NAPALM Documentation

Release 1

David Barroso

Aug 13, 2017

Contents

YANG (RFC6020) is a data modelling language, it's a way of defining how data is supposed to look like. The napalm-yang library provides a framework to use models defined with YANG in the context of network management. It provides mechanisms to transform native data/config into YANG and vice versa.

You can take a look to the following tutorial to see what this is about and how to get started.

CHAPTER 1

Installation

To install `napalm-yang` you can use pip as with any other driver:

```
pip install -U napalm-yang
```

CHAPTER 2

Documentation

Profiles

In order to correctly map YANG objects to native configuration and vice versa, `napalm-yang` uses the concept of **profiles**. Profiles, identify the type of device you are dealing with, which can vary depending on the OS, version and/or platform you are using.

If you are using a napalm driver and have access to your device, you will have access to the `profile` property which you can pass to any function that requires to know the profile. If you are not using a napalm driver or don't have access to the device, a profile is just a list of strings so you can just specify it directly. For example:

```
# Without access to the device
model.parse_config(profile=["junos"], config=my_configuration)

# With access
with driver(hostname, username, password) as d:
    model.parse_config(device=d)

# With access but overriding profile
with driver(hostname, username, password) as d:
    model.parse_config(device=d, profile=["junos13", "junos"])
```

Note: As you noticed a device may have multiple profiles. When that happens, each model that is parsed will loop through the profiles from left to right and use the first profile that implements that model (note that a YANG model is often comprised of multiple modules). This is useful as there might be small variances between different systems but not enough to justify reimplementing everything.

You can find the profiles here but what exactly is a profile? A profile is a bunch of YAML files that follows the structure of a YANG model and describes two things:

1. How to parse native configuration/state and map it into a model.
2. How to translate a model and map it into native configuration.

For example, for a given interface, the snippet below specifies how to map configuration into the `openconfig_interface` model on EOS:

```
enabled:
    _process:
        - mode: is_present
          regexp: "(?P<value>no shutdown)"
          from: "{{ parse_bookmarks.interface[interface_key] }}"
description:
    _process:
        - mode: search
          regexp: "description (?P<value>.*)"
          from: "{{ parse_bookmarks.interface[interface_key] }}"
mtu:
    _process:
        - mode: search
          regexp: "mtu (?P<value>[0-9]+)"
          from: "{{ parse_bookmarks.interface[interface_key] }}"
```

And the following snippet how to map the same attributes from the `openconfig_interface` to native configuration:

```
enabled:
    _process:
        - mode: element
          value: "    shutdown\n"
          when: "{{ not model }}"
description:
    _process:
        - mode: element
          value: "    description {{ model }}\n"
          negate: "    default description"
mtu:
    _process:
        - mode: element
          value: "    mtu {{ model }}\n"
          negate: "    default mtu\n"
```

Note: Profiles can also deal with structured data like XML or JSON.

As you can see it's not extremely difficult to understand what they are doing, in the next section we will learn how to write our own profiles.

YANG Basics

It's not really necessary to understand how YANG works to write a profile but you need some basic understanding.

Basic Types

- **container** - A container is just a placeholder, sort of like a map or dictionary. A container doesn't store any information per se, instead, it contains attributes of any type. For example, the following `config` object would be a valid container with three attributes of various types:

```
container config:
    leaf description: string
    leaf mtu: uint16
    leaf enabled: boolean
```

- **leaf** - A leaf is an attribute that stores information. Leafs are of a type and values have to be valid for the given type. For example:

```
leaf descrpition: string # Any string is valid
leaf mtu: uint16         # -1 is not valid but 1500 is
leaf enabled: boolean    # true, false, 1, 0, True, False are valid
```

Note: There can be further restrictions, for example the leaf `prefix-length` is of type `uint8` but it's further restricted with the option `range 0..32`

- **YANG lists** - A YANG list represents a container in the tree that will represent individual members of a list. For example:

```
container interfaces:
    list interface:
        container config:
            leaf description: string
            leaf mtu: uint16
            leaf enabled: boolean
```

As we start adding elements to the interface list, each individual interface will have it's own attributes. For example:

```
interfaces:
    interface["eth1"]:
        config:
            description: "An interface"
            mtu: 1500
            enabled: true
    interface["eth2"]:
        config:
            description: "Another interface"
            mtu: 9000
            enabled: false
```

Writing Profiles

As it's been already mentioned, a profile is a bunch of YAML files that describe how to map native configuration and how to translate an object into native configuration. In order to read native configuration we will use **parsers**, to translate a YANG model into native configuration we will use **translators**.

