Collection structure

- Collection directories and files
  - galaxy.yml
  - extensions directory
  - docs directory
    - Plugin and module documentation
  - plugins directory
    - module_utils
  - roles directory
  - playbooks directory
  - tests directory
  - meta directory
    - runtime.yml
    - execution-environment.yml

A collection is a simple data structure. None of the directories are required unless you have specific content that belongs in one of them. A collection does require a `galaxy.yml` file at the root level of the collection. This file contains all of the metadata that Galaxy and other tools need in order to package, build and publish the collection.

Collection directories and files

A collection can contain these directories and files:

```
collection/
├── docs/
├── extensions/
│   └── audit/
│       └── event_query.yml
├── galaxy.yml
```

```
├── meta/
│   └── runtime.yml
├── plugins/
│   ├── modules/
│   │   └── module1.py
│   ├── inventory/
│   └── .../
├── README.md
├── roles/
│   ├── role1/
│   ├── role2/
│   └── .../
├── playbooks/
│   ├── files/
│   ├── vars/
│   ├── templates/
│   └── tasks/
└── tests/
```

> * Ansible only accepts `.md` extensions for the `README` file and any files in the `/docs` folder. * See the ansible-collections GitHub Org for examples of collection structure. * Not all directories are currently in use. Those are placeholders for future features.

galaxy.yml

A collection must have a `galaxy.yml` file that contains the necessary information to build a collection artifact. See `collections_galaxy_meta` for details.

extensions directory

Use the `extensions` directory to add content and metadata that extends the functionality of your collection beyond the standard plugins and roles. Each subdirectory groups related files for a specific capability. For example, the `extensions/audit/event_query.yml` file defines resource reporting queries that map module return values to a standardized resource taxonomy. See `resource_reporting` for details on adding resource reporting to your collection.

docs directory

Use the `docs` folder to describe how to use the roles and plugins the collection provides, role requirements, and so on.

For certified collections, Automation Hub displays documents written in markdown in the main `docs` directory with no subdirectories. This documentation is not published to docs.ansible.com.

For community collections included in the Ansible PyPI package, docs.ansible.com displays documents written in reStructuredText (.rst) in a docsite/rst/ subdirectory. Define the structure of your extra documentation in `docs/docsite/extra-docs.yml`:

```
---
sections:
- title: Scenario Guide
  toctree:
    - scenario_guide
```

The index page of the documentation for your collection displays the title you define in `docs/docsite/extra-docs.yml` with a link to your extra documentation. For an example, see the community.docker collection repo and the community.docker collection documentation.

You can add extra links to your collection index page and plugin pages with the `docs/docsite/links.yml` file. This populates the links under Description and Communications headings as well as links at the end of the individual plugin pages. See the collection_template links.yml file for a complete description of the structure and use of this file to create links.

Plugin and module documentation

Keep the specific documentation for plugins and modules embedded as Python docstrings. Use `ansible-doc` to view documentation for plugins inside a collection:

```
ansible-doc -t lookup my_namespace.my_collection.lookup1
```

The `ansible-doc` command requires the fully qualified collection name (FQCN) to display specific plugin documentation. In this example, `my_namespace` is the Galaxy namespace and `my_collection` is the collection name within that namespace.

> The Galaxy namespace of an Ansible collection is defined in the `galaxy.yml` file. It can be different from the GitHub organization or repository name.

plugins directory

Add a 'per plugin type' specific subdirectory here, including `module_utils` which is usable not only by modules, but by most plugins by using their FQCN. This is a way to distribute modules, lookups, filters, and so on without having to import a role in every play.

Vars plugins in collections are not loaded automatically, and always require being explicitly enabled by using their fully qualified collection name. See `enable_vars` for details.

