
Molecule Documentation

Release 3.6.2.dev10

AUTHORS.rst

March 01, 2022

CONTENTS

1	About Ansible Molecule	3
2	Installation and Upgrade	5
3	Using Molecule	9
4	Common Molecule Use Cases	25
5	Contributing to Molecule	35
6	References and Appendices	39
7	External Resources	41
	Index	43

ABOUT ANSIBLE MOLECULE

Molecule project is designed to aid in the development and testing of [Ansible](#) roles.

Molecule provides support for testing with multiple instances, operating systems and distributions, virtualization providers, test frameworks and testing scenarios.

Molecule encourages an approach that results in consistently developed roles that are well-written, easily understood and maintained.

Molecule supports only the latest two major versions of Ansible (N/N-1), meaning that if the latest version is 2.9.x, we will also test our code with 2.8.x.

Once installed, the command line can be called using any of the methods below:

```
molecule ...  
python3 -m molecule ... # python module calling method
```


INSTALLATION AND UPGRADE

2.1 Installation

This document assumes the developer has a basic understanding of python packaging, and how to install and manage python on the system executing Molecule.

2.1.1 Requirements

Depending on the driver chosen, you may need to install additional OS packages. See `INSTALL.rst`, which is created when initializing a new scenario.

- Python \geq 3.8 with Ansible \geq 2.8

CentOS 8

```
$ sudo dnf install -y gcc python3-pip python3-devel openssl-devel python3-libs
```

Ubuntu 16.x

```
$ sudo apt update
$ sudo apt install -y python3-pip libssl-dev
```

2.1.2 Pip

`pip` is the only supported installation method.

Warning: Ansible is not listed as a direct dependency of molecule package because we only call it as a command line tool. You may want to install it using your distribution package installer.

```
$ python3 -m pip install molecule ansible-core
```

Warning: `ansible` and `ansible-base` pip **extras** were removed in molecule 4.0.0. If you used them, please switch to explicit package mention to avoid problem with newer versions of molecule.

Keep in mind that on selinux supporting systems, if you install into a virtual environment, you may face [issue](#) even if selinux is not enabled or is configured to be permissive.

It is your responsibility to assure that soft dependencies of Ansible are available on your controller or host machines.

Warning: It is highly recommended that you install molecule in a [virtual environment](#). This will provide a modern copy of [setuptools](#) which is mandatory in order for molecule to be installed successfully and function correctly. If you cannot use a virtual environment then you can attempt a package upgrade with the following:

```
$ python3 -m pip install --upgrade --user setuptools
```

Warning: Pip v19 series has an [isolation bug](#) of setuptools being exposed to the package build env. That is why it's highly recommended to upgrade user setuptools even when using a proper virtualenv as shown above.

Requirements

Depending on the driver chosen, you may need to install additional python packages. See the driver's documentation or `INSTALL.rst`, which is created when initializing a new scenario.

Install

Install Molecule:

```
$ python3 -m pip install --user "molecule"
```

Molecule does not include `ansible-lint` (nor does the `lint extra`), but is easily installed separately:

```
$ python3 -m pip install --user "molecule ansible-lint"
```

Molecule uses the “delegated” driver by default. Other drivers can be installed separately from PyPI, such as the `molecule-docker` driver. If you would like to use `docker` as the molecule driver, the installation command would look like this:

```
$ python3 -m pip install --user "molecule[docker]"
```

Other drivers, such as `molecule-podman`, `molecule-vagrant`, `molecule-azure` or `molecule-hetzner` are also available.

Installing molecule package also installed its main script `molecule`, usually in `PATH`. Users should know that molecule can also be called as a python module, using `python -m molecule ...`. This alternative method has some benefits:

- allows to explicitly control which python interpreter is used by molecule
- allows molecule installation at user level without even needing to have the script in `PATH`.

Note: We also have a continuous pre-release process which is provided for early adoption and feedback purposes only. It is available from test.pypi.org/project/molecule and can be installed like so:

```
python3 -m pip install \  
--index-url https://test.pypi.org/simple \  

```

(continues on next page)

(continued from previous page)

```
--extra-index-url https://pypi.org/simple \  
molecule==2.21.dev46
```

Where 2.21.dev46 is the latest available pre-release version. Please check the [release history](#) listing for the available releases.

2.1.3 Docker

Molecule is built into a Docker image by the [Toolset](#) project.

Any questions or bugs related to use of Molecule from within a container should be addressed by the Toolset project.

2.1.4 Source

Due to the rapid pace of development on this tool, you might want to install and update a bleeding-edge version of Molecule from Git.

Follow the instructions below to do the initial install and subsequent updates.

The package distribution that you'll get installed will be autogenerated and will contain a commit hash information making it easier to refer to certain unstable version should the need to send a bug report arise.

Warning: Please avoid using `--editable/-e` [development mode](#) when installing Molecule with Pip. This not very well supported and only needed when doing development. For contributing purposes, you can rely on the `tox` command line interface. Please see [our testing guide](#) for further details.

Requirements

CentOS 8

```
$ sudo dnf install -y libffi-devel git
```

Ubuntu 16.x

```
$ sudo apt install -y libffi-dev git
```

Install

```
$ python3 -m pip install -U git+https://github.com/ansible-community/molecule
```


USING MOLECULE

3.1 Getting Started Guide

The following guide will step through an example of developing and testing a new Ansible role. After reading this guide, you should be familiar with the basics of how to use Molecule and what it can offer.

