Chapter 3 - Ad-Hoc Commands

In the previous chapter, we ended our exploration of local infrastructure testing with Vagrant by creating a very simple Ansible playbook. Earlier still, we used a simple ansible ad-hoc command to run a one-off command on a remote server.

We'll dive deeper into playbooks in coming chapters; for now, we'll explore how Ansible helps you quickly perform common tasks on, and gather data from, one or many servers with ad-hoc commands.

Conducting an orchestra

The number of servers managed by an individual administrator has risen dramatically in the past decade, especially as virtualization and growing cloud application usage has become standard fare. As a result, admins have had to find new ways of managing servers in a streamlined fashion.

On any given day, a systems administrator has many tasks:

- Apply patches and updates via `yum`, `apt`, and other package managers.
- Check resource usage (disk space, memory, CPU, swap space, network).
- Check log files.
- Manage system users and groups.
- Manage DNS settings, hosts files, etc.
- Copy files to and from servers.
- Deploy applications or run application maintenance.
- Reboot servers.
- Manage cron jobs.

Nearly all of these tasks can be (and usually are) at least partially automated—but some often need a human touch, especially when it comes to diagnosing issues in

real time. And in today's complex multi-server environments, logging into servers individually is not a workable solution.

Ansible allows admins to run ad-hoc commands on one or hundreds of machines at the same time, using the `ansible` command. In Chapter 1, we ran a couple of commands (`ping` and `free -m`) on a server that we added to our Ansible inventory file. This chapter will explore ad-hoc commands and multi-server environments in much greater detail. Even if you decide to ignore the rest of Ansible's powerful features, you will be able to manage your servers much more efficiently after reading this chapter.

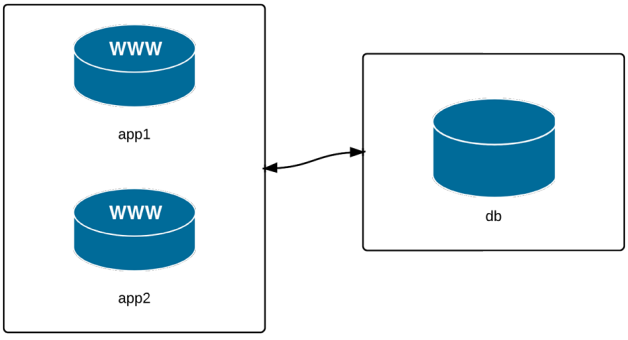
Some of the examples in this chapter will display how you can configure certain aspects of a server with ad-hoc commands. It is usually more appropriate to contain all configuration within playbooks and templates, so it's easier to provision your servers (running the playbook the first time) and then ensure their configuration is idempotent (you can run the playbooks over and over again, and your servers will be in the correct state).

The examples in this chapter are for illustration purposes only, and all might not be applicable to your environment. But even if you *only* used Ansible for server management and running individual tasks against groups of servers, and didn't use Ansible's playbook functionality at all, you'd still have a great orchestration and deployment tool in Ansible!

Build infrastructure with Vagrant for testing

For the rest of this chapter, since we want to do a bunch of experimentation without damaging any production servers, we're going to use Vagrant's powerful multi-machine capabilities to configure a few servers which we'll manage with Ansible.

Earlier, we used Vagrant to boot up one virtual machine running CentOS 7. In that example, we used all of Vagrant's default configuration defined in the Vagrantfile. In this example, we'll use Vagrant's powerful multi-machine management features.



Three servers: two application, one database.

We're going to manage three VMs: two app servers and a database server. Many simple web applications and websites have a similar architecture, and even though this may not reflect the vast realm of infrastructure combinations that exist, it will be enough to highlight Ansible's server management abilities.

To begin, create a new folder somewhere on your local drive (I like using `~/VMs/[dir]`), and create a new blank file named `Vagrantfile` (this is how we describe our virtual machines to Vagrant). Open the file in your favorite editor, add the following, and save the file:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # General Vagrant VM configuration.
  config.vm.box = "geerlingguy/centos7"
  config.ssh.insert_key = false
  config.vm.synced_folder ".", "/vagrant", disabled: true
  config.vm.provider :virtualbox do |v|
    v.memory = 256
    v.linked_clone = true
  end

```

```
  # Application server 1.
  config.vm.define "app1" do |app|
    app.vm.hostname = "orc-app1.test"
    app.vm.network :private_network, ip: "192.168.60.4"
  end

  # Application server 2.
  config.vm.define "app2" do |app|
    app.vm.hostname = "orc-app2.test"
    app.vm.network :private_network, ip: "192.168.60.5"
  end