Both parsers and translators follow three basic rules:

1. One directory per module.
2. One file per model.
3. Exact same representation of the model inside the file:

For example:

```
$ tree napalm_yang/mappings/eos/parsers/config
napalm_yang/mappings/eos/parsers/config
- napalm-if-ip
|   - secondary.yaml
- openconfig-if-ip
|   - ipv4.yaml
- openconfig-interfaces
|   - interfaces.yaml
- openconfig-vlan
    - routed-vlan.yaml
    - vlan.yaml

4 directories, 5 files
$ cat napalm_yang/mappings/eos/parsers/config/openconfig-vlan/vlan.yaml
---
metadata:
    (trimmed for brevity)

vlan:
    (trimmed for brevity)
    config:
        (trimmed for brevity)
        vlan_id:
            (trimmed for brevity)
```

If we check the content of the file `vlan.yaml` we can clearly see two parts:

- **metadata** - This part specifies what parser or translator we want to use as there are several depending on the type of data we are parsing from or translating to and some options that the parser/translator might need. For example:

```
metadata:
    processor: XMLParser
    execute:
        - method: _rpc
          args: []
          kwargs:
              get: "<get-configuration/>"
```

In this case we are using the `XMLParser` parser and in order to get the data we need from the device we have to call the method `_rpc` with the `args` and `kwargs` parameters. This is, by the way, an RPC call for a junos device.

- **vlan** - This is the part that follows the model specification. In this case is `vlan` but in others it might be `interfaces`, `addressess` or something else, this will be model dependent but it's basically whatever it's not `metadata`. This part will follow the model specification and add rules on each attribute to tell the parser/translator what needs to be done. For example:

```
vlan:
    _process: unnecessary
    config:
        _process: unnecessary
        vlan_id:
            _process:
              - mode: xpath
                xpath: "vlan-id"
                from: "{{ parse_bookmarks['parent'] }}"
```

As we are dealing with a parser we have to specify the `_process` attribute at each step (translators require the attribute _process). There are two special types of actions; unnecessary and not_implemented. Both do exactly the same, skip any action and move onto the next attribute. The only difference is purely aesthetically and for documentation purposes.

Something else worth noting is that each attribute inside _process/_process is evaluated as a jinja2 template so you can do variable substitutions, evaluations, etc...

Parsers

Parsers are responsible for mapping native configuration/show_commands to a YANG model.

Special actions

Most actions depend on the parser you are using, however, some are common to all of them:

unnecessary

This makes the parser skip the field and continue processing the tree.

not_implemented

This makes the parser stop processing the tree underneath this value. For example:

```
field_1:
    process: unnecessary
field_2:
    process: not_implemented
    subfield_1:
        process: ...
    subfield_2:
        process: ...
field_3:
    ...
```

The not_implemented action will stop the parser from processing subfield_1 and subfield_2 and move directly onto field_3.

gate

Works like not_implemented but accepts a condition. For example:

```
protocols:
    protocol:
        bgp:
            _process:
              - mode: gate
                when: "{{ protocol_key != 'bgp bgp' }}"
            global:
                ...
```

The snippet above will only process the bgp subtree if the condition is **not** met.

Special fields

When parsing attributes, some fields may depend on the parser you are using but some will be available regardless. Some may be even be mandatory.

mode

- **Mandatory**: Yes
- **Description**: Which parsing/translation action to use for this particular field.
- **Example**: Parse the description field with a simple regular expression:

```
_process:
  - mode: search
    regexp: "description (?P<value>.*)"
    from: "{{ bookmarks.interface[interface_key] }}"
```

when

- **Mandatory**: No
- **Description**: The evaluation of this field will determine if the action is executed or skipped. This action is probably not very useful when parsing but it's available if you need it.
- **Example**: Configure `switchport` on IOS devices only if the interface is not a Loopback or a Management interface:

```
ipv4:
    _process: unnecessary
    config:
        _process: unnecessary
        enabled:
            _process:
                - mode: element
                  value: "    no switchport\n"
                  negate: "    switchport\n"
                  in: "interface.{{ interface_key }}"
                  when: "{{ model and interface_key[0:4] not in ['Mana', 'Loop'] }
↪}"
```

from

- **Mandatory**: Yes
- **Description**: Configuration to read. In combination with `bookmarks` provides the content we are operating with.
- **Example**: Get IP addresses from both both interfaces and subinterfaces:

```
address:
    _process:
      - mode: xpath
        xpath: "family/inet/address"
        key: name
        from: "{{ bookmarks['parent'] }}"
```

Special Variables

keys

When traversing lists, you will have all the relevant keys for the object available, including on nested lists. Let's see it with an example, let's say we are currently parsing `interfaces/interface["et1"]/subinterfaces/subinterface["0"].ipv4.addresses.address["10.0.0.1"]`. At this particular point you will have the following keys available:

- **address_key** - `10.0.0.1`
- **subinterface_key** - `0`
- **interface_key** - `et1`
- **parent_key** - `0`

When a list is traversed you will always have available a key with name `$(attribute)_key`. In addition, you will have `parent_key` as the key of the immediate parent object. In the example above, `parent_key` will correspond to `0` as it's the immediate parent of the address object.

bookmarks

Bookmarks are points of interest in the configuration. Usually, you will be gathering blocks of configurations and parsing on those but sometimes, the configuration you need might be somewhere else. For those cases, you will be able to access those with the bookmarks. Using the same example as before, `interfaces/interface["et1"]/subinterfaces/subinterface["0"].ipv4.addresses.address["10.0.0.1"]`, you will have the following bookmarks:

- `bookmarks.interfaces` - The root of the configuration
- `bookmarks.interface["et1"]` - The block of configuration that corresponds to the interface `et1`
- `bookmarks.subinterface["0"]` - The block of configuration that corresponds to the subinterface `0` of `et1`.
- `bookmarks.address["10.0.0.1"]` - The block of configuration for the address belonging to the subinterface.
- `bookmarks.parent` - The block of configuration for the immediate parent, in this case, the subinterface `0`.