Contents

- *Getting Started Guide*
 - *Creating a new role*
 - *Molecule Scenarios*
 - *The Scenario Layout*
 - *Inspecting the `molecule.yml`*
 - *Run test sequence commands*
 - *Run a full test sequence*

Note: In order to complete this guide by hand, you will need to additionally install [Docker](#). Molecule requires an external Python dependency for the Docker driver which is provided when installing Molecule using `pip install 'molecule[docker]'`.

3.1.1 Creating a new role

Molecule uses [galaxy](#) under the hood to generate conventional role layouts. If you've ever worked with Ansible roles before, you'll be right at home. If not, please review the [Roles](#) guide to see what each folder is responsible for.

To generate a new role with Molecule, simply run:

```
$ molecule init role acme.my_new_role --driver-name docker
```

You should then see a `my_new_role` folder in your current directory.

Note: For future reference, if you want to initialize Molecule within an existing role, you would use the `molecule init scenario -r my_role_name` command from within the role's directory (e.g. `my_role_name/`).

3.1.2 Molecule Scenarios

You will notice one new folder which is the `molecule` folder.

In this folder is a single *Scenario* called `default`.

Scenarios are the starting point for a lot of powerful functionality that Molecule offers. For now, we can think of a scenario as a test suite for your newly created role. You can have as many scenarios as you like and Molecule will run one after the other.

3.1.3 The Scenario Layout

Within the `molecule/default` folder, we find a number of files and directories:

```
$ ls
INSTALL.rst  molecule.yml  converge.yml  verify.yml
```

- `INSTALL.rst` contains instructions on what additional software or setup steps you will need to take in order to allow Molecule to successfully interface with the driver.
- `molecule.yml` is the central configuration entrypoint for Molecule. With this file, you can configure each tool that Molecule will employ when testing your role.
- `converge.yml` is the playbook file that contains the call for your role. Molecule will invoke this playbook with `ansible-playbook` and run it against an instance created by the driver.
- `verify.yml` is the Ansible file used for testing as Ansible is the default *Verifier*. This allows you to write specific tests against the state of the container after your role has finished executing. Other verifier tools are available (Note that `TestInfra` was the default verifier prior to molecule version 3).

Note: If the `verify.yml` playbook does not explicitly `include_role` your role, the `library` and `module_utils` provided by your role are not available in the playbook by default. If you need those for testing but would like to avoid re-running your role, consider adding an empty task file `init.yml` to your role and use `tasks_from` to include your role in the `verify.yml` playbook:

```
- name: Verify
  hosts: all
  become: true
  tasks:
  - name: Initialize role without actually running it
    include_role:
      name: my_role
      tasks_from: init

# Start testing: can use role library now
```

3.1.4 Inspecting the `molecule.yml`

The `molecule.yml` is for configuring Molecule. It is a `YAML` file whose keys represent the high level components that Molecule provides. These are:

- The *Dependency* manager. Molecule uses `galaxy` by default to resolve your role dependencies.
- The *Driver* provider. Molecule uses `Docker` by default. Molecule uses the driver to delegate the task of creating instances.
- The *Lint* command. Molecule can call external commands to ensure that best practices are encouraged. Note: `ansible-lint` is not included with molecule or molecule[lint].
- The *Platforms* definitions. Molecule relies on this to know which instances to create, name and to which group each instance belongs. If you need to test your role against multiple popular distributions (CentOS, Fedora, Debian), you can specify that in this section.
- The *Provisioner*. Molecule only provides an Ansible provisioner. Ansible manages the life cycle of the instance based on this configuration.
- The *Scenario* definition. Molecule relies on this configuration to control the scenario sequence order.
- The *Verifier* framework. Molecule uses Ansible by default to provide a way to write specific state checking tests (such as deployment smoke tests) on the target instance.

3.1.5 Run test sequence commands

Let's create the first Molecule managed instance with the Docker driver.

First, ensure that `Docker` is running with the typical sanity check:

```
$ docker run hello-world
```

Now, we can tell Molecule to create an instance with:

```
$ molecule create
```

We can verify that Molecule has created the instance and they're up and running with:

```
$ molecule list
```

Now, let's add a task to our `tasks/main.yml` like so:

```
- name: Molecule Hello World!
  debug:
    msg: Hello, World!
```

We can then tell Molecule to test our role against our instance with:

```
$ molecule converge
```

If we want to manually inspect the instance afterwards, we can run:

```
$ molecule login
```

We now have a free hand to experiment with the instance state.

Finally, we can exit the instance and destroy it with:

```
$ molecule destroy
```

Note: If Molecule reports any errors, it can be useful to pass the `--debug` option to get more verbose output.

3.1.6 Run a full test sequence

Molecule provides commands for manually managing the lifecycle of the instance, scenario, development and testing tools. However, we can also tell Molecule to manage this automatically within a *Scenario* sequence.

The full lifecycle sequence can be invoked with:

```
$ molecule test
```

Note: It can be particularly useful to pass the `--destroy=never` flag when invoking `molecule test` so that you can tell Molecule to run the full sequence but not destroy the instance if one step fails.

3.2 Continuous integration

Molecule output will use ANSI colors if stdout is an interactive TTY and TERM value seems to support it. You can define `PY_COLORS=1` to force use of ANSI colors, which can be handy for some CI systems.

3.2.1 GitHub Actions

GitHub Actions runs a CI pipeline, much like any others, that's built into GitHub.