  # Database server.
  config.vm.define "db" do |db|
    db.vm.hostname = "orc-db.test"
    db.vm.network :private_network, ip: "192.168.60.6"
  end
end
```

This Vagrantfile defines the three servers we want to manage, and gives each one a unique hostname, machine name (for VirtualBox), and IP address. For simplicity's sake, all three servers will be running CentOS 7.

Open up a terminal window and change directory to the same folder where the Vagrantfile you just created exists. Enter `vagrant up` to let Vagrant begin building the three VMs. If you already downloaded the box while building the example from Chapter 2, this process shouldn't take too long—maybe 3-5 minutes.

While that's going on, we'll work on telling Ansible about the servers, so we can start managing them right away.

Inventory file for multiple servers

There are many ways you can tell Ansible about the servers you manage, but the most standard, and simplest, is to add them to your system's main Ansible inventory file, which is located at `/etc/ansible/hosts`. If you didn't create the file in the previous

chapter, go ahead and create the file now; make sure your user account has read permissions for the file.

Add the following to the file:

```
# Lines beginning with a # are comments, and are only included for
# illustration. These comments are overkill for most inventory files.

# Application servers
[app]
192.168.60.4
192.168.60.5

# Database server
[db]
192.168.60.6

# Group 'multi' with all servers
[multi:children]
app
db

# Variables that will be applied to all servers
[multi:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
```

Let's step through this example, group by group:

1. The first block puts both of our application servers into an 'app' group.
2. The second block puts the database server into a 'db' group.
3. The third block tells ansible to define a new group 'multi', with child groups, and we add in both the 'app' and 'db' groups.
4. The fourth block adds variables to the multi group that will be applied to *all* servers within multi and all its children.

We'll dive deeper into variables, group definitions, group hierarchy, and other Inventory file topics later. For now, we just want Ansible to know about our servers, so we can start managing them quickly.

Save the updated inventory file, and then check to see if Vagrant has finished building the three VMs. Once Vagrant has finished, we can start managing the servers with Ansible.

Your first ad-hoc commands

One of the first things you need to do is to check in on your servers. Let's make sure they're configured correctly, have the right time and date (we don't want any time synchronization-related errors in our application!), and have enough free resources to run an application.

Many of the things we're manually checking here should also be monitored by an automated system on production servers; the best way to prevent disaster is to know when it could be coming, and to fix the problem *before* it happens. You should use tools like Munin, Nagios, Cacti, Hyperic, etc. to ensure you have a good idea of your servers' past and present resource usage! If you're running a website or web application available over the Internet, you should probably also use an external monitoring solution like Pingdom or Server Check.in.

Discover Ansible's parallel nature

First, I want to make sure Vagrant configured the VMs with the right hostnames. Use ansible with the `-a` argument 'hostname' to run hostname against all the servers:

```
$ ansible multi -a "hostname"
```

Ansible will run this command against all three of the servers, and return the results (if Ansible can't reach one server, it will show an error for that server, but continue running the command on the others).

If Ansible reports `No hosts matched` or returns some other inventory-related error, try setting the `ANSIBLE_INVENTORY` environment variable explicitly: `export ANSIBLE_INVENTORY=/etc/ansible/hosts`. Generally Ansible will read the file in `/etc/ansible/hosts` automatically, but depending on how you installed Ansible, you may need to explicitly set `ANSIBLE_INVENTORY` for the `ansible` command to work correctly.

If you get an error like `The authenticity of host '192.168.60.5' can't be established.`, this is because SSH has a security feature which requires you to confirm the server's 'host key' the first time you connect. You can type `yes` on the command line (in this case multiple times) to dismiss the warning and accept the hostkey, or you can also set the environment variable `ANSIBLE_HOST_KEY_CHECKING=False`. The behavior for host key acceptance can also be configured in your SSH configuration file.

You may have noticed that the command was not run on each server in the order you'd expect. Go ahead and run the command a few more times, and see the order:

```
# First run results:
192.168.60.5 | success | rc=0 >>
orc-app2.test

192.168.60.6 | success | rc=0 >>
orc-db.test

192.168.60.4 | success | rc=0 >>
orc-app1.test
```

```
# Second run results:
192.168.60.6 | success | rc=0 >>
orc-db.test

192.168.60.5 | success | rc=0 >>
orc-app2.test

192.168.60.4 | success | rc=0 >>
orc-app1.test
```

By default, Ansible will run your commands in parallel, using multiple process forks, so the command will complete more quickly. If you're managing a few servers, this may not be much quicker than running the command serially, on one server after the other, but even managing 5-10 servers, you'll notice a dramatic speedup if you use Ansible's parallelism (which is enabled by default).

Run the same command again, but this time, add the argument `-f 1` to tell Ansible to use only one fork (basically, to perform the command on each server in sequence):