Cache plugins in collections may be used for fact caching, but are not supported for inventory plugins.

module_utils

When coding with `module_utils` in a collection, the Python `import` statement needs to take into account the FQCN along with the `ansible_collections` convention. The resulting Python import will look like `from ansible_collections.{namespace}.{collection}.plugins.module_utils.{util} import {something}`

The following example snippets show a Python and PowerShell module using both default Ansible `module_utils` and those provided by a collection. In this example the namespace is `community`, the collection is `test_collection`. In the Python example the `module_util` in question is called `qradar` such that the FQCN is `community.test_collection.plugins.module_utils.qradar`:

```
from ansible.module_utils.basic import AnsibleModule
from ansible.module_utils.common.text.converters import to_text

from ansible.module_utils.six.moves.urllib.parse import urlencode, quote_plus
from ansible.module_utils.six.moves.urllib.error import HTTPError
from ansible_collections.community.test_collection.plugins.module_utils.qradar import
QRadarRequest

argspec = dict(
    name=dict(required=True, type='str'),
    state=dict(choices=['present', 'absent'], required=True),
)

module = AnsibleModule(
    argument_spec=argspec,
    supports_check_mode=True
)

qradar_request = QRadarRequest(
    module,
    headers={"Content-Type": "application/json"},
```

```
    not_rest_data_keys=['state']
)
```

Note that importing something from an `__init__.py` file requires using the file name:

```
from ansible_collections.namespace.collection_name.plugins.callback.__init__ import
CustomBaseClass
```

In the PowerShell example the `module_util` in question is called `hyperv` such that the FQCN is `community.test_collection.plugins.module_utils.hyperv`:

```
#!powershell
#AnsibleRequires -CSharpUtil Ansible.Basic
#AnsibleRequires -PowerShell
ansible_collections.community.test_collection.plugins.module_utils.hyperv

$spec = @{
    name = @{ required = $true; type = "str" }
    state = @{ required = $true; choices = @("present", "absent") }
}
$module = [Ansible.Basic.AnsibleModule]::Create($args, $spec)

Invoke-HyperVFunction -Name $module.Params.name

$module.ExitJson()
```

roles directory

Collection roles are mostly the same as existing roles, but with a couple of limitations:

> - Role names are now limited to contain only lowercase alphanumeric characters, plus `_` and start with an alpha character.
> - Roles in a collection cannot contain plugins any more. Plugins must live in the collection `plugins` directory tree. Each plugin is accessible to all roles in the collection.

The directory name of the role is used as the role name. Therefore, the directory name must comply with the above role name rules. The collection import into Galaxy will fail if a role name does not comply with these rules.

You can migrate 'traditional roles' into a collection but they must follow the rules above. You may need to rename roles if they don't conform. You will have to move or link any role-based plugins to the collection specific directories.

> For roles imported into Galaxy directly from a GitHub repository, setting the `role_name` value in the role's metadata overrides the role name used by Galaxy. For collections, that value is ignored. When importing a collection, Galaxy uses the role directory as the name of the role and ignores the `role_name` metadata value.

playbooks directory

In prior releases, you could reference playbooks in this directory using the full path to the playbook file from the command line. In ansible-core 2.11 and later, you can use the FQCN, `namespace.collection.playbook` (with or without extension), to reference the playbooks from the command line or from `import_playbook`. This will keep the playbook in 'collection context', as if you had added `collections: [ namespace.collection ]` to it.

You can have most of the subdirectories you would expect, such `files/`, `vars/` or `templates/` but not `roles/` since those are handled already in the collection.

Also, playbooks within a collection follow the same guidelines as any playbooks except for these few adjustments:

> - Directory: It must be in the `playbooks/` directory.
> - Hosts: The host should be defined as a variable so the users of a playbook do not mistakenly run the plays against their entire inventory (if the host is set to all). For example - `hosts: '{{target|default("all")}}'`.

To run the plays, users can now use such commands as `ansible-playbook --e 'targets=webservers'` or `ansible-playbook --limit webservers`. Either way, the collection owner should document their playbooks and how to use them in the `docs/` folder or `README` file.

tests directory

Ansible Collections are tested much like Ansible itself, by using the ansible-test utility which is released as part of Ansible, version 2.9.0 and newer. Because Ansible Collections are tested using the same tooling as Ansible itself, by using the ansible-test, all Ansible developer documentation for testing is applicable for authoring Collections Tests with one key concept to keep in mind.

