Plugins

- Action plugins
- Become plugins
- Cache plugins
- Callback plugins
- Cliconf plugins
- Connection plugins
- Docs fragments
- Filter plugins
- Httpapi plugins
- Inventory plugins
- Lookup plugins
- Modules
- Module utilities
- Netconf plugins
- Working with plugins
- Shell plugins
- Strategy plugins
- Terminal plugins
- Test plugins
- Vars plugins

Action plugins

Action plugins act in conjunction with `modules <module_plugins>` to execute the actions required by playbook tasks. They usually execute automatically in the background doing prerequisite work before modules execute.

The 'normal' action plugin is used for modules that do not already have an action plugin. If necessary, you can `create custom action plugins <developing_actions>`.

Enabling action plugins

You can enable a custom action plugin by either dropping it into the `action_plugins` directory adjacent to your play, inside a role, or by putting it in one of the action plugin directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

Using action plugins

Action plugins are executed by default when an associated module is used; no additional action is required.

Plugin list

You cannot list action plugins directly, they show up as their counterpart modules:

Use `ansible-doc -l` to see the list of available modules. Use `ansible-doc <name>` to see plugin-specific documentation and examples. This should note if the module has a corresponding action plugin.

`cache_plugins`
Cache plugins

`callback_plugins`
Callback plugins

`connection_plugins`
Connection plugins

`inventory_plugins`
Inventory plugins

`shell_plugins`
Shell plugins

`strategy_plugins`
Strategy plugins

`vars_plugins`
Vars plugins

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

Become plugins

2.8

Become plugins work to ensure that Ansible can use certain privilege escalation systems when running the basic commands to work with the target machine as well as the modules required to execute the tasks specified in the play.

These utilities (`sudo`, `su`, `doas`, and so on) generally let you 'become' another user to execute a command with the permissions of that user.

Enabling Become Plugins

The become plugins shipped with Ansible are already enabled. Custom plugins can be added by placing them into a `become_plugins` directory adjacent to your play, inside a role, or by placing them in one of the become plugin directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

Using Become Plugins

In addition to the default configuration settings in `ansible_configuration_settings` or the `--become-method` command line option, you can use the `become_method` keyword in a play or, if you need to be 'host specific', the connection variable `ansible_become_method` to select the plugin to use.

You can further control the settings for each plugin with other configuration options detailed in the plugin themselves (linked below).

Plugin List

You can use `ansible-doc -t become -l` to see the list of available plugins. Use `ansible-doc -t become <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`inventory_plugins`
Inventory plugins

`callback_plugins`
Callback plugins

`filter_plugins`
Filter plugins

`test_plugins`
Test plugins

`lookup_plugins`
Lookup plugins

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

Cache plugins

Cache plugins allow Ansible to store gathered facts or inventory source data without the performance hit of retrieving them from the source.

The default cache plugin is the `memory <memory_cache>` plugin, which only caches the data for the current execution of Ansible. Other plugins with persistent storage are available to allow caching of the data across runs. Some of these cache plugins write to files, and others write to databases.

You can use different cache plugins for inventory and facts. If you enable inventory caching without setting an inventory-specific cache plugin, Ansible uses the fact cache plugin for both facts and inventory. If necessary, you can `create custom cache plugins <developing_cache_plugins>`.

A cache plugin implementation is an internal implementation detail, and should not be relied upon by external uses. The format of the data or how it is stored is a concern for the plugin, and this data may change or even be absent. As such, the cache of `ansible.builtin.jsonfile#cache`, `community.general.redis#cache`, or any other cache plugin should not be interrogated external to the cache plugin itself.

The cache maintained by a cache plugin is only to be used indirectly within a playbook, without any concept that the data even came from a cache.

Enabling fact cache plugins

Fact caching is always enabled. However, only one fact cache plugin can be active at a time. You can select the cache plugin to use for fact caching in the Ansible configuration, either with an environment variable:

```
export ANSIBLE_CACHE_PLUGIN=jsonfile
```

or in the `ansible.cfg` file:

```
[defaults]
fact_caching=redis
```

If the cache plugin is in a collection use the fully qualified name:

```
[defaults]
fact_caching = namespace.collection_name.cache_plugin_name
```

To enable a custom cache plugin, save it in one of the directory sources configured in `ansible.cfg <ansible_configuration_settings>` or in a collection and then reference it by FQCN.

You also need to configure other settings specific to each plugin. Consult the individual plugin documentation or the Ansible `configuration <ansible_configuration_settings>` for more details.

> [!NOTE] The existence of the cache, or an individual item in the cache should not be a hard requirement. Playbooks should not be written in a way as to potentially failing if the cache or a specific value is missing. A cache hit or miss should not affect a playbook operation.

Enabling inventory cache plugins

Inventory caching is disabled by default. To cache inventory data, you must enable inventory caching and then select the specific cache plugin you want to use. Not all inventory plugins support caching, so check the documentation for the inventory plugin(s) you want to use. You can enable inventory caching with an environment variable:

```
export ANSIBLE_INVENTORY_CACHE=True
```

or in the `ansible.cfg` file:

```
[inventory]
cache=True
```

or if the inventory plugin accepts a YAML configuration source, in the configuration file:

```
# dev.aws_ec2.yaml
plugin: aws_ec2
cache: True
```

Only one inventory cache plugin can be active at a time. You can set it with an environment variable:

```
export ANSIBLE_INVENTORY_CACHE_PLUGIN=jsonfile
```

or in the `ansible.cfg` file:

```
[inventory]
cache_plugin=jsonfile
```

or if the inventory plugin accepts a YAML configuration source, in the configuration file:

```
# dev.aws_ec2.yaml
plugin: aws_ec2
cache_plugin: jsonfile
```

To cache inventory with a custom plugin in your plugin path, follow the `developer guide on cache plugins<developing_cache_plugins>`.