Note you can use keys instead and do `bookmarks.subinterface[parent_key]` or `bookmarks.subinterface[subinterface_key]`.

extra_vars

Some actions let's you provide additional information for later use. Those will be stored on the `extra_vars` dictionary. For example:

```
address:
    _process:
      - mode: block
        regexp: "(?P<block>ip address (?P<key>(?P<ip>.*))\\/(?P<prefix>\\d+))(?P
↪<secondary> secondary)*"
        from: "{{ bookmarks['parent'] }}"
    config:
        _process: unnecessary
        ip:
            _process:
```

```
            - mode: value
              value: "{{ extra_vars.ip }}"
```

The first regexp captures a bunch of vars that later can be used by just reading them from `extra_Vars`.

Metadata

The metadata tells the profile how to process that module and how to get the necessary data from the device. For example:

```
---
metadata:
    parser: XMLParser
    execute:
        - method: _rpc
          args: []
          kwargs:
              get: "<get-configuration/>"
```

- **execute** is a list of calls to do to from the device to extract the data.
  - **method** is the method from the device to call.
  - **args** are the numbered/ordered arguments for the method
  - **kwargs** are the keyword arguments for the method

In addition, some methods like `parse_config` and `parse_state` may have mechanisms to pass the information needed to the parser instead of relying on a live device to obtain it. For parsers, you will just have to pass a string with the same information the profile is trying to gather.

XMLParser

This extractor will read an XML an extract data from it.

To illustrate the examples below we will use the following configuration:

```
<configuration>
    <interfaces>
        <interface>
            <name>ge-0/0/0</name>
            <description>adasdasd</description>
        </interface>
        <interface>
            <name>lo0</name>
            <disable/>
        </interface>
    </interfaces>
</configuration>
```

List - xpath

Advances in the XML document up to the point where the relevant list of elements is found.

Arguments:

- **xpath** (mandatory): elements to traverse
- **key** (mandatory): which element is the key of the list
- **post_process_filter** (optional): modify the key with this Jinja2 expression

Example:

Starting from the root, the following action will move us to `interface` so we can parse each interface individually:

```
interface:
    _process:
      - mode: xpath
        xpath: "interfaces/interface"
        key: name
        from: "{{ bookmarks.interfaces }}"
```

This means after this action we will have a list of interface blocks like this:

```
- <interface>
    <name>ge-0/0/0</name>
    <description>adasdasd</description>
  </interface>
- <interface>
    <name>lo0</name>
    <disable/>
  </interface>
```

And we will be able to keep processing them individually.

Leaf - xpath

Extracts a value from an element.

Arguments:

- **xpath** (mandatory): element to extract
- **regexp** (optional): Apply regexp to the value of the element. Must capture `value` group. See "leaf - map" example for more details.
- **default** (optional): Set this value if no element is found.
- **attribute** (optional): Instead of the `text` of the element extracted, extract this attribute of the element.

Example:

For each interface, read the element `description` and map it into the object:

```
description:
    _process:
      - mode: xpath
        xpath: description
        from: "{{ bookmarks['parent'] }}"
```

Leaf - value

Apply a user-defined value to the object.

Arguments:

- **value** (mandatory): What value to apply

Example:

In the following example we can assign a value we already have to the `interface.name` attribute:

```
name:
    _process:
      - mode: value
        value: "{{ interface_key }}"
```

Leaf - map

Extract value and do a lookup to choose value.

Arguments:

- **xpath** (mandatory): Same as `xpath` action.
- **regexp** (optional): Same as `xpath` action.
- **map** (mandatory): Dictionary where we will do the lookup action.

Example:

We can read an element, extract some information and then apply the lookup function, for example, we can read the interface name, extract some of the first few characters and figure out the type of interface like this:

```
type:
    _process:
      - mode: map
        xpath: name
        regexp: "(?P<value>[a-z]+).*"
        from: "{{ bookmarks['parent'] }}"
        map:
            ge: ethernetCsmacd
            lo: softwareLoopback
            ae: ieee8023adLag
```

The regular expression will give *ge* and *lo* which we can map into *ethernetCsmacd* and *ieee8023adLag* respectively.

Leaf - is_absent

Works exactly like `xpath` but if the evaluation is `None`, it will return `True`.

Example:

We could check if an interface is enabled with this:

```
enabled:
    _process:
      - mode: is_absent
        xpath: "disable"
        from: "{{ bookmarks['parent'] }}"
```

As *disable* is missing in the interface *ge-0/0/0* we know it's enabled while *lo0* will be not as it was present.

Leaf - is_present

Works exactly like `xpath` but if the evaluation is `None`, it will return `False`.

TextParser

Will apply regular expressions to text to extract data from it.

