Developing Cisco ACI modules

This is a brief walk-through of how to create new Cisco ACI modules for Ansible.

For more information about Cisco ACI, look at the :ref:`Cisco ACI user guide <aci_guide>`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]developing_modules_general_aci.rst, line 8); *backlink***
>
> Unknown interpreted text role "ref".

What's covered in this section:

Introduction

The cisco.aci collection already includes a large number of Cisco ACI modules, however the ACI object model is huge and covering all possible functionality would easily cover more than 1500 individual modules.

If you need specific functionality, you have 2 options:

- Learn the ACI object model and use the low-level APIC REST API using the :ref:`aci_rest <aci_rest_module>` module

  > **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]developing_modules_general_aci.rst, line 25); *backlink***
  >
  > Unknown interpreted text role "ref".

- Write your own dedicated modules, which is actually quite easy

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]developing_modules_general_aci.rst, line 28)**
>
> Unknown directive type "seealso".
>
> ```
> .. seealso::
>
>    `Ansible ACI collection <https://github.com/CiscoDevNet/ansible-aci>`_
>        Github repository of the ansible ACI collection
>    :ref:`hacking_collections`
>        Information on how to contribute to collections.
>    `ACI Fundamentals: ACI Policy Model <https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/aci-fundamentals,
>        A good introduction to the ACI object model.
>    `APIC Management Information Model reference <https://developer.cisco.com/docs/apic-mim-ref/>`_
>        Complete reference of the APIC object model.
>    `APIC REST API Configuration Guide <https://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/2-x/rest_cfg/2_1_x/b_(
>        Detailed guide on how the APIC REST API is designed and used, incl. many examples.
> ```

So let's look at how a typical ACI module is built up.

ACI module structure

Importing objects from Python libraries

The following imports are standard across ACI modules:

```
from ansible.module_utils.aci import ACIModule, aci_argument_spec
from ansible.module_utils.basic import AnsibleModule
```

Defining the argument spec

The first line adds the standard connection parameters to the module. After that, the next section will update the `argument_spec` dictionary with module-specific parameters. The module-specific parameters should include:

- the object_id (usually the name)
- the configurable properties of the object
- the parent object IDs (all parents up to the root)
- only child classes that are a 1-to-1 relationship (1-to-many/many-to-many require their own module to properly manage)
- the state
  - `state: absent` to ensure object does not exist
  - `state: present` to ensure the object and configs exist; this is also the default
  - `state: query` to retrieve information about objects in the class

```
def main():
    argument_spec = aci_argument_spec()
    argument_spec.update(
        object_id=dict(type='str', aliases=['name']),
        object_prop1=dict(type='str'),
        object_prop2=dict(type='str', choices=['choice1', 'choice2', 'choice3']),
        object_prop3=dict(type='int'),
        parent_id=dict(type='str'),
        child_object_id=dict(type='str'),
        child_object_prop=dict(type='str'),
        state=dict(type='str', default='present', choices=['absent', 'present', 'query']),
    )
```

> **Hint**
>
> Do not provide default values for configuration arguments. Default values could cause unintended changes to the object.

Using the AnsibleModule object

The following section creates an AnsibleModule instance. The module should support check-mode, so we pass the `argument_spec` and `supports_check_mode` arguments. Since these modules support querying the APIC for all objects of the module's class, the object/parent IDs should only be required if `state: absent` or `state: present`.

```
module = AnsibleModule(
    argument_spec=argument_spec,
    supports_check_mode=True,
    required_if=[
        ['state', 'absent', ['object_id', 'parent_id']],
        ['state', 'present', ['object_id', 'parent_id']],
    ],
)
```

Mapping variable definition

Once the AnsibleModule object has been initiated, the necessary parameter values should be extracted from `params` and any data validation should be done. Usually the only params that need to be extracted are those related to the ACI object configuration and its child configuration. If you have integer objects that you would like to validate, then the validation should be done here, and the `ACIModule.payload()` method will handle the string conversion.

```
object_id = object_id
object_prop1 = module.params['object_prop1']
object_prop2 = module.params['object_prop2']
object_prop3 = module.params['object_prop3']
if object_prop3 is not None and object_prop3 not in range(x, y):
    module.fail_json(msg='Valid object_prop3 values are between x and (y-1)')
child_object_id = module.params['child_object_id']
child_object_prop = module.params['child_object_prop']
state = module.params['state']
```

Using the ACIModule object

The ACIModule class handles most of the logic for the ACI modules. The ACIModule extends functionality to the AnsibleModule object, so the module instance must be passed into the class instantiation.

```
aci = ACIModule(module)
```

The ACIModule has six main methods that are used by the modules:

- construct_url
- get_existing
- payload
- get_diff
- post_config
- delete_config

The first two methods are used regardless of what value is passed to the `state` parameter.