An action to clone a repo as `molecule_demo`, and run `molecule test` in `ubuntu`.

```
---
name: Molecule Test
on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      max-parallel: 4
      matrix:
        python-version: [3.8, 3.9]

    steps:
      - uses: actions/checkout@v2
        with:
          path: molecule_demo
      - name: Set up Python ${ matrix.python-version }
        uses: actions/setup-python@v2
        with:
          python-version: ${ matrix.python-version }
      - name: Install dependencies
```

(continues on next page)

(continued from previous page)

```

run: |
    python3 -m pip install --upgrade pip
    python3 -m pip install -r requirements.txt
- name: Test with molecule
run: |
    molecule test

```

If you need access to requirements in private repositories, create a token with the required privileges, then define a `GIT_CREDENTIALS` secret for your repository with a value looking like `https://username:token@github.com/`, and finally add the following step before *Test with molecule*

```

- name: Setup git credentials
  uses: fusion-engineering/setup-git-credentials@v2
  with:
    credentials: ${secrets.GIT_CREDENTIALS}

```

3.2.2 Travis CI

Travis is a CI platform, which can be used to test Ansible roles.

A `.travis.yml` testing a role named `fool` with the Docker driver.

```

---
sudo: required
language: python
services:
  - docker
install:
  - python3 -m pip install molecule
  # - python3 -m pip install required driver (e.g. docker, shade, boto, apache-libcloud)
script:
  - molecule test

```

A `.travis.yml` using `Tox` as described below.

```

---
sudo: required
language: python
services:
  - docker
install:
  - python3 -m pip install tox-travis
script:
  - tox

```

3.2.3 Gitlab CI

Gitlab includes its own CI. Pipelines are usually defined in `.gitlab-ci.yml` file in the top folder of a repository, to be run on Gitlab Runners.

Here is an example using Docker in Docker

```
---
image: docker:stable-dind

services:
  - docker:dind

before_script:
  - apk add --no-cache
    python3 python3-dev py3-pip gcc git curl build-base
    autoconf automake py3-cryptography linux-headers
    musl-dev libffi-dev openssl-dev openssh
  - docker info
  - python3 --version
  - python3 -m pip install ansible molecule[docker]
  - ansible --version
  - molecule --version

molecule:
  stage: test
  script:
    - cd roles/testrole && molecule test
```

GitLab Runner is used to run your jobs and send the results back to GitLab. By tagging a Runner for the types of jobs it can handle, you can make sure shared Runners will only run the jobs they are equipped to run.

Here is another example using Docker, virtualenv and tags on Centos 7.

```
---
stages:
  - test

variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.pip"
  GIT_STRATEGY: clone

cache:
  paths:
    - .pip/
    - virtenv/

before_script:
  - python -V
  - pip install virtualenv
  - virtualenv virtenv
  - source virtenv/bin/activate
  - pip install ansible molecule docker
  - ansible --version
  - molecule --version
```

(continues on next page)

(continued from previous page)

```

- docker --version

molecule:
  stage: test
  tags:
    - molecule-jobs
  script:
    - molecule test

```

3.2.4 Azure Pipelines

Azure Pipelines projects rely on the `azure-pipelines.yml` file within the root folder of a repository. There are a number of pre-requisites for running CI/CD within Azure on self-hosted runners if you intend on using the `UsePythonVersion` task. Details of this can be found in the [Use Python Version Task](#) documentation.

```

---
trigger:
- main

pool:
  vmImage: ubuntu-16.04

steps:
- checkout: git://project-name/role-name
  path: role-name
- task: UsePythonVersion@0
  inputs:
    versionSpec: '3.8'
- script: python -m pip install "molecule[lint]" "python-vagrant" "molecule-vagrant"
  ↪"ansible"
  displayName: Install dependencies
- script: python -m pip install "python-tss-sdk"
  displayName: Role-specific dependencies
- script: |
  export PATH="$PATH:/home/<user>/.local/bin/"
  cd $(Agent.BuildDirectory)/role-name
  molecule test
  displayName: Test relevant platforms

```

Whilst the pipeline checks out your code initially as part of the pipeline task, by default, it checks it out into a directory named `s` within `$(Agent.BuildDirectory)`. If you checkout one other repository, the `s` is substituted with the path provided in that checkout. If you checkout multiple roles (e.g. some private roles within your Azure organisation) then the `s` structure is used, hence the importance of the `cd $(Agent.BuildDirectory)/role-name` which ensures you are in the correct directory regardless of format. Check the [Azure Build Variables](#) documentation for more detailed information on these.

The `export PATH` is required to ensure you can use the `molecule/ansible` shell scripts. Azure doesn't add these by

default.

3.2.5 Jenkins Pipeline

Jenkins projects can also be defined in a file, by default named Jenkinsfile in the top folder of a repository. Two syntax are available, Declarative and Scripted. Here is an example using the declarative syntax, setting up a virtualenv and testing an Ansible role via Molecule.

```
pipeline {
  agent {
    // Node setup : minimal centos7, plugged into Jenkins, and
    // git config --global http.sslVerify false
    // sudo yum -y install https://centos7.iuscommunity.org/ius-release.rpm
    // sudo yum -y install python36u python36u-pip python36u-devel git curl gcc
    // git config --global http.sslVerify false
    // sudo curl -fsSL get.docker.com | bash
    label 'Molecule_Slave'
  }
  stages {
    stage ('Get latest code') {
      steps {
        checkout scm
      }
    }
    stage ('Setup Python virtual environment') {
      steps {
        sh '''
          export HTTP_PROXY=http://10.123.123.123:8080
          export HTTPS_PROXY=http://10.123.123.123:8080
          python3 -m pip install virtualenv
          virtualenv virtenv
          source virtenv/bin/activate
          python3 -m pip install --upgrade ansible molecule docker
          '''
      }
    }
    stage ('Display versions') {
      steps {
        sh '''
          source virtenv/bin/activate
          docker -v
          python -V
          ansible --version
          molecule --version
          '''
      }
    }
  }
}
```

