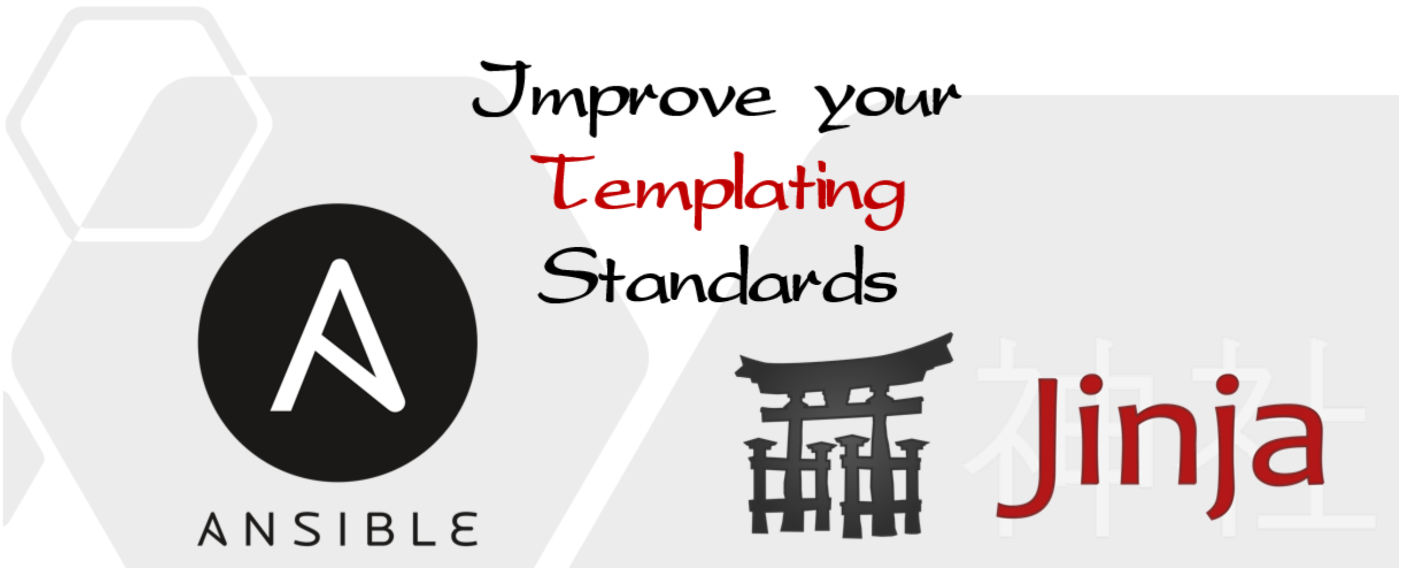
Jinja2:
Data model transformation



In network automation, the most critical step is to build a proper data model.

This data model must follow some design rules:

- Completeness
- No redundancy
- Modularity to allow easy changes
- Hierarchy to allow inheritance of the data belonging to a higher layer

For now, as long as your project goes beyond elementary scripts, the perfect data model does not exist and we have to deal with imperfections, which may dramatically affect the quality of the last link of the chain: your templates.

In this article, I mention several ways of transforming the data model inside the template, improving the template quality.

Of course, the goal is not to create new variables with the {% set %} command, but to convert raw data into structured data, to group structured data, or to reverse the structure of the data when needed. Most of the time, the magic happens inside an existing for loop.

Group unstructured data inside lists or dictionaries

If your data source is an excel file, using one row per device, it is very likely that some of your data look like the following example:

```
lan1:   eth 0/1
ip1:    10.0.0.1
mask1:  /24
lan2:   eth 0/2
ip2:    20.0.0.1
mask2:  /24
lan3:   eth 0/3
```

```
ip3:    30.0.0.1
mask3:  /16
lan4:
ip4:
mask4
```

You may use the following template to deal with, but it is error prone, redundant and hardly extensible.

```
interface {{ lan1 }}
  ip address {{ ip1 }}{{ mask1}}
interface {{ lan2 }}
  ip address {{ ip2 }}{{ mask2}}
…
```

Morever, if you add some checking to deal with empty data, the template becomes quickly less than attractive:

```
{% if lan1 %}
interface {{ lan1 }}
  ip address {{ ip1 }}{{ mask1 }}
   {% if lan2 %}
..
      {% if lan4 %}
interface {{ lan4 }}
  ip address {{ ip4 }}{{ mask4 }}
      {% endif %}
   {% endif %}
{% endif %}
```

What would a relevant data model for our use case look like?