Constructing URLs

The `construct_url()` method is used to dynamically build the appropriate URL to interact with the object, and the appropriate filter string that should be appended to the URL to filter the results.

- When the `state` is not `query`, the URL is the base URL to access the APIC plus the distinguished name to access the object. The filter string will restrict the returned data to just the configuration data.
- When `state` is `query`, the URL and filter string used depends on what parameters are passed to the object. This method handles the complexity so that it is easier to add new modules and so that all modules are consistent in what type of data is returned.

> **Note**
>
> Our design goal is to take all ID parameters that have values, and return the most specific data possible. If you do not supply any ID parameters to the task, then all objects of the class will be returned. If your task does consist of ID parameters sed, then the data for the specific object is returned. If a partial set of ID parameters are passed, then the module will use the IDs that are passed to build the URL and filter strings appropriately.

The `construct_url()` method takes 2 required arguments:

- **self** - passed automatically with the class instance
- **root_class** - A dictionary consisting of `aci_class`, `aci_rn`, `target_filter`, and `module_object` keys
  - **aci_class**: The name of the class used by the APIC, for example `fvTenant`
  - **aci_rn**: The relative name of the object, for example `tn-ACME`
  - **target_filter**: A dictionary with key-value pairs that make up the query string for selecting a subset of entries, for example `{'name': 'ACME'}`
  - **module_object**: The particular object for this class, for example `ACME`

Example:

```
aci.construct_url(
    root_class=dict(
        aci_class='fvTenant',
        aci_rn='tn-{0}'.format(tenant),
        target_filter={'name': tenant},
        module_object=tenant,
    ),
)
```

Some modules, like `aci_tenant`, are the root class and so they would not need to pass any additional arguments to the method.

The `construct_url()` method takes 4 optional arguments, the first three imitate the root class as described above, but are for child objects:

- subclass_1 - A dictionary consisting of `aci_class`, `aci_rn`, `target_filter`, and `module_object` keys
  - Example: Application Profile Class (AP)
- subclass_2 - A dictionary consisting of `aci_class`, `aci_rn`, `target_filter`, and `module_object` keys
  - Example: End Point Group (EPG)
- subclass_3 - A dictionary consisting of `aci_class`, `aci_rn`, `target_filter`, and `module_object` keys
  - Example: Binding a Contract to an EPG
- child_classes - The list of APIC names for the child classes supported by the modules.
  - This is a list, even if it is a list of one
  - These are the unfriendly names used by the APIC
  - These are used to limit the returned child_classes when possible
  - Example: `child_classes=['fvRsBDSubnetToProfile', 'fvRsNdPfxPol']`

> **Note**
>
> Sometimes the APIC will require special characters ([,], and -) or will use object metadata in the name ("vlanns" for VLAN pools); the module should handle adding special characters or joining of multiple parameters in order to keep expected inputs simple.

Getting the existing configuration

Once the URL and filter string have been built, the module is ready to retrieve the existing configuration for the object:

- `state: present` retrieves the configuration to use as a comparison against what was entered in the task. All values that are different than the existing values will be updated.

- `state: absent` uses the existing configuration to see if the item exists and needs to be deleted.
- `state: query` uses this to perform the query for the task and report back the existing data.

```
aci.get_existing()
```

When state is present

When `state: present`, the module needs to perform a diff against the existing configuration and the task entries. If any value needs to be updated, then the module will make a POST request with only the items that need to be updated. Some modules have children that are in a 1-to-1 relationship with another object; for these cases, the module can be used to manage the child objects.

Building the ACI payload

The `aci.payload()` method is used to build a dictionary of the proposed object configuration. All parameters that were not provided a value in the task will be removed from the dictionary (both for the object and its children). Any parameter that does have a value will be converted to a string and added to the final dictionary object that will be used for comparison against the existing configuration.

The `aci.payload()` method takes two required arguments and 1 optional argument, depending on if the module manages child objects.

- `aci_class` is the APIC name for the object's class, for example `aci_class='fvBD'`
- `class_config` is the appropriate dictionary to be used as the payload for the POST request
  - The keys should match the names used by the APIC.
  - The values should be the corresponding value in `module.params`; these are the variables defined above
- `child_configs` is optional, and is a list of child config dictionaries.
  - The child configs include the full child object dictionary, not just the attributes configuration portion.
  - The configuration portion is built the same way as the object.

```
aci.payload(
    aci_class=aci_class,
    class_config=dict(
        name=bd,
        descr=description,
        type=bd_type,
    ),
    child_configs=[
        dict(
            fvRsCtx=dict(
                attributes=dict(
                    tnFvCtxName=vrf
                ),
            ),
        ),
    ],
)
```

