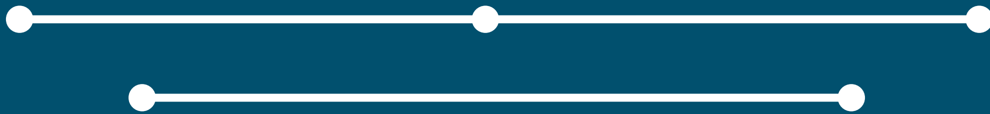
Opensource.com

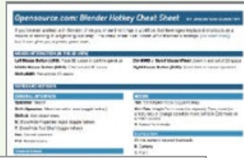
Ansible Automation for SysAdmins



A quickstart guide to Ansible

Open Source Cheat Sheets

Visit our cheat sheets collection for free downloads, including:



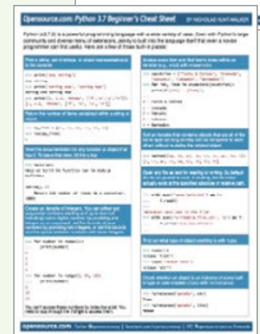
Blender: Discover the most commonly and frequently used hotkeys and mouse button presses.

Containers: Learn the lingo and get the basics in this quick and easy containers primer.

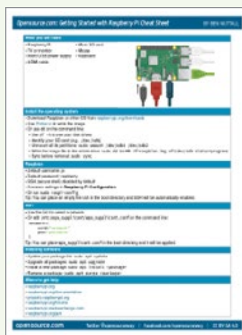
Go: Find out about many uses of the go executable and the most important packages in the Go standard library.

Inkscape: Inkscape is an incredibly powerful vector graphics program that you can use to draw scaleable illustrations or edit vector artwork that other people have created.

Linux Networking: In this downloadable PDF cheat sheet, get a list of Linux utilities and commands for managing servers and networks.



Python 3.7: This cheat sheet rounds up a few built-in pieces to get new Python programmers started.



Raspberry Pi: See what you need to boot your Pi, how to install the operating system, how to enable SSH and connect to WiFi, how to install software and update your system, and links for where to get further help.

SSH: Most people know SSH as a tool for remote login, which it is, but it can be used in many other ways.

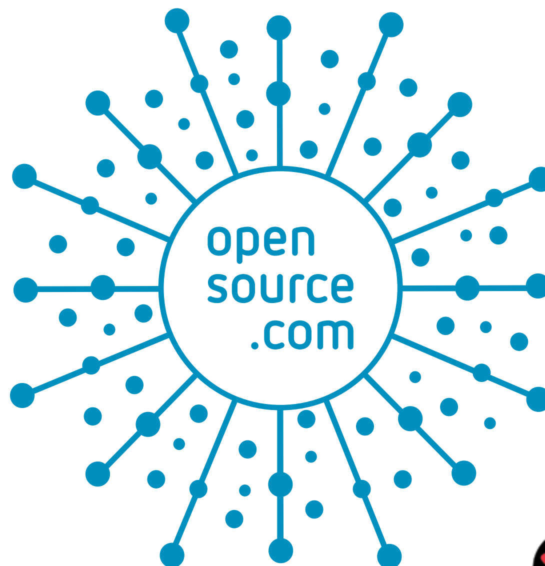
What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: https://opensource.com/story

Email us: open@opensource.com

Chat with us in Freenode IRC: #opensource.com



SUPPORTED BY RED HAT

INTRODUCTION

Introduction 5

CHAPTERS

Tips for success when getting started with Ansible 6
How to use Ansible to patch systems and install applications 8
A sysadmin's guide to Ansible: How to simplify tasks 10
Testing Ansible roles with Molecule 14
Using Ansible for deploying serverless applications 17
4 Ansible playbooks you should try 19

GET INVOLVED | ADDITIONAL RESOURCES

Get involved | Additional Resources 22
Write for Us | Keep in Touch 23

Introduction

BY CHRIS SHORT

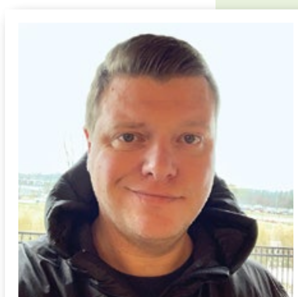
A LOT OF GREAT TOOLS have come and gone over the years. But none of them have made an impact as large as the one that Ansible has made in the IT automation space. From servers to networks to public cloud providers to serverless to Kubernetes… Ansible has a *lot* of use cases.

