Developing network resource modules

Understanding network and security resource modules

Network and security devices separate configuration into sections (such as interfaces, VLANs, and so on) that apply to a network or security service. Ansible resource modules take advantage of this to allow users to configure subsections or resources within the device configuration. Resource modules provide a consistent experience across different network and security devices. For example, a network resource module may only update the configuration for a specific portion of the network interfaces, VLANs, ACLs, and so on for a network device. The resource module:

1. Fetches a piece of the configuration (fact gathering), for example, the interfaces configuration.
2. Converts the returned configuration into key-value pairs.
3. Places those key-value pairs into an internal independent structured data format.

Now that the configuration data is normalized, the user can update and modify the data and then use the resource module to send the configuration data back to the device. This results in a full round-trip configuration update without the need for manual parsing, data manipulation, and data model management.

The resource module has two top-level keys - `config` and `state`:

- `config` defines the resource configuration data model as key-value pairs. The type of the `config` option can be `dict` or `list of dict` based on the resource managed. That is, if the device has a single global configuration, it should be a `dict` (for example, a global LLDP configuration). If the device has multiple instances of configuration, it should be of type `list` with each element in the list of type `dict` (for example, interfaces configuration).
- `state` defines the action the resource module takes on the end device.

The `state` for a new resource module should support the following values (as applicable for the devices that support them):

merged
: Ansible merges the on-device configuration with the provided configuration in the task.

replaced
: Ansible replaces the on-device configuration subsection with the provided configuration subsection in the task.

overridden
: Ansible overrides the on-device configuration for the resource with the provided configuration in the task. Use caution with this state as you could remove your access to the device (for example, by overriding the management interface configuration).

deleted
: Ansible deletes the on-device configuration subsection and restores any default settings.

gathered
: Ansible displays the resource details gathered from the network device and accessed with the `gathered` key in the result.

rendered
: Ansible renders the provided configuration in the task in the device-native format (for example, Cisco IOS CLI). Ansible returns this rendered configuration in the `rendered` key in the result. Note this state does not communicate with the network device and can be used offline.

parsed
: Ansible parses the configuration from the `running_configuration` option into Ansible structured data in the `parsed` key in the result. Note this does not gather the configuration from the network device so this state can be used offline.

Modules in Ansible-maintained collections must support these state values. If you develop a module with only "present" and "absent" for state, you may submit it to a community collection.

> **Note**
>
> The states `rendered`, `gathered`, and `parsed` do not perform any change on the device.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst`, **line 60**)
>
> Unknown directive type "seealso".
>
> ```
> .. seealso::
>
>    `Deep Dive on VLANs Resource Modules for Network Automation <https://www.ansible.com/blog/deep-dive-on-vlans-resource-modul
>            Walkthrough of how state values are implemented for VLANs.
> ```

Developing network and security resource modules

The Ansible Engineering team ensures the module design and code pattern within Ansible-maintained collections is uniform across resources and across platforms to give a vendor-independent feel and deliver good quality code. We recommend you use the resource module builder to develop a resource module.

The highlevel process for developing a resource module is:

1. Create and share a resource model design in the resource module models repository as a PR for review.
2. Download the latest version of the resource module builder.
3. Run the `resource module builder` to create a collection scaffold from your approved resource model.
4. Write the code to implement your resource module.
5. Develop integration and unit tests to verify your resource module.

6. Create a PR to the appropriate collection that you want to add your new resource module to. See :ref:`contributing_maintained_collections` for details on determining the correct collection for your module.

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 79); *backlink*
>
> Unknown interpreted text role "ref".

Understanding the model and resource module builder

The resource module builder is an Ansible Playbook that helps developers scaffold and maintain an Ansible resource module. It uses a model as the single source of truth for the module. This model is a `yaml` file that is used for the module DOCUMENTATION section and the argument spec.

