**Intro to Playbooks**

**About Playbooks**

Playbooks are a completely different way to use ansible than in adhoc task execution mode, and are particularly powerful.

Simply put, playbooks are the basis for a really simple configuration management and multi-machine deployment system, unlike any that already exist, and one that is very well suited to deploying complex applications.

Playbooks can declare configurations, but they can also orchestrate steps of any manual ordered process, even as different steps must bounce back and forth between sets of machines in particular orders. They can launch tasks synchronously or asynchronously.

While you might run the main /usr/bin/ansible program for ad-hoc tasks, playbooks are more likely to be kept in source control and used to push out your configuration or assure the configurations of your remote systems are in spec.

There are also some full sets of playbooks illustrating a lot of these techniques in the [ansible-examples repository](https://github.com/ansible/ansible-examples). We’d recommend looking at these in another tab as you go along.

There are also many jumping off points after you learn playbooks, so hop back to the documentation index after you’re done with this section.

**Playbook Language Example**

Playbooks are expressed in YAML format (see [YAML Syntax](https://docs.ansible.com/ansible/2.6/reference_appendices/YAMLSyntax.html#yaml-syntax)) and have a minimum of syntax, which intentionally tries to not be a programming language or script, but rather a model of a configuration or a process.

Each playbook is composed of one or more ‘plays’ in a list.

The goal of a play is to map a group of hosts to some well defined roles, represented by things ansible calls tasks. At a basic level, a task is nothing more than a call to an ansible module (see [Working With Modules](https://docs.ansible.com/ansible/2.6/user_guide/modules.html)).

By composing a playbook of multiple ‘plays’, it is possible to orchestrate multi-machine deployments, running certain steps on all machines in the webservers group, then certain steps on the database server group, then more commands back on the webservers group, etc.

“plays” are more or less a sports analogy. You can have quite a lot of plays that affect your systems to do different things. It’s not as if you were just defining one particular state or model, and you can run different plays at different times.

For starters, here’s a playbook that contains just one play:

**---**

- hosts: webservers

vars:

http\_port: 80

max\_clients: 200

remote\_user: root

tasks:

- name: ensure apache is at the latest version

yum:

name: httpd

state: latest

- name: write the apache config file

template:

src: /srv/httpd.j2

dest: /etc/httpd.conf

notify:

- restart apache

- name: ensure apache is running

service:

name: httpd

state: started

handlers:

- name: restart apache

service:

name: httpd

state: restarted

Playbooks can contain multiple plays. You may have a playbook that targets first the web servers, and then the database servers. For example:

**---**

- hosts: webservers

remote\_user: root

tasks:

- name: ensure apache is at the latest version

yum:

name: httpd

state: latest

- name: write the apache config file

template:

src: /srv/httpd.j2

dest: /etc/httpd.conf

- hosts: databases

remote\_user: root

tasks:

- name: ensure postgresql is at the latest version

yum:

name: postgresql

state: latest

- name: ensure that postgresql is started

service:

name: postgresql

state: started

You can use this method to switch between the host group you’re targeting, the username logging into the remote servers, whether to sudo or not, and so forth. Plays, like tasks, run in the order specified in the playbook: top to bottom.

Below, we’ll break down what the various features of the playbook language are.

**Basics**

**Hosts and Users**

For each play in a playbook, you get to choose which machines in your infrastructure to target and what remote user to complete the steps (called tasks) as.

The hosts line is a list of one or more groups or host patterns, separated by colons, as described in the [Working with Patterns](https://docs.ansible.com/ansible/2.6/user_guide/intro_patterns.html) documentation. The remote\_user is just the name of the user account:

**---**

- hosts: webservers

remote\_user: root

**Note**

The remote\_user parameter was formerly called just user. It was renamed in Ansible 1.4 to make it more distinguishable from the **user** module (used to create users on remote systems).