Here, the data model that we would like to deal with, is:

```
interfaces :  [ name, ip, mask ]
```

The template would have been:

```
{% for itf in interfaces %}
interface {{ itf.name }}
  ip address {{ itf.ip }}{{ itf.mask}}
{% endfor %}
```

Simple, readable, extensible and without code redundancy.

Now, we see that a simple encapsulation of our data into lists is enough to bring us near a proper data model :

```
{% for name, ip, mask in     [    [ lan1, ip1, mask1 ],
                                  [ lan2, ip2, mask2 ],
                                  [ lan3, ip3, mask3 ],
                                  [ lan4, ip4, mask4 ]   ] %}
interface {{ name  }}
  ip address {{ ip }}{{ mask }}
{% endfor %}
```

The resulting template is more compact, more readable and without redundancy.

Dealing with empty values is simple enough, using the postfix if statement inside the for loop :

```
{% for name, ip, mask in     [    [ lan1, ip1, mask1 ],
                                  [ lan2, ip2, mask2 ],
                                  [ lan3, ip3, mask3 ],
                                  [ lan4, ip4, mask4 ]  ]
                  if name -%}
interface {{ name  }}
  ip address {{ ip }}{{ mask }}
{% endfor %}
```

Alternatively, the template can be written with lists of dictionaries instead of lists of lists:

```
{% for itf in     [  { 'name':  lan1,  'ip': ip1, 'mask':  mask1 },
                     { 'name':  lan2,  'ip': ip2, 'mask':  mask2 },
                     { 'name':  lan3,  'ip': ip3, 'mask':  mask3 },  ]
                  if itf.name -%}
interface {{ itf.name  }}
  ip address {{ itf.ip }}{{ itf.mask }}
{% endfor %}
```

Or even better using the zip filter (see below) :

```
{% for itf in [
      dict ( ['name', 'ip', 'mask'] | zip([ lan1, ip1, mask1]) ),
      dict ( ['name', 'ip', 'mask'] | zip([ lan2, ip2, mask2]) ),
      dict ( ['name', 'ip', 'mask'] | zip([ lan3, ip3, mask3]) ),
      dict ( ['name', 'ip', 'mask'] | zip([ lan4, ip4, mask4]) ),
               ]   if itf.name -%}
interface {{ itf.name  }}
  ip address {{ itf.ip }}{{ itf.mask }}
{% endfor %}
```

Iterate over multiple lists simultaneously

An improvement to the initial model is to group the raw data into several lists:

```
lans:  [ eth0/1, eth0/2, eth0/3 ]
ips:   [ 10.0.0.1, 20.0.0.1, 30.0.0.1]
masks: [ /24, /24, /16]
```

A working template may look like this one :

```
{% for lan in lans %}
interface {{ lan }}
  ip address {{ ips[loop.index0] }}{{ masks[loop.index0] }}
{% endfor %}
```

However, it imports some new variable, makes one list distinctive and contains some hidden time bombs (forget the 0 in index0 and wait for a non /24 mask…).

Again, instead of blaming the model, we may transform the data to our target model. Here using zip will iterate on the three lists simultaneously:

```
{% for lan, ip, mask in lans|zip(ips, masks) %}
interface {{ lan }}
  ip address {{ ip }}{{ mask }}
{% endfor %}
```

Zip simply inserts the elements of the lists lans, ips and masks, one by one, into a new list, and delivers them to the for loop, which consumes them 3 by 3. As a matter of fact, we have transposed the existing data model.

Filters are better than tests

Suppose you get the following data as input:

```
routes_ipv4:
   - subnet:   10.0.0.0/8
     nexthop:  192.0.0.1
   - subnet:   172.16.0.0/12
     nexthop:  192.0.0.1
routes_ipv6:
   - subnet:    2001::/16
     nexthop:   2001:db8::1
```

This is probably because the user interface has to distinguish ipv4 and ipv6 routes, but your template will be shorter if you combine both ipv4 and ipv6 routes into the same list and iterate only once.