```
$ ansible multi -a "hostname" -f 1
192.168.60.4 | success | rc=0 >>
orc-app1.test

192.168.60.5 | success | rc=0 >>
orc-app2.test

192.168.60.6 | success | rc=0 >>
orc-db.test
```

Run the same command over and over again, and it will always return results in the same order. It's fairly rare that you will ever need to do this, but it's much more frequent that you'll want to *increase* the value (like `-f 10`, or `-f 25`... depending on how much your system and network connection can handle) to speed up the process of running commands on tens or hundreds of servers.

Most people place the target of the action (`multi`) before the command/action itself ("on X servers, run Y command"), but if your brain works in the reverse order ("run Y command on X servers"), you could put the target *after* the other arguments (`ansible -a "hostname" multi`)—the commands are equivalent.

Learning about your environment

Now that we trust Vagrant's ability to set hostnames correctly, let's make sure everything else is in order.

First, let's make sure the servers have disk space available for our application:

```
$ ansible multi -a "df -h"
192.168.60.6 | success | rc=0 >>
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   19G 1014M   18G   6% /
devtmpfs                 111M     0  111M   0% /dev
tmpfs                    120M     0  120M   0% /dev/shm
tmpfs                    120M  4.3M  115M   4% /run
tmpfs                    120M     0  120M   0% /sys/fs/cgroup
/dev/sda1                497M  124M  374M  25% /boot
none                     233G  217G   17G  94% /vagrant

192.168.60.5 | success | rc=0 >>
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   19G 1014M   18G   6% /
devtmpfs                 111M     0  111M   0% /dev
tmpfs                    120M     0  120M   0% /dev/shm
tmpfs                    120M  4.3M  115M   4% /run
tmpfs                    120M     0  120M   0% /sys/fs/cgroup
/dev/sda1                497M  124M  374M  25% /boot
none                     233G  217G   17G  94% /vagrant

192.168.60.4 | success | rc=0 >>
Filesystem               Size  Used Avail Use% Mounted on
/dev/mapper/centos-root   19G 1014M   18G   6% /
devtmpfs                 111M     0  111M   0% /dev
tmpfs                    120M     0  120M   0% /dev/shm
tmpfs                    120M  4.3M  115M   4% /run
tmpfs                    120M     0  120M   0% /sys/fs/cgroup
/dev/sda1                497M  124M  374M  25% /boot
none                     233G  217G   17G  94% /vagrant
```

It looks like we have plenty of room for now; our application is pretty lightweight.

Second, let's also make sure there is enough memory on our servers:

```
$ ansible multi -a "free -m"
192.168.60.4 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        187         50          4          1         69
-/+ buffers/cache:        116        121
Swap:         1055          0       1055

192.168.60.6 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        190         47          4          1         72
-/+ buffers/cache:        116        121
Swap:         1055          0       1055

192.168.60.5 | success | rc=0 >>
             total       used       free     shared    buffers     cached
Mem:           238        186         52          4          1         67
-/+ buffers/cache:        116        121
Swap:         1055          0       1055
```

Memory is pretty tight, but since we're running three VMs on our localhost, we need to be a little conservative.

Third, let's make sure the date and time on each server is in sync:

```
$ ansible multi -a "date"
192.168.60.5 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2021

192.168.60.4 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2021

192.168.60.6 | success | rc=0 >>
Sat Feb  1 20:23:08 UTC 2021
```

Most applications are written with slight tolerances for per-server time jitter, but it's always a good idea to make sure the times on the different servers are as close as

possible, and the simplest way to do that is to use the Network Time Protocol, which is easy enough to configure. We'll do that next, using Ansible's modules to make the process painless.

To get an exhaustive list of all the environment details ('facts', in Ansible's lingo) for a particular server (or for a group of servers), use the command `ansible [host-or-group] -m setup`. This will provide a list of every minute bit of detail about the server (including file systems, memory, OS, network interfaces… you name it, it's in the list).

Make changes using Ansible modules

We want to install the NTP daemon on the server to keep the time in sync. Instead of running the command `yum install -y ntp` on each of the servers, we'll use ansible's `yum` module to do the same (just like we did in the playbook example earlier, but this time using an ad-hoc command).

```
$ ansible multi -b -m yum -a "name=ntp state=present"
```

You should see three simple 'success' messages, reporting no change, since NTP was already installed on the three machines; this confirms everything is in working order.

The `-b` option (alias for `--become`) tells Ansible to run the command with become (basically, run commands with 'sudo'. This will work fine with our Vagrant VMs, but if you're running commands against a server where your user account requires a password for privilege escalation, you should also pass in `-K` (alias for `--ask-become-pass`), so you can enter your password when Ansible needs it.

Now we'll make sure the NTP daemon is started and set to run on boot. We could use two separate commands, `service ntpd start` and `chkconfig ntpd on`, but we'll use Ansible's `service` module instead.

