Jinja for network engineers: Cheatsheet

Summary

1 Sandbox tools 2
2 Basic Syntax 2
2.1 Variables naming 2
2.2 Output formatting 3
2.2.1 Line Spaces 3
2.2.2 C printf-style formatting 3
2.2.3 Python-style formatting 4
2.3 Jinja operators 5
2.4 Tests 6
2.5 Useful Filters 6
2.5.1 Sort filter 7
2.5.2 Map filter 7
2.5.3 Ipaddr filter 8
3 Navigate into data structures 10
3.1 Lists and Tuples 10
3.1.1 Parse a list 10
3.1.2 Parse two lists in parallel 10
3.2 Dictionaries 11
3.3 List of dictionaries 12
4 Reorganize data structures 13
4.1 Merge multiple dictionaries 13
4.2 Group Items by Attribute 13
5 Json_query 14
6 Data Set 16

Complete Data set is at the end of the document

1 Sandbox tools

online	https://j2live.ttl255.com/ with **Ansible** checkbox checked
offline	https://github.com/PJO2/jinja2-live

2 Basic Syntax

`{{ }}` is for substitutions	`{{ variable }}`
`.attr` or `["attr"]` access to the attr key	`{{ dict.attribute }}`
`{% %}` is for statements	set, for-endfor loops, if-elif-else-endif tests
`{# #}` is for comments	`{# loop for every IPv6 interfaces #}`

`{% macro myfunc() %}` to declare macros with or without parameters	`{%- macro myfunc(param1, param2, …) %}`
	`Hello {{ param1 }}, {{ param2 }}`
	`{%- endmacro %}`
	`{{ myfunc ("world", "this is Jinja") }}`

`{% raw %}` to suspend jinja interpreter	`{% raw %}`
	`{{ this will not be substituted }}`
	`{% this will not give a syntax error %}`
	`{% endraw %}`

`{% include %}` to insert a j2 file to the template	`{% include 'routing/bgp.j2' %}`
`{% import %}` to add a library of macros	`{% import 'macro/banners.j2' as ban %}`

2.1 Variables naming

Avoid using – and / in key names: Although they are valid in YAML, they prevent the use of dot notation

```
{{ interface.ip-addr }}      is interpreted as {{ (interface.ip) - addr }}.
{{ interface['ip-addr'] }}   is correct
```

As a best practice, add an **s** at the end of variable names representing **iterable objects** like lists or dictionaries — to make their usage immediately obvious in loops:

```
{% for vrf in network.vrfs %} {# parse a list #}
…
{% for ce in network.sites.CEs %} {# parse a dict #}
…
```

2.2 Output formatting

2.2.1 Line Spaces

`{%- …`	suppress the line break before
`… -%}`	suppress the line break after
`{%- … -%}`	suppress both line breaks Usually this is the correct way

Examples:

`{%- for i in [1,2,3,4] %}`
`{{ i }}`
`{%- endfor %}`
	1
	2
	3
	4

`{%- for i in [1,2,3,4] -%}`
`{{ i }}`
`{%- endfor %}`
	1234

2.2.2 C printf-style formatting

The % operator works similarly to C-style printf formatting :

```
{{ "%s is %d !" % ("Answer", 42) }}
Output:
Answer is 42 !
```

Formatting specifications:

`"%s" % var`	character string
`"%d" % var`	decimal integer
`"%x" % var`	hexadecimal integer
`"%10s" % var`	left padding with spaces
`"%04d" % var`	left padding with zeros
`"%*s" % (n, var)`	left padding with spaces until n chars, does not truncate var
`"%0*d" % (n, var)`	left padding with zeros until n chars, does not truncate var
`"%*.*s" % (n, n, var)`	left padding with spaces until n chars, truncates var if len(var) > n

Example: generate a Route Distinguisher in the form
<AS>:1<VRF ID padded to 4 digits><PE ID padded to 3 digits>

```
vrf definition {{ vrf.name }}
  rd {{ "%d:1%04d%03d" % (bgp.as, vrf.id, pe.id) }}
```

```
Input:
vrf:
   name: MGMT
   id:   950
bgp:
   as: 65000
pe:
   name: Lyon
   id:   69

Output:
vrf definition MGMT
  rd 65000:10950069
```