The initial data model

```
routes_ipv4:  [ subnet, nexthop ]
routes_ipv6:  [ subnet, nexthop ]
```

will be transformed into :

```
routes:  [ subnet, nexthop ]
```

The template has eventually to know the route's type: this a task for jinja's filters. Here we use the ipaddr filter available with Ansible to check the subnet's type.

The basic template

```
router bgp 65000
  address-family ipv4
{% for route in routes_ipv4 %}
  network {{ route.subnet }}
{% endfor %}
  exit
  address-family ipv6
{% for route in routes_ipv6 %}
  network {{ route.subnet }}
{% endfor %}
  Exit
```

Is now transformed into :

```
router bgp 65000
{% for route in    routes_ipv4 + routes_ipv6 %}
   address-family ipv{{ route.subnet | ipaddr('version') }}
   network {{ route.subnet }}
   exit
{% endfor %}
```

A simple list concatenation, followed by a filter, has reduced the template size by half, improving readability and maintainability.

Of course, both templates give the same result:

```
router bgp 65000
```

```
   address-family ipv4
   network 10.0.0.0/8
   exit
   address-family ipv4
   network 172.16.0.0/12
   exit
   address-family ipv6
   network 2001::/16
   exit
```

Add new data inside for loops

Now, you have to convert some input data into values:

```
routes:
   - subnet:   10.0.0.0/8
     nexthop:  192.0.0.1
     mode:     backup
   - subnet:   172.16.0.0/12
     nexthop:  192.0.0.1
     mode:     nominal
```

A quite acceptable template is based on Python's ternary operator:

```
{% for route in routes %}
ip route {{ route.subnet }} {{ route.nexthop }} {{ '150' if route.mode=='backup' else
'10' }}
{% endfor %}
```

Which returns:

```
ip route 10.0.0.0/8 192.0.0.1 150
ip route 172.16.0.0/12 192.0.0.1 10
```

However, adding new modes, let us say *equal cost*, *last resort*, … quickly ruins this logic.

Under Python, the proper way is to use a dictionary which maps each mode to a distance:

```
mapping = { 'nominal': 10, 'backup': 150, 'last resort': 250, … }
print ( mapping[route.mode] )
```

Of course, same principle can be applied to Jinja :

```
{% for route in routes %}
  ip route {{ route.subnet }} {{ route.nexthop }} {{
             { 'backup': 150, 'nominal': 10, 'last resort': 250 } [route.mode]
                                                   }}
{% endfor %}
```

However, I do not like to insert new data in the middle of a template, especially if you have to reuse the dictionary. Let us move the mapping into the loop.

```
{% for route, distance in routes |
                          zip (
                                routes |
                                map( attribute='mode' ) |
                                map ( 'extract', { 'backup': 150, 'nominal': 10 } )
```

```
                              ) %}

ip route {{ route.subnet }} {{ route.nexthop }} {{ distance }}
{% endfor %}
```

Of course, the syntax is not especially light (and based on Ansible add-ons), but still readable:

To each element of the list routes, you add (zip) a new element based on the dictionary `{ 'backup': 150, 'nominal': 10, 'last resort': 250 }` indexed by the value **route.mode**, which is the role of the extract filter (available with the Ansible jinja's extensions).

Another way, a priori cleaner, is to add a new key/value entry to the route dictionary. Using the above dictionary matching approach, I have not found better than this syntax:

```
{%
  for route in routes
           | map('combine',
                routes | map( attribute='mode' )
                       | map ( 'extract', { 'backup': 150, 'nominal': 10 } )
                       | json_query('[].{\"distance\": @}')
               )
%}
ip route {{ route.subnet }} {{ route.nexthop }} {{ route.distance }}
{% endfor %}
```

So forget this syntax, and try a new approach based on selectattr and combine filters:

```
{% for route in
   routes|selectattr('mode','equalto','backup')|map('combine', { 'distance': '150' })
+  routes|selectattr('mode','equalto','nominal')|map('combine', { 'distance': '10' })
+  routes|selectattr('mode','equalto','last resort')|map('combine', { 'distance': '250'
})
  %}
ip route {{ route.subnet }} {{ route.nexthop }} {{ route.distance }}
{% endfor %}
```

But again, do not try to be too clever !

