1 Python REST API with ACI (part 2)

We have seen in the previous post:

https://www.linkedin.com/posts/activity-6843926822833418240-o8-d

… how to login into Cisco APIC to get information about classes and data, and store them into an excel file. We will now see in this post how we will interact with ACI to configure something on the system. The important thing to keep in mind, is that most 'automation' activities related to new configurations are based on the same process:

- prepare a template file
- replace variables defined in this template
- perform REST API 'post' queries

Sometimes you have a GUI to click here and there, but as far as I've seen on Cisco NSO or 'Versa Sd-Wan' the general approach to the problem is exactly the same.

1.1 Old style CLI

Let's consider for example the following nx-os configuration:

```
interface port-channel10.25
  description TEST
  mtu 4000
  encapsulation dot1q 25
  vrf member TEST
  no ip redirects
  ip address 10.1.1.1/24
  no shutdown
```

We have explained in the previous post what is the 'json' way of representing data, so the following:

```
{
  "interface": {
    "attributes": {
       "name": "port-channel10.25",
       "description": "TEST"
       "mtu": "4000",
       "encapsulation": "dot1q",
       "vrf": "TEST",
       "redirects": "no",
       "ipv4_address": "10.1.1.1/24",
       "admin_state": "up"
    }
  }
}
```

… could be a json object representing the above interface.

1.2 Jinja templates

Jinja2 is a Python library to manage template files. To avoid rewriting by hand the above json object for every new interface or to change configurations regarding an already existing interface, you write down something like the following:

```
{
  "interface": {
    "attributes": {
       "name": "{{ if_name }}",
       "description": "{{ descr }}",
       "mtu": "{{ mtu }}",
       "encapsulation": "{{ encap_type }}",
       "vrf": "{{ vrf }}",
       "redirects": "{{ redir }}",
       "ipv4_address": "{{ ipv4_addr }}",
       "admin_state": "{{ adm_state }}",
    }
  }
}
```

… variables are written between double curly brackets like for example {{ mtu }}. Of course Jinja2 library is more powerful than this, some programming languages structures are allowed to be used:

```
{% for intf in intf_list %}
   <print something here>
{% endfor %}
```

A detailed explanation of what can be done with Jinja2 is outside the scope of this small document.
Let's now consider the following Python function, always taken from here:
https://github.com/carlmontanari/acipdt

The function is taken from '**FabTnPol**' class, which can be imported if necessary to create/configure 'layer2' ACI objects. As a first step are pre-defined optional and mandatory arguments. In case arguments are not passed to the function, the empty values will be used.

```
    # Method must be called with the following kwargs.
    # tn_name: The name of the Tenant
    # name: Name of the BD
    # arp: yes | no
    # mdest: bd-flood | drop | encap-flood
    # mcast: flood | opt-flood
    # unicast: yes | no
    # unk_unicast: proxy | flood
    # vrf: Name of associated VRF
    # status: created | created,modified | deleted
    # multicast (Optional): yes | no -- multicast routing tick box
    def bd(self, **kwargs):
        required_args = {'tn_name': '',
                         'name': '',
                         'arp': '',
                         'mdest': '',
                         'mcast': '',
                         'unicast': '',
                         'unk_unicast': '',
                         'status': ''}
```

```
        optional_args = {'limitlearn': 'yes',
                         'multicast': 'no',
                         'vrf': '',
                         'descr': ''}

        templateVars = process_kwargs(required_args, optional_args, **kwargs)
        if templateVars['status'] not in valid_status:
            raise InvalidArg('Status invalid')
```

This is the Jinja2 template file that will be used, the subsequent function calls replace the variables with those that have been passed before.

```
        template_file = "bd.json"
        template = self.templateEnv.get_template(template_file)
        payload = template.render(templateVars)
```

Follows the https '**post**' query with the json payload:

```
        uri = ('mo/uni/tn-{}/BD-{}'
               .format(templateVars['tn_name'], templateVars['name']))
        status = post(self.apic, payload, self.cookies, uri, template_file)
        return status
```