(continues on next page)

(continued from previous page)

```

stage ('Molecule test') {
  steps {
    sh '''
      source virtenv/bin/activate
      molecule test
    '''
  }
}

```

The following *Jenkinsfile* uses the Toolset image.

```

pipeline {
  agent {
    docker {
      image 'quay.io/ansible/toolset'
      args '-v /var/run/docker.sock:/var/run/docker.sock'
    }
  }

  stages {

    stage ('Display versions') {
      steps {
        sh '''
          docker -v
          python -V
          ansible --version
          molecule --version
        '''
      }
    }

    stage ('Molecule test') {
      steps {
        sh 'sudo molecule test --all'
      }
    }

  } // close stages
} // close pipeline

```

Note: For Jenkins to work properly using a *Multibranch Pipeline* or a *GitHub Organisation* - as used by Blue Ocean, the role name in the scenario converge.yml should be changed to perform a lookup of the role root directory. For example :

```
---
```

(continues on next page)

(continued from previous page)

```
- name: Converge
  hosts: all
  roles:
    - role: "{{ lookup('env', 'MOLECULE_PROJECT_DIRECTORY') | basename }}"
```

This is the cleaner of the current choices. See [issue1567_comment](#) for additional detail.

3.2.6 Tox

Tox is a generic virtualenv management, and test command line tool. Tox can be used in conjunction with [Factors](#) and Molecule, to perform scenario tests.

To test the role against multiple versions of Ansible.

```
[tox]
minversion = 1.8
envlist = py{27}-ansible{20,21,22}
skipdist = true

[testenv]
passenv = *
deps =
  -requirements.txt
  ansible20: ansible==2.0.2.0
  ansible21: ansible==2.1.2.0
  ansible22: ansible==2.2.0.0
commands =
  molecule test
```

To view the factor generated tox environments run `tox -l`.

If using the `--parallel` functionality of Tox (version 3.7 onwards), Molecule must be made aware of the parallel testing by setting a `MOLECULE_EPHEMERAL_DIRECTORY` environment variable per environment. In addition, we export a `TOX_ENVNAME` environment variable, it's the name of our tox env.

```
[tox]
minversion = 3.7
envlist = py{36}_ansible{23,24}
skipdist = true

[testenv]
deps =
  -requirements.txt
  ansible23: ansible==2.3
  ansible24: ansible==2.4
commands =
  molecule test
setenv =
  TOX_ENVNAME={envname}
  MOLECULE_EPHEMERAL_DIRECTORY=/tmp/{envname}
```

You also must include the `TOX_ENVNAME` variable in name of each platform in `molecule.yml` configuration file. This way, their names won't create any conflict.

```
---
dependency:
  name: galaxy
driver:
  name: docker
platforms:
  - name: instance1-$TOX_ENVNAME
    image: mariadb
  - name: instance2-$TOX_ENVNAME
    image: retr0h/centos7-systemd-ansible:latest
    privileged: True
    command: /usr/sbin/init
provisioner:
  name: ansible
verifier:
  name: testinfra
```

3.3 Command Line Reference

3.3.1 Check

3.3.2 Clean Up

3.3.3 Converge

Converge will execute the sequence necessary to converge the instances.

3.3.4 Create

3.3.5 Dependency

3.3.6 Destroy

3.3.7 Idempotence

3.3.8 Init

3.3.9 Lint

3.3.10 List

3.3.11 Login

3.3.12 Matrix

Matrix will display the subcommand's ordered list of actions, which can be changed in `scenario` configuration.

3.3.13 Prepare

3.3.14 Side Effect

3.3.15 Syntax

3.3.16 Test

Test will execute the sequence necessary to test the instances.

3.3.17 Verify

3.4 Configuration

3.4.1 Prerun

In order to help Ansible find used modules and roles, molecule will perform a prerun set of actions. These involve installing dependencies from `requirements.yml` specified at project level, install a standalone role or a collection. The destination is `project_dir/.cache` and the code itself was reused from `ansible-lint`, which has to do the same actions. (Note: `ansible-lint` is not included with molecule.)

This assures that when you include a role inside molecule playbooks, Ansible will be able to find that role, and that the include is exactly the same as the one you are expecting to use in production.

If for some reason the prerun action does not suits your needs, you can still disable it by adding `prerun: false` inside the configuration file.

Keep in mind that you can add this value to the `.config/molecule/config.yml` file, in your `$HOME` or at the root of your project, in order to avoid adding it to each scenario.

3.4.2 Variable Substitution

class `molecule.interpolation.Interpolator`

Configuration options may contain environment variables.

For example, suppose the shell contains `VERIFIER_NAME=testinfra` and the following `molecule.yml` is supplied.

```
verifier:  
  - name: ${VERIFIER_NAME}
```

Molecule will substitute `$VERIFIER_NAME` with the value of the `VERIFIER_NAME` environment variable.

Warning: If an environment variable is not set, Molecule substitutes with an empty string.

Both `$VARIABLE` and `${VARIABLE}` syntax are supported. Extended shell-style features, such as `${VARIABLE-default}` and `${VARIABLE:-default}` are also supported. Even the default as another environment variable is supported like `${VARIABLE-$DEFAULT}` or `${VARIABLE:-$DEFAULT}`. An empty string is returned when both variables are undefined.

If a literal dollar sign is needed in a configuration, use a double dollar sign (`$$`).

Molecule will substitute special `MOLECULE_` environment variables defined in `molecule.yml`.

Important: Remember, the `MOLECULE_` namespace is reserved for Molecule. Do not prefix your own variables with `MOLECULE_`.