Example: format output like the CLI command `show interfaces accounting`

```
{%-  for itf, acc in interfaces.items() %}
{{ itf }}
  {%- for protocol, stats in acc.items() %}
{{ "%14s %9s %10d %10d %10d %10d" % (' ', protocol, stats[0], stats[1], stats[2], stats[3]) }}
  {%-endfor %}{# for each protocol #}
{%-endfor %} {# for each interface #}

Input:
interfaces:
    eth0/0:
      IP: [ 105, 10328, 97, 420 ]
      MPLS: [ 27, 2828, 58, 7446 ]

Output:
eth0/0
                       IP        105      10328         97        420
                     MPLS         27       2828         58       7446
```

2.2.3 Python-style formatting

Applies a format with the python f-string method.

```
{{ "{} is {} !".format ("Answer", 42) }}
```

Using named variables:

```
{{ "{txt} is {intg} !".format (intg=42, txt="Answer") }}
```

Format of Input string specification:

```
{ [#param]:[position][#length][type][name] }
```

Common Format Specifiers:

`{:<14}`	right (default) padding
`{:^14}`	center padding
`{:>14}`	left padding
`{:0>4d}`	force integer conversion, left padding with zero
`{:0>4X}`	convert to hexa, left pad with 0 (dump two bytes in hexa)

Example:

```
{%- for ce in network.sites.CEs %}
{{ "CE: {:>10} is connected to PE {}" . format (ce.name, ce.WAN.pe) }}
{%- endfor %}

Output:
CE:     Eiffel is connected to PE Paris
CE:   Chapelle is connected to PE Paris
CE:    Lumiere is connected to PE Lyon
CE:      Amphi is connected to PE Lyon
```

Mimic `show interfaces accounting`. Unlike the % operator, format allows the * operator to flatten the list stats

```
{%-  for itf, acc in interfaces.items() %}
{{ itf }}
  {%- for protocol, stats in acc.items() %}
{{ "{:14s} {:>9s} {:10d} {:10d} {:10d} {:10d}".format(' ', protocol, *stats) }}
  {%-endfor %}{# for each protocol #}
{%-endfor %} {# for each interface #}
```

2.3 Jinja operators

`~`	string concatenation
`//`	for Euclidian division
`%`	modulo for integers, formatting for strings
`+-*`	are the standards operators
`/`	may return a float (ex 4/3)
`and, or, not`	classic boolean operators
`a if test else b`	ternary operator with poor readability `{{ interface.ip if interface.ip is defined else "dhcp" }}` for clarity, use **default** or Ansible **ternary** filter

2.4 Tests

`is defined/undefined`	`{% if is defined interface.ip %}`
`equalto`	used with selectattr `{% for itf in itfs | selectattr("name","equalto","Loopback") %}`
`in`	`{{ ("ip" in interface) | ternary (interface.ip, "dhcp") }}`

2.5 Useful Filters

Jinja's filters are calls to Python functions or methods:

```
{{ variable | filter(args) }}   ➔ print ( filter(variable, args) )
{{ instance | filter(args) }}     print ( instance.filter(args) )
```

The filters below are ordered by usefulness, based on my personal experience as a network engineer. For a complete list, refer to the official Jinja documentation.

`default`	return the left value if defined, else the default argument `{{ interface.ip | default ("dhcp") }}`
`ternary (ansible)`	`{{ interface.ip is defined | ternary (interface.ip, "dhcp")}}`
`selectattr/rejectattr`	Filters an iterable object based on a test applied to each item's attribute
`format`	applies a printf-style format. Prefer using either **%** operator or .**format()**, see 2.2.2 and 2.2.3
`int`	converts to integer, useful to force + to act as an integer addition or to get 0 if the input can not be converted
`first/last`	keep the first/last item in a list
`items()`	split a dict into key, value
`list`	converts an iterator into a list
`join`	converts a list into a string and add an optional separator
`map`	process a list
`replace`	replace substring with another one
`regex_replace`	replace using regex with captures and back references. remember to escape '\' : (\\d, \\1, \\2, ...)
`type_debug (Ansible)`	return the python type of the variable
`from_json`	converts a json string to a dictionary
`to_json/to_xml`	Converts a dictionary to a JSON or XML string, which is useful when crafting payloads for APIs.
`unique`	removes duplicates

`sort/dictsort`	sort a list/dictionnary
`groupby`	Groups items into **key-value pairs**, creating a dict.
`select`	filter a list
`upper/lower`	convert string to uppercase/lowercas
`basename/dirname`	extracts filename from a path

