Intro to Playbooks

Playbooks are a completely different way to use ansible than in ad-hoc 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.

Playbooks are expressed in `YAML` format 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.

ℹ **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.

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.

Here's a playbook, first-playbook.yml that contains two plays: (First play starting from line-2, and the second play starts from line-15)

```
---
- 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
```

⚠ **Note:**

- 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.**

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 Patterns: targeting hosts and groups documentation. The `remote_user` is just the name of the user account:

```
---
- hosts: webservers
  remote_user: root
```

⚠ **Note:**

- You must ensure that the host or group is first defined in the **inventory file** we created earlier.
- The host defined in the inventory file must match the host used in the playbook and all connection information for the host is retrieved from the inventory file.

Modules

Modules (also referred to as "task plugins" or "library plugins") are **discrete units of code** that can be used from the **command line** or in a **playbook task**. Ansible executes each module, usually on the remote target node, and collects return values.

You can execute modules from the command line:

```
$ ansible webservers -m service -a "name=httpd state=started"
$ ansible webservers -m ping
$ ansible webservers -m command -a "/sbin/reboot -t now"
```

Each module supports taking arguments. Nearly all modules take key=value arguments, space delimited. Some modules take no arguments, and the command/shell modules simply take the string of the command you want to run.

From playbooks, Ansible modules are executed in a very similar way:

```
- name: reboot the servers
  command: /sbin/reboot -t now
```

All modules return JSON format data. This means modules can be written in any programming language. Modules should be **idempotent**, and should avoid making any changes if they detect that the current state matches the desired final state. When used in an Ansible playbook, modules can trigger `change events` in the form of notifying `handlers` to run additional tasks.

Tasks

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.

⚠ **Note:** The goal of each task is to execute a module, with very specific arguments. Variables can be used in arguments to modules.

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.

Here is what a basic task looks like:

```
tasks:
  - name: make sure apache is running
    service:
      name: httpd
      state: started
```

Handlers

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
```

Inventory File

Ansible works against multiple managed nodes or “hosts” in your infrastructure at the same time, using a list or group of lists know as `inventory`. Once your inventory is defined, you use 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, and/or pull inventory.

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.

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:
```

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 [webservers]. You can create groups that track:

- What - An application, stack or microservice. (For example, database servers, web servers, etc).
- Where - A datacenter or region, to talk to local DNS, storage, etc. (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:
    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:
      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 dbservers, 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:
```

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.

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]"
```

> *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.

Complementary Lesson about Playbooks