The resource module builder has the following capabilities:

- Uses a defined model to scaffold a resource module directory layout and initial class files.
- Scaffolds either an Ansible role or a collection.
- Subsequent uses of the resource module builder will only replace the module arspec and file containing the module docstring.
- Allows you to store complex examples along side the model in the same directory.
- Maintains the model as the source of truth for the module and use resource module builder to update the source files as needed.
- Generates working sample modules for both `<network_os>_<resource>` and `<network_os>_facts`.

Accessing the resource module builder

To access the resource module builder:

1. clone the github repository:

   ```
   git clone https://github.com/ansible-network/resource_module_builder.git
   ```

2. Install the requirements:

   ```
   pip install -r requirements.txt
   ```

Creating a model

You must create a model for your new resource. The model is the single source of truth for both the argspec and docstring, keeping them in sync. Once your model is approved, you can use the resource module builder to generate three items based on the model:

- The scaffold for a new module
- The argspec for the new module
- The docstring for the new module

For any subsequent changes to the functionality, update the model first and use the resource module builder to update the module argspec and docstring.

For example, the resource model builder includes the `myos_interfaces.yml` sample in the :file:`models` directory, as seen below:

> **System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 124); *backlink*
>
> Unknown interpreted text role "file".

```
---
GENERATOR_VERSION: '1.0'

NETWORK_OS: myos
RESOURCE: interfaces
COPYRIGHT: Copyright 2019 Red Hat
LICENSE: gpl-3.0.txt

DOCUMENTATION: |
  module: myos_interfaces
  version_added: 1.0.0
  short_description: 'Manages <xxxx> attributes of <network_os> <resource>'
  description: 'Manages <xxxx> attributes of <network_os> <resource>.'
  author: Ansible Network Engineer
 notes:
    - 'Tested against <network_os> <version>'
  options:
    config:
      description: The provided configuration
      type: list
      elements: dict
      suboptions:
        name:
          type: str
          description: The name of the <resource>
        some_string:
          type: str
          description:
          - The some_string_01
          choices:
          - choice_a
          - choice_b
          - choice_c
          default: choice_a
        some_bool:
          description:
          - The some_bool.
          type: bool
        some_int:
          description:
          - The some_int.
          type: int
          version added: '1.1.0'
        some dict:
          type: dict
          description:
          - The some_dict.
          suboptions:
            property_01:
              description:
              - The property_01
              type: str
    state:
      description:
      - The state of the configuration after module completion.
      type: str
      choices:
```

```
      - merged
      - replaced
      - overridden
      - deleted
      default: merged
EXAMPLES:
  - deleted_example_01.txt
  - merged_example_01.txt
  - overridden_example_01.txt
  - replaced_example_01.txt
```

Notice that you should include examples for each of the states that the resource supports. The resource module builder also includes these in the sample model.

Share this model as a PR for review at resource module models repository. You can also see more model examples at that location.

Creating a collection scaffold from a resource model

To use the resource module builder to create a collection scaffold from your approved resource model:

```
ansible-playbook -e rm_dest=<destination for modules and module utils> \
                 -e structure=collection \
                 -e collection_org=<collection_org> \
                 -e collection_name=<collection_name> \
                 -e model=<model> \
                 site.yml
```

Where the parameters are as follows:

- `rm_dest`: The directory where the resource module builder places the files and directories for the resource module and facts modules.
- `structure`: The directory layout type (role or collection)
  - `role`: Generate a role directory layout.
  - `collection`: Generate a collection directory layout.
- `collection_org`: The organization of the collection, required when *structure=collection*.
- `collection_name`: The name of the collection, required when *structure=collection*.
- `model`: The path to the model file.