Let's have a look at the "bd.json" file:

```
{
    "fvBD": {
        "attributes": {
            "arpFlood": "{{arp}}",
            "dn": "uni/tn-{{tn_name}}/BD-{{name}}",
            "epMoveDetectMode": "",
            "limitIpLearnToSubnets": "{{limitlearn}}",
            "llAddr": "::",
            "mac": "00:22:BD:F8:19:FF",
            "multiDstPktAct": "{{mdest}}",
            "name": "{{name}}",
            "unicastRoute": "{{unicast}}",
            "unkMacUcastAct": "{{unk_unicast}}",
            "unkMcastAct": "{{mcast}}",
            "mcastAllow": "{{multicast}}",
            "status": "{{status}}",
            "descr": "{{descr}}"
        }
    }
}
```

… which looks like the example we have seen before. The above showed function can be called simply in the following way (tnConf is an instance of the FabTnPol class to which the bd function belongs to):

```
            status = tnConf.bd(tn_name = ten_name,
                               name = bd_name,
                               arp = 'yes',
                               mdest = 'bd-flood',
                               mcast = 'flood',
                               unicast = 'no',
                               unk_unicast = 'flood',
                               status = 'created',
                               vrf = vrf_name,
                               descr = description)
```

In case other parameters have to be configured (routing protocols, bgp neighbors, …) the template files will obviously change, the variables to be passed will be different, but the concepts remain the same.
The "status" object attribute can be one of the following values:

```
# status: created | created,modified | deleted
```

Created: the object MUST be new, in case it's not an error will be raised and the post operation will be rejected

Created,modified: the object can be new or already existent, in the latter case it will be modified accordingly to the post operation.

Deleted: the object should exist or an error will be received by the APIC.

1.3 Data retrieval

How is data passed and retrieved to perform 'post' queries toward the APIC ? This is up to you, or the software that interacts with ACI's APIC: data can be retrieved through a GUI (which is what happens when you do configurations through ACI Graphical User Interface). One other possible way to do it, would be that of using an excel file with a standard predefined format. A lot of checks could be performed regarding data consistency, and to avoid potential errors and mistakes. This is the value of a good automation. In case every week there are hundreds of EPGs to be created/destroyed, such an excel file to perform 'bulk' operations would be the real valuable automation, that would save a lot of time and potential configuration errors.
For example, the excel file could be something like the following:

tenant	vrf	vlan_number	l2_vlan_name	app_profile	ip_addr
TEST	TEST_VRF	Vlan2621	L2_NAME_BD	<APP_PROF_NAME>_ANP	10.1.1.14/28

route_type	descr	epg	interfaces
private	Epg description	<EPG_NAME>_EPG	vPC_TEST Leaf-812, Eth1/3

… and the following rules could be followed before proceeding with all the configurations:

```
1) check if the tenant already exists
2) check if the vrf already exists
3) check if the BD already exists
4) check if the app profile already exists
5) check if the EPG already exists
6) print out two tables with the above data
7) check if the specified port-channel or vPC interface exist (they should), in the
    interfaces column (to which EPGs should be added)
8) check if the physical interface exists (it should) and is configured, if it is not
   raise a fatal error.
9) check if the tenant, vrf or bd names are empty (they shouldn't)
10) if the bd already exists, checks if it has an ip subnet configured, in case
    in the excel file the subnet is defined, raise a fatal error
11) in case the route_type is not specified, it will be "private". In case a value
    is specified, it must a good one (private/public/shared)
12) in case the EPG name is not specified, the L2 vlan name + "_EPG" will be used
13) in case the APP name is not specified, the L2 vlan name + "_ANP" will be used
```

```
14) successfully configured objects are green colored, already existing objects are
    yellow colored, unsuccessful queries are red colored
```

More in general, automation should take care of all the update process, also regarding for example ip subnets and their insertion in an IPAM (IP Address Management) tool.

Riccardo Andreetta