To cache inventory with a cache plugin in a collection, use the FQCN:

```
[inventory]
cache_plugin=collection_namespace.collection_name.cache_plugin
```

If you enable caching for inventory plugins without selecting an inventory-specific cache plugin, Ansible falls back to caching inventory with the fact cache plugin you configured. Consult the individual inventory plugin documentation or the Ansible `configuration <ansible_configuration_settings>` for more details.

Using cache plugins

Cache plugins are used automatically once they are enabled.

Plugin list

You can use `ansible-doc -t cache -l` to see the list of available plugins. Use `ansible-doc -t cache <plugin name>` to see plugin-specific documentation and examples.

`action_plugins`
Action plugins

`callback_plugins`
Callback plugins

`connection_plugins`
Connection plugins

`inventory_plugins`
Inventory plugins

`shell_plugins`
Shell plugins

`strategy_plugins`
Strategy plugins

`vars_plugins`
Vars plugins

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

Callback plugins

Callback plugins enable adding new behaviors to Ansible when responding to events. By default, callback plugins control most of the output you see when running the command line programs, but can also be used to add additional output, integrate with other tools and marshal the events to a storage backend. If necessary, you can `create custom callback plugins <developing_callbacks>`.

Example callback plugins

The `log_plays <log_plays_callback>` callback is an example of how to record playbook events to a log file, and the `mail <mail_callback>` callback sends email on playbook failures.

The `say <say_callback>` callback responds with a computer-synthesized speech in relation to playbook events.

Enabling callback plugins

You can activate a custom callback, depending on its `NEEDS_ENABLED` property, by either dropping it into one of the callback directory sources configured in `ansible.cfg <ansible_configuration_settings>` or in a collection and referencing it in configuration by FQCN.

Plugins are loaded in alphanumeric order. For example, a plugin implemented in a file named 1_first.py would run before a plugin file named 2_second.py.

Most callbacks shipped with Ansible are disabled by default and need to be enabled in your `ansible.cfg <ansible_configuration_settings>` file in order to function. For example:

```
#callbacks_enabled = timer, mail, profile_roles,
collection_namespace.collection_name.custom_callback
```

Setting a callback plugin for `ansible-playbook`

You can only have one plugin be the main manager of your console output. If you want to replace the default, you should define `CALLBACK_TYPE = stdout` in the subclass and then configure the stdout plugin in `ansible.cfg <ansible_configuration_settings>`. For example:

```
stdout_callback = dense
```

or for my custom callback:

```
stdout_callback = mycallback
```

This only affects `ansible-playbook` by default.

Setting a callback plugin for ad hoc commands

The `ansible` ad hoc command specifically uses a different callback plugin for stdout, so there is an extra setting in `ansible_configuration_settings` you need to add to use the stdout callback defined above:

```
[defaults]
bin_ansible_callbacks=True
```

You can also set this as an environment variable:

```
export ANSIBLE_LOAD_CALLBACK_PLUGINS=1
```

Types of callback plugins

There are three types of callback plugins:

stdout callback plugins

> These plugins handle the main console output. Only one can be active. They always get the event first; the rest of the callbacks get the event in the order they are configured.

aggregate callback plugins

>

> Aggregate callbacks can add additional console output next to a stdout callback. This can be aggregate information at the end of a playbook run, additional per-task output, or anything else.

notification callback plugins

> Notification callbacks inform other applications, services, or systems. This can be anything from logging to databases, informing on errors in Instant Messaging applications, or sending emails when a server is unreachable.

Plugin list

You can use `ansible-doc -t callback -l` to see the list of available plugins. Use `ansible-doc -t callback <plugin name>` to see plugin-specific documentation and examples.

`action_plugins`
Action plugins

`cache_plugins`
Cache plugins

`connection_plugins`
Connection plugins

`inventory_plugins`
Inventory plugins

`shell_plugins`
Shell plugins

`strategy_plugins`
Strategy plugins

`vars_plugins`
Vars plugins

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

Cliconf plugins

Cliconf plugins are abstractions over the CLI interface to network devices. They provide a standard interface for Ansible to execute tasks on those network devices.

These plugins generally correspond one-to-one to network device platforms. Ansible loads the appropriate cliconf plugin automatically based on the `ansible_network_os` variable.

Adding cliconf plugins

You can extend Ansible to support other network devices by dropping a custom plugin into the `cliconf_plugins` directory.

Using cliconf plugins

The cliconf plugin to use is determined automatically from the `ansible_network_os` variable. There should be no reason to override this functionality.

Most cliconf plugins can operate without configuration. A few have additional options that can be set to affect how tasks are translated into CLI commands.

Plugins are self-documenting. Each plugin should document its configuration options.

