Ansible Quickstart

Ansible concepts	Control node Managed nodes Inventory 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. Modules: 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.
	Introduction to ad hoc commands

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

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 keyword	`$ ansible atlanta -a "/sbin/reboot" -f 10 -u username`
Managing files	`$ ansible webservers -m ansible.builtin.file -a "dest=/path/to/c mode=755 owner=mdehaan group=mdehaan state=directory"`
Managing packages	`$ ansible webservers -m ansible.builtin.yum -a "name=acme state=present"`

Managing users and groups	`$ ansible all -m ansible.builtin.user -a "name=foo password=<crypted password here>"`
	`$ ansible all -m ansible.builtin.user -a "name=foo state=absent"`

Managing services	`$ ansible webservers -m ansible.builtin.service -a "name=httpd state=started"`
Working with playbooks	

Templating (Jinja2)
 	
Default
Zip
random
shuffle
random
Min,max
unique
Union,intersect,difference
Join,split
`match search`
`mypath is directory`
`mypath is file`
`mypath is exists`	

Get the current time	The now() Jinja2 function retrieves a Python datetime object or a string representation for the current time. The now() function supports 2 arguments: utc Specify True to get the current time in UTC. Defaults to False. fmt Accepts a strftime string that returns a formatted date time string.
Providing default values	`{{ some_variable | default(5) }}`

Making variables optional	`- name: Touch files with an optional mode`
	`  ansible.builtin.file:`
	`    dest: "{{ item.path }}"`
	`    state: touch`
	`    mode: "{{ item.mode | default(omit) }}"`
	`  loop:`
	`    - path: /tmp/foo`
	`    - path: /tmp/bar`
	`    - path: /tmp/baz`
	`      mode: "0444"`
	
	In this example, the default mode for the files `/tmp/foo` and `/tmp/bar` is determined by the umask of the system. Ansible does not send a

	value for `mode`. Only the third file, `/tmp/baz`, receives the mode=0444 option.
Defining different values for true/false/null (ternary)	`{{ (status == 'needs_restart') | ternary('restart', 'continue') }}`
Discovering the data type	`{{ myvar | type_debug }}`
Transforming dictionaries into lists	`{{ dict | dict2items }}`

Combining items from multiple lists: zip and zip_longest	`- name: Give me list combo of two lists`
	`  ansible.builtin.debug:`
	`   msg: "{{ [1,2,3,4,5,6] | zip(['a','b','c','d','e','f']) | list }}"`
	
	`# => [[1, "a"], [2, "b"], [3, "c"], [4, "d"], [5, "e"], [6, "f"]]`
	
	`- name: Give me shortest combo of two lists`
	`  ansible.builtin.debug:`
	`    msg: "{{ [1,2,3] | zip(['a','b','c','d','e','f']) | list }}"`
	
	`# => [[1, "a"], [2, "b"], [3, "c"]]`

Random items or numbers	**To get a random item from a list:**
	`"{{ ['a','b','c'] | random }}"`
	`# => 'c'`
	
	To get a random number between 0 (inclusive) and a specified integer (exclusive):
	`"{{ 60 | random }} * * * * root /script/from/cron"`
	`# => '21 * * * * root /script/from/cron'`

Shuffling a list	`{{ ['a','b','c'] | shuffle }}`
	`# => ['c','a','b']`
	`{{ ['a','b','c'] | shuffle }}`
	`# => ['b','c','a']`

Managing list variables	`To get the minimum value from list of numbers:`
	`{{ list1 | min }}`
	
	To get the minimum value in a list of objects:
	
	`{{ [{'val': 1}, {'val': 2}] | min(attribute='val') }}`

	To get the maximum value from a list of numbers: `{{ [3, 4, 2] | max }}`

Selecting from sets or lists (set theory)	**To get a unique set from a list:**
	`# list1: [1, 2, 5, 1, 3, 4, 10]`
	`{{ list1 | unique }}`
	`# => [1, 2, 5, 3, 4, 10]`
	
	To get a union of two lists:
	`# list1: [1, 2, 5, 1, 3, 4, 10]`
	`# list2: [1, 2, 3, 4, 5, 11, 99]`
	`{{ list1 | union(list2) }}`
	`# => [1, 2, 5, 1, 3, 4, 10, 11, 99]`
	
