oVirt Ansible Modules

The set of modules for interacting with oVirt/RHV are currently part of the community.general collection (on Galaxy, source code repository). This document serves as developer coding guidelines for creating oVirt/RHV modules.

Naming

- All modules should start with an `ovirt_` prefix.
- All modules should be named after the resource it manages in singular form.
- All modules that gather information should have a `_info` suffix.

Interface

- Every module should return the ID of the resource it manages.
- Every module should return the dictionary of the resource it manages.
- Never change the name of the parameter, as we guarantee backward compatibility. Use aliases instead.
- If a parameter can't achieve idempotency for any reason, please document it.

Interoperability

- All modules should work against all minor versions of version 4 of the API. Version 3 of the API is not supported.

Libraries

- All modules should use `ovirt_full_argument_spec` or `ovirt_info_full_argument_spec` to pick up the standard input (such as auth and `fetch_nested`).
- All modules should use `extends_documentation_fragment`: ovirt to go along with `ovirt_full_argument_spec`.
- All info modules should use `extends_documentation_fragment`: `ovirt_info` to go along with `ovirt_info_full_argument_spec`.
- Functions that are common to all modules should be implemented in the `module_utils/ovirt.py` file, so they can be reused.
- Python SDK version 4 must be used.

New module development

Please read :ref:`developing_modules`, first to know what common properties, functions and features every module must have.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\dev_guide\platforms\(ansible-devel)(docs)(docsite)(rst)(dev_guide)(platforms)ovirt_dev_guide.rst, line 53); *backlink***
>
> Unknown interpreted text role "ref".

In order to achieve idempotency of oVirt entity attributes, a helper class was created. The first thing you need to do is to extend this class and override a few methods:

```
try:
    import ovirtsdk4.types as otypes
except ImportError:
    pass

from ansible.module_utils.ovirt import (
    BaseModule,
    equal
)

class ClustersModule(BaseModule):

    # The build method builds the entity we want to create.
```

```
    # Always be sure to build only the parameters the user specified
    # in their yaml file, so we don't change the values which we shouldn't
    # change. If you set the parameter to None, nothing will be changed.
    def build_entity(self):
        return otypes.Cluster(
            name=self.param('name'),
            comment=self.param('comment'),
            description=self.param('description'),
        )

    # The update_check method checks if the update is needed to be done on
    # the entity. The equal method doesn't check the values which are None,
    # which means it doesn't check the values which user didn't set in yaml.
    # All other values are checked and if there is found some mismatch,
    # the update method is run on the entity, the entity is build by
    # 'build_entity' method. You don't have to care about calling the update,
    # it's called behind the scene by the 'BaseModule' class.
    def update_check(self, entity):
        return (
            equal(self.param('comment'), entity.comment)
            and equal(self.param('description'), entity.description)
        )
```

The code above handle the check if the entity should be updated, so we don't update the entity if not needed and also it construct the needed entity of the SDK.

```
from ansible.module_utils.basic import AnsibleModule
from ansible.module_utils.ovirt import (
    check_sdk,
    create_connection,
    ovirt_full_argument_spec,
)

# This module will support two states of the cluster,
# either it will be present or absent. The user can
# specify three parameters: name, comment and description,
# The 'ovirt_full_argument_spec' function, will merge the
# parameters created here with some common one like 'auth':
argument_spec = ovirt_full_argument_spec(
    state=dict(
        choices=['present', 'absent'],
        default='present',
    ),
    name=dict(default=None, required=True),
    description=dict(default=None),
    comment=dict(default=None),
)

# Create the Ansible module, please always implement the
# feature called 'check_mode', for 'create', 'update' and
# 'delete' operations it's implemented by default in BaseModule:
module = AnsibleModule(
    argument_spec=argument_spec,
    supports_check_mode=True,
)

# Check if the user has Python SDK installed:
check_sdk(module)

try:
    auth = module.params.pop('auth')

    # Create the connection to the oVirt engine:
    connection = create_connection(auth)

    # Create the service which manages the entity:
    clusters_service = connection.system_service().clusters_service()

    # Create the module which will handle create, update and delete flow:
    clusters_module = ClustersModule(
        connection=connection,
        module=module,
        service=clusters_service,
    )

    # Check the state and call the appropriate method:
    state = module.params['state']
    if state == 'present':
        ret = clusters_module.create()
    elif state == 'absent':
        ret = clusters_module.remove()
```

```
    # The return value of the 'create' and 'remove' method is dictionary
    # with the 'id' of the entity we manage and the type of the entity
    # with filled in attributes of the entity. The 'change' status is
    # also returned by those methods:
    module.exit_json(**ret)
except Exception as e:
    # Modules can't raises exception, it always must exit with
    # 'module.fail_json' in case of exception. Always use
    # 'exception=traceback.format_exc' for debugging purposes:
    module.fail_json(msg=str(e), exception=traceback.format_exc())
finally:
    # Logout only in case the user passed the 'token' in 'auth'
    # parameter:
    connection.close(logout=auth.get('token') is None)
```

If your module must support action handling (for example, virtual machine start) you must ensure that you handle the states of the virtual machine correctly, and document the behavior of the module:

```
if state == 'running':
    ret = vms_module.action(
        action='start',
        post_action=vms_module._post_start_action,
        action_condition=lambda vm: (
            vm.status not in [
                otypes.VmStatus.MIGRATING,
                otypes.VmStatus.POWERING_UP,
                otypes.VmStatus.REBOOT_IN_PROGRESS,
                otypes.VmStatus.WAIT_FOR_LAUNCH,
                otypes.VmStatus.UP,
                otypes.VmStatus.RESTORING_STATE,
            ]
        ),
        wait_condition=lambda vm: vm.status == otypes.VmStatus.UP,
        # Start action kwargs:
        use_cloud_init=use_cloud_init,
        use_sysprep=use_sysprep,
        # ...
    )
```

As you can see from the preceding example, the `action` method accepts the `action_condition` and `wait_condition`, which are methods which accept the virtual machine object as a parameter, so you can check whether the virtual machine is in a proper state before the action. The rest of the parameters are for the `start` action. You may also handle pre- or post- action tasks by defining `pre_action` and `post_action` parameters.

Testing

- Integration testing is currently done in oVirt's CI system on Jenkins and on GitHub.
- Please consider using these integration tests if you create a new module or add a new feature to an existing module.