Viewing cliconf plugins

These plugins have migrated to collections on Ansible Galaxy. If you installed Ansible version 2.10 or later using `pip`, you have access to several cliconf plugins. You can use `ansible-doc -t cliconf -l` to see the list of available plugins. Use `ansible-doc -t cliconf <plugin name>` to see plugin-specific documentation and examples.

`Ansible for Network Automation<network_guide>`
An overview of using Ansible to automate networking devices.

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

Connection plugins

Connection plugins allow Ansible to connect to the target hosts so it can execute tasks on them. Ansible ships with many connection plugins, but only one can be used per host at a time.

By default, Ansible ships with several connection plugins. The most commonly used are the `paramiko SSH<paramiko_connection>`, native ssh (just called `ssh<ssh_connection>`), and `local<local_connection>` connection types. All of these can be used in playbooks and with `/usr/bin/ansible` to decide how you want to talk to remote machines. If necessary, you can `create custom connection plugins <developing_connection_plugins>`. To change the connection plugin for your tasks, you can use the `connection` keyword.

The basics of these connection types are covered in the `getting started<intro_getting_started>` section.

`ssh` plugins

Because SSH is the default protocol used in system administration and the protocol most used in Ansible, SSH options are included in the command line tools. See `ansible-playbook` for more details.

Using connection plugins

You can set the connection plugin globally with `configuration<ansible_configuration_settings>`, at the command line (`-c`, `--connection`), as a `keyword <playbook_keywords>` in your play, or by setting a `variable<behavioral_parameters>`, most often in your inventory. For example, for Windows machines, you might want to set the `winrm <winrm_connection>` plugin as an inventory variable.

Most connection plugins can operate with minimal configuration. By default, they use the `inventory hostname<inventory_hostnames_lookup>` and defaults to find the target host.

Plugins are self-documenting. Each plugin should document its configuration options. The following are connection variables common to most connection plugins:

`ansible_host<magic_variables_and_hostvars>`
The name of the host to connect to, if different from the `inventory <intro_inventory>` hostname.

`ansible_port<faq_setting_users_and_ports>`
The ssh port number, for `ssh <ssh_connection>` and `paramiko_ssh <paramiko_connection>` it defaults

to 22.

`ansible_user<faq_setting_users_and_ports>`
The default username to use for log in. Most plugins default to the 'current user running Ansible'.

Each plugin might also have a specific version of a variable that overrides the general version. For example, `ansible_ssh_host` for the `ssh <ssh_connection>` plugin.

Plugin list

You can use `ansible-doc -t connection -l` to see the list of available plugins. Use `ansible-doc -t connection <plugin name>` to see plugin-specific documentation and examples.

`Working with Playbooks<working_with_playbooks>`
An introduction to playbooks

`callback_plugins`
Callback plugins

`filter_plugins`
Filter plugins

`test_plugins`
Test plugins

`lookup_plugins`
Lookup plugins

`vars_plugins`
Vars plugins

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

Docs fragments

Docs fragments allow you to document common parameters of multiple plugins or modules in a single place.

Enabling docs fragments

You can add a custom docs fragment by dropping it into a `doc_fragments` directory adjacent to your collection or role, just like any other plugin.

Using docs fragments

Only collection developers and maintainers use docs fragments. For more information on using docs fragments, see `module_docs_fragments` or `docfragments_collections`.

`developing_modules_general`
An introduction to creating Ansible modules

`developing_collections`
A guide to creating Ansible collections

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

Filter plugins

Filter plugins manipulate data. With the right filter, you can extract a particular value, transform data types and formats, perform mathematical calculations, split and concatenate strings, insert dates and times, and do much more. Ansible uses the `standard filters <jinja2:builtin-filters>` shipped with Jinja2 and adds some specialized filter plugins. You can `create custom Ansible filters as plugins <developing_filter_plugins>`.

Enabling filter plugins

You can add a custom filter plugin by dropping it into a `filter_plugins` directory adjacent to your play, inside a role, or by putting it in one of the filter plugin directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

Using filter plugins

You can use filters anywhere you can use templating in Ansible: in a play, in variables file, or a Jinja2 template for the `template <template_module>` module. For more information on using filter plugins, see `playbooks_filters`. Filters can return any type of data, but if you want to always return a boolean, (`true` or `false`) you should be looking at a test instead.

```
vars:
   yaml_string: "{{ some_variable|to_yaml }}"
```

Filters are the preferred way to manipulate data in Ansible, you can identify a filter because it is normally preceded by a `|`, with the expression on the left of it being the first input of the filter. Additional parameters may be passed into the filter itself as you would to most programming functions. These parameters can be either `positional` (passed in order) or `named` (passed as key=value pairs). When passing both types, positional arguments should go first.

```
passing_positional: "{{ (x == 32) | ternary('x is 32', 'x is not 32') }}"
passing_extra_named_parameters: "{{ some_variable | to_yaml(indent=8, width=1337) }}"
passing_both: "{{ some_variable | ternary('true value', 'false value', none_val='NULL') }}"
```

In the documentation, filters will always have a C(input) option that corresponds to the expression to the left of c(|). A C(positional:) field in the documentation will show which options are positional and in which order they are required.