To explain how this parser works, let's use the following configuration:

```
interface Ethernet1
    no switchport
!
interface Ethernet1.1
    description blah
!
interface Loopback1
    no switchport
    ip address 192.168.0.1/24
    ip address 192.168.1.1/24 secondary
!
```

Note: The regular expressions on this parser have the `MULTILINE` and `IGNORECASE` flags turned on.

List - block

Using a regular expression it divides the configuration in blocks where each block is relevant for a different element of the list.

Arguments:

- **regexp** (mandatory) - Regular expression to apply. Note that it must capture two things at least; `block`, which will be the entire block of configuration relevant for the interface and `key`, which will be the key of the element.
- **mandatory** (optional) will force the creation of one or more elements by specifying them manually in a dict the `key`, `block` (can be empty string) and any potential `extra_vars` you may want to specify.
- **composite_key** (optional) is a list of attributes captured in the regexp to be used as the key for the element.
- **flat** (optional) if set to `true` (default is `false`) the parser will understand the configuration for the element consists of flat commands instead of nested (for example BGP neighbors or static routes)
- **key** (optional) set key manually
- **post_process_filter** (optional) - Modify the key with this Jinja expression. `key` and `extra_vars` variables are available.

Example 1

Capture the interfaces:

```
_process:
  - mode: block
    regexp: "(?P<block>interface (?P<key>(\\w|-)*\\d+)\n(?:.|\n)*?^!$)"
    from: "{{ bookmarks.interfaces }}"
```

So the regexp is basically doing two things. Capturing each block of text that starts with `interface (a word)(a number)\n` (no dots allowed as a dot means it's subinterface) and then finishing in `!`. It's also getting the `key`. So after this step we will have a list like:

```
- key: Ethernet1
  block: interface Ethernet1
            no switchport
        !
- key: Loopback1
  block: interface Loopback1
            no switchport
            ip address 192.168.0.1/24
            ip address 192.168.1.1/24 secondary
        !
```

Note that `Ethernet1.1` is missing as it's not matching the key.

Example 2

As we process `Ethernet1` we will want it's subinterfaces so we can use a similar regexp as before but looking for a `dot` in the key, using the `interface_key (Ethernet1)` as part of the regexp. We also have to make sure in the from we went back to the root of the config:

```
subinterface:
    _process:
      - mode: block
        regexp: "(?P<block>interface {{interface_key}}\\.(?P<key>\\d+)\\n(?:.
↪|\\n)*?^!$)"
        from: "{{ bookmarks.interfaces }}"
```

Example 3

Sometimes we can get easily more information in one go than just the `key` and the `block`. For those cases we can capture more groups and they will be stored in the `extra_vars` dictionary:

```
address:
    _process:
      - mode: block
        regexp: "(?P<block>ip address (?P<key>(?P<ip>.*))\\/(?P<prefix>
↪\\d+))(?P<secondary> secondary)*"
        from: "{{ bookmarks['parent'] }}"
```

Example 4

In some cases native configuration might be "flat" but nested in a YANG model. This is the case of the *global* or *default* VRF, in those cases, it is hard you may want to ensure that *global* VRF is always created:

```
_process:
  - mode: block
    regexp: "(?P<block>vrf definition (?P<key>(.*))\n(?:.|\n)*?^!$)"
    from: "{{ bookmarks['network-instances'][0] }}"
    mandatory:
        - key: "global"
```

```
          block: ""
          extra_vars: {}
```

Example 5

Some list elements have composite keys, if that's the case, use the composite key to tell the parser how to map captured elements to the composite key:

```
protocols:
    _process: unnecessary
    protocol:
        _process:
            - mode: block
              regexp: "(?P<block>router (?P<protocol_name>(bgp))\\s*(?P
↪<process_id>\\d+)*\n(?:.|\n)*?)^(!|   vrf \\w+)$"
              from: "{{ bookmarks['network-instances'][0] }}"
              composite_key: [protocol_name, protocol_name]
              when: "{{ network_instance_key == 'global' }}"
```

Example 6

Some list elements (like static routes or BGP neighbors) are configured as a flat list of commands instead of nested. By default, if you would try to parse each command individually the parser would try to create a new element with each line and fail as multiple lines belong to the same element but they are treated independently. By setting `flat:  true` this behavior is changed and subsequent commands will update an already created object:

```
bgp:
    neighbors:
        neighbor:
            _process:
                - mode: block
                  regexp: "(?P<block>neighbor (?P<key>\\d+.\\d+.\\d+.\\d+).*)
↪"
                  from: "{{ bookmarks['protocol'][protocol_key] }}"
                  flat: true
```

Example 7

In some rare cases you might not be able to extract the key directly from the configuration. For example, the `static` protocol consists of `ip route` commands. In that case you can set the key yourself:

```
protocols:
    protocol:
        _process:
            - mode: block
              regexp: "(?P<block>ip route .*\n(?:.|\n)*?^!$)"
              from: "{{ bookmarks['network-instances'][0] }}"
              key: "static static"
```

Example 8

Sometimes you need to transform the key value. For example, static routes require the prefix in CIDR format, but Cisco IOS outputs routes in `<network> <mask>` format. In that case you can use `post_process_filter` to apply additional filters:

```
static:
    _process:
        - mode: block
```

```
          regexp: "(?P<block>ip route (?P<key>\\d+\\S+ \\d+\\S+).*)"
          from: "{{ bookmarks['network-instances'][0] }}"
          post_process_filter: "{{ key|addrmask_to_cidr }}"
```

Leaf - search

Extract `value` from a regexp.