To use the resource module builder to create a role scaffold:

```
ansible-playbook -e rm_dest=<destination for modules and module utils> \
                 -e structure=role \
                 -e model=<model> \
                 site.yml
```

Examples

Collection directory layout

This example shows the directory layout for the following:

- `network_os`: myos
- `resource`: interfaces

```
ansible-playbook -e rm dest=~/github/rm example \
                 -e structure=collection \
                 -e collection org=cidrblock \
                 -e collection name=my collection \
                 -e model=models/myos/interfaces/myos_interfaces.yml \
                 site.yml
```

```
â”œâ”€â”€ docs
â”œâ”€â”€ LICENSE.txt
â”œâ”€â”€ playbooks
â”œâ”€â”€ plugins
|   â”œâ”€â”€ action
|   â”œâ”€â”€ filter
|   â”œâ”€â”€ inventory
|   â”œâ”€â”€ modules
|   |   â”œâ”€â”€ __init__.py
|   |   â”œâ”€â”€ myos_facts.py
|   |   â””â”€â”€ myos_interfaces.py
|   â””â”€â”€ module_utils
|       â”œâ”€â”€ __init__.py
|       â””â”€â”€ network
|           â”œâ”€â”€ __init__.py
|           â””â”€â”€ myos
|               â”œâ”€â”€ argspec
|               |   â”œâ”€â”€ facts
|               |   |   â”œâ”€â”€ facts.py
|               |   |   â””â”€â”€ __init__.py
|               |   â”œâ”€â”€ __init__.py
|               |   â””â”€â”€ interfaces
|               |       â”œâ”€â”€ __init__.py
|               |       â””â”€â”€ interfaces.py
|               â”œâ”€â”€ config
|               |   â”œâ”€â”€ __init__.py
|               |   â””â”€â”€ interfaces
|               |       â”œâ”€â”€ __init__.py
|               |       â””â”€â”€ interfaces.py
|               â”œâ”€â”€ facts
|               |   â”œâ”€â”€ facts.py
|               |   â”œâ”€â”€ __init__.py
|               |   â””â”€â”€ interfaces
|               |       â”œâ”€â”€ __init__.py
|               |       â””â”€â”€ interfaces.py
|               â”œâ”€â”€ __init__.py
|               â””â”€â”€ utils
|                   â”œâ”€â”€ __init__.py
|                   â””â”€â”€ utils.py
â”œâ”€â”€ README.md
â””â”€â”€ roles
```

Role directory layout

This example displays the role directory layout for the following:

- `network_os`: myos
- `resource`: interfaces

```
ansible-playbook -e rm_dest=~/github/rm_example/roles/my_role \
                 -e structure=role \
                 -e model=models/myos/interfaces/myos_interfaces.yml \
                 site.yml
```

```
roles
â””â”€â”€ my role
    â”œâ”€â”€ library
```

```
â”‚   â”œâ”€â”€ __init__.py
â”‚   â”œâ”€â”€ myos_facts.py
â”‚   â””â”€â”€ myos_interfaces.py
â”œâ”€â”€ LICENSE.txt
â”œâ”€â”€ module_utils
â”‚   â”œâ”€â”€ __init__.py
â”‚   â””â”€â”€ network
â”‚       â”œâ”€â”€ __init__.py
â”‚       â””â”€â”€ myos
â”‚           â”œâ”€â”€ argspec
â”‚           â”‚   â”œâ”€â”€ facts
â”‚           â”‚   â”‚   â”œâ”€â”€ facts.py
â”‚           â”‚   â”‚   â””â”€â”€ __init__.py
â”‚           â”‚   â”œâ”€â”€ __init__.py
â”‚           â”‚   â””â”€â”€ interfaces
â”‚           â”‚       â”œâ”€â”€ __init__.py
â”‚           â”‚       â””â”€â”€ interfaces.py
â”‚           â”œâ”€â”€ config
â”‚           â”‚   â”œâ”€â”€ __init__.py
â”‚           â”‚   â””â”€â”€ interfaces
â”‚           â”‚       â”œâ”€â”€ __init__.py
â”‚           â”‚       â””â”€â”€ interfaces.py
â”‚           â”œâ”€â”€ facts
â”‚           â”‚   â”œâ”€â”€ facts.py
â”‚           â”‚   â”œâ”€â”€ __init__.py
â”‚           â”‚   â””â”€â”€ interfaces
â”‚           â”‚       â”œâ”€â”€ __init__.py
â”‚           â”‚       â””â”€â”€ interfaces.py
â”‚           â”œâ”€â”€ __init__.py
â”‚           â””â”€â”€ utils
â”‚               â”œâ”€â”€ __init__.py
â”‚               â””â”€â”€ utils.py
â””â”€â”€ README.md
```