	To get the intersection of 2 lists (unique list of all items in both):
	`# list1: [1, 2, 5, 3, 4, 10]`
	`# list2: [1, 2, 3, 4, 5, 11, 99]`
	`{{ list1 | intersect(list2) }}`
	`# => [1, 2, 5, 3, 4]`
	
	To get the difference of 2 lists (items in 1 that don’t exist in 2):
	`# list1: [1, 2, 5, 1, 3, 4, 10]`
	`# list2: [1, 2, 3, 4, 5, 11, 99]`
	`{{ list1 | difference(list2) }}`
	`# => [10]`

IP address filters	**To test if a string is a valid IP address:** `{{ myvar | ansible.netcommon.ipaddr }}`

Manipulating strings

To concatenate a list into a string:

```
{{ list | join(" ") }}
```

To split a sting into a list:

```
{{ csv_string | split(",") }}
```

Tests

Tests in Jinja are a way of evaluating template expressions and returning True or False. Jinja ships with many of these. See builtin tests in the official Jinja template documentation.

The syntax for using a jinja test is as follows:

```
variable is test_name
E.g.
result is failed
```

Testing strings

```
vars:
  url: "http://example.com/users/foo/resources/bar"

tasks:
    - debug:
        msg: "matched pattern 1"
      when: url is match("http://example.com/users/.*/resources/")

    - debug:
        msg: "matched pattern 2"
      when: url is search("/users/.*/resources/.*")

    - debug:
        msg: "matched pattern 3"
      when: url is search("/users/")

    - debug:
        msg: "matched pattern 4"
      when: url is regex("example.com/\w+/foo")
```

`match` succeeds if it finds the pattern at the beginning of the string, while `search` succeeds if it finds the pattern anywhere within string.

Comparing versions

To compare a version number, such as checking if the `ansible_facts['distribution_version']` version is greater than or equal to ‘12.04’, you can use the `version` test.

	The `version` test can also be used to evaluate the `ansible_facts['distribution_version']`: `{{ ansible_facts['distribution_version'] is version('12.04', '>=') }}` If `ansible_facts['distribution_version']` is greater than or equal to 12.04, this test returns True, otherwise False.

Testing paths	`- debug:`
	`    msg: "path is a directory"`
	`  when: mypath is directory`
	
	`- debug:`
	`    msg: "path is a file"`
	`  when: mypath is file`
	
	`- debug:`
	`    msg: "path is a symlink"`
	`  when: mypath is link`
	
	`- debug:`
	`    msg: "path already exists"`
	`  when: mypath is exists`
	
	`- debug:`
	`    msg: "path is {{ (mypath is abs)|ternary('absolute','relative')}}"`
	
	`- debug:`
	`    msg: "path is the same file as path2"`
	`  when: mypath is same_file(path2)`

Lookups	`vars:`
	`  motd_value: "{{ lookup('file', '/etc/motd') }}"`
	`tasks:`
	`  - debug:`
	`      msg: "motd value is {{ motd_value }}"`

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.

Playbook execution	At a minimum, each play defines two things: • the managed nodes to target, using a pattern • at least one task to execute **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 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.
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

The default behavior described above is the linear strategy. Ansible offers other strategies, including the debug strategy (see also Debugging tasks) and the free strategy, which allows each host to run until the end of the play as fast as it can:

```yaml
- 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:

```ini
[defaults]
strategy = free
```

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`:

```ini
[defaults]
forks = 30
```

or pass it on the command line: ansible-playbook -f 30 my_playbook.yml.

Setting the batch size with `serial`

By default, Ansible runs in parallel against all the hosts in the pattern 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:
	
	`---`
	`- name: test play`
	`  hosts: webservers`
	`  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 'webservers', Ansible would execute the play completely (both tasks) on 3 of the hosts before moving on to the next 3 hosts:
	
	https://medium.com/devops-srilanka/difference-between-forks-and-serial-in-ansible-48677ebe3f36

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`

Running playbooks	To run your playbook, use the ansible-playbook command: `ansible-playbook playbook.yml -f 10`
Verifying playbooks	The ansible-playbook command offers several options for verification, including `--check`, `--diff`, `--list-hosts`, `--list-tasks`, and `--syntax-check`.

ansible-lint	`$ 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`

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.
	
	`- 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`

OS-specific variables	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'] }}.yml"`
	`    - debug:`
	`        var: asdf`
	
	This pulls in variables from the group_vars/os_CentOS.yml file.

Understanding privilege escalation: 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`.
	To run a command as the `apache` user:
	`- name: Run a command as the apache user`
	`  command: somecommand`
	`  become: yes`
	`  become_user: apache`

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

Loops, Conditionals , Tests, Blocks and Handlers

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`

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' }`

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 }}"`

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 }}"`

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 keyword to control the number of hosts executing at one time:
	
	`---`
	`- hosts: webservers`
	`  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 }}
	```

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'] }}"`

Local playbooks	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`

Basic conditionals

Basic conditionals with	`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	`tasks:`

```yaml
  - 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:

```yaml
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 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:

```yaml
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"
```

Conditions based on registered variables

```yaml
- 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()`.

```yaml
- 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() }}"
```

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 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
```

Using conditionals in loops

```
tasks:
    - name: Run with items greater than 5
      ansible.builtin.command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
```

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. Ansible applies the condition to every task, and evaluates each task separately.

```
# 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
```

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`

Conditionals with roles	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

Selecting variables files based on facts	`---`
	`- 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`

Commonly-used facts	**ansible_facts['distribution']**
	Possible values (sample, not complete list):
	
	`Alpine`
	`CentOS`

	`Debian`
	`Fedora`
	`Ubuntu`
	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`

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
 	
Block
 	
Rescue
 	
Always
	`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.

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.

```
- 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.

Handler

Handler example

```
- 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.

	If you need handlers to run before the end of the play, add a task to flush them using the 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: ...
	```

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"
	```

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	`- name: Do not count this as a failure`
	`  ansible.builtin.command: /bin/false`
	`  ignore_errors: yes`

Handlers and failure	Ansible runs 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.

Defining failure
`failed_when`
	Ansible lets you define what “failure” means in each task using the `failed_when` conditional.
	
	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`

Defining “changed” `changed_when`	Ansible lets you define when a particular task has “changed” a remote node using the `changed_when` conditional. `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`

Aborting a play on all hosts
`Any_errors_fatal`
`max_fail_percentage`
	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 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
	```

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
	```

Roles, inventory, variables and Vault

Creating reusable 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 for details about creating and using roles.

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`

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
  - include_tasks
  - include_vars
- For static re-use, add an `import_*` task in the tasks section of a play:
  - import_role
  - Import_tasks

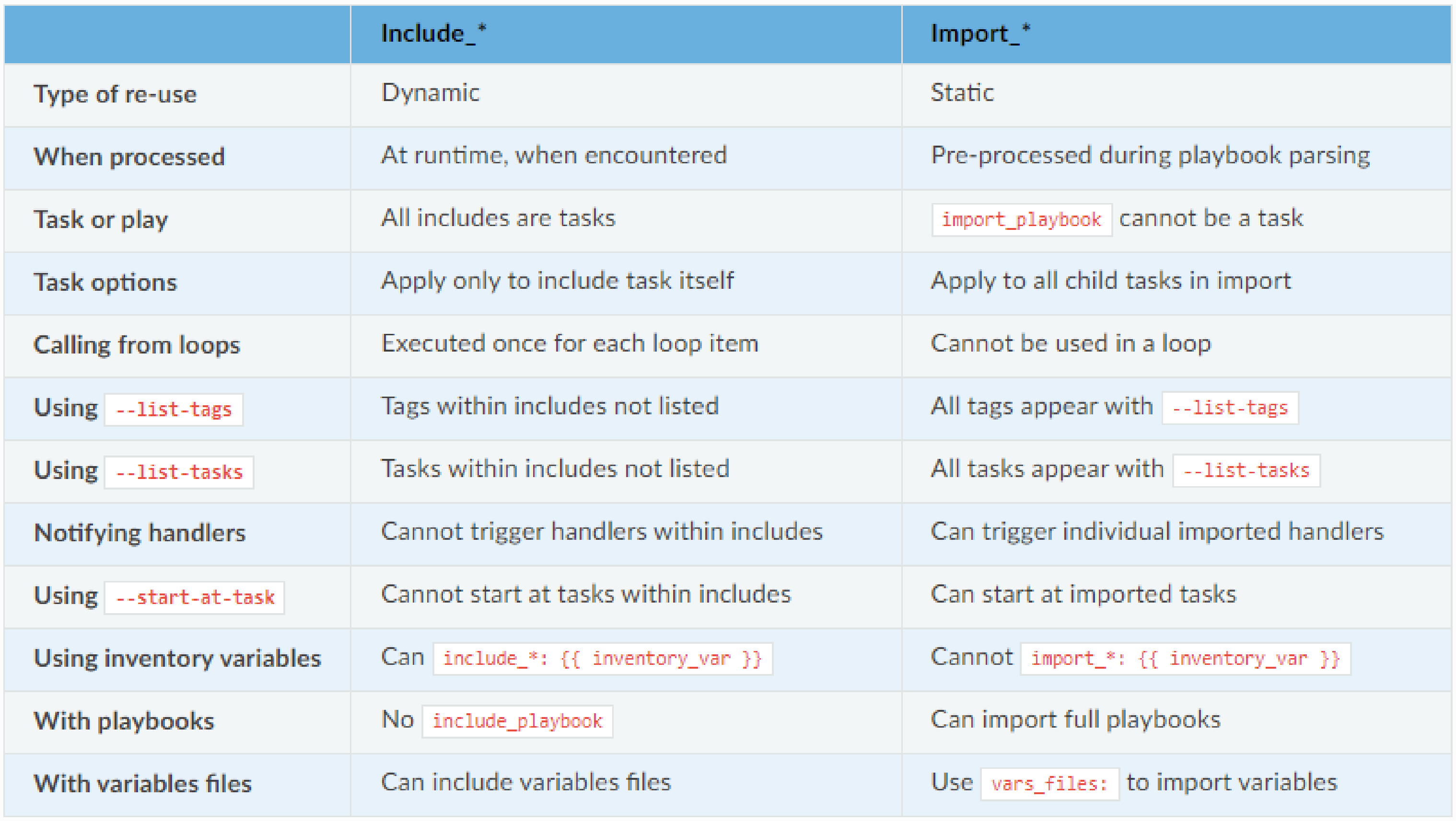
You can still use the bare roles keyword at the play level to incorporate a role in a playbook statically. However, the bare include keyword, once used for both task files and playbook-level includes, is now deprecated.

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.

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.



	Include_*	Import_*
Type of re-use	Dynamic	Static
When processed	At runtime, when encountered	Pre-processed during playbook parsing
Task or play	All includes are tasks	`import_playbook` cannot be a task
Task options	Apply only to include task itself	Apply to all child tasks in import
Calling from loops	Executed once for each loop item	Cannot be used in a loop
Using `--list-tags`	Tags within includes not listed	All tags appear with `--list-tags`
Using `--list-tasks`	Tasks within includes not listed	All tasks appear with `--list-tasks`
Notifying handlers	Cannot trigger handlers within includes	Can trigger individual imported handlers
Using `--start-at-task`	Cannot start at tasks within includes	Can start at imported tasks
Using inventory variables	Can `include_*: {{ inventory_var }}`	Cannot `import_*: {{ inventory_var }}`
With playbooks	No `include_playbook`	Can import full playbooks
With variables files	Can include variables files	Use `vars_files:` to import variables

You can pass variables to imports. You must pass variables if you want to run an

```
tasks:
- import_tasks: wordpress.yml
  vars:
    wp_user: timmy
```

imported file more than once in a playbook. For example:

```
- import_tasks: wordpress.yml
  vars:
    wp_user: alice

- import_tasks: wordpress.yml
  vars:
    wp_user: bob
```

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

```
# 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 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 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:

```yaml
# 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
```

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

```yaml
---
- hosts: webservers
  roles:
    - common
    - webservers
```

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.

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:

```yaml
---
- 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 conditionally include a role:

```yaml
---
```

	```
	- 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.
	```
	---
	- 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
	  ...
	```

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.

Adding tags to individual tasks

```yaml
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
```

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 and define the tags at that level. For example, we could edit the NTP example shown above to use a block:

```yaml
# myrole/tasks/main.yml
tasks:
- 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
```

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`).

```yaml
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
```

Selecting or skipping tags when you run a playbook

For example, to run only tasks and blocks tagged `configuration` and `packages` in a very long playbook:

```
ansible-playbook example.yml --tags "configuration,packages"
```

To run all tasks except those tagged `packages`:

```
ansible-playbook example.yml --skip-tags "packages"
```

How to build your inventory

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. Introduced in version 2.4, Ansible has Inventory Plugins to make this flexible and customizable.

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`

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).

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`

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 for details.

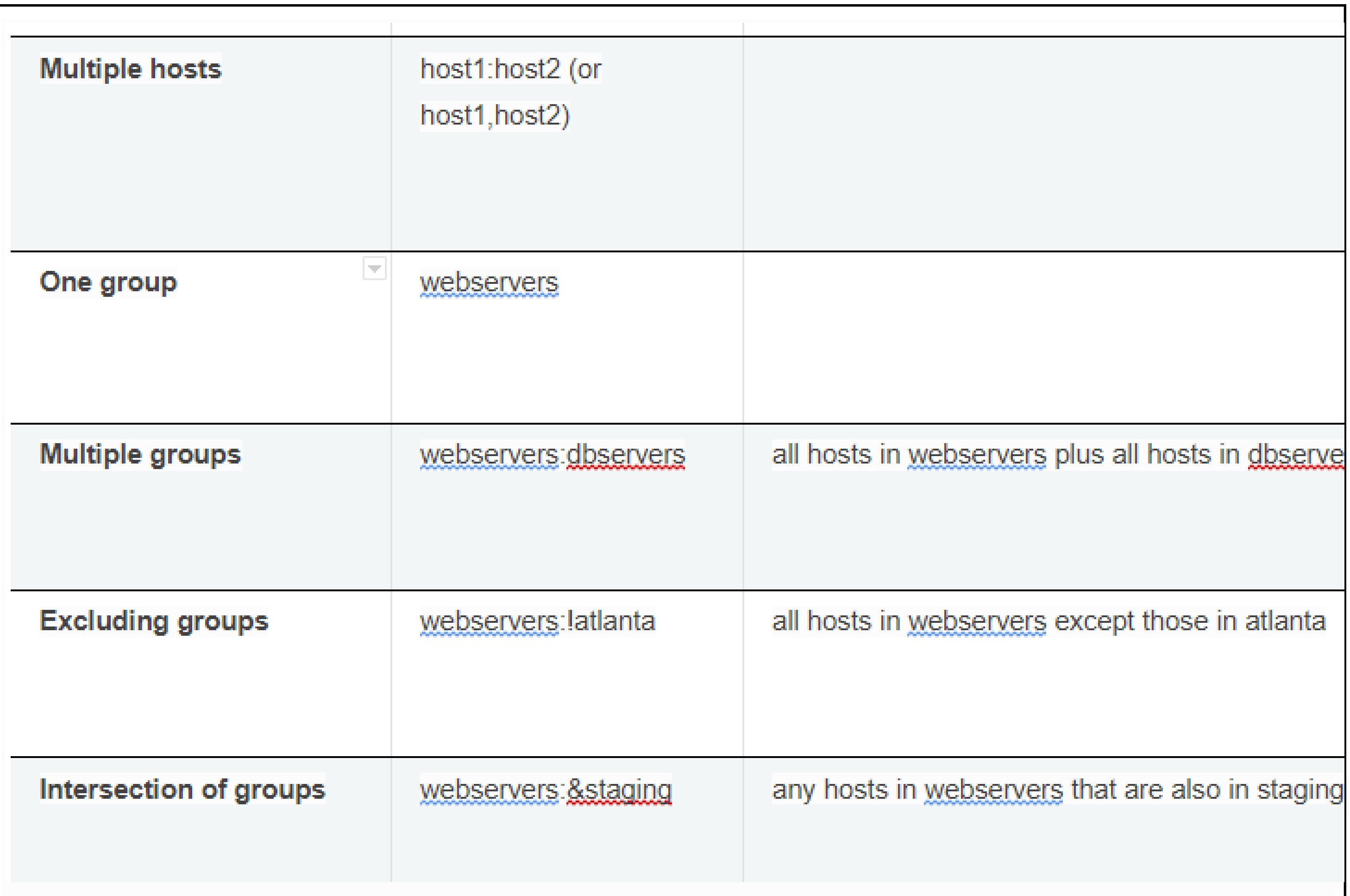
Assigning a variable to one machine: host variables	You can easily assign a variable to a single host, then use it later in playbooks. In INI:
	
	`[atlanta]`
	`host1 http_port=80 maxRequestsPerChild=808`
	`host2 http_port=303 maxRequestsPerChild=909`

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`
	
	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 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.

Connecting to hosts: behavioral inventory parameters	**ansible_host** **ansible_port** **ansible_user** **ansible_password** **ansible_ssh_private_key_file** **ansible_ssh_common_args** **ansible_sftp_extra_args / ansible_scp_extra_args / ansible_ssh_extra_args** **ansible_become**

	ansible_become_method / ansible_become_user /
	
	 ansible_become_password
	
	 ansible_python_interpreter: The target host python path. This is useful for systems with more than one Python
	
	 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`

Using patterns	You use a pattern almost any time you execute an ad hoc command or a playbook.



Multiple hosts	host1:host2 (or host1,host2)	
One group	webservers	
Multiple groups	webservers:dbservers	all hosts in webservers plus all hosts in dbserve
Excluding groups	webservers:!atlanta	all hosts in webservers except those in atlanta
Intersection of groups	webservers:&staging	any hosts in webservers that are also in staging

Managing host key checking

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` environment variable:

```
$ export ANSIBLE_HOST_KEY_CHECKING=False
```

Working with command line tools	
ansible	**--ask-vault-password, --ask-vault-pass** **--become-user <BECOME_USER>** **--list-hosts** **--playbook-dir <BASEDIR>** **--private-key <PRIVATE_KEY_FILE>, --key-file <PRIVATE_KEY_FILE>** **--scp-extra-args / --sftp-extra-args / --ssh-common-args / --ssh-extra-args** **--syntax-check** **--vault-id** **--vault-password-file, --vault-pass-file** **-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 **-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) **-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) **-i, --inventory, --inventory-file** specify inventory host path or comma separated host list. –inventory-file is deprecated **-u <REMOTE_USER>, --user <REMOTE_USER>** connect as this user (default=None) **-v, --verbose** verbose mode (-vvv for more, -vvvv to enable connection debugging)
ansible-config	**Synopsis** `usage: ansible-config [-h] [--version] [-v] {list,dump,view} ...` **Description** Config command line class

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

ansible-galaxy	• role ○ role init ○ role remove ○ role delete

	○ role list ○ role search ○ role import ○ role setup ○ role info ○ role install

ansible-playbook	
	```
	usage: ansible-playbook [-h] [--version] [-v] [-k]
	                        [--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]
	[--force-handlers]
	                        [--flush-cache] [-b] [--become-method
	BECOME_METHOD]
	                        [--become-user BECOME_USER] [-K] [-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 ...]
	```

ansible-pull	**pulls playbooks from a VCS repo and executes them for the local host**
	
	Synopsis
	```
	usage: ansible-pull [-h] [--version] [-v] [-k]
	                    [--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] [--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] [--purge] [-o] [-s SLEEP] [-f] [-d DEST]
[-U URL]
                 [--full] [-C CHECKOUT] [--accept-host-key]
                 [-m MODULE_NAME] [--verify-commit] [--clean]
                 [--track-subs] [--check] [--diff]
                 [playbook.yml [playbook.yml ...]]
```

ansible-vault

encryption/decryption utility for Ansible data files

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.

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

You can use Jinja2 syntax in playbooks. For example:

```
ansible.builtin.template:
  src: foo.cfg.j2
  dest: '{{ remote_install_path }}/foo.cfg'
```

Defining variables as lists

```
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] }}"`

Defining variables as key:value dictionaries	`foo:`
	`  field1: one`
	`  field2: two`

Referencing key:value dictionary variables	`foo['field1']`
	`foo.field1`

Registering variables	You can create variables from the output of an Ansible task with the task keyword `register`.
	
	`- 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`

Referencing nested variables	Many registered variables (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 }}`

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 page has details on setting host variables and 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	This example shows how you can include variables defined in an external file:
	`---`
	`- hosts: all`
	`  remote_user: root`
	`  vars:`

```yaml
    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:

```yaml
---
# in the above example, this would be vars/external_vars.yml
somevar: somevalue
password: magic
```

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.

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

Variable precedence: Where should I put a variable?

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)

—-------------

1. Role defaults
2. Inventory variables
3. Inventory group vars
4. Inventory host vars
5. Playbook group vars then host vars
6. Host facts
7. Play vars
8. Role vars
9. Block vars
10. Task vars
11. Extra vars from command line

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

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 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) or defined in AWX or on Red Hat Ansible Automation 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 for more info. For example:

```
---
# 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 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
```
