-" means descending, no prefix means ascending (sort=-hostname)

A HTTP header with the name X-Total-Count will show the unfiltered total number of devices in the database.

Add devices

A single device can be added by sending a REST call with a JSON strcuture describing the device as data. The JSON strcuture should have the following format:

- hostname (mandatory)
- site_id (optional)
- site (optional)
- description (optional)
- management_ip (optional)
- infra_ip (optional)
- dhcp_ip (optional)
- serial (optional)
- ztp_mac (optional)
- platform (mandatory)
- vendor (optional)
- model (optional)
- os_version (optional)
- synchronized (optional)
- state (mandatory)
- device_type (mandatory)

There are four mandatory fields that can not be left out: hostname, state, platform and device_type.

Device state can be one of the following:

```
UNKNOWN:          Unhandled programming error
PRE_CONFIGURED:   Pre-populated, not seen yet
DHCP_BOOT:        Something booted via DHCP, unknown device
DISCOVERED:       Something booted with base config, temp ssh access for conf push
INIT:             Moving to management VLAN, applying base template
MANAGED:          Correct managament and accessible via conf push
MANAGED_NOIF:     Only base system template managed, no interfaces?
UNMANAGED:        Device no longer maintained by conf push
```

The mandatory field device_type can be:

- UNKNOWN

- ACCESS
- DIST
- CORE

If you specify a device_type of CORE or DIST but do not specify management_ip or infra_ip these will be selected automatically from the next available IP from the network specified in the settings repository.

Example CURL call:

```
curl -H "Content-Type: application/json" -X POST -d
'{"hostname":"foo", "state":"UNKNOWN", "device_type":"DIST", "platform": "eos"}'
https://hostname/api/v1.0/device
```

Modify devices

An existing device can be modified, in that case the devices ID should be appended to the URL. The URL will then have the following format:

```
https://hostname/api/v1.0/device/10
```

Where 10 is the device ID.

To modify a device, use the same JSON data as for adding new devices:

```
curl -H "Content-Type: application/json" -X PUT -d
'{"state": "UNMANAGED", "device_type": "DIST"}'
https://hostname/api/v1.0/device/10
```

Warning: changing of management_ip or infra_ip can result in unreachable devices that is not recoverable via API! Changing of hostname is possible but a resync of all neighbor devices will be needed.

Remove devices

To remove a device, pass the device ID in a DELTE call:

```
curl -X DELETE https://hostname/api/v1.0/device/10
```

There is also the option to factory default and reboot the device when removing it. This can be done like this:

```
curl -H "Content-Type: application/json" -X DELETE -d
'{"factory_default": true}' https://hostname/api/v1.0/device/10
```

Preview config

To preview what config would be generated for a device without actually touching the device use generate_config:

```
curl https://hostname/api/v1.0/device/<device_hostname>/generate_config
```

This will return both the generated configuration based on the template for this device type, and also a list of available vaiables that could be used in the template.

View previous config

You can also view previous versions of the configuration for a device. All previous configurations are saved in the job database and can be found using either a specific Job ID (using job_id=), a number of steps to walk backward to find a previous configuration (previous=), or using a date to find the last configuration applied to the device before that date.

```
curl "https://hostname/api/v1.0/device/<device_hostname>/previous_config?before=2020-
↪04-07T12:03:05"

curl "https://hostname/api/v1.0/device/<device_hostname>/previous_config?previous=1"

curl "https://hostname/api/v1.0/device/<device_hostname>/previous_config?job_id=12"
```

If you want to restore a device to a previous configuration you can send a POST:

```
curl "https://hostname/api/v1.0/device/<device_hostname>/previous_config" -X POST -d '
↪{"job_id": 12, "dry_run": true}' -H "Content-Type: application/json"
```

When sending a POST you must specify an exact job_id to restore. The job must have finished with a successful status for the specified device. The device will change to UNMANAGED state since it's no longer in sync with current templates and settings.

Apply static config

You can also test a static configuration specified in the API call directly instead of generating the configuration via templates and settings. This can be useful when developing new templates (see template_dry_run.py tool) when you don't wish to do the commit/push/refresh/sync workflow for every iteration. By default only dry_run are allowed, but you can configure api.yml to allow apply config live run as well.

```
curl "https://hostname/api/v1.0/device/<device_hostname>/apply_config" -X POST -d '{
↪"full_config": "hostname eosdist1\n...", "dry_run": True}' -H "Content-Type:␣
↪application/json"
```

This will schedule a job to send the configuration to the device.

Initialize check

Before initializing a new device you can run a pre-check API call. This will perform some basic device state checks and check that compatible LLDP neighbors are found. For access devices it will try and find a compatible mgmtdomain and for core/dist devices it will check that interfaces facing neighbors are set to the correct ifclass. It is possible that the init will fail even if the initcheck passed.

To test if a device is compatible for DIST ZTP run:

```
curl https://localhost/api/v1.0/device_initcheck/45 -d '{"hostname": "dist3", "device_
↪type": "DIST"}' -X POST -H "Content-Type: application/json"
```

Example output:

```
{
    "status": "success",
    "data": {
        "linknets": [
            {
                "description": null,
```

(continues on next page)

(continued from previous page)


```
                "device_a_hostname": "dist3",
                "device_a_ip": "10.198.0.0",
                "device_a_port": "Ethernet3",
                "device_b_hostname": "core1",
                "device_b_ip": "10.198.0.1",
                "device_b_port": "Ethernet3",
                "ipv4_network": "10.198.0.0/31",
                "site_id": null
            }
        ],
        "linknets_compatible": true,
        "neighbors_compatible": false,
        "neighbors_error": "Not enough linknets (1 of 2) were detected",
        "parsed_args": {
            "device_id": 2,
            "new_hostname": "dist3",
            "device_type": "DIST",
            "neighbors": null
        },
        "compatible": false
    }
}
```


Status success in this case means all checks were able to complete, but if you check the "compatible" key it says false which means this device is actually not compatible for DIST ZTP at the moment. We did find a compatible linknet, but there were not enough neighboring devices of the correct device type found. If you want to perform some non-standard configuration like trying ZTP with just one neighbor you can manually specify what neighbors you expect to see instead ("neighbors": ["core1"]). Other arguments that can be passed to device_init should also be valid here, like "mlag_peer_id" and "mlag_peer_hostname" for access MLAG pairs.

If the checks can not be performed at all, like when the device is not found or an invalid device type is specified the API call will return a 400 or 500 error instead.

Initialize device

For a more detailed explanation see documentation under Howto *Zero-touch provisioning of access switch*.

To initialize a single ACCESS type device:

```
curl https://localhost/api/v1.0/device_init/45 -d '{"hostname": "ex2300-top", "device_
↪type": "ACCESS"}' -X POST -H "Content-Type: application/json"
```

The device must be in state DISCOVERED to start initialization. The device must be able to detect compatible uplink devices via LLDP for initialization to finish.

To initialize a pair of ACCESS devices as an MLAG pair:

```
curl https://localhost/api/v1.0/device_init/45 -d '{"hostname": "a1", "device_type":
↪"ACCESS", "mlag_peer_id": 46, "mlag_peer_hostname": "a2"}' -X POST -H "Content-
↪Type: application/json"
```

For MLAG pairs the devices must be able to dectect it's peer via LLDP neighbors and compatible uplink devices for initialization to finish.