See `testing_collections` for specific information on how to test collections with `ansible-test`.

When reading the `developing_testing` documentation, there will be content that applies to running Ansible from source code through a Git clone, which is typical of an Ansible developer. However, it is not always typical for an Ansible Collection author to be running Ansible from source but instead from a stable release, and to create Collections it is not necessary to run Ansible from source. Therefore, when references of dealing with ansible-test binary paths, command completion, or environment variables are presented throughout the `developing_testing` documentation; keep in mind that it is not needed for Ansible Collection Testing because the act of installing the stable release of Ansible containing ansible-test is expected to setup those things for you.

meta directory

runtime.yml

A collection can store some additional metadata in a `runtime.yml` file in the collection's `meta` directory. The `runtime.yml` file supports the top level keys:

- *requires_ansible*:
  The version of Ansible Core (ansible-core) required to use the collection. Multiple versions can be separated with a comma.

  ```
  requires_ansible: ">=2.10,<2.11"
  ```

  > although the version is a PEP440 Version Specifier under the hood, Ansible deviates from PEP440 behavior by truncating prerelease segments from the Ansible version. This means that Ansible 2.11.0b1 is compatible with something that `requires_ansible: ">=2.11"`.

- *plugin_routing*:
  Content in a collection that Ansible needs to load from another location or that has been deprecated/removed. The top level keys of `plugin_routing` are types of plugins, with individual plugin names as subkeys. To define a new location for a plugin, set the `redirect` field to another name. To deprecate a plugin, use the `deprecation` field to provide a custom warning message and the removal version or date. If the plugin has been renamed or moved to a new location, the `redirect` field should also be provided. If a plugin is being removed entirely, `tombstone` can be used for the fatal error message and removal version or date.

  ```
  plugin_routing:
    inventory:
      kubevirt:
        redirect: community.general.kubevirt
  ```

```
    my_inventory:
      tombstone:
        removal_version: "2.0.0"
        warning_text: my_inventory has been removed. Please use other_inventory
instead.
  modules:
    my_module:
      deprecation:
        removal_date: "2021-11-30"
        warning_text: my_module will be removed in a future release of this
collection. Use another.collection.new_module instead.
      redirect: another.collection.new_module
    podman_image:
      redirect: containers.podman.podman_image
  module_utils:
    ec2:
      redirect: amazon.aws.ec2
    util_dir.subdir.my_util:
      redirect: namespace.name.my_util
```

- *import_redirection*
  A mapping of names for Python import statements and their redirected locations.

```
import_redirection:
  ansible.module_utils.old_utility:
    redirect:
ansible_collections.namespace_name.collection_name.plugins.module_utils.new_location
```

- *action_groups*
  A mapping of groups and the list of action plugin and module names they contain. They may also have a special 'metadata' dictionary in the list, which can be used to include actions from other groups.

```
action_groups:
  groupname:
    # The special metadata dictionary. All action/module names should be strings.
    - metadata:
        extend_group:
          - another.collection.groupname
          - another_group
    - my_action
```

```
another_group:
  - my_module
  - another.collection.another_module
```

execution-environment.yml

If your collection has requirements, you can specify them in the `execution-environment.yml` file in the `meta` directory. This ensures users do not need to add these requirements manually when building `Execution Environments<getting_started_ee_index>` containing your collection. See the collection-level metadata guide for details.

`distributing_collections`
Learn how to package and publish your collection

`contributing_maintained_collections`
Guidelines for contributing to selected collections

`Communication<communication>`
Got questions? Need help? Want to share your ideas? Visit the Ansible communication guide

---

□□□ #2
□ □□□ □□□ 2026-10-31 14:34:10 UTC
□ □□□ □□□ 2026-10-31 14:34:11 UTC