A file may be placed in the root of the project as `.env.yml`, and Molecule will read variables when rendering `molecule.yml`. See command usage.

Construct Interpolator.

There are following environment variables available in `molecule.yml`:

MOLECULE_DEBUG If debug is turned on or off

MOLECULE_FILE Path to molecule config file

MOLECULE_ENV_FILE Path to molecule environment file

MOLECULE_STATE_FILE ?

MOLECULE_INVENTORY_FILE Path to generated inventory file

MOLECULE_EPHEMERAL_DIRECTORY Path to generated directory, usually `~/.cache/molecule/<scenario-name>`

MOLECULE_SCENARIO_DIRECTORY Path to scenario directory

MOLECULE_PROJECT_DIRECTORY Path to your project directory

MOLECULE_INSTANCE_CONFIG ?

MOLECULE_DEPENDENCY_NAME Dependency type name, usually 'galaxy'

MOLECULE_DRIVER_NAME Name of the molecule scenario driver

MOLECULE_PROVISIONER_NAME Name of the provisioner tool (usually 'ansible')

MOLECULE_REPORT Name of HTML file where to dump execution report.

MOLECULE_SCENARIO_NAME Name of the scenario

MOLECULE_VERBOSITY Determine Ansible verbosity level.

MOLECULE_VERIFIER_NAME Name of the verifier tool (usually 'ansible')

MOLECULE_VERIFIER_TEST_DIRECTORY ?

3.4.3 Dependency

Testing roles may rely upon additional dependencies. Molecule handles managing these dependencies by invoking configurable dependency managers.

Ansible Galaxy

Shell

3.4.4 Driver

Molecule uses [Ansible](#) to manage instances to operate on. Molecule supports any provider [Ansible](#) supports. This work is offloaded to the *provisioner*.

The driver's name is specified in *molecule.yml*, and can be overridden on the command line. Molecule will remember the last successful driver used, and continue to use the driver for all subsequent subcommands, or until the instances are destroyed by Molecule.

Important: The verifier must support the Ansible provider for proper Molecule integration.

The driver's python package requires installation.

Delegated

3.4.5 Lint

Starting with v3, Molecule handles project linting by invoking and external lint commands as exemplified below.

The decision to remove the complex linting support was not easily taken as we do find it very useful. The issue was that molecule runs on scenarios and linting is usually performed at repository level.

It makes little sense to perform linting in more than one place per project. Molecule was able to use up to three linters and while it was aimed to be flexible about them, it ended up creating more confusions to the users. We decided to maximize flexibility by just calling an external shell command.

Note: `ansible-lint` is not included with molecule. The `molecule[lint]` extra does not install `ansible-lint`.

```
lint: |
  set -e
  yamllint .
  ansible-lint
  flake8
```

The older format is no longer supported and you have to update the `molecule.yml` when you upgrade. If you don't want to do any linting, it will be enough to remove all lint related sections from the file.

```
# old v2 format, no longer supported
lint:
  name: yamllint
  enabled: true
provisioner:
  lint:
    name: ansible-lint
    options: ...
    env: ...
verifier:
  lint:
    name: flake8
```

3.4.6 Platforms

3.4.7 Provisioner

Molecule handles provisioning and converging the role.

Ansible

3.4.8 Scenario

Molecule treats scenarios as a first-class citizens, with a top-level configuration syntax.

3.4.9 State

An internal bookkeeping mechanism.

3.4.10 Verifier

Molecule handles role testing by invoking configurable verifiers.

Ansible

Testinfra

COMMON MOLECULE USE CASES

4.1 Common Molecule Use Cases

4.1.1 Running inside a container

Molecule is built into a Docker image by the [Toolset](#) project.

Any questions or bugs related to use of Molecule from within a container should be addressed by the [Toolset](#) project.

4.1.2 Docker With Non-Privileged User

The default Molecule Docker driver executes Ansible playbooks as the root user. If your workflow requires a non-privileged user, then adapt `molecule.yml` and `Dockerfile.j2` as follows.

Append the following code block to the end of `Dockerfile.j2`. It creates an `ansible` user with passwordless sudo privileges.

The variable `SUDO_GROUP` depends on the target distribution.

```
# Create `ansible` user with sudo permissions and membership in `DEPLOY_GROUP`
ENV ANSIBLE_USER=ansible SUDO_GROUP=wheel DEPLOY_GROUP=deployer
RUN set -xe \
  && groupadd -r ${ANSIBLE_USER} \
  && groupadd -r ${DEPLOY_GROUP} \
  && useradd -m -g ${ANSIBLE_USER} ${ANSIBLE_USER} \
  && usermod -aG ${SUDO_GROUP} ${ANSIBLE_USER} \
  && usermod -aG ${DEPLOY_GROUP} ${ANSIBLE_USER} \
  && sed -i "/^%${SUDO_GROUP}/s/ALL$/NOPASSWD:ALL/g" /etc/sudoers
```

Modify `provisioner.inventory` in `molecule.yml` as follows:

```
platforms:
  - name: instance
    image: quay.io/centos/centos:stream8
    # ...
```

```
provisioner:
  name: ansible
  # ...
  inventory:
    host_vars:
```

(continues on next page)

(continued from previous page)

```
# setting for the platform instance named 'instance'
instance:
  ansible_user: ansible
```

Make sure to use your **platform instance name**. In this case `instance`.