Using the collection

This example shows how to use the generated collection in a playbook:

```
---
- hosts: myos101
  gather_facts: False
  tasks:
  - cidrblock.my_collection.myos_interfaces:
    register: result
  - debug:
      var: result
  - cidrblock.my_collection.myos_facts:
  - debug:
      var: ansible_network_resources
```

Using the role

This example shows how to use the generated role in a playbook:

```
- hosts: myos101
  gather_facts: False
  roles:
  - my_role

- hosts: myos101
  gather_facts: False
  tasks:
  - myos_interfaces:
    register: result
  - debug:
      var: result
  - myos facts:
  - debug:
      var: ansible_network_resources
```

Resource module structure and workflow

The resource module structure includes the following components:

Module

- `library/<ansible_network_os>_<resource>.py`.
- Imports the `module_utils` resource package and calls `execute_module` API:

```
def main():
    result = <resource_package>(module).execute_module()
```

Module argspec

- `module_utils/<ansible_network_os>/argspec/<resource>/`.
- Argspec for the resource.

Facts

- `module_utils/<ansible_network_os>/facts/<resource>/`.
- Populate facts for the resource.
- Entry in `module_utils/<ansible_network_os>/facts/facts.py` for `get_facts` API to keep `<ansible_network_os>_facts` module and facts gathered for the resource module in sync for every subset.
- Entry of Resource subset in FACTS_RESOURCE_SUBSETS list in `module_utils/<ansible_network_os>/facts/facts.py` to make facts collection work.

Module package in module_utils

- `module_utils/<ansible_network_os>/<config>/<resource>/`.
- Implement `execute_module` API that loads the configuration to device and generates the result with `changed`, `commands`, `before` and `after` keys.
- Call `get_facts` API that returns the `<resource>` configuration facts or return the difference if the device has onbox diff support.
- Compare facts gathered and given key-values if diff is not supported.
- Generate final configuration.

Utils

- `module_utils/<ansible_network_os>/utils`.
- Utilities for the `<ansible_network_os>` platform.

Running `ansible-test sanity` and `tox` on resource modules

You should run `ansible-test sanity` and `tox -elinters` from the collection root directory before pushing your PR to an Ansible-maintained collection. The CI runs both and will fail if these tests fail. See :ref:`developing_testing` for details on `ansible-test sanity`.

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

Unknown interpreted text role "ref".

To install the necessary packages:

1. Ensure you have a valid Ansible development environment configured. See :ref:`environment_setup` for details.

   System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 445); *backlink*

   Unknown interpreted text role "ref".

2. Run `pip install -r requirements.txt` from the collection root directory.

Running `tox -elinters`:

- Reads :file:`tox.ini` from the collection root directory and installs required dependencies (such as `black` and `flake8`).

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 451); *backlink*

  Unknown interpreted text role "file".

- Runs these with preconfigured options (such as line-length and ignores.)
- Runs `black` in check mode to show which files will be formatted without actually formatting them.