Plugin list

You can use `ansible-doc -t filter -l` to see the list of available plugins. Use `ansible-doc -t filter <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`inventory_plugins`
Inventory plugins

`callback_plugins`
Callback plugins

`test_plugins`
Test plugins

`lookup_plugins`
Lookup plugins

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

Httpapi plugins

Httpapi plugins tell Ansible how to interact with a remote device's HTTP-based API and execute tasks on the device.

Each plugin represents a particular dialect of API. Some are platform-specific (Arista eAPI, Cisco NXAPI), while others might be usable on a variety of platforms (RESTCONF). Ansible loads the appropriate httpapi plugin automatically based on the `ansible_network_os` variable.

Adding httpapi plugins

You can extend Ansible to support other APIs by dropping a custom plugin into the `httpapi_plugins` directory. See `developing_plugins_httpapi` for details.

Using httpapi plugins

The httpapi plugin to use is determined automatically from the `ansible_network_os` variable.

Most httpapi plugins can operate without configuration. Additional options may be defined by each plugin.

Plugins are self-documenting. Each plugin should document its configuration options.

The following sample playbook shows the httpapi plugin for an Arista network device, assuming an inventory variable set as `ansible_network_os=eos` for the httpapi plugin to trigger off:

```
- hosts: leaf01
  connection: httpapi
  gather_facts: false
  tasks:

    - name: type a simple arista command
      eos_command:
        commands:
          - show version | json
      register: command_output
```

```
    - name: print command output to terminal window
      debug:
        var: command_output.stdout[0]["version"]
```

See the full working example on GitHub.

Viewing httpapi plugins

These plugins have migrated to collections on Ansible Galaxy. If you installed Ansible version 2.10 or later using `pip`, you have access to several httpapi plugins. You can use `ansible-doc -t httpapi -l` to see the list of available plugins. Use `ansible-doc -t httpapi <plugin name>` to see plugin-specific documentation and examples.

`Ansible for Network Automation<network_guide>`
An overview of using Ansible to automate networking devices.

`Developing network modules<developing_modules_network>`
How to develop network modules.

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

Inventory plugins

Inventory plugins allow users to point at data sources to compile the inventory of hosts that Ansible uses to target tasks, either using the `-i /path/to/file` and/or `-i 'host1, host2'` command line parameters or from other configuration sources. If necessary, you can `create custom inventory plugins <developing_inventory_plugins>`.

Enabling inventory plugins

Most inventory plugins shipped with Ansible are enabled by default or can be used with the `auto` plugin.

In some circumstances, for example, if the inventory plugin does not use a YAML configuration file, you may need to enable the specific plugin. You can do this by setting `enable_plugins` in your `ansible.cfg <ansible_configuration_settings>` file in the `[inventory]` section. Modifying this will override the default list of enabled plugins. Here is the default list of enabled plugins that ships with Ansible:

```
[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml
```

If the plugin is in a collection and is not being picked up by the auto statement, you can append the fully qualified name:

```
[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml,
namespace.collection_name.inventory_plugin_name
```

Or, if it is a local plugin, perhaps stored in the path set by `DEFAULT_INVENTORY_PLUGIN_PATH`, you could reference it as follows:

```
[inventory]
enable_plugins = host_list, script, auto, yaml, ini, toml, my_plugin
```

If you use a plugin that supports a YAML configuration source, make sure that the name matches the name provided in the `plugin` entry of the inventory source file. For other plugins you must either save it in one of the directory sources configured in `ansible.cfg <ansible_configuration_settings>` and enable it or add to a collection and then reference it by FQCN.

Using inventory plugins

To use an inventory plugin, you must provide an inventory source. Most of the time this is a file containing host information or a YAML configuration file with options for the plugin. You can use the `-i` flag to provide inventory sources or configure a default inventory path.

```
ansible hostname -i inventory_source -m ansible.builtin.ping
```

To start using an inventory plugin with a YAML configuration source, create a file with the accepted file name schema documented for the plugin in question, then add `plugin: plugin_name`. Use the fully qualified name if the plugin is in a collection.

> [!NOTE] Inventory plugins have required name patterns to which they must conform, for example:
>
> An inventory that includes the `kubevirt.core.kubevirt` inventory plugin must use the `*.kubevirt.yml` filename pattern. An inventory that includes the `servicenow.servicenow.now` inventory plugin must use the `*.servicenow.yml` filename pattern.

```
# demo.aws_ec2.yml
plugin: amazon.aws.aws_ec2
```

Each plugin should document any naming restrictions. In addition, the YAML config file must end with the extension `yml` or `yaml` to be enabled by default with the `auto` plugin (otherwise, see the section above on enabling plugins).

After providing any required options, you can view the populated inventory with `ansible-inventory -i demo.aws_ec2.yml --graph`:

```
@all:
  |--@aws_ec2:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
  |--@ungrouped:
```

If you are using an inventory plugin in a playbook-adjacent collection and want to test your setup with `ansible-inventory`, use the `--playbook-dir` flag.

Your inventory source might be a directory of inventory configuration files. The constructed inventory plugin only operates on those hosts already in inventory, so you may want the constructed inventory configuration parsed at a particular point (such as last). Ansible parses the

directory recursively, alphabetically. You cannot configure the parsing approach, so name your files to make it work predictably. Inventory plugins that extend constructed features directly can work around that restriction by adding constructed options in addition to the inventory plugin options. Otherwise, you can use `-i` with multiple sources to impose a specific order, for example `-i demo.aws_ec2.yml -i clouds.yml -i constructed.yml`.