Performing the request

The `get_diff()` method is used to perform the diff, and takes only one required argument, `aci_class`. Example: `aci.get_diff(aci_class='fvBD')`

The `post_config()` method is used to make the POST request to the APIC if needed. This method doesn't take any arguments and handles check mode. Example: `aci.post_config()`

Example code

```
if state == 'present':
    aci.payload(
        aci_class='<object APIC class>',
        class_config=dict(
            name=object_id,
            prop1=object_prop1,
            prop2=object_prop2,
            prop3=object_prop3,
        ),
        child_configs=[
            dict(
                '<child APIC class>'=dict(
                    attributes=dict(
                        child_key=child_object_id,
                        child_prop=child_object_prop
                    ),
                ),
            ),
        ],
    )

    aci.get_diff(aci_class='<object APIC class>')

    aci.post_config()
```

When state is absent

If the task sets the state to absent, then the `delete_config()` method is all that is needed. This method does not take any arguments, and handles check mode.

```
elif state == 'absent':
    aci.delete_config()
```

Exiting the module

To have the module exit, call the ACIModule method `exit_json()`. This method automatically takes care of returning the common return values for you.

```
    aci.exit_json()

if   name _ == '__main__':
    main()
```

Testing ACI library functions

You can test your `construct_url()` and `payload()` arguments without accessing APIC hardware by using the following python script:

```
#!/usr/bin/python
import json
from ansible.module_utils.network.aci.aci import ACIModule

# Just another class mimicking a bare AnsibleModule class for construct_url() and payload() methods
class AltModule():
    params = dict(
        host='dummy',
        port=123,
        protocol='https',
        state='present',
        output_level='debug',
    )

# A sub-class of ACIModule to overload __init__ (we don't need to log into APIC)
class AltACIModule(ACIModule):
    def   init  (self):
        self.result = dict(changed=False)
```

```
        self.module = AltModule()
        self.params = self.module.params

# Instantiate our version of the ACI module
aci = AltACIModule()

# Define the variables you need below
aep = 'AEP'
aep_domain = 'uni/phys-DOMAIN'

# Below test the construct_url() arguments to see if it produced correct results
aci.construct_url(
    root_class=dict(
        aci_class='infraAttEntityP',
        aci_rn='infra/attentp-{}'.format(aep),
        target_filter={'name': aep},
        module_object=aep,
    ),
    subclass_1=dict(
        aci_class='infraRsDomP',
        aci_rn='rsdomP-[{}]'.format(aep_domain),
        target_filter={'tDn': aep_domain},
        module_object=aep_domain,
    ),
)

# Below test the payload arguments to see if it produced correct results
aci.payload(
    aci_class='infraRsDomP',
    class_config=dict(tDn=aep_domain),
)

# Print the URL and proposed payload
print 'URL:', json.dumps(aci.url, indent=4)
print 'PAYLOAD:', json.dumps(aci.proposed, indent=4)
```

This will result in:

```
URL: "https://dummy/api/mo/uni/infra/attentp-AEP/rsdomP-[phys-DOMAIN].json"
PAYLOAD: {
    "infraRsDomP": {
        "attributes": {
            "tDn": "phys-DOMAIN"
        }
    }
}
```

Testing for sanity checks

For legacy versions of ansible, you can run from your fork something like:

```
$ ansible-test sanity --python 2.7 lib/ansible/modules/network/aci/aci_tenant.py
```

Meanwhile, the ACI modules have moved into a collection. Please refer to the links below, which provide detailed guidance how to setup your environment and test the collection.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]developing_modules_general_aci.rst, line 407)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`hacking_collections`
        Information how to setup your environment to contribute to collections
   :ref:`testing_sanity`
        Information on how to build sanity tests.
   `Ansible ACI collection <https://github.com/CiscoDevNet/ansible-aci>`_
       Github repository of the ansible ACI collection
```

Testing ACI integration tests

You can run this:

```
$ ansible-test network-integration --continue-on-error --allow-unsupported --diff -v aci_tenant
```

Note

You may need to add `--python 2.7` or `--python 3.6` in order to use the correct python version for performing tests.

You may want to edit the used inventory at *test/integration/inventory.networking* and add something like:

```
[aci:vars]
aci_hostname=my-apic-1
aci_username=admin
aci_password=my-password
aci_use_ssl=yes
aci_use_proxy=no

[aci]
localhost ansible_ssh_host=127.0.0.1 ansible_connection=local
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\[ansible-devel][docs][docsite][rst][dev_guide]developing_modules_general_aci.rst, line 441)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`testing_integration`
       Information on how to build integration tests.
```

Testing for test coverage

You can run this:

```
$ ansible-test network-integration --python 2.7 --allow-unsupported --coverage aci_tenant
$ ansible-test coverage report
```