Testing resource modules

The tests rely on a role generated by the resource module builder. After changes to the resource module builder, the role should be regenerated and the tests modified and run as needed. To generate the role after changes:

```
rm -rf rmb_tests/roles/my_role
ansible-playbook -e rm_dest=./rmb_tests/roles/my_role \
                 -e structure=role \
                 -e model=models/myos/interfaces/myos_interfaces.yml \
                 site.yml
```

Resource module integration tests

High-level integration test requirements for new resource modules are as follows:

1. Write a test case for every state.
2. Write additional test cases to test the behavior of the module when an empty `config.yaml` is given.
3. Add a round trip test case. This involves a `merge` operation, followed by `gather_facts`, a `merge` update with additional configuration, and then reverting back to the base configuration using the previously gathered facts with the `state` set to `overridden`.
4. Wherever applicable, assertions should check after and before `dicts` against a hard coded Source of Truth.

We use Zuul as the CI to run the integration test.

- To view the report, click :guilabel:`Details` on the CI comment in the PR

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 485); *backlink*

  Unknown interpreted text role "guilabel".

- To view a failure report, click :guilabel:`ansible/check` and select the failed test.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 486); *backlink*

  Unknown interpreted text role "guilabel".

- To view logs while the test is running, check for your PR number in the Zuul status board.
- To fix static test failure locally, run the :command:`tox -e black` **inside the root folder of collection**.

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 488); *backlink*

  Unknown interpreted text role "command".

To view The Ansible run logs and debug test failures:

1. Click the failed job to get the summary, and click :guilabel:`Logs` for the log.

   System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 492); *backlink*

   Unknown interpreted text role "guilabel".

2. Click :guilabel:`console` and scroll down to find the failed test.

   System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 493); *backlink*

Unknown interpreted text role "guilabel".

3. Click :guilabel:`>` next to the failed test for complete details.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 494); *backlink*

Unknown interpreted text role "guilabel".

Integration test structure

Each test case should generally follow this pattern:

- setup â€"> test â€"> assert â€"> test again (for idempotency) â€"> assert â€"> tear down (if needed) -> done. This keeps test playbooks from becoming monolithic and difficult to troubleshoot.
- Include a name for each task that is not an assertion. You can add names to assertions as well, but it is easier to identify the broken task within a failed test if you add a name for each task.
- Files containing test cases must end in `.yaml`

Implementation

For platforms that support `connection: local` *and* `connection: network_cli` use the following guidance:

- Name the :file:`targets/` directories after the module name.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 511); *backlink*

Unknown interpreted text role "file".

- The :file:`main.yaml` file should just reference the transport.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 512); *backlink*

Unknown interpreted text role "file".

The following example walks through the integration tests for the `vyos.vyos.vyos_l3_interfaces` module in the vyos.vyos collection:

`test/integration/targets/vyos_l3_interfaces/tasks/main.yaml`

```
---
- import_tasks: cli.yaml
  tags:
    - cli
```

`test/integration/targets/vyos_l3_interfaces/tasks/cli.yaml`

```
---
- name: collect all cli test cases
  find:
    paths: "{{ role_path }}/tests/cli"
    patterns: "{{ testcase }}.yaml"
  register: test_cases
  delegate_to: localhost

- name: set test_items
  set_fact: test_items="{{ test_cases.files | map(attribute='path') | list }}"

- name: run test cases (connection=network_cli)
  include_tasks:
     file: "{{ test_case_to_run }}"
  vars:
     ansible_connection: network_cli
  with_items: "{{ test_items }}"
  loop_control:
    loop_var: test_case_to_run

- name: run test case (connection=local)
  include_tasks:
     file: "{{ test_case_to_run }}"
  vars:
     ansible connection: local
     ansible become: no
  with first found: "{{ test_items }}"
  loop control:
    loop_var: test_case_to_run
```