You can create dynamic groups using host variables with the constructed `keyed_groups` option. The option `groups` can also be used to create groups and `compose` creates and modifies host variables. Here is an aws_ec2 example utilizing constructed features:

```
# demo.aws_ec2.yml
plugin: amazon.aws.aws_ec2
regions:
  - us-east-1
  - us-east-2
keyed_groups:
  # add hosts to tag_Name_value groups for each aws_ec2 host's tags.Name variable
  - key: tags.Name
    prefix: tag_Name_
    separator: ""
  # If you have a tag called "Role" which has the value "Webserver", this will add the group
  # role_Webserver and add any hosts that have that tag assigned to it.
  - key: tags.Role
    prefix: role
groups:
  # add hosts to the group development if any of the dictionary's keys or values is the word
'devel'
  development: "'devel' in (tags|list)"
  # add hosts to the "private_only" group if the host doesn't have a public IP associated to
it
  private_only: "public_ip_address is not defined"
compose:
  # use a private address where a public one isn't assigned
  ansible_host: public_ip_address|default(private_ip_address)
  # alternatively, set the ansible_host variable to connect with the private IP address
without changing the hostname
  # ansible_host: private_ip_address
  # if you *must* set a string here (perhaps to identify the inventory source if you have
multiple
  # accounts you want to use as sources), you need to wrap this in two sets of quotes, either
' then "
  # or " then '
```

```
  some_inventory_wide_string: '"Yes, you need both types of quotes here"'
```

Now the output of `ansible-inventory -i demo.aws_ec2.yml --graph`:

```
@all:
  |--@aws_ec2:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
  |  |--...
  |--@development:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
  |--@role_Webserver
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |--@tag_Name_ECS_Instance:
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
  |--@tag_Name_Test_Server:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |--@ungrouped
```

If a host does not have the variables in the configuration above (in other words, `tags.Name`, `tags`, `private_ip_address`), the host will not be added to groups other than those that the inventory plugin creates and the `ansible_host` host variable will not be modified.

Inventory plugins that support caching can use the general settings for the fact cache defined in the `ansible.cfg` file's `[defaults]` section or define inventory-specific settings in the `[inventory]` section. Individual plugins can define plugin-specific cache settings in their config file:

```
# demo.aws_ec2.yml
plugin: amazon.aws.aws_ec2
cache: true
cache_plugin: ansible.builtin.jsonfile
cache_timeout: 7200
cache_connection: /tmp/aws_inventory
cache_prefix: aws_ec2
```

Here is an example of setting inventory caching with some fact caching defaults for the cache plugin used and the timeout in an `ansible.cfg` file:

```
[defaults]
fact_caching = ansible.builtin.jsonfile
fact_caching_connection = /tmp/ansible_facts
```

```
cache_timeout = 3600

[inventory]
cache = yes
cache_connection = /tmp/ansible_inventory
```

Plugin list

You can use `ansible-doc -t inventory -l` to see the list of available plugins. Use `ansible-doc -t inventory <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`callback_plugins`
Callback plugins

`connection_plugins`
Connection plugins

`filter_plugins`
Filter plugins

`test_plugins`
Test plugins

`lookup_plugins`
Lookup plugins

`vars_plugins`
Vars plugins

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

Lookup plugins

Lookup plugins are an Ansible-specific extension to the Jinja2 templating language. You can use lookup plugins to access data from outside sources (files, databases, key/value stores, APIs, and other services) within your playbooks. Like all `templating <playbooks_templating>`, lookups execute and are evaluated on the Ansible control machine. Ansible makes the data returned by a lookup plugin available using the standard templating system. You can use lookup plugins to load variables or templates with information from external sources. You can `create custom lookup plugins <developing_lookup_plugins>`.

> [!NOTE] - Lookups are executed with a working directory relative to the role or play, as opposed to local tasks, which are executed relative to the executed script. - Pass `wantlist=True` to lookups to use in Jinja2 template "for" loops. - By default, lookup return values are marked as unsafe for security reasons. If you trust the outside source for your lookup accesses, pass `allow_unsafe=True` to allow Jinja2 templates to evaluate lookup values.

> [!WARNING] - Some lookups pass arguments to a shell. When using variables from a remote/untrusted source, use the |quote filter to ensure safe usage.

Enabling lookup plugins

Ansible enables all lookup plugins it can find. You can activate a custom lookup by either dropping it into a `lookup_plugins` directory adjacent to your play, inside the `plugins/lookup/` directory of a collection you have installed, inside a standalone role, or in one of the lookup directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

Using lookup plugins

You can use lookup plugins anywhere you can use templating in Ansible: in a play, in variables file, or a Jinja2 template for the `template <template_module>` module. For more information on using lookup plugins, see `playbooks_lookups`.

```
vars:
  file_contents: "{{ lookup('file', 'path/to/file.txt') }}"
```

Lookups are an integral part of loops. Wherever you see `with_`, the part after the underscore is the name of a lookup. For this reason, lookups are expected to output lists; for example, `with_items` uses the `items <items_lookup>` lookup:

```
tasks:
  - name: count to 3
    debug: msg={{ item }}
    with_items: [1, 2, 3]
```