```
$ ansible multi -b -m service -a "name=ntpd state=started \
enabled=yes"
```

All three servers should show a success message like:

```
"changed": true,
"enabled": true,
"name": "ntpd",
"state": "started"
```

If you run the exact same command again, everything will be the same, but Ansible will report that nothing has changed, so the `"changed"` value becomes `false`.

When you use Ansible's modules instead of plain shell commands, you can use the powers of abstraction and idempotency offered by Ansible. Even if you're running shell commands, you could wrap them in Ansible's `shell` or `command` modules (like `ansible multi -m shell -a "date"`), but for these kind of commands, there's usually no need to use an Ansible module when running them ad-hoc.

The last thing we should do is check to make sure our servers are synced closely to the official time on the NTP server:

```
$ ansible multi -b -a "service ntpd stop"
$ ansible multi -b -a "ntpdate -q 0.rhel.pool.ntp.org"
$ ansible multi -b -a "service ntpd start"
```

For the `ntpdate` command to work, the `ntpd` service has to be stopped, so we stop the service, run the command to check our jitter, then start the service again.

In my test, I was within three one-hundredths of a second on all three servers—close enough for my purposes.

Configure groups of servers, or individual servers

Now that we've been able to get all our servers to a solid baseline (e.g. all of them at least have the correct time), we need to set up the application servers, then the database server.

Since we set up two separate groups in our inventory file, `app` and `db`, we can target commands to just the servers in those groups.

Configure the Application servers

Our hypothetical web application uses Django, so we need to make sure Django and its dependencies are installed. Django is not in the official CentOS yum repository, but we can install it using Python's easy_install (which, conveniently, has an Ansible module).

```
$ ansible app -b -m yum -a "name=MySQL-python state=present"
$ ansible app -b -m yum -a "name=python-setuptools state=present"
$ ansible app -b -m easy_install -a "name=django<2 state=present"
```

You could also install django using `pip`, which can be installed via `easy_install` (since Ansible's `easy_install` module doesn't allow you to uninstall packages like `pip` does), but for simplicity's sake, we've installed it with `easy_install`.

Check to make sure Django is installed and working correctly.

```
$ ansible app -a "python -c 'import django; \
print django.get_version()'"
192.168.60.5 | SUCCESS | rc=0 >>
1.11.12

192.168.60.4 | SUCCESS | rc=0 >>
1.11.12
```

Things look like they're working correctly on our app servers. We can now move on to our database server.

Almost all of the configuration we've done in this chapter would be much better off in an Ansible playbook (which will be explored in greater depth throughout the rest of this book). This chapter demonstrates how easy it is to manage multiple servers—for whatever purpose—using Ansible. Even if you set up and configure servers by hand using shell commands, using Ansible will save you a ton of time and help you do everything in the most secure and efficient manner possible.

Configure the Database servers

We configured the application servers using the `app` group defined in Ansible's main inventory, and we can configure the database server (currently the only server in the `db` group) using the similarly-defined `db` group.

Let's install MariaDB, start it, and configure the server's firewall to allow access on MariaDB's default port, 3306.

```
$ ansible db -b -m yum -a "name=mariadb-server state=present"
$ ansible db -b -m service -a "name=mariadb state=started \
enabled=yes"
$ ansible db -b -a "iptables -F"
$ ansible db -b -a "iptables -A INPUT -s 192.168.60.0/24 -p tcp \
-m tcp --dport 3306 -j ACCEPT"
```

If you try connecting to the database from the app servers (or your host machine) at this point, you won't be able to connect, since MariaDB still needs to be set up. Typically, you'd do this by logging into the server and running `mysql_secure_installation`. Luckily, though, Ansible can control a MariaDB server with its assorted `mysql_*` modules. For now, we need to allow MySQL access for one user from our app servers. The MySQL modules require the `MySQL-python` module to be present on the managed server.

Why MariaDB and not MySQL? RHEL 7 and CentOS 7 have MariaDB as the default supported MySQL-compatible database server. Some of the tooling around MariaDB still uses the old 'MySQL*' naming syntax, but if you're used to MySQL, things work similarly with MariaDB.