Arguments:

- **regexp** (mandatory) - Regular expression to apply. Note the regular expression has to capture the `value` at least but it can capture others if you want.
- **default** (optional) - Value to assign if the regexp returns nothing.

Example.

Get the description of an interface:

```
description:
    _process:
      - mode: search
        regexp: "description (?P<value>.*)"
        from: "{{ bookmarks.interface[interface_key] }}"
```

Leaf - value

Apply a user-defined value to the object.

Arguments:

- **value** (mandatory): What value to apply

Example.

Evaluate a value we already extracted and set model to `True` if is not `None`:

```
secondary:
    _process:
      - mode: value
        value: "{{ extra_vars.secondary != None }}"
```

Leaf - is_absent

Works exactly like search but if the evaluation is `None`, it will return `True`.

Example.

Check if an interface is an IP interface or not:

```
ipv4:
    _process: unnecessary
    config:
        _process: unnecessary
        enabled:
            _process:
              - mode: is_absent
```

```
                regexp: "(?P<value>^\\W*switchport$)"
                from: "{{ bookmarks['parent'] }}"
```

Leaf - is_present

Works exactly like search but if the evaluation is `None`, it will return `False`.

Example.

Check if an interface is enabled:

```
enabled:
    _process:
      - mode: is_present
        regexp: "(?P<value>no shutdown)"
        from: "{{ bookmarks.interface[interface_key] }}"
```

Leaf - map

Works exactly like search but we do a lookup of the value on a map.

Arguments:

- **regexp** (mandatory) - Same as `search`
- **default** (optional) - Same as `search`
- **map** (optional) - Map where to do the lookup function.

Example.

Check type of interface by extracting the name and doing a lookup:

```
_process:
  - mode: map
    regexp: "(?P<value>(\\w|-)*)\\d+"
    from: "{{ interface_key }}"
    map:
        Ethernet: ethernetCsmacd
        Management: ethernetCsmacd
        Loopback: softwareLoopback
        Port-Channel: ieee8023adLag
        Vlan: l3ipvlan
```

Translators

Translators are responsible for transforming a model into native configuration.

Special actions

Most actions depend on the parser you are using, however, some are common to all of them:

unnecessary

This makes the parser skip the field and continue processing the tree.

not_implemented

This makes the parser stop processing the tree underneath this value. For example:

```
field_1:
    process: unnecessary
field_2:
    process: not_implemented
    subfield_1:
        process: ...
    subfield_2:
        process: ...
field_3:
    ...
```

The `not_implemented` action will stop the parser from processing `subfield_1` and `subfield_2` and move directly onto `field_3`.

gate

Works like `not_implemented` but accepts a condition. For example:

```
protocols:
    protocol:
        bgp:
            _process:
              - mode: gate
                when: "{{ protocol_key != 'bgp bgp' }}"
            global:
                ...
```

The snippet above will only process the `bgp` subtree if the condition is **not** met.

Special fields

When translating an object, some fields might depend on the translator you are using but some will available regardless. Some may be even be mandatory.

mode

- **mandatory**: yes
- **description**: which parsing/translation action to use for this particular field
- **example**: Translate description attribute of an interface to native configuration:

```
description:
    _process:
        - mode: element
```

```
            value: "    description {{ model }}\n"
            negate: "    default description"
```

when

- **mandatory**: no
- **description**: the evaluation of this field will determine if the action is executed or skipped. This action is probably not very useful when parsing but it's available if you need it.
- **example**: configure `switchport` on IOS devices only if the interface is not a loopback or a management interface:

```
ipv4:
    _process: unnecessary
    config:
        _process: unnecessary
        enabled:
            _process:
                - mode: element
                  value: "    no switchport\n"
                  negate: "    switchport\n"
                  in: "interface.{{ interface_key }}"
                  when: "{{ model and interface_key[0:4] not in ['mana', 'loop'] }
↪}"
```

in

- **mandatory**: no
- **description**: where to add the configuration. Sometimes the configuration might have to be installed on a different object from the one you are parsing. For example, when configuring a tagged subinterface on junos you will have to add also a `vlan-tagging` option on the parent interface. On `IOS/EOS`, when configuring interfaces, you have to also add the configuration in the root of the configuration and not as a child of the parent interface:

```
vlan:
    _process: unnecessary
    config:
        _process: unnecessary
        vlan_id:
            _process:
                - mode: element
                  element: "vlan-tagging"
                  in: "interface.{{ interface_key }}" # <--- add element to␣
↪parent interface
                  when: "{{ model > 0 }}"
                  value: null
                - mode: element
                  element: "vlan-id"
                  when: "{{ model > 0 }}"

(...)
subinterface:
    _process:
```

```
        mode: container
        key_value: "interface {{ interface_key}}.{{ subinterface_key }}\n"
        negate: "no interface {{ interface_key}}.{{ subinterface_key }}\n"
        in: "interfaces"                          # <--- add element to root of
↪configuration
```