Reverse a data structure with groupby

Groupby is perhaps the most interesting filter to deal with incorrect data model.

For instance, this data model will not help you to configure VRFs:

```
interfaces:
    - name:   eth 0/1
      subnet: 10.0.0.0/16
      vrf:    BLUE
    - name:   eth 0/2
      subnet: 20.0.0.0/16
      vrf:    BLUE
    - name:   eth 0/3
      subnet: 30.0.0.0/16
      vrf:    GREEN
```

Suppose you create a template like this one:

```
{% for itf in interfaces %}
vrf definition {{ itf.vrf }}
   rd 1:9{{ "%04d" | format(loop.index) }}
interface {{ itf.name }}
   vrf forwarding {{ itf.vrf }}
   ip address {{ itf.addr }}
{% endfor %}
```

You will get:

```
vrf definition BLUE
   rd 1:90001
interface eth 0/1
   vrf forwarding BLUE
   ip address 10.0.0.1/16

vrf definition BLUE
   rd 1:90002
interface eth 0/2
   vrf forwarding BLUE
   ip address 20.0.0.1/16
…
```

The interfaces are correctly configured, however you have assigned two different identifiers to the same VRF.

The issue is that the data model is correct from the interface point of view, but incorrect from the VRF point of view.

You have this data model :

```
interfaces:
     vrf
```

While you need

```
vrfs :  [ interfaces ]
```

Fortunately, the groupby filter will sort the elements of a list according to an attribute, and

```
{{ interfaces | groupby('vrf') | to_yaml }}
```

Returns :

```
- BLUE
  - {name: eth 0/1, subnet: 10.0.0.0/16, vrf: BLUE}
  - {name: eth 0/2, subnet: 20.0.0.0/16, vrf: BLUE}
- GREEN
  - {name: eth 0/3, subnet: 30.0.0.0/16, vrf: GREEN}
```

Which is more or less the data model we are expecting!

The for loop takes two arguments : it retrieves first the name of the VRF, then the interfaces dictionaries belonging to this VRF :

```
{% for vrf, itfs in interfaces | groupby('vrf') %}
{{ vrf }}  {{ itfs }}
{% endfor %}
```

For our use case, since we do need the second argument, we use the well known variable _, and the template becomes:

```
{% for vrf, _ in interfaces|groupby('vrf') %}
```

```
vrf definition {{ vrf }}
   rd 1:9{{ "%04d" | format(loop.index) }}
{% endfor %}
{% for itf in interfaces %}
interface {{ itf.name }}
   vrf forwarding {{ itf.vrf }}
   ip address {{ itf.addr }}
{% endfor %}
```

Quite simple and readable and the template returns correctly:

```
vrf definition BLUE
   rd 1:90001

interface eth 0/1
   vrf forwarding BLUE
   ip address 10.0.0.1/16

interface eth 0/2
   vrf forwarding BLUE
   ip address 20.0.0.1/16
```

Note that if we have to look at the data inside the groups, we have to use a second iteration. But if you refer to the expected data model, you will see that it is exactly what we are asking for:

```
vrfs :  [ interfaces ]
```

For instance, suppose that we want to configure routes advertisement vrf by vrf in a router bgp context, a fitting template is

```
router bgp 65500
{% for vrf, itfs in interfaces | groupby('vrf') %}
   address-family ipv4 vrf {{ vrf }}
   {% for itf in itfs %}
   network {{ itf.subnet }}
   {% endfor %}
   exit
{% endfor %}
```

And you get

```
router bgp 65500
   address-family ipv4 vrf BLUE
      network 10.0.0.0/16
      network 20.0.0.0/16
   exit
```

Conclusion

If your data model does not fit a specific task, you have the choice between using multiple inline if statements and code duplication, or using Jinja2's multiple tools to modify the data structures.

The second option usually makes templates easier to read and to maintain.