You can combine lookups with `filters <playbooks_filters>`, `tests <playbooks_tests>` and even each other to do some complex data generation and manipulation. For example:

```
tasks:
  - name: Complicated chained lookups and filters
    debug: msg="find the answer here:\n{{ lookup('url', 'https://google.com/search/?q=' +
item|urlencode)|join(' ') }}"
    with_nested:
      - "{{ lookup('consul_kv', 'bcs/' + lookup('file', '/the/question') + ', host=localhost,
port=2000')|shuffle }}"
      - "{{ lookup('sequence', 'end=42 start=2 step=2')|map('log', 4)|list) }}"
      - ['a', 'c', 'd', 'c']
```

2.6

You can control how errors behave in all lookup plugins by setting `errors` to `ignore`, `warn`, or `strict`. The default setting is `strict`, which causes the task to fail if the lookup returns an error. For example:

To ignore lookup errors:

```
- name: if this file does not exist, I do not care .. file plugin itself warns anyway ...
  debug: msg="{{ lookup('file', '/nosuchfile', errors='ignore') }}"
```

```
[WARNING]: Unable to find '/nosuchfile' in expected paths (use -vvvvv to see paths)


ok: [localhost] => {
    "msg": ""
}
```

To get a warning instead of a failure:

```
- name: if this file does not exist, let me know, but continue
  debug: msg="{{ lookup('file', '/nosuchfile', errors='warn') }}"
```

```
[WARNING]: Unable to find '/nosuchfile' in expected paths (use -vvvvv to see paths)

[WARNING]: An unhandled exception occurred while running the lookup plugin 'file'. Error was a
<class 'ansible.errors.AnsibleError'>, original message: could not locate file in lookup:
/nosuchfile

ok: [localhost] => {
    "msg": ""
}
```

To get a fatal error (the default):

```
- name: if this file does not exist, FAIL (this is the default)
  debug: msg="{{ lookup('file', '/nosuchfile', errors='strict') }}"
```

```
[WARNING]: Unable to find '/nosuchfile' in expected paths (use -vvvvv to see paths)

fatal: [localhost]: FAILED! => {"msg": "An unhandled exception occurred while running the
lookup plugin 'file'. Error was a <class 'ansible.errors.AnsibleError'>, original message:
could not locate file in lookup: /nosuchfile"}
```

Forcing lookups to return lists: `query` and `wantlist=True`

2.5

In Ansible 2.5, a new Jinja2 function called `query` was added for invoking lookup plugins. The difference between `lookup` and `query` is largely that `query` will always return a list. The default behavior of `lookup` is to return a string of comma-separated values. `lookup` can be explicitly configured to return a list using `wantlist=True`.

This feature provides an easier and more consistent interface for interacting with the new `loop` keyword while maintaining backward compatibility with other uses of `lookup`.

The following examples are equivalent:

```
lookup('dict', dict_variable, wantlist=True)


query('dict', dict_variable)
```

As demonstrated above, the behavior of `wantlist=True` is implicit when using `query`.

Additionally, `q` was introduced as a short form of `query`:

```
q('dict', dict_variable)
```

Plugin list

You can use `ansible-doc -t lookup -l` to see the list of available plugins. Use `ansible-doc -t lookup <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`inventory_plugins`
Ansible inventory plugins

`callback_plugins`
Ansible callback plugins

`filter_plugins`
Jinja2 filter plugins

`test_plugins`
Jinja2 test plugins

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

Modules

Modules are the main building blocks of Ansible playbooks. Although we do not generally speak of "module plugins", a module is a type of plugin. For a developer-focused description of the differences between modules and other plugins, see `modules_vs_plugins`.

Enabling modules

You can enable a custom module by dropping it into one of these locations:

- any directory added to the `ANSIBLE_LIBRARY` environment variable (`$ANSIBLE_LIBRARY` takes a colon-separated list like `$PATH`)
- `~/.ansible/plugins/modules/`
- `/usr/share/ansible/plugins/modules/`

For more information on using local custom modules, see `local_modules`.

Using modules

For information on using modules in ad hoc tasks, see `intro_adhoc`. For information on using modules in playbooks, see `playbooks_intro`.

`about_playbooks`
An introduction to playbooks

`developing_modules_general`
An introduction to creating Ansible modules

`developing_collections`
A guide to creating Ansible collections

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

Module utilities

Module utilities contain shared code used by multiple plugins. You can write `custom module utilities <developing_module_utilities>`.

Enabling module utilities

You can add a custom module utility by dropping it into a `module_utils` directory adjacent to your collection or role, just like any other plugin.

Using module utilities

For information on using module utilities, see `developing_module_utilities`.

`developing_modules_general`
An introduction to creating Ansible modules

`developing_collections`
A guide to creating Ansible collections

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

Netconf plugins

Netconf plugins are abstractions over the Netconf interface to network devices. They provide a standard interface for Ansible to execute tasks on those network devices.

These plugins generally correspond one-to-one to network device platforms. Ansible loads the appropriate netconf plugin automatically based on the `ansible_network_os` variable. If the platform supports standard Netconf implementation as defined in the Netconf RFC specification, Ansible loads the `default` netconf plugin. If the platform supports propriety Netconf RPCs, Ansible loads the platform-specific netconf plugin.

Adding netconf plugins

You can extend Ansible to support other network devices by dropping a custom plugin into the `netconf_plugins` directory.