2.5.1 Sort filter

Use sort/dictsort filters if you need to sort a list/dictionary.

```
{%- for vrf in network.vrfs | sort("id") %}
{{ vrf }}
{%- endfor %}
```

2.5.2 Map filter

map applies a filter to each element in a list.

```
{{ network.vrfs | map(attribute='name') | list }}

Output:
 ['Orange', 'Green', 'MGMT']
```

new configuration after migration 1Gb->10Gb

```
{%- for te_itf in network.MPLS.PEs[0].mpls_interfaces | map ('replace', 'Gi', 'Te') %}
{{ te_itf }}
{%- endfor %}
```

new configuration after device migration

```
{%- for itf in network.MPLS.PEs[0].mpls_interfaces |
               map ('regex_replace', 'Gi(\\d)/(\\d)', 'Te0/\\1/0/\\2') %}
{{ itf }}
{%- endfor %}
```

2.5.3 Ipaddr filter

Usage:

```
{{ address | ipaddr('test or operation') }}
```

List of ip filters

```
{% set ips = ['192.2.0.2/30', '192.2.0.99/24', '2001:db8::1/64' ] %}
{% for query in [   '1', '-1', '10',
                    'succ', 'prev', 'peer',
                    'type', 'bool', 'int', 'version', 'size',
                    'address', 'ip', 'host',
                    'network', 'subnet', 'prefix', 'netmask', 'hostmask',
                    'hostnet', 'router', 'gateway', 'gw', 'host/prefix', 'address/prefix',
                    'broadcast', 'unicast', 'multicast', 'private', 'public',
                    'loopback', 'lo',
                    'revdns', 'wrap', 'ipv6', 'v6', 'ipv4', 'v4', 'cidr', 'net',
] %}
{{ "{}\t{}".format(query, ips | map ('ipaddr', query ) | join('\t')) }}
{%- endfor %}
```

filter	applied to		
ipaddr(?)	192.2.0.1/30	192.2.0.1/24	2001:db8::1/64
1	192.2.0.1/30	192.2.0.1/24	2001:db8::1/64
-1	192.2.0.3/30	192.2.0.255/24	2001:db8::ffff:ffff:ffff:ffff/64
10	False	192.2.0.10/24	2001:db8::a/64
succ/next_usable	False	192.2.0.100/24	2001:db8::2/64
prev	192.2.0.1/30	192.2.0.98/24	False
peer	192.2.0.1	False	False
type	address	address	address
bool	True	True	True
int	3221356546/30	3221356643/24	42540766411282592856903984951653826561/64
version	4	4	6
size	4	256	18446744073709551616
address	192.2.0.2	192.2.0.99	2001:db8::1
ip	192.2.0.2	192.2.0.99	2001:db8::1
host	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
network	192.2.0.0	192.2.0.0	2001:db8::
subnet	192.2.0.0/30	192.2.0.0/24	2001:db8::/64
prefix	30	24	64
netmask	255.255.255.252	255.255.255.0	ffff:ffff:ffff:ffff::
hostmask	0.0.0.3	0.0.0.255	::ffff:ffff:ffff:ffff
hostnet	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
router	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
gateway	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
gw	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
host/prefix	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64

address/prefix	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
broadcast	192.2.0.3	192.2.0.255	2001:db8::ffff:ffff:ffff:ffff
unicast	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
multicast	False	False	False
private	False	False	False
public	192.2.0.2/30	192.2.0.99/24	2001:db8::1/64
loopback	False	False	False
lo	False	False	False
wrap	192.2.0.2/30	192.2.0.99/24	[2001:db8::1]/64
ipv6	::ffff:192.2.0.2/126	::ffff:192.2.0.99/120	2001:db8::1/64
v6	::ffff:192.2.0.2/126	::ffff:192.2.0.99/120	2001:db8::1/64
ipv4	192.2.0.2/30	192.2.0.99/24	False