`test/integration/targets/vyos_l3_interfaces/tests/cli/overridden.yaml`

```
---
- debug:
 msg: START vyos_l3_interfaces merged integration tests on connection={{ ansible_connection
   }}

- import_tasks: _remove_config.yaml

- block:

 - import_tasks: _populate.yaml

 - name: Overrides all device configuration with provided configuration
   register: result
   vyos.vyos.vyos_l3_interfaces: &id001
     config:

       - name: eth0
         ipv4:

           - address: dhcp

       - name: eth1
         ipv4:

           - address: 192.0.2.15/24
     state: overridden
```

```
 - name: Assert that before dicts were correctly generated
   assert:
     that:
       - "{{ populate | symmetric_difference(result['before']) |length == 0 }}"

 - name: Assert that correct commands were generated
   assert:
     that:
       - "{{ overridden['commands'] | symmetric_difference(result['commands'])\
         \ |length == 0 }}"

 - name: Assert that after dicts were correctly generated
   assert:
     that:
       - "{{ overridden['after'] | symmetric_difference(result['after']) |length\
         \ == 0 }}"

 - name: Overrides all device configuration with provided configurations (IDEMPOTENT)
   register: result
   vyos.vyos.vyos_l3_interfaces: *id001

 - name: Assert that the previous task was idempotent
   assert:
     that:
       - result['changed'] == false

 - name: Assert that before dicts were correctly generated
   assert:
     that:
       - "{{ overridden['after'] | symmetric_difference(result['before']) |length\
         \ == 0 }}"
always:

 - import_tasks: _remove_config.yaml
```

Detecting test resources at runtime

Your tests should detect resources (such as interfaces) at runtime rather than hard-coding them into the test. This allows the test to run on a variety of systems.

For example:

```
- name: Collect interface list
  connection: ansible.netcommon.network_cli
  register: intout
  cisco.nxos.nxos_command:
    commands:
      - show interface brief | json

- set_fact:
    intdataraw: "{{ intout.stdout_lines[0]['TABLE_interface']['ROW_interface'] }}"

- set_fact:
    nxos_int1: '{{ intdataraw[1].interface }}'

- set_fact:
    nxos_int2: '{{ intdataraw[2].interface }}'

- set_fact:
    nxos_int3: '{{ intdataraw[3].interface }}'
```

See the complete test example of this at https://github.com/ansible-collections/cisco.nxos/blob/master/tests/integration/targets/prepare_nxos_tests/tasks/main.yml.

Running network integration tests

Ansible uses Zuul to run an integration test suite on every PR, including new tests introduced by that PR. To find and fix problems in network modules, run the network integration test locally before you submit a PR.

First, create an inventory file that points to your test machines. The inventory group should match the platform name (for example, `eos`, `ios`):

```
cd test/integration
cp inventory.network.template inventory.networking
${EDITOR:-vi} inventory.networking
# Add in machines for the platform(s) you wish to test
```

To run these network integration tests, use `ansible-test network-integration --inventory </path/to/inventory> <tests_to_run>`:

```
ansible-test network-integration  --inventory ~/myinventory -vvv vyos facts
ansible-test network-integration  --inventory ~/myinventory -vvv vyos_.*
```

To run all network tests for a particular platform:

```
ansible-test network-integration --inventory  /path/to-collection-module/test/integration/inventory.networking vyos_.*
```

This example will run against all `vyos` modules. Note that `vyos_.*` is a regex match, not a bash wildcard - include the . if you modify this example.

To run integration tests for a specific module:

```
ansible-test network-integration --inventory  /path/to-collection-module/test/integration/inventory.networking vyos_l3_interfaces
```

To run a single test case on a specific module:

```
# Only run vyos l3 interfaces/tests/cli/gathered.yaml
ansible-test network-integration --inventory  /path/to-collection-module/test/integration/inventory.networking vyos_l3_interfaces -
```

To run integration tests for a specific transport:

```
 # Only run nxapi test
ansible-test network-integration --inventory  /path/to-collection-module/test/integration/inventory.networking  --tags="nxapi" nxos

# Skip any cli tests
 ansible-test network-integration --inventory  /path/to-collection-module/test/integration/inventory.networking  --skip-tags="cli"
```

See test/integration/targets/nxos_bgp/tasks/main.yaml for how this is implemented in the tests.