Using netconf plugins

The netconf plugin to use is determined automatically from the `ansible_network_os` variable. There should be no reason to override this functionality.

Most netconf plugins can operate without configuration. A few have additional options that can be set to affect how tasks are translated into netconf commands. A ncclient device-specific handler name can be set in the netconf plugin or else the value of `default` is used as per ncclient device handler.

Plugins are self-documenting. Each plugin should document its configuration options.

Listing netconf plugins

These plugins have migrated to collections on Ansible Galaxy. If you installed Ansible version 2.10 or later using `pip`, you have access to several netconf plugins. You can use `ansible-doc -t netconf -l` to see the list of available plugins. Use `ansible-doc -t netconf <plugin name>` to see plugin-specific documentation and examples.

`Ansible for Network Automation<network_guide>`
An overview of using Ansible to automate networking devices.

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

Working with plugins

Plugins are pieces of code that augment Ansible's core functionality. Ansible uses a plugin architecture to enable a rich, flexible and expandable feature set.

Ansible ships with a number of handy plugins, and you can easily write your own.

This section covers the various types of plugins that are included with Ansible:

action become cache callback cliconf connection docs_fragment filter httpapi inventory lookup module module_util netconf shell strategy terminal test vars

`plugin_filtering_config`
Controlling access to modules

`ansible_configuration_settings`
Ansible configuration documentation and settings

`command_line_tools`
Ansible tools, description and options

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

Shell plugins

Shell plugins work to ensure that the basic commands Ansible runs are properly formatted to work with the target machine and allow the user to configure certain behaviors related to how Ansible executes tasks.

Enabling shell plugins

You can add a custom shell plugin by dropping it into a `shell_plugins` directory adjacent to your play, inside a role, or by putting it in one of the shell plugin directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

> [!WARNING] You should not alter which plugin is used unless you have a setup in which the default `/bin/sh` is not a POSIX-compatible shell or is not available for execution.

Using shell plugins

In addition to the default configuration settings in `ansible_configuration_settings`, you can use the connection variable `ansible_shell_type <ansible_shell_type>` to select the plugin to use. In this case, you will also want to update the `ansible_shell_executable <ansible_shell_executable>` to match.

Plugin list

You can further control the settings for each plugin with other configuration options detailed in the plugin themselves. You can use `ansible-doc -t shell -l` to see the list of available plugins. Use `ansible-doc -t shell <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`inventory_plugins`
Inventory plugins

`callback_plugins`
Callback plugins

`filter_plugins`
Filter plugins

`test_plugins`
Test plugins

`lookup_plugins`
Lookup plugins

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

Strategy plugins

Strategy plugins control the flow of play execution by handling task and host scheduling. For more information on using strategy plugins and other ways to control execution order, see `playbooks_strategies`.

Enabling strategy plugins

All strategy plugins shipped with Ansible are enabled by default. You can enable a custom strategy plugin by putting it in one of the lookup directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

Using strategy plugins

Only one strategy plugin can be used in a play, but you can use different ones for each play in a playbook or ansible run. By default, Ansible uses the `linear <linear_strategy>` plugin. You can change this default in Ansible `configuration <ansible_configuration_settings>` using an environment variable:

```
export ANSIBLE_STRATEGY=free
```

or in the `ansible.cfg` file:

```
[defaults]
strategy=linear
```

You can also specify the strategy plugin in the play with the `strategy keyword <playbook_keywords>` in a play:

```
- hosts: all
  strategy: debug
  tasks:
    - copy:
        src: myhosts
        dest: /etc/hosts
      notify: restart_tomcat
```

```
    - package:
        name: tomcat
        state: present

  handlers:
    - name: restart_tomcat
      service:
        name: tomcat
        state: restarted
```

Plugin list

You can use `ansible-doc -t strategy -l` to see the list of available plugins. Use `ansible-doc -t strategy <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`inventory_plugins`
Inventory plugins

`callback_plugins`
Callback plugins

`filter_plugins`
Filter plugins

`test_plugins`
Test plugins

`lookup_plugins`
Lookup plugins

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

Terminal plugins

Terminal plugins contain information on how to ensure that a particular network device's SSH shell is properly initialized to be used with Ansible. This typically includes disabling automatic paging, detecting errors in output, and enabling privileged mode if the device supports and requires it.

These plugins correspond one-to-one to network device platforms. Ansible loads the appropriate terminal plugin automatically based on the `ansible_network_os` variable.

Adding terminal plugins

You can extend Ansible to support other network devices by dropping a custom plugin into the `terminal_plugins` directory.

Using terminal plugins

Ansible determines which terminal plugin to use automatically from the `ansible_network_os` variable. There should be no reason to override this functionality.

Terminal plugins operate without configuration. All options to control the terminal are exposed in the `network_cli` connection plugin.

Plugins are self-documenting. Each plugin should document its configuration options.

`Ansible for Network Automation<network_guide>`
An overview of using Ansible to automate networking devices.

`connection_plugins`
Connection plugins

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

Test plugins

Test plugins evaluate template expressions and return True or False. With test plugins you can create `conditionals <playbooks_conditionals>` to implement the logic of your tasks, blocks, plays, playbooks, and roles. Ansible uses the standard tests shipped as part of Jinja and adds some specialized test plugins. You can `create custom Ansible test plugins <developing_test_plugins>`.