Update facts

To update the facts about a device (serial number, vendor, model and OS version) use this API call:

```
curl https://localhost/api/v1.0/device_update_facts -d '{"hostname": "eosdist1"}' -X␣
↪POST -H "Content-Type: application/json"
```

This will schedule a job to log in to the device, get the facts and update the database. You can perform this action on both MANAGED and UNMANAGED devices. UNMANAGED devices might not be reachable so this could be a good test-call before moving the device back to the MANAGED state.

Update interfaces

To update the list of available interfaces on an ACCESS device use this API call:

```
curl https://localhost/api/v1.0/device_update_interfaces -d '{"hostname": "eosaccess"}
↪' -X POST -H "Content-Type: application/json"
```

This will schedule a job to log in to the device and get a list of physical interfaces and put them in the interface database. Existing interfaces will not be changed unless you specify "replace": true. Interfaces that no longer exists on the device will be deleted from the interface database, except for UPLINK and MLAG_PEER ports which will not be deleted automatically. If you specify "delete_all": true then all interfaces will be removed, including UPLINK and MLAG_PEER ports (dangerous!). If you want to re-populate MLAG_PEER ports you have to specify the argument "mlag_peer_hostname" to indicate what peer device you expect to see.

Renew certificates

To manually request installation/renewal of a new device certificate use the device_cert API:

```
curl https://localhost/api/v1.0/device_cert -d '{"hostname": "eosdist1", "action":
↪"RENEW"}' -X POST -H "Content-Type: application/json"
```

This will schedule a job to generate a new key and certificate for the specified device(s) and copy them to the device(s). The certificate will be signed by the NMS CA (specified in api.yml).

Either one of "hostname" or "group" arguments must be specified. The "action" argument must be specified and the only valid action for now is "RENEW".

1.3.2 Groups

This API is used to list groups. Groups are configured in the settings and can be defined to include one or more devices.

Show groups

To show all groups the following REST call can be used:

```
curl https://hostname/api/v1.0/groups
```

That will return a JSON structure with all group names and the hostnames of all devices in the group:

```
{
  "status": "success",
  "data": {
      "groups": {
          "group_0": [
              "testdevice_a",
          ],
          "group_1": [
              "testdevice_b",
              "testdevice_c"
          ]
      }
   }
}
```

Show specific group

To show a single group specify the group name in the path:

```
curl https://hostname/api/v1.0/groups/mygroup
```

Show specific group OS versions

To show the OS versions of the devices in a group:

```
curl https://hostname/api/v1.0/groups/MY_EOS_DEVICES/os_versions
```

Output:

```
{
    "status": "success",
    "data": {
        "groups": {
            "MY_EOS_DEVICES": {
                "4.21.1.1F-10146868.42111F": [
                    "eosaccess"
                ],
                "4.22.3M-14418192.4223M": [
                    "eosdist1",
                    "eosdist2"
                ]
            }
        }
    }
}
```

Define groups

New groups can be defined in the settings repository. *settings*

1.3.3 Jobs

The jobs API can retreive information about jobs in CNaaS-NMS, any task that takes more than a few seconds to complete or needs specific scheduling requirements will be executed as a job in CNaaS-NMS. A job will always be in one of these states:

- SCHEDULED: The job has been scheduled to run at a later time
- RUNNING: The job is currently running
- FINISHED: The job has completed, all parts were executed but some parts might have errors
- EXCEPTION: The job has stopped with an error/exception, all parts might not have been executed

Jobs can only be retreived using this API, they can not be started. Jobs will automatically be started when using for example the device_syncto API.

List jobs

To fetch information about all jobs:

```
curl http://hostname/api/v1.0/jobs
```

The number of jobs to be retreived can be limited by using the pagination system, like this:

```
curl http://hostname/api/v1.0/jobs?per_page=50&page=2
```

The result will look like this:

```
{
  "status": "success",
  "data": {
    "jobs": [
      {
        "id": 101,
        "status": "FINISHED",
        "scheduled_time": "2019-12-05T13:06:03.319761",
        "start_time": "2019-12-05T13:06:03.375200",
        "finish_time": "2019-12-05T13:06:05.775562",
        "function_name": "sync_devices",
        "scheduled_by": null,
        "comment": null,
        "ticket_ref": null,
        "next_job_id": null,
        "result": {
          "devices": {
            "eosdist": {
              "failed": false,
              "job_tasks": [
                {
                  "diff": "",
                  "failed": false,
                  "result": null,
```

(continues on next page)

(continued from previous page)


```
                    "task_name": "push_sync_device"
                  },
                  {
                    "diff": "",
                    "failed": false,
                    "result": "hostname eosdist\n...",
                    "task_name": "Generate device config"
                  },
                  {
                    "diff": "@@ -16,8 +16,6 @@\n +hostname eosdist\n...",
                    "failed": false,
                    "result": null,
                    "task_name": "Sync device config"
                  }
                ]
              }
            }
          },
          "exception": null,
          "finished_devices": [
            "eosdist"
          ],
          "change_score": 21
        }
      ]
    }
}
```


It is possible to filter out specific jobs by using query parameters in the same way as the devices API. In addition to filtering out entries you can also filter out parts of the job result for "syncto" type jobs, if you are only interested in the diffs and don't want to see the full generated config for example:

```
curl "http://hostname/api/v1.0/jobs?filter\[function.name\]\[contains\]=sync&filter_
↪jobresult=config"
```

The finished_devices attribute will be populated as devices are finishing. This value will only be updated for every other second to not keep the database too busy.

It's also possible to query a single job by job ID:

```
curl http://hostname/api/v1.0/job/5
```

Locks

Some jobs running in CNaaS will require a lock to make sure that jobs are not interfering with each other. For example, only a single syncto job should be running at the same time or things might break in unexpected ways. To keep track of who is currently holding the lock for a particular feature a record is kept in the database. If something unexpected happens this lock might need to be manually cleared.

List current locks:

```
curl http://hostname/api/v1.0/joblocks
```

Manually clear/delete a lock (make sure that no jobs are running first):

```
curl http://hostname/api/v1.0/joblocks -X DELETE -d '{"name": "devices"}' -H "Content-
↪Type: application/json"
```

1.3.4 Links

This API is used to retreive and modify information about links between devices. Links between access switches and distribution switches are normally Layer2 ports and thus have no IP information, while links between dist and core devices are Layer3 ports and have information about IP addressing.

An access switch normally has two uplink ports connected to two different distribution switches like in the example below.

Show links

To get information about existing links in the database, use a GET query to api/v1.0/linknets

This can be done using CURL:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/linknets
```

The result will look like this:


```
{
    "status": "success",
    "data": {
        "linknets": [
            {
                "id": 8,
                "ipv4_network": null,
                "device_a_id": 13,
                "device_a_ip": null,
                "device_a_port": "Ethernet1",
                "device_b_id": 9,
                "device_b_ip": null,
                "device_b_port": "Ethernet1",
                "site_id": null,
                "description": null
            },
            {
                "id": 10,
                "ipv4_network": null,
                "device_a_id": 13,
                "device_a_ip": null,
                "device_a_port": "Ethernet2",
                "device_b_id": 12,
                "device_b_ip": null,
                "device_b_port": "Ethernet2",
                "site_id": null,
                "description": null
            }
        ]
    }
}
```


In the result above device_a is the access switch and device_b is the dist switch.

Manually provision a new link

When adding switches manually instead of using ZTP you might also need to manually create linknets.

Example:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/linknets -X␣
↪POST -d '{"device_a": "eosdist", "device_a_port": "Ethernet3", "device_b": "eosdist2
↪", "device_b_port": "Ethernet3"}' -H "Content-Type: application/json"
```

Output:

```
{
    "status": "success",
    "data": {
        "id": 25,
        "ipv4_network": "10.198.0.0/31",
        "device_a_id": 9,
        "device_a_ip": "10.198.0.0",
        "device_a_port": "Ethernet3",
        "device_b_id": 12,
        "device_b_ip": "10.198.0.1",
        "device_b_port": "Ethernet3",
        "site_id": null,
        "description": null
    }
}
```

A new IP subnet is automatically allocated as the next free /31 from the settings variable underlay->infra_link_net

Manually deleting a link

If you made some error while manually adding a link you can delete it and recreate it. Use this carefully since it might affect reachability for the entire fabric.