Note: This field follows the same logic as the *bookmarks* special field.

continue_negating

- **mandatory**: no
- **description**: This option, when added to a container, will make the framework to also negate children.
- **example**: We can use as an example the "network-instances" model. In the model, BGP is inside the `network-instance` container, however, in EOS and other platforms that BGP configuration is decoupled from the VRF, so in order to tell the framework to delete also the direct children you will have to use this option. For example:

```
network-instance:
    _process:
        - mode: container
          key_value: "vrf definition {{ network_instance_key }}\n"
          negate: "no vrf definition {{ network_instance_key }}\n"
          continue_negating: true
          end: "    exit\n"
          when: "{{ network_instance_key != 'global' }}"
    ...
    protocols:
        _process: unnecessary
        protocol:
            _process:
              - mode: container
                key_value: "router bgp {{ model.bgp.global_.config.as_ }}\n  vrf {
↪{ network_instance_key}}\n"
                negate: "router bgp {{ model.bgp.global_.config.as_ }}\n  no vrf {
↪{ network_instance_key}}\n"
                end: "    exit\n"
                when: "{{ protocol_key == 'bgp bgp' and network_instance_key !=
↪'global' }}"
                replace: false
                in: "network-instances"
```

The example above will generate:

```
no vrf definition blah
router bgp ASN
   no vrf blah
```

Without `continue_negating` it would just generate:

```
no vrf definition blah
```

Special variables

keys

See *keys*.

model

This is the current model/attribute being translated. You have the entire object at your disposal, not only it's value so you can do things like:

```
vlan_id:
    _process:
        - mode: element
          value: "    encapsulation dot1q vlan {{ model }}\n"
```

Or:

```
config:
    _process: unnecessary
    ip:
        _process: unnecessary
    prefix_length:
        _process:
            - mode: element
              value: "    ip address {{ model._parent.ip }}/{{ model }} {{ 'secondary' if model._parent.secondary else '' }}\n"
              negate: "    default ip address {{ model._parent.ip }}/{{ model }}\n"
```

XMLTranslator

XMLTranslator is responsible of translating a model into XML configuration.

Metadata

- **xml_root** - Set this value on the root of the model to instantiate the XML object.

For example:

```
---
metadata:
    processor: XMLTranslator
    xml_root: configuration
```

This will instantiate the XML object <configuration/>.

Container - container

Creates a container.

Arguments:

- **container** (mandatory) - Which container to create

- **replace** (optional) - Whether this element has to be replaced in case of merge/replace or it's not necessary (remember XML is hierarchical which means you can unset things directly in the root).

Example:

Create the `interfaces` container:

```
_process:
  . mode: container
    container: interfaces
    replace: true
```

List - container

For each element of the list, create a container.

Arguments:

- **container** (mandatory) - Which container to create
- **key_element** (mandatory) - Lists require a key element, this is the name of the element.
- **key_value** (mandatory) - Key element value.

Example:

Create interfaces:

```
interface:
    _process:
      . mode: container
        container: interface
        key_element: name
        key_value: "{{ interface_key }}"
```

This will result elements such as:

```
<interface>
  <name>ge-0/0/0</name>
</interface>
<interface>
  <name>lo0</name>
</interface>
```

Leaf - element

Adds an element to a container.

Arguments:

- **element** (mandatory): Element name.
- **value** (optional): Override value. Default is value of the object.

Example 1:

Configure description:

```
description:
    _process:
        - mode: element
          element: description
```

Example 2:

Enable or disable an interface:

```
enabled:
    _process:
        - mode: element
          element: "disable"
          when: "{{ not model }}"
          value: null
```

We override the value and set it to `null` because to disable we just have to create the element, we don't have to set any value.

Example 3:

Configure an IP address borrowing values from other fields:

```
config:
    _process: unnecessary
    ip:
        _process: unnecessary
    prefix_length:
        _process:
            - mode: element
              element: name
              value: "{{ model._parent.ip }}/{{ model }}"
              when: "{{ model }}"
```

TextTranslator

TextTranslator is responsible of translating a model into text configuration.

Metadata

- **root** - Set to true if this is the root of the model.

List - container

Create/Removes each element of the list.

Arguments:

- **key_value** (mandatory): How to create the element.
- **negate** (mandatory): How to eliminate/default the element.
- **replace** (optional): Whether the element has to be defaulted or not during the replace operation.
- **end** (optional): Closing command to signal end of element

Example 1:

Create/Default interfaces:

```
interfaces:
    _process: unnecessary
    interface:
        _process:
          . mode: container
            key_value: "interface {{ interface_key }}\n"
            negate: "{{ 'no' if interface_key[0:4] in ['Port', 'Loop'] else
↪'default' }} interface {{ interface_key }}\n"
            end: "    exit\n"
```

Example 2:

Configure IP addresses. As the parent interface is defaulted already, don't do it again:

```
address:
    _process:
      . mode: container
        key_value: "    ip address {{ model.config.ip }} {{ model.config.
↪prefix_length|cidr_to_netmask }}{{ ' secondary' if model.config.secondary␣
↪else '' }}\n"
        negate: "    default ip address {{ model.config.ip }} {{ model.
↪config.prefix_length|cidr_to_netmask }}{{ ' secondary' if model.config.
↪secondary else '' }}\n"
        replace: false
```

Leaf - element

Configures an attribute.