An example for a different platform instance name:

```
platforms:
- name: centos8
  image: quay.io/centos/centos:stream8
  # ...
```

```
provisioner:
  name: ansible
  # ...
inventory:
  host_vars:
    # setting for the platform instance named 'centos8'
    centos8:
      ansible_user: ansible
```

To test it, add the following task to `tasks/main.yml`. It fails, because the non-privileged user is not allowed to create a folder in `/opt/`. This needs to be performed using `sudo`.

To perform the task using `sudo`, uncomment `become: yes`. Now the task will succeed.

```
- name: Create apps dir
  file:
    path: /opt/examples
    owner: ansible
    group: deployer
    mode: 0775
    state: directory
  # become: yes
```

Don't forget to run `molecule destroy` if image has already been created.

4.1.3 Podman inside Docker

Sometimes your CI system comes prepared to run with Docker but you want to test podman into it. This `prepare.yml` playbook would let podman run inside a privileged Docker host by adding some required settings:

```
- name: prepare
  hosts: podman-in-docker
  tasks:
    - name: install fuse-overlayfs
      package:
        name:
          - fuse-overlayfs

    - name: create containers config dir
```

(continues on next page)

(continued from previous page)

```

file:
  group: root
  mode: a=rX,u+w
  owner: root
  path: /etc/containers
  state: directory

- name: make podman use fuse-overlayfs storage
copy:
  content: |
    # See man 5 containers-storage.conf for more information
    [storage]
    driver = "overlay"
    [storage.options.overlay]
    mount_program = "/usr/bin/fuse-overlayfs"
    mountopt = "nodev,metacopy=on"
  dest: /etc/containers/storage.conf
  group: root
  mode: a=r,u+w
  owner: root

- name: make podman use cgroupfs cgroup manager
copy:
  content: |
    # See man 5 libpod.conf for more information
    cgroup_manager = "cgroupfs"
  dest: /etc/containers/libpod.conf
  group: root
  mode: a=r,u+w
  owner: root

```

Another option is to configure the same settings directly into the `molecule.yml` definition:

```

driver:
  name: podman
platforms:
  - name: podman-in-docker
    # ... other options
  cgroup_manager: cgroupfs
  storage_opt: overlay.mount_program=/usr/bin/fuse-overlayfs
  storage_driver: overlay

```

At the time of writing, Gitlab CI shared runners run privileged Docker hosts and are suitable for these workarounds.

4.1.4 Systemd Container

To start a service which requires systemd, in a non-privileged container, configure `molecule.yml` with a systemd compliant image, tmpfs, volumes, and command as follows.

```
platforms:
- name: instance
  image: quay.io/centos/centos:stream8
  command: /sbin/init
  tmpfs:
  - /run
  - /tmp
  volumes:
  - /sys/fs/cgroup:/sys/fs/cgroup:ro
```

When needed, such security profiles can be reused (for example the one available in Fedora):

```
platforms:
- name: instance
  image: debian:stretch
  command: /sbin/init
  security_opts:
  - seccomp=path/to/seccomp.json
  tmpfs:
  - /run
  - /tmp
  volumes:
  - /sys/fs/cgroup:/sys/fs/cgroup:ro
```

The developer can also opt to start the container with extended privileges, by either giving it `SYS_ADMIN` capabilities or running it in privileged mode.

Important: Use caution when using privileged mode or `SYS_ADMIN` capabilities as it grants the container elevated access to the underlying system.

To limit the scope of the extended privileges, grant `SYS_ADMIN` capabilities along with the same image, command, and volumes as shown in the non-privileged example.

```
platforms:
- name: instance
  image: quay.io/centos/centos:stream8
  command: /sbin/init
  capabilities:
  - SYS_ADMIN
  volumes:
  - /sys/fs/cgroup:/sys/fs/cgroup:ro
```

To start the container in privileged mode, set the `privileged` flag along with the same image and command as shown in the non-privileged example.

```
platforms:
- name: instance
  image: quay.io/centos/centos:stream8
```

(continues on next page)

(continued from previous page)

```
command: /sbin/init
privileged: True
```

4.1.5 Monolith Repo

Molecule is generally used to test roles in isolation. However, it can also test roles from a monolith repo.

The role initialized with Molecule (baz in this case) would simply reference the dependent roles via its `converge.yml` or meta dependencies.

Molecule can test complex scenarios leveraging this technique.

```
$ cd monolith-repo/roles/baz
$ molecule test
```

Molecule is simply setting the `ANSIBLE_*` environment variables. To view the environment variables set during a Molecule operation pass the `--debug` flag.

```
$ molecule --debug test

DEBUG: ANSIBLE ENVIRONMENT
---
ANSIBLE_CONFIG: /private/tmp/monolith-repo/roles/baz/molecule/default/.molecule/ansible.
↳ cfg
ANSIBLE_FILTER_PLUGINS: /Users/jodewey/.pyenv/versions/2.7.13/lib/python2.7/site-
↳ packages/molecule/provisioner/ansible/plugins/filters:/private/tmp/monolith-repo/roles/
↳ baz/plugins/filters:/private/tmp/monolith-repo/roles/baz/molecule/default/.molecule/
↳ plugins/filters
ANSIBLE_LIBRARY: /Users/jodewey/.pyenv/versions/2.7.13/lib/python2.7/site-packages/
↳ molecule/provisioner/ansible/plugins/libraries:/private/tmp/monolith-repo/roles/baz/
↳ library:/private/tmp/monolith-repo/roles/baz/molecule/default/.molecule/library
ANSIBLE_ROLES_PATH: /private/tmp/monolith-repo/roles:/private/tmp/monolith-repo/roles/
↳ baz/molecule/default/.molecule/roles
```