Remote users can also be defined per task:

**---**

- hosts: webservers

remote\_user: root

tasks:

- name: test connection

ping:

remote\_user: yourname

Support for running things as another user is also available (see [Understanding Privilege Escalation](https://docs.ansible.com/ansible/2.6/user_guide/become.html)):

**---**

- hosts: webservers

remote\_user: yourname

become: yes

You can also use become on a particular task instead of the whole play:

**---**

- hosts: webservers

remote\_user: yourname

tasks:

- service:

name: nginx

state: started

become: yes

become\_method: sudo

You can also login as you, and then become a user different than root:

**---**

- hosts: webservers

remote\_user: yourname

become: yes

become\_user: postgres

You can also use other privilege escalation methods, like su:

**---**

- hosts: webservers

remote\_user: yourname

become: yes

become\_method: su

If you need to specify a password to sudo, run ansible-playbook with --ask-become-pass or when using the old sudo syntax --ask-sudo-pass (-K). If you run a become playbook and the playbook seems to hang, it’s probably stuck at the privilege escalation prompt. Just *Control-C* to kill it and run it again adding the appropriate password.

**Important**

When using become\_user to a user other than root, the module arguments are briefly written into a random tempfile in /tmp. These are deleted immediately after the command is executed. This only occurs when changing privileges from a user like ‘bob’ to ‘timmy’, not when going from ‘bob’ to ‘root’, or logging in directly as ‘bob’ or ‘root’. If it concerns you that this data is briefly readable (not writable), avoid transferring unencrypted passwords with *become\_user* set. In other cases, /tmp is not used and this does not come into play. Ansible also takes care to not log password parameters.

*New in version 2.4.*

You can also control the order in which hosts are run. The default is to follow the order supplied by the inventory:

- hosts: all

order: sorted

gather\_facts: False

tasks:

- debug:

var: inventory\_hostname

Possible values for order are:

**inventory:**

The default. The order is ‘as provided’ by the inventory

**reverse\_inventory:**

As the name implies, this reverses the order ‘as provided’ by the inventory

**sorted:**

Hosts are alphabetically sorted by name

**reverse\_sorted:**

Hosts are sorted by name in reverse alphabetical order

**shuffle:**

Hosts are randomly ordered each run

**Tasks list**

Each play contains a list of tasks. Tasks are executed in order, one at a time, against all machines matched by the host pattern, before moving on to the next task. It is important to understand that, within a play, all hosts are going to get the same task directives. It is the purpose of a play to map a selection of hosts to tasks.

When running the playbook, which runs top to bottom, hosts with failed tasks are taken out of the rotation for the entire playbook. If things fail, simply correct the playbook file and rerun.

The goal of each task is to execute a module, with very specific arguments. Variables, as mentioned above, can be used in arguments to modules.

Modules should be idempotent, that is, running a module multiple times in a sequence should have the same effect as running it just once. One way to achieve idempotency is to have a module check whether its desired final state has already been achieved, and if that state has been achieved, to exit without performing any actions. If all the modules a playbook uses are idempotent, then the playbook itself is likely to be idempotent, so re-running the playbook should be safe.

The **command** and **shell** modules will typically rerun the same command again, which is totally ok if the command is something like chmod or setsebool, etc. Though there is a creates flag available which can be used to make these modules also idempotent.

Every task should have a name, which is included in the output from running the playbook. This is human readable output, and so it is useful to provide good descriptions of each task step. If the name is not provided though, the string fed to ‘action’ will be used for output.

Tasks can be declared using the legacy action: module options format, but it is recommended that you use the more conventional module: options format. This recommended format is used throughout the documentation, but you may encounter the older format in some playbooks.

Here is what a basic task looks like. As with most modules, the service module takes key=value arguments:

tasks:

- name: make sure apache is running

service:

name: httpd

state: started

The **command** and **shell** modules are the only modules that just take a list of arguments and don’t use the key=value form. This makes them work as simply as you would expect:

tasks:

- name: enable selinux

command: /sbin/setenforce 1

The **command** and **shell** module care about return codes, so if you have a command whose successful exit code is not zero, you may wish to do this:

tasks:

- name: run this command and ignore the result

shell: /usr/bin/somecommand || /bin/true

Or this:

tasks:

- name: run this command and ignore the result

shell: /usr/bin/somecommand

ignore\_errors: True

If the action line is getting too long for comfort you can break it on a space and indent any continuation lines:

tasks:

- name: Copy ansible inventory file to client

copy: src=/etc/ansible/hosts dest=/etc/ansible/hosts

owner=root group=root mode=0644

Variables can be used in action lines. Suppose you defined a variable called vhost in the vars section, you could do this:

tasks:

- name: create a virtual host file for {{ vhost }}

template:

src: somefile.j2

dest: /etc/httpd/conf.d/{{ vhost }}

Those same variables are usable in templates, which we’ll get to later.