Arguments:

- **value** (mandatory): How to configure the attribute
- **negate** (mandatory): How to default the attribute

Example 1:

Configure description:

```
description:
    _process:
        - mode: element
          value: "    description {{ model }}\n"
          negate: "    default description"
```

Example 2:

Configure an IP address borrowing values from other fields:

```
address:
    _process: unnecessary
    config:
        _process: unnecessary
        ip:
            _process: unnecessary
        prefix_length:
```

```
          _process:
              - mode: element
                value: "    ip address {{ model._parent.ip }}/{{ model }} {
↪{ 'secondary' if model._parent.secondary else '' }}\n"
                negate: "    default ip address {{ model._parent.ip }}/{{␣
↪model }} {{ 'secondary' if model._parent.secondary else '' }}\n"
```

API

Models

Models are generated by `pyangbind` so it's better to check it's documentation for up to date information: http://pynms.io/pyangbind/generic_methods/

Utils

`napalm_yang.utils.`**`model_to_dict`**(*model*, *mode=''*)

Given a model, return a representation of the model in a dict.

This is mostly useful to have a quick visual represenation of the model.

Parameters

- **`model`** (*`PybindBase`*) – Model to transform.
- **`mode`** (*`string`*) – Whether to print config, state or all elements ("" for all)

Returns A dictionary representing the model.

Return type dict

Examples

```
>>> config = napalm_yang.base.Root()
>>>
>>> # Adding models to the object
>>> config.add_model(napalm_yang.models.openconfig_interfaces())
>>> config.add_model(napalm_yang.models.openconfig_vlan())
>>> # Printing the model in a human readable format
>>> pretty_print(napalm_yang.utils.model_to_dict(config))
>>> {
>>>     "openconfig-interfaces:interfaces [rw]": {
>>>         "interface [rw]": {
>>>             "config [rw]": {
>>>                 "description [rw]": "string",
>>>                 "enabled [rw]": "boolean",
>>>                 "mtu [rw]": "uint16",
>>>                 "name [rw]": "string",
>>>                 "type [rw]": "identityref"
>>>             },
>>>             "hold_time [rw]": {
>>>                 "config [rw]": {
>>>                     "down [rw]": "uint32",
```

```
>>>                         "up [rw]": "uint32"
    (trimmed for clarity)
```

napalm_yang.utils.**diff** (*f*, *s*)

Given two models, return the difference between them.

Parameters

- **f** (*Pybindbase*) – First element.
- **s** (*Pybindbase*) – Second element.

Returns A dictionary highlighting the differences.

Return type dict

Examples

```
>>> diff = napalm_yang.utils.diff(candidate, running)
>>> pretty_print(diff)
>>> {
>>>     "interfaces": {
>>>         "interface": {
>>>             "both": {
>>>                 "Port-Channel1": {
>>>                     "config": {
>>>                         "mtu": {
>>>                             "first": "0",
>>>                             "second": "9000"
>>>                         }
>>>                     }
>>>                 }
>>>             },
>>>             "first_only": [
>>>                 "Loopback0"
>>>             ],
>>>             "second_only": [
>>>                 "Loopback1"
>>>             ]
>>>         }
>>>     }
>>> }
```

Root

class napalm_yang.base.**Root**

Bases: object

This is a container you can use as root for your other models.

Examples

```
>>> config = napalm_yang.base.Root()
>>>
>>> # Adding models to the object
```

```
>>> config.add_model(napalm_yang.models.openconfig_interfaces())
>>> config.add_model(napalm_yang.models.openconfig_vlan())
```

add_model (*model*, *force=False*)
Add a model.

The model will be asssigned to a class attribute with the YANG name of the model.

Parameters

- **model** (*PybindBase*) – Model to add.
- **force** (*bool*) – If not set, verify the model is in SUPPORTED_MODELS

Examples

```
>>> import napalm_yang
>>> config = napalm_yang.base.Root()
>>> config.add_model(napalm_yang.models.openconfig_interfaces)
>>> config.interfaces
<pyangbind.lib.yangtypes.YANGBaseClass object at 0x10bef6680>
```

compliance_report (*validation_file='validate.yml'*)
Return a compliance report. Verify that the device complies with the given validation file and writes a compliance report file. See https://napalm.readthedocs.io/en/latest/validate.html.

elements ()

get (*filter=False*)
Returns a dictionary with the values of the model. Note that the values of the leafs are YANG classes.

Parameters **filter** (*bool*) – If set to `True`, show only values that have been set.

Returns A dictionary with the values of the model.