```
$ ansible db -b -m yum -a "name=MySQL-python state=present"
$ ansible db -b -m mysql_user -a "name=django host=% password=12345 \
priv=*.*:ALL state=present"
```

At this point, you should be able to create or deploy a Django application on the app servers, then point it at the database server with the username django and password 12345.

The MySQL configuration used here is for example/development purposes only! There are a few other things you should do to secure a production MySQL server, including removing the test database, adding a password for the root user account, restricting the IP addresses allowed to access port 3306 more closely, and some other minor cleanups. Some of these things will be covered later in this book, but, as always, you are responsible for securing your servers—make sure you're doing it correctly!

Make changes to just one server

Congratulations! You now have a small web application environment running Django and MySQL. It's not much, and there's not even a load balancer in front of the app servers to spread out the requests; but we've configured everything pretty quickly, and without ever having to log into a server. What's even more impressive is that you could run any of the ansible commands again (besides a couple of the simple shell commands), and they wouldn't change anything—they would return `"changed": false`, giving you peace of mind that the original configuration is intact.

Now that your local infrastructure has been running a while, you notice (hypothetically, of course) that the logs indicate one of the two app servers' time has gotten way out of sync with the others, likely because the NTP daemon has crashed or somehow been stopped. Quickly, you enter the following command to check the status of `ntpd`:

```
$ ansible app -b -a "service ntpd status"
```

Then, you restart the service on the affected app server:

```
$ ansible app -b -a "service ntpd restart" --limit "192.168.60.4"
```

In this command, we used the `--limit` argument to limit the command to a specific host in the specified group. `--limit` will match either an exact string or a regular expression (prefixed with ~). The above command could be stated more simply if you want to apply the command to only the `.4` server (assuming you know there are no other servers with the an IP address ending in .4), the following would work exactly the same:

```
# Limit hosts with a simple pattern (asterisk is a wildcard).
$ ansible app -b -a "service ntpd restart" --limit "*.4"

# Limit hosts with a regular expression (prefix with a tilde).
$ ansible app -b -a "service ntpd restart" --limit ~".*\.4"
```

In these examples, we've been using IP addresses instead of hostnames, but in many real-world scenarios, you'll probably be using hostnames like nyc-dev-1.example.com; being able to match on regular expressions is often helpful.

Try to reserve the `--limit` option for running commands on single servers. If you often find yourself running commands on the same set of servers using `--limit`, consider instead adding them to a group in your inventory file. That way you can enter `ansible [my-new-group-name] [command]`, and save yourself a few keystrokes.

Manage users and groups

One of the most common uses for Ansible's ad-hoc commands in my day-to-day usage is user and group management. I don't know how many times I've had to re-read the man pages or do a Google search just to remember which arguments I need to create a user with or without a home folder, add the user to certain groups, etc.

Ansible's `user` and `group` modules make things pretty simple and standard across any Linux flavor.

First, add an `admin` group on the app servers for the server administrators:

```
$ ansible app -b -m group -a "name=admin state=present"
```

The `group` module is pretty simple; you can remove a group by setting `state=absent`, set a group id with `gid=[gid]`, and indicate that the group is a system group with `system=yes`.

Now add the user `johndoe` to the app servers with the group I just created and give him a home folder in `/home/johndoe` (the default location for most Linux distributions). Simple:

```
$ ansible app -b -m user -a "name=johndoe group=admin createhome=yes"
```

If you want to automatically create an SSH key for the new user (if one doesn't already exist), you can run the same command with the additional parameter `generate_ssh_key=yes`. You can also set:

- The UID of the user with `uid=[uid]`
- The user's shell with `shell=[shell]`
- The users's password with `password=[encrypted-password]`

What if you want to delete the account?

```
$ ansible app -b -m user -a "name=johndoe state=absent remove=yes"
```

You can do just about anything you could do with `useradd`, `userdel`, and `usermod` using Ansible's `user` module, except you can do it more easily. The official documentation of the User module[47] explains all the possibilities in great detail.

Manage packages

We've already used the `yum` module on our example CentOS infrastructure to ensure certain packages are installed. Ansible has a variety of package management modules for any flavor of Linux, but there's also a generic `package` module that can be used for easier cross-platform Ansible usage.

If you want to install a generic package like `git` on any Debian, RHEL, Fedora, Ubuntu, CentOS, FreeBSD, etc. system, you can use the command:

```
$ ansible app -b -m package -a "name=git state=present"
```

`package` works much the same as `yum`, `apt`, and other package management modules. Later in the book we'll explore ways of dealing with multi-platform package management where package names differ between OSes.

[47] http://docs.ansible.com/user_module.html

Manage files and directories

Another common use for ad-hoc commands is remote file management. Ansible makes it easy to copy files from your host to remote servers, create directories, manage file and directory permissions and ownership, and delete files or directories.

Get information about a file

If you need to check a file's permissions, MD5, or owner, use Ansible's `stat` module:

```
$ ansible multi -m stat -a "path=/etc/environment"
```

This gives the same information you'd get when running the `stat` command, but passes back information in JSON, which can be parsed a little more easily (or, later, used in playbooks to conditionally do or not do certain tasks).

Copy a file to the servers

You probably use `scp` and/or `rsync` to copy files and directories to remote servers, and while Ansible has more advanced file copy modules like `rsync`, most file copy operations can be completed with Ansible's `copy` module:

```
$ ansible multi -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

The `src` can be a file or a directory. If you include a trailing slash, only the contents of the directory will be copied into the `dest`. If you omit the trailing slash, the contents *and* the directory itself will be copied into the `dest`.