Now in a very basic playbook all the tasks will be listed directly in that play, though it will usually make more sense to break up tasks as described in [Creating Reusable Playbooks](https://docs.ansible.com/ansible/2.6/user_guide/playbooks_reuse.html).

**Action Shorthand**

*New in version 0.8.*

Ansible prefers listing modules like this:

template:

src: templates/foo.j2

dest: /etc/foo.conf

Early versions of Ansible used the following format, which still works:

action: template src=templates/foo.j2 dest=/etc/foo.conf

**Handlers: Running Operations On Change**

As we’ve mentioned, modules should be idempotent and can relay when they have made a change on the remote system. Playbooks recognize this and have a basic event system that can be used to respond to change.

These ‘notify’ actions are triggered at the end of each block of tasks in a play, and will only be triggered once even if notified by multiple different tasks.

For instance, multiple resources may indicate that apache needs to be restarted because they have changed a config file, but apache will only be bounced once to avoid unnecessary restarts.

Here’s an example of restarting two services when the contents of a file change, but only if the file changes:

- name: template configuration file

template:

src: template.j2

dest: /etc/foo.conf

notify:

- restart memcached

- restart apache

The things listed in the notify section of a task are called handlers.

Handlers are lists of tasks, not really any different from regular tasks, that are referenced by a globally unique name, and are notified by notifiers. If nothing notifies a handler, it will not run. Regardless of how many tasks notify a handler, it will run only once, after all of the tasks complete in a particular play.

Here’s an example handlers section:

handlers:

- name: restart memcached

service:

name: memcached

state: restarted

- name: restart apache

service:

name: apache

state: restarted

As of Ansible 2.2, handlers can also “listen” to generic topics, and tasks can notify those topics as follows:

handlers:

- name: restart memcached

service:

name: memcached

state: restarted

listen: "restart web services"

- name: restart apache

service:

name: apache

state:restarted

listen: "restart web services"

tasks:

- name: restart everything

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**

* Notify handlers are always run in the same 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.
* If two handler tasks have the same name, only one will run. [\*](https://github.com/ansible/ansible/issues/4943)
* You cannot notify a handler that is defined inside of an include. As of Ansible 2.1, this does work, however the include must be *static*.

Roles are described later on, but it’s worthwhile to point out that:

* handlers notified within pre\_tasks, tasks, and post\_tasks sections are automatically flushed in the end of section where they were notified;
* handlers notified within roles section are automatically flushed in the end of tasks section, but before any taskshandlers.

If you ever want to flush all the handler commands immediately you can do this:

tasks:

- shell: some tasks go here

- meta: flush\_handlers

- shell: some other tasks

In the above example any queued up handlers would be processed early when the meta statement was reached. This is a bit of a niche case but can come in handy from time to time.

**Executing A Playbook**

Now that you’ve learned playbook syntax, how do you run a playbook? It’s simple. Let’s run a playbook using a parallelism level of 10:

ansible-playbook playbook.yml -f 10

**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) available to configure ansible-pull via a crontab from push mode.

**Tips and Tricks**

To check the syntax of a playbook, use ansible-playbook with the --syntax-check flag. This will run the playbook file through the parser to ensure its included files, roles, etc. have no syntax problems.

Look at the bottom of the playbook execution for 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.

If you ever want to see detailed output from successful modules as well as unsuccessful ones, use the --verbose flag. This is available in Ansible 0.5 and later.

To see what hosts would be affected by a playbook before you run it, you can do this:

ansible-playbook playbook.yml --list-hosts