Return type dict

Example

```
>>> pretty_print(config.get(filter=True))
>>> {
>>>     "interfaces": {
>>>         "interface": {
>>>             "et1": {
>>>                 "config": {
>>>                     "description": "My description",
>>>                     "mtu": 1500
>>>                 },
>>>                 "name": "et1"
>>>             },
>>>             "et2": {
>>>                 "config": {
>>>                     "description": "Another description",
>>>                     "mtu": 9000
>>>                 },
>>>                 "name": "et2"
>>>             }
```

```
>>>             }
>>>         }
>>> }
```

load_dict (*data*, *overwrite=False*)
Load a dictionary into the model.

Parameters

- **data** (*dict*) – Dictionary to loead
- **overwrite** (*bool*) – Whether the data present in the model should be overwritten by the
- **in the dictor not.** (*data*) –

Examples

```
>>> vlans_dict = {
>>>     "vlans": { "vlan": { 100: {
>>>                         "config": {
>>>                             "vlan_id": 100, "name": "production"}},
>>>                     200: {
>>>                         "config": {
>>>                             "vlan_id": 200, "name": "dev"}}}}}
>>> config.load_dict(vlans_dict)
>>> print(config.vlans.vlan.keys())
... [200, 100]
>>> print(100, config.vlans.vlan[100].config.name)
... (100, u'production')
>>> print(200, config.vlans.vlan[200].config.name)
... (200, u'dev')
```

parse_config (*device=None*, *profile=None*, *native=None*, *attrs=None*)
Parse native configuration and load it into the corresponding models. Only models that have been added to the root object will be parsed.

If `native` is passed to the method that's what we will parse, otherwise, we will use the `device` to retrieve it.

Parameters

- **device** (*NetworkDriver*) – Device to load the configuration from.
- **profile** (*list*) – Profiles that the device supports. If no `profile` is passed it will be read from `device`.
- **native** (*list of strings*) – Native configuration to parse.

Examples

```
>>> # Load from device
>>> running_config = napalm_yang.base.Root()
>>> running_config.add_model(napalm_yang.models.openconfig_interfaces)
>>> running_config.parse_config(device=d)
```

```
>>> # Load from file
>>> with open("junos.config", "r") as f:
>>>     config = f.read()
>>>
>>> running_config = napalm_yang.base.Root()
>>> running_config.add_model(napalm_yang.models.openconfig_interfaces)
>>> running_config.parse_config(native=config, profile="junos")
```

parse_state (*device=None*, *profile=None*, *native=None*, *attrs=None*)

Parse native state and load it into the corresponding models. Only models that have been added to the root object will be parsed.

If `native` is passed to the method that's what we will parse, otherwise, we will use the `device` to retrieve it.

Parameters

- **device** (*NetworkDriver*) – Device to load the configuration from.
- **profile** (*list*) – Profiles that the device supports. If no `profile` is passed it will be read from `device`.
- **native** (*list string*) – Native output to parse.

Examples

```
>>> # Load from device
>>> state = napalm_yang.base.Root()
>>> state.add_model(napalm_yang.models.openconfig_interfaces)
>>> state.parse_config(device=d)
```

```
>>> # Load from file
>>> with open("junos.state", "r") as f:
>>>     state_data = f.read()
>>>
>>> state = napalm_yang.base.Root()
>>> state.add_model(napalm_yang.models.openconfig_interfaces)
>>> state.parse_config(native=state_data, profile="junos")
```

to_dict (*filter=True*)

Returns a dictionary with the values of the model. Note that the values of the leafs are evaluated to python types.

Parameters **filter** (*bool*) – If set to `True`, show only values that have been set.

Returns A dictionary with the values of the model.

Return type dict

Example

```
>>> pretty_print(config.to_dict(filter=True))
>>> {
>>>     "interfaces": {
>>>         "interface": {
>>>             "et1": {
```

```
>>>                 "config": {
>>>                     "description": "My description",
>>>                     "mtu": 1500
>>>                 },
>>>                 "name": "et1"
>>>             },
>>>             "et2": {
>>>                 "config": {
>>>                     "description": "Another description",
>>>                     "mtu": 9000
>>>                 },
>>>                 "name": "et2"
>>>             }
>>>         }
>>>     }
>>> }
```

translate_config (*profile*, *merge=None*, *replace=None*)

Translate the object to native configuration.

In this context, merge and replace means the following:

- **Merge** - Elements that exist in both `self` and `merge` will use by default the values in `merge` unless `self` specifies a new one. Elements that exist only in `self` will be translated as they are and elements present only in `merge` will be removed.
- **Replace** - All the elements in `replace` will either be removed or replaced by elements in `self`.

You can specify one of `merge`, `replace` or none of them. If none of them are set we will just translate configuration.

Parameters

- **profile** (*list*) – Which profiles to use.
- **merge** (`Root`) – Object we want to merge with.
- **replace** (`Root`) – Object we want to replace.

Jinja2 Filters

IP address

FAQ

Some YAML files are insanely largely. Can I break them down into multiple files?

Yes, you can with the `!include relative/path/to/file.yaml` directive. For example:

```
# ./main.yaml
my_key:
    blah: asdasdasd
    bleh: !include includes/bleh.yaml

# ./includes/bleh.yaml
```

```
qwe: 1
asd: 2
```

Will result in the final object:

```
my_key:
    blah: asdasdasd
    bleh:
        qwe: 1
        asd: 2
```

Python Module Index

n

Index