Use Case: Advertise IPv4 and IPv6 Routes with `ipaddr`, Loops, and Macros:

```
{%- macro BGP_Advertise (as, vrf, subnets) %}
router bgp {{ as }}
  address-family ipv4 vrf {{ vrf }}
  {%- for subnet in subnets if subnet | ipaddr('version') == 4 %}
  network {{ subnet | ipaddr('network') }} mask {{ subnet | ipaddr('netmask') }}
  {%- endfor %}
  exit-address-family
  !
  address-family ipv6 vrf {{ vrf }}
  {%- for subnet in subnets if subnet | ipaddr('version') == 6 %}
  network {{ subnet }}
  {%- endfor %}
  exit-address-family
{% endmacro %}

{{ BGP_Advertise (65000, 'MGMT', ['10.10.10.0/24', '10.10.1.0/24', '2001:db8:ad64::/64'] ) }}

Output:
router bgp 65000
  address-family ipv4 vrf MGMT
  network 10.10.10.0 mask 255.255.255.0
  network 10.10.1.0 mask 255.255.255.0
  exit-address-family
  !
  address-family ipv6 vrf MGMT
  network 2001:db8:ad64::/64
  exit-address-family
```

3 Navigate into data structures

This chapter shows how to efficiently navigate and manipulate structured data like lists and dictionaries, a key skill for building dynamic and context-aware network templates.

3.1 Lists and Tuples

3.1.1 Parse a list

Use a for loop

```
{%- for vrf in network.vrfs %}
vrf {{ vrf.name }} has id: {{ vrf.id }}
{%- endfor %}

Output:
vrf Orange has id: 1
vrf Green has id: 2
vrf MGMT has id: 950
```

A powerful feature of Jinja is the ability to add if conditions directly inside for loops, allowing you to filter elements while iterating.

```
// retrieve only customers VRF
{%- for vrf in network.vrfs if vrf.id<950 %}
vrf {{ vrf.name }} has id: {{ vrf.id }}
{%- endfor %}
Output:
vrf Orange has id: 1
vrf Green has id: 2
```

3.1.2 Parse two lists in parallel

Jinja allows you to iterate over multiple lists at the same time using zip or zip_longest (Ansible only).

```
// parse two lists in //
{%- for ce, pe in network.sites.CEs | zip(network.MPLS.PEs) %}
{{ ce.name }} and {{ pe.name }}
{%- endfor %}

Output:
Eiffel and Lyon
Chapelle and Paris


{%- for ce, pe in network.sites.CEs | zip_longest (network.MPLS.PEs) %}
```

```
{{ ce.name }} and {{ pe.name }}
{%- endfor %}

Output:
Eiffel and Lyon
Chapelle and Paris
Lumiere and
Amphi and
```

3.2 Dictionaries

When working with dictionaries in Jinja, .items() is the common way to loop through key–value pairs.

```
{%- for key, value in network.items() %}
{{ key }} --> {{ value }}
{%- endfor %}

Output:
vrfs --> [{'name': 'Orange', 'id': 1}, {'name': 'Green', 'id': 2}, {'name': 'MGMT', …
MPLS --> {'bgp': {'as': 65000}, 'PEs': [{'name': 'Lyon', 'id': 6, 'mpls_interfaces':…
sites --> {'CEs': [{'name': 'Eiffel', 'vrf': 'Orange', 'WAN': {'pe': 'Paris', 'inter…
```

Similar to list, the for loop allows conditions. If we move vrfs from a list to a dictionary, the previous example would be:

```
{% set vrfs = {
  'Orange': 1,
  'Green': 2,
  'MGMT': 950
} %}
{%- for name, id in vrfs.items() if id<950 %}
{{ name }} --> {{ id }}
{%- endfor %}

Output:
Orange --> 1
Green --> 2
```

You can use **update** to add or modify keys in a dictionary at runtime. This is particularly useful when enriching YAML data before rendering.

Here we add the vrf id attribute if it is not already defined.

```
{% for vrf in network.vrfs %}
  {% if vrf.id is not defined %}
    {% set _ = vrf.update({'id': loop.index}) %}
  {% endif %}
```

```
{% endfor %}
{{ network.vrfs }}

Input:
Remove id for vrf Orange/Green

Output:
 [{'name': 'Orange', 'id': 1}, {'name': 'Green', 'id': 2}, {'name': 'MGMT', 'id': 950}]
```

3.3 List of dictionaries

When working with a list of dictionaries, you can use **selectattr** to filter items based on an attribute's value.