For more options:

```
ansible-test network-integration --help
```

If you need additional help or feedback, reach out in the `#ansible-network` chat channel (using Matrix at ansible.im or using IRC at irc.libera.chat).

Unit test requirements

High-level unit test requirements that new resource modules should follow:

1. Write test cases for all the states with all possible combinations of config values.
2. Write test cases to test the error conditions (negative scenarios).
3. Check the value of `changed` and `commands` keys in every test case.

We run all unit test cases on our Zuul test suite, on the latest python version supported by our CI setup.

Use the :ref:`same procedure <using_zuul_resource_modules>` as the integration tests to view Zuul unit tests reports and logs.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 734); *backlink*

Unknown interpreted text role "ref".

See :ref:`unit module testing <testing_units_modules>` for general unit test details.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 736); *backlink*

Unknown interpreted text role "ref".

Example: Unit testing Ansible network resource modules

This section walks through an example of how to develop unit tests for Ansible resource modules.

See :ref:`testing_units` and :ref:`testing_units_modules` for general documentation on Ansible unit tests for modules. Please read those pages first to understand unit tests and why and when you should use them.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 748); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 748); *backlink*

Unknown interpreted text role "ref".

Using mock objects to unit test Ansible network resource modules

Mock objects can be very useful in building unit tests for special or difficult cases, but they can also lead to complex and confusing coding situations. One good use for mocks would be to simulate an API. The `mock` Python package is bundled with Ansible (use `import units.compat.mock`).

You can mock the device connection and output from the device as follows:

```
self.mock_get_config = patch( "ansible_collections.ansible.netcommon.plugins.module_utils.network.common.network.Config.get_config"
)
self.get_config = self.mock_get_config.start()

self.mock_load_config = patch(
"ansible_collections.ansible.netcommon.plugins.module_utils.network.common.network.Config.load_config"
)
self.load_config = self.mock_load_config.start()

self.mock_get_resource_connection_config = patch(
"ansible_collections.ansible.netcommon.plugins.module_utils.network.common.cfg.base.get_resource_connection"
)
self.get_resource_connection_config = (self.mock_get_resource_connection_config.start())

self.mock_get_resource_connection_facts = patch(
"ansible_collections.ansible.netcommon.plugins.module_utils.network.common.facts.facts.get_resource_connection"
)
self.get_resource_connection_facts = (self.mock_get_resource_connection_facts.start())

self.mock edit config = patch(
"ansible_collections.arista.eos.plugins.module_utils.network.eos.providers.providers.CliProvider.edit_config"
)
self.edit_config = self.mock_edit_config.start()

self.mock execute show command = patch(
"ansible_collections.arista.eos.plugins.module_utils.network.eos.facts.l2_interfaces.l2_interfaces.L2_interfacesFacts.get_device_da
)
self.execute_show_command = self.mock_execute_show_command.start()
```

The facts file of the module now includes a new method, `get_device_data`. Call `get_device_data` here to emulate the device output.

Mocking device data

To mock fetching results from devices or provide other complex data structures that come from external libraries, you can use `fixtures` to read in pre-generated data. The text files for this pre-generated data live in `test/units/modules/network/PLATFORM/fixtures/`. See for example the eos_l2_interfaces.cfg file.

Load data using the `load_fixture` method and set this data as the return value of the `get_device_data` method in the facts file:

```
def load_fixtures(self, commands=None, transport='cli'):
    def load_from_file(*args, **kwargs):
        return load_fixture('eos_l2_interfaces_config.cfg')
    self.execute_show_command.side_effect = load_from_file
```

See the unit test file test_eos_l2_interfaces for a practical example.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\network\dev_guide\[ansible-devel][docs][docsite][rst][network][dev_guide]developing_resource_modules_network.rst, line 819)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`testing_units`
       Deep dive into developing unit tests for Ansible modules
   :ref:`testing_running_locally`
       Running tests locally including gathering and reporting coverage data
   :ref:`developing_modules_general`
       Get started developing a module
```