The `copy` module is perfect for single-file copies, and works very well with small directories. When you want to copy hundreds of files, especially in very deeply-nested directory structures, you should consider either copying then expanding an archive of the files with Ansible's `unarchive` module, or using Ansible's `synchronize` or `rsync` modules.

Retrieve a file from the servers

The `fetch` module works almost exactly the same as the `copy` module, except in reverse. The major difference is that files will be copied down to the local `dest` in a directory structure that matches the host from which you copied them. For example, use the following command to grab the hosts file from the servers:

```
$ ansible multi -b -m fetch -a "src=/etc/hosts dest=/tmp"
```

Fetch will, by default, put the `/etc/hosts` file from each server into a folder in the destination with the name of the host (in our case, the three IP addresses), then in the location defined by `src`. So, the db server's hosts file will end up in `/tmp/192.168.60.6/etc/hosts`.

You can add the parameter `flat=yes`, and set the `dest` to `dest=/tmp/` (add a trailing slash), to make Ansible fetch the files directly into the `/tmp` directory. However, filenames must be unique for this to work, so it's not as useful when copying down files from multiple hosts. Only use `flat=yes` if you're copying files from a single host.

Create directories and files

You can use the `file` module to create files and directories (like `touch`), manage permissions and ownership on files and directories, modify SELinux properties, and create symlinks.

Here's how to create a directory:

```
$ ansible multi -m file -a "dest=/tmp/test mode=644 state=directory"
```

Here's how to create a symlink (set `state=link`):

```
$ ansible multi -m file -a "src=/src/file dest=/dest/symlink \
state=link"
```

The `src` is the symlink's target file, and the `dest` is the path where the symlink itself should be.

Delete directories and files

You can set the `state` to `absent` to delete a file, directory, or symlink.

```
$ ansible multi -m file -a "dest=/tmp/test state=absent"
```

There are many simple ways to manage files remotely using Ansible. We've briefly covered the `copy` and `file` modules here, but be sure to read the documentation for the other file-management modules like `lineinfile`, `ini_file`, and `unarchive`. This book will cover these additional modules in depth in later chapters (when dealing with playbooks).

Run operations in the background

Some operations take quite a while (minutes or even hours). For example, when you run `yum update` or `apt-get update && apt-get dist-upgrade`, it could be a few minutes before all the packages on your servers are updated.

In these situations, you can tell Ansible to run the commands asynchronously, and poll the servers to see when the commands finish. When you're only managing one server, this is not really helpful, but if you have many servers, Ansible starts the command *very* quickly on all your servers (especially if you set a higher `--forks` value), then polls the servers for status until they're all up to date.

To run a command in the background, you set the following options:

- `-B <seconds>`: the maximum amount of time (in seconds) to let the job run.
- `-P <seconds>`: the amount of time (in seconds) to wait between polling the servers for an updated job status.

  Note: As of Ansible 2.0, asynchronous polling on the command line (via the `-P` flag) no longer displays output in real time. Please follow the progress of this bug's resolution in the issue v2 async events are not triggered[48]. For now, you can still run jobs in the background and get job status information separately, but you can't see the polling status real-time.

[48] https://github.com/ansible/ansible/issues/14681

Update servers asynchronously with asynchronous jobs

Let's run `yum -y update` on all our servers to get them up to date. If you set `-P 0`, Ansible fires off the command on the servers, then prints the background job information to the screen and exits:

```
$ ansible multi -b -B 3600 -P 0 -a "yum -y update"
192.168.60.4 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "ansible_job_id": "333866395772.18507",
    "changed": true,
    "finished": 0,
    "results_file": "/root/.ansible_async/333866395772.18507",
    "started": 1
}

... [other hosts] ...
```

While the background task is running, you can check on the status elsewhere using Ansible's `async_status` module, as long as you have the `ansible_job_id` value to pass in as `jid`:

```
$ ansible multi -b -m async_status -a "jid=169825235950.3572"
192.168.60.6 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "ansible_job_id": "761320869373.18496",
    "changed": true,
    "cmd": [
        "yum",
        "-y",
```

```
        "update"
    ],
    "delta": "0:00:09.017121",
    "end": "2019-08-23 02:32:58.727209",
    "finished": 1,
    "rc": 0,
    "start": "2019-08-23 02:32:49.710088",
    ...
    "stdout_lines": [
        "Loaded plugins: fastestmirror",
        "Determining fastest mirrors",
        " * base: mirror.genesisadaptive.com",
        " * extras: mirror.mobap.edu",
        " * updates: mirror.mobap.edu",
        "No packages marked for update"
    ]
}
```

For tasks you don't track remotely, it's usually a good idea to log the progress of the task *somewhere*, and also send some sort of alert on failure—especially, for example, when running backgrounded tasks that perform backup operations, or when running business-critical database maintenance tasks.