Enabling test plugins

You can add a custom test plugin by dropping it into a `test_plugins` directory adjacent to your play, inside a role, or by putting it in one of the test plugin directory sources configured in `ansible.cfg <ansible_configuration_settings>`.

Using test plugins

You can use tests anywhere you can use templating in Ansible: in a play, in a variables file, or in a Jinja2 template for the `template <template_module>` module. For more information on using test plugins, see `playbooks_tests`.

Tests always return `True` or `False`, they are always a boolean, if you need a different return type, you should be looking at filters.

You can recognize test plugins by the use of the `is` statement in a template, they can also be used as part of the `select` family of filters.

```
vars:
  is_ready: '{{ task_result is success }}'

tasks:
- name: conditionals are always in 'template' context
  action: dostuff
  when: task_result is failed
```

Tests will always have an `_input` and this is normally what is on the left side of `is`. Tests can also take additional parameters as you would to most programming functions. These parameters can be either `positional` (passed in order) or `named` (passed as key=value pairs). When passing both types, positional arguments should go first.

```
tasks:
- name: pass a positional parameter to match test
  action: dostuff
  when: myurl is match("https://example.com/users/.*/resources")

- name: pass named parameter to truthy test
  action: dostuff
  when: myvariable is truthy(convert_bool=True)

- name: pass both types to 'version' test
  action: dostuff
  when: sample_semver_var is version('2.0.0-rc.1+build.123', 'lt', version_type='semver')
```

Using test plugins with lists

As mentioned above, one way to use tests is with the `select` family of filters (`select`, `reject`, `selectattr`, `rejectattr`).

```
# give me only defined variables from a list of variables, using 'defined' test
good_vars: "{{ all_vars|select('defined') }}"

# this uses the 'equalto' test to filter out non 'fixed' type of addresses from a list
only_fixed_addresses:  "{{ all_addresses|selectattr('type', 'equalto', 'fixed') }}"

# this does the opposite of the previous one
only_fixed_addresses:  "{{ all_addresses|rejectattr('type', 'equalto', 'fixed') }}"
```

Plugin list

You can use `ansible-doc -t test -l` to see the list of available plugins. Use `ansible-doc -t test <plugin name>` to see plugin-specific documentation and examples.

`about_playbooks`
An introduction to playbooks

`playbooks_tests`
Using tests

`playbooks_conditionals`
Using conditional statements

`filter_plugins`
Filter plugins

`playbooks_tests`
Using tests

`lookup_plugins`
Lookup plugins

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

Vars plugins

Vars plugins inject additional variable data into Ansible runs that did not come from an inventory source, playbook, or command line. Playbook constructs like 'host_vars' and 'group_vars' work using vars plugins. For more details about variables in Ansible, see `playbooks_variables`.

Vars plugins were partially implemented in Ansible 2.0 and rewritten to be fully implemented starting with Ansible 2.4.

The `host_group_vars <host_group_vars_vars>` plugin shipped with Ansible enables reading variables from `host_variables` and `group_variables`.

Enabling vars plugins

You can activate a custom vars plugin by either dropping it into a `vars_plugins` directory adjacent to your play, inside a role, or by putting it in one of the directory sources configured in `ansible.cfg <ansible_configuration_settings>`. For a vars plugin to run during inventory build you cannot enable it in a play or role as these are not loaded until later. If they are only going to run at task execution, there are no limitations on where they are provided.

Most vars plugins are disabled by default. To enable a vars plugin, set `vars_plugins_enabled` in the `defaults` section of `ansible.cfg <ansible_configuration_settings>` or set the `ANSIBLE_VARS_ENABLED` environment variable to the list of vars plugins you want to execute. By default, the `host_group_vars <host_group_vars_vars>` plugin shipped with Ansible is enabled.

Starting in Ansible 2.10, you can use vars plugins in collections. All vars plugins in collections must be explicitly enabled and must use the fully qualified collection name in the format `namespace.collection_name.vars_plugin_name`.

```
[defaults]
vars_plugins_enabled = host_group_vars,namespace.collection_name.vars_plugin_name
```

Using vars plugins

By default, vars plugins are used on demand automatically after they are enabled.

Starting in Ansible 2.10, vars plugins can be made to run at specific times. ansible-inventory does not use these settings, and always loads vars plugins.

The global setting `RUN_VARS_PLUGINS` can be set in `ansible.cfg` using `run_vars_plugins` in the `defaults` section or by the `ANSIBLE_RUN_VARS_PLUGINS` environment variable. The default option, `demand`, runs any enabled vars plugins relative to inventory sources whenever variables are demanded by tasks. You can use the option `start` to run any enabled vars plugins relative to inventory sources after importing that inventory source instead.

You can also control vars plugin execution on a per-plugin basis for vars plugins that support the `stage` option. To run the `host_group_vars <host_group_vars_vars>` plugin after importing inventory you can add the following to `ansible.cfg <ansible_configuration_settings>`:

```
[vars_host_group_vars]
stage = inventory
```

Plugin list

You can use `ansible-doc -t vars -l` to see the list of available vars plugins. Use `ansible-doc -t vars <plugin name>` to see plugin-specific documentation and examples.

`cache_plugins`
Cache plugins

`lookup_plugins`
Lookup plugins

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