Example: extract customer VRF names (id less than vrf MGMT)

```
{{ network.vrfs
   | selectattr('id', 'lt', 950)
   | map(attribute='name')
   | join(', ') }}

Output:
Orange, Green
```

List CE devices VRF by VRF :

```
{%- for vrf_name in network.vrfs | map(attribute="name") %}
VRF {{ vrf_name }}:
  {%- for ce in network.sites.CEs if ce.vrf == vrf_name %}
  - {{ ce.name }}
  {%- endfor %}
{%- endfor %}

Output:
VRF Orange:
  - Eiffel
  - Lumiere
VRF Green:
  - Chapelle
  - Amphi
VRF MGMT:
```

4 Reorganize data structures

4.1 Merge multiple dictionaries

Use the + operator to concatenate two list.

```
{%- for dev in network.sites.CEs + network.MPLS.PEs %}
{{ dev.name }}
{%- endfor %}

Output:
Eiffel
Chapelle
Lumiere
Amphi
Lyon
Paris
```

4.2 Group Items by Attribute

The **groupby** filter groups a list of dictionaries into a structure organized by a shared attribute. It's especially useful for nested structures.

Useful to group CEs by their associated PE:

```
{%- for pe, devs in network.sites.CEs | groupby ("WAN.pe") %}
{{ pe }} -> {{ devs | map(attribute="name") | join(', ') }}
{%- endfor %}

Output:
Lyon -> Lumiere, Amphi
Paris -> Eiffel, Chapelle
```

Or list CE belonging to each VRF in a single loop:

```
{%- for vrf, ces in network.sites.CEs | groupby ("vrf") %}
VRF {{ vrf }}
   - {{ ces | map(attribute="name") | join("\n   - ") }}
{%- endfor %}
```

5 Json_query

The **json_query** filter is based on the JMESPath syntax and is ideal when **selectattr** becomes too limited :

- Traverse nested structures (list of lists, or nested dictionaries)
- Filter multiple attributes
- Extract only specific key from a dictionary

Usage:

```
{{ data | json_query('expression') }}
```

expression has symbols, selectors and filters

Symbols:

```
*      wildcard : every attributes or complete list
?      filter
@      current item in selection
`      (backtick) literal for strings/numbers
```

Selectors are:

```
foo    Select key foodata.foo
foo.barNested key    data.foo.bar
foo[0] First element in a list     data.devices[0]
foo[*] All elements (wildcard projection) devices[*].hostname
[*].barProject field bar from all elements devices[*].ip
```

Filters on items or attributes:

```
[?contains(@, `:`)]  string contains ':'
contains/starts_with/ends_with/length

[?networks]   has networks fields
[?starts_with(name, `Par`)] string test on attribute name
[?name == 'value']   Equal
[?name != 'value']   Not equal
[?id < 950]   Greater than
[?name == 'Paris' && id==7] AND condition
[?
```

Tips: use [?condition] to filter lists, Use backticks (`) for strings and numbers inside filters.

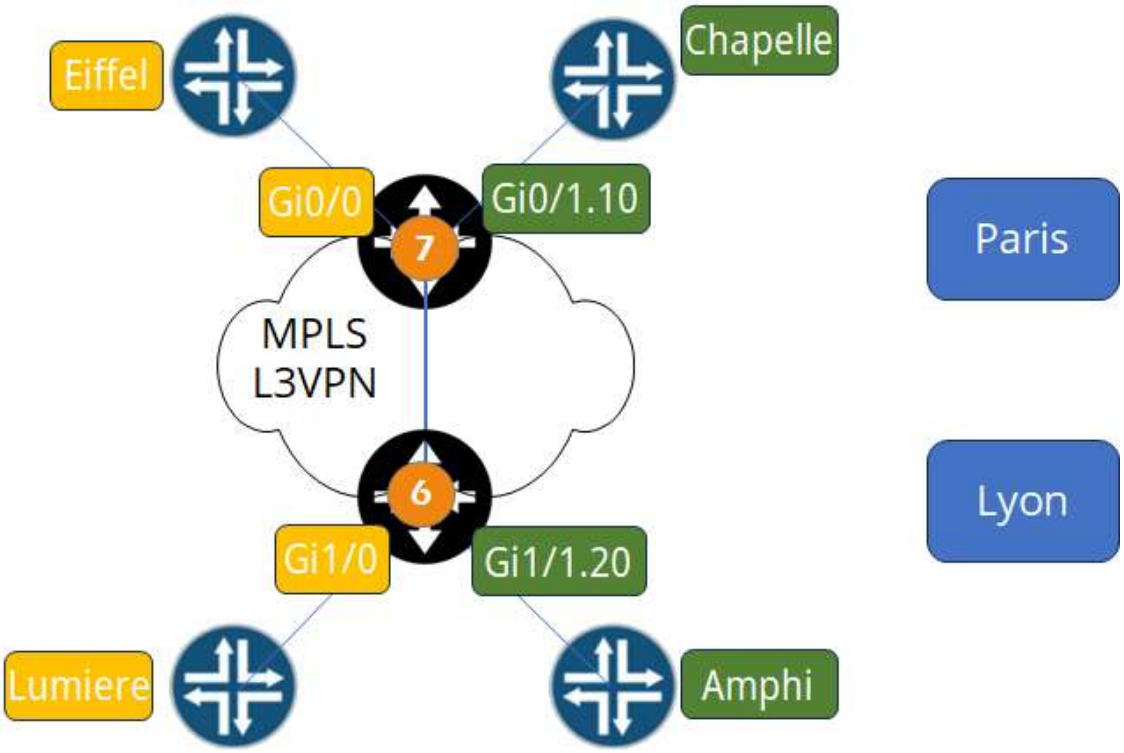
Nested Lists: [*] keeps the structures, [] flattens it

```
[*].list[*]   Flatten sublists
[].[a, b]     Return multiple fields as tuples
```