Example:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/linknets -X␣
↪DELETE -d '{"id": 25}' -H "Content-Type: application/json"
```

1.3.5 Management domains

Management domain can be retreived, added, updated an removed using this API.

Get all managment domains

All management domains can be listed using CURL:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/mgmtdomains
```

That will return a JSON structured response which describes all domains available:

```
{
    "status": "success",
    "data": {
        "mgmtdomains": [
            {
                "id": 10,
                "ipv4_gw": "10.0.6.1/24",
                "device_a_id": 9,
                "device_a_ip": null,
                "device_b_id": 12,
                "device_b_ip": null,
                "site_id": null,
                "vlan": 600,
                "description": null,
                "esi_mac": null,
                "device_a": "eosdist",
                "device_b": "eosdist2"
            }
        ]
    }
}
```


Note that some of these fields does not have a use case (yet).

You can also specify one specifc mgmtdomain to query by using:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/mgmtdomain/10
```

Add management domain

To add a new management domain we can to call the API with a few fields set in a JSON structure:

```
* ipv4_gw (mandatory): The IPv4 gateway to be used, should be expressed with a prefix␣
↪(10.0.0.1/24)
* device_a (mandatory): Hostname of the first device
* device_b (mandatory): Hostname of the second device
* vlan (mandatory): A VLAN ID
```

device_a and device_b should be a pair of DIST devices that are connected to a specific set of access devices that should share the same management network. It's also possible to specify two CORE devices if there is a need to have the gateway/routing for all access switch management done in the CORE layer instead. In the case where two CORE devices are specified there should only be one single mgmtdomain defined for the entire NMS, and this mgmtdomain can only contain exactly two CORE devices even if there are more CORE devices in the network.

Example using CURL:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/mgmtdomains -
↪H "Content-Type: application/json" -X POST -d '{"ipv4_gw": "10.0.6.1/24", "device_a
↪": "dist1", "device_b": "dist2", "vlan": 600}'
```

Update management domain

To update a doamin, we can send a PUT request and specify its ID. We will have to send the same JSON structure as when adding a new domain:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/mgmtdomain/4 -
↪H "Content-Type: application/json" -X PUT -d '{"ipv4_gw": "10.0.6.1/24", "device_a
↪": "dist1", "device_b": "dist2", "vlan": 600}'
```

Remove management domain

We can also remove an existing domain by sending a DELETE request and specify its ID:

Once again using CURL:

```
curl -s -H "Authorization: Bearer $JWT_AUTH_TOKEN" ${CNAASURL}/api/v1.0/mgmtdomain/4 -
↪X DELETE
```

1.3.6 Repository

This API can be used to retrevie information about repositorys and also to update them.

It supports GET and PUT.

Get repository information

To get information about the last change to a repository we can use CURL:

```
curl -X GET http://hostname/api/v1.0/repository/settings
```

This will return information about who did the last commit to this repository:

```
{
 "status": "success",
 "data": "Commit cdf978245c3782ec391ffa2bda3ca540577ad36f master by Kristofer Hallin␣
↪at 2019-06-13 10:07:32+02:00\n"
}
```

Refresh repository

To refresh the contents of a repository we can use a PUT request. CNaaS will then update the corresponding Git repository.

```
curl -H "Content-Type: application/json" -X PUT http://hostname/api/v1.0/repository/
↪settings -d '{"action": "REFRESH"}'
```

We should then get a response back stating the last commit that was done:

```
{
 "status": "success",
 "data": "Commit cdf978245c3782ec391ffa2bda3ca540577ad36f master by Kristofer Hallin␣
↪at 2019-06-13 10:07:32+02:00\n"
}
```

1.3.7 Settings

This API can be used to retreveive all settings defied in the settings repository.

We can sort settings per hostname or device type.

To get all settings:

```
curl https://hostname/api/v1.0/settings
```

This will return all settings for all devices:


```
{
 "status": "success",
 "data": {
     "settings": {
         "ntp_servers": [
             {
                 "host": "194.58.202.148"
             },
             {
                 "host": "256.256.256.256"
             }
         ],
         "radius_servers": [
             {
                 "host": "10.100.2.3"
             }
         ],
         "syslog_servers": null
     },
     "settings_origin": {
         "ntp_servers": "device",
         "radius_servers": "devicetype",
         "groups": "global"
     }
   }
}
```


We can also chose to only get settings for a specific hostname:

```
curl https://hostname/api/v1.0/settings?hostname=eosaccess
```

Or by a specific type of devices:

```
curl https://hostname/api/v1.0/settings?device_type=access
```

Settings model

You can also retrieve the model used by the currently running version of CNaaS-NMS to verify the settings. This might help you understand why certain settings are not considered valid.

Example:

```
curl https://hostname/api/v1.0/settings/model
```

$ref fields mean that the definition of this "field" is set somewhere else in this output. Look under the "definitions" part. Patterns are Python regular expressions.

You can also test out specific settings by sending them to the API with a POST call like this:

```
curl https://hostname/api/v1.0/settings/model -X POST -d '{"radius_servers": [{"host
↪": "1.1.1.1"}]}' -H "Content-Type: application/json"
```

Example with invalid IP/hostname:

```
curl https://hostname/api/v1.0/settings/model -X POST -d '{"radius_servers": [{"host
↪": "10.0.0.500"}]}' -H "Content-Type: application/json"
```

Output:

```
{
  "status": "error",
  "message": "Validation error for setting radius_servers->0->host, bad value: 10.0.0.
↪500 (value origin: API POST data)\nMessage: string does not match regex \"^((?:(?
↪:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?
↪)|([a-z0-9-]{1,63}\\.)([a-z-][a-z0-9-]{1,62}\\.?)+)$\", field should be: Hostname,␣
↪FQDN or IP address\n"
}
```

1.3.8 Plugins

This API can be used to retrevie information about what plugins are currently used.

It supports GET and PUT.

Get repository information

To get information about currently loaded plugins and available plugin variables:

```
curl -X GET http://hostname/api/v1.0/plugins
```

Example output:

```
{
    "status": "success",
    "data": {
        "loaded_plugins": [
            "cnaas_nms.plugins.filewriter"
        ],
        "plugindata": {
            "plugins": [
                {
                    "filename": "filewriter.py",
                    "vars": {
                        "logfile": "/tmp/filewriter.log"
                    }
                }
            ]
        }
    }
}
```

Run plugin selftests

Plugins can define a selftest function to test that it can access it's system API etc. This can be tested by calling PUT on the plugins url.