You can also run tasks in Ansible playbooks in the background, asynchronously, by defining an `async` and `poll` parameter on the play. We'll discuss playbook task backgrounding in later chapters.

Check log files

Sometimes, when debugging application errors, or diagnosing outages or other problems, you need to check server log files. Any common log file operation (like using `tail`, `cat`, `grep`, etc.) works through the `ansible` command, with a few caveats:

1. Operations that continuously monitor a file, like `tail -f`, won't work via Ansible, because Ansible only displays output after the operation is complete, and you won't be able to send the Control-C command to stop following the file. Someday, the async module might have this feature, but for now, it's not possible.
2. It's not a good idea to run a command that returns a huge amount of data via stdout via Ansible. If you're going to cat a file larger than a few KB, you should probably log into the server(s) individually.
3. If you redirect and filter output from a command run via Ansible, you need to use the `shell` module instead of Ansible's default `command` module (add `-m shell` to your commands).

As a simple example, let's view the last few lines of the messages log file on each of our servers:

```
$ ansible multi -b -a "tail /var/log/messages"
```

As stated in the caveats, if you want to filter the messages log with something like `grep`, you can't use Ansible's default `command` module, but instead, `shell`:

```
$ ansible multi -b -m shell -a "tail /var/log/messages | \
grep ansible-command | wc -l"

192.168.60.5 | success | rc=0 >>
12

192.168.60.4 | success | rc=0 >>
12

192.168.60.6 | success | rc=0 >>
14
```

This command shows how many ansible commands have been run on each server (the numbers you get may be different).

Manage cron jobs

Periodic tasks run via cron are managed by a system's crontab. Normally, to change cron job settings on a server, you would log into the server, use `crontab -e` under the account where the cron jobs reside, and type in an entry with the interval and job.

Ansible makes managing cron jobs easy with its `cron` module. If you want to run a shell script on all the servers every day at 4 a.m., add the cron job with:

```
$ ansible multi -b -m cron -a "name='daily-cron-all-servers' \
hour=4 job='/path/to/daily-script.sh'"
```

Ansible will assume `*` for all values you don't specify (valid values are `day`, `hour`, `minute`, `month`, and `weekday`). You could also specify special time values like `reboot`, `yearly`, or `monthly` using `special_time=[value]`. You can also set the user the job will run under via `user=[user]`, and create a backup of the current crontab by passing `backup=yes`.

What if we want to remove the cron job? Simple enough, use the same `cron` command, and pass the name of the cron job you want to delete, and `state=absent`:

```
$ ansible multi -b -m cron -a "name='daily-cron-all-servers' \
state=absent"
```

You can also use Ansible to manage custom crontab files; use the same syntax as you used earlier, but specify the location to the cron file with: `cron_file=cron_file_name` (where `cron_file_name` is a cron file located in `/etc/cron.d`).

Ansible denotes Ansible-managed crontab entries by adding a comment on the line above the entry like `#Ansible: daily-cron-all-servers`. It's best to leave things be in the crontab itself, and always manage entries via ad-hoc commands or playbooks using Ansible's `cron` module.

Deploy a version-controlled application

For simple application deployments, where you may need to update a git checkout, or copy a new bit of code to a group of servers, then run a command to finish the deployment, Ansible's ad-hoc mode can help. For more complicated deployments, use Ansible playbooks and rolling update features (which will be discussed in later chapters) to ensure successful deployments with zero downtime.

In the example below, I'll assume we're running a simple application on one or two servers, in the directory `/opt/myapp`. This directory is a git repository cloned from a central server or a service like GitHub, and application deployments and updates are done by updating the clone, then running a shell script at `/opt/myapp/scripts/update.sh`.

First, update the git checkout to the application's new version branch, `1.2.4`, on all the app servers:

```
$ ansible app -b -m git -a "repo=git://example.com/path/to/repo.git \
dest=/opt/myapp update=yes version=1.2.4"
```

Ansible's git module lets you specify a branch, tag, or even a specific commit with the `version` parameter (in this case, we chose to checkout tag `1.2.4`, but if you run the command again with a branch name, like `prod`, Ansible will happily do that instead). To force Ansible to update the checked-out copy, we passed in `update=yes`. The `repo` and `dest` options should be self-explanatory.