Some other great 😊 examples :

```
// Get names of vrf (traversing the vrfs list)
{{ network.vrfs | json_query('[].name') }}
// same as {{ network.vrfs | map (attribute="name") | list }}

// Get WAN ip addresses
{{ network.sites.CEs | json_query("[].WAN.ip") }}
// same as {{ network.sites.CEs | map(attribute="WAN") | map(attribute="ip") | list }}

// Get the ID of the VRF named "Green" (Combine filtering + extraction)
{{ network.vrfs | json_query("[?name=='Green'].id") | first }}
//same as {%- for vrf in network.vrfs if vrf.name=='Green' %}
          {{vrf.id}}
          {%- endfor %}

# Flatten sublists (traverse multiple lists)
{{ network.sites.CEs | json_query("[*].networks[*]") }}  returns a list of lists
{{ network.sites.CEs | json_query("[*].networks[*]") | flatten }} is a list
{{ network.sites.CEs | json_query("[].networks[]") }}   is also a list

// Get the list of all routes (whatever the CE)
{{ network.sites.CEs | json_query("[].networks[]") }}

// get all ipv6 routes
{{ network.sites.CEs | json_query('[].networks[?contains(@, `:`)]') | flatten }}

// retrieve peer mpls PE
// PE points on Paris
{% set PE=network.MPLS.PEs | selectattr('name', 'equalto', current_pe) | first %}
// search for peer
{{ network.MPLS.PEs | json_query('[?name == `' ~PE.name ~'`].mpls_interfaces') | first }}
// same as {{ devices.PEs | json_query('[?name==`' + currentPE + '`]') | first }}
// get only the key
{{ network.MPLS.PEs | json_query(
           '[?name == `' ~PE.name ~'`].mpls_interfaces | [0] | keys(@) | [0] ') }}
```

6 Data Set

```yaml
current_pe: Paris
network:
  vrfs:
      - name: Orange
        id:   1
      - name: Green
        id:   2
      - name: MGMT
        id:   950
  MPLS:
     bgp:
       as: 65000
     PEs:
       - name: Lyon
         id: 6
         mpls_interfaces:
            Te0/0:
                ip: 172.20.0.0/31
                peer: Paris
       - name: Paris
         id: 7
         mpls_interfaces:
            Te1/0:
                peer: Lyon
                ip: 172.20.0.1/31
  sites:
     CEs:
      - name: Eiffel
        vrf: Orange
        WAN:
           pe: Paris
           interface: Gi0/0
           ip: 10.0.0.0/30
        networks: [ '192.168.10.0/24', '2001:db8:10::/64' ]
      - name: Chapelle
        vrf: Green
        WAN:
           pe: Paris
           interface: Gi0/1.10
           ip: 10.0.0.4/30
      - name: Lumiere
        vrf: Orange
        WAN:
           pe: Lyon
           interface: Gi1/0
           ip: 10.0.1.0/30
      - name: Amphi
        vrf: Green
        WAN:
           pe: Lyon
           interface: Gi0/1.20
           ip: 10.0.4.0/30
```

If you prefer visual context, here is the network topology behind the dataset: