

01AnsibleUserGuide-GettingStarted	3
02AnsibleUserGuide-Concepts	15
03AnsibleUserGuide-Concepts-UG	17
04AnsibleUserGuide-Concepts-Commands	24
05AnsibleUserGuide-Concepts-01Commands	30
05AnsibleUserGuide-Concepts-01PlayBooks	35
05AnsibleUserGuide-Concepts-02Commands	37
05AnsibleUserGuide-Concepts-02PlayBooks	41
05AnsibleUserGuide-Concepts-03Commands	47
05AnsibleUserGuide-Concepts-03PlayBooks	53
05AnsibleUserGuide-Concepts-04Commands	59
05AnsibleUserGuide-Concepts-04PlayBooks	62
05AnsibleUserGuide-Concepts-05Commands	80
05AnsibleUserGuide-Concepts-05PlayBooks	93
05AnsibleUserGuide-Concepts-06Commands	109
05AnsibleUserGuide-Concepts-06PlayBooks	113
05AnsibleUserGuide-Concepts-07Commands	118
05AnsibleUserGuide-Concepts-07PlayBooks	124
05AnsibleUserGuide-Concepts-08Commands	137
05AnsibleUserGuide-Concepts-08PlayBooks	143
05AnsibleUserGuide-Concepts-09Commands	148
05AnsibleUserGuide-Concepts-09PlayBooks	154
05AnsibleUserGuide-Concepts-10PlayBooks	159
05AnsibleUserGuide-Concepts-11PlayBooks	166
05AnsibleUserGuide-Concepts-12PlayBooks	170
05AnsibleUserGuide-Concepts-13PlayBooks	177
05AnsibleUserGuide-Concepts-14PlayBooks	193
05AnsibleUserGuide-Concepts-15PlayBooks	200
05AnsibleUserGuide-Concepts-16PlayBooks	212
05AnsibleUserGuide-Concepts-17PlayBooks	231

05AnsibleUserGuide-Concepts-18PlayBooks	239
05AnsibleUserGuide-Concepts-19PlayBooks	245
06AnsibleUserGuide-Concepts-01Variables	249
06AnsibleUserGuide-Concepts-02Variables	264
06AnsibleUserGuide-Concepts-03Variables	269
07AnsibleUserGuide-Concepts-01Vaults	285
07AnsibleUserGuide-Concepts-02Vaults	304
08AnsibleUserGuide-Concepts-Modules	308
09AnsibleUserGuide-Concepts-01Tasks	312
09AnsibleUserGuide-Concepts-02Tasks	315
09AnsibleUserGuide-Concepts-03Tasks	317
10AnsibleUserGuide-Concepts-Execution	325

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

User Guide

📌 Note

Making Open Source More Inclusive

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. We ask that you open an issue or pull request if you come upon a term that we have missed. For more details, see [our CTO Chris Wright's message](https://www.redhat.com/en/blog/making-open-source-more-inclusive-eradicating-problematic-language) (<https://www.redhat.com/en/blog/making-open-source-more-inclusive-eradicating-problematic-language>).

Welcome to the Ansible User Guide! This guide covers how to work with Ansible, including using the command line, working with inventory, interacting with data, writing tasks, plays, and playbooks; executing playbooks, and reference materials. This page outlines the most common situations and questions that bring readers to this section. If you prefer a traditional table of contents, you can find one at the bottom of the page.

Getting started

- I'd like an overview of how Ansible works. Where can I find:
 - a [quick video overview](#) ([quickstart.html#quickstart-guide](#)).
 - a [text introduction](#) ([intro_getting_started.html#intro-getting-started](#)).
- I'm ready to learn about Ansible. What [Ansible concepts](#) ([basic_concepts.html#basic-concepts](#)) do I need to learn?
- I want to use Ansible without writing a playbook. How do I use [ad hoc commands](#) ([intro_adhoc.html#intro-adhoc](#))?

Writing tasks, plays, and playbooks

- I'm writing my first playbook. What should I [know before I begin](#) ([playbooks_best_practices.html#playbooks-tips-and-tricks](#))?
- I have a specific use case for a task or play:

- Executing tasks with elevated privileges or as a different user with [become](#) ([playbooks_become.html#playbooks-become](#)).
- Repeating a task once for each item in a list with [loops](#) ([playbooks_loops.html#playbooks-loops](#)).
- Executing tasks on a different machine with [delegation](#) ([playbooks_delegation.html#playbooks-delegation](#)).
- Running tasks only when certain conditions apply with [conditionals](#) ([playbooks_conditionals.html#playbooks-conditionals](#)) and evaluating conditions with [tests](#) ([playbooks_tests.html#playbooks-tests](#)).
- Grouping a set of tasks together with [blocks](#) ([playbooks_blocks.html#playbooks-blocks](#)).
- Running tasks only when something has changed with [handlers](#) ([playbooks_handlers.html#handlers](#)).
- Changing the way Ansible [handles failures](#) ([playbooks_error_handling.html#playbooks-error-handling](#)).
- Setting remote [environment values](#) ([playbooks_environment.html#playbooks-environment](#)).
- I want to take advantage of the power of re-usable Ansible artifacts. How do I create re-usable [files](#) ([playbooks_reuse.html#playbooks-reuse](#)) and [roles](#) ([playbooks_reuse_roles.html#playbooks-reuse-roles](#))?
- I need to incorporate one file or playbook inside another. What is the difference between [including and importing](#) ([playbooks_reuse.html#dynamic-vs-static](#))?
- I want to run selected parts of my playbook. How do I add and use [tags](#) ([playbooks_tags.html#tags](#))?

Working with inventory

- I have a list of servers and devices I want to automate. How do I create [inventory](#) ([intro_inventory.html#intro-inventory](#)) to track them?
- I use cloud services and constantly have servers and devices starting and stopping. How do I track them using [dynamic inventory](#) ([intro_dynamic_inventory.html#intro-dynamic-inventory](#))?
- I want to automate specific sub-sets of my inventory. How do I use [patterns](#) ([intro_patterns.html#intro-patterns](#))?

Interacting with data

- I want to use a single playbook against multiple systems with different attributes. How do I use [variables](#) ([playbooks_variables.html#playbooks-variables](#)) to handle the differences?
- I want to retrieve data about my systems. How do I access [Ansible facts](#) ([playbooks_vars_facts.html#vars-and-facts](#))?
- I need to access sensitive data like passwords with Ansible. How can I protect that data with [Ansible vault](#) ([vault.html#vault](#))?

- I want to change the data I have, so I can use it in a task. How do I use [filters](#) ([playbooks_filters.html#playbooks-filters](#)) to transform my data?
- I need to retrieve data from an external datastore. How do I use [lookups](#) ([playbooks_lookups.html#playbooks-lookups](#)) to access databases and APIs?
- I want to ask playbook users to supply data. How do I get user input with [prompts](#) ([playbooks_prompts.html#playbooks-prompts](#))?
- I use certain modules frequently. How do I streamline my inventory and playbooks by [setting default values for module parameters](#) ([playbooks_module_defaults.html#module-defaults](#))?

Executing playbooks

Once your playbook is ready to run, you may need to use these topics:

- Executing “dry run” playbooks with [check mode and diff](#) ([playbooks_checkmode.html#check-mode-dry](#)).
- Running playbooks while troubleshooting with [start and step](#) ([playbooks_startnstep.html#playbooks-start-and-step](#)).
- Correcting tasks during execution with the [Ansible debugger](#) ([playbooks_debugger.html#playbook-debugger](#)).
- Controlling how my playbook executes with [strategies and more](#) ([playbooks_strategies.html#playbooks-strategies](#)).
- Running tasks, plays, and playbooks [asynchronously](#) ([playbooks_async.html#playbooks-async](#)).

Advanced features and reference

- Using [advanced syntax](#) ([playbooks_advanced_syntax.html#playbooks-advanced-syntax](#)).
- Manipulating [complex data](#) ([complex_data_manipulation.html#complex-data-manipulation](#)).
- Using [plugins](#) ([./plugins/plugins.html#plugins-lookup](#)).
- Using [playbook keywords](#) ([./reference_appendices/playbooks_keywords.html#playbook-keywords](#)).
- Using [command-line tools](#) ([command_line_tools.html#command-line-tools](#)).
- Rejecting [specific modules](#) ([plugin_filtering_config.html#plugin-filtering-config](#)).
- Module [maintenance](#) ([modules_support.html#modules-support](#)).

Traditional Table of Contents

If you prefer to read the entire User Guide, here's a list of the pages in order:

- [Ansible Quickstart Guide](#) ([quickstart.html](#)).
- [Ansible concepts](#) ([basic_concepts.html](#)).

- [Control node \(basic_concepts.html#control-node\)](#)
- [Managed nodes \(basic_concepts.html#managed-nodes\)](#)
- [Inventory \(basic_concepts.html#inventory\)](#)
- [Collections \(basic_concepts.html#collections\)](#)
- [Modules \(basic_concepts.html#modules\)](#)
- [Tasks \(basic_concepts.html#tasks\)](#)
- [Playbooks \(basic_concepts.html#playbooks\)](#)
- [Getting Started \(intro_getting_started.html\)](#)
 - [Selecting machines from inventory \(intro_getting_started.html#selecting-machines-from-inventory\)](#)
 - [Connecting to remote nodes \(intro_getting_started.html#connecting-to-remote-nodes\)](#)
 - [Copying and executing modules \(intro_getting_started.html#copying-and-executing-modules\)](#)
 - [Resources \(intro_getting_started.html#resources\)](#)
 - [Next steps \(intro_getting_started.html#next-steps\)](#)
- [Introduction to ad hoc commands \(intro_adhoc.html\)](#)
 - [Why use ad hoc commands? \(intro_adhoc.html#why-use-ad-hoc-commands\)](#)
 - [Use cases for ad hoc tasks \(intro_adhoc.html#use-cases-for-ad-hoc-tasks\)](#)
- [Working with playbooks \(playbooks.html\)](#)
 - [Templating \(Jinja2\) \(playbooks_templating.html\)](#)
 - [Advanced playbooks features \(playbooks_special_topics.html\)](#)
 - [Playbook Example: Continuous Delivery and Rolling Upgrades \(guide_rolling_upgrade.html\)](#)
- [Intro to playbooks \(playbooks_intro.html\)](#)
 - [Playbook syntax \(playbooks_intro.html#playbook-syntax\)](#)
 - [Playbook execution \(playbooks_intro.html#playbook-execution\)](#)
 - [Ansible-Pull \(playbooks_intro.html#ansible-pull\)](#)
 - [Verifying playbooks \(playbooks_intro.html#verifying-playbooks\)](#)
- [Tips and tricks \(playbooks_best_practices.html\)](#)
 - [General tips \(playbooks_best_practices.html#general-tips\)](#)
 - [Playbook tips \(playbooks_best_practices.html#playbook-tips\)](#)
 - [Inventory tips \(playbooks_best_practices.html#inventory-tips\)](#)
 - [Execution tricks \(playbooks_best_practices.html#execution-tricks\)](#)
- [Understanding privilege escalation: become \(become.html\)](#)
 - [Using become \(become.html#using-become\)](#)
 - [Risks and limitations of become \(become.html#risks-and-limitations-of-become\)](#)
 - [Become and network automation \(become.html#become-and-network-automation\)](#)
 - [Become and Windows \(become.html#become-and-windows\)](#)

- [Loops \(playbooks_loops.html\)](#)
 - [Comparing `loop` and `with *` \(playbooks_loops.html#comparing-loop-and-with\)](#)
 - [Standard loops \(playbooks_loops.html#standard-loops\)](#)
 - [Registering variables with a loop \(playbooks_loops.html#registering-variables-with-a-loop\)](#)
 - [Complex loops \(playbooks_loops.html#complex-loops\)](#)
 - [Ensuring list input for `loop` : using `query` rather than `lookup` \(playbooks_loops.html#ensuring-list-input-for-loop-using-query-rather-than-lookup\)](#)
 - [Adding controls to loops \(playbooks_loops.html#adding-controls-to-loops\)](#)
 - [Migrating from `with X` to `loop` \(playbooks_loops.html#migrating-from-with-x-to-loop\)](#)
- [Controlling where tasks run: delegation and local actions \(playbooks_delegation.html\)](#)
 - [Tasks that cannot be delegated \(playbooks_delegation.html#tasks-that-cannot-be-delegated\)](#)
 - [Delegating tasks \(playbooks_delegation.html#delegating-tasks\)](#)
 - [Delegation and parallel execution \(playbooks_delegation.html#delegation-and-parallel-execution\)](#)
 - [Delegating facts \(playbooks_delegation.html#delegating-facts\)](#)
 - [Local playbooks \(playbooks_delegation.html#local-playbooks\)](#)
- [Conditionals \(playbooks_conditionals.html\)](#)
 - [Basic conditionals with `when` \(playbooks_conditionals.html#basic-conditionals-with-when\)](#)
 - [Commonly-used facts \(playbooks_conditionals.html#commonly-used-facts\)](#)
- [Tests \(playbooks_tests.html\)](#)
 - [Test syntax \(playbooks_tests.html#test-syntax\)](#)
 - [Testing strings \(playbooks_tests.html#testing-strings\)](#)
 - [Vault \(playbooks_tests.html#vault\)](#)
 - [Testing truthiness \(playbooks_tests.html#testing-truthiness\)](#)
 - [Comparing versions \(playbooks_tests.html#comparing-versions\)](#)
 - [Set theory tests \(playbooks_tests.html#set-theory-tests\)](#)
 - [Testing if a list contains a value \(playbooks_tests.html#testing-if-a-list-contains-a-value\)](#)
 - [Testing if a list value is True \(playbooks_tests.html#testing-if-a-list-value-is-true\)](#)
 - [Testing paths \(playbooks_tests.html#testing-paths\)](#)
 - [Testing size formats \(playbooks_tests.html#testing-size-formats\)](#)
 - [Testing task results \(playbooks_tests.html#testing-task-results\)](#)
- [Blocks \(playbooks_blocks.html\)](#)
 - [Grouping tasks with blocks \(playbooks_blocks.html#grouping-tasks-with-blocks\)](#)
 - [Handling errors with blocks \(playbooks_blocks.html#handling-errors-with-blocks\)](#)
- [Handlers: running operations on change \(playbooks_handlers.html\)](#)

- [Handler example \(playbooks_handlers.html#handler-example\)](#)
- [Controlling when handlers run \(playbooks_handlers.html#controlling-when-handlers-run\)](#)
- [Using variables with handlers \(playbooks_handlers.html#using-variables-with-handlers\)](#)
- [Error handling in playbooks \(playbooks_error_handling.html\)](#)
 - [Ignoring failed commands \(playbooks_error_handling.html#ignoring-failed-commands\)](#)
 - [Ignoring unreachable host errors \(playbooks_error_handling.html#ignoring-unreachable-host-errors\)](#)
 - [Resetting unreachable hosts \(playbooks_error_handling.html#resetting-unreachable-hosts\)](#)
 - [Handlers and failure \(playbooks_error_handling.html#handlers-and-failure\)](#)
 - [Defining failure \(playbooks_error_handling.html#defining-failure\)](#)
 - [Defining “changed” \(playbooks_error_handling.html#defining-changed\)](#)
 - [Ensuring success for command and shell \(playbooks_error_handling.html#ensuring-success-for-command-and-shell\)](#)
 - [Aborting a play on all hosts \(playbooks_error_handling.html#aborting-a-play-on-all-hosts\)](#)
 - [Controlling errors in blocks \(playbooks_error_handling.html#controlling-errors-in-blocks\)](#)
- [Setting the remote environment \(playbooks_environment.html\)](#)
 - [Setting the remote environment in a task \(playbooks_environment.html#setting-the-remote-environment-in-a-task\)](#)
- [Working with language-specific version managers \(playbooks_environment.html#working-with-language-specific-version-managers\)](#)
- [Re-using Ansible artifacts \(playbooks_reuse.html\)](#)
 - [Creating re-usable files and roles \(playbooks_reuse.html#creating-re-usable-files-and-roles\)](#)
 - [Re-using playbooks \(playbooks_reuse.html#re-using-playbooks\)](#)
 - [Re-using files and roles \(playbooks_reuse.html#re-using-files-and-roles\)](#)
 - [Re-using tasks as handlers \(playbooks_reuse.html#re-using-tasks-as-handlers\)](#)
- [Roles \(playbooks_reuse_roles.html\)](#)
 - [Role directory structure \(playbooks_reuse_roles.html#role-directory-structure\)](#)
 - [Storing and finding roles \(playbooks_reuse_roles.html#storing-and-finding-roles\)](#)
 - [Using roles \(playbooks_reuse_roles.html#using-roles\)](#)
 - [Role argument validation \(playbooks_reuse_roles.html#role-argument-validation\)](#)
 - [Running a role multiple times in one playbook \(playbooks_reuse_roles.html#running-a-role-multiple-times-in-one-playbook\)](#)
 - [Using role dependencies \(playbooks_reuse_roles.html#using-role-dependencies\)](#)
 - [Embedding modules and plugins in roles \(playbooks_reuse_roles.html#embedding-modules-and-plugins-in-roles\)](#)

- [Sharing roles: Ansible Galaxy \(playbooks_reuse_roles.html#sharing-roles-ansible-galaxy\)](#).
- [Including and importing \(playbooks_reuse_includes.html\)](#)
- [Tags \(playbooks_tags.html\)](#)
 - [Adding tags with the tags keyword \(playbooks_tags.html#adding-tags-with-the-tags-keyword\)](#)
 - [Special tags: always and never \(playbooks_tags.html#special-tags-always-and-never\)](#)
 - [Selecting or skipping tags when you run a playbook \(playbooks_tags.html#selecting-or-skipping-tags-when-you-run-a-playbook\)](#)
- [How to build your inventory \(intro_inventory.html\)](#)
 - [Inventory basics: formats, hosts, and groups \(intro_inventory.html#inventory-basics-formats-hosts-and-groups\)](#)
 - [Adding variables to inventory \(intro_inventory.html#adding-variables-to-inventory\)](#)
 - [Assigning a variable to one machine: host variables \(intro_inventory.html#assigning-a-variable-to-one-machine-host-variables\)](#)
 - [Assigning a variable to many machines: group variables \(intro_inventory.html#assigning-a-variable-to-many-machines-group-variables\)](#)
 - [Organizing host and group variables \(intro_inventory.html#organizing-host-and-group-variables\)](#)
 - [How variables are merged \(intro_inventory.html#how-variables-are-merged\)](#)
 - [Using multiple inventory sources \(intro_inventory.html#using-multiple-inventory-sources\)](#)
 - [Connecting to hosts: behavioral inventory parameters \(intro_inventory.html#connecting-to-hosts-behavioral-inventory-parameters\)](#)
 - [Inventory setup examples \(intro_inventory.html#inventory-setup-examples\)](#)
- [Working with dynamic inventory \(intro_dynamic_inventory.html\)](#)
 - [Inventory script example: Cobbler \(intro_dynamic_inventory.html#inventory-script-example-cobbler\)](#)
 - [Inventory script example: OpenStack \(intro_dynamic_inventory.html#inventory-script-example-openstack\)](#)
 - [Other inventory scripts \(intro_dynamic_inventory.html#other-inventory-scripts\)](#)
 - [Using inventory directories and multiple inventory sources \(intro_dynamic_inventory.html#using-inventory-directories-and-multiple-inventory-sources\)](#)
 - [Static groups of dynamic groups \(intro_dynamic_inventory.html#static-groups-of-dynamic-groups\)](#)
- [Patterns: targeting hosts and groups \(intro_patterns.html\)](#)
 - [Using patterns \(intro_patterns.html#using-patterns\)](#)
 - [Common patterns \(intro_patterns.html#common-patterns\)](#)
 - [Limitations of patterns \(intro_patterns.html#limitations-of-patterns\)](#)
 - [Advanced pattern options \(intro_patterns.html#advanced-pattern-options\)](#) Search this site

- [Patterns and ad-hoc commands \(intro_patterns.html#patterns-and-ad-hoc-commands\)](#)
- [Patterns and ansible-playbook flags \(intro_patterns.html#patterns-and-ansible-playbook-flags\)](#)
- [Connection methods and details \(connection_details.html\)](#)
 - [ControlPersist and paramiko \(connection_details.html#controlpersist-and-paramiko\)](#)
 - [Setting a remote user \(connection_details.html#setting-a-remote-user\)](#)
 - [Setting up SSH keys \(connection_details.html#setting-up-ssh-keys\)](#)
 - [Running against localhost \(connection_details.html#running-against-localhost\)](#)
 - [Managing host key checking \(connection_details.html#managing-host-key-checking\)](#)
 - [Other connection methods \(connection_details.html#other-connection-methods\)](#)
- [Working with command line tools \(command_line_tools.html\)](#)
 - [ansible \(../cli/ansible.html\)](#)
 - [ansible-config \(../cli/ansible-config.html\)](#)
 - [ansible-console \(../cli/ansible-console.html\)](#)
 - [ansible-doc \(../cli/ansible-doc.html\)](#)
 - [ansible-galaxy \(../cli/ansible-galaxy.html\)](#)
 - [ansible-inventory \(../cli/ansible-inventory.html\)](#)
 - [ansible-playbook \(../cli/ansible-playbook.html\)](#)
 - [ansible-pull \(../cli/ansible-pull.html\)](#)
 - [ansible-vault \(../cli/ansible-vault.html\)](#)
- [Using Variables \(playbooks_variables.html\)](#)
 - [Creating valid variable names \(playbooks_variables.html#creating-valid-variable-names\)](#)
 - [Simple variables \(playbooks_variables.html#simple-variables\)](#)
 - [When to quote variables \(a YAML gotcha\) \(playbooks_variables.html#when-to-quote-variables-a-yaml-gotcha\)](#)
 - [List variables \(playbooks_variables.html#list-variables\)](#)
 - [Dictionary variables \(playbooks_variables.html#dictionary-variables\)](#)
 - [Registering variables \(playbooks_variables.html#registering-variables\)](#)
 - [Referencing nested variables \(playbooks_variables.html#referencing-nested-variables\)](#)
 - [Transforming variables with Jinja2 filters \(playbooks_variables.html#transforming-variables-with-jinja2-filters\)](#)
 - [Where to set variables \(playbooks_variables.html#where-to-set-variables\)](#)
 - [Variable precedence: Where should I put a variable? \(playbooks_variables.html#variable-precedence-where-should-i-put-a-variable\)](#)
 - [Using advanced variable syntax \(playbooks_variables.html#using-advanced-variable-syntax\)](#)
- [Discovering variables: facts and magic variables \(playbooks_vars_facts.html\)](#)
 - [Ansible facts \(playbooks_vars_facts.html#ansible-facts\)](#)
 - [Information about Ansible: magic variables \(playbooks_vars_facts.html#information-about-ansible-magic-variables\)](#)

- [Encrypting content with Ansible Vault \(vault.html\)](#)
 - [Managing vault passwords \(vault.html#managing-vault-passwords\)](#)
 - [Encrypting content with Ansible Vault \(vault.html#id1\)](#)
 - [Using encrypted variables and files \(vault.html#using-encrypted-variables-and-files\)](#)
 - [Configuring defaults for using encrypted content \(vault.html#configuring-defaults-for-using-encrypted-content\)](#)
 - [When are encrypted files made visible? \(vault.html#when-are-encrypted-files-made-visible\)](#)
 - [Format of files encrypted with Ansible Vault \(vault.html#format-of-files-encrypted-with-ansible-vault\)](#)
- [Using filters to manipulate data \(playbooks_filters.html\)](#)
 - [Handling undefined variables \(playbooks_filters.html#handling-undefined-variables\)](#)
 - [Defining different values for true/false/null \(ternary\) \(playbooks_filters.html#defining-different-values-for-true-false-null-ternary\)](#)
 - [Managing data types \(playbooks_filters.html#managing-data-types\)](#)
 - [Formatting data: YAML and JSON \(playbooks_filters.html#formatting-data-yaml-and-json\)](#)
 - [Combining and selecting data \(playbooks_filters.html#combining-and-selecting-data\)](#)
 - [Randomizing data \(playbooks_filters.html#randomizing-data\)](#)
 - [Managing list variables \(playbooks_filters.html#managing-list-variables\)](#)
 - [Selecting from sets or lists \(set theory\) \(playbooks_filters.html#selecting-from-sets-or-lists-set-theory\)](#)
 - [Calculating numbers \(math\) \(playbooks_filters.html#calculating-numbers-math\)](#)
 - [Managing network interactions \(playbooks_filters.html#managing-network-interactions\)](#)
 - [Hashing and encrypting strings and passwords \(playbooks_filters.html#hashing-and-encrypting-strings-and-passwords\)](#)
 - [Manipulating text \(playbooks_filters.html#manipulating-text\)](#)
 - [Manipulating strings \(playbooks_filters.html#manipulating-strings\)](#)
 - [Managing UUIDs \(playbooks_filters.html#managing-uuids\)](#)
 - [Handling dates and times \(playbooks_filters.html#handling-dates-and-times\)](#)
 - [Getting Kubernetes resource names \(playbooks_filters.html#getting-kubernetes-resource-names\)](#)
- [Lookups \(playbooks_lookups.html\)](#)
 - [Using lookups in variables \(playbooks_lookups.html#using-lookups-in-variables\)](#)
- [Interactive input: prompts \(playbooks_prompts.html\)](#)
 - [Encrypting values supplied by `vars_prompt` \(playbooks_prompts.html#encrypting-values-supplied-by-vars-prompt\)](#)
 - [Allowing special characters in `vars_prompt` values \(playbooks_prompts.html#allowing-special-characters-in-vars-prompt-values\)](#)
- [Module defaults \(playbooks_module_defaults.html\)](#)

- [Module defaults groups \(playbooks_module_defaults.html#module-defaults-groups\)](#)
- [Validating tasks: check mode and diff mode \(playbooks_checkmode.html\)](#)
 - [Using check mode \(playbooks_checkmode.html#using-check-mode\)](#)
 - [Using diff mode \(playbooks_checkmode.html#using-diff-mode\)](#)
- [Executing playbooks for troubleshooting \(playbooks_startnstep.html\)](#)
 - [start-at-task \(playbooks_startnstep.html#start-at-task\)](#)
 - [Step mode \(playbooks_startnstep.html#step-mode\)](#)
- [Debugging tasks \(playbooks_debugger.html\)](#)
 - [Enabling the debugger \(playbooks_debugger.html#enabling-the-debugger\)](#)
 - [Resolving errors in the debugger \(playbooks_debugger.html#resolving-errors-in-the-debugger\)](#)
 - [Available debug commands \(playbooks_debugger.html#available-debug-commands\)](#)
 - [How the debugger interacts with the free strategy \(playbooks_debugger.html#how-the-debugger-interacts-with-the-free-strategy\)](#)
- [Controlling playbook execution: strategies and more \(playbooks_strategies.html\)](#)
 - [Selecting a strategy \(playbooks_strategies.html#selecting-a-strategy\)](#)
 - [Setting the number of forks \(playbooks_strategies.html#setting-the-number-of-forks\)](#)
 - [Using keywords to control execution \(playbooks_strategies.html#using-keywords-to-control-execution\)](#)
- [Asynchronous actions and polling \(playbooks_async.html\)](#)
 - [Asynchronous ad hoc tasks \(playbooks_async.html#asynchronous-ad-hoc-tasks\)](#)
 - [Asynchronous playbook tasks \(playbooks_async.html#asynchronous-playbook-tasks\)](#)
- [Advanced Syntax \(playbooks_advanced_syntax.html\)](#)
 - [Unsafe or raw strings \(playbooks_advanced_syntax.html#unsafe-or-raw-strings\)](#)
 - [YAML anchors and aliases: sharing variable values \(playbooks_advanced_syntax.html#yaml-anchors-and-aliases-sharing-variable-values\)](#)
- [Data manipulation \(complex_data_manipulation.html\)](#)
 - [Loops and list comprehensions \(complex_data_manipulation.html#loops-and-list-comprehensions\)](#)
 - [Complex Type transformations \(complex_data_manipulation.html#complex-type-transformations\)](#)
- [Rejecting modules \(plugin_filtering_config.html\)](#)
- [Sample Ansible setup \(sample_setup.html\)](#)
 - [Sample directory layout \(sample_setup.html#sample-directory-layout\)](#)
 - [Alternative directory layout \(sample_setup.html#alternative-directory-layout\)](#)
 - [Sample group and host variables \(sample_setup.html#sample-group-and-host-variables\)](#)

- [Sample playbooks organized by function \(sample_setup.html#sample-playbooks-organized-by-function\)](#)
- [Sample task and handler files in a function-based role \(sample_setup.html#sample-task-and-handler-files-in-a-function-based-role\)](#)
- [What the sample setup enables \(sample_setup.html#what-the-sample-setup-enables\)](#)
- [Organizing for deployment or configuration \(sample_setup.html#organizing-for-deployment-or-configuration\)](#)
- [Using local Ansible modules \(sample_setup.html#using-local-ansible-modules\)](#)
- [Working With Modules \(modules.html\)](#)
 - [Introduction to modules \(modules_intro.html\)](#)
 - [Module Maintenance & Support \(modules_support.html\)](#)
 - [Return Values \(../reference_appendices/common_return_values.html\)](#)
- [Working with plugins \(../plugins/plugins.html\)](#)
 - [Action plugins \(../plugins/action.html\)](#)
 - [Become plugins \(../plugins/become.html\)](#)
 - [Cache plugins \(../plugins/cache.html\)](#)
 - [Callback plugins \(../plugins/callback.html\)](#)
 - [Cliconf plugins \(../plugins/cliconf.html\)](#)
 - [Connection plugins \(../plugins/connection.html\)](#)
 - [Docs fragments \(../plugins/docs_fragment.html\)](#)
 - [Filter plugins \(../plugins/filter.html\)](#)
 - [Httpapi plugins \(../plugins/httpapi.html\)](#)
 - [Inventory plugins \(../plugins/inventory.html\)](#)
 - [Lookup plugins \(../plugins/lookup.html\)](#)
 - [Modules \(../plugins/module.html\)](#)
 - [Module utilities \(../plugins/module_util.html\)](#)
 - [Netconf plugins \(../plugins/netconf.html\)](#)
 - [Shell plugins \(../plugins/shell.html\)](#)
 - [Strategy plugins \(../plugins/strategy.html\)](#)
 - [Terminal plugins \(../plugins/terminal.html\)](#)
 - [Test plugins \(../plugins/test.html\)](#)
 - [Vars plugins \(../plugins/vars.html\)](#)
- [Playbook Keywords \(../reference_appendices/playbooks_keywords.html\)](#)
 - [Play \(../reference_appendices/playbooks_keywords.html#play\)](#)
 - [Role \(../reference_appendices/playbooks_keywords.html#role\)](#)
 - [Block \(../reference_appendices/playbooks_keywords.html#block\)](#)
 - [Task \(../reference_appendices/playbooks_keywords.html#task\)](#)
- [Ansible and BSD \(intro_bsd.html\)](#)
 - [Connecting to BSD nodes \(intro_bsd.html#connecting-to-bsd-nodes\)](#)
 - [Bootstrapping BSD \(intro_bsd.html#bootstrapping-bsd\)](#)
 - [Setting the Python interpreter \(intro_bsd.html#setting-the-python-interpreter\)](#)

- [Which modules are available? \(intro_bsd.html#which-modules-are-available\)](#)
- [Using BSD as the control node \(intro_bsd.html#using-bsd-as-the-control-node\)](#)
- [BSD facts \(intro_bsd.html#bsd-facts\)](#)
- [BSD efforts and contributions \(intro_bsd.html#bsd-efforts-and-contributions\)](#)
- [Windows Guides \(windows.html\)](#)
 - [Setting up a Windows Host \(windows_setup.html\)](#)
 - [Windows Remote Management \(windows_winrm.html\)](#)
 - [Using Ansible and Windows \(windows_usage.html\)](#)
 - [Desired State Configuration \(windows_dsc.html\)](#)
 - [Windows performance \(windows_performance.html\)](#)
 - [Windows Frequently Asked Questions \(windows_faq.html\)](#)
- [Using collections \(collections_using.html\)](#)
 - [Installing collections \(collections_using.html#installing-collections\)](#)
 - [Downloading collections \(collections_using.html#downloading-collections\)](#)
 - [Listing collections \(collections_using.html#listing-collections\)](#)
 - [Verifying collections \(collections_using.html#verifying-collections\)](#)
 - [Using collections in a Playbook \(collections_using.html#using-collections-in-a-playbook\)](#)
 - [Simplifying module names with the `collections` keyword \(collections_using.html#simplifying-module-names-with-the-collections-keyword\)](#)
 - [Using a playbook from a collection \(collections_using.html#using-a-playbook-from-a-collection\)](#)

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Ansible concepts

These concepts are common to all uses of Ansible. You need to understand them to use Ansible for any kind of automation. This basic introduction provides the background you need to follow the rest of the User Guide.

- [Control node](#)
- [Managed nodes](#)
- [Inventory](#)
- [Collections](#)
- [Modules](#)
- [Tasks](#)
- [Playbooks](#)

Control node

Any machine with Ansible installed. You can run Ansible commands and playbooks by invoking the `ansible` or `ansible-playbook` command from any control node. You can use any computer that has a Python installation as a control node - laptops, shared desktops, and servers can all run Ansible. However, you cannot use a Windows machine as a control node. You can have multiple control nodes.

Managed nodes

The network devices (and/or servers) you manage with Ansible. Managed nodes are also sometimes called “hosts”. Ansible is not installed on managed nodes.

Inventory

A list of managed nodes. An inventory file is also sometimes called a “hostfile”. Your inventory can specify information like IP address for each managed node. An inventory can also organize managed nodes, creating and nesting groups for easier scaling. To learn more about inventory, see [the Working with Inventory \(intro_inventory.html#intro-inventory\)](#) section.

Collections

Collections are a distribution format for Ansible content that can include playbooks, roles, modules, and plugins. You can install and use collections through [Ansible Galaxy](https://galaxy.ansible.com) (<https://galaxy.ansible.com>). To learn more about collections, see [Using collections](#) ([collections_using.html#collections](#)).

Modules

The units of code Ansible executes. Each module has a particular use, from administering users on a specific type of database to managing VLAN interfaces on a specific type of network device. You can invoke a single module with a task, or invoke several different modules in a playbook. Starting in Ansible 2.10, modules are grouped in collections. For an idea of how many collections Ansible includes, take a look at the [Collection Index](#) ([../collections/index.html#list-of-collections](#)).

Tasks

The units of action in Ansible. You can execute a single task once with an ad hoc command.

Playbooks

Ordered lists of tasks, saved so you can run those tasks in that order repeatedly. Playbooks can include variables as well as tasks. Playbooks are written in YAML and are easy to read, write, share and understand. To learn more about playbooks, see [Intro to playbooks](#) ([playbooks_intro.html#about-playbooks](#)).

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Getting Started

Now that you have read the [installation guide](#) ([../installation_guide/intro_installation.html#installation-guide](#)) and installed Ansible on a control node, you are ready to learn how Ansible works. A basic Ansible command or playbook:

- selects machines to execute against from inventory
- connects to those machines (or network devices, or other managed nodes), usually over SSH
- copies one or more modules to the remote machines and starts execution there

Ansible can do much more, but you should understand the most common use case before exploring all the powerful configuration, deployment, and orchestration features of Ansible. This page illustrates the basic process with a simple inventory and an ad hoc command. Once you understand how Ansible works, you can read more details about [ad hoc commands](#) ([intro_adhoc.html#intro-adhoc](#)), organize your infrastructure with [inventory](#) ([intro_inventory.html#intro-inventory](#)), and harness the full power of Ansible with [playbooks](#) ([playbooks_intro.html#playbooks-intro](#)).

- [Selecting machines from inventory](#)
 - [Action: create a basic inventory](#)
 - [Beyond the basics](#)
- [Connecting to remote nodes](#)
 - [Action: check your SSH connections](#)
 - [Beyond the basics](#)
- [Copying and executing modules](#)
 - [Action: run your first Ansible commands](#)
 - [Action: Run your first playbook](#)
 - [Beyond the basics](#)
- [Resources](#)
- [Next steps](#)

Selecting machines from inventory

Ansible reads information about which machines you want to manage from your inventory. Although you can pass an IP address to an ad hoc command, you need inventory to take advantage of the full flexibility and repeatability of Ansible.

Action: create a basic inventory

For this basic inventory, edit (or create) `/etc/ansible/hosts` and add a few remote systems to it. For this example, use either IP addresses or FQDNs:

```
192.0.2.50
aserver.example.org
bserver.example.org
```

Beyond the basics

Your inventory can store much more than IPs and FQDNs. You can create [aliases](#) ([intro_inventory.html#inventory-aliases](#)), set variable values for a single host with [host vars](#) ([intro_inventory.html#host-variables](#)), or set variable values for multiple hosts with [group vars](#) ([intro_inventory.html#group-variables](#)).

Connecting to remote nodes

Ansible communicates with remote machines over the [SSH protocol](#) (<https://www.ssh.com/ssh/protocol/>). By default, Ansible uses native OpenSSH and connects to remote machines using your current user name, just as SSH does.

Action: check your SSH connections

Confirm that you can connect using SSH to all the nodes in your inventory using the same username. If necessary, add your public SSH key to the `authorized_keys` file on those systems.

Beyond the basics

You can override the default remote user name in several ways, including:

- passing the `-u` parameter at the command line
- setting user information in your inventory file
- setting user information in your configuration file
- setting environment variables

See [Controlling how Ansible behaves: precedence rules](#) ([../reference_appendices/general_precedence.html#general-precedence-rules](#)) for details on the (sometimes unintuitive) precedence of each method of passing user information. You can read more about connections in [Connection methods and details](#) ([connection_details.html#connections](#)).

Copying and executing modules

Once it has connected, Ansible transfers the modules required by your command or playbook to the remote machine(s) for execution.

Action: run your first Ansible commands

Use the ping module to ping all the nodes in your inventory:

```
$ ansible all -m ping
```

You should see output for each host in your inventory, similar to this:

```
aserver.example.org | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python"
  },
  "changed": false,
  "ping": "pong"
}
```

You can use `-u` as one way to specify the user to connect as, by default Ansible uses SSH, which defaults to the 'current user'.

Now run a live command on all of your nodes:

```
$ ansible all -a "/bin/echo hello"
```

You should see output for each host in your inventory, similar to this:

```
aserver.example.org | CHANGED | rc=0 >>
hello
```

Action: Run your first playbook

Playbooks are used to pull together tasks into reusable units.

Ansible does not store playbooks for you; they are simply YAML documents that you store and manage, passing them to Ansible to run as needed.

In a directory of your choice you can create your first playbook in a file called mytask.yaml:

```
---
- name: My playbook
  hosts: all
  tasks:
    - name: Leaving a mark
      command: "touch /tmp/ansible_was_here"
```

You can run this command as follows:

```
$ ansible-playbook mytask.yaml
```

and may see output like this:

```

PLAY [My playbook]
*****

TASK [Gathering Facts]
*****

ok: [aserver.example.org]
ok: [aserver.example.org]
ok: [192.0.2.50]
fatal: [192.0.2.50]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to
the host via ssh: ssh: connect to host 192.0.2.50 port 22: No route to host",
"unreachable": true}

TASK [Leaving a mark]
*****

[WARNING]: Consider using the file module with state=touch rather than running 'touch'.
If you need to use command because file is
insufficient you can add 'warn: false' to this command task or set
'command_warnings=False' in ansible.cfg to get rid of this message.
changed: [aserver.example.org]
changed: [bserver.example.org]

PLAY RECAP
*****

aserver.example.org      : ok=2    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
bserver.example.org      : ok=2    changed=1    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
192.0.2.50               : ok=0    changed=0    unreachable=1    failed=0
skipped=0    rescued=0    ignored=0

```

Read on to learn more about controlling which nodes your playbooks execute on, more sophisticated tasks, and the meaning of the output.

Beyond the basics

By default Ansible uses SFTP to transfer files. If the machine or device you want to manage does not support SFTP, you can switch to SCP mode in [Configuring Ansible](#) ([./installation_guide/intro_configuration.html#intro-configuration](#)). The files are placed in a temporary directory and executed from there.

If you need privilege escalation (sudo and similar) to run a command, pass the `become` flags:

```

# as bruce
$ ansible all -m ping -u bruce
# as bruce, sudoing to root (sudo is default method)
$ ansible all -m ping -u bruce --become
# as bruce, sudoing to batman
$ ansible all -m ping -u bruce --become --become-user batman

```

You can read more about privilege escalation in [Understanding privilege escalation: become \(become.html#become\)](#).

Congratulations! You have contacted your nodes using Ansible. You used a basic inventory file and an ad hoc command to direct Ansible to connect to specific remote nodes, copy a module file there and execute it, and return output. You have a fully working infrastructure.

Resources

- [Product Demos \(https://github.com/ansible/product-demos\)](https://github.com/ansible/product-demos)
- [Katakoda \(https://katacoda.com/rhel-labs\)](https://katacoda.com/rhel-labs)
- [Workshops \(https://github.com/ansible/workshops\)](https://github.com/ansible/workshops)
- [Ansible Examples \(https://github.com/ansible/ansible-examples\)](https://github.com/ansible/ansible-examples)
- [Ansible Baseline \(https://github.com/ansible/ansible-baseline\)](https://github.com/ansible/ansible-baseline)

Next steps

Next you can read about more real-world cases in [Introduction to ad hoc commands \(intro_adhoc.html#intro-adhoc\)](#), explore what you can do with different modules, or read about the Ansible [Working with playbooks \(playbooks.html#working-with-playbooks\)](#) language. Ansible is not just about running commands, it also has powerful configuration management and deployment features.

📌 See also

How to build your inventory (intro_inventory.html#intro-inventory)

More information about inventory

Introduction to ad hoc commands (intro_adhoc.html#intro-adhoc)

Examples of basic commands

Working with playbooks (playbooks.html#working-with-playbooks)

Learning Ansible's configuration management language

Ansible Demos (https://github.com/ansible/product-demos)

Demonstrations of different Ansible usecases

RHEL Labs (https://katacoda.com/rhel-labs)

Labs to provide further knowledge on different topics

Mailing List (https://groups.google.com/group/ansible-project)

Questions? Help? Ideas? Stop by the list on Google Groups

Real-time chat (../community/communication.html#communication-irc)

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Introduction to ad hoc commands

An Ansible ad hoc command uses the `/usr/bin/ansible` command-line tool to automate a single task on one or more managed nodes. ad hoc commands are quick and easy, but they are not reusable. So why learn about ad hoc commands first? ad hoc commands demonstrate the simplicity and power of Ansible. The concepts you learn here will port over directly to the playbook language. Before reading and executing these examples, please read [How to build your inventory \(intro_inventory.html#intro-inventory\)](#).

- [Why use ad hoc commands?](#)
- [Use cases for ad hoc tasks](#)
 - [Rebooting servers](#)
 - [Managing files](#)
 - [Managing packages](#)
 - [Managing users and groups](#)
 - [Managing services](#)
 - [Gathering facts](#)
 - [Patterns and ad-hoc commands](#)

Why use ad hoc commands?

ad hoc commands are great for tasks you repeat rarely. For example, if you want to power off all the machines in your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook. An ad hoc command looks like this:

```
$ ansible [pattern] -m [module] -a "[module options]"
```

You can learn more about [patterns \(intro_patterns.html#intro-patterns\)](#) and [modules \(modules.html#working-with-modules\)](#) on other pages.

Use cases for ad hoc tasks

ad hoc tasks can be used to reboot servers, copy files, manage packages and users, and much more. You can use any Ansible module in an ad hoc task. ad hoc tasks, like playbooks, use a declarative model, calculating and executing the actions required to reach a specified final state. They achieve a form of idempotence by checking the current state before they begin and doing nothing unless the current state is different from the specified final state.

Rebooting servers

The default module for the `ansible` command-line utility is the [`ansible.builtin.command` module](#) ([`./collections/ansible/builtin/command_module.html#command-module`](#)). You can use an ad hoc task to call the command module and reboot all web servers in Atlanta, 10 at a time. Before Ansible can do this, you must have all servers in Atlanta listed in a group called `[atlanta]` in your inventory, and you must have working SSH credentials for each machine in that group. To reboot all the servers in the `[atlanta]` group:

```
$ ansible atlanta -a "/sbin/reboot"
```

By default Ansible uses only 5 simultaneous processes. If you have more hosts than the value set for the fork count, Ansible will talk to them, but it will take a little longer. To reboot the `[atlanta]` servers with 10 parallel forks:

```
$ ansible atlanta -a "/sbin/reboot" -f 10
```

`/usr/bin/ansible` will default to running from your user account. To connect as a different user:

```
$ ansible atlanta -a "/sbin/reboot" -f 10 -u username
```

Rebooting probably requires privilege escalation. You can connect to the server as `username` and run the command as the `root` user by using the [`become`](#) ([`become.html#become`](#)) keyword:

```
$ ansible atlanta -a "/sbin/reboot" -f 10 -u username --become [--ask-become-pass]
```

If you add `--ask-become-pass` or `-K`, Ansible prompts you for the password to use for privilege escalation (`sudo`/`su`/`pfexec`/`doas`/etc).

The [command module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/command_module.html#command-module) ([../collections/ansible/builtin/command_module.html#command-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/command_module.html#command-module)) does not support extended shell syntax like piping and redirects (although shell variables will always work). If your command requires shell-specific syntax, use the *shell* module instead. Read more about the differences on the [Working With Modules](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html#shell-module) ([modules.html#working-with-modules](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html#shell-module)) page.

So far all our examples have used the default 'command' module. To use a different module, pass `-m` for module name. For example, to use the [ansible.builtin.shell module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html#shell-module) ([../collections/ansible/builtin/shell_module.html#shell-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/shell_module.html#shell-module)):

```
$ ansible raleigh -m ansible.builtin.shell -a 'echo $TERM'
```

When running any command with the Ansible *ad hoc* CLI (as opposed to [Playbooks](https://docs.ansible.com/ansible/latest/playbooks.html#working-with-playbooks) ([playbooks.html#working-with-playbooks](https://docs.ansible.com/ansible/latest/playbooks.html#working-with-playbooks))), pay particular attention to shell quoting rules, so the local shell retains the variable and passes it to Ansible. For example, using double rather than single quotes in the above example would evaluate the variable on the box you were on.

Managing files

An ad hoc task can harness the power of Ansible and SCP to transfer many files to multiple machines in parallel. To transfer a file directly to all servers in the [atlanta] group:

```
$ ansible atlanta -m ansible.builtin.copy -a "src=/etc/hosts dest=/tmp/hosts"
```

If you plan to repeat a task like this, use the [ansible.builtin.template](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/template_module.html#template-module) ([../collections/ansible/builtin/template_module.html#template-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/template_module.html#template-module)) module in a playbook.

The [ansible.builtin.file](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html#file-module) ([../collections/ansible/builtin/file_module.html#file-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html#file-module)) module allows changing ownership and permissions on files. These same options can be passed directly to the `copy` module as well:

```
$ ansible webserver -m ansible.builtin.file -a "dest=/srv/foo/a.txt mode=600"
$ ansible webserver -m ansible.builtin.file -a "dest=/srv/foo/b.txt mode=600
owner=mdehaan group=mdehaan"
```

The `file` module can also create directories, similar to `mkdir -p`:

```
$ ansible webservers -m ansible.builtin.file -a "dest=/path/to/c mode=755 owner=mdehaan group=mdehaan state=directory"
```

As well as delete directories (recursively) and delete files:

```
$ ansible webservers -m ansible.builtin.file -a "dest=/path/to/c state=absent"
```

Managing packages

You might also use an ad hoc task to install, update, or remove packages on managed nodes using a package management module like yum. To ensure a package is installed without updating it:

```
$ ansible webservers -m ansible.builtin.yum -a "name=acme state=present"
```

To ensure a specific version of a package is installed:

```
$ ansible webservers -m ansible.builtin.yum -a "name=acme-1.5 state=present"
```

To ensure a package is at the latest version:

```
$ ansible webservers -m ansible.builtin.yum -a "name=acme state=latest"
```

To ensure a package is not installed:

```
$ ansible webservers -m ansible.builtin.yum -a "name=acme state=absent"
```

Ansible has modules for managing packages under many platforms. If there is no module for your package manager, you can install packages using the command module or create a module for your package manager.

Managing users and groups

You can create, manage, and remove user accounts on your managed nodes with ad hoc tasks:

```
$ ansible all -m ansible.builtin.user -a "name=foo password=<crypted password here>"

$ ansible all -m ansible.builtin.user -a "name=foo state=absent"
```

See the [ansible.builtin.user](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html#user-module) ([../collections/ansible/builtin/user_module.html#user-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html#user-module)) module documentation for details on all of the available options, including how to manipulate groups and group membership.

Managing services

Ensure a service is started on all webserver:

```
$ ansible webserver -m ansible.builtin.service -a "name=httpd state=started"
```

Alternatively, restart a service on all webserver:

```
$ ansible webserver -m ansible.builtin.service -a "name=httpd state=restarted"
```

Ensure a service is stopped:

```
$ ansible webserver -m ansible.builtin.service -a "name=httpd state=stopped"
```

Gathering facts

Facts represent discovered variables about a system. You can use facts to implement conditional execution of tasks but also just to get ad hoc information about your systems. To see all facts:

```
$ ansible all -m ansible.builtin.setup
```

You can also filter this output to display only certain facts, see the [ansible.builtin.setup](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/setup_module.html#setup-module) ([../collections/ansible/builtin/setup_module.html#setup-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/setup_module.html#setup-module)) module documentation for details.

Patterns and ad-hoc commands

See the [patterns \(intro_patterns.html#intro-patterns\)](#) documentation for details on all of the available options, including how to limit using patterns in ad-hoc commands.

Now that you understand the basic elements of Ansible execution, you are ready to learn to automate repetitive tasks using [Ansible Playbooks \(playbooks_intro.html#playbooks-intro\)](#).

See also

[Configuring Ansible \(../installation_guide/intro_configuration.html#intro-configuration\)](#)

All about the Ansible config file

[Collection Index \(../collections/index.html#list-of-collections\)](#)

Browse existing collections, modules, and plugins

[Working with playbooks \(playbooks.html#working-with-playbooks\)](#)

Using Ansible for configuration management & deployment

[Mailing List \(https://groups.google.com/group/ansible-project\)](#)

Questions? Help? Ideas? Stop by the list on Google Groups

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible

Define and run a single task ‘playbook’ against a set of hosts

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible [-h] [--version] [-v] [-b] [--become-method BECOME_METHOD]
               [--become-user BECOME_USER]
               [-K | --become-password-file BECOME_PASSWORD_FILE]
               [-i INVENTORY] [--list-hosts] [-l SUBSET] [-P POLL_INTERVAL]
               [-B SECONDS] [-o] [-t TREE] [--private-key PRIVATE_KEY_FILE]
               [-u REMOTE_USER] [-c CONNECTION] [-T TIMEOUT]
               [--ssh-common-args SSH_COMMON_ARGS]
               [--sftp-extra-args SFTP_EXTRA_ARGS]
               [--scp-extra-args SCP_EXTRA_ARGS]
               [--ssh-extra-args SSH_EXTRA_ARGS]
               [-k | --connection-password-file CONNECTION_PASSWORD_FILE] [-C]
               [--syntax-check] [-D] [-e EXTRA_VARS] [--vault-id VAULT_IDS]
               [--ask-vault-password | --vault-password-file VAULT_PASSWORD_FILES]
               [-f FORKS] [-M MODULE_PATH] [--playbook-dir BASEDIR]
               [--task-timeout TASK_TIMEOUT] [-a MODULE_ARGS] [-m MODULE_NAME]
               pattern
```

Description

is an extra-simple tool/framework/API for doing ‘remote things’. this command allows you to define and run a single task ‘playbook’ against a set of hosts

[Search this site](#)

Common Options

--ask-vault-password, --ask-vault-pass

ask for vault password

--become-method <BECOME_METHOD>

privilege escalation method to use (default=sudo), use *ansible-doc -t become -l* to list valid choices.

--become-password-file <BECOME_PASSWORD_FILE>, --become-pass-file <BECOME_PASSWORD_FILE>

Become password file

--become-user <BECOME_USER>

run operations as this user (default=root)

--connection-password-file <CONNECTION_PASSWORD_FILE>, --conn-pass-file <CONNECTION_PASSWORD_FILE>

Connection password file

--list-hosts

outputs a list of matching hosts; does not execute anything else

--playbook-dir <BASEDIR>

Since this tool does not use playbooks, use this as a substitute playbook directory. This sets the relative path for many features including roles/ group_vars/ etc.

--private-key <PRIVATE_KEY_FILE>, --key-file <PRIVATE_KEY_FILE>

use this file to authenticate the connection

--scp-extra-args <SCP_EXTRA_ARGS>

specify extra arguments to pass to scp only (e.g. -l)

--sftp-extra-args <SFTP_EXTRA_ARGS>

specify extra arguments to pass to sftp only (e.g. -f, -l)

--ssh-common-args <SSH_COMMON_ARGS>

specify common arguments to pass to sftp/scp/ssh (e.g. ProxyCommand)

--ssh-extra-args <SSH_EXTRA_ARGS>

specify extra arguments to pass to ssh only (e.g. -R)

--syntax-check

perform a syntax check on the playbook, but do not execute it

--task-timeout <TASK_TIMEOUT>

set task timeout limit in seconds, must be positive integer.

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-B <SECONDS>, --background <SECONDS>

run asynchronously, failing after X seconds (default=N/A)

-C, --check

don't make any changes; instead, try to predict some of the changes that may occur

-D, --diff

when changing (small) files and templates, show the differences in those files; works great with -check

-K, --ask-become-pass

ask for privilege escalation password

-M, --module-path

prepend colon-separated path(s) to module library
(default=~/.ansible/plugins/modules:/usr/share/ansible/plugins/modules)

-P <POLL_INTERVAL>, --poll <POLL_INTERVAL>

set the poll interval if using -B (default=15)

-T <TIMEOUT>, --timeout <TIMEOUT>

override the connection timeout in seconds (default=10)

-a <MODULE_ARGS>, --args <MODULE_ARGS>

The action's options in space separated k=v format: -a 'opt1=val1 opt2=val2'

-b, --become

run operations with become (does not imply password prompting)

-c <CONNECTION>, --connection <CONNECTION>

connection type to use (default=smart)

-e, --extra-vars

set additional variables as key=value or YAML/JSON, if filename prepend with @

-f <FORKS>, --forks <FORKS>

specify number of parallel processes to use (default=5)

-h, --help

show this help message and exit

-i, --inventory, --inventory-file

specify inventory host path or comma separated host list. --inventory-file is deprecated

-k, --ask-pass

ask for connection password

-l <SUBSET>, --limit <SUBSET>

further limit selected hosts to an additional pattern

-m <MODULE_NAME>, --module-name <MODULE_NAME>

Name of the action to execute (default=command)

-o, --one-line

condense output

-t <TREE>, --tree <TREE>

log output to this directory

-u <REMOTE_USER>, --user <REMOTE_USER>

connect as this user (default=None)

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Environment

The following environment variables may be specified.

`ANSIBLE_CONFIG` [\(../reference_appendices/config.html#envvar-ANSIBLE_CONFIG\)](#) – Override the default ansible config file

Many more are available for most options in `ansible.cfg`

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Working with playbooks

Playbooks record and execute Ansible's configuration, deployment, and orchestration functions. They can describe a policy you want your remote systems to enforce, or a set of steps in a general IT process.

If Ansible modules are the tools in your workshop, playbooks are your instruction manuals, and your inventory of hosts are your raw material.

At a basic level, playbooks can be used to manage configurations of and deployments to remote machines. At a more advanced level, they can sequence multi-tier rollouts involving rolling updates, and can delegate actions to other hosts, interacting with monitoring servers and load balancers along the way.

Playbooks are designed to be human-readable and are developed in a basic text language. There are multiple ways to organize playbooks and the files they include, and we'll offer up some suggestions on that and making the most out of Ansible.

You should look at [Example Playbooks \(https://github.com/ansible/ansible-examples\)](https://github.com/ansible/ansible-examples) while reading along with the playbook documentation. These illustrate best practices as well as how to put many of the various concepts together.

- [Templating \(Jinja2\) \(playbooks_templating.html\)](#)
 - [Using filters to manipulate data \(playbooks_filters.html\)](#)
 - [Tests \(playbooks_tests.html\)](#)
 - [Lookups \(playbooks_lookups.html\)](#)
 - [Python3 in templates \(playbooks_python_version.html\)](#)
 - [Get the current time \(playbooks_templating.html#get-the-current-time\)](#)
- [Advanced playbooks features \(playbooks_special_topics.html\)](#)
- [Playbook Example: Continuous Delivery and Rolling Upgrades \(guide_rolling_upgrade.html\)](#)
 - [What is continuous delivery? \(guide_rolling_upgrade.html#what-is-continuous-delivery\)](#)

- [Site deployment \(guide_rolling_upgrade.html#site-deployment\)](#)
- [Reusable content: roles \(guide_rolling_upgrade.html#reusable-content-roles\)](#)
- [Configuration: group variables \(guide_rolling_upgrade.html#configuration-group-variables\)](#)
- [The rolling upgrade \(guide_rolling_upgrade.html#the-rolling-upgrade\)](#)
- [Managing other load balancers \(guide_rolling_upgrade.html#managing-other-load-balancers\)](#)
- [Continuous delivery end-to-end \(guide_rolling_upgrade.html#continuous-delivery-end-to-end\)](#)

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-config

View ansible configuration.

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Actions](#)
 - [list](#)
 - [dump](#)
 - [view](#)
 - [init](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-config [-h] [--version] [-v] {list,dump,view,init} ...
```

Description

Config command line class

Common Options

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-h, --help

show this help message and exit

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Actions

list

list and output available configs

-c <CONFIG_FILE>, --config <CONFIG_FILE>

path to configuration file, defaults to first file found in precedence.

-t <TYPE>, --type <TYPE>

Filter down to a specific plugin type.

dump

Shows the current settings, merges ansible.cfg if specified

--only-changed, --changed-only

Only show configurations that have changed from the default

-c <CONFIG_FILE>, --config <CONFIG_FILE>

path to configuration file, defaults to first file found in precedence.

-t <TYPE>, --type <TYPE>

Filter down to a specific plugin type.

view

Displays the current config file

-c <CONFIG_FILE>, --config <CONFIG_FILE>

path to configuration file, defaults to first file found in precedence.

-t <TYPE>, --type <TYPE>

Filter down to a specific plugin type.

init

--disabled

Prefixes all entries with a comment character to disable them

--format <FORMAT>, -f <FORMAT>

Output format for init

-c <CONFIG_FILE>, --config <CONFIG_FILE>

path to configuration file, defaults to first file found in precedence.

-t <TYPE>, --type <TYPE>

Filter down to a specific plugin type.

Environment

The following environment variables may be specified.

`ANSIBLE_CONFIG` ([../reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in `ansible.cfg`

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Intro to playbooks

Ansible Playbooks offer a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications. If you need to execute a task with Ansible more than once, write a playbook and put it under source control. Then you can use the playbook to push out new configuration or confirm the configuration of remote systems. The playbooks in the [ansible-examples repository](#) (<https://github.com/ansible/ansible-examples>) illustrate many useful techniques. You may want to look at these in another tab as you read the documentation.

Playbooks can:

- declare configurations
- orchestrate steps of any manual ordered process, on multiple sets of machines, in a defined order
- launch tasks synchronously or [asynchronously \(playbooks_async.html#playbooks-async\)](#)
- [Playbook syntax](#)
- [Playbook execution](#)
 - [Task execution](#)
 - [Desired state and 'idempotency'](#)
 - [Running playbooks](#)
- [Ansible-Pull](#)
- [Verifying playbooks](#)
 - [ansible-lint](#)

Playbook syntax

Playbooks are expressed in YAML format with a minimum of syntax. If you are not familiar with YAML, look at our overview of [YAML Syntax](#) ([../reference_appendices/YAMLSyntax.html#yaml-syntax](#)) and consider installing an add-on for your text editor (see [Other Tools and Programs](#) ([../community/other_tools_and_programs.html#other-tools-and-programs](#))) to help you write clean YAML syntax in your playbooks.

A playbook is composed of one or more ‘plays’ in an ordered list. The terms ‘playbook’ and ‘play’ are sports analogies. Each play executes part of the overall goal of the playbook, running one or more tasks. Each task calls an Ansible module.

Playbook execution

A playbook runs in order from top to bottom. Within each play, tasks also run in order from top to bottom. Playbooks with multiple ‘plays’ can orchestrate multi-machine deployments, running one play on your web servers, then another play on your database servers, then a third play on your network infrastructure, and so on. At a minimum, each play defines two things:

- the managed nodes to target, using a [pattern](#) ([intro_patterns.html#intro-patterns](#)).
- at least one task to execute

📌 Note

In Ansible 2.10 and later, we recommend you use the fully-qualified collection name in your playbooks to ensure the correct module is selected, because multiple collections can contain modules with the same name (for example, `user`). See [Using collections in a Playbook](#) ([collections_using.html#collections-using-playbook](#)).

In this example, the first play targets the web servers; the second play targets the database servers.

```

---
- name: Update web servers
  hosts: webservers
  remote_user: root

  tasks:
    - name: Ensure apache is at the latest version
      ansible.builtin.yum:
        name: httpd
        state: latest
    - name: Write the apache config file
      ansible.builtin.template:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf

- name: Update db servers
  hosts: databases
  remote_user: root

  tasks:
    - name: Ensure postgresql is at the latest version
      ansible.builtin.yum:
        name: postgresql
        state: latest
    - name: Ensure that postgresql is started
      ansible.builtin.service:
        name: postgresql
        state: started

```

Your playbook can include more than just a hosts line and tasks. For example, the playbook above sets a `remote_user` for each play. This is the user account for the SSH connection. You can add other [Playbook Keywords](#) ([../reference_appendices/playbooks_keywords.html#playbook-keywords](#)) at the playbook, play, or task level to influence how Ansible behaves. Playbook keywords can control the [connection plugin](#) ([../plugins/connection.html#connection-plugins](#)), whether to use [privilege escalation](#) ([become.html#become](#)), how to handle errors, and more. To support a variety of environments, Ansible lets you set many of these parameters as command-line flags, in your Ansible configuration, or in your inventory. Learning the [precedence rules](#) ([../reference_appendices/general_precedence.html#general-precedence-rules](#)) for these sources of data will help you as you expand your Ansible ecosystem.

Task execution

By default, Ansible executes each task in order, one at a time, against all machines matched by the host pattern. Each task executes a module with specific arguments. When a task has executed on all target machines, Ansible moves on to the next task. You can use [strategies](#) ([playbooks_strategies.html#playbooks-strategies](#)) to change this default behavior. Within each play, Ansible applies the same task directives to all hosts. If a task fails on a host, Ansible takes that host out of the rotation for the rest of the playbook.

When you run a playbook, Ansible returns information about connections, the `name` lines of all your plays and tasks, whether each task has succeeded or failed on each machine, and whether each task has made a change on each machine. At the bottom of the playbook execution, Ansible provides a summary of the nodes that were targeted and how they performed. General failures and fatal “unreachable” communication attempts are kept separate in the counts.

Desired state and ‘idempotency’

Most Ansible modules check whether the desired final state has already been achieved, and exit without performing any actions if that state has been achieved, so that repeating the task does not change the final state. Modules that behave this way are often called ‘idempotent.’ Whether you run a playbook once, or multiple times, the outcome should be the same. However, not all playbooks and not all modules behave this way. If you are unsure, test your playbooks in a sandbox environment before running them multiple times in production.

Running playbooks

To run your playbook, use the [ansible-playbook \(../cli/ansible-playbook.html#ansible-playbook\)](https://cli/ansible-playbook.html#ansible-playbook) command.

```
ansible-playbook playbook.yml -f 10
```

Use the `--verbose` flag when running your playbook to see detailed output from successful modules as well as unsuccessful ones.

Ansible-Pull

Should you want to invert the architecture of Ansible, so that nodes check in to a central location, instead of pushing configuration out to them, you can.

The `ansible-pull` is a small script that will checkout a repo of configuration instructions from git, and then run `ansible-playbook` against that content.

Assuming you load balance your checkout location, `ansible-pull` scales essentially infinitely.

Run `ansible-pull --help` for details.

There’s also a [clever playbook \(https://github.com/ansible/ansible-examples/blob/master/language_features/ansible_pull.yml\)](https://github.com/ansible/ansible-examples/blob/master/language_features/ansible_pull.yml) available to configure `ansible-pull` via a crontab from push mode.

Verifying playbooks

You may want to verify your playbooks to catch syntax errors and other problems before you run them. The [ansible-playbook \(../cli/ansible-playbook.html#ansible-playbook\)](#) command offers several options for verification, including `--check`, `--diff`, `--list-hosts`, `--list-tasks`, and `--syntax-check`. The [Tools for validating playbooks \(../community/other_tools_and_programs.html#validate-playbook-tools\)](#) describes other tools for validating and testing playbooks.

ansible-lint

You can use [ansible-lint \(https://docs.ansible.com/ansible-lint/index.html\)](#) for detailed, Ansible-specific feedback on your playbooks before you execute them. For example, if you run `ansible-lint` on the playbook called `verify-apache.yml` near the top of this page, you should get the following results:

```
$ ansible-lint verify-apache.yml
[403] Package installs should not use latest
verify-apache.yml:8
Task/Handler: ensure apache is at the latest version
```

The [ansible-lint default rules \(https://docs.ansible.com/ansible-lint/rules/default_rules.html\)](#) page describes each error. For `[403]`, the recommended fix is to change `state: latest` to `state: present` in the playbook.

See also

[ansible-lint \(https://docs.ansible.com/ansible-lint/index.html\)](#)

Learn how to test Ansible Playbooks syntax

[YAML Syntax \(../reference_appendices/YAMLSyntax.html#yaml-syntax\)](#)

Learn about YAML syntax

[Tips and tricks \(playbooks_best_practices.html#playbooks-best-practices\)](#)

Tips for managing playbooks in the real world

[Collection Index \(../collections/index.html#list-of-collections\)](#)

Browse existing collections, modules, and plugins

[Should you develop a module? \(../dev_guide/developing_modules.html#developing-modules\)](#)

Learn to extend Ansible by writing your own modules

[Patterns: targeting hosts and groups \(intro_patterns.html#intro-patterns\)](#)

Search this site

Learn about how to select hosts

GitHub examples directory (<https://github.com/ansible/ansible-examples>)

Complete end-to-end playbook examples

Mailing List (<https://groups.google.com/group/ansible-project>)

Questions? Help? Ideas? Stop by the list on Google Groups

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-console

REPL console for executing Ansible tasks.

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-console [-h] [--version] [-v] [-b]
                        [--become-method BECOME_METHOD]
                        [--become-user BECOME_USER]
                        [-K | --become-password-file BECOME_PASSWORD_FILE]
                        [-i INVENTORY] [--list-hosts] [-l SUBSET]
                        [--private-key PRIVATE_KEY_FILE] [-u REMOTE_USER]
                        [-c CONNECTION] [-T TIMEOUT]
                        [--ssh-common-args SSH_COMMON_ARGS]
                        [--sftp-extra-args SFTP_EXTRA_ARGS]
                        [--scp-extra-args SCP_EXTRA_ARGS]
                        [--ssh-extra-args SSH_EXTRA_ARGS]
                        [-k | --connection-password-file CONNECTION_PASSWORD_FILE]
                        [-C] [--syntax-check] [-D] [--vault-id VAULT_IDS]
                        [--ask-vault-password | --vault-password-file VAULT_PASSWORD_FILES]
                        [-f FORKS] [-M MODULE_PATH] [--playbook-dir BASEDIR]
                        [-e EXTRA_VARS] [--task-timeout TASK_TIMEOUT] [--step]
                        [pattern]
```

Description

A REPL that allows for running ad-hoc tasks against a chosen inventory from a nice shell with built-in tab completion (based on dominis' ansible-shell).

It supports several commands, and you can modify its configuration at runtime:

- *cd [pattern]*: change host/group (you can use host patterns eg.: app*.dc*:*app01*)
- *list*: list available hosts in the current path
- *list groups*: list groups included in the current path
- *become*: toggle the become flag
- *!:* forces shell module instead of the ansible module (!yum update -y)
- *verbosity [num]*: set the verbosity level
- *forks [num]*: set the number of forks
- *become_user [user]*: set the become_user
- *remote_user [user]*: set the remote_user
- *become_method [method]*: set the privilege escalation method
- *check [bool]*: toggle check mode
- *diff [bool]*: toggle diff mode
- *timeout [integer]*: set the timeout of tasks in seconds (0 to disable)
- *help [command/module]*: display documentation for the command or module
- *exit*: exit ansible-console

Common Options

--ask-vault-password, --ask-vault-pass

ask for vault password

--become-method <BECOME_METHOD>

privilege escalation method to use (default=sudo), use *ansible-doc -t become -l* to list valid choices.

--become-password-file <BECOME_PASSWORD_FILE>, --become-pass-file <BECOME_PASSWORD_FILE>

Become password file

--become-user <BECOME_USER>

run operations as this user (default=root)

--connection-password-file <CONNECTION_PASSWORD_FILE>, --conn-pass-file <CONNECTION_PASSWORD_FILE>

Connection password file

--list-hosts

Search this site

outputs a list of matching hosts; does not execute anything else

--playbook-dir <BASEDIR>

Since this tool does not use playbooks, use this as a substitute playbook directory. This sets the relative path for many features including roles/ group_vars/ etc.

--private-key <PRIVATE_KEY_FILE>, --key-file <PRIVATE_KEY_FILE>

use this file to authenticate the connection

--scp-extra-args <SCP_EXTRA_ARGS>

specify extra arguments to pass to scp only (e.g. -l)

--sftp-extra-args <SFTP_EXTRA_ARGS>

specify extra arguments to pass to sftp only (e.g. -f, -l)

--ssh-common-args <SSH_COMMON_ARGS>

specify common arguments to pass to sftp/scp/ssh (e.g. ProxyCommand)

--ssh-extra-args <SSH_EXTRA_ARGS>

specify extra arguments to pass to ssh only (e.g. -R)

--step

one-step-at-a-time: confirm each task before running

--syntax-check

perform a syntax check on the playbook, but do not execute it

--task-timeout <TASK_TIMEOUT>

set task timeout limit in seconds, must be positive integer.

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-C, --check

don't make any changes; instead, try to predict some of the changes that may occur

-D, --diff

when changing (small) files and templates, show the differences in those files; works great with -check

-K, --ask-become-pass

ask for privilege escalation password

-M, --module-path

prepend colon-separated path(s) to module library
(default=~/.ansible/plugins/modules:/usr/share/ansible/plugins/modules)

-T <TIMEOUT>, --timeout <TIMEOUT>

override the connection timeout in seconds (default=10)

-b, --become

run operations with become (does not imply password prompting)

-c <CONNECTION>, --connection <CONNECTION>

connection type to use (default=smart)

-e, --extra-vars

set additional variables as key=value or YAML/JSON, if filename prepend with @

-f <FORKS>, --forks <FORKS>

specify number of parallel processes to use (default=5)

-h, --help

show this help message and exit

-i, --inventory, --inventory-file

specify inventory host path or comma separated host list. -inventory-file is deprecated

-k, --ask-pass

ask for connection password

-l <SUBSET>, --limit <SUBSET>

further limit selected hosts to an additional pattern

-u <REMOTE_USER>, --user <REMOTE_USER>

connect as this user (default=None)

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Environment

The following environment variables may be specified.

`ANSIBLE_CONFIG` ([../reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in `ansible.cfg`

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

Search this site

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Tips and tricks

These tips and tricks have helped us optimize our Ansible usage, and we offer them here as suggestions. We hope they will help you organize content, write playbooks, maintain inventory, and execute Ansible. Ultimately, though, you should use Ansible in the way that makes most sense for your organization and your goals.

- [General tips](#)
 - [Keep it simple](#)
 - [Use version control](#)
- [Playbook tips](#)
 - [Use whitespace](#)
 - [Always name tasks](#)
 - [Always mention the state](#)
 - [Use comments](#)
- [Inventory tips](#)
 - [Use dynamic inventory with clouds](#)
 - [Group inventory by function](#)
 - [Separate production and staging inventory](#)
 - [Keep vaulted variables safely visible](#)
- [Execution tricks](#)
 - [Try it in staging first](#)
 - [Update in batches](#)
 - [Handling OS and distro differences](#)

General tips

These concepts apply to all Ansible activities and artifacts.

Keep it simple

Whenever you can, do things simply. Use advanced features only when necessary, and select the feature that best matches your use case. For example, you will probably not need `vars`, `vars_files`, `vars_prompt` and `--extra-vars` all at once, while also using an external

inventory file. If something feels complicated, it probably is. Take the time to look for a simpler solution.

Use version control

Keep your playbooks, roles, inventory, and variables files in git or another version control system and make commits to the repository when you make changes. Version control gives you an audit trail describing when and why you changed the rules that automate your infrastructure.

Playbook tips

These tips help make playbooks and roles easier to read, maintain, and debug.

Use whitespace

Generous use of whitespace, for example, a blank line before each block or task, makes a playbook easy to scan.

Always name tasks

Task names are optional, but extremely useful. In its output, Ansible shows you the name of each task it runs. Choose names that describe what each task does and why.

Always mention the state

For many modules, the 'state' parameter is optional. Different modules have different default settings for 'state', and some modules support several 'state' settings. Explicitly setting 'state=present' or 'state=absent' makes playbooks and roles clearer.

Use comments

Even with task names and explicit state, sometimes a part of a playbook or role (or inventory/variable file) needs more explanation. Adding a comment (any line starting with '#') helps others (and possibly yourself in future) understand what a play or task (or variable setting) does, how it does it, and why.

Inventory tips

These tips help keep your inventory well organized.

Use dynamic inventory with clouds

With cloud providers and other systems that maintain canonical lists of your infrastructure, use [dynamic inventory](#) ([intro_dynamic_inventory.html#intro-dynamic-inventory](#)) to retrieve those lists instead of manually updating static inventory files. With cloud resources, you can use tags to differentiate production and staging environments.

Group inventory by function

A system can be in multiple groups. See [How to build your inventory](#) ([intro_inventory.html#intro-inventory](#)) and [Patterns: targeting hosts and groups](#) ([intro_patterns.html#intro-patterns](#)). If you create groups named for the function of the nodes in the group, for example *webservers* or *dbservers*, your playbooks can target machines based on function. You can assign function-specific variables using the group variable system, and design Ansible roles to handle function-specific use cases. See [Roles](#) ([playbooks_reuse_roles.html#playbooks-reuse-roles](#)).

Separate production and staging inventory

You can keep your production environment separate from development, test, and staging environments by using separate inventory files or directories for each environment. This way you pick with `-i` what you are targeting. Keeping all your environments in one file can lead to surprises!

Keep vaulted variables safely visible

You should encrypt sensitive or secret variables with Ansible Vault. However, encrypting the variable names as well as the variable values makes it hard to find the source of the values. You can keep the names of your variables accessible (by `grep`, for example) without exposing any secrets by adding a layer of indirection:

1. Create a `group_vars/` subdirectory named after the group.
2. Inside this subdirectory, create two files named `vars` and `vault`.
3. In the `vars` file, define all of the variables needed, including any sensitive ones.
4. Copy all of the sensitive variables over to the `vault` file and prefix these variables with `vault_`.
5. Adjust the variables in the `vars` file to point to the matching `vault_` variables using jinja2 syntax: `db_password: {{ vault_db_password }}`.
6. Encrypt the `vault` file to protect its contents.
7. Use the variable name from the `vars` file in your playbooks.

When running a playbook, Ansible finds the variables in the unencrypted file, which pulls the sensitive variable values from the encrypted file. There is no limit to the number of variable and vault files or their names.

Execution tricks

These tips apply to using Ansible, rather than to Ansible artifacts.

Try it in staging first

Testing changes in a staging environment before rolling them out in production is always a great idea. Your environments need not be the same size and you can use group variables to control the differences between those environments.

Update in batches

Use the 'serial' keyword to control how many machines you update at once in the batch. See [Controlling where tasks run: delegation and local actions](#) ([playbooks_delegation.html#playbooks-delegation](#)).

Handling OS and distro differences

Group variables files and the `group_by` module work together to help Ansible execute across a range of operating systems and distributions that require different settings, packages, and tools. The `group_by` module creates a dynamic group of hosts matching certain criteria. This group does not need to be defined in the inventory file. This approach lets you execute different tasks on different operating systems or distributions. For example:

```
---

- name: talk to all hosts just so we can learn about them
  hosts: all
  tasks:
    - name: Classify hosts depending on their OS distribution
      group_by:
        key: os_{{ ansible_facts['distribution'] }}

# now just on the CentOS hosts...

- hosts: os_CentOS
  gather_facts: False
  tasks:
    - # tasks that only happen on CentOS go in this play
```

The first play categorizes all systems into dynamic groups based on the operating system name. Later plays can use these groups as patterns on the `hosts` line. You can also add group-specific settings in group vars files. All three names must match: the name created by the `group_by` task, the name of the pattern in subsequent plays, and the name of the group vars file. For example:


```
---
# file: group_vars/all
asdf: 10

---
# file: group_vars/os_CentOS.yml
asdf: 42
```

In this example, CentOS machines get the value of '42' for asdf, but other machines get '10'. This can be used not only to set variables, but also to apply certain roles to only certain systems.

You can use the same setup with `include_vars` when you only need OS-specific variables, not tasks:

```
- hosts: all
  tasks:
    - name: Set OS distribution dependent variables
      include_vars: "os_{{ ansible_facts['distribution'] }}.yaml"
    - debug:
        var: asdf
```

This pulls in variables from the group_vars/os_CentOS.yml file.

❗ See also

[YAML Syntax \(../reference_appendices/YAMLSyntax.html#yaml-syntax\)](#)

Learn about YAML syntax

[Working with playbooks \(playbooks.html#working-with-playbooks\)](#)

Review the basic playbook features

[Collection Index \(../collections/index.html#list-of-collections\)](#)

Browse existing collections, modules, and plugins

[Should you develop a module? \(../dev_guide/developing_modules.html#developing-modules\)](#)

Learn how to extend Ansible by writing your own modules

[Patterns: targeting hosts and groups \(intro_patterns.html#intro-patterns\)](#)

Learn about how to select hosts

[GitHub examples directory \(https://github.com/ansible/ansible-examples\)](#)

Complete playbook files from the github project source

Mailing List (<https://groups.google.com/group/ansible-project>)

Questions? Help? Ideas? Stop by the list on Google Groups

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-doc

plugin documentation tool

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-doc [-h] [--version] [-v] [-M MODULE_PATH]
                  [--playbook-dir BASEDIR]
                  [-t
{become,cache,callback,cliconf,connection,httpapi,inventory,lookup,netconf,shell,vars,mod
                  [-j] [-r ROLES_PATH]
                  [-e ENTRY_POINT | -s | -F | -l | --metadata-dump]
                  [plugin [plugin ...]]
```

Description

displays information on modules installed in Ansible libraries. It displays a terse listing of plugins and their short descriptions, provides a printout of their DOCUMENTATION strings, and it can create a short “snippet” which can be pasted into a playbook.

Common Options

--metadata-dump

For internal testing only Dump json metadata for all plugins.

--playbook-dir <BASEDIR>

Since this tool does not use playbooks, use this as a substitute playbook directory. This sets the relative path for many features including roles/ group_vars/ etc.

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-F, --list_files

Show plugin names and their source files without summaries (implies -list). A supplied argument will be used for filtering, can be a namespace or full collection name.

-M, --module-path

prepend colon-separated path(s) to module library
(default=~/.ansible/plugins/modules:/usr/share/ansible/plugins/modules)

-e <ENTRY_POINT>, --entry-point <ENTRY_POINT>

Select the entry point for role(s).

-h, --help

show this help message and exit

-j, --json

Change output into json format.

-l, --list

List available plugins. A supplied argument will be used for filtering, can be a namespace or full collection name.

-r, --roles-path

The path to the directory containing your roles.

-s, --snippet

Show playbook snippet for these plugin types: inventory, lookup, module

-t <TYPE>, --type <TYPE>

Choose which plugin type (defaults to “module”). Available plugin types are : ('become', 'cache', 'callback', 'cliconf', 'connection', 'httpapi', 'inventory', 'lookup', 'netconf', 'shell', 'vars', 'module', 'strategy', 'role', 'keyword')

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Environment

The following environment variables may be specified.

ANSIBLE_CONFIG [\(../reference_appendices/config.html#envvar-ANSIBLE_CONFIG\)](#) – Override the default ansible config file

Many more are available for most options in ansible.cfg

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Understanding privilege escalation: become

Ansible uses existing privilege escalation systems to execute tasks with root privileges or with another user's permissions. Because this feature allows you to 'become' another user, different from the user that logged into the machine (remote user), we call it `become`. The `become` keyword uses existing privilege escalation tools like *sudo*, *su*, *pfexec*, *doas*, *pbrun*, *dzdo*, *ksu*, *runas*, *machinectl* and others.

- [Using become](#)
 - [Become directives](#)
 - [Become connection variables](#)
 - [Become command-line options](#)
- [Risks and limitations of become](#)
 - [Risks of becoming an unprivileged user](#)
 - [Not supported by all connection plugins](#)
 - [Only one method may be enabled per host](#)
 - [Privilege escalation must be general](#)
 - [May not access environment variables populated by pamd systemd](#)
- [Become and network automation](#)
 - [Setting enable mode for all tasks](#)
 - [Passwords for enable mode](#)
 - [authorize and auth_pass](#)
- [Become and Windows](#)
 - [Administrative rights](#)
 - [Local service accounts](#)
 - [Become without setting a password](#)
 - [Accounts without a password](#)
 - [Become flags for Windows](#)
 - [Limitations of become on Windows](#)

Using become

You can control the use of `become` with play or task directives, connection variables, or at the command line. If you set privilege escalation properties in multiple ways, review the [general precedence rules \(./reference_appendices/general_precedence.html#general-precedence-rules\)](https://docs.ansible.com/ansible/latest/reference_appendices/general_precedence.html#general-precedence-rules) to understand which settings will be used.

A full list of all become plugins that are included in Ansible can be found in the [Plugin List \(./plugins/become.html#become-plugin-list\)](https://docs.ansible.com/ansible/latest/plugins/become.html#become-plugin-list).

Become directives

You can set the directives that control `become` at the play or task level. You can override these by setting connection variables, which often differ from one host to another. These variables and directives are independent. For example, setting `become_user` does not set `become`.

become

set to `yes` to activate privilege escalation.

become_user

set to user with desired privileges — the user you *become*, NOT the user you login as.

Does NOT imply `become: yes`, to allow it to be set at host level. Default value is `root`.

become_method

(at play or task level) overrides the default method set in `ansible.cfg`, set to use any of the [Become plugins \(./plugins/become.html#become-plugins\)](https://docs.ansible.com/ansible/latest/plugins/become.html#become-plugins).

become_flags

(at play or task level) permit the use of specific flags for the tasks or role. One common use is to change the user to nobody when the shell is set to nologin. Added in Ansible 2.2.

For example, to manage a system service (which requires `root` privileges) when connected as a non-`root` user, you can use the default value of `become_user` (`root`):

```
- name: Ensure the httpd service is running
  service:
    name: httpd
    state: started
    become: yes
```

To run a command as the `apache` user:

```
- name: Run a command as the apache user
  command: somecommand
  become: yes
  become_user: apache
```

To do something as the `nobody` user when the shell is nologin:

```
- name: Run a command as nobody
  command: somecommand
  become: yes
  become_method: su
  become_user: nobody
  become_flags: '-s /bin/sh'
```

To specify a password for sudo, run `ansible-playbook` with `--ask-become-pass` (`-K` for short). If you run a playbook utilizing `become` and the playbook seems to hang, most likely it is stuck at the privilege escalation prompt. Stop it with `CTRL-c`, then execute the playbook with `-K` and the appropriate password.

Become connection variables

You can define different `become` options for each managed node or group. You can define these variables in inventory or use them as normal variables.

ansible_become

overrides the `become` directive, decides if privilege escalation is used or not.

ansible_become_method

which privilege escalation method should be used

ansible_become_user

set the user you become through privilege escalation; does not imply `ansible_become: yes`

ansible_become_password

set the privilege escalation password. See [Using encrypted variables and files \(vault.html#playbooks-vault\)](#) for details on how to avoid having secrets in plain text

ansible_common_remote_group

determines if Ansible should try to `chgrp` its temporary files to a group if `setfacl` and `chown` both fail. See [Risks of becoming an unprivileged user](#) for more information. Added in version 2.10.

For example, if you want to run all tasks as `root` on a server named `webserver`, but you can only connect as the `manager` user, you could use an inventory entry like this: [Search this site](#)


```
webserver ansible_user=manager ansible_become=yes
```

❗ Note

The variables defined above are generic for all become plugins but plugin specific ones can also be set instead. Please see the documentation for each plugin for a list of all options the plugin has and how they can be defined. A full list of become plugins in Ansible can be found at [Become plugins \(./plugins/become.html#become-plugins\)](https://docs.ansible.com/ansible/latest/plugins/become.html#become-plugins).

Become command-line options

--ask-become-pass, -K

ask for privilege escalation password; does not imply become will be used. Note that this password will be used for all hosts.

--become, -b

run operations with become (no password implied)

--become-method=*BECOME_METHOD*

privilege escalation method to use (default=sudo), valid choices: [sudo | su | pbrun | pfexec | doas | dzdo | ksu | runas | machinectl]

--become-user=*BECOME_USER*

run operations as this user (default=root), does not imply --become/-b

Risks and limitations of become

Although privilege escalation is mostly intuitive, there are a few limitations on how it works. Users should be aware of these to avoid surprises.

Risks of becoming an unprivileged user

Ansible modules are executed on the remote machine by first substituting the parameters into the module file, then copying the file to the remote machine, and finally executing it there.

Everything is fine if the module file is executed without using `become`, when the `become_user` is root, or when the connection to the remote machine is made as root. In these cases Ansible creates the module file with permissions that only allow reading by the user and root, or only allow reading by the unprivileged user being switched to.

However, when both the connection user and the `become_user` are unprivileged, the module file is written as the user that Ansible connects as (the `remote_user`), but the file needs to be readable by the user Ansible is set to `become`. The details of how Ansible solves this can vary based on platform. However, on POSIX systems, Ansible solves this problem in the following way:

First, if **setfacl** is installed and available in the remote `PATH`, and the temporary directory on the remote host is mounted with POSIX.1e filesystem ACL support, Ansible will use POSIX ACLs to share the module file with the second unprivileged user.

Next, if POSIX ACLs are **not** available or **setfacl** could not be run, Ansible will attempt to change ownership of the module file using **chown** for systems which support doing so as an unprivileged user.

New in Ansible 2.11, at this point, Ansible will try **chmod +a** which is a macOS-specific way of setting ACLs on files.

New in Ansible 2.10, if all of the above fails, Ansible will then check the value of the configuration setting `ansible_common_remote_group`. Many systems will allow a given user to change the group ownership of a file to a group the user is in. As a result, if the second unprivileged user (the `become_user`) has a UNIX group in common with the user Ansible is connected as (the `remote_user`), and if `ansible_common_remote_group` is defined to be that group, Ansible can try to change the group ownership of the module file to that group by using **chgrp**, thereby likely making it readable to the `become_user`.

At this point, if `ansible_common_remote_group` was defined and a **chgrp** was attempted and returned successfully, Ansible assumes (but, importantly, does not check) that the new group ownership is enough and does not fall back further. That is, Ansible **does not check** that the `become_user` does in fact share a group with the `remote_user`; so long as the command exits successfully, Ansible considers the result successful and does not proceed to check `allow_world_readable_tmpfiles` per below.

If `ansible_common_remote_group` is **not** set and the **chown** above it failed, or if `ansible_common_remote_group` is set but the **chgrp** (or following group-permissions **chmod**) returned a non-successful exit code, Ansible will lastly check the value of `allow_world_readable_tmpfiles`. If this is set, Ansible will place the module file in a world-readable temporary directory, with world-readable permissions to allow the `become_user` (and incidentally any other user on the system) to read the contents of the file. **If any of the parameters passed to the module are sensitive in nature, and you do not trust the remote machines, then this is a potential security risk.**

Once the module is done executing, Ansible deletes the temporary file.

Several ways exist to avoid the above logic flow entirely:

[Search this site](#)

- Use *pipelining*. When pipelining is enabled, Ansible does not save the module to a temporary file on the client. Instead it pipes the module to the remote python interpreter's stdin. Pipelining does not work for python modules involving file transfer (for example: [copy\(../collections/ansible/builtin/copy_module.html#copy-module\)](#), [fetch\(../collections/ansible/builtin/fetch_module.html#fetch-module\)](#), [template\(../collections/ansible/builtin/template_module.html#template-module\)](#)), or for non-python modules.
- Avoid becoming an unprivileged user. Temporary files are protected by UNIX file permissions when you `become` root or do not use `become`. In Ansible 2.1 and above, UNIX file permissions are also secure if you make the connection to the managed machine as root and then use `become` to access an unprivileged account.

⚠ Warning

Although the Solaris ZFS filesystem has filesystem ACLs, the ACLs are not POSIX.1e filesystem acls (they are NFSv4 ACLs instead). Ansible cannot use these ACLs to manage its temp file permissions so you may have to resort to `allow_world_readable_tmpfiles` if the remote machines use ZFS.

Changed in version 2.1.

Ansible makes it hard to unknowingly use `become` insecurely. Starting in Ansible 2.1, Ansible defaults to issuing an error if it cannot execute securely with `become`. If you cannot use pipelining or POSIX ACLs, must connect as an unprivileged user, must use `become` to execute as a different unprivileged user, and decide that your managed nodes are secure enough for the modules you want to run there to be world readable, you can turn on `allow_world_readable_tmpfiles` in the `ansible.cfg` file. Setting `allow_world_readable_tmpfiles` will change this from an error into a warning and allow the task to run as it did prior to 2.1.

Changed in version 2.10.

Ansible 2.10 introduces the above-mentioned `ansible_common_remote_group` fallback. As mentioned above, if enabled, it is used when `remote_user` and `become_user` are both unprivileged users. Refer to the text above for details on when this fallback happens.

⚠ Warning

As mentioned above, if `ansible_common_remote_group` and `allow_world_readable_tmpfiles` are both enabled, it is unlikely that the world-readable fallback will ever trigger, and yet Ansible might still be unable to access the module file. This is because after the group ownership change is successful, Ansible does not fall back any further, and also does not do any check to ensure that the `become_user` is actually a member of the “common

group”. This is a design decision made by the fact that doing such a check would require another round-trip connection to the remote machine, which is a time-expensive operation. Ansible does, however, emit a warning in this case.

Not supported by all connection plugins

Privilege escalation methods must also be supported by the connection plugin used. Most connection plugins will warn if they do not support become. Some will just ignore it as they always run as root (jail, chroot, and so on).

Only one method may be enabled per host

Methods cannot be chained. You cannot use `sudo /bin/su -` to become a user, you need to have privileges to run the command as that user in sudo or be able to su directly to it (the same for pbrun, pfexec or other supported methods).

Privilege escalation must be general

You cannot limit privilege escalation permissions to certain commands. Ansible does not always use a specific command to do something but runs modules (code) from a temporary file name which changes every time. If you have `‘/sbin/service’` or `‘/bin/chmod’` as the allowed commands this will fail with ansible as those paths won’t match with the temporary file that Ansible creates to run the module. If you have security rules that constrain your sudo/pbrun/doas environment to running specific command paths only, use Ansible from a special account that does not have this constraint, or use AWX or the [Red Hat Ansible Automation Platform \(../reference_appendices/tower.html#ansible-platform\)](#) to manage indirect access to SSH credentials.

May not access environment variables populated by pamd systemd

For most Linux distributions using `systemd` as their init, the default methods used by `become` do not open a new “session”, in the sense of systemd. Because the `pam_systemd` module will not fully initialize a new session, you might have surprises compared to a normal session opened through ssh: some environment variables set by `pam_systemd`, most notably `XDG_RUNTIME_DIR`, are not populated for the new user and instead inherited or just emptied.

This might cause trouble when trying to invoke systemd commands that depend on `XDG_RUNTIME_DIR` to access the bus:

```
$ echo $XDG_RUNTIME_DIR

$ systemctl --user status
Failed to connect to bus: Permission denied
```

To force `become` to open a new systemd session that goes through `pam_systemd`, you can use `become_method: machinectl`.

For more information, see [this systemd issue](https://github.com/systemd/systemd/issues/825#issuecomment-127917622) (<https://github.com/systemd/systemd/issues/825#issuecomment-127917622>).

Become and network automation

As of version 2.6, Ansible supports `become` for privilege escalation (entering `enable` mode or privileged EXEC mode) on all Ansible-maintained network platforms that support `enable` mode. Using `become` replaces the `authorize` and `auth_pass` options in a `provider` dictionary.

You must set the connection type to either `connection: ansible.netcommon.network_cli` or `connection: ansible.netcommon.httpapi` to use `become` for privilege escalation on network devices. Check the [Platform Options \(../network/user_guide/platform_index.html#platform-options\)](https://docs.ansible.com/ansible/latest/network/user_guide/platform_index.html#platform-options) documentation for details.

You can use escalated privileges on only the specific tasks that need them, on an entire play, or on all plays. Adding `become: yes` and `become_method: enable` instructs Ansible to enter `enable` mode before executing the task, play, or playbook where those parameters are set.

If you see this error message, the task that generated it requires `enable` mode to succeed:

```
Invalid input (privileged mode required)
```

To set `enable` mode for a specific task, add `become` at the task level:

```
- name: Gather facts (eos)
  arista.eos.eos_facts:
    gather_subset:
      - "!hardware"
  become: yes
  become_method: enable
```

To set enable mode for all tasks in a single play, add `become` at the play level:

```
- hosts: eos-switches
  become: yes
  become_method: enable
  tasks:
    - name: Gather facts (eos)
      arista.eos.eos_facts:
        gather_subset:
          - "!hardware"
```

Setting enable mode for all tasks

Often you wish for all tasks in all plays to run using privilege mode, that is best achieved by using `group_vars`:

`group_vars/eos.yml`

```
ansible_connection: ansible.netcommon.network_cli
ansible_network_os: arista.eos.eos
ansible_user: myuser
ansible_become: yes
ansible_become_method: enable
```

Passwords for enable mode

If you need a password to enter `enable` mode, you can specify it in one of two ways:

- providing the `--ask-become-pass` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-K](https://cli/ansible-playbook.html#cmdoption-ansible-playbook-K)) command line option
- setting the `ansible_become_password` connection variable

⚠ Warning

As a reminder passwords should never be stored in plain text. For information on encrypting your passwords and other secrets with Ansible Vault, see [Encrypting content with Ansible Vault \(vault.html#vault\)](#).

authorize and auth_pass

Ansible still supports `enable` mode with `connection: local` for legacy network playbooks. To enter `enable` mode with `connection: local`, use the module options `authorize` and `auth_pass`:

```
- hosts: eos-switches
  ansible_connection: local
  tasks:
    - name: Gather facts (eos)
      eos_facts:
        gather_subset:
          - "!hardware"
      provider:
        authorize: yes
        auth_pass: "{{ secret_auth_pass }}"
```

We recommend updating your playbooks to use `become` for network-device `enable` mode consistently. The use of `authorize` and of `provider` dictionaries will be deprecated in future. Check the [Platform Options \(./network/user_guide/platform_index.html#platform-options\)](#) and [Network modules \(https://docs.ansible.com/ansible/2.9/modules/list_of_network_modules.html#network-modules\)](#) documentation for details.

Become and Windows

Since Ansible 2.3, `become` can be used on Windows hosts through the `runas` method. Become on Windows uses the same inventory setup and invocation arguments as `become` on a non-Windows host, so the setup and variable names are the same as what is defined in this document.

While `become` can be used to assume the identity of another user, there are other uses for it with Windows hosts. One important use is to bypass some of the limitations that are imposed when running on WinRM, such as constrained network delegation or accessing forbidden system calls like the WUA API. You can use `become` with the same user as `ansible_user` to bypass these limitations and run commands that are not normally accessible in a WinRM session.

Administrative rights

Many tasks in Windows require administrative privileges to complete. When using the `runas` become method, Ansible will attempt to run the module with the full privileges that are available to the remote user. If it fails to elevate the user token, it will continue to use the limited token during execution.

A user must have the `SeDebugPrivilege` to run a become process with elevated privileges. This privilege is assigned to Administrators by default. If the debug privilege is not available, the become process will run with a limited set of privileges and groups.

To determine the type of token that Ansible was able to get, run the following task:

```
- Check my user name
ansible.windows.win_whoami:
become: yes
```

The output will look something similar to the below:


```

ok: [windows] => {
  "account": {
    "account_name": "vagrant-domain",
    "domain_name": "DOMAIN",
    "sid": "S-1-5-21-3088887838-4058132883-1884671576-1105",
    "type": "User"
  },
  "authentication_package": "Kerberos",
  "changed": false,
  "dns_domain_name": "DOMAIN.LOCAL",
  "groups": [
    {
      "account_name": "Administrators",
      "attributes": [
        "Mandatory",
        "Enabled by default",
        "Enabled",
        "Owner"
      ],
      "domain_name": "BUILTIN",
      "sid": "S-1-5-32-544",
      "type": "Alias"
    },
    {
      "account_name": "INTERACTIVE",
      "attributes": [
        "Mandatory",
        "Enabled by default",
        "Enabled"
      ],
      "domain_name": "NT AUTHORITY",
      "sid": "S-1-5-4",
      "type": "WellKnownGroup"
    }
  ],
  "impersonation_level": "SecurityAnonymous",
  "label": {
    "account_name": "High Mandatory Level",
    "domain_name": "Mandatory Label",
    "sid": "S-1-16-12288",
    "type": "Label"
  },
  "login_domain": "DOMAIN",
  "login_time": "2018-11-18T20:35:01.9696884+00:00",
  "logon_id": 114196830,
  "logon_server": "DC01",
  "logon_type": "Interactive",
  "privileges": {
    "SeBackupPrivilege": "disabled",
    "SeChangeNotifyPrivilege": "enabled-by-default",
    "SeCreateGlobalPrivilege": "enabled-by-default",
    "SeCreatePagefilePrivilege": "disabled",
    "SeCreateSymbolicLinkPrivilege": "disabled",
    "SeDebugPrivilege": "enabled",
    "SeDelegateSessionUserImpersonatePrivilege": "disabled",
    "SeImpersonatePrivilege": "enabled-by-default",
    "SeIncreaseBasePriorityPrivilege": "disabled",
    "SeIncreaseQuotaPrivilege": "disabled",
    "SeIncreaseWorkingSetPrivilege": "disabled",
    "SeLoadDriverPrivilege": "disabled",
    "SeManageVolumePrivilege": "disabled",
    "SeProfileSingleProcessPrivilege": "disabled",
    "SeRemoteShutdownPrivilege": "disabled",

```

```

    "SeRestorePrivilege": "disabled",
    "SeSecurityPrivilege": "disabled",
    "SeShutdownPrivilege": "disabled",
    "SeSystemEnvironmentPrivilege": "disabled",
    "SeSystemProfilePrivilege": "disabled",
    "SeSystemtimePrivilege": "disabled",
    "SeTakeOwnershipPrivilege": "disabled",
    "SeTimeZonePrivilege": "disabled",
    "SeUndockPrivilege": "disabled"
  },
  "rights": [
    "SeNetworkLogonRight",
    "SeBatchLogonRight",
    "SeInteractiveLogonRight",
    "SeRemoteInteractiveLogonRight"
  ],
  "token_type": "TokenPrimary",
  "upn": "vagrant-domain@DOMAIN.LOCAL",
  "user_flags": []
}

```

Under the `label` key, the `account_name` entry determines whether the user has Administrative rights. Here are the labels that can be returned and what they represent:

- `Medium`: Ansible failed to get an elevated token and ran under a limited token. Only a subset of the privileges assigned to user are available during the module execution and the user does not have administrative rights.
- `High`: An elevated token was used and all the privileges assigned to the user are available during the module execution.
- `System`: The `NT AUTHORITY\System` account is used and has the highest level of privileges available.

The output will also show the list of privileges that have been granted to the user. When the privilege value is `disabled`, the privilege is assigned to the logon token but has not been enabled. In most scenarios these privileges are automatically enabled when required.

If running on a version of Ansible that is older than 2.5 or the normal `runas` escalation process fails, an elevated token can be retrieved by:

- Set the `become_user` to `System` which has full control over the operating system.
- Grant `SeTcbPrivilege` to the user Ansible connects with on WinRM. `SeTcbPrivilege` is a high-level privilege that grants full control over the operating system. No user is given this privilege by default, and care should be taken if you grant this privilege to a user or group. For more information on this privilege, please see [Act as part of the operating system \(https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn221957\(v=ws.11\)\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/dn221957(v=ws.11)). You can use the below task to set this privilege on a Windows host:

```
- name: grant the ansible user the SeTcbPrivilege right
  ansible.windows.win_user_right:
    name: SeTcbPrivilege
    users: '{{ansible_user}}'
    action: add
```

- Turn UAC off on the host and reboot before trying to become the user. UAC is a security protocol that is designed to run accounts with the `least privilege` principle. You can turn UAC off by running the following tasks:

```
- name: turn UAC off
  win_regedit:
    path: HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\system
    name: EnableLUA
    data: 0
    type: dword
    state: present
    register: uac_result

- name: reboot after disabling UAC
  win_reboot:
    when: uac_result is changed
```

❗ Note

Granting the `SeTcbPrivilege` or turning UAC off can cause Windows security vulnerabilities and care should be given if these steps are taken.

Local service accounts

Prior to Ansible version 2.5, `become` only worked on Windows with a local or domain user account. Local service accounts like `System` or `NetworkService` could not be used as `become_user` in these older versions. This restriction has been lifted since the 2.5 release of Ansible. The three service accounts that can be set under `become_user` are:

- System
- NetworkService
- LocalService

Because local service accounts do not have passwords, the `ansible_become_password` parameter is not required and is ignored if specified.

Become without setting a password

As of Ansible 2.8, `become` can be used to become a Windows local or domain account without requiring a password for that account. For this method to work, the following requirements must be met:

- The connection user has the `SeDebugPrivilege` privilege assigned
- The connection user is part of the `BUILTIN\Administrators` group
- The `become_user` has either the `SeBatchLogonRight` or `SeNetworkLogonRight` user right

Using `become` without a password is achieved in one of two different methods:

- Duplicating an existing logon session's token if the account is already logged on
- Using S4U to generate a logon token that is valid on the remote host only

In the first scenario, the `become` process is spawned from another logon of that user account. This could be an existing RDP logon, console logon, but this is not guaranteed to occur all the time. This is similar to the `Run only when user is logged on` option for a Scheduled Task.

In the case where another logon of the `become` account does not exist, S4U is used to create a new logon and run the module through that. This is similar to the `Run whether user is logged on or not` with the `Do not store password` option for a Scheduled Task. In this scenario, the `become` process will not be able to access any network resources like a normal WinRM process.

To make a distinction between using `become` with no password and becoming an account that has no password make sure to keep `ansible_become_password` as undefined or set `ansible_become_password: .`

Note

Because there are no guarantees an existing token will exist for a user when Ansible runs, there's a high chance the `become` process will only have access to local resources. Use `become` with a password if the task needs to access network resources

Accounts without a password

Warning

As a general security best practice, you should avoid allowing accounts without passwords.

Ansible can be used to become a Windows account that does not have a password (like the `Guest` account). To become an account without a password, set up the variables like normal but set `ansible_become_password: ''`.

Before become can work on an account like this, the local policy [Accounts: Limit local account use of blank passwords to console logon only](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/jj852174(v=ws.11)) ([https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/jj852174\(v=ws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/jj852174(v=ws.11))) must be disabled. This can either be done through a Group Policy Object (GPO) or with this Ansible task:

```
- name: allow blank password on become
  ansible.windows.win_regedit:
    path: HKLM:\SYSTEM\CurrentControlSet\Control\Lsa
    name: LimitBlankPasswordUse
    data: 0
    type: dword
    state: present
```

❗ Note

This is only for accounts that do not have a password. You still need to set the account's password under `ansible_become_password` if the `become_user` has a password.

Become flags for Windows

Ansible 2.5 added the `become_flags` parameter to the `runas` become method. This parameter can be set using the `become_flags` task directive or set in Ansible's configuration using `ansible_become_flags`. The two valid values that are initially supported for this parameter are `logon_type` and `logon_flags`.

❗ Note

These flags should only be set when becoming a normal user account, not a local service account like LocalSystem.

The key `logon_type` sets the type of logon operation to perform. The value can be set to one of the following:

- `interactive`: The default logon type. The process will be run under a context that is the same as when running a process locally. This bypasses all WinRM restrictions and is the recommended method to use.
- `batch`: Runs the process under a batch context that is similar to a scheduled task with a password set. This should bypass most WinRM restrictions and is useful if the `become_user` is not allowed to log on interactively.
- `new_credentials`: Runs under the same credentials as the calling user, but outbound connections are run under the context of the `become_user` and `become_password`, similar to `runas.exe /netonly`. The `logon_flags` flag should also be set to `netcredentials_only`.

Use this flag if the process needs to access a network resource (like an SMB share) using a different set of credentials.

- `network` : Runs the process under a network context without any cached credentials. This results in the same type of logon session as running a normal WinRM process without credential delegation, and operates under the same restrictions.
- `network_cleartext` : Like the `network` logon type, but instead caches the credentials so it can access network resources. This is the same type of logon session as running a normal WinRM process with credential delegation.

For more information, see `dwLogonType` (<https://docs.microsoft.com/en-gb/windows/desktop/api/winbase/nf-winbase-logonusera>).

The `logon_flags` key specifies how Windows will log the user on when creating the new process. The value can be set to none or multiple of the following:

- `with_profile` : The default logon flag set. The process will load the user's profile in the `HKEY_USERS` registry key to `HKEY_CURRENT_USER` .
- `netcredentials_only` : The process will use the same token as the caller but will use the `become_user` and `become_password` when accessing a remote resource. This is useful in inter-domain scenarios where there is no trust relationship, and should be used with the `new_credentials` `logon_type` .

By default `logon_flags=with_profile` is set, if the profile should not be loaded set `logon_flags=` or if the profile should be loaded with `netcredentials_only` , set `logon_flags=with_profile,netcredentials_only` .

For more information, see `dwLogonFlags` (<https://docs.microsoft.com/en-gb/windows/desktop/api/winbase/nf-winbase-createprocesswithtokenw>).

Here are some examples of how to use `become_flags` with Windows tasks:

- **name:** copy a file from a fileshare with custom credentials
ansible.windows.win_copy:
src: \\server\share\data\file.txt
dest: C:\temp\file.txt
remote_src: yes
vars:
ansible_become: yes
ansible_become_method: runas
ansible_become_user: DOMAIN\user
ansible_become_password: Password01
ansible_become_flags: logon_type=new_credentials logon_flags=netcredentials_only
- **name:** run a command under a batch logon
ansible.windows.win_whoami:
become: yes
become_flags: logon_type=batch
- **name:** run a command and not load the user profile
ansible.windows.win_whomai:
become: yes
become_flags: logon_flags=

Limitations of become on Windows

- Running a task with `async` and `become` on Windows Server 2008, 2008 R2 and Windows 7 only works when using Ansible 2.7 or newer.
- By default, the become user logs on with an interactive session, so it must have the right to do so on the Windows host. If it does not inherit the `SeAllowLogOnLocally` privilege or inherits the `SeDenyLogOnLocally` privilege, the become process will fail. Either add the privilege or set the `logon_type` flag to change the logon type used.
- Prior to Ansible version 2.3, become only worked when `ansible_winrm_transport` was either `basic` or `credssp`. This restriction has been lifted since the 2.4 release of Ansible for all hosts except Windows Server 2008 (non R2 version).
- The Secondary Logon service `seclogon` must be running to use `ansible_become_method: runas`

📌 See also

Mailing List (<https://groups.google.com/forum/#!forum/ansible-project>)

Questions? Help? Ideas? Stop by the list on Google Groups

Real-time chat ([../community/communication.html#communication-irc](https://community.ansible.com/#communication-irc))

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-galaxy

Perform various Role and Collection related operations.

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Actions](#)
 - [collection](#)
 - [collection download](#)
 - [collection init](#)
 - [collection build](#)
 - [collection publish](#)
 - [collection install](#)
 - [collection list](#)
 - [collection verify](#)
 - [role](#)
 - [role init](#)
 - [role remove](#)
 - [role delete](#)
 - [role list](#)
 - [role search](#)
 - [role import](#)
 - [role setup](#)
 - [role info](#)
 - [role install](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis


```
usage: ansible-galaxy [-h] [--version] [-v] TYPE ...
```

Description

command to manage Ansible roles in shared repositories, the default of which is Ansible Galaxy <https://galaxy.ansible.com>.

Common Options

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-h, --help

show this help message and exit

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Actions

collection

Perform the action on an Ansible Galaxy collection. Must be combined with a further action like init/install as listed below.

collection download

--clear-response-cache

Clear the existing server response cache.

--no-cache

Do not use the server response cache.

--pre

Include pre-release versions. Semantic versioning pre-releases are ignored by default

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-n, --no-deps

Don't download collection(s) listed as dependencies.

-p <DOWNLOAD_PATH>, --download-path <DOWNLOAD_PATH>

The directory to download the collections to.

-r <REQUIREMENTS>, --requirements-file <REQUIREMENTS>

A file containing a list of collections to be downloaded.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

collection init

Creates the skeleton framework of a role or collection that complies with the Galaxy metadata format. Requires a role or collection name. The collection name must be in the format `<namespace>.<collection>`.

--collection-skeleton <COLLECTION_SKELETON>

The path to a collection skeleton that the new collection should be based upon.

--init-path <INIT_PATH>

The path in which the skeleton collection will be created. The default is the current working directory.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-f, --force

Force overwriting an existing role or collection

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

collection build

Build an Ansible Galaxy collection artifact that can be stored in a central repository like Ansible Galaxy. By default, this command builds from the current working directory. You can optionally pass in the collection input path (where the `galaxy.yml` file is).

--output-path <OUTPUT_PATH>

The path in which the collection is built to. The default is the current working directory.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-f, --force

Force overwriting an existing role or collection

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

collection publish

Publish a collection into Ansible Galaxy. Requires the path to the collection tarball to publish.

--import-timeout <IMPORT_TIMEOUT>

The time to wait for the collection import process to finish.

--no-wait

Don't wait for import validation results.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

collection install

--clear-response-cache

Clear the existing server response cache.

--force-with-deps

Force overwriting an existing collection and its dependencies.

--no-cache

Do not use the server response cache.

--pre

Include pre-release versions. Semantic versioning pre-releases are ignored by default

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-U, --upgrade

Upgrade installed collection artifacts. This will also update dependencies unless --no-deps is provided

-c, --ignore-certs

Ignore SSL certificate validation errors.

-f, --force

Force overwriting an existing role or collection

-i, --ignore-errors

Ignore errors during installation and continue with the next specified collection. This will not ignore dependency conflict errors.

-n, --no-deps

Don't download collections listed as dependencies.

-p <COLLECTIONS_PATH>, --collections-path <COLLECTIONS_PATH>

The path to the directory containing your collections.

-r <REQUIREMENTS>, --requirements-file <REQUIREMENTS>

A file containing a list of collections to be installed.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

collection list

List installed collections or roles

--format <OUTPUT_FORMAT>

Format to display the list of collections in.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at
<https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-p, --collections-path

One or more directories to search for collections in addition to the default COLLECTIONS_PATHS. Separate multiple paths with ':

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

collection verify

--offline

Validate collection integrity locally without contacting server for canonical manifest hash.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-i, --ignore-errors

Ignore errors during verification and continue with the next specified collection.

-p, --collections-path

One or more directories to search for collections in addition to the default COLLECTIONS_PATHS. Separate multiple paths with '·'.

-r <REQUIREMENTS>, --requirements-file <REQUIREMENTS>

A file containing a list of collections to be verified.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role

Perform the action on an Ansible Galaxy role. Must be combined with a further action like delete/install/init as listed below.

role init

Creates the skeleton framework of a role or collection that complies with the Galaxy metadata format. Requires a role or collection name. The collection name must be in the format `<namespace>.<collection>`.

--init-path <INIT_PATH>

The path in which the skeleton role will be created. The default is the current working directory.

--offline

Don't query the galaxy API when creating roles

--role-skeleton <ROLE_SKELETON>

The path to a role skeleton that the new role should be based upon.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

--type <ROLE_TYPE>

Initialize using an alternate role type. Valid types include: 'container', 'apb' and 'network'.

-c, --ignore-certs

Ignore SSL certificate validation errors.

-f, --force

Force overwriting an existing role or collection

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role remove

removes the list of roles passed as arguments from the local system.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-p, --roles-path

The path to the directory containing your roles. The default is the first writable one configured via DEFAULT_ROLES_PATH:

~/ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role delete

Search this site

Delete a role from Ansible Galaxy.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role list

List installed collections or roles

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at <https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-p, --roles-path

The path to the directory containing your roles. The default is the first writable one configured via DEFAULT_ROLES_PATH:

~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role search

searches for roles on the Ansible Galaxy server

--author <AUTHOR>

GitHub username

--galaxy-tags <GALAXY_TAGS>

list of galaxy tags to filter by

--platforms <PLATFORMS>

list of OS platforms to filter by

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at

<https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role import

used to import a role into Ansible Galaxy

--branch <REFERENCE>

The name of a branch to import. Defaults to the repository's default branch (usually master)

--no-wait

Don't wait for import results.

--role-name <ROLE_NAME>

The name the role should have, if different than the repo name

--status

Check the status of the most recent import request for given github_user/github_repo.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at

<https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role setup

Setup an integration from Github or Travis for Ansible Galaxy roles

--list

List all of your integrations.

--remove <REMOVE_ID>

Remove the integration matching the provided ID value. Use --list to see ID values.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at
<https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-p, --roles-path

The path to the directory containing your roles. The default is the first writable one configured via DEFAULT_ROLES_PATH:

~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role info

prints out detailed information about an installed role as well as info available from the galaxy API.

--offline

Don't query the galaxy API when creating roles

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at
<https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-p, --roles-path

The path to the directory containing your roles. The default is the first writable one configured via DEFAULT_ROLES_PATH:

~/ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles

-s <API_SERVER>, --server <API_SERVER>

The Galaxy API server URL

role install

--force-with-deps

Force overwriting an existing role and its dependencies.

--token <API_KEY>, --api-key <API_KEY>

The Ansible Galaxy API key which can be found at

<https://galaxy.ansible.com/me/preferences> (<https://galaxy.ansible.com/me/preferences>).

-c, --ignore-certs

Ignore SSL certificate validation errors.

-f, --force

Force overwriting an existing role or collection

-g, --keep-scm-meta

Use tar instead of the scm archive option when packaging the role.

-i, --ignore-errors

Ignore errors and continue with the next specified role.

-n, --no-deps

Don't download roles listed as dependencies.

-p, --roles-path

The path to the directory containing your roles. The default is the first writable one configured via `DEFAULT_ROLES_PATH`:

`~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles`

`-r <REQUIREMENTS>, --role-file <REQUIREMENTS>`

A file containing a list of roles to be installed.

`-s <API_SERVER>, --server <API_SERVER>`

The Galaxy API server URL

Environment

The following environment variables may be specified.

`ANSIBLE_CONFIG` ([../reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in `ansible.cfg`

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the `AUTHORS` file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Loops

Ansible offers the `loop`, `with_<lookup>`, and `until` keywords to execute a task multiple times. Examples of commonly-used loops include changing ownership on several files and/or directories with the [file module \(../collections/ansible/builtin/file_module.html#file-module\)](#), creating multiple users with the [user module \(../collections/ansible/builtin/user_module.html#user-module\)](#), and repeating a polling step until a certain result is reached.

❗ Note

- We added `loop` in Ansible 2.5. It is not yet a full replacement for `with_<lookup>`, but we recommend it for most use cases.
- We have not deprecated the use of `with_<lookup>` - that syntax will still be valid for the foreseeable future.
- We are looking to improve `loop` syntax - watch this page and the [changelog \(https://github.com/ansible/ansible/tree/devel/changelogs\)](#) for updates.
- [Comparing `loop` and `with_*`](#)
- [Standard loops](#)
 - [Iterating over a simple list](#)
 - [Iterating over a list of hashes](#)
 - [Iterating over a dictionary](#)
- [Registering variables with a loop](#)
- [Complex loops](#)
 - [Iterating over nested lists](#)
 - [Retrying a task until a condition is met](#)
 - [Looping over inventory](#)
- [Ensuring list input for `loop`: using `query` rather than `lookup`](#)
- [Adding controls to loops](#)
 - [Limiting loop output with `label`](#)
 - [Pausing within a loop](#)
 - [Tracking progress through a loop with `index_var`](#)
 - [Defining inner and outer variable names with `loop_var`](#)

- [Extended loop variables](#)
- [Accessing the name of your loop_var](#)
- [Migrating from with X to loop](#)
 - [with_list](#)
 - [with_items](#)
 - [with_indexed_items](#)
 - [with_flattened](#)
 - [with_together](#)
 - [with_dict](#)
 - [with_sequence](#)
 - [with_subelements](#)
 - [with_nested/with_cartesian](#)
 - [with_random_choice](#)

Comparing `loop` and `with *`

- The `with_<lookup>` keywords rely on [Lookup plugins \(../plugins/lookup.html#lookup-plugins\)](#) - even `items` is a lookup.
- The `loop` keyword is equivalent to `with_list`, and is the best choice for simple loops.
- The `loop` keyword will not accept a string as input, see [Ensuring list input for loop: using query rather than lookup](#).
- Generally speaking, any use of `with_*` covered in [Migrating from with X to loop](#) can be updated to use `loop`.
- Be careful when changing `with_items` to `loop`, as `with_items` performed implicit single-level flattening. You may need to use `flatten(1)` with `loop` to match the exact outcome. For example, to get the same output as:

```
with_items:
- 1
- [2,3]
- 4
```

you would need

```
loop: "{{ [1, [2, 3], 4] | flatten(1) }}"
```

- Any `with_*` statement that requires using `lookup` within a loop should not be converted to use the `loop` keyword. For example, instead of doing:

```
loop: "{{ lookup('fileglob', '*.txt', wantlist=True) }}"
```

it's cleaner to keep

```
with_fileglob: '*.txt'
```

Standard loops

Iterating over a simple list

Repeated tasks can be written as standard loops over a simple list of strings. You can define the list directly in the task.

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
    - testuser1
    - testuser2
```

You can define the list in a variables file, or in the 'vars' section of your play, then refer to the name of the list in the task.

```
loop: "{{ somelist }}"
```

Either of these examples would be the equivalent of

```
- name: Add user testuser1
  ansible.builtin.user:
    name: "testuser1"
    state: present
    groups: "wheel"

- name: Add user testuser2
  ansible.builtin.user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

You can pass a list directly to a parameter for some plugins. Most of the packaging modules, like [yum](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/yum_module.html#yum-module) ([../collections/ansible/builtin/yum_module.html#yum-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/yum_module.html#yum-module)) and [apt](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html#apt-module) ([../collections/ansible/builtin/apt_module.html#apt-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html#apt-module)), have this capability. When available, passing the list to a parameter is better than looping over the task. For example

Search this site

```
- name: Optimal yum
  ansible.builtin.yum:
    name: "{{ list_of_packages }}"
    state: present

- name: Non-optimal yum, slower and may cause issues with interdependencies
  ansible.builtin.yum:
    name: "{{ item }}"
    state: present
  loop: "{{ list_of_packages }}"
```

Check the [module documentation](#)

(https://docs.ansible.com/ansible/2.9/modules/modules_by_category.html#modules-by-category) to see if you can pass a list to any particular module's parameter(s).

Iterating over a list of hashes

If you have a list of hashes, you can reference subkeys in a loop. For example:

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

When combining [conditionals](#) ([playbooks_conditionals.html#playbooks-conditionals](#)) with a loop, the `when:` statement is processed separately for each item. See [Basic conditionals with when](#) ([playbooks_conditionals.html#the-when-statement](#)) for examples.

Iterating over a dictionary

To loop over a dict, use the [dict2items](#) ([playbooks_filters.html#dict-filter](#)):

```
- name: Using dict2items
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ tag_data | dict2items }}"
  vars:
    tag_data:
      Environment: dev
      Application: payment
```

Here, we are iterating over `tag_data` and printing the key and the value from it.

Registering variables with a loop

You can register the output of a loop as a variable. For example

```
- name: Register loop output as a variable
  ansible.builtin.shell: "echo {{ item }}"
  loop:
    - "one"
    - "two"
  register: echo
```

When you use `register` with a loop, the data structure placed in the variable will contain a `results` attribute that is a list of all responses from the module. This differs from the data structure returned when using `register` without a loop.

```
{
  "changed": true,
  "msg": "All items completed",
  "results": [
    {
      "changed": true,
      "cmd": "echo \"one\" ",
      "delta": "0:00:00.003110",
      "end": "2013-12-19 12:00:05.187153",
      "invocation": {
        "module_args": "echo \"one\"",
        "module_name": "shell"
      },
      "item": "one",
      "rc": 0,
      "start": "2013-12-19 12:00:05.184043",
      "stderr": "",
      "stdout": "one"
    },
    {
      "changed": true,
      "cmd": "echo \"two\" ",
      "delta": "0:00:00.002920",
      "end": "2013-12-19 12:00:05.245502",
      "invocation": {
        "module_args": "echo \"two\"",
        "module_name": "shell"
      },
      "item": "two",
      "rc": 0,
      "start": "2013-12-19 12:00:05.242582",
      "stderr": "",
      "stdout": "two"
    }
  ]
}
```

Subsequent loops over the registered variable to inspect the results may look like [Search this site](#)

```
- name: Fail if return code is not 0
  ansible.builtin.fail:
    msg: "The command ({{ item.cmd }}) did not have a 0 return code"
  when: item.rc != 0
  loop: "{{ echo.results }}"
```

During iteration, the result of the current item will be placed in the variable.

```
- name: Place the result of the current item in the variable
  ansible.builtin.shell: echo "{{ item }}"
  loop:
    - one
    - two
  register: echo
  changed_when: echo.stdout != "one"
```

Complex loops

Iterating over nested lists

You can use Jinja2 expressions to iterate over complex lists. For example, a loop can combine nested lists.

```
- name: Give users access to multiple databases
  community.mysql.mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  loop: "{{ ['alice', 'bob'] | product(['clientdb', 'employeedb', 'providerdb']) | list
  }}"
```

Retrying a task until a condition is met

New in version 1.4.

You can use the `until` keyword to retry a task until a certain condition is met. Here's an example:

```
- name: Retry a task until a certain condition is met
  ansible.builtin.shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

This task runs up to 5 times with a delay of 10 seconds between each attempt. If the result of any attempt has “all systems go” in its stdout, the task succeeds. The default value for “retries” is 3 and “delay” is 5.

To see the results of individual retries, run the play with `-vv`.

When you run a task with `until` and register the result as a variable, the registered variable will include a key called “attempts”, which records the number of the retries for the task.

❗ Note

You must set the `until` parameter if you want a task to retry. If `until` is not defined, the value for the `retries` parameter is forced to 1.

Looping over inventory

To loop over your inventory, or just a subset of it, you can use a regular `loop` with the `ansible_play_batch` or `groups` variables.

```
- name: Show all the hosts in the inventory
  ansible.builtin.debug:
    msg: "{{ item }}"
    loop: "{{ groups['all'] }}"

- name: Show all the hosts in the current play
  ansible.builtin.debug:
    msg: "{{ item }}"
    loop: "{{ ansible_play_batch }}"
```

There is also a specific lookup plugin `inventory_hostnames` that can be used like this

```
- name: Show all the hosts in the inventory
  ansible.builtin.debug:
    msg: "{{ item }}"
    loop: "{{ query('inventory_hostnames', 'all') }}"

- name: Show all the hosts matching the pattern, ie all but the group www
  ansible.builtin.debug:
    msg: "{{ item }}"
    loop: "{{ query('inventory_hostnames', 'all:!www') }}"
```

More information on the patterns can be found in [Patterns: targeting hosts and groups \(intro_patterns.html#intro-patterns\)](#).

Ensuring list input for `loop`: using `query` rather than `lookup`

The `loop` keyword requires a list as input, but the `lookup` keyword returns a string of comma-separated values by default. Ansible 2.5 introduced a new Jinja2 function named `query(..plugins/lookup.html#query)`, that always returns a list, offering a simpler interface and more predictable output from lookup plugins when using the `loop` keyword.

You can force `lookup` to return a list to `loop` by using `wantlist=True`, or you can use `query` instead.

The following two examples do the same thing.

```
loop: "{{ query('inventory_hostnames', 'all') }}"

loop: "{{ lookup('inventory_hostnames', 'all', wantlist=True) }}"
```

Adding controls to loops

New in version 2.1.

The `loop_control` keyword lets you manage your loops in useful ways.

Limiting loop output with `label`

New in version 2.2.

When looping over complex data structures, the console output of your task can be enormous. To limit the displayed output, use the `label` directive with `loop_control`.

```
- name: Create servers
  digital_ocean:
    name: "{{ item.name }}"
    state: present
  loop:
    - name: server1
      disks: 3gb
      ram: 15Gb
      network:
        nic01: 100Gb
        nic02: 10Gb
      ...
  loop_control:
    label: "{{ item.name }}"
```

The output of this task will display just the `name` field for each `item` instead of the entire contents of the multi-line `{{ item }}` variable.

❗ Note

This is for making console output more readable, not protecting sensitive data. If there is sensitive data in `loop`, set `no_log: yes` on the task to prevent disclosure.

Pausing within a loop

New in version 2.2.

To control the time (in seconds) between the execution of each item in a task loop, use the `pause` directive with `loop_control`.

```
# main.yml
- name: Create servers, pause 3s before creating next
  community.digitalocean.digital_ocean:
    name: "{{ item }}"
    state: present
  loop:
    - server1
    - server2
  loop_control:
    pause: 3
```

Tracking progress through a loop with `index_var`

New in version 2.5.

To keep track of where you are in a loop, use the `index_var` directive with `loop_control`. This directive specifies a variable name to contain the current loop index.

```
- name: Count our fruit
  ansible.builtin.debug:
    msg: "{{ item }}" with index {{ my_idx }}
  loop:
    - apple
    - banana
    - pear
  loop_control:
    index_var: my_idx
```

❗ Note

`index_var` is 0 indexed.

Defining inner and outer variable names with `loop_var`

New in version 2.1.

You can nest two looping tasks using `include_tasks`. However, by default Ansible sets the loop variable `item` for each loop. This means the inner, nested loop will overwrite the value of `item` from the outer loop. You can specify the name of the variable for each loop using `loop_var` with `loop_control`.

```
# main.yml
- include_tasks: inner.yml
  loop:
    - 1
    - 2
    - 3
  loop_control:
    loop_var: outer_item

# inner.yml
- name: Print outer and inner items
  ansible.builtin.debug:
    msg: "outer item={{ outer_item }} inner item={{ item }}"
  loop:
    - a
    - b
    - c
```

Note

If Ansible detects that the current loop is using a variable which has already been defined, it will raise an error to fail the task.

Extended loop variables

New in version 2.8.

As of Ansible 2.8 you can get extended loop information using the `extended` option to loop control. This option will expose the following information.

Variable	Description
<code>ansible_loop.allitems</code>	The list of all items in the loop
<code>ansible_loop.index</code>	The current iteration of the loop. (1 indexed)
<code>ansible_loop.index0</code>	The current iteration of the loop. (0 indexed)
<code>ansible_loop.revindex</code>	The number of iterations from the end of the loop (1 indexed)
<code>ansible_loop.revindex0</code>	The number of iterations from the end of the loop (0 indexed)

<code>ansible_loop.first</code>	<code>True</code> if first iteration
<code>ansible_loop.last</code>	<code>True</code> if last iteration
<code>ansible_loop.length</code>	The number of items in the loop
<code>ansible_loop.previtem</code>	The item from the previous iteration of the loop. Undefined during the first iteration
<code>ansible_loop.nextitem</code>	The item from the following iteration of the loop. Undefined during the last iteration

```
loop_control:
  extended: yes
```

❗ Note

When using `loop_control.extended` more memory will be utilized on the control node. This is a result of `ansible_loop.allitems` containing a reference to the full loop data for every loop. When serializing the results for display in callback plugins within the main ansible process, these references may be dereferenced causing memory usage to increase.

Accessing the name of your loop_var

New in version 2.8.

As of Ansible 2.8 you can get the name of the value provided to `loop_control.loop_var` using the `ansible_loop_var` variable

For role authors, writing roles that allow loops, instead of dictating the required `loop_var` value, you can gather the value via the following

```
"{{ lookup('vars', ansible_loop_var) }}"
```

Migrating from with_X to loop

In most cases, loops work best with the `loop` keyword instead of `with_X` style loops. The `loop` syntax is usually best expressed using filters instead of more complex use of `query` or `lookup`.

These examples show how to convert many common `with_` style loops to `loop` and filters.

with_list

`with_list` is directly replaced by `loop`.

```
- name: with_list
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_list:
    - one
    - two

- name: with_list -> loop
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop:
    - one
    - two
```

with items

`with_items` is replaced by `loop` and the `flatten` filter.

```
- name: with_items
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_items: "{{ items }}"

- name: with_items -> loop
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
```

with indexed items

`with_indexed_items` is replaced by `loop`, the `flatten` filter and `loop_control.index_var`.

```
- name: with_indexed_items
  ansible.builtin.debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  with_indexed_items: "{{ items }}"

- name: with_indexed_items -> loop
  ansible.builtin.debug:
    msg: "{{ index }}" - "{{ item }}"
  loop: "{{ items|flatten(levels=1) }}"
  loop_control:
    index_var: index
```

with flattened

`with_flattened` is replaced by `loop` and the `flatten` filter.

```
- name: with_flattened
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_flattened: "{{ items }}"

- name: with_flattened -> loop
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ items|flatten }}"
```

with_together

`with_together` is replaced by `loop` and the `zip` filter.

```
- name: with_together
  ansible.builtin.debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  with_together:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_together -> loop
  ansible.builtin.debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  loop: "{{ list_one|zip(list_two)|list }}"
```

Another example with complex data

```
- name: with_together -> loop
  ansible.builtin.debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}" - "{{ item.2 }}"
  loop: "{{ data[0]|zip(*data[1:])|list }}"
  vars:
    data:
      - ['a', 'b', 'c']
      - ['d', 'e', 'f']
      - ['g', 'h', 'i']
```

with_dict

`with_dict` can be substituted by `loop` and either the `dictsort` or `dict2items` filters.

```
- name: with_dict
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  with_dict: "{{ dictionary }}"

- name: with_dict -> loop (option 1)
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ dictionary|dict2items }}"

- name: with_dict -> loop (option 2)
  ansible.builtin.debug:
    msg: "{{ item.0 }} - {{ item.1 }}"
  loop: "{{ dictionary|dictsort }}"
```

with_sequence

`with_sequence` is replaced by `loop` and the `range` function, and potentially the `format` filter.

```
- name: with_sequence
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_sequence: start=0 end=4 stride=2 format=testuser%02x

- name: with_sequence -> loop
  ansible.builtin.debug:
    msg: "{{ 'testuser%02x' | format(item) }}"
  # range is exclusive of the end point
  loop: "{{ range(0, 4 + 1, 2)|list }}"
```

with_subelements

`with_subelements` is replaced by `loop` and the `subelements` filter.

```
- name: with_subelements
  ansible.builtin.debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  with_subelements:
    - "{{ users }}"
    - mysql.hosts

- name: with_subelements -> loop
  ansible.builtin.debug:
    msg: "{{ item.0.name }} - {{ item.1 }}"
  loop: "{{ users|subelements('mysql.hosts') }}"
```

with_nested/with_cartesian

`with_nested` and `with_cartesian` are replaced by loop and the `product` filter.

```
- name: with_nested
  ansible.builtin.debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  with_nested:
    - "{{ list_one }}"
    - "{{ list_two }}"

- name: with_nested -> loop
  ansible.builtin.debug:
    msg: "{{ item.0 }}" - "{{ item.1 }}"
  loop: "{{ list_one|product(list_two)|list }}"
```

with random choice

`with_random_choice` is replaced by just use of the `random` filter, without need of `loop`.

```
- name: with_random_choice
  ansible.builtin.debug:
    msg: "{{ item }}"
  with_random_choice: "{{ my_list }}"

- name: with_random_choice -> loop (No loop is needed here)
  ansible.builtin.debug:
    msg: "{{ my_list|random }}"
  tags: random
```

❗ See also

[Intro to playbooks \(playbooks_intro.html#about-playbooks\)](#)

An introduction to playbooks

[Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#)

Playbook organization by roles

[Tips and tricks \(playbooks_best_practices.html#playbooks-best-practices\)](#)

Tips and tricks for playbooks

[Conditionals \(playbooks_conditionals.html#playbooks-conditionals\)](#)

Conditional statements in playbooks

[Using Variables \(playbooks_variables.html#playbooks-variables\)](#)

All about variables

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

Search this site

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-inventory

None

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-inventory [-h] [--version] [-v] [-i INVENTORY]
                        [--vault-id VAULT_IDS]
                        [--ask-vault-password | --vault-password-file
VAULT_PASSWORD_FILES]
                        [--playbook-dir BASEDIR] [-e EXTRA_VARS] [--list]
                        [--host HOST] [--graph] [-y] [--toml] [--vars]
                        [--export] [--output OUTPUT_FILE]
                        [host|group]
```

Description

used to display or dump the configured inventory as Ansible sees it

Common Options

--ask-vault-password, --ask-vault-pass

ask for vault password

Search this site

--export

When doing an `-list`, represent in a way that is optimized for export, not as an accurate representation of how Ansible has processed it

--graph

create inventory graph, if supplying pattern it must be a valid group name

--host <HOST>

Output specific host info, works as inventory script

--list

Output all hosts info, works as inventory script

--list-hosts

==SUPPRESS==

--output <OUTPUT_FILE>

When doing `-list`, send the inventory to a file instead of to the screen

--playbook-dir <BASEDIR>

Since this tool does not use playbooks, use this as a substitute playbook directory. This sets the relative path for many features including `roles/` `group_vars/` etc.

--toml

Use TOML format instead of default JSON, ignored for `-graph`

--vars

Add vars to graph display, ignored unless used with `-graph`

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-e, --extra-vars

set additional variables as key=value or YAML/JSON, if filename prepend with @

-h, --help

show this help message and exit

-i, --inventory, --inventory-file

specify inventory host path or comma separated host list. --inventory-file is deprecated

-l, --limit

==SUPPRESS==

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

-y, --yaml

Use YAML format instead of default JSON, ignored for --graph

Environment

The following environment variables may be specified.

ANSIBLE_CONFIG ([./reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in ansible.cfg

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Controlling where tasks run: delegation and local actions

By default Ansible gathers facts and executes all tasks on the machines that match the `hosts` line of your playbook. This page shows you how to delegate tasks to a different machine or group, delegate facts to specific machines or groups, or run an entire playbook locally. Using these approaches, you can manage inter-related environments precisely and efficiently. For example, when updating your webservers, you might need to remove them from a load-balanced pool temporarily. You cannot perform this task on the webservers themselves. By delegating the task to localhost, you keep all the tasks within the same play.

- [Tasks that cannot be delegated](#)
- [Delegating tasks](#)
- [Delegation and parallel execution](#)
- [Delegating facts](#)
- [Local playbooks](#)

Tasks that cannot be delegated

Some tasks always execute on the controller. These tasks, including `include`, `add_host`, and `debug`, cannot be delegated.

Delegating tasks

If you want to perform a task on one host with reference to other hosts, use the `delegate_to` keyword on a task. This is ideal for managing nodes in a load balanced pool or for controlling outage windows. You can use delegation with the [serial](#) ([playbooks_strategies.html#rolling-update-batch-size](#)) keyword to control the number of hosts executing at one time:

```

---
- hosts: webserver
  serial: 5

  tasks:
    - name: Take out of load balancer pool
      ansible.builtin.command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1

    - name: Actual steps would go here
      ansible.builtin.yum:
        name: acme-web-stack
        state: latest

    - name: Add back to load balancer pool
      ansible.builtin.command: /usr/bin/add_back_to_pool {{ inventory_hostname }}
      delegate_to: 127.0.0.1

```

The first and third tasks in this play run on 127.0.0.1, which is the machine running Ansible. There is also a shorthand syntax that you can use on a per-task basis: `local_action`. Here is the same playbook as above, but using the shorthand syntax for delegating to 127.0.0.1:

```

---
# ...

tasks:
  - name: Take out of load balancer pool
    local_action: ansible.builtin.command /usr/bin/take_out_of_pool {{
inventory_hostname }}

# ...

  - name: Add back to load balancer pool
    local_action: ansible.builtin.command /usr/bin/add_back_to_pool {{
inventory_hostname }}

```

You can use a local action to call 'rsync' to recursively copy files to the managed servers:

```

---
# ...

tasks:
  - name: Recursively copy files from management server to target
    local_action: ansible.builtin.command rsync -a /path/to/files {{
inventory_hostname }}:/path/to/target/

```

Note that you must have passphrase-less SSH keys or an ssh-agent configured for this to work, otherwise rsync asks for a passphrase.

To specify more arguments, use the following syntax:

Search this site

```

---
# ...

tasks:
  - name: Send summary mail
    local_action:
      module: community.general.mail
      subject: "Summary Mail"
      to: "{{ mail_recipient }}"
      body: "{{ mail_body }}"
    run_once: True

```

❗ Note

- The *ansible_host* variable and other connection variables, if present, reflects information about the host a task is delegated to, not the *inventory_hostname*.

❗ Warning

Although you can `delegate_to` a host that does not exist in inventory (by adding IP address, DNS name or whatever requirement the connection plugin has), doing so does not add the host to your inventory and might cause issues. Hosts delegated to in this way do not inherit variables from the “all” group, so variables like connection user and key are missing. If you must `delegate_to` a non-inventory host, use the [add host module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/add_host_module.html#add-host-module) ([../collections/ansible/builtin/add_host_module.html#add-host-module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/add_host_module.html#add-host-module)).

Delegation and parallel execution

By default Ansible tasks are executed in parallel. Delegating a task does not change this and does not handle concurrency issues (multiple forks writing to the same file). Most commonly, users are affected by this when updating a single file on a single delegated to host for all hosts (using the `copy`, `template`, or `lineinfile` modules, for example). They will still operate in parallel forks (default 5) and overwrite each other.

This can be handled in several ways:

```

- name: "handle concurrency with a loop on the hosts with `run_once: true`"
  lineinfile: "<options here>"
  run_once: true
  loop: '{{ ansible_play_hosts_all }}'

```

By using an intermediate play with *serial: 1* or using *throttle: 1* at task level, for more detail see [Controlling playbook execution: strategies and more](https://docs.ansible.com/ansible/latest/playbooks_strategies.html#playbooks-strategies) ([playbooks_strategies.html#playbooks-strategies](https://docs.ansible.com/ansible/latest/playbooks_strategies.html#playbooks-strategies)).

Delegating facts

Delegating Ansible tasks is like delegating tasks in the real world - your groceries belong to you, even if someone else delivers them to your home. Similarly, any facts gathered by a delegated task are assigned by default to the *inventory_hostname* (the current host), not to the host which produced the facts (the delegated to host). To assign gathered facts to the delegated host instead of the current host, set `delegate_facts` to `true`:

```
---
- hosts: app_servers

  tasks:
    - name: Gather facts from db servers
      ansible.builtin.setup:
        delegate_to: "{{ item }}"
        delegate_facts: true
        loop: "{{ groups['dbservers'] }}"
```

This task gathers facts for the machines in the *dbservers* group and assigns the facts to those machines, even though the play targets the *app_servers* group. This way you can lookup `hostvars['dbhost1']['ansible_default_ipv4']['address']` even though *dbservers* were not part of the play, or left out by using `-limit`.

Local playbooks

It may be useful to use a playbook locally on a remote host, rather than by connecting over SSH. This can be useful for assuring the configuration of a system by putting a playbook in a crontab. This may also be used to run a playbook inside an OS installer, such as an Anaconda kickstart.

To run an entire playbook locally, just set the `hosts:` line to `hosts: 127.0.0.1` and then run the playbook like so:

```
ansible-playbook playbook.yml --connection=local
```

Alternatively, a local connection can be used in a single playbook play, even if other plays in the playbook use the default remote connection type:

```
---
- hosts: 127.0.0.1
  connection: local
```

❗ Note

If you set the connection to local and there is no `ansible_python_interpreter` set, modules will run under `/usr/bin/python` and not under `{{ ansible_playbook_python }}`. Be sure to set `ansible_python_interpreter: "{{ ansible_playbook_python }}"` in `host_vars/localhost.yml`, for example. You can avoid this issue by using `local_action` or `delegate_to: localhost` instead.

❗ See also

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[Controlling playbook execution: strategies and more \(playbooks_strategies.html#playbooks-strategies\)](#)

More ways to control how and where Ansible executes

[Ansible Examples on GitHub \(https://github.com/ansible/ansible-examples\)](https://github.com/ansible/ansible-examples)

Many examples of full-stack deployments

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-playbook

Runs Ansible playbooks, executing the defined tasks on the targeted hosts.

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-playbook [-h] [--version] [-v] [--private-key PRIVATE_KEY_FILE]
                        [-u REMOTE_USER] [-c CONNECTION] [-T TIMEOUT]
                        [--ssh-common-args SSH_COMMON_ARGS]
                        [--sftp-extra-args SFTP_EXTRA_ARGS]
                        [--scp-extra-args SCP_EXTRA_ARGS]
                        [--ssh-extra-args SSH_EXTRA_ARGS]
                        [-k | --connection-password-file CONNECTION_PASSWORD_FILE]
                        [--force-handlers] [--flush-cache] [-b]
                        [--become-method BECOME_METHOD]
                        [--become-user BECOME_USER]
                        [-K | --become-password-file BECOME_PASSWORD_FILE]
                        [-t TAGS] [--skip-tags SKIP_TAGS] [-C]
                        [--syntax-check] [-D] [-i INVENTORY] [--list-hosts]
                        [-l SUBSET] [-e EXTRA_VARS] [--vault-id VAULT_IDS]
                        [--ask-vault-password | --vault-password-file
VAULT_PASSWORD_FILES]
                        [-f FORKS] [-M MODULE_PATH] [--list-tasks]
                        [--list-tags] [--step] [--start-at-task START_AT_TASK]
                        playbook [playbook ...]
```

Description

the tool to run *Ansible playbooks*, which are a configuration and multinode deployment system. See the project home page (<https://docs.ansible.com>) for more information.

Common Options

--ask-vault-password, --ask-vault-pass

ask for vault password

--become-method <BECOME_METHOD>

privilege escalation method to use (default=sudo), use *ansible-doc -t become -l* to list valid choices.

--become-password-file <BECOME_PASSWORD_FILE>, --become-pass-file <BECOME_PASSWORD_FILE>

Become password file

--become-user <BECOME_USER>

run operations as this user (default=root)

--connection-password-file <CONNECTION_PASSWORD_FILE>, --conn-pass-file <CONNECTION_PASSWORD_FILE>

Connection password file

--flush-cache

clear the fact cache for every host in inventory

--force-handlers

run handlers even if a task fails

--list-hosts

outputs a list of matching hosts; does not execute anything else

--list-tags

list all available tags

--list-tasks

list all tasks that would be executed

--private-key <PRIVATE_KEY_FILE>, --key-file <PRIVATE_KEY_FILE>

use this file to authenticate the connection

--scp-extra-args <SCP_EXTRA_ARGS>

specify extra arguments to pass to scp only (e.g. -l)

--sftp-extra-args <SFTP_EXTRA_ARGS>

specify extra arguments to pass to sftp only (e.g. -f, -l)

--skip-tags

only run plays and tasks whose tags do not match these values

--ssh-common-args <SSH_COMMON_ARGS>

specify common arguments to pass to sftp/scp/ssh (e.g. ProxyCommand)

--ssh-extra-args <SSH_EXTRA_ARGS>

specify extra arguments to pass to ssh only (e.g. -R)

--start-at-task <START_AT_TASK>

start the playbook at the task matching this name

--step

one-step-at-a-time: confirm each task before running

--syntax-check

perform a syntax check on the playbook, but do not execute it

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-C, --check

don't make any changes; instead, try to predict some of the changes that may occur

-D, --diff

when changing (small) files and templates, show the differences in those files; works great with -check

-K, --ask-become-pass

ask for privilege escalation password

-M, --module-path

prepend colon-separated path(s) to module library
(default=~/.ansible/plugins/modules:/usr/share/ansible/plugins/modules)

-T <TIMEOUT>, --timeout <TIMEOUT>

override the connection timeout in seconds (default=10)

-b, --become

run operations with become (does not imply password prompting)

-c <CONNECTION>, --connection <CONNECTION>

connection type to use (default=smart)

-e, --extra-vars

set additional variables as key=value or YAML/JSON, if filename prepend with @

-f <FORKS>, --forks <FORKS>

specify number of parallel processes to use (default=5)

-h, --help

show this help message and exit

-i, --inventory, --inventory-file

specify inventory host path or comma separated host list. -inventory-file is deprecated

-k, --ask-pass

ask for connection password

-l <SUBSET>, --limit <SUBSET>

further limit selected hosts to an additional pattern

-t, --tags

only run plays and tasks tagged with these values

-u <REMOTE_USER>, --user <REMOTE_USER>

connect as this user (default=None)

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Environment

The following environment variables may be specified.

`ANSIBLE_CONFIG` ([./reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in `ansible.cfg`

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Conditionals

In a playbook, you may want to execute different tasks, or have different goals, depending on the value of a fact (data about the remote system), a variable, or the result of a previous task. You may want the value of some variables to depend on the value of other variables. Or you may want to create additional groups of hosts based on whether the hosts match other criteria. You can do all of these things with conditionals.

Ansible uses Jinja2 [tests \(playbooks tests.html#playbooks-tests\)](#) and [filters \(playbooks filters.html#playbooks-filters\)](#) in conditionals. Ansible supports all the standard tests and filters, and adds some unique ones as well.

Note

There are many options to control execution flow in Ansible. You can find more examples of supported conditionals at

<https://jinja.palletsprojects.com/en/latest/templates/#comparisons>
(<https://jinja.palletsprojects.com/en/latest/templates/#comparisons>).

- [Basic conditionals with `when`](#)
 - [Conditionals based on ansible facts](#)
 - [Conditions based on registered variables](#)
 - [Conditionals based on variables](#)
 - [Using conditionals in loops](#)
 - [Loading custom facts](#)
 - [Conditionals with re-use](#)
 - [Conditionals with imports](#)
 - [Conditionals with includes](#)
 - [Conditionals with roles](#)
 - [Selecting variables, files, or templates based on facts](#)
 - [Selecting variables files based on facts](#)
 - [Selecting files and templates based on facts](#)
- [Commonly-used facts](#)
 - [ansible facts\['distribution'\]](#)
 - [ansible facts\['distribution major version'\]](#)

- `ansible_facts['os_family']`

Basic conditionals with when

The simplest conditional statement applies to a single task. Create the task, then add a `when` statement that applies a test. The `when` clause is a raw Jinja2 expression without double curly braces (see [group_by_module](#) (https://docs.ansible.com/ansible/5/collections/ansible/builtin/group_by_module.html#group-by-module)). When you run the task or playbook, Ansible evaluates the test for all hosts. On any host where the test passes (returns a value of True), Ansible runs that task. For example, if you are installing mysql on multiple machines, some of which have SELinux enabled, you might have a task to configure SELinux to allow mysql to run. You would only want that task to run on machines that have SELinux enabled:

```
tasks:
- name: Configure SELinux to start mysql on any port
  ansible.posix.seboolean:
    name: mysql_connect_any
    state: true
    persistent: yes
  when: ansible_selinux.status == "enabled"
  # all variables can be used directly in conditionals without double curly braces
```

Conditionals based on ansible facts

Often you want to execute or skip a task based on facts. Facts are attributes of individual hosts, including IP address, operating system, the status of a filesystem, and many more. With conditionals based on facts:

- You can install a certain package only when the operating system is a particular version.
- You can skip configuring a firewall on hosts with internal IP addresses.
- You can perform cleanup tasks only when a filesystem is getting full.

See [Commonly-used facts](#) for a list of facts that frequently appear in conditional statements. Not all facts exist for all hosts. For example, the `'lsb_major_release'` fact used in an example below only exists when the `lsb_release` package is installed on the target host. To see what facts are available on your systems, add a debug task to your playbook:

```
- name: Show facts available on the system
  ansible.builtin.debug:
    var: ansible_facts
```

tasks:

- **name:** Shut down Debian flavored systems
ansible.builtin.command: /sbin/shutdown -t now
when: ansible_facts['os_family'] == "Debian"

If you have multiple conditions, you can group them with parentheses:

tasks:

- **name:** Shut down CentOS 6 and Debian 7 systems
ansible.builtin.command: /sbin/shutdown -t now
when: (ansible_facts['distribution'] == "CentOS" and
ansible_facts['distribution_major_version'] == "6") or
(ansible_facts['distribution'] == "Debian" and
ansible_facts['distribution_major_version'] == "7")

You can use logical operators (<https://jinja.palletsprojects.com/en/latest/templates/#logic>) to combine conditions. When you have multiple conditions that all need to be true (that is, a logical `and`), you can specify them as a list:

tasks:

- **name:** Shut down CentOS 6 systems
ansible.builtin.command: /sbin/shutdown -t now
when:
 - ansible_facts['distribution'] == "CentOS"
 - ansible_facts['distribution_major_version'] == "6"

If a fact or variable is a string, and you need to run a mathematical comparison on it, use a filter to ensure that Ansible reads the value as an integer:

tasks:

- **ansible.builtin.shell:** echo "only on Red Hat 6, derivatives, and later"
when: ansible_facts['os_family'] == "RedHat" and ansible_facts['lsb']
['major_release'] | int >= 6

Conditions based on registered variables

Often in a playbook you want to execute or skip a task based on the outcome of an earlier task. For example, you might want to configure a service after it is upgraded by an earlier task. To create a conditional based on a registered variable:

1. Register the outcome of the earlier task as a variable.
2. Create a conditional test based on the registered variable.

You create the name of the registered variable using the `register` keyword. A registered variable always contains the status of the task that created it as well as any output that task generated. You can use registered variables in templates and action lines as well as in conditional `when` statements. You can access the string contents of the registered variable using `variable.stdout`. For example:

```
- name: Test play
  hosts: all

  tasks:

    - name: Register a variable
      ansible.builtin.shell: cat /etc/motd
      register: motd_contents

    - name: Use the variable in conditional statement
      ansible.builtin.shell: echo "motd contains the word hi"
      when: motd_contents.stdout.find('hi') != -1
```

You can use registered results in the loop of a task if the variable is a list. If the variable is not a list, you can convert it into a list, with either `stdout_lines` or with `variable.stdout.split()`. You can also split the lines by other fields:

```
- name: Registered variable usage as a loop list
  hosts: all
  tasks:

    - name: Retrieve the list of home directories
      ansible.builtin.command: ls /home
      register: home_dirs

    - name: Add home dirs to the backup spooler
      ansible.builtin.file:
        path: /mnt/bkspool/{{ item }}
        src: /home/{{ item }}
        state: link
      loop: "{{ home_dirs.stdout_lines }}"
      # same as loop: "{{ home_dirs.stdout.split() }}"
```

The string content of a registered variable can be empty. If you want to run another task only on hosts where the stdout of your registered variable is empty, check the registered variable's string contents for emptiness:

```
- name: check registered variable for emptiness
hosts: all

tasks:

  - name: List contents of directory
    ansible.builtin.command: ls mydir
    register: contents

  - name: Check contents for emptiness
    ansible.builtin.debug:
      msg: "Directory is empty"
    when: contents.stdout == ""
```

Ansible always registers something in a registered variable for every host, even on hosts where a task fails or Ansible skips a task because a condition is not met. To run a follow-up task on these hosts, query the registered variable for `is skipped` (not for “undefined” or “default”). See [Registering variables \(playbooks variables.html#registered-variables\)](#) for more information. Here are sample conditionals based on the success or failure of a task. Remember to ignore errors if you want Ansible to continue executing on a host when a failure occurs:

```
tasks:

- name: Register a variable, ignore errors and continue
  ansible.builtin.command: /bin/false
  register: result
  ignore_errors: true

- name: Run only if the task that registered the "result" variable fails
  ansible.builtin.command: /bin/something
  when: result is failed

- name: Run only if the task that registered the "result" variable succeeds
  ansible.builtin.command: /bin/something_else
  when: result is succeeded

- name: Run only if the task that registered the "result" variable is skipped
  ansible.builtin.command: /bin/still/something_else
  when: result is skipped
```

Note

Older versions of Ansible used `success` and `fail`, but `succeeded` and `failed` use the correct tense. All of these options are now valid.

Conditionals based on variables

You can also create conditionals based on variables defined in the playbooks or inventory. Because conditionals require boolean input (a test must evaluate as True to trigger the condition), you must apply the `| bool` filter to non boolean variables, such as string variables with content like 'yes', 'on', '1', or 'true'. You can define variables like this:

```
vars:
  epic: true
  monumental: "yes"
```

With the variables above, Ansible would run one of these tasks and skip the other:

```
tasks:
- name: Run the command if "epic" or "monumental" is true
  ansible.builtin.shell: echo "This certainly is epic!"
  when: epic or monumental | bool

- name: Run the command if "epic" is false
  ansible.builtin.shell: echo "This certainly isn't epic!"
  when: not epic
```

If a required variable has not been set, you can skip or fail using Jinja2's *defined* test. For example:

```
tasks:
- name: Run the command if "foo" is defined
  ansible.builtin.shell: echo "I've got '{{ foo }}' and am not afraid to use it!"
  when: foo is defined

- name: Fail if "bar" is undefined
  ansible.builtin.fail: msg="Bailing out. This play requires 'bar'"
  when: bar is undefined
```

This is especially useful in combination with the conditional import of vars files (see below). As the examples show, you do not need to use `{{ }}` to use variables inside conditionals, as these are already implied.

Using conditionals in loops

If you combine a `when` statement with a [loop](#) ([playbooks loops.html#playbooks-loops](#)), Ansible processes the condition separately for each item. This is by design, so you can execute the task on some items in the loop and skip it on other items. For example:

tasks:

- **name:** Run with items greater than 5
ansible.builtin.command: echo {{ item }}
loop: [0, 2, 4, 6, 8, 10]
when: item > 5

If you need to skip the whole task when the loop variable is undefined, use the `|default` filter to provide an empty iterator. For example, when looping over a list:

- **name:** Skip the whole task when a loop variable is undefined
ansible.builtin.command: echo {{ item }}
loop: "{{ mylist|default([]) }}"
when: item > 5

You can do the same thing when looping over a dict:

- **name:** The same as above using a dict
ansible.builtin.command: echo {{ item.key }}
loop: "{{ query('dict', mydict|default({})) }}"
when: item.value > 5

Loading custom facts

You can provide your own facts, as described in [Should you develop a module?](#) ([../dev_guide/developing_modules.html#developing-modules](#)). To run them, just make a call to your own custom fact gathering module at the top of your list of tasks, and variables returned there will be accessible to future tasks:

tasks:

- **name:** Gather site specific fact data
action: site_facts
- **name:** Use a custom fact
ansible.builtin.command: /usr/bin/thingy
when: my_custom_fact_just_retrieved_from_the_remote_system == '1234'

Conditionals with re-use

You can use conditionals with re-usable tasks files, playbooks, or roles. Ansible executes these conditional statements differently for dynamic re-use (includes) and for static re-use (imports). See [Re-using Ansible artifacts \(playbooks reuse.html#playbooks-reuse\)](#) for more information on re-use in Ansible.

Conditionals with imports

When you add a conditional to an import statement, Ansible applies the condition to all tasks within the imported file. This behavior is the equivalent of [Tag inheritance: adding tags to multiple tasks \(playbooks_tags.html#tag-inheritance\)](#). Ansible applies the condition to every task, and evaluates each task separately. For example, you might have a playbook called

`main.yml` and a tasks file called `other_tasks.yml`:

```
# all tasks within an imported file inherit the condition from the import statement
# main.yml
- import_tasks: other_tasks.yml # note "import"
  when: x is not defined

# other_tasks.yml
- name: Set a variable
  ansible.builtin.set_fact:
    x: foo

- name: Print a variable
  ansible.builtin.debug:
    var: x
```

Ansible expands this at execution time to the equivalent of:

```
- name: Set a variable if not defined
  ansible.builtin.set_fact:
    x: foo
  when: x is not defined
  # this task sets a value for x

- name: Do the task if "x" is not defined
  ansible.builtin.debug:
    var: x
  when: x is not defined
  # Ansible skips this task, because x is now defined
```

Thus if `x` is initially undefined, the `debug` task will be skipped. If this is not the behavior you want, use an `include_*` statement to apply a condition only to that statement itself.

You can apply conditions to `import_playbook` as well as to the other `import_*` statements. When you use this approach, Ansible returns a 'skipped' message for every task on every host that does not match the criteria, creating repetitive output. In many cases the [group by module \(./collections/ansible/builtin/group_by_module.html#group-by-module\)](#) can be a more streamlined way to accomplish the same objective; see [Handling OS and distro differences \(playbooks_best_practices.html#os-variance\)](#).

Conditionals with includes

When you use a conditional on an `include_*` statement, the condition is applied only to the include task itself and not to any other tasks within the included file(s). To contrast with the example used for conditionals on imports above, look at the same playbook and tasks file, but using an include instead of an import:

```
# Includes let you re-use a file to define a variable when it is not already defined

# main.yml
- include_tasks: other_tasks.yml
  when: x is not defined

# other_tasks.yml
- name: Set a variable
  ansible.builtin.set_fact:
    x: foo

- name: Print a variable
  ansible.builtin.debug:
    var: x
```

Ansible expands this at execution time to the equivalent of:

```
# main.yml
- include_tasks: other_tasks.yml
  when: x is not defined
  # if condition is met, Ansible includes other_tasks.yml

# other_tasks.yml
- name: Set a variable
  ansible.builtin.set_fact:
    x: foo
  # no condition applied to this task, Ansible sets the value of x to foo

- name: Print a variable
  ansible.builtin.debug:
    var: x
  # no condition applied to this task, Ansible prints the debug statement
```

By using `include_tasks` instead of `import_tasks`, both tasks from `other_tasks.yml` will be executed as expected. For more information on the differences between `include` v `import` see [Re-using Ansible artifacts \(playbooks_reuse.html#playbooks-reuse\)](#).

Conditionals with roles

There are three ways to apply conditions to roles:

- Add the same condition or conditions to all tasks in the role by placing your `when` statement under the `roles` keyword. See the example in this section.
- Add the same condition or conditions to all tasks in the role by placing your `when` statement on a static `import_role` in your playbook.
- Add a condition or conditions to individual tasks or blocks within the role itself. This is the only approach that allows you to select or skip some tasks within the role based on your `when` statement. To select or skip tasks within the role, you must have conditions set on individual tasks or blocks, use the dynamic `include_role` in your playbook, and add the condition or conditions to the include. When you use this approach, Ansible applies the condition to the include itself plus any tasks in the role that also have that `when` statement.

When you incorporate a role in your playbook statically with the `roles` keyword, Ansible adds the conditions you define to all the tasks in the role. For example:

```
- hosts: webservers
  roles:
    - role: debian_stock_config
      when: ansible_facts['os_family'] == 'Debian'
```

Selecting variables, files, or templates based on facts

Sometimes the facts about a host determine the values you want to use for certain variables or even the file or template you want to select for that host. For example, the names of packages are different on CentOS and on Debian. The configuration files for common services are also different on different OS flavors and versions. To load different variables file, templates, or other files based on a fact about the hosts:

1. name your vars files, templates, or files to match the Ansible fact that differentiates them
2. select the correct vars file, template, or file for each host with a variable based on that Ansible fact

Ansible separates variables from tasks, keeping your playbooks from turning into arbitrary code with nested conditionals. This approach results in more streamlined and auditable configuration rules because there are fewer decision points to track.

Selecting variables files based on facts

You can create a playbook that works on multiple platforms and OS versions with a minimum of syntax by placing your variable values in vars files and conditionally importing them. If you want to install Apache on some CentOS and some Debian servers, create variables files with YAML keys and values. For example:

```
---
# for vars/RedHat.yml
apache: httpd
somethingelse: 42
```

Then import those variables files based on the facts you gather on the hosts in your playbook:

```
---
- hosts: webservers
  remote_user: root
  vars_files:
    - "vars/common.yml"
    - [ "vars/{{ ansible_facts['os_family'] }}.yml", "vars/os_defaults.yml" ]
  tasks:
    - name: Make sure apache is started
      ansible.builtin.service:
        name: '{{ apache }}'
        state: started
```

Ansible gathers facts on the hosts in the webservers group, then interpolates the variable “ansible_facts[os_family]” into a list of filenames. If you have hosts with Red Hat operating systems (CentOS, for example), Ansible looks for ‘vars/RedHat.yml’. If that file does not exist, Ansible attempts to load ‘vars/os_defaults.yml’. For Debian hosts, Ansible first looks for ‘vars/Debian.yml’, before falling back on ‘vars/os_defaults.yml’. If no files in the list are found, Ansible raises an error.

Selecting files and templates based on facts

You can use the same approach when different OS flavors or versions require different configuration files or templates. Select the appropriate file or template based on the variables assigned to each host. This approach is often much cleaner than putting a lot of conditionals into a single template to cover multiple OS or package versions.

For example, you can template out a configuration file that is very different between, say, CentOS and Debian:

```
- name: Template a file
  ansible.builtin.template:
    src: "{{ item }}"
    dest: /etc/myapp/foo.conf
  loop: "{{ query('first_found', { 'files': myfiles, 'paths': mypaths }) }}"
  vars:
    myfiles:
      - "{{ ansible_facts['distribution'] }}.conf"
      - default.conf
    mypaths: [ 'search_location_one/somedir/', '/opt/other_location/somedir/' ]
```

Commonly-used facts

The following Ansible facts are frequently used in conditionals.

ansible facts['distribution']

Possible values (sample, not complete list):

```
Alpine
Altlinux
Amazon
Archlinux
ClearLinux
Coreos
CentOS
Debian
Fedora
Gentoo
Mandriva
NA
OpenWrt
OracleLinux
RedHat
Slackware
SLES
SMGL
SUSE
Ubuntu
VMwareESX
```

ansible facts['distribution major_version']

The major version of the operating system. For example, the value is *16* for Ubuntu 16.04.

ansible facts['os_family']

Possible values (sample, not complete list):

AIX
Alpine
Altlinux
Archlinux
Darwin
Debian
FreeBSD
Gentoo
HP-UX
Mandrake
RedHat
SGML
Slackware
Solaris
Suse
Windows

❗ See also

[Working with playbooks \(playbooks.html#working-with-playbooks\)](#)

An introduction to playbooks

[Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#)

Playbook organization by roles

[Tips and tricks \(playbooks_best_practices.html#playbooks-best-practices\)](#)

Tips and tricks for playbooks

[Using Variables \(playbooks_variables.html#playbooks-variables\)](#)

All about variables

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-pull

pulls playbooks from a VCS repo and executes them for the local host

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-pull [-h] [--version] [-v] [--private-key PRIVATE_KEY_FILE]
                  [-u REMOTE_USER] [-c CONNECTION] [-T TIMEOUT]
                  [--ssh-common-args SSH_COMMON_ARGS]
                  [--sftp-extra-args SFTP_EXTRA_ARGS]
                  [--scp-extra-args SCP_EXTRA_ARGS]
                  [--ssh-extra-args SSH_EXTRA_ARGS]
                  [-k | --connection-password-file CONNECTION_PASSWORD_FILE]
                  [--vault-id VAULT_IDS]
                  [--ask-vault-password | --vault-password-file VAULT_PASSWORD_FILES]
                  [-e EXTRA_VARS] [-t TAGS] [--skip-tags SKIP_TAGS]
                  [-i INVENTORY] [--list-hosts] [-l SUBSET] [-M MODULE_PATH]
                  [-K | --become-password-file BECOME_PASSWORD_FILE]
                  [--purge] [-o] [-s SLEEP] [-f] [-d DEST] [-U URL] [--full]
                  [-C CHECKOUT] [--accept-host-key] [-m MODULE_NAME]
                  [--verify-commit] [--clean] [--track-sub] [--check]
                  [--diff]
                  [playbook.yml [playbook.yml ...]]
```

Description

Used to pull a remote copy of ansible on each managed node, each set to run via cron and update playbook source via a source repository. This inverts the default *push* architecture of ansible into a *pull* architecture, which has near-limitless scaling potential.

The setup playbook can be tuned to change the cron frequency, logging locations, and parameters to ansible-pull. This is useful both for extreme scale-out as well as periodic remediation. Usage of the 'fetch' module to retrieve logs from ansible-pull runs would be an excellent way to gather and analyze remote logs from ansible-pull.

Common Options

--accept-host-key

adds the hostkey for the repo url if not already added

--ask-vault-password, --ask-vault-pass

ask for vault password

--become-password-file <BECOME_PASSWORD_FILE>, --become-pass-file <BECOME_PASSWORD_FILE>

Become password file

--check

don't make any changes; instead, try to predict some of the changes that may occur

--clean

modified files in the working repository will be discarded

--connection-password-file <CONNECTION_PASSWORD_FILE>, --conn-pass-file <CONNECTION_PASSWORD_FILE>

Connection password file

--diff

when changing (small) files and templates, show the differences in those files; works great with -check

--full

Do a full clone, instead of a shallow one.

--list-hosts

outputs a list of matching hosts; does not execute anything else

--private-key <PRIVATE_KEY_FILE>, --key-file <PRIVATE_KEY_FILE>

use this file to authenticate the connection

--purge

purge checkout after playbook run

--scp-extra-args <SCP_EXTRA_ARGS>

specify extra arguments to pass to scp only (e.g. -l)

--sftp-extra-args <SFTP_EXTRA_ARGS>

specify extra arguments to pass to sftp only (e.g. -f, -l)

--skip-tags

only run plays and tasks whose tags do not match these values

--ssh-common-args <SSH_COMMON_ARGS>

specify common arguments to pass to sftp/scp/ssh (e.g. ProxyCommand)

--ssh-extra-args <SSH_EXTRA_ARGS>

specify extra arguments to pass to ssh only (e.g. -R)

--track-subs

submodules will track the latest changes. This is equivalent to specifying the `--remote` flag to `git submodule update`

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

--verify-commit

verify GPG signature of checked out commit, if it fails abort running the playbook. This needs the corresponding VCS module to support such an operation

--version

Search this site

show program's version number, config file location, configured module search path, module location, executable location and exit

-C <CHECKOUT>, --checkout <CHECKOUT>

branch/tag/commit to checkout. Defaults to behavior of repository module.

-K, --ask-become-pass

ask for privilege escalation password

-M, --module-path

prepend colon-separated path(s) to module library
(default=~/.ansible/plugins/modules:/usr/share/ansible/plugins/modules)

-T <TIMEOUT>, --timeout <TIMEOUT>

override the connection timeout in seconds (default=10)

-U <URL>, --url <URL>

URL of the playbook repository

-c <CONNECTION>, --connection <CONNECTION>

connection type to use (default=smart)

-d <DEST>, --directory <DEST>

absolute path of repository checkout directory (relative paths are not supported)

-e, --extra-vars

set additional variables as key=value or YAML/JSON, if filename prepend with @

-f, --force

run the playbook even if the repository could not be updated

-h, --help

show this help message and exit

-i, --inventory, --inventory-file

specify inventory host path or comma separated host list. --inventory-file is deprecated

-k, --ask-pass

Search this site

ask for connection password

-l <SUBSET>, --limit <SUBSET>

further limit selected hosts to an additional pattern

-m <MODULE_NAME>, --module-name <MODULE_NAME>

Repository module name, which ansible will use to check out the repo. Choices are ('git', 'subversion', 'hg', 'bzd'). Default is git.

-o, --only-if-changed

only run the playbook if the repository has been updated

-s <SLEEP>, --sleep <SLEEP>

sleep for random interval (between 0 and n number of seconds) before starting. This is a useful way to disperse git requests

-t, --tags

only run plays and tasks tagged with these values

-u <REMOTE_USER>, --user <REMOTE_USER>

connect as this user (default=None)

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Environment

The following environment variables may be specified.

ANSIBLE_CONFIG ([./reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in ansible.cfg

Files

/etc/ansible/ansible.cfg – Config file, used if present

~/.ansible.cfg – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), ansible-config(1), ansible-console(1), ansible-doc(1), ansible-galaxy(1), ansible-inventory(1), ansible-playbook(1), ansible-pull(1), ansible-vault(1),

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Blocks

Blocks create logical groups of tasks. Blocks also offer ways to handle task errors, similar to exception handling in many programming languages.

- [Grouping tasks with blocks](#)
- [Handling errors with blocks](#)

Grouping tasks with blocks

All tasks in a block inherit directives applied at the block level. Most of what you can apply to a single task (with the exception of loops) can be applied at the block level, so blocks make it much easier to set data or directives common to the tasks. The directive does not affect the block itself, it is only inherited by the tasks enclosed by a block. For example, a *when* statement is applied to the tasks within a block, not to the block itself.

Block example with named tasks inside the block

tasks:

- **name:** Install, configure, and start Apache

block:

- **name:** Install httpd and memcached
ansible.builtin.yum:
 - name:**
 - httpd
 - memcached
 - state:** present
 - **name:** Apply the foo config template
ansible.builtin.template:
 - src:** templates/src.j2
 - dest:** /etc/foo.conf
 - **name:** Start service bar and enable it
ansible.builtin.service:
 - name:** bar
 - state:** started
 - enabled:** True
- when:** ansible_facts['distribution'] == 'CentOS'
become: true
become_user: root
ignore_errors: yes

In the example above, the ‘when’ condition will be evaluated before Ansible runs each of the three tasks in the block. All three tasks also inherit the privilege escalation directives, running as the root user. Finally, `ignore_errors: yes` ensures that Ansible continues to execute the playbook even if some of the tasks fail.

Names for blocks have been available since Ansible 2.3. We recommend using names in all tasks, within blocks or elsewhere, for better visibility into the tasks being executed when you run the playbook.

Handling errors with blocks

You can control how Ansible responds to task errors using blocks with `rescue` and `always` sections.

Rescue blocks specify tasks to run when an earlier task in a block fails. This approach is similar to exception handling in many programming languages. Ansible only runs rescue blocks after a task returns a ‘failed’ state. Bad task definitions and unreachable hosts will not trigger the rescue block.

Block error handling example

tasks:

- **name:** Handle the error

block:

- **name:** Print a message
ansible.builtin.debug:
 msg: 'I execute normally'
- **name:** Force a failure
ansible.builtin.command: /bin/false
- **name:** Never print this
ansible.builtin.debug:
 msg: 'I never execute, due to the above task failing, :-('

rescue:

- **name:** Print when errors
ansible.builtin.debug:
 msg: 'I caught an error, can do stuff here to fix it, :-)'

You can also add an `always` section to a block. Tasks in the `always` section run no matter what the task status of the previous block is.

Block with always section

- **name:** Always do X

block:

- **name:** Print a message
ansible.builtin.debug:
 msg: 'I execute normally'
- **name:** Force a failure
ansible.builtin.command: /bin/false
- **name:** Never print this
ansible.builtin.debug:
 msg: 'I never execute :-('

always:

- **name:** Always do this
ansible.builtin.debug:
 msg: "This always executes, :-)"

Together, these elements offer complex error handling.

Block with all sections

```
- name: Attempt and graceful roll back demo
```

```
block:
```

```
- name: Print a message
```

```
  ansible.builtin.debug:
```

```
    msg: 'I execute normally'
```

```
- name: Force a failure
```

```
  ansible.builtin.command: /bin/false
```

```
- name: Never print this
```

```
  ansible.builtin.debug:
```

```
    msg: 'I never execute, due to the above task failing, :-( '
```

```
rescue:
```

```
- name: Print when errors
```

```
  ansible.builtin.debug:
```

```
    msg: 'I caught an error'
```

```
- name: Force a failure in middle of recovery! >:-)
```

```
  ansible.builtin.command: /bin/false
```

```
- name: Never print this
```

```
  ansible.builtin.debug:
```

```
    msg: 'I also never execute :-( '
```

```
always:
```

```
- name: Always do this
```

```
  ansible.builtin.debug:
```

```
    msg: "This always executes"
```

The tasks in the `block` execute normally. If any tasks in the block return `failed`, the `rescue` section executes tasks to recover from the error. The `always` section runs regardless of the results of the `block` and `rescue` sections.

If an error occurs in the block and the rescue task succeeds, Ansible reverts the failed status of the original task for the run and continues to run the play as if the original task had succeeded. The rescued task is considered successful, and does not trigger `max_fail_percentage` or `any_errors_fatal` configurations. However, Ansible still reports a failure in the playbook statistics.

You can use blocks with `flush_handlers` in a rescue task to ensure that all handlers run even if an error occurs:

Block run handlers in error handling

tasks:

- **name:** Attempt and graceful roll back demo

block:

- **name:** Print a message
ansible.builtin.debug:
 msg: 'I execute normally'
changed_when: yes
notify: run me even after an error
- **name:** Force a failure
ansible.builtin.command: /bin/false

rescue:

- **name:** Make sure all handlers run
meta: flush_handlers

handlers:

- **name:** Run me even after an error
ansible.builtin.debug:
 msg: 'This handler runs even on error'

New in version 2.1.

Ansible provides a couple of variables for tasks in the `rescue` portion of a block:

ansible_failed_task

The task that returned 'failed' and triggered the rescue. For example, to get the name use

```
ansible_failed_task.name
```

ansible_failed_result

The captured return result of the failed task that triggered the rescue. This would equate to having used this var in the `register` keyword.

❗ See also

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#)

Playbook organization by roles

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

ansible-vault

encryption/decryption utility for Ansible data files

- [Synopsis](#)
- [Description](#)
- [Common Options](#)
- [Actions](#)
 - [create](#)
 - [decrypt](#)
 - [edit](#)
 - [view](#)
 - [encrypt](#)
 - [encrypt_string](#)
 - [rekey](#)
- [Environment](#)
- [Files](#)
- [Author](#)
- [License](#)
- [See also](#)

Synopsis

```
usage: ansible-vault [-h] [--version] [-v]
                  {create,decrypt,edit,view,encrypt,encrypt_string,rekey}
                  ...
```

Description

can encrypt any structured data file used by Ansible. This can include *group_vars/* or *host_vars/* inventory variables, variables loaded by *include_vars* or *vars_files*, or variable files passed on the ansible-playbook command line with *-e @file.yml* or *-e @file.json*. Role variables

and defaults are also included!

Because Ansible tasks, handlers, and other objects are data, these can also be encrypted with vault. If you'd like to not expose what variables you are using, you can keep an individual task file entirely encrypted.

Common Options

--version

show program's version number, config file location, configured module search path, module location, executable location and exit

-h, --help

show this help message and exit

-v, --verbose

verbose mode (-vvv for more, -vvvv to enable connection debugging)

Actions

create

create and open a file in an editor that will be encrypted with the provided vault secret when closed

--ask-vault-password, --ask-vault-pass

ask for vault password

--encrypt-vault-id <ENCRYPT_VAULT_ID>

the vault id used to encrypt (required if more than one vault-id is provided)

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

decrypt

decrypt the supplied file using the provided vault secret

Search this site

--ask-vault-password, --ask-vault-pass

ask for vault password

--output <OUTPUT_FILE>

output file name for encrypt or decrypt; use - for stdout

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

edit

open and decrypt an existing vaulted file in an editor, that will be encrypted again when closed

--ask-vault-password, --ask-vault-pass

ask for vault password

--encrypt-vault-id <ENCRYPT_VAULT_ID>

the vault id used to encrypt (required if more than one vault-id is provided)

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

view

open, decrypt and view an existing vaulted file using a pager using the supplied vault secret

--ask-vault-password, --ask-vault-pass

ask for vault password

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

encrypt

encrypt the supplied file using the provided vault secret

--ask-vault-password, --ask-vault-pass

ask for vault password

--encrypt-vault-id <ENCRYPT_VAULT_ID>

the vault id used to encrypt (required if more than one vault-id is provided)

--output <OUTPUT_FILE>

output file name for encrypt or decrypt; use - for stdout

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

encrypt_string

encrypt the supplied string using the provided vault secret

--ask-vault-password, --ask-vault-pass

ask for vault password

--encrypt-vault-id <ENCRYPT_VAULT_ID>

the vault id used to encrypt (required if more than one vault-id is provided)

--output <OUTPUT_FILE>

output file name for encrypt or decrypt; use - for stdout

--show-input

Do not hide input when prompted for the string to encrypt

--stdin-name <ENCRYPT_STRING_STDIN_NAME>

Specify the variable name for stdin

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

-n, --name

Specify the variable name

-p, --prompt

Prompt for the string to encrypt

rekey

re-encrypt a vaulted file with a new secret, the previous secret is required

--ask-vault-password, --ask-vault-pass

ask for vault password

--encrypt-vault-id <ENCRYPT_VAULT_ID>

the vault id used to encrypt (required if more than one vault-id is provided)

--new-vault-id <NEW_VAULT_ID>

the new vault identity to use for rekey

--new-vault-password-file <NEW_VAULT_PASSWORD_FILE>

new vault password file for rekey

--vault-id

the vault identity to use

--vault-password-file, --vault-pass-file

vault password file

Environment

Search this site

The following environment variables may be specified.

`ANSIBLE_CONFIG` ([../reference_appendices/config.html#envvar-ANSIBLE_CONFIG](#)) – Override the default ansible config file

Many more are available for most options in `ansible.cfg`

Files

`/etc/ansible/ansible.cfg` – Config file, used if present

`~/.ansible.cfg` – User config file, overrides the default config if present

Author

Ansible was originally written by Michael DeHaan.

See the *AUTHORS* file for a complete list of contributors.

License

Ansible is released under the terms of the GPLv3+ License.

See also

ansible(1), *ansible-config(1)*, *ansible-console(1)*, *ansible-doc(1)*, *ansible-galaxy(1)*, *ansible-inventory(1)*, *ansible-playbook(1)*, *ansible-pull(1)*, *ansible-vault(1)*,

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Handlers: running operations on change

Sometimes you want a task to run only when a change is made on a machine. For example, you may want to restart a service if a task updates the configuration of that service, but not if the configuration is unchanged. Ansible uses handlers to address this use case. Handlers are tasks that only run when notified. Each handler should have a globally unique name.

- [Handler example](#)
- [Controlling when handlers run](#)
- [Using variables with handlers](#)

Handler example

This playbook, `verify-apache.yml`, contains a single play with a handler.

```

---
- name: Verify apache installation
  hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
    - name: Ensure apache is at the latest version
      ansible.builtin.yum:
        name: httpd
        state: latest

    - name: Write the apache config file
      ansible.builtin.template:
        src: /srv/httpd.j2
        dest: /etc/httpd.conf
      notify:
        - Restart apache

    - name: Ensure apache is running
      ansible.builtin.service:
        name: httpd
        state: started

  handlers:
    - name: Restart apache
      ansible.builtin.service:
        name: httpd
        state: restarted

```

In this example playbook, the second task notifies the handler. A single task can notify more than one handler.

```

- name: Template configuration file
  ansible.builtin.template:
    src: template.j2
    dest: /etc/foo.conf
  notify:
    - Restart memcached
    - Restart apache

  handlers:
    - name: Restart memcached
      ansible.builtin.service:
        name: memcached
        state: restarted

    - name: Restart apache
      ansible.builtin.service:
        name: apache
        state: restarted

```

Controlling when handlers run

By default, handlers run after all the tasks in a particular play have been completed. This approach is efficient, because the handler only runs once, regardless of how many tasks notify it. For example, if multiple tasks update a configuration file and notify a handler to restart Apache, Ansible only bounces Apache once to avoid unnecessary restarts.

If you need handlers to run before the end of the play, add a task to flush them using the [meta module](https://docs.ansible.com/ansible/latest/collections/ansible/builtin/meta_module.html#meta-module) (`./collections/ansible/builtin/meta_module.html#meta-module`), which executes Ansible actions.

```
tasks:
  - name: Some tasks go here
    ansible.builtin.shell: ...

  - name: Flush handlers
    meta: flush_handlers

  - name: Some other tasks
    ansible.builtin.shell: ...
```

The `meta: flush_handlers` task triggers any handlers that have been notified at that point in the play.

Using variables with handlers

You may want your Ansible handlers to use variables. For example, if the name of a service varies slightly by distribution, you want your output to show the exact name of the restarted service for each target machine. Avoid placing variables in the name of the handler. Since handler names are templated early on, Ansible may not have a value available for a handler name like this:

```
handlers:
  # This handler name may cause your play to fail!
  - name: Restart "{{ web_service_name }}"
```

If the variable used in the handler name is not available, the entire play fails. Changing that variable mid-play **will not** result in newly created handler.

Instead, place variables in the task parameters of your handler. You can load the values using `include_vars` like this:

```

tasks:
  - name: Set host variables based on distribution
    include_vars: "{{ ansible_facts.distribution }}.yaml"

handlers:
  - name: Restart web service
    ansible.builtin.service:
      name: "{{ web_service_name | default('httpd') }}"
      state: restarted

```

Handlers can also “listen” to generic topics, and tasks can notify those topics as follows:

```

handlers:
  - name: Restart memcached
    ansible.builtin.service:
      name: memcached
      state: restarted
      listen: "restart web services"

  - name: Restart apache
    ansible.builtin.service:
      name: apache
      state: restarted
      listen: "restart web services"

tasks:
  - name: Restart everything
    ansible.builtin.command: echo "this task will restart the web services"
    notify: "restart web services"

```

This use makes it much easier to trigger multiple handlers. It also decouples handlers from their names, making it easier to share handlers among playbooks and roles (especially when using 3rd party roles from a shared source like Galaxy).

❗ Note

- Handlers always run in the order they are defined, not in the order listed in the notify-statement. This is also the case for handlers using *listen*.
- Handler names and *listen* topics live in a global namespace.
- Handler names are templatable and *listen* topics are not.
- Use unique handler names. If you trigger more than one handler with the same name, the first one(s) get overwritten. Only the last one defined will run.
- You can notify a handler defined inside a static include.
- You cannot notify a handler defined inside a dynamic include.
- A handler can not run `import_role` or `include_role`.

When using handlers within roles, note that:

- handlers notified within `pre_tasks`, `tasks`, and `post_tasks` sections are automatically flushed at the end of section where they were notified.
- handlers notified within `roles` section are automatically flushed at the end of `tasks` section, but before any `tasks` handlers.
- handlers are play scoped and as such can be used outside of the role they are defined in.

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Error handling in playbooks

When Ansible receives a non-zero return code from a command or a failure from a module, by default it stops executing on that host and continues on other hosts. However, in some circumstances you may want different behavior. Sometimes a non-zero return code indicates success. Sometimes you want a failure on one host to stop execution on all hosts. Ansible provides tools and settings to handle these situations and help you get the behavior, output, and reporting you want.

- [Ignoring failed commands](#)
- [Ignoring unreachable host errors](#)
- [Resetting unreachable hosts](#)
- [Handlers and failure](#)
- [Defining failure](#)
- [Defining “changed”](#)
- [Ensuring success for command and shell](#)
- [Aborting a play on all hosts](#)
 - [Aborting on the first error: `any_errors_fatal`](#)
 - [Setting a maximum failure percentage](#)
- [Controlling errors in blocks](#)

Ignoring failed commands

By default Ansible stops executing tasks on a host when a task fails on that host. You can use `ignore_errors` to continue on in spite of the failure.

```
- name: Do not count this as a failure
  ansible.builtin.command: /bin/false
  ignore_errors: yes
```

The `ignore_errors` directive only works when the task is able to run and returns a value of 'failed'. It does not make Ansible ignore undefined variable errors, connection failures, execution issues (for example, missing packages), or syntax errors.

Ignoring unreachable host errors

New in version 2.7.

You can ignore a task failure due to the host instance being 'UNREACHABLE' with the `ignore_unreachable` keyword. Ansible ignores the task errors, but continues to execute future tasks against the unreachable host. For example, at the task level:

```
- name: This executes, fails, and the failure is ignored
  ansible.builtin.command: /bin/true
  ignore_unreachable: yes

- name: This executes, fails, and ends the play for this host
  ansible.builtin.command: /bin/true
```

And at the playbook level:

```
- hosts: all
  ignore_unreachable: yes
  tasks:
    - name: This executes, fails, and the failure is ignored
      ansible.builtin.command: /bin/true

    - name: This executes, fails, and ends the play for this host
      ansible.builtin.command: /bin/true
      ignore_unreachable: no
```

Resetting unreachable hosts

If Ansible cannot connect to a host, it marks that host as 'UNREACHABLE' and removes it from the list of active hosts for the run. You can use *meta: clear_host_errors* to reactivate all hosts, so subsequent tasks can try to reach them again.

Handlers and failure

Ansible runs [handlers](#) ([playbooks_handlers.html#handlers](#)) at the end of each play. If a task notifies a handler but another task fails later in the play, by default the handler does *not* run on that host, which may leave the host in an unexpected state. For example, a task could update a configuration file and notify a handler to restart some service. If a task later in the same play fails, the configuration file might be changed but the service will not be restarted.

You can change this behavior with the `--force-handlers` command-line option, by including `force_handlers: True` in a play, or by adding `force_handlers = True` to `ansible.cfg`. When handlers are forced, Ansible will run all notified handlers on all hosts, even hosts with failed

tasks. (Note that certain errors could still prevent the handler from running, such as a host becoming unreachable.)

Defining failure

Ansible lets you define what “failure” means in each task using the `failed_when` conditional. As with all conditionals in Ansible, lists of multiple `failed_when` conditions are joined with an implicit `and`, meaning the task only fails when *all* conditions are met. If you want to trigger a failure when any of the conditions is met, you must define the conditions in a string with an explicit `or` operator.

You may check for failure by searching for a word or phrase in the output of a command

```
- name: Fail task when the command error output prints FAILED
  ansible.builtin.command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

or based on the return code

```
- name: Fail task when both files are identical
  ansible.builtin.raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

You can also combine multiple conditions for failure. This task will fail if both conditions are true:

```
- name: Check if a file exists in temp and fail task if it does
  ansible.builtin.command: ls /tmp/this_should_not_be_here
  register: result
  failed_when:
    - result.rc == 0
    - '"No such" not in result.stdout'
```

If you want the task to fail when only one condition is satisfied, change the `failed_when` definition to

```
failed_when: result.rc == 0 or "No such" not in result.stdout
```

If you have too many conditions to fit neatly into one line, you can split it into a multi-line YAML value with `>`.

```
- name: example of many failed_when conditions with OR
  ansible.builtin.shell: "./myBinary"
  register: ret
  failed_when: >
    ("No such file or directory" in ret.stdout) or
    (ret.stderr != '') or
    (ret.rc == 10)
```

Defining “changed”

Ansible lets you define when a particular task has “changed” a remote node using the `changed_when` conditional. This lets you determine, based on return codes or output, whether a change should be reported in Ansible statistics and whether a handler should be triggered or not. As with all conditionals in Ansible, lists of multiple `changed_when` conditions are joined with an implicit `and`, meaning the task only reports a change when *all* conditions are met. If you want to report a change when any of the conditions is met, you must define the conditions in a string with an explicit `or` operator. For example:

```
tasks:

- name: Report 'changed' when the return code is not equal to 2
  ansible.builtin.shell: /usr/bin/billybass --mode="take me to the river"
  register: bass_result
  changed_when: "bass_result.rc != 2"

- name: This will never report 'changed' status
  ansible.builtin.shell: wall 'beep'
  changed_when: False
```

You can also combine multiple conditions to override “changed” result.

```
- name: Combine multiple conditions to override 'changed' result
  ansible.builtin.command: /bin/fake_command
  register: result
  ignore_errors: True
  changed_when:
    - '"ERROR" in result.stderr'
    - result.rc == 2
```

See [Defining failure](#) for more conditional syntax examples.

Ensuring success for command and shell

The [command](#) ([../collections/ansible/builtin/command_module.html#command-module](#)) and [shell](#) ([../collections/ansible/builtin/shell_module.html#shell-module](#)) modules care about return codes, so if you have a command whose successful exit code is not zero, you can do this:

```
tasks:
- name: Run this command and ignore the result
  ansible.builtin.shell: /usr/bin/somecommand || /bin/true
```

Aborting a play on all hosts

Sometimes you want a failure on a single host, or failures on a certain percentage of hosts, to abort the entire play on all hosts. You can stop play execution after the first failure happens with `any_errors_fatal`. For finer-grained control, you can use `max_fail_percentage` to abort the run after a given percentage of hosts has failed.

Aborting on the first error: any_errors_fatal

If you set `any_errors_fatal` and a task returns an error, Ansible finishes the fatal task on all hosts in the current batch, then stops executing the play on all hosts. Subsequent tasks and plays are not executed. You can recover from fatal errors by adding a [rescue section](#) ([playbooks blocks.html#block-error-handling](#)) to the block. You can set `any_errors_fatal` at the play or block level.

```
- hosts: somehosts
  any_errors_fatal: true
  roles:
    - myrole

- hosts: somehosts
  tasks:
    - block:
        - include_tasks: mytasks.yml
      any_errors_fatal: true
```

You can use this feature when all tasks must be 100% successful to continue playbook execution. For example, if you run a service on machines in multiple data centers with load balancers to pass traffic from users to the service, you want all load balancers to be disabled before you stop the service for maintenance. To ensure that any failure in the task that disables the load balancers will stop all other tasks:

```

---
- hosts: load_balancers_dc_a
  any_errors_fatal: true

  tasks:
    - name: Shut down datacenter 'A'
      ansible.builtin.command: /usr/bin/disable-dc

- hosts: frontends_dc_a

  tasks:
    - name: Stop service
      ansible.builtin.command: /usr/bin/stop-software

    - name: Update software
      ansible.builtin.command: /usr/bin/upgrade-software

- hosts: load_balancers_dc_a

  tasks:
    - name: Start datacenter 'A'
      ansible.builtin.command: /usr/bin/enable-dc

```

In this example Ansible starts the software upgrade on the front ends only if all of the load balancers are successfully disabled.

Setting a maximum failure percentage

By default, Ansible continues to execute tasks as long as there are hosts that have not yet failed. In some situations, such as when executing a rolling update, you may want to abort the play when a certain threshold of failures has been reached. To achieve this, you can set a maximum failure percentage on a play:

```

---
- hosts: webservers
  max_fail_percentage: 30
  serial: 10

```

The `max_fail_percentage` setting applies to each batch when you use it with [serial](#) ([playbooks_strategies.html#rolling-update-batch-size](#)). In the example above, if more than 3 of the 10 servers in the first (or any) batch of servers failed, the rest of the play would be aborted.

Note

The percentage set must be exceeded, not equaled. For example, if serial were set to 4 and you wanted the task to abort the play when 2 of the systems failed, set the `max_fail_percentage` at 49 rather than 50.

Controlling errors in blocks

You can also use blocks to define responses to task errors. This approach is similar to exception handling in many programming languages. See [Handling errors with blocks](#) ([playbooks_blocks.html#block-error-handling](#)), for details and examples.

📌 See also

Intro to playbooks ([playbooks_intro.html#playbooks-intro](#))

An introduction to playbooks

Tips and tricks ([playbooks_best_practices.html#playbooks-best-practices](#))

Tips and tricks for playbooks

Conditionals ([playbooks_conditionals.html#playbooks-conditionals](#))

Conditional statements in playbooks

Using Variables ([playbooks_variables.html#playbooks-variables](#))

All about variables

User Mailing List (<https://groups.google.com/group/ansible-devel>)

Have a question? Stop by the google group!

Real-time chat ([../community/communication.html#communication-irc](#))

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Setting the remote environment

New in version 1.1.

You can use the `environment` keyword at the play, block, or task level to set an environment variable for an action on a remote host. With this keyword, you can enable using a proxy for a task that does http requests, set the required environment variables for language-specific version managers, and more.

When you set a value with `environment:` at the play or block level, it is available only to tasks within the play or block that are executed by the same user. The `environment:` keyword does not affect Ansible itself, Ansible configuration settings, the environment for other users, or the execution of other plugins like lookups and filters. Variables set with `environment:` do not automatically become Ansible facts, even when you set them at the play level. You must include an explicit `gather_facts` task in your playbook and set the `environment` keyword on that task to turn these values into Ansible facts.

- [Setting the remote environment in a task](#)

Setting the remote environment in a task

You can set the environment directly at the task level.

```
- hosts: all
  remote_user: root

  tasks:

    - name: Install cobbler
      ansible.builtin.package:
        name: cobbler
        state: present
      environment:
        http_proxy: http://proxy.example.com:8080
```

You can re-use environment settings by defining them as variables in your play and accessing them in a task as you would access any stored Ansible variable.

```
- hosts: all
  remote_user: root

  # create a variable named "proxy_env" that is a dictionary
  vars:
    proxy_env:
      http_proxy: http://proxy.example.com:8080

  tasks:

    - name: Install cobbler
      ansible.builtin.package:
        name: cobbler
        state: present
        environment: "{{ proxy_env }}"
```

You can store environment settings for re-use in multiple playbooks by defining them in a `group_vars` file.

```
---
# file: group_vars/boston

ntp_server: ntp.bos.example.com
backup: bak.bos.example.com
proxy_env:
  http_proxy: http://proxy.bos.example.com:8080
  https_proxy: http://proxy.bos.example.com:8080
```

You can set the remote environment at the play level.

```
- hosts: testing

  roles:
    - php
    - nginx

  environment:
    http_proxy: http://proxy.example.com:8080
```

These examples show proxy settings, but you can provide any number of settings this way.

Working with language-specific version managers

Some language-specific version managers (such as `renv` and `nvm`) require you to set environment variables while these tools are in use. When using these tools manually, you usually source some environment variables from a script or from lines added to your shell configuration file. In Ansible, you can do this with the `environment` keyword at the play level.

```
---
### A playbook demonstrating a common npm workflow:
# - Check for package.json in the application directory
# - If package.json exists:
#   * Run npm prune
#   * Run npm install

- hosts: application
  become: false

  vars:
    node_app_dir: /var/local/my_node_app

  environment:
    NVM_DIR: /var/local/nvm
    PATH: /var/local/nvm/versions/node/v4.2.1/bin:{{ ansible_env.PATH }}

  tasks:
    - name: Check for package.json
      ansible.builtin.stat:
        path: '{{ node_app_dir }}/package.json'
      register: packagejson

    - name: Run npm prune
      ansible.builtin.command: npm prune
      args:
        chdir: '{{ node_app_dir }}'
      when: packagejson.stat.exists

    - name: Run npm install
      community.general.npm:
        path: '{{ node_app_dir }}'
      when: packagejson.stat.exists
```

❗ Note

The example above uses `ansible_env` as part of the `PATH`. Basing variables on `ansible_env` is risky. Ansible populates `ansible_env` values by gathering facts, so the value of the variables depends on the `remote_user` or `become_user` Ansible used when gathering those facts. If you change `remote_user`/`become_user` the values in `ansible_env` may not be the ones you expect.

❗ Warning

Environment variables are normally passed in clear text (shell plugin dependent) so they are not a recommended way of passing secrets to the module being executed.

You can also specify the environment at the task level.

```
---
- name: Install ruby 2.3.1
  ansible.builtin.command: rbenv install {{ rbenv_ruby_version }}
  args:
    creates: '{{ rbenv_root }}/versions/{{ rbenv_ruby_version }}/bin/ruby'
  vars:
    rbenv_root: /usr/local/rbenv
    rbenv_ruby_version: 2.3.1
  environment:
    CONFIGURE_OPTS: '--disable-install-doc'
    RBENV_ROOT: '{{ rbenv_root }}'
    PATH: '{{ rbenv_root }}/bin:{{ rbenv_root }}/shims:{{ rbenv_plugins }}/ruby-
build/bin:{{ ansible_env.PATH }}
```

❗ See also

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(./community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Re-using Ansible artifacts

You can write a simple playbook in one very large file, and most users learn the one-file approach first. However, breaking tasks up into different files is an excellent way to organize complex sets of tasks and reuse them. Smaller, more distributed artifacts let you re-use the same variables, tasks, and plays in multiple playbooks to address different use cases. You can use distributed artifacts across multiple parent playbooks or even multiple times within one playbook. For example, you might want to update your customer database as part of several different playbooks. If you put all the tasks related to updating your database in a tasks file, you can re-use them in many playbooks while only maintaining them in one place.

- [Creating re-usable files and roles](#)
- [Re-using playbooks](#)
- [Re-using files and roles](#)
 - [Includes: dynamic re-use](#)
 - [Imports: static re-use](#)
 - [Comparing includes and imports: dynamic and static re-use](#)
- [Re-using tasks as handlers](#)
 - [Triggering included \(dynamic\) handlers](#)
 - [Triggering imported \(static\) handlers](#)

Creating re-usable files and roles

Ansible offers four distributed, re-usable artifacts: variables files, task files, playbooks, and roles.

- A variables file contains only variables.
- A task file contains only tasks.
- A playbook contains at least one play, and may contain variables, tasks, and other content. You can re-use tightly focused playbooks, but you can only re-use them statically, not dynamically.
- A role contains a set of related tasks, variables, defaults, handlers, and even modules or other plugins in a defined file-tree. Unlike variables files, task files, or playbooks, roles can be easily uploaded and shared via Ansible Galaxy. See [Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#) for details about creating and using roles.

New in version 2.4.

Re-using playbooks

You can incorporate multiple playbooks into a main playbook. However, you can only use imports to re-use playbooks. For example:

```
- import_playbook: webservers.yml
- import_playbook: databases.yml
```

Importing incorporates playbooks in other playbooks statically. Ansible runs the plays and tasks in each imported playbook in the order they are listed, just as if they had been defined directly in the main playbook.

You can select which playbook you want to import at runtime by defining your imported playbook filename with a variable, then passing the variable with either `--extra-vars` or the `vars` keyword. For example:

```
- import_playbook: "/path/to/{{ import_from_extra_var }}"
- import_playbook: "{{ import_from_vars }}"
vars:
  import_from_vars: /path/to/one_playbook.yml
```

If you run this playbook with `ansible-playbook my_playbook -e import_from_extra_var=other_playbook.yml`, Ansible imports both `one_playbook.yml` and `other_playbook.yml`.

Re-using files and roles

Ansible offers two ways to re-use files and roles in a playbook: dynamic and static.

- For dynamic re-use, add an `include_*` task in the tasks section of a play:
 - [include_role \(../collections/ansible/builtin/include_role_module.html#include-role-module\)](#)
 - [include_tasks \(../collections/ansible/builtin/include_tasks_module.html#include-tasks-module\)](#)
 - [include_vars \(../collections/ansible/builtin/include_vars_module.html#include-vars-module\)](#)
- For static re-use, add an `import_*` task in the tasks section of a play:
 - [import_role \(../collections/ansible/builtin/import_role_module.html#import-role-module\)](#)
 - [import_tasks \(../collections/ansible/builtin/import_tasks_module.html#import-tasks-module\)](#)

Task include and import statements can be used at arbitrary depth.

You can still use the bare [roles \(playbooks_reuse_roles.html#roles-keyword\)](#) keyword at the play level to incorporate a role in a playbook statically. However, the bare [include \(../collections/ansible/builtin/include_module.html#include-module\)](#) keyword, once used for both task files and playbook-level includes, is now deprecated.

Includes: dynamic re-use

Including roles, tasks, or variables adds them to a playbook dynamically. Ansible processes included files and roles as they come up in a playbook, so included tasks can be affected by the results of earlier tasks within the top-level playbook. Included roles and tasks are similar to handlers - they may or may not run, depending on the results of other tasks in the top-level playbook.

The primary advantage of using `include_*` statements is looping. When a loop is used with an include, the included tasks or role will be executed once for each item in the loop.

The filenames for included roles, tasks, and vars are templated before inclusion.

You can pass variables into includes. See [Variable precedence: Where should I put a variable? \(playbooks_variables.html#ansible-variable-precedence\)](#) for more details on variable inheritance and precedence.

Imports: static re-use

Importing roles, tasks, or playbooks adds them to a playbook statically. Ansible pre-processes imported files and roles before it runs any tasks in a playbook, so imported content is never affected by other tasks within the top-level playbook.

The filenames for imported roles and tasks support templating, but the variables must be available when Ansible is pre-processing the imports. This can be done with the `vars` keyword or by using `--extra-vars`.

You can pass variables to imports. You must pass variables if you want to run an imported file more than once in a playbook. For example:

```
tasks:
- import_tasks: wordpress.yml
  vars:
    wp_user: timmy

- import_tasks: wordpress.yml
  vars:
    wp_user: alice

- import_tasks: wordpress.yml
  vars:
    wp_user: bob
```

See [Variable precedence: Where should I put a variable? \(playbooks_variables.html#ansible-variable-precedence\)](#) for more details on variable inheritance and precedence.

Comparing includes and imports: dynamic and static re-use

Each approach to re-using distributed Ansible artifacts has advantages and limitations. You may choose dynamic re-use for some playbooks and static re-use for others. Although you can use both dynamic and static re-use in a single playbook, it is best to select one approach per playbook. Mixing static and dynamic re-use can introduce difficult-to-diagnose bugs into your playbooks. This table summarizes the main differences so you can choose the best approach for each playbook you create.

	Include_*	Import_*
Type of re-use	Dynamic	Static
When processed	At runtime, when encountered	Pre-processed during play
Task or play	All includes are tasks	<code>import_playbook</code> cannot
Task options	Apply only to include task itself	Apply to all child tasks in
Calling from loops	Executed once for each loop item	Cannot be used in a loop
Using <code>--list-tags</code>	Tags within includes not listed	All tags appear with <code>--li</code>
Using <code>--list-tasks</code>	Tasks within includes not listed	All tasks appear with <code>--l</code>
Notifying handlers	Cannot trigger handlers within includes	Can trigger individual imp
Using <code>--start-at-task</code>	Cannot start at tasks within includes	Can start at imported task

	Include_*	Import_*
Using inventory variables	Can <code>include_*: {{ inventory_var }}</code>	Cannot <code>import_*: {{ inv</code>
With playbooks	No <code>include_playbook</code>	Can import full playbooks
With variables files	Can include variables files	Use <code>vars_files:</code> to impo

❗ Note

- There are also big differences in resource consumption and performance, imports are quite lean and fast, while includes require a lot of management and accounting.

Re-using tasks as handlers

You can also use includes and imports in the [Handlers: running operations on change \(playbooks handlers.html#handlers\)](#) section of a playbook. For instance, if you want to define how to restart Apache, you only have to do that once for all of your playbooks. You might make a `restarts.yml` file that looks like:

```
# restarts.yml
- name: Restart apache
  ansible.builtin.service:
    name: apache
    state: restarted

- name: Restart mysql
  ansible.builtin.service:
    name: mysql
    state: restarted
```

You can trigger handlers from either an import or an include, but the procedure is different for each method of re-use. If you include the file, you must notify the include itself, which triggers all the tasks in `restarts.yml`. If you import the file, you must notify the individual task(s) within `restarts.yml`. You can mix direct tasks and handlers with included or imported tasks and handlers.

Triggering included (dynamic) handlers

Includes are executed at run-time, so the name of the include exists during play execution, but the included tasks do not exist until the include itself is triggered. To use the `Restart apache` task with dynamic re-use, refer to the name of the include itself. This approach triggers all tasks in the included file as handlers. For example, with the task file shown above:

```
- name: Trigger an included (dynamic) handler
hosts: localhost
handlers:
  - name: Restart services
    include_tasks: restarts.yml
tasks:
  - command: "true"
    notify: Restart services
```

Triggering imported (static) handlers

Imports are processed before the play begins, so the name of the import no longer exists during play execution, but the names of the individual imported tasks do exist. To use the `Restart apache` task with static re-use, refer to the name of each task or tasks within the imported file. For example, with the task file shown above:

```
- name: Trigger an imported (static) handler
hosts: localhost
handlers:
  - name: Restart services
    import_tasks: restarts.yml
tasks:
  - command: "true"
    notify: Restart apache
  - command: "true"
    notify: Restart mysql
```

❗ See also

Utilities modules

(https://docs.ansible.com/ansible/2.9/modules/list_of_utilities_modules.html#utilities-modules)

Documentation of the `include*` and `import*` modules discussed here.

Working with playbooks ([playbooks.html#working-with-playbooks](#))

Review the basic Playbook language features

Using Variables ([playbooks_variables.html#playbooks-variables](#))

All about variables in playbooks

Conditionals ([playbooks_conditionals.html#playbooks-conditionals](#))

Conditionals in playbooks

Loops ([playbooks_loops.html#playbooks-loops](#))

Loops in playbooks

[Tips and tricks \(playbooks_best_practices.html#playbooks-best-practices\)](#)

Tips and tricks for playbooks

[Galaxy User Guide \(../galaxy/user_guide.html#ansible-galaxy\)](#)

How to share roles on galaxy, role management

[GitHub Ansible examples \(https://github.com/ansible/ansible-examples\)](https://github.com/ansible/ansible-examples)

Complete playbook files from the GitHub project source

[Mailing List \(https://groups.google.com/group/ansible-project\)](https://groups.google.com/group/ansible-project)

Questions? Help? Ideas? Stop by the list on Google Groups

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Roles

Roles let you automatically load related vars, files, tasks, handlers, and other Ansible artifacts based on a known file structure. After you group your content in roles, you can easily reuse them and share them with other users.

- [Role directory structure](#)
- [Storing and finding roles](#)
- [Using roles](#)
 - [Using roles at the play level](#)
 - [Including roles: dynamic reuse](#)
 - [Importing roles: static reuse](#)
- [Role argument validation](#)
 - [Specification format](#)
 - [Sample specification](#)
- [Running a role multiple times in one playbook](#)
 - [Passing different parameters](#)
 - [Using `allow_duplicates: true`](#)
- [Using role dependencies](#)
 - [Running role dependencies multiple times in one play](#)
- [Embedding modules and plugins in roles](#)
- [Sharing roles: Ansible Galaxy](#)

Role directory structure

An Ansible role has a defined directory structure with eight main standard directories. You must include at least one of these directories in each role. You can omit any directories the role does not use. For example:

```
# playbooks
site.yml
webservers.yml
fooservers.yml
roles/
  common/
    tasks/
    handlers/
    library/
    files/
    templates/
    vars/
    defaults/
    meta/
  webservers/
    tasks/
    defaults/
    meta/
```

By default Ansible will look in each directory within a role for a `main.yml` file for relevant content (also `main.yaml` and `main`):

- `tasks/main.yml` - the main list of tasks that the role executes.
- `handlers/main.yml` - handlers, which may be used within or outside this role.
- `library/my_module.py` - modules, which may be used within this role (see [Embedding modules and plugins in roles](#) for more information).
- `defaults/main.yml` - default variables for the role (see [Using Variables \(playbooks variables.html#playbooks-variables\)](#) for more information). These variables have the lowest priority of any variables available, and can be easily overridden by any other variable, including inventory variables.
- `vars/main.yml` - other variables for the role (see [Using Variables \(playbooks variables.html#playbooks-variables\)](#) for more information).
- `files/main.yml` - files that the role deploys.
- `templates/main.yml` - templates that the role deploys.
- `meta/main.yml` - metadata for the role, including role dependencies.

You can add other YAML files in some directories. For example, you can place platform-specific tasks in separate files and refer to them in the `tasks/main.yml` file:

```
# roles/example/tasks/main.yml
- name: Install the correct web server for RHEL
  import_tasks: redhat.yml
  when: ansible_facts['os_family']|lower == 'redhat'

- name: Install the correct web server for Debian
  import_tasks: debian.yml
  when: ansible_facts['os_family']|lower == 'debian'

# roles/example/tasks/redhat.yml
- name: Install web server
  ansible.builtin.yum:
    name: "httpd"
    state: present

# roles/example/tasks/debian.yml
- name: Install web server
  ansible.builtin.apt:
    name: "apache2"
    state: present
```

Roles may also include modules and other plugin types in a directory called `library`. For more information, please refer to [Embedding modules and plugins in roles](#) below.

Storing and finding roles

By default, Ansible looks for roles in the following locations:

- in collections, if you are using them
- in a directory called `roles/`, relative to the playbook file
- in the configured [roles_path](#) ([../reference_appendices/config.html#default-roles-path](#)).
The default search path is
`~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles`.
- in the directory where the playbook file is located

If you store your roles in a different location, set the [roles_path](#) ([../reference_appendices/config.html#default-roles-path](#)) configuration option so Ansible can find your roles. Checking shared roles into a single location makes them easier to use in multiple playbooks. See [Configuring Ansible](#) ([../installation_guide/intro_configuration.html#intro-configuration](#)) for details about managing settings in `ansible.cfg`.

Alternatively, you can call a role with a fully qualified path:

```
---
- hosts: webservers
  roles:
    - role: '/path/to/my/roles/common'
```

Using roles

You can use roles in three ways:

- at the play level with the `roles` option: This is the classic way of using roles in a play.
- at the tasks level with `include_role`: You can reuse roles dynamically anywhere in the `tasks` section of a play using `include_role`.
- at the tasks level with `import_role`: You can reuse roles statically anywhere in the `tasks` section of a play using `import_role`.

Using roles at the play level

The classic (original) way to use roles is with the `roles` option for a given play:

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

When you use the `roles` option at the play level, for each role 'x':

- If `roles/x/tasks/main.yml` exists, Ansible adds the tasks in that file to the play.
- If `roles/x/handlers/main.yml` exists, Ansible adds the handlers in that file to the play.
- If `roles/x/vars/main.yml` exists, Ansible adds the variables in that file to the play.
- If `roles/x/defaults/main.yml` exists, Ansible adds the variables in that file to the play.
- If `roles/x/meta/main.yml` exists, Ansible adds any role dependencies in that file to the list of roles.
- Any copy, script, template or include tasks (in the role) can reference files in `roles/x/{files,templates,tasks}/` (dir depends on task) without having to path them relatively or absolutely.

When you use the `roles` option at the play level, Ansible treats the roles as static imports and processes them during playbook parsing. Ansible executes your playbook in this order:

- Any `pre_tasks` defined in the play.
- Any handlers triggered by `pre_tasks`.
- Each role listed in `roles:`, in the order listed. Any role dependencies defined in the role's `meta/main.yml` run first, subject to tag filtering and conditionals. See [Using role dependencies](#) for more details.
- Any `tasks` defined in the play.
- Any handlers triggered by the roles or tasks.
- Any `post_tasks` defined in the play.

- Any handlers triggered by post_tasks.

Note

If using tags with tasks in a role, be sure to also tag your pre_tasks, post_tasks, and role dependencies and pass those along as well, especially if the pre/post tasks and role dependencies are used for monitoring outage window control or load balancing. See [Tags \(playbooks_tags.html#tags\)](#) for details on adding and using tags.

You can pass other keywords to the `roles` option:

```
---
- hosts: webservers
  roles:
    - common
    - role: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
        tags: typeA
    - role: foo_app_instance
      vars:
        dir: '/opt/b'
        app_port: 5001
        tags: typeB
```

When you add a tag to the `role` option, Ansible applies the tag to ALL tasks within the role.

When using `vars:` within the `roles:` section of a playbook, the variables are added to the play variables, making them available to all tasks within the play before and after the role. This behavior can be changed by [DEFAULT_PRIVATE_ROLE_VARS \(./reference_appendices/config.html#default-private-role-vars\)](#).

Including roles: dynamic reuse

You can reuse roles dynamically anywhere in the `tasks` section of a play using `include_role`. While roles added in a `roles` section run before any other tasks in a playbook, included roles run in the order they are defined. If there are other tasks before an `include_role` task, the other tasks will run first.

To include a role:

```

---
- hosts: webservers
  tasks:
    - name: Print a message
      ansible.builtin.debug:
        msg: "this task runs before the example role"

    - name: Include the example role
      include_role:
        name: example

    - name: Print a message
      ansible.builtin.debug:
        msg: "this task runs after the example role"

```

You can pass other keywords, including variables and tags, when including roles:

```

---
- hosts: webservers
  tasks:
    - name: Include the foo_app_instance role
      include_role:
        name: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
      tags: typeA
  ...

```

When you add a [tag](#) ([playbooks_tags.html#tags](#)) to an `include_role` task, Ansible applies the tag *only* to the include itself. This means you can pass `--tags` to run only selected tasks from the role, if those tasks themselves have the same tag as the include statement. See [Selectively running tagged tasks in re-usable files \(playbooks_tags.html#selective-reuse\)](#) for details.

You can conditionally include a role:

```

---
- hosts: webservers
  tasks:
    - name: Include the some_role role
      include_role:
        name: some_role
      when: "ansible_facts['os_family'] == 'RedHat'"

```

Importing roles: static reuse

You can reuse roles statically anywhere in the `tasks` section of a play using `import_role`. The behavior is the same as using the `roles` keyword. For example:

```
---
- hosts: webservers
  tasks:
    - name: Print a message
      ansible.builtin.debug:
        msg: "before we run our role"

    - name: Import the example role
      import_role:
        name: example

    - name: Print a message
      ansible.builtin.debug:
        msg: "after we ran our role"
```

You can pass other keywords, including variables and tags, when importing roles:

```
---
- hosts: webservers
  tasks:
    - name: Import the foo_app_instance role
      import_role:
        name: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
    ...
```

When you add a tag to an `import_role` statement, Ansible applies the tag to *all* tasks within the role. See [Tag inheritance: adding tags to multiple tasks \(playbooks tags.html#tag-inheritance\)](#) for details.

Role argument validation

Beginning with version 2.11, you may choose to enable role argument validation based on an argument specification. This specification is defined in the `meta/argument_specs.yml` file (or with the `.yaml` file extension). When this argument specification is defined, a new task is inserted at the beginning of role execution that will validate the parameters supplied for the role against the specification. If the parameters fail validation, the role will fail execution.

Note

Ansible also supports role specifications defined in the role `meta/main.yml` file, as well. However, any role that defines the specs within this file will not work on versions below 2.11. For this reason, we recommend using the `meta/argument_specs.yml` file to maintain this site

backward compatibility.

❗ Note

When role argument validation is used on a role that has defined dependencies, then validation on those dependencies will run before the dependent role, even if argument validation fails for the dependent role.

Specification format

The role argument specification must be defined in a top-level `argument_specs` block within the role `meta/argument_specs.yml` file. All fields are lower-case.

entry-point-name:	<ul style="list-style-type: none">• The name of the role entry point.• This should be <code>main</code> in the case of an unspecified entry point.• This will be the base name of the tasks file to execute, with no <code>.yml</code> or <code>.yaml</code> extension.
short_description:	<ul style="list-style-type: none">• A short, one-line description of the entry point.• The <code>short_description</code> is displayed by <code>ansible-doc</code>.
description:	<ul style="list-style-type: none">• A longer description that may contain multiple lines.
author:	<ul style="list-style-type: none">• Name of the entry point authors.• Use a multi-line list if there is more than one author.
options:	<ul style="list-style-type: none">• Options are often called “parameters” or “arguments”.• For each role option (argument), you may include:<ul style="list-style-type: none">option-name: <ul style="list-style-type: none">• The name of the option/argument.description: <ul style="list-style-type: none">• Detailed explanation of what the option does, written in full sentences.type: <ul style="list-style-type: none">• The data type of the option. See ../dev_guide/developing_roles.md#role-argument-spec for allowed values for <code>type</code>.• If an option is of type <code>list</code>, <code>dict</code>, or <code>boolean</code>, you must specify <code>required: true</code>.required: <ul style="list-style-type: none">• Only needed if <code>true</code>.• If missing, the option is not required.default: <ul style="list-style-type: none">• The default value for the option.

- If `required` is false/missing, ['null' if missing).
- Ensure that the default value value in the code. The actual (always come from `defaults/m`
- The default field must not be unless it requires additional in
- If the option is a boolean valu values recognized by Ansible: Choose the one that reads be

choices:

- List of option values.
- Should be absent if empty.

elements:

- Specifies the data type for list

options:

- If this option takes a dict or lis structure here.

Sample specification

```
# roles/myapp/meta/argument_specs.yml
---
argument_specs:
  # roles/myapp/tasks/main.yml entry point
  main:
    short_description: The main entry point for the myapp role.
    options:
      myapp_int:
        type: "int"
        required: false
        default: 42
        description: "The integer value, defaulting to 42."

      myapp_str:
        type: "str"
        required: true
        description: "The string value"

  # roles/maypp/tasks/alternate.yml entry point
  alternate:
    short_description: The alternate entry point for the myapp role.
    options:
      myapp_int:
        type: "int"
        required: false
        default: 1024
        description: "The integer value, defaulting to 1024."
```

Running a role multiple times in one playbook

Ansible only executes each role once, even if you define it multiple times, unless the parameters defined on the role are different for each definition. For example, Ansible only runs the role `foo` once in a play like this:

```
---
- hosts: webservers
  roles:
    - foo
    - bar
    - foo
```

You have two options to force Ansible to run a role more than once.

Passing different parameters

If you pass different parameters in each role definition, Ansible runs the role more than once. Providing different variable values is not the same as passing different role parameters. You must use the `roles` keyword for this behavior, since `import_role` and `include_role` do not accept role parameters.

This playbook runs the `foo` role twice:

```
---
- hosts: webservers
  roles:
    - { role: foo, message: "first" }
    - { role: foo, message: "second" }
```

This syntax also runs the `foo` role twice;

```
---
- hosts: webservers
  roles:
    - role: foo
      message: "first"
    - role: foo
      message: "second"
```

In these examples, Ansible runs `foo` twice because each role definition has different parameters.

Using `allow_duplicates: true`

Search this site

Add `allow_duplicates: true` to the `meta/main.yml` file for the role:

```
# playbook.yml
---
- hosts: webservers
  roles:
    - foo
    - foo

# roles/foo/meta/main.yml
---
allow_duplicates: true
```

In this example, Ansible runs `foo` twice because we have explicitly enabled it to do so.

Using role dependencies

Role dependencies let you automatically pull in other roles when using a role. Ansible does not execute role dependencies when you include or import a role. You must use the `roles` keyword if you want Ansible to execute role dependencies.

Role dependencies are prerequisites, not true dependencies. The roles do not have a parent/child relationship. Ansible loads all listed roles, runs the roles listed under `dependencies` first, then runs the role that lists them. The play object is the parent of all roles, including roles called by a `dependencies` list.

Role dependencies are stored in the `meta/main.yml` file within the role directory. This file should contain a list of roles and parameters to insert before the specified role. For example:

```
# roles/myapp/meta/main.yml
---
dependencies:
- role: common
  vars:
    some_parameter: 3
- role: apache
  vars:
    apache_port: 80
- role: postgres
  vars:
    dbname: blarg
    other_parameter: 12
```

Ansible always executes roles listed in `dependencies` before the role that lists them. Ansible executes this pattern recursively when you use the `roles` keyword. For example, if you list role `foo` under `roles:`, role `foo` lists role `bar` under `dependencies` in its `meta/main.yml`

file, and role `bar` lists role `baz` under `dependencies` in its `meta/main.yml`, Ansible executes `baz`, then `bar`, then `foo`.

Running role dependencies multiple times in one play

Ansible treats duplicate role dependencies like duplicate roles listed under `roles:`: Ansible only executes role dependencies once, even if defined multiple times, unless the parameters, tags, or when clause defined on the role are different for each definition. If two roles in a play both list a third role as a dependency, Ansible only runs that role dependency once, unless you pass different parameters, tags, when clause, or use `allow_duplicates: true` in the role you want to run multiple times. See [Galaxy role dependencies](https://docs.ansible.com/ansible/latest/reference_appendices/glossary.html#term-Lazy-Evaluation) ([../galaxy/user_guide.html#galaxy-dependencies](https://docs.ansible.com/ansible/latest/reference_appendices/glossary.html#term-Lazy-Evaluation)) for more details.

Note

Role deduplication does not consult the invocation signature of parent roles. Additionally, when using `vars:` instead of role params, there is a side effect of changing variable scoping. Using `vars:` results in those variables being scoped at the play level. In the below example, using `vars:` would cause `n` to be defined as `4` through the entire play, including roles called before it.

In addition to the above, users should be aware that role de-duplication occurs before variable evaluation. This means that [Lazy Evaluation](https://docs.ansible.com/ansible/latest/reference_appendices/glossary.html#term-Lazy-Evaluation) ([../reference_appendices/glossary.html#term-Lazy-Evaluation](https://docs.ansible.com/ansible/latest/reference_appendices/glossary.html#term-Lazy-Evaluation)) may make seemingly different role invocations equivalently the same, preventing the role from running more than once.

For example, a role named `car` depends on a role named `wheel` as follows:

```
---
dependencies:
  - role: wheel
    n: 1
  - role: wheel
    n: 2
  - role: wheel
    n: 3
  - role: wheel
    n: 4
```

And the `wheel` role depends on two roles: `tire` and `brake`. The `meta/main.yml` for `wheel` would then contain the following:

```
---
dependencies:
  - role: tire
  - role: brake
```

And the `meta/main.yml` for `tire` and `brake` would contain the following:

```
---
allow_duplicates: true
```

The resulting order of execution would be as follows:

```
tire(n=1)
brake(n=1)
wheel(n=1)
tire(n=2)
brake(n=2)
wheel(n=2)
...
car
```

To use `allow_duplicates: true` with role dependencies, you must specify it for the role listed under `dependencies`, not for the role that lists it. In the example above, `allow_duplicates: true` appears in the `meta/main.yml` of the `tire` and `brake` roles. The `wheel` role does not require `allow_duplicates: true`, because each instance defined by `car` uses different parameter values.

Note

See [Using Variables \(playbooks_variables.html#playbooks-variables\)](#) for details on how Ansible chooses among variable values defined in different places (variable inheritance and scope). Also deduplication happens ONLY at the play level, so multiple plays in the same playbook may rerun the roles.

Embedding modules and plugins in roles

If you write a custom module (see [Should you develop a module? \(../dev_guide/developing_modules.html#developing-modules\)](#)) or a plugin (see [Developing plugins \(../dev_guide/developing_plugins.html#developing-plugins\)](#)), you might wish to distribute it as part of a role. For example, if you write a module that helps configure your

company's internal software, and you want other people in your organization to use this module, but you do not want to tell everyone how to configure their Ansible library path, you can include the module in your `internal_config` role.

To add a module or a plugin to a role: Alongside the 'tasks' and 'handlers' structure of a role, add a directory named 'library' and then include the module directly inside the 'library' directory.

Assuming you had this:

```
roles/
  my_custom_modules/
    library/
      module1
      module2
```

The module will be usable in the role itself, as well as any roles that are called *after* this role, as follows:

```
---
- hosts: webservers
  roles:
    - my_custom_modules
    - some_other_role_using_my_custom_modules
    - yet_another_role_using_my_custom_modules
```

If necessary, you can also embed a module in a role to modify a module in Ansible's core distribution. For example, you can use the development version of a particular module before it is released in production releases by copying the module and embedding the copy in a role. Use this approach with caution, as API signatures may change in core components, and this workaround is not guaranteed to work.

The same mechanism can be used to embed and distribute plugins in a role, using the same schema. For example, for a filter plugin:

```
roles/
  my_custom_filter/
    filter_plugins
      filter1
      filter2
```

These filters can then be used in a Jinja template in any role called after 'my_custom_filter'.

[Ansible Galaxy](https://galaxy.ansible.com) (<https://galaxy.ansible.com>) is a free site for finding, downloading, rating, and reviewing all kinds of community-developed Ansible roles and can be a great way to get a jumpstart on your automation projects.

The client `ansible-galaxy` is included in Ansible. The Galaxy client allows you to download roles from Ansible Galaxy, and also provides an excellent default framework for creating your own roles.

Read the [Ansible Galaxy documentation](https://galaxy.ansible.com/docs/) (<https://galaxy.ansible.com/docs/>) page for more information

❗ See also

[Galaxy User Guide](#) ([../galaxy/user_guide.html#ansible-galaxy](#))

How to create new roles, share roles on Galaxy, role management

[YAML Syntax](#) ([../reference_appendices/YAMLSyntax.html#yaml-syntax](#))

Learn about YAML syntax

[Working with playbooks](#) ([playbooks.html#working-with-playbooks](#))

Review the basic Playbook language features

[Tips and tricks](#) ([playbooks_best_practices.html#playbooks-best-practices](#))

Tips and tricks for playbooks

[Using Variables](#) ([playbooks_variables.html#playbooks-variables](#))

Variables in playbooks

[Conditionals](#) ([playbooks_conditionals.html#playbooks-conditionals](#))

Conditionals in playbooks

[Loops](#) ([playbooks_loops.html#playbooks-loops](#))

Loops in playbooks

[Tags](#) ([playbooks_tags.html#tags](#))

Using tags to select or skip roles/tasks in long playbooks

[Collection Index](#) ([../collections/index.html#list-of-collections](#))

Browse existing collections, modules, and plugins

[Should you develop a module?](#) ([../dev_guide/developing_modules.html#developing-modules](#))

Extending Ansible by writing your own modules

[GitHub Ansible examples](#) (<https://github.com/ansible/ansible-examples>)

Search this site

Complete playbook files from the GitHub project source

Mailing List (<https://groups.google.com/group/ansible-project>)

Questions? Help? Ideas? Stop by the list on Google Groups

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Re-using Ansible artifacts

You can write a simple playbook in one very large file, and most users learn the one-file approach first. However, breaking tasks up into different files is an excellent way to organize complex sets of tasks and reuse them. Smaller, more distributed artifacts let you re-use the same variables, tasks, and plays in multiple playbooks to address different use cases. You can use distributed artifacts across multiple parent playbooks or even multiple times within one playbook. For example, you might want to update your customer database as part of several different playbooks. If you put all the tasks related to updating your database in a tasks file, you can re-use them in many playbooks while only maintaining them in one place.

- [Creating re-usable files and roles](#)
- [Re-using playbooks](#)
- [Re-using files and roles](#)
 - [Includes: dynamic re-use](#)
 - [Imports: static re-use](#)
 - [Comparing includes and imports: dynamic and static re-use](#)
- [Re-using tasks as handlers](#)
 - [Triggering included \(dynamic\) handlers](#)
 - [Triggering imported \(static\) handlers](#)

Creating re-usable files and roles

Ansible offers four distributed, re-usable artifacts: variables files, task files, playbooks, and roles.

- A variables file contains only variables.
- A task file contains only tasks.
- A playbook contains at least one play, and may contain variables, tasks, and other content. You can re-use tightly focused playbooks, but you can only re-use them statically, not dynamically.
- A role contains a set of related tasks, variables, defaults, handlers, and even modules or other plugins in a defined file-tree. Unlike variables files, task files, or playbooks, roles can be easily uploaded and shared via Ansible Galaxy. See [Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#) for details about creating and using roles.

New in version 2.4.

Re-using playbooks

You can incorporate multiple playbooks into a main playbook. However, you can only use imports to re-use playbooks. For example:

```
- import_playbook: webservers.yml
- import_playbook: databases.yml
```

Importing incorporates playbooks in other playbooks statically. Ansible runs the plays and tasks in each imported playbook in the order they are listed, just as if they had been defined directly in the main playbook.

You can select which playbook you want to import at runtime by defining your imported playbook filename with a variable, then passing the variable with either `--extra-vars` or the `vars` keyword. For example:

```
- import_playbook: "/path/to/{{ import_from_extra_var }}"
- import_playbook: "{{ import_from_vars }}"
vars:
  import_from_vars: /path/to/one_playbook.yml
```

If you run this playbook with `ansible-playbook my_playbook -e import_from_extra_var=other_playbook.yml`, Ansible imports both `one_playbook.yml` and `other_playbook.yml`.

Re-using files and roles

Ansible offers two ways to re-use files and roles in a playbook: dynamic and static.

- For dynamic re-use, add an `include_*` task in the tasks section of a play:
 - [include_role \(../collections/ansible/builtin/include_role_module.html#include-role-module\)](#)
 - [include_tasks \(../collections/ansible/builtin/include_tasks_module.html#include-tasks-module\)](#)
 - [include_vars \(../collections/ansible/builtin/include_vars_module.html#include-vars-module\)](#)
- For static re-use, add an `import_*` task in the tasks section of a play:
 - [import_role \(../collections/ansible/builtin/import_role_module.html#import-role-module\)](#)
 - [import_tasks \(../collections/ansible/builtin/import_tasks_module.html#import-tasks-module\)](#)

Task include and import statements can be used at arbitrary depth.

You can still use the bare [roles \(playbooks_reuse_roles.html#roles-keyword\)](#) keyword at the play level to incorporate a role in a playbook statically. However, the bare [include \(../collections/ansible/builtin/include_module.html#include-module\)](#) keyword, once used for both task files and playbook-level includes, is now deprecated.

Includes: dynamic re-use

Including roles, tasks, or variables adds them to a playbook dynamically. Ansible processes included files and roles as they come up in a playbook, so included tasks can be affected by the results of earlier tasks within the top-level playbook. Included roles and tasks are similar to handlers - they may or may not run, depending on the results of other tasks in the top-level playbook.

The primary advantage of using `include_*` statements is looping. When a loop is used with an include, the included tasks or role will be executed once for each item in the loop.

The filenames for included roles, tasks, and vars are templated before inclusion.

You can pass variables into includes. See [Variable precedence: Where should I put a variable? \(playbooks_variables.html#ansible-variable-precedence\)](#) for more details on variable inheritance and precedence.

Imports: static re-use

Importing roles, tasks, or playbooks adds them to a playbook statically. Ansible pre-processes imported files and roles before it runs any tasks in a playbook, so imported content is never affected by other tasks within the top-level playbook.

The filenames for imported roles and tasks support templating, but the variables must be available when Ansible is pre-processing the imports. This can be done with the `vars` keyword or by using `--extra-vars`.

You can pass variables to imports. You must pass variables if you want to run an imported file more than once in a playbook. For example:

```
tasks:
- import_tasks: wordpress.yml
  vars:
    wp_user: timmy

- import_tasks: wordpress.yml
  vars:
    wp_user: alice

- import_tasks: wordpress.yml
  vars:
    wp_user: bob
```

See [Variable precedence: Where should I put a variable? \(playbooks_variables.html#ansible-variable-precedence\)](#) for more details on variable inheritance and precedence.

Comparing includes and imports: dynamic and static re-use

Each approach to re-using distributed Ansible artifacts has advantages and limitations. You may choose dynamic re-use for some playbooks and static re-use for others. Although you can use both dynamic and static re-use in a single playbook, it is best to select one approach per playbook. Mixing static and dynamic re-use can introduce difficult-to-diagnose bugs into your playbooks. This table summarizes the main differences so you can choose the best approach for each playbook you create.

	Include_*	Import_*
Type of re-use	Dynamic	Static
When processed	At runtime, when encountered	Pre-processed during play
Task or play	All includes are tasks	<code>import_playbook</code> cannot
Task options	Apply only to include task itself	Apply to all child tasks in
Calling from loops	Executed once for each loop item	Cannot be used in a loop
Using <code>--list-tags</code>	Tags within includes not listed	All tags appear with <code>--li</code>
Using <code>--list-tasks</code>	Tasks within includes not listed	All tasks appear with <code>--l</code>
Notifying handlers	Cannot trigger handlers within includes	Can trigger individual imp
Using <code>--start-at-task</code>	Cannot start at tasks within includes	Can start at imported task

	Include_*	Import_*
Using inventory variables	Can <code>include_*: {{ inventory_var }}</code>	Cannot <code>import_*: {{ inv</code>
With playbooks	No <code>include_playbook</code>	Can import full playbooks
With variables files	Can include variables files	Use <code>vars_files:</code> to impo

❗ Note

- There are also big differences in resource consumption and performance, imports are quite lean and fast, while includes require a lot of management and accounting.

Re-using tasks as handlers

You can also use includes and imports in the [Handlers: running operations on change \(playbooks handlers.html#handlers\)](#) section of a playbook. For instance, if you want to define how to restart Apache, you only have to do that once for all of your playbooks. You might make a `restarts.yml` file that looks like:

```
# restarts.yml
- name: Restart apache
  ansible.builtin.service:
    name: apache
    state: restarted

- name: Restart mysql
  ansible.builtin.service:
    name: mysql
    state: restarted
```

You can trigger handlers from either an import or an include, but the procedure is different for each method of re-use. If you include the file, you must notify the include itself, which triggers all the tasks in `restarts.yml`. If you import the file, you must notify the individual task(s) within `restarts.yml`. You can mix direct tasks and handlers with included or imported tasks and handlers.

Triggering included (dynamic) handlers

Includes are executed at run-time, so the name of the include exists during play execution, but the included tasks do not exist until the include itself is triggered. To use the `Restart apache` task with dynamic re-use, refer to the name of the include itself. This approach triggers all tasks in the included file as handlers. For example, with the task file shown above:

```
- name: Trigger an included (dynamic) handler
hosts: localhost
handlers:
  - name: Restart services
    include_tasks: restarts.yml
tasks:
  - command: "true"
    notify: Restart services
```

Triggering imported (static) handlers

Imports are processed before the play begins, so the name of the import no longer exists during play execution, but the names of the individual imported tasks do exist. To use the `Restart apache` task with static re-use, refer to the name of each task or tasks within the imported file. For example, with the task file shown above:

```
- name: Trigger an imported (static) handler
hosts: localhost
handlers:
  - name: Restart services
    import_tasks: restarts.yml
tasks:
  - command: "true"
    notify: Restart apache
  - command: "true"
    notify: Restart mysql
```

❗ See also

Utilities modules

(https://docs.ansible.com/ansible/2.9/modules/list_of_utilities_modules.html#utilities-modules)

Documentation of the `include*` and `import*` modules discussed here.

Working with playbooks ([playbooks.html#working-with-playbooks](#))

Review the basic Playbook language features

Using Variables ([playbooks_variables.html#playbooks-variables](#))

All about variables in playbooks

Conditionals ([playbooks_conditionals.html#playbooks-conditionals](#))

Conditionals in playbooks

Loops ([playbooks_loops.html#playbooks-loops](#))

Loops in playbooks

[Tips and tricks \(playbooks_best_practices.html#playbooks-best-practices\)](#)

Tips and tricks for playbooks

[Galaxy User Guide \(../galaxy/user_guide.html#ansible-galaxy\)](#)

How to share roles on galaxy, role management

[GitHub Ansible examples \(https://github.com/ansible/ansible-examples\)](https://github.com/ansible/ansible-examples)

Complete playbook files from the GitHub project source

[Mailing List \(https://groups.google.com/group/ansible-project\)](https://groups.google.com/group/ansible-project)

Questions? Help? Ideas? Stop by the list on Google Groups

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Tags

If you have a large playbook, it may be useful to run only specific parts of it instead of running the entire playbook. You can do this with Ansible tags. Using tags to execute or skip selected tasks is a two-step process:

1. Add tags to your tasks, either individually or with tag inheritance from a block, play, role, or import.
 2. Select or skip tags when you run your playbook.
- [Adding tags with the tags keyword](#)
 - [Adding tags to individual tasks](#)
 - [Adding tags to includes](#)
 - [Tag inheritance: adding tags to multiple tasks](#)
 - [Adding tags to blocks](#)
 - [Adding tags to plays](#)
 - [Adding tags to roles](#)
 - [Adding tags to imports](#)
 - [Tag inheritance for includes: blocks and the `apply` keyword](#)
 - [Special tags: always and never](#)
 - [Selecting or skipping tags when you run a playbook](#)
 - [Previewing the results of using tags](#)
 - [Selectively running tagged tasks in re-usable files](#)
 - [Configuring tags globally](#)

Adding tags with the tags keyword

You can add tags to a single task or include. You can also add tags to multiple tasks by defining them at the level of a block, play, role, or import. The keyword `tags` addresses all these use cases. The `tags` keyword always defines tags and adds them to tasks; it does not select or skip tasks for execution. You can only select or skip tasks based on tags at the command line when you run a playbook. See [Selecting or skipping tags when you run a playbook](#) for more details.

Adding tags to individual tasks

At the simplest level, you can apply one or more tags to an individual task. You can add tags to tasks in playbooks, in task files, or within a role. Here is an example that tags two tasks with different tags:

```
tasks:
- name: Install the servers
  ansible.builtin.yum:
    name:
      - httpd
      - memcached
    state: present
  tags:
    - packages
    - webservers

- name: Configure the service
  ansible.builtin.template:
    src: templates/src.j2
    dest: /etc/foo.conf
  tags:
    - configuration
```

You can apply the same tag to more than one individual task. This example tags several tasks with the same tag, “ntp”:

```

---
# file: roles/common/tasks/main.yml

- name: Install ntp
  ansible.builtin.yum:
    name: ntp
    state: present
  tags: ntp

- name: Configure ntp
  ansible.builtin.template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
  notify:
    - restart ntpd
  tags: ntp

- name: Enable and run ntpd
  ansible.builtin.service:
    name: ntpd
    state: started
    enabled: yes
  tags: ntp

- name: Install NFS utils
  ansible.builtin.yum:
    name:
      - nfs-utils
      - nfs-util-lib
    state: present
  tags: filesharing

```

If you ran these four tasks in a playbook with `--tags ntp`, Ansible would run the three tasks tagged `ntp` and skip the one task that does not have that tag.

Adding tags to includes

You can apply tags to dynamic includes in a playbook. As with tags on an individual task, tags on an `include_*` task apply only to the include itself, not to any tasks within the included file or role. If you add `mytag` to a dynamic include, then run that playbook with `--tags mytag`, Ansible runs the include itself, runs any tasks within the included file or role tagged with `mytag`, and skips any tasks within the included file or role without that tag. See [Selectively running tagged tasks in re-usable files](#) for more details.

You add tags to includes the same way you add tags to any other task:

```

---
# file: roles/common/tasks/main.yml

- name: Dynamic re-use of database tasks
  include_tasks: db.yml
  tags: db

```

You can add a tag only to the dynamic include of a role. In this example, the `foo` tag will *not* apply to tasks inside the `bar` role:

```
---
- hosts: webserver
  tasks:
    - name: Include the bar role
      include_role:
        name: bar
      tags:
        - foo
```

With plays, blocks, the `role` keyword, and static imports, Ansible applies tag inheritance, adding the tags you define to every task inside the play, block, role, or imported file. However, tag inheritance does *not* apply to dynamic re-use with `include_role` and `include_tasks`. With dynamic re-use (includes), the tags you define apply only to the include itself. If you need tag inheritance, use a static import. If you cannot use an import because the rest of your playbook uses includes, see [Tag inheritance for includes: blocks and the `apply` keyword](#) for ways to work around this behavior.

Tag inheritance: adding tags to multiple tasks

If you want to apply the same tag or tags to multiple tasks without adding a `tags` line to every task, you can define the tags at the level of your play or block, or when you add a role or import a file. Ansible applies the tags down the dependency chain to all child tasks. With roles and imports, Ansible appends the tags set by the `roles` section or import to any tags set on individual tasks or blocks within the role or imported file. This is called tag inheritance. Tag inheritance is convenient, because you do not have to tag every task. However, the tags still apply to the tasks individually.

Adding tags to blocks

If you want to apply a tag to many, but not all, of the tasks in your play, use a [block](#) ([playbooks_blocks.html#playbooks-blocks](#)) and define the tags at that level. For example, we could edit the NTP example shown above to use a block:

```
# myrole/tasks/main.yml
- name: ntp tasks
  tags: ntp
  block:
    - name: Install ntp
      ansible.builtin.yum:
        name: ntp
        state: present

    - name: Configure ntp
      ansible.builtin.template:
        src: ntp.conf.j2
        dest: /etc/ntp.conf
      notify:
        - restart ntpd

    - name: Enable and run ntpd
      ansible.builtin.service:
        name: ntpd
        state: started
        enabled: yes

- name: Install NFS utils
  ansible.builtin.yum:
    name:
      - nfs-utils
      - nfs-util-lib
    state: present
  tags: filesharing
```

Adding tags to plays

If all the tasks in a play should get the same tag, you can add the tag at the level of the play. For example, if you had a play with only the NTP tasks, you could tag the entire play:

```

- hosts: all
  tags: ntp
  tasks:
    - name: Install ntp
      ansible.builtin.yum:
        name: ntp
        state: present

    - name: Configure ntp
      ansible.builtin.template:
        src: ntp.conf.j2
        dest: /etc/ntp.conf
      notify:
        - restart ntpd

    - name: Enable and run ntpd
      ansible.builtin.service:
        name: ntpd
        state: started
        enabled: yes

- hosts: fileserver
  tags: filesharing
  tasks:
    ...

```

Adding tags to roles

There are three ways to add tags to roles:

1. Add the same tag or tags to all tasks in the role by setting tags under `roles`. See examples in this section.
2. Add the same tag or tags to all tasks in the role by setting tags on a static `import_role` in your playbook. See examples in [Adding tags to imports](#).
3. Add a tag or tags to individual tasks or blocks within the role itself. This is the only approach that allows you to select or skip some tasks within the role. To select or skip tasks within the role, you must have tags set on individual tasks or blocks, use the dynamic `include_role` in your playbook, and add the same tag or tags to the include. When you use this approach, and then run your playbook with `--tags foo`, Ansible runs the include itself plus any tasks in the role that also have the tag `foo`. See [Adding tags to includes](#) for details.

When you incorporate a role in your playbook statically with the `roles` keyword, Ansible adds any tags you define to all the tasks in the role. For example:

```

roles:
- role: webserver
  vars:
    port: 5000
    tags: [ web, foo ]

```

or:

```
---
- hosts: webservers
  roles:
    - role: foo
      tags:
        - bar
        - baz
    # using YAML shorthand, this is equivalent to:
    # - { role: foo, tags: ["bar", "baz"] }
```

Adding tags to imports

You can also apply a tag or tags to all the tasks imported by the static `import_role` and `import_tasks` statements:

```
---
- hosts: webservers
  tasks:
    - name: Import the foo role
      import_role:
        name: foo
      tags:
        - bar
        - baz

    - name: Import tasks from foo.yml
      import_tasks: foo.yml
      tags: [ web, foo ]
```

Tag inheritance for includes: blocks and the `apply` keyword

By default, Ansible does not apply tag inheritance to dynamic re-use with `include_role` and `include_tasks`. If you add tags to an include, they apply only to the include itself, not to any tasks in the included file or role. This allows you to execute selected tasks within a role or task file - see [Selectively running tagged tasks in re-usable files](#) when you run your playbook.

If you want tag inheritance, you probably want to use imports. However, using both includes and imports in a single playbook can lead to difficult-to-diagnose bugs. For this reason, if your playbook uses `include_*` to re-use roles or tasks, and you need tag inheritance on one include, Ansible offers two workarounds. You can use the `apply` keyword:

```
- name: Apply the db tag to the include and to all tasks in db.yml
  include_tasks:
    file: db.yml
    # adds 'db' tag to tasks within db.yml
  apply:
    tags: db
  # adds 'db' tag to this 'include_tasks' itself
  tags: db
```

Or you can use a block:

```
- block:
  - name: Include tasks from db.yml
    include_tasks: db.yml
  tags: db
```

Special tags: always and never

Ansible reserves two tag names for special behavior: always and never. If you assign the `always` tag to a task or play, Ansible will always run that task or play, unless you specifically skip it (`--skip-tags always`).

For example:

```
tasks:
- name: Print a message
  ansible.builtin.debug:
    msg: "Always runs"
  tags:
  - always

- name: Print a message
  ansible.builtin.debug:
    msg: "runs when you use tag1"
  tags:
  - tag1
```

⚠ Warning

- Fact gathering is tagged with ‘always’ by default. It is only skipped if you apply a tag and then use a different tag in `--tags` or the same tag in `--skip-tags` .

⚠ Warning

- The role argument specification validation task is tagged with ‘always’ by default. This validation will be skipped if you use `--skip-tags always` .

Search this site

New in version 2.5.

If you assign the `never` tag to a task or play, Ansible will skip that task or play unless you specifically request it (`--tags never`).

For example:

```
tasks:
- name: Run the rarely-used debug task
  ansible.builtin.debug:
    msg: '{{ showmevar }}'
    tags: [ never, debug ]
```

The rarely-used debug task in the example above only runs when you specifically request the `debug` or `never` tags.

Selecting or skipping tags when you run a playbook

Once you have added tags to your tasks, includes, blocks, plays, roles, and imports, you can selectively execute or skip tasks based on their tags when you run `ansible-playbook` (`./cli/ansible-playbook.html#ansible-playbook`). Ansible runs or skips all tasks with tags that match the tags you pass at the command line. If you have added a tag at the block or play level, with `roles`, or with an import, that tag applies to every task within the block, play, role, or imported role or file. If you have a role with lots of tags and you want to call subsets of the role at different times, either use it with dynamic includes, or split the role into multiple roles.

`ansible-playbook` (`./cli/ansible-playbook.html#ansible-playbook`) offers five tag-related command-line options:

- `--tags all` - run all tasks, ignore tags (default behavior)
- `--tags [tag1, tag2]` - run only tasks with either the tag `tag1` or the tag `tag2`
- `--skip-tags [tag3, tag4]` - run all tasks except those with either the tag `tag3` or the tag `tag4`
- `--tags tagged` - run only tasks with at least one tag
- `--tags untagged` - run only tasks with no tags

For example, to run only tasks and blocks tagged `configuration` and `packages` in a very long playbook:

```
ansible-playbook example.yml --tags "configuration,packages"
```

Search this site

To run all tasks except those tagged `packages`:

```
ansible-playbook example.yml --skip-tags "packages"
```

Previewing the results of using tags

When you run a role or playbook, you might not know or remember which tasks have which tags, or which tags exist at all. Ansible offers two command-line flags for [ansible-playbook](https://docs.ansible.com/ansible/latest/cli/ansible-playbook.html#ansible-playbook) ([../cli/ansible-playbook.html#ansible-playbook](https://docs.ansible.com/ansible/latest/cli/ansible-playbook.html#ansible-playbook)) that help you manage tagged playbooks:

- `--list-tags` - generate a list of available tags
- `--list-tasks` - when used with `--tags tagname` or `--skip-tags tagname`, generate a preview of tagged tasks

For example, if you do not know whether the tag for configuration tasks is `config` or `conf` in a playbook, role, or tasks file, you can display all available tags without running any tasks:

```
ansible-playbook example.yml --list-tags
```

If you do not know which tasks have the tags `configuration` and `packages`, you can pass those tags and add `--list-tasks`. Ansible lists the tasks but does not execute any of them.

```
ansible-playbook example.yml --tags "configuration,packages" --list-tasks
```

These command-line flags have one limitation: they cannot show tags or tasks within dynamically included files or roles. See [Comparing includes and imports: dynamic and static re-use \(playbooks_reuse.html#dynamic-vs-static\)](https://docs.ansible.com/ansible/latest/playbooks_reuse.html#dynamic-vs-static) for more information on differences between static imports and dynamic includes.

Selectively running tagged tasks in re-usable files

If you have a role or a tasks file with tags defined at the task or block level, you can selectively run or skip those tagged tasks in a playbook if you use a dynamic include instead of a static import. You must use the same tag on the included tasks and on the include statement itself. For example you might create a file with some tagged and some untagged tasks:

```
# mixed.yml
tasks:
- name: Run the task with no tags
  ansible.builtin.debug:
    msg: this task has no tags

- name: Run the tagged task
  ansible.builtin.debug:
    msg: this task is tagged with mytag
  tags: mytag

- block:
  - name: Run the first block task with mytag
    ...
  - name: Run the second block task with mytag
    ...
  tags:
  - mytag
```

And you might include the tasks file above in a playbook:

```
# myplaybook.yml
- hosts: all
  tasks:
  - name: Run tasks from mixed.yml
    include_tasks:
      name: mixed.yml
    tags: mytag
```

When you run the playbook with `ansible-playbook -i hosts myplaybook.yml --tags "mytag"`, Ansible skips the task with no tags, runs the tagged individual task, and runs the two tasks in the block.

Configuring tags globally.

If you run or skip certain tags by default, you can use the [TAGS_RUN](#) ([../reference_appendices/config.html#tags-run](#)) and [TAGS_SKIP](#) ([../reference_appendices/config.html#tags-skip](#)) options in Ansible configuration to set those defaults.

📌 See also

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#)

Playbook organization by roles

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](#)

Search this site

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

How to build your inventory

Ansible works against multiple managed nodes or “hosts” in your infrastructure at the same time, using a list or group of lists known as inventory. Once your inventory is defined, you use [patterns](#) ([intro_patterns.html#intro-patterns](#)) to select the hosts or groups you want Ansible to run against.

The default location for inventory is a file called `/etc/ansible/hosts`. You can specify a different inventory file at the command line using the `-i <path>` option. You can also use multiple inventory files at the same time as described in [Using multiple inventory sources](#), and/or pull inventory from dynamic or cloud sources or different formats (YAML, ini, and so on), as described in [Working with dynamic inventory](#) ([intro_dynamic_inventory.html#intro-dynamic-inventory](#)). Introduced in version 2.4, Ansible has [Inventory plugins](#) ([../plugins/inventory.html#inventory-plugins](#)) to make this flexible and customizable.

- [Inventory basics: formats, hosts, and groups](#)
 - [Default groups](#)
 - [Hosts in multiple groups](#)
 - [Adding ranges of hosts](#)
- [Adding variables to inventory](#)
- [Assigning a variable to one machine: host variables](#)
 - [Inventory aliases](#)
- [Assigning a variable to many machines: group variables](#)
 - [Inheriting variable values: group variables for groups of groups](#)
- [Organizing host and group variables](#)
- [How variables are merged](#)
- [Using multiple inventory sources](#)
- [Connecting to hosts: behavioral inventory parameters](#)
 - [Non-SSH connection types](#)
- [Inventory setup examples](#)
 - [Example: One inventory per environment](#)
 - [Example: Group by function](#)
 - [Example: Group by location](#)

Inventory basics: formats, hosts, and groups

The inventory file can be in one of many formats, depending on the inventory plugins you have. The most common formats are INI and YAML. A basic INI `/etc/ansible/hosts` might look like this:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

The headings in brackets are group names, which are used in classifying hosts and deciding what hosts you are controlling at what times and for what purpose. Group names should follow the same guidelines as [Creating valid variable names \(playbooks variables.html#valid-variable-names\)](#).

Here's that same basic inventory file in YAML format:

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

Default groups

There are two default groups: `all` and `ungrouped`. The `all` group contains every host. The `ungrouped` group contains all hosts that don't have another group aside from `all`. Every host will always belong to at least 2 groups (`all` and `ungrouped` or `all` and some other group). Though `all` and `ungrouped` are always present, they can be implicit and not appear in group listings like `group_names`.

Hosts in multiple groups

You can (and probably will) put each host in more than one group. For example a production webserver in a datacenter in Atlanta might be included in groups called [prod] and [atlanta] and [webserver]. You can create groups that track:

- What - An application, stack or microservice (for example, database servers, web servers, and so on).
- Where - A datacenter or region, to talk to local DNS, storage, and so on (for example, east, west).
- When - The development stage, to avoid testing on production resources (for example, prod, test).

Extending the previous YAML inventory to include what, when, and where would look like:

```
all:
  hosts:
    mail.example.com:
  children:
    webserver:
      hosts:
        foo.example.com:
        bar.example.com:
    dbserver:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
    east:
      hosts:
        foo.example.com:
        one.example.com:
        two.example.com:
    west:
      hosts:
        bar.example.com:
        three.example.com:
    prod:
      hosts:
        foo.example.com:
        one.example.com:
        two.example.com:
    test:
      hosts:
        bar.example.com:
        three.example.com:
```

You can see that `one.example.com` exists in the `dbserver`, `east`, and `prod` groups.

You can also use nested groups to simplify `prod` and `test` in this inventory, for the same result:

```

all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
  east:
    hosts:
      foo.example.com:
      one.example.com:
      two.example.com:
  west:
    hosts:
      bar.example.com:
      three.example.com:
  prod:
    children:
      east:
  test:
    children:
      west:

```

You can find more examples on how to organize your inventories and group your hosts in [Inventory setup examples](#).

Adding ranges of hosts

If you have a lot of hosts with a similar pattern, you can add them as a range rather than listing each hostname separately:

In INI:

```

[webservers]
www[01:50].example.com

```

In YAML:

```

...
webservers:
  hosts:
    www[01:50].example.com:

```

You can specify a stride (increments between sequence numbers) when defining a numeric range of hosts:

In INI:

```
[webservers]
www[01:50:2].example.com
```

In YAML:

```
...
webservers:
  hosts:
    www[01:50:2].example.com:
```

For numeric patterns, leading zeros can be included or removed, as desired. Ranges are inclusive. You can also define alphabetic ranges:

```
[databases]
db-[a:f].example.com
```

Adding variables to inventory

You can store variable values that relate to a specific host or group in inventory. To start with, you may add variables directly to the hosts and groups in your main inventory file. As you add more and more managed nodes to your Ansible inventory, however, you will likely want to store variables in separate host and group variable files. See [Defining variables in inventory \(playbooks_variables.html#define-variables-in-inventory\)](#) for details.

Assigning a variable to one machine: host variables

You can easily assign a variable to a single host, then use it later in playbooks. In INI:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

In YAML:


```
atlanta:
  hosts:
    host1:
      http_port: 80
      maxRequestsPerChild: 808
    host2:
      http_port: 303
      maxRequestsPerChild: 909
```

Unique values like non-standard SSH ports work well as host variables. You can add them to your Ansible inventory by adding the port number after the hostname with a colon:

```
badwolf.example.com:5309
```

Connection variables also work well as host variables:

```
[targets]

localhost          ansible_connection=local
other1.example.com  ansible_connection=ssh      ansible_user=myuser
other2.example.com  ansible_connection=ssh      ansible_user=myotheruser
```

Note

If you list non-standard SSH ports in your SSH config file, the `openssh` connection will find and use them, but the `paramiko` connection will not.

Inventory aliases

You can also define aliases in your inventory:

In INI:

```
jumper ansible_port=5555 ansible_host=192.0.2.50
```

In YAML:

```
...
hosts:
  jumper:
    ansible_port: 5555
    ansible_host: 192.0.2.50
```

In the above example, running Ansible against the host alias “jumper” will connect to 192.0.2.50 on port 5555. See [behavioral inventory parameters](#) to further customize the connection to hosts.

❗ Note

Values passed in the INI format using the `key=value` syntax are interpreted differently depending on where they are declared:

- When declared inline with the host, INI values are interpreted as Python literal structures (strings, numbers, tuples, lists, dicts, booleans, None). Host lines accept multiple `key=value` parameters per line. Therefore they need a way to indicate that a space is part of a value rather than a separator.
- When declared in a `:vars` section, INI values are interpreted as strings. For example `var=FALSE` would create a string equal to ‘FALSE’. Unlike host lines, `:vars` sections accept only a single entry per line, so everything after the `=` must be the value for the entry.
- If a variable value set in an INI inventory must be a certain type (for example, a string or a boolean value), always specify the type with a filter in your task. Do not rely on types set in INI inventories when consuming variables.
- Consider using YAML format for inventory sources to avoid confusion on the actual type of a variable. The YAML inventory plugin processes variable values consistently and correctly.

Generally speaking, this is not the best way to define variables that describe your system policy. Setting variables in the main inventory file is only a shorthand. See [Organizing host and group variables](#) for guidelines on storing variable values in individual files in the ‘host_vars’ directory.

Assigning a variable to many machines: group variables

If all hosts in a group share a variable value, you can apply that variable to an entire group at once. In INI:

```
[atlanta]
host1
host2

[atlanta:vars]
ntp_server=ntp.atlanta.example.com
proxy=proxy.atlanta.example.com
```

In YAML:

```
atlanta:
  hosts:
    host1:
    host2:
  vars:
    ntp_server: ntp.atlanta.example.com
    proxy: proxy.atlanta.example.com
```

Group variables are a convenient way to apply variables to multiple hosts at once. Before executing, however, Ansible always flattens variables, including inventory variables, to the host level. If a host is a member of multiple groups, Ansible reads variable values from all of those groups. If you assign different values to the same variable in different groups, Ansible chooses which value to use based on internal [rules for merging](#).

Inheriting variable values: group variables for groups of groups

You can make groups of groups using the `:children` suffix in INI or the `children:` entry in YAML. You can apply variables to these groups of groups using `:vars` or `vars:`:

In INI:

```
[atlanta]
host1
host2

[raleigh]
host2
host3

[southeast:children]
atlanta
raleigh

[southeast:vars]
some_server=foo.southeast.example.com
halon_system_timeout=30
self_destruct_countdown=60
escape_pods=2

[usa:children]
southeast
northeast
southwest
northwest
```

In YAML:

```

all:
  children:
    usa:
      children:
        southeast:
          children:
            atlanta:
              hosts:
                host1:
                host2:
            raleigh:
              hosts:
                host2:
                host3:
          vars:
            some_server: foo.southeast.example.com
            halon_system_timeout: 30
            self_destruct_countdown: 60
            escape_pods: 2
        northeast:
        northwest:
        southwest:

```

If you need to store lists or hash data, or prefer to keep host and group specific variables separate from the inventory file, see [Organizing host and group variables](#).

Child groups have a couple of properties to note:

- Any host that is member of a child group is automatically a member of the parent group.
- A child group's variables will have higher precedence (override) a parent group's variables.
- Groups can have multiple parents and children, but not circular relationships.
- Hosts can also be in multiple groups, but there will only be **one** instance of a host, merging the data from the multiple groups.

Organizing host and group variables

Although you can store variables in the main inventory file, storing separate host and group variables files may help you organize your variable values more easily. Host and group variable files must use YAML syntax. Valid file extensions include '.yaml', '.yml', '.json', or no file extension. See [YAML Syntax \(../reference_appendices/YAMLSyntax.html#yaml-syntax\)](#) if you are new to YAML.

Ansible loads host and group variable files by searching paths relative to the inventory file or the playbook file. If your inventory file at `/etc/ansible/hosts` contains a host named 'foosball' that belongs to two groups, 'raleigh' and 'webserver', that host will use variables in YAML files at the following locations:

```
/etc/ansible/group_vars/raleigh # can optionally end in '.yaml', '.yml', or '.json'
/etc/ansible/group_vars/webserver
/etc/ansible/host_vars/foosball
```

For example, if you group hosts in your inventory by datacenter, and each datacenter uses its own NTP server and database server, you can create a file called

`/etc/ansible/group_vars/raleigh` to store the variables for the `raleigh` group:

```
---
ntp_server: acme.example.org
database_server: storage.example.org
```

You can also create *directories* named after your groups or hosts. Ansible will read all the files in these directories in lexicographical order. An example with the ‘raleigh’ group:

```
/etc/ansible/group_vars/raleigh/db_settings
/etc/ansible/group_vars/raleigh/cluster_settings
```

All hosts in the ‘raleigh’ group will have the variables defined in these files available to them. This can be very useful to keep your variables organized when a single file gets too big, or when you want to use [Ansible Vault \(vault.html#playbooks-vault\)](http://vault.html#playbooks-vault) on some group variables.

You can also add `group_vars/` and `host_vars/` directories to your playbook directory. The `ansible-playbook` command looks for these directories in the current working directory by default. Other Ansible commands (for example, `ansible`, `ansible-console`, and so on) will only look for `group_vars/` and `host_vars/` in the inventory directory. If you want other commands to load group and host variables from a playbook directory, you must provide the `--playbook-dir` option on the command line. If you load inventory files from both the playbook directory and the inventory directory, variables in the playbook directory will override variables set in the inventory directory.

Keeping your inventory file and variables in a git repo (or other version control) is an excellent way to track changes to your inventory and host variables.

How variables are merged

By default variables are merged/flattened to the specific host before a play is run. This keeps Ansible focused on the Host and Task, so groups don’t really survive outside of inventory and host matching. By default, Ansible overwrites variables including the ones defined for a group

and/or host (see [DEFAULT_HASH_BEHAVIOUR](#) ([../reference_appendices/config.html#default-hash-behaviour](#))). The order/precedence is (from lowest to highest):

- all group (because it is the 'parent' of all other groups)
- parent group
- child group
- host

By default Ansible merges groups at the same parent/child level in ASCII order, and the last group loaded overwrites the previous groups. For example, an `a_group` will be merged with `b_group` and `b_group` vars that match will overwrite the ones in `a_group`.

You can change this behavior by setting the group variable `ansible_group_priority` to change the merge order for groups of the same level (after the parent/child order is resolved). The larger the number, the later it will be merged, giving it higher priority. This variable defaults to `1` if not set. For example:

```
a_group:
  vars:
    testvar: a
    ansible_group_priority: 10
b_group:
  vars:
    testvar: b
```

In this example, if both groups have the same priority, the result would normally have been `testvar == b`, but since we are giving the `a_group` a higher priority the result will be `testvar == a`.

Note

`ansible_group_priority` can only be set in the inventory source and not in `group_vars/`, as the variable is used in the loading of `group_vars`.

Using multiple inventory sources

You can target multiple inventory sources (directories, dynamic inventory scripts or files supported by inventory plugins) at the same time by giving multiple inventory parameters from the command line or by configuring `ANSIBLE_INVENTORY` ([../reference_appendices/config.html#envvar-ANSIBLE_INVENTORY](#)). This can be useful when you want to target normally separate environments, like staging and production, at the same time for a specific action.

Target two sources from the command line like this:

```
ansible-playbook get_logs.yml -i staging -i production
```

Keep in mind that if there are variable conflicts in the inventories, they are resolved according to the rules described in [How variables are merged](#) and [Variable precedence: Where should I put a variable? \(playbooks_variables.html#ansible-variable-precedence\)](#). The merging order is controlled by the order of the inventory source parameters. If `[all:vars]` in staging inventory defines `myvar = 1`, but production inventory defines `myvar = 2`, the playbook will be run with `myvar = 2`. The result would be reversed if the playbook was run with `-i production -i staging`.

Aggregating inventory sources with a directory

You can also create an inventory by combining multiple inventory sources and source types under a directory. This can be useful for combining static and dynamic hosts and managing them as one inventory. The following inventory combines an inventory plugin source, a dynamic inventory script, and a file with static hosts:

```
inventory/
  openstack.yml          # configure inventory plugin to get hosts from Openstack cloud
  dynamic-inventory.py   # add additional hosts with dynamic inventory script
  static-inventory       # add static hosts and groups
  group_vars/
    all.yml              # assign variables to all hosts
```

You can target this inventory directory simply like this:

```
ansible-playbook example.yml -i inventory
```

It can be useful to control the merging order of the inventory sources if there's variable conflicts or group of groups dependencies to the other inventory sources. The inventories are merged in ASCII order according to the filenames so the result can be controlled by adding prefixes to the files:

```
inventory/
  01-openstack.yml       # configure inventory plugin to get hosts from Openstack
cloud
  02-dynamic-inventory.py # add additional hosts with dynamic inventory script
  03-static-inventory     # add static hosts
  group_vars/
    all.yml              # assign variables to all hosts
```

If `01-openstack.yml` defines `myvar = 1` for the group `all`, `02-dynamic-inventory.py` defines `myvar = 2`, and `03-static-inventory` defines `myvar = 3`, the playbook will be run with `myvar = 3`.

For more details on inventory plugins and dynamic inventory scripts see [Inventory plugins](#) ([../plugins/inventory.html#inventory-plugins](#)) and [Working with dynamic inventory](#) ([intro_dynamic_inventory.html#intro-dynamic-inventory](#)).

Connecting to hosts: behavioral inventory parameters

As described above, setting the following variables control how Ansible interacts with remote hosts.

Host connection:

Note

Ansible does not expose a channel to allow communication between the user and the ssh process to accept a password manually to decrypt an ssh key when using the ssh connection plugin (which is the default). The use of `ssh-agent` is highly recommended.

ansible_connection

Connection type to the host. This can be the name of any of ansible's connection plugins. SSH protocol types are `smart`, `ssh` or `paramiko`. The default is `smart`. Non-SSH based types are described in the next section.

General for all connections:

ansible_host

The name of the host to connect to, if different from the alias you wish to give to it.

ansible_port

The connection port number, if not the default (22 for ssh)

ansible_user

The user name to use when connecting to the host

ansible_password

The password to use to authenticate to the host (never store this variable in plain text; always use a vault. See [Keep vaulted variables safely visible](#) ([playbooks_best_practices.html#tip-for-variables-and-vaults](#)))

Specific to the SSH connection:

ansible_ssh_private_key_file

Private key file used by ssh. Useful if using multiple keys and you don't want to use SSH agent.

ansible_ssh_common_args

This setting is always appended to the default command line for **sftp**, **scp**, and **ssh**. Useful to configure a `ProxyCommand` for a certain host (or group).

ansible_sftp_extra_args

This setting is always appended to the default **sftp** command line.

ansible_scp_extra_args

This setting is always appended to the default **scp** command line.

ansible_ssh_extra_args

This setting is always appended to the default **ssh** command line.

ansible_ssh_pipelining

Determines whether or not to use SSH pipelining. This can override the `pipelining` setting in `ansible.cfg`.

ansible_ssh_executable (added in version 2.2)

This setting overrides the default behavior to use the system **ssh**. This can override the `ssh_executable` setting in `ansible.cfg`.

Privilege escalation (see [Ansible Privilege Escalation \(become.html#become\)](#) for further details):

ansible_become

Equivalent to `ansible_sudo` or `ansible_su`, allows to force privilege escalation

ansible_become_method

Allows to set privilege escalation method

ansible_become_user

Equivalent to `ansible_sudo_user` or `ansible_su_user`, allows to set the user you become through privilege escalation

ansible_become_password

Equivalent to `ansible_sudo_password` or `ansible_su_password`, allows you to set the privilege escalation password (never store this variable in plain text; always use a vault. See [Keep vaulted variables safely visible \(playbooks_best_practices.html#tip-for-variables-and-vaults\)](#))

ansible_become_exe

Equivalent to `ansible_sudo_exe` or `ansible_su_exe`, allows you to set the executable for the escalation method selected

ansible_become_flags

Equivalent to `ansible_sudo_flags` or `ansible_su_flags`, allows you to set the flags passed to the selected escalation method. This can be also set globally in `ansible.cfg` in the `sudo_flags` option

Remote host environment parameters:

ansible_shell_type

The shell type of the target system. You should not use this setting unless you have set the `ansible_shell_executable` to a non-Bourne (sh) compatible shell. By default commands are formatted using `sh`-style syntax. Setting this to `csh` or `fish` will cause commands executed on target systems to follow those shell's syntax instead.

ansible_python_interpreter

The target host python path. This is useful for systems with more than one Python or not located at `/usr/bin/python` such as *BSD, or where `/usr/bin/python` is not a 2.X series Python. We do not use the `/usr/bin/env` mechanism as that requires the remote user's path to be set right and also assumes the `python` executable is named python, where the executable might be named something like `python2.6`.

ansible_*_interpreter

Works for anything such as ruby or perl and works just like `ansible_python_interpreter`. This replaces shebang of modules which will run on that host.

New in version 2.1.

ansible_shell_executable

This sets the shell the ansible controller will use on the target machine, overrides `executable` in `ansible.cfg` which defaults to `/bin/sh`. You should really only change it if is not possible to use `/bin/sh` (in other words, if `/bin/sh` is not installed on the target machine or cannot be run from sudo.).

Examples from an Ansible-INI host file:

```
some_host      ansible_port=2222      ansible_user=manager
aws_host       ansible_ssh_private_key_file=/home/example/.ssh/aws.pem
freebsd_host   ansible_python_interpreter=/usr/local/bin/python
ruby_module_host ansible_ruby_interpreter=/usr/bin/ruby.1.9.3
```

Non-SSH connection types

As stated in the previous section, Ansible executes playbooks over SSH but it is not limited to this connection type. With the host specific parameter `ansible_connection=<connector>`, the connection type can be changed. The following non-SSH based connectors are available:

local

This connector can be used to deploy the playbook to the control machine itself.

docker

This connector deploys the playbook directly into Docker containers using the local Docker client. The following parameters are processed by this connector:

ansible_host

The name of the Docker container to connect to.

ansible_user

The user name to operate within the container. The user must exist inside the container.

ansible_become

If set to `true` the `become_user` will be used to operate within the container.

ansible_docker_extra_args

Could be a string with any additional arguments understood by Docker, which are not command specific. This parameter is mainly used to configure a remote Docker daemon to use.

Here is an example of how to instantly deploy to created containers:

```

- name: Create a jenkins container
  community.general.docker_container:
    docker_host: myserver.net:4243
    name: my_jenkins
    image: jenkins

- name: Add the container to inventory
  ansible.builtin.add_host:
    name: my_jenkins
    ansible_connection: docker
    ansible_docker_extra_args: "--tlsverify --tlscacert=/path/to/ca.pem --
    tlscert=/path/to/client-cert.pem --tlskey=/path/to/client-key.pem -
    H=tcp://myserver.net:4243"
    ansible_user: jenkins
    changed_when: false

- name: Create a directory for ssh keys
  delegate_to: my_jenkins
  ansible.builtin.file:
    path: "/var/jenkins_home/.ssh/jupiter"
    state: directory

```

For a full list with available plugins and examples, see [Plugin list](#) ([../plugins/connection.html#connection-plugin-list](#)).

Note

If you're reading the docs from the beginning, this may be the first example you've seen of an Ansible playbook. This is not an inventory file. Playbooks will be covered in great detail later in the docs.

Inventory setup examples

See also [Sample Ansible setup](#) ([sample_setup.html#sample-setup](#)), which shows inventory along with playbooks and other Ansible artifacts.

Example: One inventory per environment

If you need to manage multiple environments it's sometimes prudent to have only hosts of a single environment defined per inventory. This way, it is harder to, for instance, accidentally change the state of nodes inside the "test" environment when you actually wanted to update some "staging" servers.

For the example mentioned above you could have an `inventory_test` file:

[dbservers]

db01.test.example.com
db02.test.example.com

[appservers]

app01.test.example.com
app02.test.example.com
app03.test.example.com

That file only includes hosts that are part of the “test” environment. Define the “staging” machines in another file called `inventory_staging`:

[dbservers]

db01.staging.example.com
db02.staging.example.com

[appservers]

app01.staging.example.com
app02.staging.example.com
app03.staging.example.com

To apply a playbook called `site.yml` to all the app servers in the test environment, use the following command:

```
ansible-playbook -i inventory_test -l appservers site.yml
```

Example: Group by function

In the previous section you already saw an example for using groups in order to cluster hosts that have the same function. This allows you, for instance, to define firewall rules inside a playbook or role affecting only database servers:

```
- hosts: dbservers
  tasks:
  - name: Allow access from 10.0.0.1
    ansible.builtin.iptables:
      chain: INPUT
      jump: ACCEPT
      source: 10.0.0.1
```

Example: Group by location

Other tasks might be focused on where a certain host is located. Let's say that

`db01.test.example.com` and `app01.test.example.com` are located in DC1 while

`db02.test.example.com` is in DC2:

[dc1]

`db01.test.example.com`

`app01.test.example.com`

[dc2]

`db02.test.example.com`

In practice, you might even end up mixing all these setups as you might need to, on one day, update all nodes in a specific data center while, on another day, update all the application servers no matter their location.

❗ See also

[Inventory plugins \(../plugins/inventory.html#inventory-plugins\)](#)

Pulling inventory from dynamic or static sources

[Working with dynamic inventory \(intro_dynamic_inventory.html#intro-dynamic-inventory\)](#)

Pulling inventory from dynamic sources, such as cloud providers

[Introduction to ad hoc commands \(intro_adhoc.html#intro-adhoc\)](#)

Examples of basic commands

[Working with playbooks \(playbooks.html#working-with-playbooks\)](#)

Learning Ansible's configuration, deployment, and orchestration language.

[Mailing List \(https://groups.google.com/group/ansible-project\)](#)

Questions? Help? Ideas? Stop by the list on Google Groups

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Working with dynamic inventory

- [Inventory script example: Cobbler](#)
- [Inventory script example: OpenStack](#)
 - [Explicit use of OpenStack inventory script](#)
 - [Implicit use of OpenStack inventory script](#)
 - [Refreshing the cache](#)
- [Other inventory scripts](#)
- [Using inventory directories and multiple inventory sources](#)
- [Static groups of dynamic groups](#)

If your Ansible inventory fluctuates over time, with hosts spinning up and shutting down in response to business demands, the static inventory solutions described in [How to build your inventory \(intro_inventory.html#inventory\)](#) will not serve your needs. You may need to track hosts from multiple sources: cloud providers, LDAP, [Cobbler \(https://cobbler.github.io\)](#), and/or enterprise CMDB systems.

Ansible integrates all of these options through a dynamic external inventory system. Ansible supports two ways to connect with external inventory: [Inventory plugins \(../plugins/inventory.html#inventory-plugins\)](#) and *inventory scripts*.

Inventory plugins take advantage of the most recent updates to the Ansible core code. We recommend plugins over scripts for dynamic inventory. You can [write your own plugin \(../dev_guide/developing_inventory.html#developing-inventory\)](#) to connect to additional dynamic inventory sources.

You can still use inventory scripts if you choose. When we implemented inventory plugins, we ensured backwards compatibility through the script inventory plugin. The examples below illustrate how to use inventory scripts.

If you prefer a GUI for handling dynamic inventory, the inventory database on AWX or [Red Hat Ansible Automation Platform \(../reference_appendices/tower.html#ansible-platform\)](#) syncs with all your dynamic inventory sources, provides web and REST access to the results,

and offers a graphical inventory editor. With a database record of all of your hosts, you can correlate past event history and see which hosts have had failures on their last playbook runs.

Inventory script example: Cobbler

Ansible integrates seamlessly with [Cobbler \(https://cobbler.github.io\)](https://cobbler.github.io), a Linux installation server originally written by Michael DeHaan and now led by James Cammarata, who works for Ansible.

While primarily used to kickoff OS installations and manage DHCP and DNS, Cobbler has a generic layer that can represent data for multiple configuration management systems (even at the same time) and serve as a 'lightweight CMDB'.

To tie your Ansible inventory to Cobbler, copy [this script \(https://raw.githubusercontent.com/ansible-community/contrib-scripts/main/inventory/cobbler.py\)](https://raw.githubusercontent.com/ansible-community/contrib-scripts/main/inventory/cobbler.py) to `/etc/ansible` and `chmod +x` the file. Run `cobblerd` any time you use Ansible and use the `-i` command line option (for example, `-i /etc/ansible/cobbler.py`) to communicate with Cobbler using Cobbler's XMLRPC API.

Add a `cobbler.ini` file in `/etc/ansible` so Ansible knows where the Cobbler server is and some cache improvements can be used. For example:

```
[cobbler]

# Set Cobbler's hostname or IP address
host = http://127.0.0.1/cobbler_api

# API calls to Cobbler can be slow. For this reason, we cache the results of an API
# call. Set this to the path you want cache files to be written to. Two files
# will be written to this directory:
#   - ansible-cobbler.cache
#   - ansible-cobbler.index

cache_path = /tmp

# The number of seconds a cache file is considered valid. After this many
# seconds, a new API call will be made, and the cache file will be updated.

cache_max_age = 900
```

First test the script by running `/etc/ansible/cobbler.py` directly. You should see some JSON data output, but it may not have anything in it just yet.

Let's explore what this does. In Cobbler, assume a scenario somewhat like the following:


```
cobbler profile add --name=webserver --distro=CentOS6-x86_64
cobbler profile edit --name=webserver --mgmt-classes="webserver" --ksmeta="a=2 b=3"
cobbler system edit --name=foo --dns-name="foo.example.com" --mgmt-classes="atlanta" --ksmeta="c=4"
cobbler system edit --name=bar --dns-name="bar.example.com" --mgmt-classes="atlanta" --ksmeta="c=5"
```

In the example above, the system 'foo.example.com' is addressable by ansible directly, but is also addressable when using the group names 'webserver' or 'atlanta'. Since Ansible uses SSH, it contacts system foo over 'foo.example.com', only, never just 'foo'. Similarly, if you tried "ansible foo", it would not find the system... but "ansible 'foo*'" would do, because the system DNS name starts with 'foo'.

The script provides more than host and group info. In addition, as a bonus, when the 'setup' module is run (which happens automatically when using playbooks), the variables 'a', 'b', and 'c' will all be auto-populated in the templates:

```
# file: /srv/motd.j2
Welcome, I am templated with a value of a={{ a }}, b={{ b }}, and c={{ c }}
```

Which could be executed just like this:

```
ansible webserver -m setup
ansible webserver -m template -a "src=/tmp/motd.j2 dest=/etc/motd"
```

❗ Note

The name 'webserver' came from Cobbler, as did the variables for the config file. You can still pass in your own variables like normal in Ansible, but variables from the external inventory script will override any that have the same name.

So, with the template above (`motd.j2`), this results in the following data being written to `/etc/motd` for system 'foo':

```
Welcome, I am templated with a value of a=2, b=3, and c=4
```

And on system 'bar' (bar.example.com):

```
Welcome, I am templated with a value of a=2, b=3, and c=5
```

And technically, though there is no major good reason to do it, this also works:

```
ansible webserver -m ansible.builtin.shell -a "echo {{ a }}"
```

So, in other words, you can use those variables in arguments/actions as well.

Inventory script example: OpenStack

If you use an OpenStack-based cloud, instead of manually maintaining your own inventory file, you can use the `openstack_inventory.py` dynamic inventory to pull information about your compute instances directly from OpenStack.

You can download the latest version of the OpenStack inventory script [here](https://raw.githubusercontent.com/openstack/ansible-collections-openstack/master/scripts/inventory/openstack_inventory.py) (https://raw.githubusercontent.com/openstack/ansible-collections-openstack/master/scripts/inventory/openstack_inventory.py).

You can use the inventory script explicitly (by passing the `-i openstack_inventory.py` argument to Ansible) or implicitly (by placing the script at `/etc/ansible/hosts`).

Explicit use of OpenStack inventory script

Download the latest version of the OpenStack dynamic inventory script and make it executable.

```
wget https://raw.githubusercontent.com/openstack/ansible-collections-openstack/master/scripts/inventory/openstack_inventory.py
chmod +x openstack_inventory.py
```

❗ Note

Do not name it `openstack.py`. This name will conflict with imports from `openstacksdk`.

Source an OpenStack RC file:

```
source openstack.rc
```

❗ Note

An OpenStack RC file contains the environment variables required by the client tools to establish a connection with the cloud provider, such as the authentication URL, user name, password and region name. For more information on how to download, create or source an OpenStack RC file, please refer to [Set environment variables using the OpenStack RC file \(https://docs.openstack.org/user-guide/common/cli_set_environment_variables_using_openstack_rc.html\)](https://docs.openstack.org/user-guide/common/cli_set_environment_variables_using_openstack_rc.html).

You can confirm the file has been successfully sourced by running a simple command, such as *nova list* and ensuring it returns no errors.

Note

The OpenStack command line clients are required to run the *nova list* command. For more information on how to install them, please refer to [Install the OpenStack command-line clients \(https://docs.openstack.org/user-guide/common/cli_install_openstack_command_line_clients.html\)](https://docs.openstack.org/user-guide/common/cli_install_openstack_command_line_clients.html).

You can test the OpenStack dynamic inventory script manually to confirm it is working as expected:

```
./openstack_inventory.py --list
```

After a few moments you should see some JSON output with information about your compute instances.

Once you confirm the dynamic inventory script is working as expected, you can tell Ansible to use the *openstack_inventory.py* script as an inventory file, as illustrated below:

```
ansible -i openstack_inventory.py all -m ansible.builtin.ping
```

Implicit use of OpenStack inventory script

Download the latest version of the OpenStack dynamic inventory script, make it executable and copy it to */etc/ansible/hosts*:

```
wget https://raw.githubusercontent.com/openstack/ansible-collections-openstack/master/scripts/inventory/openstack_inventory.py
chmod +x openstack_inventory.py
sudo cp openstack_inventory.py /etc/ansible/hosts
```

Download the sample configuration file, modify it to suit your needs and copy it to `/etc/ansible/openstack.yml`:

```
wget https://raw.githubusercontent.com/openstack/ansible-collections-  
openstack/master/scripts/inventory/openstack.yml  
vi openstack.yml  
sudo cp openstack.yml /etc/ansible/
```

You can test the OpenStack dynamic inventory script manually to confirm it is working as expected:

```
/etc/ansible/hosts --list
```

After a few moments you should see some JSON output with information about your compute instances.

Refreshing the cache

Note that the OpenStack dynamic inventory script will cache results to avoid repeated API calls. To explicitly clear the cache, you can run the `openstack_inventory.py` (or `hosts`) script with the `--refresh` parameter:

```
./openstack_inventory.py --refresh --list
```

Other inventory scripts

In Ansible 2.10 and later, inventory scripts moved to their associated collections. Many are now in the [community.general scripts/inventory directory \(https://github.com/ansible-collections/community.general/tree/main/scripts/inventory\)](https://github.com/ansible-collections/community.general/tree/main/scripts/inventory). We recommend you use [Inventory plugins \(./plugins/inventory.html#inventory-plugins\)](https://github.com/ansible-collections/community.general/tree/main/plugins/inventory) instead.

Using inventory directories and multiple inventory sources

If the location given to `-i` in Ansible is a directory (or as so configured in `ansible.cfg`), Ansible can use multiple inventory sources at the same time. When doing so, it is possible to mix both dynamic and statically managed inventory sources in the same ansible run. Instant hybrid cloud!

In an inventory directory, executable files are treated as dynamic inventory sources and most other files as static sources. Files which end with any of the following are ignored:

```
~, .orig, .bak, .ini, .cfg, .retry, .pyc, .pyo
```

You can replace this list with your own selection by configuring an

`inventory_ignore_extensions` list in `ansible.cfg`, or setting the `ANSIBLE_INVENTORY_IGNORE` ([./reference_appendices/config.html#envvar-ANSIBLE_INVENTORY_IGNORE](#)) environment variable. The value in either case must be a comma-separated list of patterns, as shown above.

Any `group_vars` and `host_vars` subdirectories in an inventory directory are interpreted as expected, making inventory directories a powerful way to organize different sets of configurations. See [Using multiple inventory sources \(intro_inventory.html#using-multiple-inventory-sources\)](#) for more information.

Static groups of dynamic groups

When defining groups of groups in the static inventory file, the child groups must also be defined in the static inventory file, otherwise ansible returns an error. If you want to define a static group of dynamic child groups, define the dynamic groups as empty in the static inventory file. For example:

```
[tag_Name_staging_foo]

[tag_Name_staging_bar]

[staging:children]
tag_Name_staging_foo
tag_Name_staging_bar
```

! See also

[How to build your inventory \(intro_inventory.html#intro-inventory\)](#)

All about static inventory files

[Mailing List \(https://groups.google.com/group/ansible-project\)](https://groups.google.com/group/ansible-project)

Questions? Help? Ideas? Stop by the list on Google Groups

[Real-time chat \(./community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Patterns: targeting hosts and groups

When you execute Ansible through an ad hoc command or by running a playbook, you must choose which managed nodes or groups you want to execute against. Patterns let you run commands and playbooks against specific hosts and/or groups in your inventory. An Ansible pattern can refer to a single host, an IP address, an inventory group, a set of groups, or all hosts in your inventory. Patterns are highly flexible - you can exclude or require subsets of hosts, use wildcards or regular expressions, and more. Ansible executes on all inventory hosts included in the pattern.

- [Using patterns](#)
- [Common patterns](#)
- [Limitations of patterns](#)
- [Advanced pattern options](#)
 - [Using variables in patterns](#)
 - [Using group position in patterns](#)
 - [Using regexes in patterns](#)
- [Patterns and ad-hoc commands](#)
- [Patterns and ansible-playbook flags](#)

Using patterns

You use a pattern almost any time you execute an ad hoc command or a playbook. The pattern is the only element of an [ad hoc command](#) ([intro_adhoc.html#intro-adhoc](#)) that has no flag. It is usually the second element:

```
ansible <pattern> -m <module_name> -a "<module options>"
```

For example:

```
ansible webservers -m service -a "name=httpd state=restarted"
```

In a playbook the pattern is the content of the `hosts:` line for each play:

```
- name: <play_name>
  hosts: <pattern>
```

For example:

```
- name: restart webserver
  hosts: webserver
```

Since you often want to run a command or playbook against multiple hosts at once, patterns often refer to inventory groups. Both the ad hoc command and the playbook above will execute against all machines in the `webserver` group.

Common patterns

This table lists common patterns for targeting inventory hosts and groups.

Description	Pattern(s)	Targets
All hosts	all (or *)	
One host	host1	
Multiple hosts	host1: host2 (or host1, host2)	
One group	webserver	
Multiple groups	webserver:dbserver	all hosts in webserver plus all hosts in dbserver
Excluding groups	webserver:!atlanta	all hosts in webserver except those in atlanta
Intersection of groups	webserver:&staging	any hosts in webserver that are also in staging

Note

You can use either a comma (`,`) or a colon (`:`) to separate a list of hosts. The comma is preferred when dealing with ranges and IPv6 addresses.

Once you know the basic patterns, you can combine them. This example:

```
webserver:dbserver:&staging:!phoenix
```


targets all machines in the groups 'webservers' and 'dbservers' that are also in the group 'staging', except any machines in the group 'phoenix'.

You can use wildcard patterns with FQDNs or IP addresses, as long as the hosts are named in your inventory by FQDN or IP address:

```
192.0.\*
\*.example.com
\*.com
```

You can mix wildcard patterns and groups at the same time:

```
one*.com:dbservers
```

Limitations of patterns

Patterns depend on inventory. If a host or group is not listed in your inventory, you cannot use a pattern to target it. If your pattern includes an IP address or hostname that does not appear in your inventory, you will see an error like this:

```
[WARNING]: No inventory was parsed, only implicit localhost is available
[WARNING]: Could not match supplied host pattern, ignoring: *.not_in_inventory.com
```

Your pattern must match your inventory syntax. If you define a host as an [alias](#) ([intro_inventory.html#inventory-aliases](#)):

```
atlanta:
  host1:
    http_port: 80
    maxRequestsPerChild: 808
    host: 127.0.0.2
```

you must use the alias in your pattern. In the example above, you must use `host1` in your pattern. If you use the IP address, you will once again get the error:

```
[WARNING]: Could not match supplied host pattern, ignoring: 127.0.0.2
```

Advanced pattern options

The common patterns described above will meet most of your needs, but Ansible offers several other ways to define the hosts and groups you want to target.

Using variables in patterns

You can use variables to enable passing group specifiers via the `-e` argument to `ansible-playbook`:

```
webservers:!{{ excluded }}:&{{ required }}
```

Using group position in patterns

You can define a host or subset of hosts by its position in a group. For example, given the following group:

```
[webservers]
cobweb
webbing
weber
```

you can use subscripts to select individual hosts or ranges within the `webservers` group:

```
webservers[0]      # == cobweb
webservers[-1]     # == weber
webservers[0:2]    # == webservers[0],webservers[1]
                  # == cobweb,webbing
webservers[1:]     # == webbing,weber
webservers[:3]     # == cobweb,webbing,weber
```

Using regexes in patterns

You can specify a pattern as a regular expression by starting the pattern with `~`:

```
~(web|db).*\.example\.com
```

Patterns and ad-hoc commands

You can change the behavior of the patterns defined in ad-hoc commands using command-line options. You can also limit the hosts you target on a particular run with the `--limit` flag.

- Limit to one host

```
$ ansible -m [module] -a "[module options]" --limit "host1"
```

- Limit to multiple hosts

```
$ ansible -m [module] -a "[module options]" --limit "host1,host2"
```

- Negated limit. Note that single quotes MUST be used to prevent bash interpolation.

```
$ ansible -m [module] -a "[module options]" --limit 'all:!host1'
```

- Limit to host group

```
$ ansible -m [module] -a "[module options]" --limit 'group1'
```

Patterns and ansible-playbook flags

You can change the behavior of the patterns defined in playbooks using command-line options. For example, you can run a playbook that defines `hosts: all` on a single host by specifying `-i 127.0.0.2,` (note the trailing comma). This works even if the host you target is not defined in your inventory. You can also limit the hosts you target on a particular run with the `--limit` flag:

```
ansible-playbook site.yml --limit datacenter2
```

Finally, you can use `--limit` to read the list of hosts from a file by prefixing the file name with `@`:

```
ansible-playbook site.yml --limit @retry_hosts.txt
```

If `RETRY_FILES_ENABLED` ([../reference_appendices/config.html#retry-files-enabled](#)) is set to `True`, a `.retry` file will be created after the `ansible-playbook` run containing a list of failed hosts from all plays. This file is overwritten each time `ansible-playbook` finishes running.

Search this site

ansible-playbook site.yml --limit @site.retry

To apply your knowledge of patterns with Ansible commands and playbooks, read [Introduction to ad hoc commands \(intro_adhoc.html#intro-adhoc\)](#) and [Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#).

📌 See also

[Introduction to ad hoc commands \(intro_adhoc.html#intro-adhoc\)](#)

Examples of basic commands

[Working with playbooks \(playbooks.html#working-with-playbooks\)](#)

Learning the Ansible configuration management language

[Mailing List \(https://groups.google.com/group/ansible-project\)](https://groups.google.com/group/ansible-project)

Questions? Help? Ideas? Stop by the list on Google Groups

[Real-time chat \(./community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Connection methods and details

This section shows you how to expand and refine the connection methods Ansible uses for your inventory.

ControlPersist and paramiko

By default, Ansible uses native OpenSSH, because it supports ControlPersist (a performance feature), Kerberos, and options in `~/.ssh/config` such as Jump Host setup. If your control machine uses an older version of OpenSSH that does not support ControlPersist, Ansible will fallback to a Python implementation of OpenSSH called 'paramiko'.

Setting a remote user

By default, Ansible connects to all remote devices with the user name you are using on the control node. If that user name does not exist on a remote device, you can set a different user name for the connection. If you just need to do some tasks as a different user, look at [Understanding privilege escalation: become \(become.html#become\)](#). You can set the connection user in a playbook:

```
---
- name: update webserver
  hosts: webserver
  remote_user: admin

  tasks:
  - name: thing to do first in this playbook
    . . .
```

as a host variable in inventory:

other1.example.com	ansible_connection=ssh	ansible_user=myuser
other2.example.com	ansible_connection=ssh	ansible_user=myotheruser

or as a group variable in inventory:

```
cloud:
  hosts:
    cloud1: my_backup.cloud.com
    cloud2: my_backup2.cloud.com
  vars:
    ansible_user: admin
```

Setting up SSH keys

By default, Ansible assumes you are using SSH keys to connect to remote machines. SSH keys are encouraged, but you can use password authentication if needed with the `--ask-pass` option. If you need to provide a password for [privilege escalation](#) ([become.html#become](#)) (sudo, pbrun, and so on), use `--ask-become-pass`.

Note

Ansible does not expose a channel to allow communication between the user and the ssh process to accept a password manually to decrypt an ssh key when using the ssh connection plugin (which is the default). The use of `ssh-agent` is highly recommended.

To set up SSH agent to avoid retyping passwords, you can do:

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_rsa
```

Depending on your setup, you may wish to use Ansible's `--private-key` command line option to specify a pem file instead. You can also add the private key file:

```
$ ssh-agent bash
$ ssh-add ~/.ssh/keypair.pem
```

Another way to add private key files without using ssh-agent is using `ansible_ssh_private_key_file` in an inventory file as explained here: [How to build your inventory \(intro_inventory.html#intro-inventory\)](#).

Running against localhost

You can run commands against the control node by using “localhost” or “127.0.0.1” for the server name:

Search this site

```
$ ansible localhost -m ping -e 'ansible_python_interpreter="/usr/bin/env python"'
```

You can specify localhost explicitly by adding this to your inventory file:

```
localhost ansible_connection=local ansible_python_interpreter="/usr/bin/env python"
```

Managing host key checking

Ansible enables host key checking by default. Checking host keys guards against server spoofing and man-in-the-middle attacks, but it does require some maintenance.

If a host is reinstalled and has a different key in 'known_hosts', this will result in an error message until corrected. If a new host is not in 'known_hosts' your control node may prompt for confirmation of the key, which results in an interactive experience if using Ansible, from say, cron. You might not want this.

If you understand the implications and wish to disable this behavior, you can do so by editing

```
/etc/ansible/ansible.cfg or ~/.ansible.cfg :
```

```
[defaults]
host_key_checking = False
```

Alternatively this can be set by the `ANSIBLE_HOST_KEY_CHECKING` ([./reference_appendices/config.html#envvar-ANSIBLE_HOST_KEY_CHECKING](#)) environment variable:

```
$ export ANSIBLE_HOST_KEY_CHECKING=False
```

Also note that host key checking in paramiko mode is reasonably slow, therefore switching to 'ssh' is also recommended when using this feature.

Other connection methods

Ansible can use a variety of connection methods beyond SSH. You can select any connection plugin, including managing things locally and managing chroot, lxc, and jail containers. A mode called 'ansible-pull' can also invert the system and have systems 'phone home' via scheduled git checkouts to pull configuration directives from a central repository.

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Using Variables

Ansible uses variables to manage differences between systems. With Ansible, you can execute tasks and playbooks on multiple different systems with a single command. To represent the variations among those different systems, you can create variables with standard YAML syntax, including lists and dictionaries. You can define these variables in your playbooks, in your [inventory](#) ([intro inventory.html#intro-inventory](#)), in re-usable [files](#) ([playbooks reuse.html#playbooks-reuse](#)) or [roles](#) ([playbooks reuse roles.html#playbooks-reuse-roles](#)), or at the command line. You can also create variables during a playbook run by registering the return value or values of a task as a new variable.

After you create variables, either by defining them in a file, passing them at the command line, or registering the return value or values of a task as a new variable, you can use those variables in module arguments, in [conditional “when” statements](#) ([playbooks conditionals.html#playbooks-conditionals](#)), in [templates](#) ([playbooks templating.html#playbooks-templating](#)), and in [loops](#) ([playbooks loops.html#playbooks-loops](#)). The [ansible-examples github repository](#) (<https://github.com/ansible/ansible-examples>) contains many examples of using variables in Ansible.

Once you understand the concepts and examples on this page, read about [Ansible facts](#) ([playbooks vars facts.html#vars-and-facts](#)), which are variables you retrieve from remote systems.

- [Creating valid variable names](#)
- [Simple variables](#)
 - [Defining simple variables](#)
 - [Referencing simple variables](#)
- [When to quote variables \(a YAML gotcha\)](#)
- [List variables](#)
 - [Defining variables as lists](#)
 - [Referencing list variables](#)
- [Dictionary variables](#)
 - [Defining variables as key:value dictionaries](#)
 - [Referencing key:value dictionary variables](#)

- [Registering variables](#)
- [Referencing nested variables](#)
- [Transforming variables with Jinja2 filters](#)
- [Where to set variables](#)
 - [Defining variables in inventory](#)
 - [Defining variables in a play](#)
 - [Defining variables in included files and roles](#)
 - [Defining variables at runtime](#)
 - [key=value format](#)
 - [JSON string format](#)
 - [vars from a JSON or YAML file](#)
- [Variable precedence: Where should I put a variable?](#)
 - [Understanding variable precedence](#)
 - [Scoping variables](#)
 - [Tips on where to set variables](#)
- [Using advanced variable syntax](#)

Creating valid variable names

Not all strings are valid Ansible variable names. A variable name can only include letters, numbers, and underscores. [Python keywords](#) (https://docs.python.org/3/reference/lexical_analysis.html#keywords) or [playbook keywords](#) ([../reference/appendices/playbooks_keywords.html#playbook-keywords](https://docs.python.org/3/reference/lexical_analysis.html#keywords)), are not valid variable names. A variable name cannot begin with a number.

Variable names can begin with an underscore. In many programming languages, variables that begin with an underscore are private. This is not true in Ansible. Variables that begin with an underscore are treated exactly the same as any other variable. Do not rely on this convention for privacy or security.

This table gives examples of valid and invalid variable names:

Valid variable names	Not valid
<code>foo</code>	<code>*foo</code> , Python keywords (https://docs.python.org/3/reference/lexical_analysis.html#keywords)
<code>foo_env</code>	playbook keywords (../reference/appendices/playbooks_keywords.html#playbook-keywords)
<code>foo_port</code>	<code>foo-port</code> , <code>foo port</code> , <code>foo.port</code>
<code>foo5</code> , <code>_foo</code>	<code>5foo</code> , <code>12</code>

Simple variables

Simple variables combine a variable name with a single value. You can use this syntax (and the syntax for lists and dictionaries shown below) in a variety of places. For details about setting variables in inventory, in playbooks, in reusable files, in roles, or at the command line, see [Where to set variables](#).

Defining simple variables

You can define a simple variable using standard YAML syntax. For example:

```
remote_install_path: /opt/my_app_config
```

Referencing simple variables

After you define a variable, use Jinja2 syntax to reference it. Jinja2 variables use double curly braces. For example, the expression `My amp goes to {{ max_amp_value }}` demonstrates the most basic form of variable substitution. You can use Jinja2 syntax in playbooks. For example:

```
ansible.builtin.template:
  src: foo.cfg.j2
  dest: '{{ remote_install_path }}/foo.cfg'
```

In this example, the variable defines the location of a file, which can vary from one system to another.

Note

Ansible allows Jinja2 loops and conditionals in [templates](#) ([playbooks_templating.html#playbooks-templating](#)) but not in playbooks. You cannot create a loop of tasks. Ansible playbooks are pure machine-parseable YAML.

When to quote variables (a YAML gotcha)

If you start a value with `{{ foo }}`, you must quote the whole expression to create valid YAML syntax. If you do not quote the whole expression, the YAML parser cannot interpret the syntax - it might be a variable or it might be the start of a YAML dictionary. For guidance on writing YAML, see the [YAML Syntax](#) ([../reference_appendices/YAMLSyntax.html#yaml-syntax](#)) documentation.

If you use a variable without quotes like this:

```
- hosts: app_servers
  vars:
    app_path: {{ base_path }}/22
```

You will see: `ERROR! Syntax Error while loading YAML.` If you add quotes, Ansible works correctly:

```
- hosts: app_servers
  vars:
    app_path: "{{ base_path }}/22"
```

List variables

A list variable combines a variable name with multiple values. The multiple values can be stored as an itemized list or in square brackets `[]`, separated with commas.

Defining variables as lists

You can define variables with multiple values using YAML lists. For example:

```
region:
  - northeast
  - southeast
  - midwest
```

Referencing list variables

When you use variables defined as a list (also called an array), you can use individual, specific fields from that list. The first item in a list is item 0, the second item is item 1. For example:

```
region: "{{ region[0] }}"
```

The value of this expression would be “northeast”.

Dictionary variables

A dictionary stores the data in key-value pairs. Usually, dictionaries are used to store related data, such as the information contained in an ID or a user profile.

Defining variables as key:value dictionaries

You can define more complex variables using YAML dictionaries. A YAML dictionary maps keys to values. For example:

```
foo:
  field1: one
  field2: two
```

Referencing key:value dictionary variables

When you use variables defined as a key:value dictionary (also called a hash), you can use individual, specific fields from that dictionary using either bracket notation or dot notation:

```
foo['field1']
foo.field1
```

Both of these examples reference the same value (“one”). Bracket notation always works. Dot notation can cause problems because some keys collide with attributes and methods of python dictionaries. Use bracket notation if you use keys which start and end with two underscores (which are reserved for special meanings in python) or are any of the known public attributes:

```
add, append, as_integer_ratio, bit_length, capitalize, center, clear, conjugate,
copy, count, decode, denominator, difference, difference_update, discard, encode,
endswith, expandtabs, extend, find, format, fromhex, fromkeys, get, has_key, hex,
imag, index, insert, intersection, intersection_update, isalnum, isalpha, isdecimal,
isdigit, isdisjoint, is_integer, islower, isnumeric, isspace, issubset, issuperset,
istitle, isupper, items, iteritems, iterkeys, itervalues, join, keys, ljust,
lower, lstrip, numerator, partition, pop, popitem, real, remove, replace, reverse,
rfind, rindex, rjust, rpartition, rsplit, rstrip, setdefault, sort, split,
splitlines, startswith, strip, swapcase, symmetric_difference,
symmetric_difference_update, title, translate, union, update, upper, values,
viewitems, viewkeys, viewvalues, zfill.
```

Registering variables

You can create variables from the output of an Ansible task with the task keyword `register`. You can use registered variables in any later tasks in your play. For example:

```
- hosts: web_servers

tasks:

  - name: Run a shell command and register its output as a variable
    ansible.builtin.shell: /usr/bin/foo
    register: foo_result
    ignore_errors: true

  - name: Run a shell command using output of the previous task
    ansible.builtin.shell: /usr/bin/bar
    when: foo_result.rc == 5
```

For more examples of using registered variables in conditions on later tasks, see [Conditionals \(playbooks conditionals.html#playbooks-conditionals\)](#). Registered variables may be simple variables, list variables, dictionary variables, or complex nested data structures. The documentation for each module includes a `RETURN` section describing the return values for that module. To see the values for a particular task, run your playbook with `-v`.

Registered variables are stored in memory. You cannot cache registered variables for use in future plays. Registered variables are only valid on the host for the rest of the current playbook run.

Registered variables are host-level variables. When you register a variable in a task with a loop, the registered variable contains a value for each item in the loop. The data structure placed in the variable during the loop will contain a `results` attribute, that is a list of all responses from the module. For a more in-depth example of how this works, see the [Loops \(playbooks loops.html#playbooks-loops\)](#) section on using register with a loop.

❗ Note

If a task fails or is skipped, Ansible still registers a variable with a failure or skipped status, unless the task is skipped based on tags. See [Tags \(playbooks tags.html#tags\)](#) for information on adding and using tags.

Referencing nested variables

Many registered variables (and [facts \(playbooks vars facts.html#vars-and-facts\)](#)) are nested YAML or JSON data structures. You cannot access values from these nested data structures with the simple `{{ foo }}` syntax. You must use either bracket notation or dot notation. For example, to reference an IP address from your facts using the bracket notation:

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

To reference an IP address from your facts using the dot notation:

```
{{ ansible_facts.eth0.ipv4.address }}
```

Transforming variables with Jinja2 filters

Jinja2 filters let you transform the value of a variable within a template expression. For example, the `capitalize` filter capitalizes any value passed to it; the `to_yaml` and `to_json` filters change the format of your variable values. Jinja2 includes many [built-in filters](https://jinja.palletsprojects.com/templates/#builtin-filters) (<https://jinja.palletsprojects.com/templates/#builtin-filters>) and Ansible supplies many more filters. To find more examples of filters, see [Using filters to manipulate data](#) ([playbooks_filters.html#playbooks-filters](#)).

Where to set variables

You can define variables in a variety of places, such as in inventory, in playbooks, in reusable files, in roles, and at the command line. Ansible loads every possible variable it finds, then chooses the variable to apply based on [variable precedence rules](#).

Defining variables in inventory.

You can define different variables for each individual host, or set shared variables for a group of hosts in your inventory. For example, if all machines in the `[Boston]` group use 'boston.ntp.example.com' as an NTP server, you can set a group variable. The [How to build your inventory](#) ([intro_inventory.html#intro-inventory](#)), page has details on setting [host variables](#) ([intro_inventory.html#host-variables](#)) and [group variables](#) ([intro_inventory.html#group-variables](#)) in inventory.

Defining variables in a play

You can define variables directly in a playbook play:

```
- hosts: webservers
  vars:
    http_port: 80
```

When you define variables in a play, they are only visible to tasks executed in that play.

Defining variables in included files and roles

You can define variables in reusable variables files and/or in reusable roles. When you define variables in reusable variable files, the sensitive variables are separated from playbooks. This separation enables you to store your playbooks in a source control software and even share the playbooks, without the risk of exposing passwords or other sensitive and personal data. For information about creating reusable files and roles, see [Re-using Ansible artifacts \(playbooks_reuse.html#playbooks-reuse\)](#).

This example shows how you can include variables defined in an external file:

```
---

- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /vars/external_vars.yml

  tasks:

- name: This is just a placeholder
  ansible.builtin.command: /bin/echo foo
```

The contents of each variables file is a simple YAML dictionary. For example:

```
---
# in the above example, this would be vars/external_vars.yml
somevar: somevalue
password: magic
```

Note

You can keep per-host and per-group variables in similar files. To learn about organizing your variables, see [Organizing host and group variables \(intro_inventory.html#splitting-out-vars\)](#).

Defining variables at runtime

You can define variables when you run your playbook by passing variables at the command line using the `--extra-vars` (or `-e`) argument. You can also request user input with a `vars_prompt` (see [Interactive input: prompts \(playbooks_prompts.html#playbooks-prompts\)](#)). When you pass variables at the command line, use a single quoted string, that contains one or more variables, in one of the formats below.

key=value format

Values passed in using the `key=value` syntax are interpreted as strings. Use the JSON format if you need to pass non-string values such as Booleans, integers, floats, lists, and so on.

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

JSON string format

```
ansible-playbook release.yml --extra-vars  
'{"version":"1.23.45","other_variable":"foo"}'  
ansible-playbook arcade.yml --extra-vars '{"pacman":"mrs","ghosts":  
["inky","pinky","clyde","sue"]}'
```

When passing variables with `--extra-vars`, you must escape quotes and other special characters appropriately for both your markup (for example, JSON), and for your shell:

```
ansible-playbook arcade.yml --extra-vars '{"name\":"Conan O\'Brien\"}'  
ansible-playbook arcade.yml --extra-vars '{"name":"Conan O\'\\\'\'Brien"}'  
ansible-playbook script.yml --extra-vars '{"dialog\":"He said \\'\'I just can\'t get  
enough of those single and double-quotes\'\'!\'\'\'\'"}'
```

If you have a lot of special characters, use a JSON or YAML file containing the variable definitions.

vars from a JSON or YAML file

```
ansible-playbook release.yml --extra-vars "@some_file.json"
```

Variable precedence: Where should I put a variable?

You can set multiple variables with the same name in many different places. When you do this, Ansible loads every possible variable it finds, then chooses the variable to apply based on variable precedence. In other words, the different variables will override each other in a certain order.

Teams and projects that agree on guidelines for defining variables (where to define certain types of variables) usually avoid variable precedence concerns. We suggest that you define each variable in one place: figure out where to define a variable, and keep it simple. For examples, see [Tips on where to set variables](#).

Some behavioral parameters that you can set in variables you can also set in Ansible configuration, as command-line options, and using playbook keywords. For example, you can define the user Ansible uses to connect to remote devices as a variable with `ansible_user`, in a configuration file with `DEFAULT_REMOTE_USER`, as a command-line option with `-u`, and with the playbook keyword `remote_user`. If you define the same parameter in a variable and by another method, the variable overrides the other setting. This approach allows host-specific settings to override more general settings. For examples and more details on the precedence of these various settings, see [Controlling how Ansible behaves: precedence rules](https://docs.ansible.com/ansible/latest/reference_appendices/general_precedence.html#general-precedence-rules) ([./reference_appendices/general_precedence.html#general-precedence-rules](https://docs.ansible.com/ansible/latest/reference_appendices/general_precedence.html#general-precedence-rules)).

Understanding variable precedence

Ansible does apply variable precedence, and you might have a use for it. Here is the order of precedence from least to greatest (the last listed variables override all other variables):

1. command line values (for example, `-u my_user`, these are not variables)
2. role defaults (defined in `role/defaults/main.yml`) ¹
3. inventory file or script group vars ²
4. inventory group_vars/all ³
5. playbook group_vars/all ³
6. inventory group_vars/* ³
7. playbook group_vars/* ³
8. inventory file or script host vars ²
9. inventory host_vars/* ³
10. playbook host_vars/* ³
11. host facts / cached set_facts ⁴
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in `role/vars/main.yml`)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts / registered vars
20. role (and include_role) params
21. include params
22. extra vars (for example, `-e "user=my_user"`)(always win precedence)

In general, Ansible gives precedence to variables that were defined more recently, more actively, and with more explicit scope. Variables in the defaults folder inside a role are easily overridden. Anything in the vars directory of the role overrides previous versions of that variable in the namespace. Host and/or inventory variables override role defaults, but explicit includes such as the vars directory or an `include_vars` task override inventory variables.

Ansible merges different variables set in inventory so that more specific settings override more generic settings. For example, `ansible_ssh_user` specified as a `group_var` is overridden by `ansible_user` specified as a `host_var`. For details about the precedence of variables set in inventory, see [How variables are merged \(intro_inventory.html#how-we-merge\)](https://docs.ansible.com/ansible/latest/intro_inventory.html#how-we-merge).

Footnotes

- [1] : Tasks in each role see their own role's defaults. Tasks defined outside of a role see the last role's defaults.
- [2] (1,2): Variables defined in inventory file or provided by dynamic inventory.
- [3] (1,2,3,4,5,6): Includes vars added by 'vars plugins' as well as `host_vars` and `group_vars` which are added by the default vars plugin shipped with Ansible.
- [4] : When created with `set_facts`'s `cacheable` option, variables have the high precedence in the play, but are the same as a host facts precedence when they come from the cache.

Note

Within any section, redefining a var overrides the previous instance. If multiple groups have the same variable, the last one loaded wins. If you define a variable twice in a play's `vars:` section, the second one wins.

Note

The previous describes the default config `hash_behaviour=replace`, switch to `merge` to only partially overwrite.

Scoping variables

You can decide where to set a variable based on the scope you want that value to have. Ansible has three main scopes:

- Global: this is set by config, environment variables and the command line
- Play: each play and contained structures, vars entries (`vars`; `vars_files`; `vars_prompt`), role defaults and vars.
- Host: variables directly associated to a host, like inventory, `include_vars`, facts or registered task outputs

Inside a template, you automatically have access to all variables that are in scope for a host, plus any registered variables, facts, and magic variables.

Tips on where to set variables

You should choose where to define a variable based on the kind of control you might want over values.

Set variables in inventory that deal with geography or behavior. Since groups are frequently the entity that maps roles onto hosts, you can often set variables on the group instead of defining them on a role. Remember: child groups override parent groups, and host variables override group variables. See [Defining variables in inventory](#) for details on setting host and group variables.

Set common defaults in a `group_vars/all` file. See [Organizing host and group variables \(intro_inventory.html#splitting-out-vars\)](#) for details on how to organize host and group variables in your inventory. Group variables are generally placed alongside your inventory file, but they can also be returned by dynamic inventory (see [Working with dynamic inventory \(intro_dynamic_inventory.html#intro-dynamic-inventory\)](#)) or defined in AWX or on [Red Hat Ansible Automation Platform \(../reference_appendices/tower.html#ansible-platform\)](#) from the UI or API:

```
---
# file: /etc/ansible/group_vars/all
# this is the site wide default
ntp_server: default-time.example.com
```

Set location-specific variables in `group_vars/my_location` files. All groups are children of the `all` group, so variables set here override those set in `group_vars/all`:

```
---
# file: /etc/ansible/group_vars/boston
ntp_server: boston-time.example.com
```

If one host used a different NTP server, you could set that in a `host_vars` file, which would override the group variable:

```
---
# file: /etc/ansible/host_vars/xyz.boston.example.com
ntp_server: override.example.com
```

Set defaults in roles to avoid undefined-variable errors. If you share your roles, other users can rely on the reasonable defaults you added in the `roles/x/defaults/main.yml` file, or they can easily override those values in inventory or at the command line. See [Roles \(playbooks_reuse_roles.html#playbooks-reuse-roles\)](#) for more info. For example: Search this site

```
---
# file: roles/x/defaults/main.yml
# if no other value is supplied in inventory or as a parameter, this value will be used
http_port: 80
```

Set variables in roles to ensure a value is used in that role, and is not overridden by inventory variables. If you are not sharing your role with others, you can define app-specific behaviors like ports this way, in `roles/x/vars/main.yml`. If you are sharing roles with others, putting variables here makes them harder to override, although they still can by passing a parameter to the role or setting a variable with `-e`:

```
---
# file: roles/x/vars/main.yml
# this will absolutely be used in this role
http_port: 80
```

Pass variables as parameters when you call roles for maximum clarity, flexibility, and visibility. This approach overrides any defaults that exist for a role. For example:

```
roles:
  - role: apache
    vars:
      http_port: 8080
```

When you read this playbook it is clear that you have chosen to set a variable or override a default. You can also pass multiple values, which allows you to run the same role multiple times. See [Running a role multiple times in one playbook \(playbooks_reuse_roles.html#run-role-twice\)](https://docs.ansible.com/ansible/playbooks_reuse_roles.html#run-role-twice) for more details. For example:

```
roles:
  - role: app_user
    vars:
      myname: Ian
  - role: app_user
    vars:
      myname: Terry
  - role: app_user
    vars:
      myname: Graham
  - role: app_user
    vars:
      myname: John
```

Variables set in one role are available to later roles. You can set variables in a

`roles/common_settings/vars/main.yml` file and use them in other roles and elsewhere in your playbook:

```
roles:
  - role: common_settings
  - role: something
    vars:
      foo: 12
  - role: something_else
```

Note

There are some protections in place to avoid the need to namespace variables. In this example, variables defined in 'common_settings' are available to 'something' and 'something_else' tasks, but tasks in 'something' have foo set at 12, even if 'common_settings' sets foo to 20.

Instead of worrying about variable precedence, we encourage you to think about how easily or how often you want to override a variable when deciding where to set it. If you are not sure what other variables are defined, and you need a particular value, use `--extra-vars` (`-e`) to override all other variables.

Using advanced variable syntax

For information about advanced YAML syntax used to declare variables and have more control over the data placed in YAML files used by Ansible, see [Advanced Syntax](#) ([playbooks_advanced_syntax.html#playbooks-advanced-syntax](#)).

See also

[Intro to playbooks](#) ([playbooks_intro.html#about-playbooks](#))

An introduction to playbooks

[Conditionals](#) ([playbooks_conditionals.html#playbooks-conditionals](#))

Conditional statements in playbooks

[Using filters to manipulate data](#) ([playbooks_filters.html#playbooks-filters](#))

Jinja2 filters and their uses

[Loops](#) ([playbooks_loops.html#playbooks-loops](#))

Looping in playbooks

[Roles](#) ([playbooks_reuse_roles.html#playbooks-reuse-roles](#))

Playbook organization by roles

[Tips and tricks \(playbooks_best_practices.html#playbooks-best-practices\)](#)

Tips and tricks for playbooks

[Special Variables \(../reference_appendices/special_variables.html#special-variables\)](#)

List of special variables

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Special Variables

Magic variables

These variables cannot be set directly by the user; Ansible will always override them to reflect internal state.

ansible_check_mode

Boolean that indicates if we are in check mode or not

ansible_config_file

The full path of used Ansible configuration file

ansible_dependent_role_names

The names of the roles currently imported into the current play as dependencies of other plays

ansible_diff_mode

Boolean that indicates if we are in diff mode or not

ansible_forks

Integer reflecting the number of maximum forks available to this run

ansible_inventory_sources

List of sources used as inventory

ansible_limit

Contents of the `--limit` CLI option for the current execution of Ansible

ansible_loop

A dictionary/map containing extended loop information when enabled via

`loop_control.extended`

ansible_loop_var

The name of the value provided to `loop_control.loop_var`. Added in `2.8`

ansible_index_var

The name of the value provided to `loop_control.index_var`. Added in `2.9`

ansible_parent_role_names

When the current role is being executed by means of an [include_role](#) ([../collections/ansible/builtin/include_role_module.html#include-role-module](#)) or [import_role](#) ([../collections/ansible/builtin/import_role_module.html#import-role-module](#)) action, this variable contains a list of all parent roles, with the most recent role (in other words, the role that included/imported this role) being the first item in the list. When multiple inclusions occur, this list lists the *last* role (in other words, the role that included this role) as the *first* item in the list. It is also possible that a specific role exists more than once in this list.

For example: When role **A** includes role **B**, inside role **B**, `ansible_parent_role_names` will equal to `['A']`. If role **B** then includes role **C**, the list becomes `['B', 'A']`.

ansible_parent_role_paths

When the current role is being executed by means of an [include_role](#) ([../collections/ansible/builtin/include_role_module.html#include-role-module](#)) or [import_role](#) ([../collections/ansible/builtin/import_role_module.html#import-role-module](#)) action, this variable contains a list of all parent roles, with the most recent role (in other words, the role that included/imported this role) being the first item in the list. Please refer to `ansible_parent_role_names` for the order of items in this list.

ansible_play_batch

List of active hosts in the current play run limited by the serial, aka 'batch'.
Failed/Unreachable hosts are not considered 'active'.

ansible_play_hosts

List of hosts in the current play run, not limited by the serial. Failed/Unreachable hosts are excluded from this list.

ansible_play_hosts_all

List of all the hosts that were targeted by the play

ansible_play_role_names

The names of the roles currently imported into the current play. This list does **not** contain the role names that are implicitly included via dependencies.

ansible_playbook_python

The path to the python interpreter being used by Ansible on the controller

ansible_role_names

The names of the roles currently imported into the current play, or roles referenced as dependencies of the roles imported into the current play.

ansible_role_name

The fully qualified collection role name, in the format of `namespace.collection.role_name`

ansible_collection_name

The name of the collection the task that is executing is a part of. In the format of

`namespace.collection`

ansible_run_tags

Contents of the `--tags` CLI option, which specifies which tags will be included for the current run. Note that if `--tags` is not passed, this variable will default to `["all"]`.

ansible_search_path

Current search path for action plugins and lookups, in other words, where we search for relative paths when you do `template: src=myfile`

ansible_skip_tags

Contents of the `--skip-tags` CLI option, which specifies which tags will be skipped for the current run.

ansible_verbosity

Current verbosity setting for Ansible

ansible_version

Dictionary/map that contains information about the current running version of ansible, it has the following keys: full, major, minor, revision and string.

group_names

List of groups the current host is part of

groups

A dictionary/map with all the groups in inventory and each group has the list of hosts that belong to it

hostvars

A dictionary/map with all the hosts in inventory and variables assigned to them

inventory_hostname

The inventory name for the 'current' host being iterated over in the play

inventory_hostname_short

The short version of *inventory_hostname*

inventory_dir

The directory of the inventory source in which the *inventory_hostname* was first defined

inventory_file

The file name of the inventory source in which the *inventory_hostname* was first defined

omit

Special variable that allows you to ‘omit’ an option in a task, for example `- user: name=bob`

```
home={{ bobs_home|default(omit) }}
```

play_hosts

Deprecated, the same as `ansible_play_batch`

ansible_play_name

The name of the currently executed play. Added in `2.8`. (*name* attribute of the play, not file name of the playbook.)

playbook_dir

The path to the directory of the playbook that was passed to the `ansible-playbook` command line.

role_name

The name of the role currently being executed.

role_names

Deprecated, the same as `ansible_play_role_names`

role_path

The path to the dir of the currently running role

Facts

These are variables that contain information pertinent to the current host (*inventory_hostname*). They are only available if gathered first. See [Discovering variables: facts and magic variables \(./user_guide/playbooks_vars_facts.html#vars-and-facts\)](#) for more information.

ansible_facts

Contains any facts gathered or cached for the *inventory_hostname*. Facts are normally gathered by the `setup(./collections/ansible/builtin/setup_module.html#setup-module)` module automatically in a play, but any module can return facts.

ansible_local

Contains any ‘local facts’ gathered or cached for the *inventory_hostname*. The keys available depend on the custom facts created. See the `setup(./collections/ansible/builtin/setup_module.html#setup-module)` module and [facts.d or local facts \(./user_guide/playbooks_vars_facts.html#local-facts\)](#) for more details.

Connection variables

Connection variables are normally used to set the specifics on how to execute actions on a target. Most of them correspond to connection plugins, but not all are specific to them; other plugins like shell, terminal and become are normally involved. Only the common ones are described as each connection/become/shell/etc plugin can define its own overrides and specific variables. See [Controlling how Ansible behaves: precedence rules \(general_precedence.html#general-precedence-rules\)](#) for how connection variables interact with [configuration settings \(config.html#ansible-configuration-settings\)](#), [command-line options \(./user_guide/command_line_tools.html#command-line-tools\)](#), and [playbook keywords \(playbooks_keywords.html#playbook-keywords\)](#).

ansible_become_user

The user Ansible 'becomes' after using privilege escalation. This must be available to the 'login user'.

ansible_connection

The connection plugin actually used for the task on the target host.

ansible_host

The ip/name of the target host to use instead of *inventory_hostname*.

ansible_python_interpreter

The path to the Python executable Ansible should use on the target host.

ansible_user

The user Ansible 'logs in' as.

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Discovering variables: facts and magic variables

With Ansible you can retrieve or discover certain variables containing information about your remote systems or about Ansible itself. Variables related to remote systems are called facts. With facts, you can use the behavior or state of one system as configuration on other systems. For example, you can use the IP address of one system as a configuration value on another system. Variables related to Ansible are called magic variables.

- [Ansible facts](#)
 - [Package requirements for fact gathering](#)
 - [Caching facts](#)
 - [Disabling facts](#)
 - [Adding custom facts](#)
 - [facts.d or local facts](#)
- [Information about Ansible: magic variables](#)
 - [Ansible version](#)

Ansible facts

Ansible facts are data related to your remote systems, including operating systems, IP addresses, attached filesystems, and more. You can access this data in the `ansible_facts` variable. By default, you can also access some Ansible facts as top-level variables with the `ansible_` prefix. You can disable this behavior using the [INJECT_FACTS_AS_VARS](#) ([../reference_appendices/config.html#inject-facts-as-vars](#)) setting. To see all available facts, add this task to a play:

```
- name: Print all available facts
  ansible.builtin.debug:
    var: ansible_facts
```

To see the ‘raw’ information as gathered, run this command at the command line:

```
ansible <hostname> -m ansible.builtin.setup
```

Facts include a large amount of variable data, which may look like this:

```

{
  "ansible_all_ipv4_addresses": [
    "REDACTED IP ADDRESS"
  ],
  "ansible_all_ipv6_addresses": [
    "REDACTED IPV6 ADDRESS"
  ],
  "ansible_apparmor": {
    "status": "disabled"
  },
  "ansible_architecture": "x86_64",
  "ansible_bios_date": "11/28/2013",
  "ansible_bios_version": "4.1.5",
  "ansible_cmdline": {
    "BOOT_IMAGE": "/boot/vmlinuz-3.10.0-862.14.4.el7.x86_64",
    "console": "ttyS0,115200",
    "no_timer_check": true,
    "nofb": true,
    "nomodeset": true,
    "ro": true,
    "root": "LABEL=cloudimg-rootfs",
    "vga": "normal"
  },
  "ansible_date_time": {
    "date": "2018-10-25",
    "day": "25",
    "epoch": "1540469324",
    "hour": "12",
    "iso8601": "2018-10-25T12:08:44Z",
    "iso8601_basic": "20181025T120844109754",
    "iso8601_basic_short": "20181025T120844",
    "iso8601_micro": "2018-10-25T12:08:44.109968Z",
    "minute": "08",
    "month": "10",
    "second": "44",
    "time": "12:08:44",
    "tz": "UTC",
    "tz_offset": "+0000",
    "weekday": "Thursday",
    "weekday_number": "4",
    "weeknumber": "43",
    "year": "2018"
  },
  "ansible_default_ipv4": {
    "address": "REDACTED",
    "alias": "eth0",
    "broadcast": "REDACTED",
    "gateway": "REDACTED",
    "interface": "eth0",
    "macaddress": "REDACTED",
    "mtu": 1500,
    "netmask": "255.255.255.0",
    "network": "REDACTED",
    "type": "ether"
  },
  "ansible_default_ipv6": {},
  "ansible_device_links": {
    "ids": {},
    "labels": {
      "xvda1": [
        "cloudimg-rootfs"
      ],
      "xvdd": [

```

```

        "config-2"
    ],
    },
    "masters": {},
    "uuids": {
        "xvda1": [
            "cac81d61-d0f8-4b47-84aa-b48798239164"
        ],
        "xvdd": [
            "2018-10-25-12-05-57-00"
        ]
    }
},
"ansible_devices": {
    "xvda": {
        "holders": [],
        "host": "",
        "links": {
            "ids": [],
            "labels": [],
            "masters": [],
            "uuids": []
        },
        "model": null,
        "partitions": {
            "xvda1": {
                "holders": [],
                "links": {
                    "ids": [],
                    "labels": [
                        "cloudimg-rootfs"
                    ],
                    "masters": [],
                    "uuids": [
                        "cac81d61-d0f8-4b47-84aa-b48798239164"
                    ]
                },
                "sectors": "83883999",
                "sectorsize": 512,
                "size": "40.00 GB",
                "start": "2048",
                "uuid": "cac81d61-d0f8-4b47-84aa-b48798239164"
            }
        },
        "removable": "0",
        "rotational": "0",
        "sas_address": null,
        "sas_device_handle": null,
        "scheduler_mode": "deadline",
        "sectors": "83886080",
        "sectorsize": "512",
        "size": "40.00 GB",
        "support_discard": "0",
        "vendor": null,
        "virtual": 1
    },
    "xvdd": {
        "holders": [],
        "host": "",
        "links": {
            "ids": [],
            "labels": [
                "config-2"
            ],
        },
    },

```



```

        "masters": [],
        "uuids": [
            "2018-10-25-12-05-57-00"
        ]
    },
    "model": null,
    "partitions": {},
    "removable": "0",
    "rotational": "0",
    "sas_address": null,
    "sas_device_handle": null,
    "scheduler_mode": "deadline",
    "sectors": "131072",
    "sectorsize": "512",
    "size": "64.00 MB",
    "support_discard": "0",
    "vendor": null,
    "virtual": 1
},
"xvde": {
    "holders": [],
    "host": "",
    "links": {
        "ids": [],
        "labels": [],
        "masters": [],
        "uuids": []
    },
    "model": null,
    "partitions": {
        "xvde1": {
            "holders": [],
            "links": {
                "ids": [],
                "labels": [],
                "masters": [],
                "uuids": []
            },
            "sectors": "167770112",
            "sectorsize": 512,
            "size": "80.00 GB",
            "start": "2048",
            "uuid": null
        }
    },
    "removable": "0",
    "rotational": "0",
    "sas_address": null,
    "sas_device_handle": null,
    "scheduler_mode": "deadline",
    "sectors": "167772160",
    "sectorsize": "512",
    "size": "80.00 GB",
    "support_discard": "0",
    "vendor": null,
    "virtual": 1
}
},
"ansible_distribution": "CentOS",
"ansible_distribution_file_parsed": true,
"ansible_distribution_file_path": "/etc/redhat-release",
"ansible_distribution_file_variety": "RedHat",
"ansible_distribution_major_version": "7",
"ansible_distribution_release": "Core",

```

```

"ansible_distribution_version": "7.5.1804",
"ansible_dns": {
  "nameservers": [
    "127.0.0.1"
  ]
},
"ansible_domain": "",
"ansible_effective_group_id": 1000,
"ansible_effective_user_id": 1000,
"ansible_env": {
  "HOME": "/home/zuul",
  "LANG": "en_US.UTF-8",
  "LESSOPEN": "|/usr/bin/lesspipe.sh %s",
  "LOGNAME": "zuul",
  "MAIL": "/var/mail/zuul",
  "PATH": "/usr/local/bin:/usr/bin",
  "PWD": "/home/zuul",
  "SELINUX_LEVEL_REQUESTED": "",
  "SELINUX_ROLE_REQUESTED": "",
  "SELINUX_USE_CURRENT_RANGE": "",
  "SHELL": "/bin/bash",
  "SHLVL": "2",
  "SSH_CLIENT": "REDACTED 55672 22",
  "SSH_CONNECTION": "REDACTED 55672 REDACTED 22",
  "USER": "zuul",
  "XDG_RUNTIME_DIR": "/run/user/1000",
  "XDG_SESSION_ID": "1",
  "_": "/usr/bin/python2"
},
"ansible_eth0": {
  "active": true,
  "device": "eth0",
  "ipv4": {
    "address": "REDACTED",
    "broadcast": "REDACTED",
    "netmask": "255.255.255.0",
    "network": "REDACTED"
  },
  "ipv6": [
    {
      "address": "REDACTED",
      "prefix": "64",
      "scope": "link"
    }
  ],
  "macaddress": "REDACTED",
  "module": "xen_netfront",
  "mtu": 1500,
  "pciid": "vif-0",
  "promisc": false,
  "type": "ether"
},
"ansible_eth1": {
  "active": true,
  "device": "eth1",
  "ipv4": {
    "address": "REDACTED",
    "broadcast": "REDACTED",
    "netmask": "255.255.224.0",
    "network": "REDACTED"
  },
  "ipv6": [
    {
      "address": "REDACTED",

```

```

        "prefix": "64",
        "scope": "link"
    }
],
"macaddress": "REDACTED",
"module": "xen_netfront",
"mtu": 1500,
"pciid": "vif-1",
"promisc": false,
"type": "ether"
},
"ansible_fips": false,
"ansible_form_factor": "Other",
"ansible_fqdn": "centos-7-rax-dfw-0003427354",
"ansible_hostname": "centos-7-rax-dfw-0003427354",
"ansible_interfaces": [
    "lo",
    "eth1",
    "eth0"
],
"ansible_is_chroot": false,
"ansible_kernel": "3.10.0-862.14.4.el7.x86_64",
"ansible_lo": {
    "active": true,
    "device": "lo",
    "ipv4": {
        "address": "127.0.0.1",
        "broadcast": "host",
        "netmask": "255.0.0.0",
        "network": "127.0.0.0"
    },
    "ipv6": [
        {
            "address": "::1",
            "prefix": "128",
            "scope": "host"
        }
    ],
    "mtu": 65536,
    "promisc": false,
    "type": "loopback"
},
"ansible_local": {},
"ansible_lsb": {
    "codename": "Core",
    "description": "CentOS Linux release 7.5.1804 (Core)",
    "id": "CentOS",
    "major_release": "7",
    "release": "7.5.1804"
},
"ansible_machine": "x86_64",
"ansible_machine_id": "2db133253c984c82aef2fafcce6f2bed",
"ansible_memfree_mb": 7709,
"ansible_memory_mb": {
    "nocache": {
        "free": 7804,
        "used": 173
    },
    "real": {
        "free": 7709,
        "total": 7977,
        "used": 268
    },
    "swap": {

```

```

        "cached": 0,
        "free": 0,
        "total": 0,
        "used": 0
    }
},
"ansible_memtotal_mb": 7977,
"ansible_mounts": [
    {
        "block_available": 7220998,
        "block_size": 4096,
        "block_total": 9817227,
        "block_used": 2596229,
        "device": "/dev/xvda1",
        "fstype": "ext4",
        "inode_available": 10052341,
        "inode_total": 10419200,
        "inode_used": 366859,
        "mount": "/",
        "options": "rw,seclabel,relatime,data=ordered",
        "size_available": 29577207808,
        "size_total": 40211361792,
        "uuid": "cac81d61-d0f8-4b47-84aa-b48798239164"
    },
    {
        "block_available": 0,
        "block_size": 2048,
        "block_total": 252,
        "block_used": 252,
        "device": "/dev/xvdd",
        "fstype": "iso9660",
        "inode_available": 0,
        "inode_total": 0,
        "inode_used": 0,
        "mount": "/mnt/config",
        "options": "ro,relatime,mode=0700",
        "size_available": 0,
        "size_total": 516096,
        "uuid": "2018-10-25-12-05-57-00"
    }
],
"ansible_nodename": "centos-7-rax-dfw-0003427354",
"ansible_os_family": "RedHat",
"ansible_pkg_mgr": "yum",
"ansible_processor": [
    "0",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "1",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "2",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "3",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "4",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "5",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "6",

```

```

    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz",
    "7",
    "GenuineIntel",
    "Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz"
],
"ansible_processor_cores": 8,
"ansible_processor_count": 8,
"ansible_processor_nproc": 8,
"ansible_processor_threads_per_core": 1,
"ansible_processor_vcpus": 8,
"ansible_product_name": "HVM domU",
"ansible_product_serial": "REDACTED",
"ansible_product_uuid": "REDACTED",
"ansible_product_version": "4.1.5",
"ansible_python": {
    "executable": "/usr/bin/python2",
    "has_sslcontext": true,
    "type": "CPython",
    "version": {
        "major": 2,
        "micro": 5,
        "minor": 7,
        "releaselevel": "final",
        "serial": 0
    },
    "version_info": [
        2,
        7,
        5,
        "final",
        0
    ]
},
"ansible_python_version": "2.7.5",
"ansible_real_group_id": 1000,
"ansible_real_user_id": 1000,
"ansible_selinux": {
    "config_mode": "enforcing",
    "mode": "enforcing",
    "policyvers": 31,
    "status": "enabled",
    "type": "targeted"
},
"ansible_selinux_python_present": true,
"ansible_service_mgr": "systemd",
"ansible_ssh_host_key_ecdsa_public": "REDACTED KEY VALUE",
"ansible_ssh_host_key_ed25519_public": "REDACTED KEY VALUE",
"ansible_ssh_host_key_rsa_public": "REDACTED KEY VALUE",
"ansible_swapfree_mb": 0,
"ansible_swaptotal_mb": 0,
"ansible_system": "Linux",
"ansible_system_capabilities": [
    ""
],
"ansible_system_capabilities_enforced": "True",
"ansible_system_vendor": "Xen",
"ansible_uptime_seconds": 151,
"ansible_user_dir": "/home/zuul",
"ansible_user_gecos": "",
"ansible_user_gid": 1000,
"ansible_user_id": "zuul",
"ansible_user_shell": "/bin/bash",
"ansible_user_uid": 1000,

```

```
"ansible_userspace_architecture": "x86_64",
"ansible_userspace_bits": "64",
"ansible_virtualization_role": "guest",
"ansible_virtualization_type": "xen",
"gather_subset": [
    "all"
],
"module_setup": true
}
```

You can reference the model of the first disk in the facts shown above in a template or playbook as:

```
{{ ansible_facts['devices']['xvda']['model'] }}
```

To reference the system hostname:

```
{{ ansible_facts['nodename'] }}
```

You can use facts in conditionals (see [Conditionals \(playbooks conditionals.html#playbooks-conditionals\)](#)) and also in templates. You can also use facts to create dynamic groups of hosts that match particular criteria, see the [group by module \(../collections/ansible/builtin/group_by_module.html#group-by-module\)](#) documentation for details.

Note

Because `ansible_date_time` is created and cached when Ansible gathers facts before each playbook run, it can get stale with long-running playbooks. If your playbook takes a long time to run, use the `pipe` filter (for example, `lookup('pipe', 'date +%Y-%m-%d.%H:%M:%S')`) or `now()` ([playbooks templating.html#templating-now](#)) with a Jinja 2 template instead of `ansible_date_time`.

Package requirements for fact gathering

On some distros, you may see missing fact values or facts set to default values because the packages that support gathering those facts are not installed by default. You can install the necessary packages on your remote hosts using the OS package manager. Known dependencies include:

- Linux Network fact gathering - Depends on the `ip` binary, commonly included in the `iproute2` package.

Caching facts

Like registered variables, facts are stored in memory by default. However, unlike registered variables, facts can be gathered independently and cached for repeated use. With cached facts, you can refer to facts from one system when configuring a second system, even if Ansible executes the current play on the second system first. For example:

```
{{ hostvars['asdf.example.com']['ansible_facts']['os_family'] }}
```

Caching is controlled by the cache plugins. By default, Ansible uses the memory cache plugin, which stores facts in memory for the duration of the current playbook run. To retain Ansible facts for repeated use, select a different cache plugin. See [Cache plugins](#) ([../plugins/cache.html#cache-plugins](#)) for details.

Fact caching can improve performance. If you manage thousands of hosts, you can configure fact caching to run nightly, then manage configuration on a smaller set of servers periodically throughout the day. With cached facts, you have access to variables and information about all hosts even when you are only managing a small number of servers.

Disabling facts

By default, Ansible gathers facts at the beginning of each play. If you do not need to gather facts (for example, if you know everything about your systems centrally), you can turn off fact gathering at the play level to improve scalability. Disabling facts may particularly improve performance in push mode with very large numbers of systems, or if you are using Ansible on experimental platforms. To disable fact gathering:

```
- hosts: whatever
  gather_facts: no
```

Adding custom facts

The setup module in Ansible automatically discovers a standard set of facts about each host. If you want to add custom values to your facts, you can write a custom facts module, set temporary facts with a `ansible.builtin.set_fact` task, or provide permanent custom facts using the facts.d directory.

facts.d or local facts

New in version 1.3.

You can add static custom facts by adding static files to facts.d, or add dynamic facts by adding executable scripts to facts.d. For example, you can add a list of all users on a host to your facts by creating and running a script in facts.d.

To use facts.d, create an `/etc/ansible/facts.d` directory on the remote host or hosts. If you prefer a different directory, create it and specify it using the `fact_path` play keyword. Add files to the directory to supply your custom facts. All file names must end with `.fact`. The files can be JSON, INI, or executable files returning JSON.

To add static facts, simply add a file with the `.fact` extension. For example, create `/etc/ansible/facts.d/preferences.fact` with this content:

```
[general]
asdf=1
bar=2
```

Note

Make sure the file is not executable as this will break the `ansible.builtin.setup` module.

The next time fact gathering runs, your facts will include a hash variable fact named `general` with `asdf` and `bar` as members. To validate this, run the following:

```
ansible <hostname> -m ansible.builtin.setup -a "filter=ansible_local"
```

And you will see your custom fact added:

```
{
  "ansible_local": {
    "preferences": {
      "general": {
        "asdf" : "1",
        "bar"  : "2"
      }
    }
  }
}
```

The `ansible_local` namespace separates custom facts created by facts.d from system facts or variables defined elsewhere in the playbook, so variables will not override each other. You can access this custom fact in a template or playbook as:


```
{{ ansible_local['preferences']['general']['asdf'] }}
```

❗ Note

The key part in the key=value pairs will be converted into lowercase inside the `ansible_local` variable. Using the example above, if the ini file contained `XYZ=3` in the `[general]` section, then you should expect to access it as: `{{ ansible_local['preferences']['general']['xyz'] }}` and not `{{ ansible_local['preferences']['general']['XYZ'] }}`. This is because Ansible uses Python's `ConfigParser` (<https://docs.python.org/3/library/configparser.html>) which passes all option names through the `optionxform` (<https://docs.python.org/3/library/configparser.html#ConfigParser.RawConfigParser.optionxform>), method and this method's default implementation converts option names to lower case.

You can also use `facts.d` to execute a script on the remote host, generating dynamic custom facts to the `ansible_local` namespace. For example, you can generate a list of all users that exist on a remote host as a fact about that host. To generate dynamic custom facts using `facts.d`:

1. Write and test a script to generate the JSON data you want.
2. Save the script in your `facts.d` directory.
3. Make sure your script has the `.fact` file extension.
4. Make sure your script is executable by the Ansible connection user.
5. Gather facts to execute the script and add the JSON output to `ansible_local`.

By default, fact gathering runs once at the beginning of each play. If you create a custom fact using `facts.d` in a playbook, it will be available in the next play that gathers facts. If you want to use it in the same play where you created it, you must explicitly re-run the setup module. For example:

```
- hosts: webservers
  tasks:

    - name: Create directory for ansible custom facts
      ansible.builtin.file:
        state: directory
        recurse: yes
        path: /etc/ansible/facts.d

    - name: Install custom ipmi fact
      ansible.builtin.copy:
        src: ipmi.fact
        dest: /etc/ansible/facts.d

    - name: Re-read facts after adding custom fact
      ansible.builtin.setup:
        filter: ansible_local
```

If you use this pattern frequently, a custom facts module would be more efficient than facts.d.

Information about Ansible: magic variables

You can access information about Ansible operations, including the python version being used, the hosts and groups in inventory, and the directories for playbooks and roles, using “magic” variables. Like connection variables, magic variables are [Special Variables](https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html#special-variables) ([../reference_appendices/special_variables.html#special-variables](https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html#special-variables)). Magic variable names are reserved - do not set variables with these names. The variable `environment` is also reserved.

The most commonly used magic variables are `hostvars`, `groups`, `group_names`, and `inventory_hostname`. With `hostvars`, you can access variables defined for any host in the play, at any point in a playbook. You can access Ansible facts using the `hostvars` variable too, but only after you have gathered (or cached) facts.

If you want to configure your database server using the value of a ‘fact’ from another node, or the value of an inventory variable assigned to another node, you can use `hostvars` in a template or on an action line:

```
{{ hostvars['test.example.com']['ansible_facts']['distribution'] }}
```

With `groups`, a list of all the groups (and hosts) in the inventory, you can enumerate all hosts within a group. For example:

```
{% for host in groups['app_servers'] %}  
    # something that applies to all app servers.  
{% endfor %}
```

You can use `groups` and `hostvars` together to find all the IP addresses in a group.

```
{% for host in groups['app_servers'] %}  
    {{ hostvars[host]['ansible_facts']['eth0']['ipv4']['address'] }}  
{% endfor %}
```

You can use this approach to point a frontend proxy server to all the hosts in your app servers group, to set up the correct firewall rules between servers, and so on. You must either cache facts or gather facts for those hosts before the task that fills out the template.

With `group_names`, a list (array) of all the groups the current host is in, you can create templated files that vary based on the group membership (or role) of the host:

```
{% if 'webserver' in group_names %}
    # some part of a configuration file that only applies to webserver
{% endif %}
```

You can use the magic variable `inventory_hostname`, the name of the host as configured in your inventory, as an alternative to `ansible_hostname` when fact-gathering is disabled. If you have a long FQDN, you can use `inventory_hostname_short`, which contains the part up to the first period, without the rest of the domain.

Other useful magic variables refer to the current play or playbook. These vars may be useful for filling out templates with multiple hostnames or for injecting the list into the rules for a load balancer.

`ansible_play_hosts` is the list of all hosts still active in the current play.

`ansible_play_batch` is a list of hostnames that are in scope for the current 'batch' of the play.

The batch size is defined by `serial`, when not set it is equivalent to the whole play (making it the same as `ansible_play_hosts`).

`ansible_playbook_python` is the path to the python executable used to invoke the Ansible command line tool.

`inventory_dir` is the pathname of the directory holding Ansible's inventory host file.

`inventory_file` is the pathname and the filename pointing to the Ansible's inventory host file.

`playbook_dir` contains the playbook base directory.

`role_path` contains the current role's pathname and only works inside a role.

`ansible_check_mode` is a boolean, set to `True` if you run Ansible with `--check`.

Ansible version

New in version 1.8.

To adapt playbook behavior to different versions of Ansible, you can use the variable `ansible_version`, which has the following structure:

```
{
  "ansible_version": {
    "full": "2.10.1",
    "major": 2,
    "minor": 10,
    "revision": 1,
    "string": "2.10.1"
  }
}
```

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Encrypting content with Ansible Vault

Ansible Vault encrypts variables and files so you can protect sensitive content such as passwords or keys rather than leaving it visible as plaintext in playbooks or roles. To use Ansible Vault you need one or more passwords to encrypt and decrypt content. If you store your vault passwords in a third-party tool such as a secret manager, you need a script to access them. Use the passwords with the [ansible-vault \(../cli/ansible-vault.html#ansible-vault\)](#) command-line tool to create and view encrypted variables, create encrypted files, encrypt existing files, or edit, re-key, or decrypt files. You can then place encrypted content under source control and share it more safely.

⚠ Warning

- Encryption with Ansible Vault **ONLY** protects ‘data at rest’. Once the content is decrypted (‘data in use’), play and plugin authors are responsible for avoiding any secret disclosure, see [no_log \(../reference/appendices/faq.html#keep-secret-data\)](#) for details on hiding output and [Steps to secure your editor](#) for security considerations on editors you use with Ansible Vault.

You can use encrypted variables and files in ad hoc commands and playbooks by supplying the passwords you used to encrypt them. You can modify your `ansible.cfg` file to specify the location of a password file or to always prompt for the password.

- [Managing vault passwords](#)
 - [Choosing between a single password and multiple passwords](#)
 - [Managing multiple passwords with vault IDs](#)
 - [Limitations of vault IDs](#)
 - [Enforcing vault ID matching](#)
 - [Storing and accessing vault passwords](#)
 - [Storing passwords in files](#)
 - [Storing passwords in third-party tools with vault password client scripts](#)
- [Encrypting content with Ansible Vault](#)
 - [Encrypting individual variables with Ansible Vault](#)
 - [Advantages and disadvantages of encrypting variables](#)
 - [Creating encrypted variables](#)

- [Viewing encrypted variables](#)
- [Encrypting files with Ansible Vault](#)
 - [Advantages and disadvantages of encrypting files](#)
 - [Creating encrypted files](#)
 - [Encrypting existing files](#)
 - [Viewing encrypted files](#)
 - [Editing encrypted files](#)
 - [Changing the password and/or vault ID on encrypted files](#)
 - [Decrypting encrypted files](#)
 - [Steps to secure your editor](#)
 - [vim](#)
 - [Emacs](#)
- [Using encrypted variables and files](#)
 - [Passing a single password](#)
 - [Passing vault IDs](#)
 - [Passing multiple vault passwords](#)
 - [Using `--vault-id` without a vault ID](#)
- [Configuring defaults for using encrypted content](#)
 - [Setting a default vault ID](#)
 - [Setting a default password source](#)
- [When are encrypted files made visible?](#)
- [Format of files encrypted with Ansible Vault](#)
 - [Ansible Vault payload format 1.1 - 1.2](#)

Managing vault passwords

Managing your encrypted content is easier if you develop a strategy for managing your vault passwords. A vault password can be any string you choose. There is no special command to create a vault password. However, you need to keep track of your vault passwords. Each time you encrypt a variable or file with Ansible Vault, you must provide a password. When you use an encrypted variable or file in a command or playbook, you must provide the same password that was used to encrypt it. To develop a strategy for managing vault passwords, start with two questions:

- Do you want to encrypt all your content with the same password, or use different passwords for different needs?
- Where do you want to store your password or passwords?

Choosing between a single password and multiple passwords

If you have a small team or few sensitive values, you can use a single password for everything you encrypt with Ansible Vault. Store your vault password securely in a file or a secret manager as described below.

If you have a larger team or many sensitive values, you can use multiple passwords. For example, you can use different passwords for different users or different levels of access. Depending on your needs, you might want a different password for each encrypted file, for each directory, or for each environment. For example, you might have a playbook that includes two vars files, one for the dev environment and one for the production environment, encrypted with two different passwords. When you run the playbook, select the correct vault password for the environment you are targeting, using a vault ID.

Managing multiple passwords with vault IDs

If you use multiple vault passwords, you can differentiate one password from another with vault IDs. You use the vault ID in three ways:

- Pass it with `--vault-id` (`../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id`) to the `ansible-vault` (`../cli/ansible-vault.html#ansible-vault`) command when you create encrypted content
- Include it wherever you store the password for that vault ID (see [Storing and accessing vault passwords](#))
- Pass it with `--vault-id` (`../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id`) to the `ansible-playbook` (`../cli/ansible-playbook.html#ansible-playbook`) command when you run a playbook that uses content you encrypted with that vault ID

When you pass a vault ID as an option to the `ansible-vault` (`../cli/ansible-vault.html#ansible-vault`) command, you add a label (a hint or nickname) to the encrypted content. This label documents which password you used to encrypt it. The encrypted variable or file includes the vault ID label in plain text in the header. The vault ID is the last element before the encrypted content. For example:

```
my_encrypted_var: !vault |
    $ANSIBLE_VAULT;1.2;AES256;dev

30613233633461343837653833666333643061636561303338373661313838333565653635353162

3263363434623733343538653462613064333634333464660a663633623939393439316636633863

61636237636537333938306331383339353265363239643939666639386530626330633337633833

6664656334373166630a363736393262666465663432613932613036303963343263623137386239

6330
```

In addition to the label, you must provide a source for the related password. The source can be a prompt, a file, or a script, depending on how you are storing your vault passwords. The pattern looks like this:

```
--vault-id label@source
```

If your playbook uses multiple encrypted variables or files that you encrypted with different passwords, you must pass the vault IDs when you run that playbook. You can use `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)) by itself, with `--vault-password-file` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-password-file](#)), or with `--ask-vault-pass` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-ask-vault-password](#)). The pattern is the same as when you create encrypted content: include the label and the source for the matching password.

See below for examples of encrypting content with vault IDs and using content encrypted with vault IDs. The `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)) option works with any Ansible command that interacts with vaults, including `ansible-vault` ([../cli/ansible-vault.html#ansible-vault](#)), `ansible-playbook` ([../cli/ansible-playbook.html#ansible-playbook](#)), and so on.

Limitations of vault IDs

Ansible does not enforce using the same password every time you use a particular vault ID label. You can encrypt different variables or files with the same vault ID label but different passwords. This usually happens when you type the password at a prompt and make a mistake. It is possible to use different passwords with the same vault ID label on purpose. For example, you could use each label as a reference to a class of passwords, rather than a single password. In this scenario, you must always know which specific password or file to use in context. However, you are more likely to encrypt two files with the same vault ID label and different passwords by mistake. If you encrypt two files with the same label but different passwords by accident, you can [rekey](#) one file to fix the issue.

Enforcing vault ID matching

By default the vault ID label is only a hint to remind you which password you used to encrypt a variable or file. Ansible does not check that the vault ID in the header of the encrypted content matches the vault ID you provide when you use the content. Ansible decrypts all files and variables called by your command or playbook that are encrypted with the password you provide. To check the encrypted content and decrypt it only when the vault ID it contains matches the one you provide with `--vault-id`, set the config option `DEFAULT VAULT ID MATCH` ([../reference_appendices/config.html#default-vault-id-match](#)). When you set `DEFAULT VAULT ID MATCH` ([../reference_appendices/config.html#default-vault-id-match](#)), each password is only used to decrypt data that was encrypted with the same label. This is efficient, predictable, and can reduce errors when different values are encrypted with different passwords.

Even with the DEFAULT VAULT ID MATCH

([../reference_appendices/config.html#default-vault-id-match](#)), setting enabled, Ansible does not enforce using the same password every time you use a particular vault ID label.

Storing and accessing vault passwords

You can memorize your vault password, or manually copy vault passwords from any source and paste them at a command-line prompt, but most users store them securely and access them as needed from within Ansible. You have two options for storing vault passwords that work from within Ansible: in files, or in a third-party tool such as the system keyring or a secret manager. If you store your passwords in a third-party tool, you need a vault password client script to retrieve them from within Ansible.

Storing passwords in files

To store a vault password in a file, enter the password as a string on a single line in the file. Make sure the permissions on the file are appropriate. Do not add password files to source control.

Storing passwords in third-party tools with vault password client scripts

You can store your vault passwords on the system keyring, in a database, or in a secret manager and retrieve them from within Ansible using a vault password client script. Enter the password as a string on a single line. If your password has a vault ID, store it in a way that works with your password storage tool.

To create a vault password client script:

- Create a file with a name ending in either `-client` or `-client.EXTENSION`
- Make the file executable
- **Within the script itself:**
 - Print the passwords to standard output
 - Accept a `--vault-id` option
 - If the script prompts for data (for example, a database password), send the prompts to standard error

When you run a playbook that uses vault passwords stored in a third-party tool, specify the script as the source within the `--vault-id` flag. For example:

```
ansible-playbook --vault-id dev@contrib/vault/vault-keyring-client.py
```

Ansible executes the client script with a `--vault-id` option so the script knows which vault ID label you specified. For example a script loading passwords from a secret manager can use the vault ID label to pick either the 'dev' or 'prod' password. The example command above results in the following execution of the client script:

```
contrib/vault/vault-keyring-client.py --vault-id dev
```

For an example of a client script that loads passwords from the system keyring, see the [vault-keyring-client script \(https://github.com/ansible-community/contrib-scripts/blob/main/vault/vault-keyring-client.py\)](https://github.com/ansible-community/contrib-scripts/blob/main/vault/vault-keyring-client.py).

Encrypting content with Ansible Vault

Once you have a strategy for managing and storing vault passwords, you can start encrypting content. You can encrypt two types of content with Ansible Vault: variables and files. Encrypted content always includes the `!vault` tag, which tells Ansible and YAML that the content needs to be decrypted, and a `|` character, which allows multi-line strings. Encrypted content created with `--vault-id` also contains the vault ID label. For more details about the encryption process and the format of content encrypted with Ansible Vault, see [Format of files encrypted with Ansible Vault](#). This table shows the main differences between encrypted variables and encrypted files:

	Encrypted variables	Encrypted files
How much is encrypted?	Variables within a plaintext file	The entire file
When is it decrypted?	On demand, only when needed	Whenever loaded or referenced ¹
What can be encrypted?	Only variables	Any structured data file

[1] : Ansible cannot know if it needs content from an encrypted file unless it decrypts the file, so it decrypts all encrypted files referenced in your playbooks and roles.

Encrypting individual variables with Ansible Vault

You can encrypt single values inside a YAML file using the [ansible-vault encrypt_string \(../cli/ansible-vault.html#ansible-vault-encrypt-string\)](#) command. For one way to keep your vaulted variables safely visible, see [Keep vaulted variables safely visible \(playbooks_best_practices.html#tip-for-variables-and-vaults\)](#).

Advantages and disadvantages of encrypting variables

With variable-level encryption, your files are still easily legible. You can mix plaintext and encrypted variables, even inline in a play or role. However, password rotation is not as simple as with file-level encryption. You cannot rekey encrypted variables. Also, variable-level encryption only works on variables. If you want to encrypt tasks or other content, you must encrypt the entire file.

Creating encrypted variables

The `ansible-vault encrypt_string` ([./cli/ansible-vault.html#ansible-vault-encrypt-string](https://docs.ansible.com/ansible/latest/cli/ansible-vault.html#ansible-vault-encrypt-string)) command encrypts and formats any string you type (or copy or generate) into a format that can be included in a playbook, role, or variables file. To create a basic encrypted variable, pass three options to the `ansible-vault encrypt_string` ([./cli/ansible-vault.html#ansible-vault-encrypt-string](https://docs.ansible.com/ansible/latest/cli/ansible-vault.html#ansible-vault-encrypt-string)) command:

- a source for the vault password (prompt, file, or script, with or without a vault ID)
- the string to encrypt
- the string name (the name of the variable)

The pattern looks like this:

```
ansible-vault encrypt_string <password_source> '<string_to_encrypt>' --name  
'<string_name_of_variable>'
```

For example, to encrypt the string 'foobar' using the only password stored in 'a_password_file' and name the variable 'the_secret':

```
ansible-vault encrypt_string --vault-password-file a_password_file 'foobar' --name  
'the_secret'
```

The command above creates this content:

```
the_secret: !vault |
    $ANSIBLE_VAULT;1.1;AES256

62313365396662343061393464336163383764373764613633653634306231386433626436623361

6134333665353966363534333632666535333761666131620a663537646436643839616531643561

63396265333966386166373632626539326166353965363262633030333630313338646335303630

3438626666666137650a353638643435666633633964366338633066623234616432373231333331
6564
```

To encrypt the string 'fooodev', add the vault ID label 'dev' with the 'dev' vault password stored in 'a_password_file', and call the encrypted variable 'the_dev_secret':

```
ansible-vault encrypt_string --vault-id dev@a_password_file 'fooodev' --name  
'the_dev_secret'
```

The command above creates this content:

```
the_dev_secret: !vault |  
    $ANSIBLE_VAULT;1.2;AES256;dev  
  
30613233633461343837653833666333643061636561303338373661313838333565653635353162  
  
3263363434623733343538653462613064333634333464660a663633623939393439316636633863  
  
61636237636537333938306331383339353265363239643939666639386530626330633337633833  
  
6664656334373166630a363736393262666465663432613932613036303963343263623137386239  
6330
```

To encrypt the string 'letmein' read from stdin, add the vault ID 'dev' using the 'dev' vault password stored in *a_password_file*, and name the variable 'db_password':

```
echo -n 'letmein' | ansible-vault encrypt_string --vault-id dev@a_password_file --  
stdin-name 'db_password'
```

⚠ Warning

Typing secret content directly at the command line (without a prompt) leaves the secret string in your shell history. Do not do this outside of testing.

The command above creates this output:

```
Reading plaintext input from stdin. (ctrl-d to end input, twice if your content
does not already have a new line)
db_password: !vault |
           $ANSIBLE_VAULT;1.2;AES256;dev

61323931353866666336306139373937316366366138656131323863373866376666353364373761

3539633234313836346435323766306164626134376564330a373530313635343535343133316133

36643666306434616266376434363239346433643238336464643566386135356334303736353136

6565633133366366360a326566323363363936613664616364623437336130623133343530333739
3039
```

To be prompted for a string to encrypt, encrypt it with the ‘dev’ vault password from ‘a_password_file’, name the variable ‘new_user_password’ and give it the vault ID label ‘dev’:

```
ansible-vault encrypt_string --vault-id dev@a_password_file --stdin-name
'new_user_password'
```

The command above triggers this prompt:

```
Reading plaintext input from stdin. (ctrl-d to end input, twice if your content does
not already have a new line)
```

Type the string to encrypt (for example, ‘hunter2’), hit ctrl-d, and wait.

⚠ Warning

Do not press after supplying the string to encrypt. That will add a newline to the encrypted value.

The sequence above creates this output:

```
new_user_password: !vault |
           $ANSIBLE_VAULT;1.2;AES256;dev

37636561366636643464376336303466613062633537323632306566653533383833366462366662

6565353063303065303831323539656138653863353230620a653638643639333133306331336365

62373737623337616130386137373461306535383538373162316263386165376131623631323434

3866363862363335620a376466656164383032633338306162326639643635663936623939666238
3161
```

You can add the output from any of the examples above to any playbook, variables file, or role for future use. Encrypted variables are larger than plain-text variables, but they protect your sensitive content while leaving the rest of the playbook, variables file, or role in plain text so you can easily read it.

Viewing encrypted variables

You can view the original value of an encrypted variable using the debug module. You must pass the password that was used to encrypt the variable. For example, if you stored the variable created by the last example above in a file called 'vars.yml', you could view the unencrypted value of that variable like this:

```
ansible localhost -m ansible.builtin.debug -a var="new_user_password" -e "@vars.yml" --vault-id dev@a_password_file

localhost | SUCCESS => {
  "new_user_password": "hunter2"
}
```

Encrypting files with Ansible Vault

Ansible Vault can encrypt any structured data file used by Ansible, including:

- group variables files from inventory
- host variables files from inventory
- variables files passed to ansible-playbook with `-e @file.yml` or `-e @file.json`
- variables files loaded by `include_vars` or `vars_files`
- variables files in roles
- defaults files in roles
- tasks files
- handlers files
- binary files or other arbitrary files

The full file is encrypted in the vault.

❗ Note

Ansible Vault uses an editor to create or modify encrypted files. See [Steps to secure your editor](#) for some guidance on securing the editor.

Advantages and disadvantages of encrypting files

File-level encryption is easy to use. Password rotation for encrypted files is straightforward with the `rekey` command. Encrypting files can hide not only sensitive values, but the names of the variables you use. However, with file-level encryption the contents of files are no longer easy to access and read. This may be a problem with encrypted tasks files. When encrypting a variables file, see [Keep vaulted variables safely visible \(playbooks best practices.html#tip-for-variables-and-vaults\)](#) for one way to keep references to these variables in a non-encrypted file. Ansible always decrypts the entire encrypted file when it is when loaded or referenced, because Ansible cannot know if it needs the content unless it decrypts it.

Creating encrypted files

To create a new encrypted data file called 'foo.yml' with the 'test' vault password from 'multi_password_file':

```
ansible-vault create --vault-id test@multi_password_file foo.yml
```

The tool launches an editor (whatever editor you have defined with `$EDITOR`, default editor is `vi`). Add the content. When you close the editor session, the file is saved as encrypted data. The file header reflects the vault ID used to create it:

```
``$ANSIBLE_VAULT;1.2;AES256;test``
```

To create a new encrypted data file with the vault ID 'my_new_password' assigned to it and be prompted for the password:

```
ansible-vault create --vault-id my_new_password@prompt foo.yml
```

Again, add content to the file in the editor and save. Be sure to store the new password you created at the prompt, so you can find it when you want to decrypt that file.

Encrypting existing files

To encrypt an existing file, use the [ansible-vault encrypt\(../cli/ansible-vault.html#ansible-vault-encrypt\)](#) command. This command can operate on multiple files at once. For example:

```
ansible-vault encrypt foo.yml bar.yml baz.yml
```

To encrypt existing files with the 'project' ID and be prompted for the password: [Search this site](#)

```
ansible-vault encrypt --vault-id project@prompt foo.yml bar.yml baz.yml
```

Viewing encrypted files

To view the contents of an encrypted file without editing it, you can use the [ansible-vault view](#) ([../cli/ansible-vault.html#ansible-vault-view](#)) command:

```
ansible-vault view foo.yml bar.yml baz.yml
```

Editing encrypted files

To edit an encrypted file in place, use the [ansible-vault edit](#) ([../cli/ansible-vault.html#ansible-vault-edit](#)) command. This command decrypts the file to a temporary file, allows you to edit the content, then saves and re-encrypts the content and removes the temporary file when you close the editor. For example:

```
ansible-vault edit foo.yml
```

To edit a file encrypted with the `vault2` password file and assigned the vault ID `pass2`:

```
ansible-vault edit --vault-id pass2@vault2 foo.yml
```

Changing the password and/or vault ID on encrypted files

To change the password on an encrypted file or files, use the [rekey](#) ([../cli/ansible-vault.html#ansible-vault-rekey](#)) command:

```
ansible-vault rekey foo.yml bar.yml baz.yml
```

This command can rekey multiple data files at once and will ask for the original password and also the new password. To set a different ID for the rekeyed files, pass the new ID to `--new-vault-id`. For example, to rekey a list of files encrypted with the 'preprod1' vault ID from the 'ppold' file to the 'preprod2' vault ID and be prompted for the new password:

```
ansible-vault rekey --vault-id preprod1@ppold --new-vault-id preprod2@prompt foo.yml  
bar.yml baz.yml
```


Decrypting encrypted files

If you have an encrypted file that you no longer want to keep encrypted, you can permanently decrypt it by running the `ansible-vault decrypt` ([./cli/ansible-vault.html#ansible-vault-decrypt](https://docs.ansible.com/ansible/latest/cli/ansible-vault.html#ansible-vault-decrypt)) command. This command will save the file unencrypted to the disk, so be sure you do not want to `edit` ([./cli/ansible-vault.html#ansible-vault-edit](https://docs.ansible.com/ansible/latest/cli/ansible-vault.html#ansible-vault-edit)), it instead.

```
ansible-vault decrypt foo.yml bar.yml baz.yml
```

Steps to secure your editor

Ansible Vault relies on your configured editor, which can be a source of disclosures. Most editors have ways to prevent loss of data, but these normally rely on extra plain text files that can have a clear text copy of your secrets. Consult your editor documentation to configure the editor to avoid disclosing secure data. The following sections provide some guidance on common editors but should not be taken as a complete guide to securing your editor.

vim

You can set the following `vim` options in command mode to avoid cases of disclosure. There may be more settings you need to modify to ensure security, especially when using plugins, so consult the `vim` documentation.

1. Disable swapfiles that act like an autosave in case of crash or interruption.

```
set noswapfile
```

2. Disable creation of backup files.

```
set nobackup
set nowritebackup
```

3. Disable the viminfo file from copying data from your current session.

```
set viminfo=
```

4. Disable copying to the system clipboard.

```
set clipboard=
```

You can optionally add these settings in `.vimrc` for all files, or just specific paths or extensions. See the `vim` manual for details.

Emacs

You can set the following Emacs options to avoid cases of disclosure. There may be more settings you need to modify to ensure security, especially when using plugins, so consult the Emacs documentation.

1. Do not copy data to the system clipboard.

```
(setq x-select-enable-clipboard nil)
```

2. Disable creation of backup files.

```
(setq make-backup-files nil)
```

3. Disable autosave files.

```
(setq auto-save-default nil)
```

Using encrypted variables and files

When you run a task or playbook that uses encrypted variables or files, you must provide the passwords to decrypt the variables or files. You can do this at the command line or in the playbook itself.

Passing a single password

If all the encrypted variables and files your task or playbook needs use a single password, you can use the `--ask-vault-pass` (`./cli/ansible-playbook.html#cmdoption-ansible-playbook-ask-vault-password`) or `--vault-password-file` (`./cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-password-file`) cli options.

To prompt for the password:

```
ansible-playbook --ask-vault-pass site.yml
```

To retrieve the password from the `/path/to/my/vault-password-file` file:

```
ansible-playbook --vault-password-file /path/to/my/vault-password-file site.yml
```

To get the password from the vault password client script `my-vault-password-client.py`:

```
ansible-playbook --vault-password-file my-vault-password-client.py
```

Passing vault IDs

You can also use the `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)) option to pass a single password with its vault label. This approach is clearer when multiple vaults are used within a single inventory.

To prompt for the password for the 'dev' vault ID:

```
ansible-playbook --vault-id dev@prompt site.yml
```

To retrieve the password for the 'dev' vault ID from the `dev-password` file:

```
ansible-playbook --vault-id dev@dev-password site.yml
```

To get the password for the 'dev' vault ID from the vault password client script `my-vault-password-client.py`:

```
ansible-playbook --vault-id dev@my-vault-password-client.py
```

Passing multiple vault passwords

If your task or playbook requires multiple encrypted variables or files that you encrypted with different vault IDs, you must use the `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)) option, passing multiple `--vault-id` options to specify the vault

IDs ('dev', 'prod', 'cloud', 'db') and sources for the passwords (prompt, file, script). . For example, to use a 'dev' password read from a file and to be prompted for the 'prod' password:

```
ansible-playbook --vault-id dev@dev-password --vault-id prod@prompt site.yml
```

By default the vault ID labels (dev, prod and so on) are only hints. Ansible attempts to decrypt vault content with each password. The password with the same label as the encrypted data will be tried first, after that each vault secret will be tried in the order they were provided on the command line.

Where the encrypted data has no label, or the label does not match any of the provided labels, the passwords will be tried in the order they are specified. In the example above, the 'dev' password will be tried first, then the 'prod' password for cases where Ansible doesn't know which vault ID is used to encrypt something.

Using `--vault-id` without a vault ID

The `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)) option can also be used without specifying a vault-id. This behavior is equivalent to `--ask-vault-pass` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-ask-vault-password](#)) or `--vault-password-file` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-password-file](#)) so is rarely used.

For example, to use a password file `dev-password` :

```
ansible-playbook --vault-id dev-password site.yml
```

To prompt for the password:

```
ansible-playbook --vault-id @prompt site.yml
```

To get the password from an executable script `my-vault-password-client.py` :

```
ansible-playbook --vault-id my-vault-password-client.py
```

Configuring defaults for using encrypted content

Setting a default vault ID

If you use one vault ID more frequently than any other, you can set the config option [DEFAULT VAULT IDENTITY LIST](#) ([../reference_appendices/config.html#default-vault-identity-list](#)) to specify a default vault ID and password source. Ansible will use the default vault ID and source any time you do not specify `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)). You can set multiple values for this option. Setting multiple values is equivalent to passing multiple `--vault-id` ([../cli/ansible-playbook.html#cmdoption-ansible-playbook-vault-id](#)) cli options.

Setting a default password source

If you use one vault password file more frequently than any other, you can set the [DEFAULT VAULT PASSWORD FILE](#) ([../reference_appendices/config.html#default-vault-password-file](#)) config option or the `ANSIBLE_VAULT_PASSWORD_FILE` ([../reference_appendices/config.html#envvar-ANSIBLE_VAULT_PASSWORD_FILE](#)) environment variable to specify that file. For example, if you set `ANSIBLE_VAULT_PASSWORD_FILE=~/.vault_pass.txt`, Ansible will automatically search for the password in that file. This is useful if, for example, you use Ansible from a continuous integration system such as Jenkins.

When are encrypted files made visible?

In general, content you encrypt with Ansible Vault remains encrypted after execution. However, there is one exception. If you pass an encrypted file as the `src` argument to the [copy](#) ([../collections/ansible/builtin/copy_module.html#copy-module](#)), [template](#) ([../collections/ansible/builtin/template_module.html#template-module](#)), [unarchive](#) ([../collections/ansible/builtin/unarchive_module.html#unarchive-module](#)), [script](#) ([../collections/ansible/builtin/script_module.html#script-module](#)) or [assemble](#) ([../collections/ansible/builtin/assemble_module.html#assemble-module](#)) module, the file will not be encrypted on the target host (assuming you supply the correct vault password when you run the play). This behavior is intended and useful. You can encrypt a configuration file or template to avoid sharing the details of your configuration, but when you copy that configuration to servers in your environment, you want it to be decrypted so local users and processes can access it.

Format of files encrypted with Ansible Vault

Ansible Vault creates UTF-8 encoded txt files. The file format includes a newline terminated header. For example:

```
$ANSIBLE_VAULT;1.1;AES256
```

or

```
$ANSIBLE_VAULT;1.2;AES256;vault-id-label
```

The header contains up to four elements, separated by semi-colons (;).

1. The format ID (`$ANSIBLE_VAULT`). Currently `$ANSIBLE_VAULT` is the only valid format ID. The format ID identifies content that is encrypted with Ansible Vault (via `vault.is_encrypted_file()`).
2. The vault format version (`1.x`). All supported versions of Ansible will currently default to '1.1' or '1.2' if a labeled vault ID is supplied. The '1.0' format is supported for reading only (and will be converted automatically to the '1.1' format on write). The format version is currently used as an exact string compare only (version numbers are not currently 'compared').
3. The cipher algorithm used to encrypt the data (`AES256`). Currently `AES256` is the only supported cipher algorithm. Vault format 1.0 used 'AES', but current code always uses 'AES256'.
4. The vault ID label used to encrypt the data (optional, `vault-id-label`) For example, if you encrypt a file with `--vault-id dev@prompt` , the vault-id-label is `dev` .

Note: In the future, the header could change. Fields after the format ID and format version depend on the format version, and future vault format versions may add more cipher algorithm options and/or additional fields.

The rest of the content of the file is the 'vaulttext'. The vaulttext is a text armored version of the encrypted ciphertext. Each line is 80 characters wide, except for the last line which may be shorter.

Ansible Vault payload format 1.1 - 1.2

The vaulttext is a concatenation of the ciphertext and a SHA256 digest with the result 'hexlified'.

'hexlify' refers to the `hexlify()` method of the Python Standard Library's [binascii](https://docs.python.org/3/library/binascii.html) (<https://docs.python.org/3/library/binascii.html>), module.

hexlify()'ed result of:

- hexlify()'ed string of the salt, followed by a newline (`0x0a`)
- hexlify()'ed string of the crypted HMAC, followed by a newline. The HMAC is:
 - a [RFC2104](https://www.ietf.org/rfc/rfc2104.txt) (<https://www.ietf.org/rfc/rfc2104.txt>) style HMAC
 - inputs are:

- The AES256 encrypted ciphertext
- A PBKDF2 key. This key, the cipher key, and the cipher IV are generated from:
 - the salt, in bytes
 - 10000 iterations
 - SHA256() algorithm
 - the first 32 bytes are the cipher key
 - the second 32 bytes are the HMAC key
 - remaining 16 bytes are the cipher IV
- hexlify()'ed string of the ciphertext. The ciphertext is:
- AES256 encrypted data. The data is encrypted using:
 - AES-CTR stream cipher
 - cipher key
 - IV
 - a 128 bit counter block seeded from an integer IV
 - the plaintext
 - the original plaintext
 - padding up to the AES256 blocksize. (The data used for padding is based on [RFC5652 \(https://tools.ietf.org/html/rfc5652#section-6.3\)](https://tools.ietf.org/html/rfc5652#section-6.3))

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Interactive input: prompts

If you want your playbook to prompt the user for certain input, add a 'vars_prompt' section. Prompting the user for variables lets you avoid recording sensitive data like passwords. In addition to security, prompts support flexibility. For example, if you use one playbook across multiple software releases, you could prompt for the particular release version.

- [Encrypting values supplied by vars_prompt](#)
- [Allowing special characters in vars_prompt values](#)

Here is a most basic example:

```
---
- hosts: all
  vars_prompt:

    - name: username
      prompt: What is your username?
      private: no

    - name: password
      prompt: What is your password?

  tasks:

    - name: Print a message
      ansible.builtin.debug:
        msg: 'Logging in as {{ username }}'
```

The user input is hidden by default but it can be made visible by setting `private: no`.

Note

Prompts for individual `vars_prompt` variables will be skipped for any variable that is already defined through the command line `--extra-vars` option, or when running from a non-interactive session (such as cron or Ansible AWX). See [Defining variables at runtime \(playbooks variables.html#passing-variables-on-the-command-line\)](#).

If you have a variable that changes infrequently, you can provide a default value that can be overridden.

```
vars_prompt:  
  
- name: release_version  
  prompt: Product release version  
  default: "1.0"
```

Encrypting values supplied by `vars_prompt`

You can encrypt the entered value so you can use it, for instance, with the user module to define a password:

```
vars_prompt:  
  
- name: my_password2  
  prompt: Enter password2  
  private: yes  
  encrypt: sha512_crypt  
  confirm: yes  
  salt_size: 7
```

If you have [Passlib](https://passlib.readthedocs.io/en/stable/) (<https://passlib.readthedocs.io/en/stable/>), installed, you can use any crypt scheme the library supports:

- *des_crypt* - DES Crypt
- *bsdi_crypt* - BSDi Crypt
- *bigcrypt* - BigCrypt
- *crypt16* - Crypt16
- *md5_crypt* - MD5 Crypt
- *bcrypt* - BCrypt
- *sha1_crypt* - SHA-1 Crypt
- *sun_md5_crypt* - Sun MD5 Crypt
- *sha256_crypt* - SHA-256 Crypt
- *sha512_crypt* - SHA-512 Crypt
- *apr_md5_crypt* - Apache's MD5-Crypt variant
- *phpass* - PHPass' Portable Hash
- *pbkdf2_digest* - Generic PBKDF2 Hashes
- *cta_pbkdf2_sha1* - Cryptacular's PBKDF2 hash
- *dlitz_pbkdf2_sha1* - Dwayne Litzenberger's PBKDF2 hash
- *scram* - SCRAM Hash
- *bsd_nthash* - FreeBSD's MCF-compatible nthash encoding

The only parameters accepted are 'salt' or 'salt_size'. You can use your own salt by defining 'salt', or have one generated automatically using 'salt_size'. By default Ansible generates a salt of size 8.

New in version 2.7.

If you do not have Passlib installed, Ansible uses the `crypt` (<https://docs.python.org/3/library/crypt.html>) library as a fallback. Ansible supports at most four crypt schemes, depending on your platform at most the following crypt schemes are supported:

- `bcrypt` - BCrypt
- `md5_crypt` - MD5 Crypt
- `sha256_crypt` - SHA-256 Crypt
- `sha512_crypt` - SHA-512 Crypt

New in version 2.8.

Allowing special characters in `vars_prompt` values

Some special characters, such as `{` and `%` can create templating errors. If you need to accept special characters, use the `unsafe` option:

```
vars_prompt:
- name: my_password_with_weird_chars
  prompt: Enter password
  unsafe: yes
  private: yes
```

❗ See also

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[Conditionals \(playbooks_conditionals.html#playbooks-conditionals\)](#)

Conditional statements in playbooks

[Using Variables \(playbooks_variables.html#playbooks-variables\)](#)

All about variables

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

Search this site

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Module defaults

If you frequently call the same module with the same arguments, it can be useful to define default arguments for that particular module using the `module_defaults` keyword.

Here is a basic example:

```
- hosts: localhost
  module_defaults:
    ansible.builtin.file:
      owner: root
      group: root
      mode: 0755
  tasks:
    - name: Create file1
      ansible.builtin.file:
        state: touch
        path: /tmp/file1

    - name: Create file2
      ansible.builtin.file:
        state: touch
        path: /tmp/file2

    - name: Create file3
      ansible.builtin.file:
        state: touch
        path: /tmp/file3
```

The `module_defaults` keyword can be used at the play, block, and task level. Any module arguments explicitly specified in a task will override any established default for that module argument.

```
- block:
  - name: Print a message
    ansible.builtin.debug:
      msg: "Different message"
  module_defaults:
    ansible.builtin.debug:
      msg: "Default message"
```

You can remove any previously established defaults for a module by specifying an empty dict.

```
- name: Create file1
  ansible.builtin.file:
    state: touch
    path: /tmp/file1
  module_defaults:
    file: {}
```

❗ Note

Any module defaults set at the play level (and block/task level when using `include_role` or `import_role`) will apply to any roles used, which may cause unexpected behavior in the role.

Here are some more realistic use cases for this feature.

Interacting with an API that requires auth.

```
- hosts: localhost
  module_defaults:
    ansible.builtin.uri:
      force_basic_auth: true
      user: some_user
      password: some_password
  tasks:
    - name: Interact with a web service
      ansible.builtin.uri:
        url: http://some.api.host/v1/whatever1

    - name: Interact with a web service
      ansible.builtin.uri:
        url: http://some.api.host/v1/whatever2

    - name: Interact with a web service
      ansible.builtin.uri:
        url: http://some.api.host/v1/whatever3
```

Setting a default AWS region for specific EC2-related modules.

```
- hosts: localhost
  vars:
    my_region: us-west-2
  module_defaults:
    amazon.aws.ec2:
      region: '{{ my_region }}'
    community.aws.ec2_instance_info:
      region: '{{ my_region }}'
    amazon.aws.ec2_vpc_net_info:
      region: '{{ my_region }}'
```

Module defaults groups

New in version 2.7.

Ansible 2.7 adds a preview-status feature to group together modules that share common sets of parameters. This makes it easier to author playbooks making heavy use of API-based modules such as cloud modules.

Group	Purpose	Ansible Version
aws	Amazon Web Services	2.7
azure	Azure	2.7
gcp	Google Cloud Platform	2.7
k8s	Kubernetes	2.8
os	OpenStack	2.8
acme	ACME	2.10
docker*	Docker	2.10
ovirt	oVirt	2.10
vmware	VMware	2.10

- The [docker_stack \(docker_stack module\)](#) module is not included in the `docker` defaults group.

Use the groups with `module_defaults` by prefixing the group name with `group/` - for example `group/aws`.

In a playbook, you can set module defaults for whole groups of modules, such as setting a common AWS region.

```
# example_play.yml
- hosts: localhost
  module_defaults:
    group/aws:
      region: us-west-2
  tasks:
    - name: Get info
      aws_s3_bucket_info:

# now the region is shared between both info modules

- name: Get info
  ec2_ami_info:
    filters:
      name: 'RHEL*7.5*'
```

In ansible-core 2.12, collections can define their own groups in the `meta/runtime.yml` file. `module_defaults` does not take the `collections` keyword into account, so the fully qualified group name must be used for new groups in `module_defaults`.

Here is an example `runtime.yml` file for a collection and a sample playbook using the group.

```
# collections/ansible_collections/ns/coll/meta/runtime.yml
action_groups:
  groupname:
    - module
    - another.collection.module
```

```
- hosts: localhost
  module_defaults:
    group/ns.coll.groupname:
      option_name: option_value
  tasks:
    - ns.coll.module:
    - another.collection.module
```

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Validating tasks: check mode and diff mode

Ansible provides two modes of execution that validate tasks: check mode and diff mode. These modes can be used separately or together. They are useful when you are creating or editing a playbook or role and you want to know what it will do. In check mode, Ansible runs without making any changes on remote systems. Modules that support check mode report the changes they would have made. Modules that do not support check mode report nothing and do nothing. In diff mode, Ansible provides before-and-after comparisons. Modules that support diff mode display detailed information. You can combine check mode and diff mode for detailed validation of your playbook or role.

- [Using check mode](#)
 - [Enforcing or preventing check mode on tasks](#)
 - [Skipping tasks or ignoring errors in check mode](#)
- [Using diff mode](#)
 - [Enforcing or preventing diff mode on tasks](#)

Using check mode

Check mode is just a simulation. It will not generate output for tasks that use [conditionals based on registered variables](#) ([playbooks_conditionals.html#conditionals-registered-vars](#)) (results of prior tasks). However, it is great for validating configuration management playbooks that run on one node at a time. To run a playbook in check mode:

```
ansible-playbook foo.yml --check
```

Enforcing or preventing check mode on tasks

New in version 2.2.

If you want certain tasks to run in check mode always, or never, regardless of whether you run the playbook with or without `--check`, you can add the `check_mode` option to those tasks:

- To force a task to run in check mode, even when the playbook is called without `--check`, set `check_mode: yes`.
- To force a task to run in normal mode and make changes to the system, even when the playbook is called with `--check`, set `check_mode: no`.

For example:

```
tasks:
- name: This task will always make changes to the system
  ansible.builtin.command: /something/to/run --even-in-check-mode
  check_mode: no

- name: This task will never make changes to the system
  ansible.builtin.lineinfile:
    line: "important config"
    dest: /path/to/myconfig.conf
    state: present
  check_mode: yes
  register: changes_to_important_config
```

Running single tasks with `check_mode: yes` can be useful for testing Ansible modules, either to test the module itself or to test the conditions under which a module would make changes. You can register variables (see [Conditionals \(playbooks conditionals.html#playbooks-conditionals\)](#)) on these tasks for even more detail on the potential changes.

Note

Prior to version 2.2 only the equivalent of `check_mode: no` existed. The notation for that was `always_run: yes`.

Skipping tasks or ignoring errors in check mode

New in version 2.1.

If you want to skip a task or ignore errors on a task when you run Ansible in check mode, you can use a boolean magic variable `ansible_check_mode`, which is set to `True` when Ansible runs in check mode. For example:

tasks:

- **name:** This task will be skipped in check mode
ansible.builtin.git:
 repo: ssh://git@github.com/mylogin/hello.git
 dest: /home/mylogin/hello
 when: not ansible_check_mode
- **name:** This task will ignore errors in check mode
ansible.builtin.git:
 repo: ssh://git@github.com/mylogin/hello.git
 dest: /home/mylogin/hello
 ignore_errors: "{{ ansible_check_mode }}"

Using diff mode

The `--diff` option for `ansible-playbook` can be used alone or with `--check`. When you run in diff mode, any module that supports diff mode reports the changes made or, if used with `--check`, the changes that would have been made. Diff mode is most common in modules that manipulate files (for example, the `template` module) but other modules might also show 'before and after' information (for example, the `user` module).

Diff mode produces a large amount of output, so it is best used when checking a single host at a time. For example:

```
ansible-playbook foo.yml --check --diff --limit foo.example.com
```

New in version 2.4.

Enforcing or preventing diff mode on tasks

Because the `--diff` option can reveal sensitive information, you can disable it for a task by specifying `diff: no`. For example:

tasks:

- **name:** This task will not report a diff when the file changes
ansible.builtin.template:
 src: secret.conf.j2
 dest: /etc/secret.conf
 owner: root
 group: root
 mode: '0600'
 diff: no

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Executing playbooks for troubleshooting

When you are testing new plays or debugging playbooks, you may need to run the same play multiple times. To make this more efficient, Ansible offers two alternative ways to execute a playbook: start-at-task and step mode.

start-at-task

To start executing your playbook at a particular task (usually the task that failed on the previous run), use the `--start-at-task` option.

```
ansible-playbook playbook.yml --start-at-task="install packages"
```

In this example, Ansible starts executing your playbook at a task named “install packages”. This feature does not work with tasks inside dynamically re-used roles or tasks (`include_*`), see [Comparing includes and imports: dynamic and static re-use \(playbooks_reuse.html#dynamic-vs-static\)](#).

Step mode

To execute a playbook interactively, use `--step`.

```
ansible-playbook playbook.yml --step
```

With this option, Ansible stops on each task, and asks if it should execute that task. For example, if you have a task called “configure ssh”, the playbook run will stop and ask.

```
Perform task: configure ssh (y/n/c):
```

Answer “y” to execute the task, answer “n” to skip the task, and answer “c” to exit step mode, executing all remaining tasks without asking.

❗ See also

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[Debugging tasks \(playbooks_debugger.html#playbook-debugger\)](#)

Using the Ansible debugger

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Debugging tasks

Ansible offers a task debugger so you can fix errors during execution instead of editing your playbook and running it again to see if your change worked. You have access to all of the features of the debugger in the context of the task. You can check or set the value of variables, update module arguments, and re-run the task with the new variables and arguments. The debugger lets you resolve the cause of the failure and continue with playbook execution.

- [Enabling the debugger](#)
 - [Enabling the debugger with the `debugger` keyword](#)
 - [Examples of using the `debugger` keyword](#)
 - [Enabling the debugger in configuration or an environment variable](#)
 - [Enabling the debugger as a strategy](#)
- [Resolving errors in the debugger](#)
- [Available debug commands](#)
 - [Print command](#)
 - [Update args command](#)
 - [Update vars command](#)
 - [Update task command](#)
 - [Redo command](#)
 - [Continue command](#)
 - [Quit command](#)
- [How the debugger interacts with the free strategy](#)

Enabling the debugger

The debugger is not enabled by default. If you want to invoke the debugger during playbook execution, you must enable it first.

Use one of these three methods to enable the debugger:

- with the debugger keyword
- in configuration or an environment variable, or
- as a strategy

Enabling the debugger with the `debugger` keyword

New in version 2.5.

You can use the `debugger` keyword to enable (or disable) the debugger for a specific play, role, block, or task. This option is especially useful when developing or extending playbooks, plays, and roles. You can enable the debugger on new or updated tasks. If they fail, you can fix the errors efficiently. The `debugger` keyword accepts five values:

Value	Result
<code>always</code>	Always invoke the debugger, regardless of the outcome
<code>never</code>	Never invoke the debugger, regardless of the outcome
<code>on_failed</code>	Only invoke the debugger if a task fails
<code>on_unreachable</code>	Only invoke the debugger if a host is unreachable
<code>on_skipped</code>	Only invoke the debugger if the task is skipped

When you use the `debugger` keyword, the value you specify overrides any global configuration to enable or disable the debugger. If you define `debugger` at multiple levels, such as in a role and in a task, Ansible honors the most granular definition. The definition at the play or role level applies to all blocks and tasks within that play or role, unless they specify a different value. The definition at the block level overrides the definition at the play or role level, and applies to all tasks within that block, unless they specify a different value. The definition at the task level always applies to the task; it overrides the definitions at the block, play, or role level.

Examples of using the `debugger` keyword

Example of setting the `debugger` keyword on a task:

```
- name: Execute a command
  ansible.builtin.command: "false"
  debugger: on_failed
```

Example of setting the `debugger` keyword on a play:

```
- name: My play
  hosts: all
  debugger: on_skipped
  tasks:
    - name: Execute a command
      ansible.builtin.command: "true"
      when: False
```

Example of setting the `debugger` keyword at multiple levels:

```
- name: Play
  hosts: all
  debugger: never
  tasks:
    - name: Execute a command
      ansible.builtin.command: "false"
      debugger: on_failed
```

In this example, the debugger is set to `never` at the play level and to `on_failed` at the task level. If the task fails, Ansible invokes the debugger, because the definition on the task overrides the definition on its parent play.

Enabling the debugger in configuration or an environment variable

New in version 2.5.

You can enable the task debugger globally with a setting in `ansible.cfg` or with an environment variable. The only options are `True` or `False`. If you set the configuration option or environment variable to `True`, Ansible runs the debugger on failed tasks by default.

To enable the task debugger from `ansible.cfg`, add this setting to the defaults section:

```
[defaults]
enable_task_debugger = True
```

To enable the task debugger with an environment variable, pass the variable when you run your playbook:

```
ANSIBLE_ENABLE_TASK_DEBUGGER=True ansible-playbook -i hosts site.yml
```

When you enable the debugger globally, every failed task invokes the debugger, unless the role, play, block, or task explicitly disables the debugger. If you need more granular control over what conditions trigger the debugger, use the `debugger` keyword.

Enabling the debugger as a strategy

If you are running legacy playbooks or roles, you may see the debugger enabled as a strategy ([../plugins/strategy.html#strategy-plugins](https://plugins/strategy.html#strategy-plugins)). You can do this at the play level, in `ansible.cfg`, or with the environment variable `ANSIBLE_STRATEGY=debug`. For example:

```
- hosts: test
  strategy: debug
  tasks:
  ...
```

Or in `ansible.cfg`:

```
[defaults]
strategy = debug
```

❗ Note

This backwards-compatible method, which matches Ansible versions before 2.5, may be removed in a future release.

Resolving errors in the debugger

After Ansible invokes the debugger, you can use the seven debugger commands to resolve the error that Ansible encountered. Consider this example playbook, which defines the `var1` variable but uses the undefined `wrong_var` variable in a task by mistake.

```
- hosts: test
  debugger: on_failed
  gather_facts: no
  vars:
    var1: value1
  tasks:
    - name: Use a wrong variable
      ansible.builtin.ping: data={{ wrong_var }}
```

If you run this playbook, Ansible invokes the debugger when the task fails. From the debug prompt, you can change the module arguments or the variables and run the task again.

Search this site


```

PLAY *****

TASK [wrong variable] *****
fatal: [192.0.2.10]: FAILED! => {"failed": true, "msg": "ERROR! 'wrong_var' is
undefined"}
Debugger invoked
[192.0.2.10] TASK: wrong variable (debug)> p result._result
{'failed': True,
 'msg': 'The task includes an option with an undefined variable. The error '
       'was: 'wrong_var' is undefined\n'
       '\n'
       'The error appears to have been in '
       "'playbooks/debugger.yml': line 7, "
       'column 7, but may\n'
       'be elsewhere in the file depending on the exact syntax problem.\n'
       '\n'
       'The offending line appears to be:\n'
       '\n'
       '   tasks:\n'
       '     - name: wrong variable\n'
       '       ^ here\n'}
[192.0.2.10] TASK: wrong variable (debug)> p task.args
{'u'data': u'{{ wrong_var }}'}
[192.0.2.10] TASK: wrong variable (debug)> task.args['data'] = '{{ var1 }}'
[192.0.2.10] TASK: wrong variable (debug)> p task.args
{'u'data': '{{ var1 }}'}
[192.0.2.10] TASK: wrong variable (debug)> redo
ok: [192.0.2.10]

PLAY RECAP *****
192.0.2.10          : ok=1    changed=0    unreachable=0    failed=0

```

Changing the task arguments in the debugger to use `var1` instead of `wrong_var` makes the task run successfully.

Available debug commands

You can use these seven commands at the debug prompt:

Command	Shortcut	Action
<code>print</code>	<code>p</code>	Print information about the task
<code>task.args[key] = value</code>	no shortcut	Update module arguments
<code>task_vars[key] = value</code>	no shortcut	Update task variables (you must <code>update_task</code> next)
<code>update_task</code>	<code>u</code>	Recreate a task with updated task variables
<code>redo</code>	<code>r</code>	Run the task again
<code>continue</code>	<code>c</code>	Continue executing, starting with the next task
<code>quit</code>	<code>q</code>	Quit the debugger

For more details, see the individual descriptions and examples below.

Print command

`print *task/task.args/task_vars/host/result*` prints information about the task.

```
[192.0.2.10] TASK: install package (debug)> p task
TASK: install package
[192.0.2.10] TASK: install package (debug)> p task.args
{'name': u'{{ pkg_name }}'}
[192.0.2.10] TASK: install package (debug)> p task_vars
{'ansible_all_ipv4_addresses': [u'192.0.2.10'],
 u'ansible_architecture': u'x86_64',
 ...
}
[192.0.2.10] TASK: install package (debug)> p task_vars['pkg_name']
u'bash'
[192.0.2.10] TASK: install package (debug)> p host
192.0.2.10
[192.0.2.10] TASK: install package (debug)> p result._result
{'_ansible_no_log': False,
 'changed': False,
 u'failed': True,
 ...
 u'msg': u"No package matching 'not_exist' is available"}
```

Update args command

`task.args[*key] = *value*` updates a module argument. This sample playbook has an invalid package name.

```
- hosts: test
  strategy: debug
  gather_facts: yes
  vars:
    pkg_name: not_exist
  tasks:
    - name: Install a package
      ansible.builtin.apt: name={{ pkg_name }}
```

When you run the playbook, the invalid package name triggers an error, and Ansible invokes the debugger. You can fix the package name by viewing, then updating the module argument.

```
[192.0.2.10] TASK: install package (debug)> p task.args
{'name': u'{{ pkg_name }}'}
[192.0.2.10] TASK: install package (debug)> task.args['name'] = 'bash'
[192.0.2.10] TASK: install package (debug)> p task.args
{'name': 'bash'}
[192.0.2.10] TASK: install package (debug)> redo
```

After you update the module argument, use `redo` to run the task again with the new args.

Update vars command

`task_vars[*key] = *value*` updates the `task_vars`. You could fix the playbook above by viewing, then updating the task variables instead of the module args.

```
[192.0.2.10] TASK: install package (debug)> p task_vars['pkg_name']
u'not_exist'
[192.0.2.10] TASK: install package (debug)> task_vars['pkg_name'] = 'bash'
[192.0.2.10] TASK: install package (debug)> p task_vars['pkg_name']
'bash'
[192.0.2.10] TASK: install package (debug)> update_task
[192.0.2.10] TASK: install package (debug)> redo
```

After you update the task variables, you must use `update_task` to load the new variables before using `redo` to run the task again.

! Note

In 2.5 this was updated from `vars` to `task_vars` to avoid conflicts with the `vars()` python function.

Update task command

New in version 2.8.

`u` or `update_task` recreates the task from the original task data structure and templates with updated task variables. See the entry [Update vars command](#) for an example of use.

Redo command

`r` or `redo` runs the task again.

Continue command

`c` or `continue` continues executing, starting with the next task.

Quit command

`q` or `quit` quits the debugger. The playbook execution is aborted.

How the debugger interacts with the free strategy.

Search this site

With the default `linear` strategy enabled, Ansible halts execution while the debugger is active, and runs the debugged task immediately after you enter the `redo` command. With the `free` strategy enabled, however, Ansible does not wait for all hosts, and may queue later tasks on one host before a task fails on another host. With the `free` strategy, Ansible does not queue or execute any tasks while the debugger is active. However, all queued tasks remain in the queue and run as soon as you exit the debugger. If you use `redo` to reschedule a task from the debugger, other queued tasks may execute before your rescheduled task. For more information about strategies, see [Controlling playbook execution: strategies and more \(playbooks_strategies.html#playbooks-strategies\)](#).

❗ See also

[Executing playbooks for troubleshooting \(playbooks_startnstep.html#playbooks-start-and-step\)](#)

Running playbooks while debugging or testing

[Intro to playbooks \(playbooks_intro.html#playbooks-intro\)](#)

An introduction to playbooks

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](#)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels

You are reading the latest community version of the Ansible documentation. Red Hat subscribers, select **2.9** in the version selection to the left for the most recent Red Hat release.

Controlling playbook execution: strategies and more

By default, Ansible runs each task on all hosts affected by a play before starting the next task on any host, using 5 forks. If you want to change this default behavior, you can use a different strategy plugin, change the number of forks, or apply one of several keywords like

`serial`.

- [Selecting a strategy](#)
- [Setting the number of forks](#)
- [Using keywords to control execution](#)
 - [Setting the batch size with](#) `serial`
 - [Restricting execution with](#) `throttle`
 - [Ordering execution based on inventory](#)
 - [Running on a single machine with](#) `run_once`

Selecting a strategy

The default behavior described above is the [linear strategy](#).

([../collections/ansible/builtin/linear_strategy.html#linear-strategy](#)). Ansible offers other strategies, including the [debug strategy](#).

([../collections/ansible/builtin/debug_strategy.html#debug-strategy](#)) (see also [Debugging tasks \(playbooks_debugger.html#playbook-debugger\)](#)) and the [free strategy](#).

([../collections/ansible/builtin/free_strategy.html#free-strategy](#)), which allows each host to run until the end of the play as fast as it can:

```
- hosts: all
  strategy: free
  tasks:
  # ...
```

You can select a different strategy for each play as shown above, or set your preferred strategy globally in `ansible.cfg`, under the `defaults` stanza:

```
[defaults]
strategy = free
```

All strategies are implemented as [strategy plugins \(../plugins/strategy.html#strategy-plugins\)](#). Please review the documentation for each strategy plugin for details on how it works.

Setting the number of forks

If you have the processing power available and want to use more forks, you can set the number in `ansible.cfg`:

```
[defaults]
forks = 30
```

or pass it on the command line: `ansible-playbook -f 30 my_playbook.yml`.

Using keywords to control execution

In addition to strategies, several [keywords \(../reference_appendices/playbooks_keywords.html#playbook-keywords\)](#) also affect play execution. You can set a number, a percentage, or a list of numbers of hosts you want to manage at a time with `serial`. Ansible completes the play on the specified number or percentage of hosts before starting the next batch of hosts. You can restrict the number of workers allotted to a block or task with `throttle`. You can control how Ansible selects the next host in a group to execute against with `order`. You can run a task on a single host with `run_once`. These keywords are not strategies. They are directives or options applied to a play, block, or task.

Other keywords that affect play execution include `ignore_errors`, `ignore_unreachable`, and `any_errors_fatal`. These options are documented in [Error handling in playbooks \(playbooks_error_handling.html#playbooks-error-handling\)](#).

Setting the batch size with `serial`

By default, Ansible runs in parallel against all the hosts in the [pattern \(intro_patterns.html#intro-patterns\)](#) you set in the `hosts:` field of each play. If you want to manage only a few machines at a time, for example during a rolling update, you can define how many hosts Ansible should manage at a single time using the `serial` keyword. [Search this site](#)

```

---
- name: test play
  hosts: webserver
  serial: 3
  gather_facts: False

  tasks:
    - name: first task
      command: hostname
    - name: second task
      command: hostname

```

In the above example, if we had 6 hosts in the group 'webserver', Ansible would execute the play completely (both tasks) on 3 of the hosts before moving on to the next 3 hosts:

```

PLAY [webserver] *****

TASK [first task] *****
changed: [web3]
changed: [web2]
changed: [web1]

TASK [second task] *****
changed: [web1]
changed: [web2]
changed: [web3]

PLAY [webserver] *****

TASK [first task] *****
changed: [web4]
changed: [web5]
changed: [web6]

TASK [second task] *****
changed: [web4]
changed: [web5]
changed: [web6]

PLAY RECAP *****
web1      : ok=2    changed=2    unreachable=0    failed=0
web2      : ok=2    changed=2    unreachable=0    failed=0
web3      : ok=2    changed=2    unreachable=0    failed=0
web4      : ok=2    changed=2    unreachable=0    failed=0
web5      : ok=2    changed=2    unreachable=0    failed=0
web6      : ok=2    changed=2    unreachable=0    failed=0

```

You can also specify a percentage with the `serial` keyword. Ansible applies the percentage to the total number of hosts in a play to determine the number of hosts per pass:

```

---
- name: test play
  hosts: webserver
  serial: "30%"

```

If the number of hosts does not divide equally into the number of passes, the final pass contains the remainder. In this example, if you had 20 hosts in the webserver group, the first batch would contain 6 hosts, the second batch would contain 6 hosts, the third batch would contain 6 hosts, and the last batch would contain 2 hosts.

You can also specify batch sizes as a list. For example:

```
---
- name: test play
  hosts: webserver
  serial:
    - 1
    - 5
    - 10
```

In the above example, the first batch would contain a single host, the next would contain 5 hosts, and (if there are any hosts left), every following batch would contain either 10 hosts or all the remaining hosts, if fewer than 10 hosts remained.

You can list multiple batch sizes as percentages:

```
---
- name: test play
  hosts: webserver
  serial:
    - "10%"
    - "20%"
    - "100%"
```

You can also mix and match the values:

```
---
- name: test play
  hosts: webserver
  serial:
    - 1
    - 5
    - "20%"
```

Note

No matter how small the percentage, the number of hosts per pass will always be 1 or greater.

Restricting execution with `throttle`

The `throttle` keyword limits the number of workers for a particular task. It can be set at the block and task level. Use `throttle` to restrict tasks that may be CPU-intensive or interact with a rate-limiting API:

```
tasks:
- command: /path/to/cpu_intensive_command
  throttle: 1
```

If you have already restricted the number of forks or the number of machines to execute against in parallel, you can reduce the number of workers with `throttle`, but you cannot increase it. In other words, to have an effect, your `throttle` setting must be lower than your `forks` or `serial` setting if you are using them together.

Ordering execution based on inventory

The `order` keyword controls the order in which hosts are run. Possible values for order are:

inventory:

(default) The order provided by the inventory for the selection requested (see note below)

reverse_inventory:

The same as above, but reversing the returned list

sorted:

Sorted alphabetically sorted by name

reverse_sorted:

Sorted by name in reverse alphabetical order

shuffle:

Randomly ordered on each run

❗ Note

the 'inventory' order does not equate to the order in which hosts/groups are defined in the inventory source file, but the 'order in which a selection is returned from the compiled inventory'. This is a backwards compatible option and while reproducible it is not normally predictable. Due to the nature of inventory, host patterns, limits, inventory plugins and the ability to allow multiple sources it is almost impossible to return such an order. For simple cases this might happen to match the file definition order, but that is not guaranteed.

Running on a single machine with `run_once`

If you want a task to run only on the first host in your batch of hosts, set `run_once` to true on that task:

```
---
# ...

tasks:

    # ...

    - command: /opt/application/upgrade_db.py
      run_once: true

    # ...
```

Ansible executes this task on the first host in the current batch and applies all results and facts to all the hosts in the same batch. This approach is similar to applying a conditional to a task such as:

```
- command: /opt/application/upgrade_db.py
  when: inventory_hostname == webservers[0]
```

However, with `run_once`, the results are applied to all the hosts. To run the task on a specific host, instead of the first host in the batch, delegate the task:

```
- command: /opt/application/upgrade_db.py
  run_once: true
  delegate_to: web01.example.org
```

As always with [delegation](#) ([playbooks_delegation.html#playbooks-delegation](#)), the action will be executed on the delegated host, but the information is still that of the original host in the task.

Note

When used together with `serial`, tasks marked as `run_once` will be run on one host in *each* serial batch. If the task must run only once regardless of `serial` mode, use `when: inventory_hostname == ansible_play_hosts_all[0]` construct.

Note

Any conditional (in other words, *when:*) will use the variables of the ‘first host’ to decide if the task runs or not, no other hosts will be tested.

❗ Note

If you want to avoid the default behavior of setting the fact for all hosts, set

```
delegate_facts: True
```

 for the specific task or block.

❗ See also

[Intro to playbooks \(playbooks intro.html#about-playbooks\)](#)

An introduction to playbooks

[Controlling where tasks run: delegation and local actions \(playbooks delegation.html#playbooks-delegation\)](#)

Running tasks on or assigning facts to specific machines

[Roles \(playbooks reuse roles.html#playbooks-reuse-roles\)](#)

Playbook organization by roles

[User Mailing List \(https://groups.google.com/group/ansible-devel\)](https://groups.google.com/group/ansible-devel)

Have a question? Stop by the google group!

[Real-time chat \(../community/communication.html#communication-irc\)](#)

How to join Ansible chat channels