If you get a message saying "Failed to find required executable git", you will need to install Git on the server. To do so, run the ad-hoc command `ansible app -b -m package -a "name=git state=present"`.

If you get a message saying the Git server has an "unknown hostkey", add the option `accept_hostkey=yes` to the command, or add the hostkey to your server's `known_hosts` file before running this command.

Then, run the application's `update.sh` shell script:

```
$ ansible app -b -a "/opt/myapp/update.sh"
```

Ad-hoc commands are fine for the simple deployments (like our example above), but you should use Ansible's more powerful and flexible application deployment features described later in this book if you have complex application or infrastructure needs. See especially the 'Rolling Updates' section later in this book.

Ansible's SSH connection history

One of Ansible's greatest features is its ability to function without running any extra applications or daemons on the servers it manages. Instead of using a proprietary protocol to communicate with the servers, Ansible uses the standard and secure SSH connection that is commonly used for basic administration on almost every Linux server running today.

Since a stable, fast, and secure SSH connection is the heart of Ansible's communication abilities, Ansible's implementation of SSH has continually improved throughout the past few years—and is still improving today.

One thing that is universal to all of Ansible's SSH connection methods is that Ansible uses the connection to transfer one or a few files defining a play or command to the remote server, then runs the play/command, then deletes the transferred file(s), and reports back the results. A fast, stable, and secure SSH connection is of paramount importance to Ansible.

Paramiko

In the beginning, Ansible used paramiko—an open source SSH2 implementation for Python—exclusively. However, as a single library for a single language (Python), development of paramiko doesn't keep pace with development of OpenSSH (the standard implementation of SSH used almost everywhere), and its performance and security is slightly worse than OpenSSH.

Ansible continues to support the use of paramiko, and even chooses it as the default for systems (like RHEL 6) which don't support ControlPersist—an option present only in OpenSSH 5.6 or newer. (ControlPersist allows SSH connections to persist so

frequent commands run over SSH don't have to go through the initial handshake over and over again until the `ControlPersist` timeout set in the server's SSH config is reached.)

OpenSSH (default)

In Ansible 1.3, Ansible defaulted to using native OpenSSH connections to connect to servers supporting ControlPersist. Ansible had this ability since version 0.5, but didn't default to it until 1.3.

Most local SSH configuration parameters (like hosts, key files, etc.) are respected, but if you need to connect via a port other than port 22 (the default SSH port), you should specify the port in an inventory file (`ansible_ssh_port` option) or when running `ansible` commands.

OpenSSH is faster, and a little more reliable, than paramiko, but there are ways to make Ansible faster still.

Faster OpenSSH with Pipelining

Modern versions of Ansible allow you to improve on the performance of Ansible's default OpenSSH implementation.

Instead of copying files, running them on the remote server, then removing them, the 'pipelining' method of OpenSSH transfer will send and execute commands for most Ansible modules directly over the SSH connection.

This method of connection can be enabled by adding `pipelining=True` under the `[ssh_connection]` section of the Ansible configuration file (`ansible.cfg`, which will be covered in more detail later).

The `pipelining=True` configuration option won't help much unless you have removed or commented the `Defaults requiretty` option in `/etc/sudoers`. This is commented out in the default configuration for most OSes, but you might want to double-check this setting to make sure you're getting the fastest connection possible!

If you're running a recent version of Mac OS X, Ubuntu, Windows with Cygwin, or most other OS for the host from which you run `ansible` and `ansible-playbook`, you should be running OpenSSH version 5.6 or later, which works perfectly with the `ControlPersist` setting used with all of Ansible's SSH connections settings.

If the host on which Ansible runs has RHEL or CentOS, however, you might need to update your version of OpenSSH so it supports the faster/persistent connection method. Any OpenSSH version 5.6 or greater should work. To install a later version, either compile from source, or use a different repository (like CentALT[49] and `yum update openssh`.

Summary

In this chapter, you learned how to build a multi-server infrastructure for testing on your local workstation using Vagrant, and you configured, monitored, and managed the infrastructure without ever logging in to an individual server. You also learned how Ansible connects to remote servers, and how to use the `ansible` command to perform tasks on many servers quickly in parallel, or one by one.

By now, you should be getting familiar with the basics of Ansible, and you should be able to start managing your own infrastructure more efficiently.

```
 ________________________________________
/ It's easier to seek forgiveness than \
\ ask for permission. (Proverb)        /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

[49] http://mirror.neu.edu.cn/CentALT/readme.txt