Molecule can be customized any number of ways. Updating the provisioner's `env` section in `molecule.yml` to suit the needs of the developer and layout of the project.

```
provisioner:
  name: ansible
  env:
    ANSIBLE_${VAR}: $VALUE
```

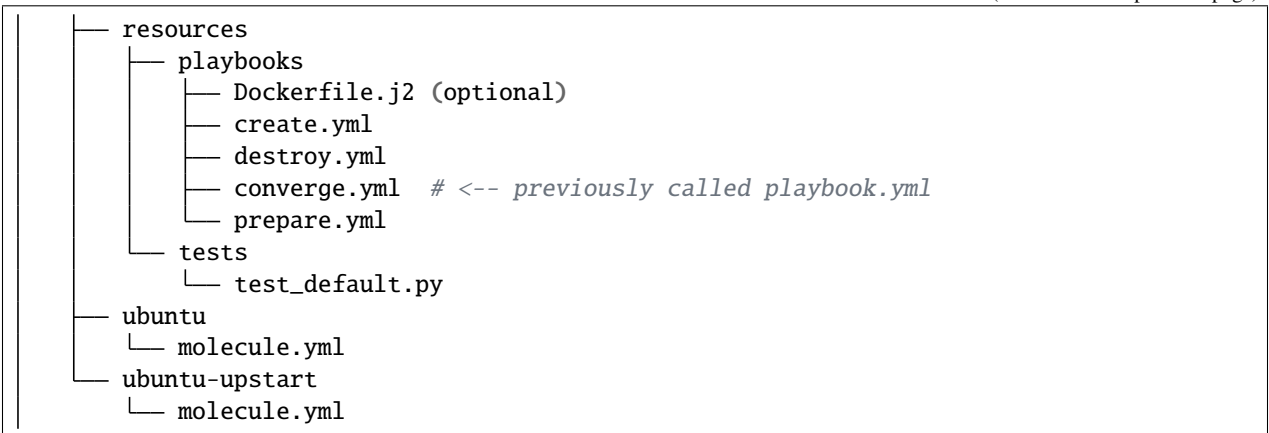
4.1.6 Sharing Across Scenarios

Playbooks and tests can be shared across scenarios.

```
$ tree shared-tests
shared-tests
├── molecule
│   └── centos
│       └── molecule.yml
```

(continues on next page)

(continued from previous page)



Tests and playbooks can be shared across scenarios.

In this example the *tests* directory lives in a shared location and `molecule.yml` points to the shared tests.

```

verifier:
  name: testinfra
  directory: ../resources/tests/

```

In this second example the actions *create*, *destroy*, *converge* and *prepare* are loaded from a shared directory.

```

provisioner:
  name: ansible
  playbooks:
    create: ../resources/playbooks/create.yml
    destroy: ../resources/playbooks/destroy.yml
    converge: ../resources/playbooks/converge.yml
    prepare: ../resources/playbooks/prepare.yml

```

4.1.7 Running Molecule processes in parallel mode

Important: This functionality should be considered experimental. It is part of ongoing work towards enabling parallelizable functionality across all moving parts in the execution of the Molecule feature set.

Note: Only the following sequences support parallelizable functionality:

- `check_sequence: molecule check --parallel`
- `destroy_sequence: molecule destroy --parallel`
- `test_sequence: molecule test --parallel`

It is currently only available for use with the Docker driver.

When Molecule receives the `--parallel` flag it will generate a **UUID** for the duration of the testing sequence and will use that unique identifier to cache the run-time state for that process. The parallel Molecule processes cached state and created instances will therefore not interfere with each other.

Molecule uses a new and separate caching folder for this in the `$HOME/.cache/molecule_parallel` location. Molecule exposes a new environment variable `MOLECULE_PARALLEL` which can enable this functionality.

It is possible to run Molecule processes in parallel using another tool to orchestrate the parallelization (such as [GNU Parallel](#) or [Pytest](#)). If you do so, make sure Molecule knows it is running in parallel mode by specifying the `--parallel` flag to your command(s) to avoid concurrency issues.

4.2 FAQ

4.2.1 Why is my idempotence action failing?

It is important to understand that Molecule does not do anything further than the default functionality of Ansible when determining if your tasks are idempotent or not. Molecule will simply run the converge action twice and check against Ansible's standard output.

Therefore, if you are seeing idempotence failures, it is typically related to the underlying Ansible report and not Molecule.

If you are facing idempotence failures and intend to raise a bug on our issue tracker, please first manually run `molecule converge` twice and confirm that Ansible itself is reporting task idempotence (`changed=0`).

4.2.2 Why does Molecule make so many shell calls?

Ansible provides a Python API. However, it is not intended for [direct consumption](#). We wanted to focus on making Molecule useful, so our efforts were spent consuming Ansible's CLI.

Since we already consume Ansible's CLI, we decided to call additional binaries through their respective CLI.

Note: This decision may be reevaluated later.

4.2.3 Why does Molecule only support Ansible versions 2.2 and later?

- Ansible 2.2 is the first good release in the Ansible 2 lineup.
- The modules needed to support the drivers did not exist pre 2.2 or were not sufficient.

4.2.4 Why are playbooks used to provision instances?

Simplicity. Ansible already supports numerous cloud providers. Too much time was spent in Molecule v1, re-implementing a feature that already existed in the core Ansible modules.

4.2.5 Have you thought about using Ansible's python API instead of playbooks?

This was [evaluated](#) early on. It was a toss up. It would provide simplicity in some situations and complexity in others. Developers know and understand playbooks. Decided against a more elegant and sexy solution.

4.2.6 Why are there multiple scenario directories and molecule.yml files?

Again, simplicity. Rather than defining an all encompassing config file opted to normalize. Molecule simply loops through each scenario applying the scenario's molecule.yml.