```
curl -H "Content-Type: application/json" -X PUT http://hostname/api/v1.0/plugins -d '{
↪"action": "selftest"}'
```

We should then get a response back with a list of return values:

```
{
    "status": "success",
    "data": {
        "result": [
            true
        ]
    }
}
```

1.3.9 Sync to device

To make the network devices synchronize their configuration to the latest version generated by CNaaS the device_syncto API call is used. This will push the latest configuration to the devices you select.

You can choose to either synchronize all devices, or just synchronize a specific type of device, or synchronize just a specific hostname.

Example API call:

```
curl https://hostname/api/v1.0/device_syncto -d '{"hostname": "eosdist", "dry_run":␣
↪true}'
-X POST -H "Content-Type: application/json"
```

This will start a "dry run" synchronization job for the device called "eosdist". A dry run job will send the newly generated configuration to the device and then generate a diff to see what lines would have been changed. The response from this API call is a reference to a job id that you can poll using the job API. This is to make sure that long running jobs does not block the client. If you synchronize many devices at the same time the job can take a very long time and the client might time out otherwise.

Example response:

```
{
  "status": "success",
  "data": "Scheduled job to synchronize eosdist",
  "job_id": "5d5aa787ba050d5fd085f1ce"
}
```

The status success in this case only means that the job was scheduled successfully, but you have to poll the job API to see that result of what was done, the job itself might still fail.

Arguments:

- hostname: Optional, the hostname of a device
- device_type: Optional, a device type (access, dist or core)
- group: Optional, a device group. A group can contain multiple devices.
- all: Optional, bool. Synchronize all devices (that are not already in sync, see resync option)
- dry_run: Dry run does not commit any configuration to the device. Boolean, defaults to true.
- force: If a device configuration has been changed outside of CNaaS the configuration hash will differ from the last known hash in the database and this will normally make CNaaS abort. If you want to override any changes made outside of CNaaS and replace them with the latest configuration from CNaaS you can set this flag to true. Boolean, defaults to false.
- auto_push: If you specify a single device by hostname and do a dry_run, setting this option will cause CNaaS to automatically push the configuration to committed/live state after doing the dry run if the change impact score (see *Change impact score*) is very low.
- resync: By default devices that are marked as synchronized in the database will not be re-synchronized, if you specify this option as true then all devices will be checked. This option does not affect syncto jobs with a specified hostname, when you select only a single device via hostname it's always re-synchronized. Defaults to false.
- comment: Optionally add a comment that is saved in the job log. This should be a string with max 255 characters.
- ticket_ref: Optionally reference a service ticket associated with this job. This should be a string with max 32 characters.

If neither hostname or device_type is specified all devices that needs to be sycnhronized will be selected.

1.3.10 Interfaces

This API is used to query and update information about physical interfaces on access switches managed by CNaaS-NMS. Interfaces for dist and core devices are managed through YAML files in the git repositories.

Show interfaces

List all interfaces on device eosaccess

```
curl https://hostname/api/v1.0/device/eosaccess/interfaces
```

The result will look something like this:

```
{
    "status": "success",
    "data": {
        "interfaces": [
            {
                "device_id": 13,
                "name": "Ethernet3",
                "configtype": "ACCESS_UPLINK",
                "data": null
            },
            {
```

(continues on next page)

(continued from previous page)


```
                "device_id": 13,
                "name": "Ethernet2",
                "configtype": "ACCESS_UPLINK",
                "data": null
            },
            {
                "device_id": 13,
                "name": "Ethernet1",
                "configtype": "ACCESS_AUTO",
                "data": null
            }
        ],
        "hostname": "eosaccess"
    }
}
```


The configtype field must use some of these pre-defined values:

- UNKNOWN: Should not be used unless there's an error in CNaaS-NMS
- UNMANAGED: This interface is not managed by CNaaS-NMS (not implemented)
- CONFIGFILE: This interface is managed via external config file (not implemented)
- CUSTOM: Use custom configuration defined in settings YAML (only implemented for DIST type devices)
- TEMPLATE: Use a pre-defined template (not implemented)
- ACCESS_AUTO: Use 802.1X configuration to automatically configure port (default)
- ACCESS_UNTAGGED: Use a static VLAN defined by name in the data field
- ACCESS_TAGGED: Use a static list of VLANs defined by names in the data field
- ACCESS_UPLINK: Uplink from access switch to dist switch
- ACCESS_DOWNLINK: Downlink from this access switch to another access switch
- MLAG_PEER: MLAG peer interface

Update interface

Set device eosaccess to use statically configured untagged VLAN with name "STUDENT" on interface Ethernet1

```
curl https://hostname/api/v1.0/device/eosaccess/interfaces -d '{"interfaces": {
↪"Ethernet1": {"configtype": "access_untagged", "data": {"untagged_vlan": "STUDENT"}}
↪}}' -X PUT -H "Content-Type: application/json"
```

Response:


```
{
    "status": "success",
    "data": {
        "updated": {
            "Ethernet1": {
                "configtype": "ACCESS_UNTAGGED",
                "data": {
                    "untagged_vlan": "STUDENT"
                }
```


(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

To change the port back to the default ACCESS_AUTO port type use:

```
curl https://hostname/api/v1.0/device/eosaccess/interfaces -d '{"interfaces": {
↪"Ethernet1": {"configtype": "access_auto"}}}' -X PUT -H "Content-Type: application/
↪json"
```

Response:

```
{
    "status": "success",
    "data": {
        "updated": {
            "Ethernet1": {
                "configtype": "ACCESS_AUTO"
            }
        }
    }
}
```

If you want to specify a statically configured port with tagged VLANs (trunk port) use an API call like this:

```
curl https://hostname/api/v1.0/device/eosaccess/interfaces -d '{"interfaces": {
↪"Ethernet1": {"configtype": "access_tagged", "data": {"tagged_vlan_list": ["STUDENTT
↪"]}}}}' -X PUT -H "Content-Type: application/json" -H "Authorization: Bearer $JWT_
↪AUTH_TOKEN"
```

Response:

```
{
    "status": "error",
    "message": {
        "errors": [
            "Some VLAN names STUDENTT are not present in eosaccess"
        ],
        "updated": {
            "Ethernet1": {
                "configtype": "ACCESS_TAGGED"
            }
        }
    }
}
```

In this case the configtype was updated but one of the names in the VLAN list was not present on this switch and therefore the VLAN list was not updated. You can check what VLAN names exist on a specific switch by using the /settings API call and specifying the hostname and then look for the vlan_name field under a specific vxlan.

Data can also contain any of these optional keys:

- description: Description for the interface, this should be a string 0-64 characters.
- enabled: Set the administrative state of the interface. Defaults to true if not set.
- aggregate_id: Identifier for configuring LACP etc. Integer value. Special value -1 means configure MLAG and use ID based on indexnum.

Setting an optional value to JSON null will remove it from the database.

To disable a port:

```
curl https://hostname/api/v1.0/device/eosaccess/interfaces -d '{"interfaces": {
↪"Ethernet1": {"data": {"enabled": false, "description": "Disabled becasue of abuse␣
↪2020-01-30 by kosmoskatten"}}}}' -X PUT -H "Content-Type: application/json" -H
↪"Authorization: Bearer $JWT_AUTH_TOKEN"
```

To re-enable and unset description:

```
curl https://hostname/api/v1.0/device/eosaccess/interfaces -d '{"interfaces": {
↪"Ethernet1": {"data": {"enabled": true, "description": null}}}}' -X PUT -H "Content-
↪Type: application/json" -H "Authorization: Bearer $JWT_AUTH_TOKEN"
```

If the list of interfaces does not match what currently exists on the device you need to run the device_update_interfaces API call (see device API).

Show interface states

To get the currently active state of interfaces on a device like admin state (is_up) etc, use:

```
curl https://hostname/api/v1.0/device/eosaccess/interface_status
```

Response:

```
{
    "status": "success",
    "data": {
        "interface_status": {
            "Management1": {
                "is_up": true,
                "is_enabled": true,
                "description": "",
                "last_flapped": 1581950162.341227,
                "speed": 1000,
                "mac_address": "08:00:27:F5:D6:58"
            }
        }
    }
}
```

Bounce interfaces

If you want to quickly disale and then re-enable an interface to reboot a PoE connected access point for example, you can use the "bounce interfaces" API. Send a list of interfaces to the specified device like this:

```
curl https://hostname/api/v1.0/device/eosaccess/interface_status -d '{"bounce_
↪interfaces": ["Ethernet1"]}' -X PUT -H "Content-Type: application/json" -H
↪"Authorization: Bearer $JWT_AUTH_TOKEN"
```

Response:

```
{
    "status": "success",
```

(continues on next page)

(continued from previous page)

```
    "data": "Bounced interfaces: Ethernet1"
}
```

You can only bounce non-uplink interfaces of ACCESS type switches. This is to prevent accidentally losing connectivity to the device.

1.3.11 Firmware

The firmware API provides an interface to download, remove and list firmware images which later can be used on managed devices.

Instead of uploading a file somewhere, you will have to tell this API where to download the file from. The API will then schedule a job and fetch the file and validate it towards the supplied SHA1 checksum.

When upgrading devices we can chose to either work on a single device or a group of devices, this is described in more detailed further down in this document.

Download firmware

To download firmware from a URL, the following method can be used:

```
curl https://hostname/api/v1.0/firmware -X POST -H "Content-Type: application/json" -
↪d '{"url": "http://remote_host/firmware.bin", "sha1":
↪"e0537400b5f134aa960603c9b715a8ce30306071", "verify_tls": false}'
```

The method will accept three attributes: url, sha1 and verify_tls:

- url: The URL to the file we should download.
- sha1: The checksum of the file.
- verify_tls: Should we validate SSL certificates or not?

That will schedule a new job which will report back the outcome of the download. The job status can be seen using the jobs API:

```
curl https://hostname/api/v1.0/jobs?limit=1
{
  "status": "success",
  "data": {
     "jobs": [
         {
             "id": "5d848e7cdd428720db72c686",
             "start_time": "2019-09-20 08:31:57.073000",
             "finish_time": "2019-09-20 08:31:58.585000",
             "status": "FINISHED",
             "function_name": "get_firmware",
             "result": "\"File downloaded from: https://remote_host/firmware.bin\"",
             "exception": null,
             "traceback": null,
             "next_job_id": null,
             "finished_devices": []
         }
     ]
   }
}
```

List firmware

The same procedure as above can be used when listing the available firmwares. The only difference is that when listing all available firmware images no job is scheuled.

To list a single firmware image:

```
curl https://hostname/api/v1.0/firmware/firmware.bin

{
  "status": "success",
  "data": "Scheduled job get firmware information",
  "job_id": "5d848f87dd428720db72c68d"
}
```

And the reponse when getting job information. Note that the result will contain the checksum of the images if it exists, otherwise an error will be given back.

```
{
 "status": "success",
 "data": {
     "jobs": [
         {
             "id": "5d848f87dd428720db72c68d",
             "start_time": "2019-09-20 08:36:24.078000",
             "finish_time": "2019-09-20 08:36:24.110000",
             "status": "FINISHED",
             "function_name": "get_firmware_chksum",
             "result": "\"e0537400b5f134aa960603c9b715a8ce30306071\"",
             "exception": null,
             "traceback": null,
             "next_job_id": null,
             "finished_devices": []
         }
     ]
   }
}
```

We can also list all available firmwares. Please note that here we don't create a job. Here we don't get the checksum for all images, since we don't want to waste cycles on computing that checksum for each and every firmware.

```
curl https://hostname/api/v1.0/firmware
{
 "status": "success",
 "data": {
     "status": "success",
     "data": {
         "files": [
             "firmware.bin",
             "firmware2.bin"
         ]
       }
     }
 }
```

Remove firmware

To remove a firmware image:

```
curl -X DELETE https://hostname/api/v1.0/firmware/firmware.bin
{
 "status": "success",
 "data": "Scheduled job to remove firmware",
 "job_id": "5d849177dd428720db72c693"
}
```

Upgrade firmware on device(s)

As of today we support upgrading firmware on Arista EOS acces switches. The upgrade procedure can do a 'pre-flight check' which will make sure there is enough disk space before attempting to download the firmware.

The API method will accept a few parameters:

- group: Optional. The name of a group, all devices in that group will be upgraded.
- hostname: Optional. If a hostname is specified, this single device will be upgraded.
- filename: Mandatory. Name of the new firmware, for example "test.swi".
- url: Optional, can also be configured as an environment variable, FIRMQRE_URL. URL to the firmware storage, for example "http://hostname/firmware/". This should typically point to the CNaaS NMS server and files will be downloaded from the CNaaS HTTP server.
- download: Optional, default is false. Only download the firmware.
- pre_flight: Optional, default is false. If true, check disk-space etc before downloading the firmware.
- post_flight: Optional, default is false. If true, update OS version after the upgrade have been finished.
- post_waittime: Optional, default is 0. Defines the time we should wait before trying to connect to an updated device.
- activate: Optional, default is false. Control whether we should install the new firmware or not.
- reboot: Optional, default is false. When the firmware is downloaded, reboot the switch.
- start_at: Schedule a firmware upgrade to be started sometime in the future.

An example CURL command can look like this:

```
curl -k -s -H "Content-Type: application/json" -X POST https://hostname/api/v1.0/
↪firmware/upgrade -d '{"group": "ACCESS", "filename": "test_firmware.swi", "url":
↪"http://hostname/", "pre-flight": true, "download": true, "activate": true, "reboot
↪": true, "start_at": "2019-12-24 00:00:00", "post_flight": true, "post_waittime":␣
↪600'}
```

The output from the job will look like this:

```
{
  "status": "success",
  "data": {
    "jobs": [
      {
        "id": "5dcd110a5670fd67a615b089",
        "start_time": "2019-11-14 08:32:11.135000",
        "finish_time": "2019-11-14 08:34:50.352000",
```

(continues on next page)

(continued from previous page)


```
        "status": "FINISHED",
        "function_name": "device_upgrade",
        "result": {
          "eoaccess": [
            {
              "name": "device_upgrade_task",
              "result": "",
              "diff": "",
              "failed": false
            },
            {
              "name": "arista_pre_flight_check",
              "result": "Pre-flight check done.",
              "diff": "",
              "failed": false
            },
            {
              "name": "arista_firmware_download",
              "result": "Firmware download done."
              "diff": "",
              "failed": false
            },
            {
              "name": "arista_firmware_activate",
              "result": "Firmware activate done.",
              "diff": "",
              "failed": false
            },
            {
              "name": "arista_device_reboot",
              "result": "Device reboot done.",
              "diff": "",
              "failed": false
            },
            {
              "result": "Post-flight, OS version updated for device eosaccess, now 4.
↪23.2F-15405360.4232F.",
              "task_name": "arista_post_flight_check",
              "diff": "",
              "failed": false
            }
          ],
          "_totals": {
            "selected_devices": 1
          }
        },
        "exception": null,
        "traceback": null,
        "next_job_id": null,
        "finished_devices": [\"eosaccess\"]
      }
    ]
  }
}
```

1.3.12 System

Version

To get the currently running version of the CNaaS-NMS API:

```
curl https://hostname/api/v1.0/system/version
```

Example output:

```
{
    "status": "success",
    "data": {
        "version": "0.2.0.dev0",
        "git_version": "Git commit 3a08c0e10d1cc31a4634d383e12394e758615747 feature.
↪settings_dhcprelay (2020-01-20 13:07:48+01:00)"
    }
}
```

1.4 Repository Reference

1.4.1 Templates

Templates for switch configurations.

In the base of this repository there should be one directory for each network operating system platform, like "eos", "junos" or "iosxr".

In each of these directories there needs to be a file called "mapping.yml", this file defines what template files should be used for each device type. For example, in mapping.yml there might be a definition of templates for an access switch specified like this:

```
ACCESS:
    entrypoint: access.j2
    dependencies:
        - managed-full.j2
```

This indicates that the starting point for the template of access switches for this platform is deffined in the Jinja2 template file called "access.j2". Additionally, this template file will depend on things defined in a file called "managed-full.j2".

The template files themselves are written using the Jinja2 templating language. Variables that are exposed from CNaaS includes:

- mgmt_ip: IPv4 management address (ex 192.168.0.10)
- mgmt_ipif: IPv4 management address including prefix length (ex 192.168.0.10/24)
- mgmt_prefixlen: Just the prefix length (ex 24)
- mgmt_vlan_id: VLAN id for management (ex 10)
- mgmt_gw: IPv4 address for the default gateway in the management network
- uplinks: A list of uplink interfaces, each interface is a dictionary with these keys:
  - ifname: Name of the physical interface (ex Ethernet1)

- access_auto: A list of access_auto interfacs. Using same keys as uplinks.
- device_model: Device model string, same as "model" in the device API. Can be used if you need model specific configuration lines.
- device_os_version: Device OS version string, same as "os_version" in the device API. Can be used if you need OS version specific configuration lines.

Additional variables available for distribution switches:

- infra_ip: IPv4 infrastructure VRF address (ex 10.199.0.0)
- infra_ipif: IPv4 infrastructure VRF address inc prefix (ex 10.199.0.0/32)
- vrfs: A list of dictionaries with two keys: "name" and "rd" (rd as in Route Distinguisher). Populated from settings defined in routing.yml.
- bgp_ipv4_peers: A list of dictionaries with the keys: "peer_hostname", "peer_infra_lo", "peer_ip" and "peer_asn". Contains one entry per directly connected dist/core device, used to build an eBGP underlay for the fabric. Populated from the links database table.
- bgp_evpn_peers: A list of dictionaries with the keys: "peer_hostname", "peer_infra_lo", "peer_asn". Contains one entry per hostname specified in settings->evpn_peers. Used to build eBGP peering for EVPN between loopbacks.
- mgmtdomains: A list of dictionaries with the keys: "ipv4_gw", "vlan", "description", "esi_mac". Populated from the mgmtdomains database table.
- asn: A private unique Autonomous System number generated from the last two octets of the infra_lo IP address on the device.

All settings configured in the settings repository are also exposed to the templates.

1.4.2 settings

Settings are defined at different levels and inherited (possibly overridden) in several steps. For example, NTP servers might be defined in the "global" settings to impact the entire managed network, but then overridden for a specific device type that needs custom NTP servers. The inheritence is defined in these steps: Global -> Core/Dist/Access -> Device specific. The directory structure looks like this:

- global
  - groups.yml: Definition of custom device groups
  - routing.yml: Definition of global routing settings like fabric underlay and VRFs
  - vxlans.yml: Definition of VXLAN/VLANs
  - base_system.yml: Base system settings
- core
  - base_system.yml: Base system settings
  - interfaces_<model>.yml: Model specific default interface settings
- dist
  - base_system.yml: Base system settings
  - interfaces_<model>.yml: Model specific default interface settings
- access:
  - base_system.yml: Base system settings

- devices:
  - <hostname>
    - base_system.yml
    - interfaces.yml
    - routing.yml

groups.yml:

Contains a dictionary named "groups", that contains a list of groups. Each group is defined as a dictionary with a single key named "group", and that key contains a dictionary with two keys:

- name: A string representing a name. No spaces.
- regex: A Python style regex that matches on device hostnames

All devices that matches the regex will be included in the group.

```
---
groups:
  - group:
      name: 'ALL'
      regex: '.*'
  - group:
      name: 'BORDER_DIST'
      regex: '(south-dist0[1-2]|north-dist0[1-2])'
  - group:
      name: 'DIST_EVEN'
      regex: '.*-dist[0-9][02468]'
  - group:
      name: 'DIST_ODD'
      regex: '.*-dist[0-9][13579]'
```

routing.yml:

Can contain the following dictionaries with specified keys:

- underlay:
  - infra_link_net: A /16 of IPv4 addresses that CNaaS-NMS can use to automatically assign addresses for infrastructure links from (ex /31 between dist-core).
  - infra_lo_net: A /16 of IPv4 addresses that CNaaS-NMS can use to automatically assign addresses for infrastructure loopback interfaces from.
  - mgmt_lo_net: A subnet for management loopbacks for dist/core devices.
- evpn_peers:
  - hostname: A hostname of a CORE (or DIST) device from the device database. The other DIST switches participating in the VXLAN/EVPN fabric will establish eBGP connections to these devices. If an empty list is provided all CORE devices will be added as evpn_peers instead.
- vrfs:
  - name: The name of the VRF. Should be one word (no spaces).
  - vrf_id: An integer between 1-65535. This ID can be used to generate unique VNI, RD and RT values for this VRF.
  - groups: A list of groups this VRF should be provisioned on.
  - import_route_targets: A list of strings containing extra route targets to import for route leaking (optional)

  - export_route_targets: A list of strings containing extra route targets to export for route leaking (optional)
  - import_policy: A string containing route policy/route map to define import behavior, useful in route leaking scenarios (optional)
  - export_policy: A string containing route policy/route map to define export behavior, useful in route leaking scenarios (optional)
- extroute_static:
  - vrfs:
    - name: Name of the VRF
    - ipv4:
      - destination: IPv4 prefix
      - nexthop: IPv4 nexthop address
      - interface: Exiting interface (optional)
      - name: Name/description of route (optional, defaults to "undefined")
      - cli_append_str: Custom configuration to append to this route (optional)
    - ipv6:
      - destination: IPv6 prefix
      - nexthop: IPv6 nexthop address
      - other options are the same as ipv4
- extroute_ospfv3:
  - vrfs:
    - name: Name of the VRF
    - ipv4_redist_routefilter: Name of a route filter (route-map) that filters what should be redistributed into OSPF
    - ipv6_redist_routefilter: Name of a route filter (route-map) that filters what should be redistributed into OSPF
    - cli_append_str: Custom configuration to add for this VRF (optional)
- extroute_bgp:
  - vrfs:
    - name: Name of the VRF
    - local_as: AS number that CNaaS NMS devices will present themselves as
    - cli_append_str: Custom configuration to append to BGP VRF config (optional)
    - neighbor_v4:
      - peer_as: AS number the remote peer
      - peer_ipv4: IPv4 address of peer
      - route_map_in: Route-map to filter incoming routes
      - route_map_out: Route-map to filter outgoing routes
      - ebgp_multihop: Configure eBGP multihop/TTL security, integer 1-255

- bfd: Set to true to enable Bidirectional Forward Detection (BFD)
- graceful_restart: Set to true to enable capability graceful restart
- next_hop_self: Set to true to always advertise this router's address as the BGP next hop
- maximum_routes: Maximum routes to receive from peer, integer 0-4294967294
- update_source: Specify local source interface for the BGP session
- auth_string: String used to calculate MD5 hash for authentication (password)
- description: Description of remote peer (optional, defaults to "undefined")
- cli_append_str: Custom configuration to append to this peer (optional)

* neighbor_v6:
  - peer_ipv6: IPv6 address of peer
  - other options are the same as neighbor_v4

routing.yml examples:

```
---
extroute_bgp:
  vrfs:
    - name: OUTSIDE
      local_as: 64667
      neighbor_v4:
        - peer_ipv4: 10.0.255.1
          peer_as: 64666
          route_map_in: fw-lab-in
          route_map_out: default-only-out
          description: "fw-lab"
          bfd: true
          graceful_restart: true
extroute_static:
  vrfs:
    - name: MGMT
      ipv4:
        - destination: 172.12.0.0/24
          nexthop: 10.0.254.1
          name: cnaas-mgmt
```

vxlans.yml:

Contains a dictinary called "vxlans", which in turn has one dictinoary per vxlan, vxlan name is the dictionary key and dictionaly values are:

- vni: VXLAN ID, 1-16777215
- vrf: VRF name. Optional unless ipv4_gw is also specified.
- vlan_id: VLAN ID, 1-4095
- vlan_name: VLAN name, single word/no spaces, max 31 characters
- ipv4_gw: IPv4 gateway address in CIDR notation, ex: 192.168.0.1/24. Optional.
- ipv4_secondaries: List of IPv4 addresses in CIDR notation. Optional.
- ipv6_gw: IPv6 address, ex: fe80::1. Optional.
- dhcp_relays: DHCP relay address. Optional.

- mtu: Define custom MTU. Optional.
- acl_ipv4_in: Access control list to apply for ingress IPv4 traffic to routed interface. Optional.
- acl_ipv4_out: Access control list to apply for egress IPv4 traffic from routed interface. Optional.
- acl_ipv6_in: Access control list to apply for ingress IPv6 traffic to routed interface. Optional.
- acl_ipv6_out: Access control list to apply for egress IPv6 traffic from routed interface. Optional.
- cli_append_str: Optional. Custom configuration to append to this interface.
- tags: List of custom strings to tag this VXLAN with. Optional.
- groups: List of group names where this VXLAN/VLAN should be provisioned. If you select an access switch the parent dist switch should be automatically provisioned.
- devices: List of device names where this VXLAN/VLAN should be provisioned. Optional.

interfaces.yml:

For dist and core devices interfaces are configured in YAML files. The interface configuration can either be done per device, or per device model. If there is a device specific folder under devices/ then the model interface settings will be ignored. Model specific YAML files should be named like the device model as listed in the devices API, but in all lower-case and with all whitespaces replaced with underscore ("_").

Keys for interfaces.yml or interfaces_<model>.yml:

- interfaces: List of dicctionaries with keys:
  - name: Interface name, like "Ethernet1"
  - ifclass: Interface class, one of: downlink, fabric, custom, port_template_*
  - config: Optional. Raw CLI config used in case "custom" ifclass was selected
- Additional interface options for port_template type:
  - untagged_vlan: Optional. Numeric VLAN ID for untagged frames.
  - tagged_vlan_list: Optional. List of allowed numeric VLAN IDs for tagged frames.
  - description: Optional. Description for the interface, this should be a string 0-64 characters.
  - enabled: Optional. Set the administrative state of the interface. Defaults to true if not set.
  - aggregate_id: Optional. Identifier for configuring LACP etc. Integer value. Special value -1 means configure MLAG and use ID based on indexnum.
  - cli_append_str: Optional. Custom configuration to append to this interface.

The "downlink" ifclass is used on DIST devices to specify that this interface is used to connect access devices. The "fabric" ifclass is used to specify that this interface is used to connect DIST or CORE devices with each other to form the switch (vxlan) fabric. Linknet data will only be configured on interfaces specified as "fabric". If no linknet data is available in the database then the fabric interface will be configured for ZTP of DIST/CORE devices by providing DHCP (relay) access. "port_template_*" is used to specify a user defined port template. This can then be used to apply some site-specific configuration via Jinja templates. For example specify "port_template_hypervisor" and build a corresponding Jinja template by matching on that ifclass.

base_system.yml:

Contains base system settings like:

- ntp_servers
- snmp_servers
- dns_servers

- syslog_servers
- flow_collectors
- dhcp_relays
- internal_vlans
- dot1x_fail_vlan: Numeric ID of authentication fail VLAN

Example of base_system.yml:

```
---
ntp_servers:
  - host: 10.255.0.1
  - host: 10.255.0.2
snmp_servers:
  - host: 10.255.0.11
dns_servers:
  - host: 10.255.0.1
  - host: 10.255.0.2
syslog_servers:
  - host: 10.255.0.21
  - host: 10.255.0.22
flow_collectors:
  - host: 10.255.0.30
    port: 6343
dhcp_relays:
  - host: 10.255.1.1
  - host: 10.255.1.2
internal_vlans:
  vlan_id_low: 3006
  vlan_id_high: 4094
dot1x_fail_vlan: 13
```

syslog_servers and radius_severs can optionally have the key "port" specified to indicate a non-defalut layer4 (TCP/UDP) port number.

internal_vlans can optionally be specified if you want to manually define the range of internal VLANs on L3 switches. You can also specify the option "allocation_order" under internal_vlans which is a custom string that defaults to "ascending". If internal_vlans is specified then a collision check will be performed for any defined vlan_ids in vxlans settings.

1.4.3 etc

Configuration files for system daemons

Directory structure:

- dhcpd/
  - dhcpd.conf: Used for ZTP DHCPd

1.5 Configuration

CNaaS NMS relies on configuration files and environment variables for configuration.

1.5.1 Config files

Config files are placed in /etc/cnaas-nms

/etc/cnaas-nms/db_config.yml

Defines how to connect to the SQL and redis databases.

/etc/cnaas-nms/api.yml

Defines parameters for the API:

- host: Defines the listening host/IP, default 0.0.0.0
- jwtcert: Defines the path to the public JWT certificate used to verify JWT tokens
- httpd_url: URL to the httpd container containing firmware images
- verify_tls: Verify certificate for connections to httpd/firmware server
- verify_tls_device: Verify TLS connections to devices, defaults to True
- cafile: Path to CA certificate used to verify device certificates. If no path is specified then the system default CAs will be used.
- cakeyfile: Path to CA key, used to sign device certificates after generation.
- certpath: Path to store generated device certificates in.
- allow_apply_config_liverun: Allow liverun on apply_config API call. Defaults to False.

/etc/cnaas-nms/repository.yml

Defines paths to git repositories.

1.5.2 Environment variables

Besides config files, cnaas-nms uses environment variables for configuration. The environment variables are typically set using docker-compose.

Docker-compose will spin up a multi container environment including the CNaaS NMS API, httpd and dhcp server, postgresql, redis and the JWT auth server.

There are various ways to set environment variables in docker-compose. The most common one is the `docker-compose.yml` file.

A list of the environment variables used by each Docker container:

cnaas_api

- `GITREPO_TEMPLATES` – templates git repository
- `GITREPO_SETTINGS` – settings git repository

- COVERAGE – calculate test coverage. 1 or 0 (yes or no)
- USERNAME_DHCP_BOOT – user name to log into devices during DHCP boot process
- PASSWORD_DHCP_BOOT
- USERNAME_DISCOVERED – user name for discovered devices
- PASSWORD_DISCOVERED
- USERNAME_INIT – user name for initialised devices
- PASSWORD_INIT
- USERNAME_MANAGED – user name for managed devices
- PASSWORD_MANAGED

cnaas_httpd

- GITREPO_TEMPLATES – templates git repository

cnaas_dhcpd

- GITREPO_ETC – git repository containing dhcpd config
- DB_PASSWORD – database password
- DB_HOSTNAME – database host
- JWT_AUTH_TOKEN – token to authenticate against the cnaas-nms REST API

cnaas_postgres

- POSTGRES_USER – database username
- POSTGRES_PASSWORD – database password
- POSTGRES_DB – name of the cnaas-nms database

1.6 Plugins

CNaaS-NMS uses an extendable and configurable plugin system that can be used to integrate external systems into certain workflows. Plugins register to predefined hooks in the CNaaS-NMS system and gets called with a set of arguments specific to that hook. Multiple plugins can register for the same hooks at the same time. For example, when a new device is added to CNaaS all plugins with the hook new_managed_device will be called with arguments like hostname and device type.

1.6.1 Configuration

/etc/cnaas-nms/plugins.yml configuration example:

```
---
plugins:
  - filename: filewriter.py
    vars:
      logfile: "/tmp/filewriter.log"
```

This file contains a list of plugins that should be loaded. Any vars defined here can be accessed from the plugin, this can be used to set URLs to other system APIs etc.

What plugins got loaded and what variables that are available can be accessed via the rest API as well.

1.6.2 Hooks

new_managed_device

Called when a new device is initialized through the init_device API call and that device reaches the "MANAGED" state.

Arguments provided:

- hostname: Hostname of the new device
- device_type: What type of device, ACCESS, DIST or CORE
- serial_number: Serial number of device
- vendor: Vendor/manufacturer of device
- model: Hardware model of device
- os_version: Operating System version of device

allocated_ipv4

Called when CNaaS-NMS has allocated a new IPv4 address and configured it on a device. Currently only called during the init_device API call.

Argumens proided:

- vrf: Name of the VRF used (ex mgmt)
- ipv4_address: IPv4 address (ex 10.0.6.6)
- ipv4_network: IPv4 network address in CIDR notation (ex 10.0.6.0/24)
- hostname: Hostname of the device that uses this address

1.7 Upgrading

Versioning schema based on Python PEP 440, <major>.<minor>.<micro> (ex 1.0.0)

When upgrading to a new major version the API will change and expose new URLs using /api/v2.0/ and so on.

When upgrading between minor versions the database schemas might change, use alembic to upgrade. Also settings options/template variables might change between minor versions.

Micro releases should only contain bug fixes and no changes in features or database schemas.

1.8 Contributing

CNaaS-NMS is open source and everything including source code, documentation etc is available to the public on Github. Please send pull requests using Github for any contributions.

1.8.1 Coding style

CNaaS-NMS is developed for python 3.7 and uses type hinting and dataclasses.

PEP8 is used for style guidelines, we try to follow it as long as it makes things more readable, and we use a locally defined maximum line length of 99 characters.

PEP440 is used for versioning using the style 0.1.0dev1, 0.1.0b1, 0.1.0rc1, 0.1.0 etc.

Unit tests should be written for all parts that can be tested individually, but since the project heavily relies on physical lab equipment some tests has to be performed only as integration tests instead. The suite of unit tests should ideally complete within 5 minutes and be run on all check-ins to any branch. Integration tests should run nightly on the master branch. The combined code coverage for unit tests plus integration tests should be around 80%.

Core components of CNaaS-NMS should use type-hinting to help prevent bugs and increase maintainability of the code. MyPy should pass without errors on the main code base (everything except possibly tests?).

Alembic is used to handle SQL schema updates/versioning.

1.9 Changelog

1.9.1 Version 1.3.2

Bug fixes:

- Fix for ZTP init of dist devices (#219,#218)

1.9.2 Version 1.3.1

New features:

- New settings for vxlans: acl_ipv4_in, acl_ipv4_out, acl_ipv6_in, acl_ipv6_out, cli_append_str
- New data options in interfaces API: bpdu_filter, tags, cli_append_str

1.9.3 Version 1.3.0

New features:

- CNaaS specific Jinja2 filters: increment_ip, ipv4_to_ipv6, isofy_ipv4 (#167)
- “aggregate_id” option for access ports to build link aggregates from access switches (#171)
- New settings for: flow_collectors, route leaking, port_template, dot1x_fail_vlan, vxlan tags, ipv4_secondaries (#178,#192,#193,#194,#195,#196,#203)
- Automatic descriptions for ACCESS_DOWNLINK type ports (#189)
- Option to filter job result output fields in API response (#197)

Bug fixes:

- Fix race condition issue where different threads could sometimes cause issues with wrong template being used when syncing multiple different operating systems in same job (#168,#176)
- Fix validation and return output for mgmtdomains API (#177)
- Cleanup of docker images (#184,#185,#186,#191)

- Update device last_seen on syncto, update facts, firmware post flight, device discovered, init step2 (#198)
- Fix factory_default: false (#200)
- Fix assigning of vxlans etc to both groups and devices at same time (#201)
- Possible fix for "weak object has gone away" (#205)
- Fixes for device synchranization status updating (#208,#209)

1.9.4 Version 1.2.1

Bugfix release.

Bug fixes:

- Fix for ZTP of fabric devices when INIT and DISCOVERED passwords are different
- Fix for mgmt_ip variable at initial fabric device sync
- Better init check error message
- Documentation fix
- Include groups with no devices in listing

1.9.5 Version 1.2.0

New features:

- ZTP support for core and diste devices (#137)
- Init check API call to test if device is compatible for ZTP without commit (#136, #156)
- Option to have model-specific default interface settings (#135)
- Post-flight check for firmware upgrade (#139)
- Abort scheduled jobs, best-effort abort of running jobs (#142)
- API call to update existing interfaces on device after ZTP (#155)
- More settings for external BGP routing, DNS servers, internal VLANs (#143, #146, #152)
- Install NMS issued certificate on new devices during ZTP (#149)
- Switch to Nornir 3.0, improved whitespace rendering in templates (#148)

Bug fixes:

- Fix blocking websockets (#138)
- Fix access downlink port detection (#141)
- Post upgrade confighash mismatch (#145)
- Discover device duplicate jobs improvements (#151)
- Trim facts fields before saving in database (#153)

1.9.6 Version 1.1.0

New features:

- New options for connecting access switches:
  - Two access switches as an MLAG pair
  - Access switch connected to other access switch
- New template variables:
  - device_model: Hardware model of this device
  - device_os_version: OS version of this device
- Get/restore previous config versions for a device
- API call to update facts (serial,os version etc) about device
- Websocket event improvements for logs, jobs and device updates

1.9.7 Version 1.0.0

New features:

- Syncto for core devices
- Access interface updates via API calls, "port bounce"
- Static, BGP and OSPF external routing template support
- eBGP / EVPN fabric template support
- VXLAN definition improvements (dhcp relay, mtu)

1.9.8 Version 0.2.0

New features:

- Syncto for dist devices
- VXLAN definitions in settings
- Firmware upgrade for Arista

1.9.9 Version 0.1.0

Initial test release including device database, syncto and ZTP for access devices, git repository refresh etc.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search