Note: This decision may be reevaluated later.

4.2.7 Are there similar tools to Molecule?

- Ansible's own [Testing Strategies](#)
- [ansible-test](#) (abandoned?)
- [RoleSpec](#)

4.2.8 Can I run Molecule processes in parallel?

Please see [Running Molecule processes in parallel mode](#) for usage.

4.2.9 Can I specify random instance IDs in my molecule.yml?

This depends on the CI provider but the basic recipe is as follows.

Setup your molecule.yml to look like this:

```
platforms:  
- name: "instance-`${INSTANCE_UUID}`"
```

Then in your CI provider environment, for example, Gitlab CI, setup:

```
variables:  
  INSTANCE_UUID: "${CI_JOB_ID}"
```

Where CI_JOB_ID is the random variable that Gitlab provides.

Molecule will resolve the INSTANCE_UUID environment variable when creating and looking up the instance name. You can confirm all is in working order by running `molecule list`.

4.2.10 Can I test Ansible Collections with Molecule?

This is not currently officially supported. Also, collections remain in “tech preview” status. However, you can take a look at [this blog post](#) outlining a workable “DIY” solution as a stop gap for now.

4.2.11 Does Molecule support monorepos?

Yes, roles contained in a [monorepo](#) with other roles are automatically picked up and `ANSIBLE_ROLES_PATH` is set accordingly. See [this page](#) for more information.

4.2.12 How can I add development/testing-only dependencies?

Sometimes, it’s desirable to only run a dependency role when developing your role with molecule, but not impose a hard dependency on the role itself; for example when you rely on one of its side effects. This can be achieved by an approach like this in your role’s `meta/main.yml`:

```
---
dependencies:
  - role: <your-dependee-role>
    when: lookup('env', 'MOLECULE_FILE')
```


CONTRIBUTING TO MOLECULE

5.1 Contributing

- To see what's planned see the [Molecule Project Board](#).
- Join the Molecule [community working group](#) if you would like to influence the direction of the project.

5.1.1 Talk to us

Join us in [#ansible-molecule](#) on [libera.chat](#) irc, or [molecule-users Forum](#).

The full list of Ansible email lists and IRC channels can be found in the [communication page](#).

Guidelines

- We are interested in various different kinds of improvement for Molecule; please feel free to raise an [Issue](#) if you would like to work on something major to ensure efficient collaboration and avoid duplicate effort.
- Create a topic branch from where you want to base your work.
- Make sure you have added tests for your changes.
- Although not required, it is good to sign off commits using `git commit --signoff`, and agree that usage of `--signoff` constitutes agreement with the terms of [DCO 1.1](#).
- Run all the tests to ensure nothing else was accidentally broken.
- Reformat the code by following the formatting section below.
- Submit a pull request.

Code Of Conduct

Please see our [Code of Conduct](#) document.

Pull Request Life Cycle and Governance

- If your PRs get stuck [join us on IRC](#) or add to the [working group agenda](#).
- The code style is what is enforced by CI, everything else is off-topic.
- All PRs must be reviewed by one other person. This is enforced by GitHub. Larger changes require +2.

Testing

Molecule has an extensive set of unit and functional tests. Molecule uses [Tox](#) factors to generate a matrix of python x Ansible x unit/functional tests. Manual setup required as of this time.

5.1.2 Dependencies

Tests will be skipped when the driver's binary is not present.

Install the test framework [Tox](#).

```
$ python3 -m pip install tox
```

5.1.3 Full

Run all tests, including linting and coverage reports. This should be run prior to merging or submitting a pull request.

```
$ tox
```

5.1.4 List available scenarios

List all available scenarios. This is useful to target specific Python and Ansible version for the functional and unit tests.

```
$ tox -av
```

5.1.5 Unit

Run all unit tests with coverage.

```
$ tox -e 'py{27,35,36,37,38}-unit'
```

Run all unit tests for a specific version of Python .

```
$ tox -e py37-unit
```


5.1.6 Linting

Linting is performed by a combination of linters.

Run all the linters (some perform changes to conform the code to the style rules).

```
$ tox -e lint
```

5.1.7 Documentation

Generate the documentation, using `sphinx`.

```
$ tox -e docs
```

Documentation

5.1.8 Working with InterSphinx

In the `conf.py`, we define an `intersphinx_mapping` which provides the base URLs for conveniently linking to other Sphinx documented projects. In order to find the correct link syntax and text you can link to, you can quickly inspect the reference from the command line.

For example, if we would like to link to a specific part of the Ansible documentation, we could first run the following command:

```
python -m sphinx.ext.intersphinx https://docs.ansible.com/ansible/latest/objects.inv
```

And then see the entire Sphinx listing. We see entries that look like:

```
py:attribute
  AnsibleModule._debug  api/index.html#AnsibleModule._debug
```

With which we can link out to using the following syntax:

```
:py:attribute:`AnsibleModule._debug`
```

Credits

Based on the good work of John Dewey ([@retr0h](#)) and other [contributors](#). Active member list can be seen at [Molecule working group](#).

REFERENCES AND APPENDICES

- genindex

EXTERNAL RESOURCES

Below you can see a list of useful articles and presentations, recently updated being listed first:

- [Ansible Collections: Role Tests with Molecule](#) @ericssystemin
- [Molecule v3 Slides](#) @ssbarnea.
- [Testing your Ansible roles with Molecule](#) @geerlinguy
- [How to test Ansible and don't go nuts](#) @ultral

INDEX

I

Interpolator (*class in molecule.interpolation*), 20