Happy birthday, Ansible! We assembled this book to celebrate Ansible's seventh birthday. Whether you recently read the Ansible Getting Started doc [1] and are just beginning your Ansible journey or have been going at it for quite some time, this book—much like the Ansible community—offers a little something for everyone.

We hope to spark your imagination about what you can automate next. Here's to seven years of Ansible!

Links

[1] https://docs.ansible.com/ansible/latest/user_guide/intro_getting_started.html



Author

Red Hat Ansible | CNCF Ambassador | DevOps | opensource.com Community Moderator | Writes devopsish.com | Partially Disabled USAF Veteran | He/Him

Tips for success when getting started with Ansible

BY JOSE DELAROSA

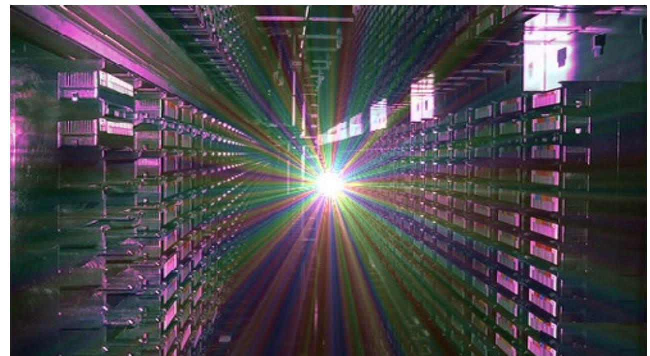
Key information for automating your data center with Ansible.

ANSIBLE IS AN OPEN SOURCE AUTOMATION TOOL used to configure servers, install software, and perform a wide variety of IT tasks from one central location. It is a one-to-many agentless mechanism where all instructions are run from a control machine that communicates with remote clients over SSH, although other protocols are also supported.

While targeted for system administrators with privileged access who routinely perform tasks such as installing and configuring applications, Ansible can also be used by non-privileged users. For example, a database administrator using the `mysql` login ID could use Ansible to create databases, add users, and define access-level controls.

Let's go over a very simple example where a system administrator provisions 100 servers each day and must run a series of Bash commands on each one before handing it off to users.

This is a simple example, but should illustrate how easily commands can be specified in yaml files and executed on remote servers. In a heterogeneous environment, conditional statements can be added so that certain commands are only executed in certain servers (e.g., "only execute `yum` commands in systems that are not Ubuntu or Debian").



One important feature in Ansible is that a playbook describes a *desired* state in a computer system, so a playbook can be run multiple times against a server without impacting its state. If a certain task has already been implemented (e.g., "user `sysman` already exists"), then Ansible simply ignores it and moves on.

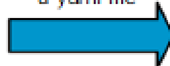
Definitions

- **Tasks:** A task is the smallest unit of work. It can be an action like "Install a database," "Install a web server," "Create a firewall rule," or "Copy this configuration file to that server."

Say you provision 100 servers each day and you run these commands on each one:

```
$ groupadd admin
$ useradd -c "Sys Admin" -g admin -m sysman
$ mkdir /opt/tools
$ chmod 755 /opt/tools
$ chown sysman /opt/tools
$ yum -y install httpd
$ yum -y update
$ systemctl enable httpd
$ systemctl start httpd
$ rm /etc/motd
```

Can be replaced with statements in a yaml file

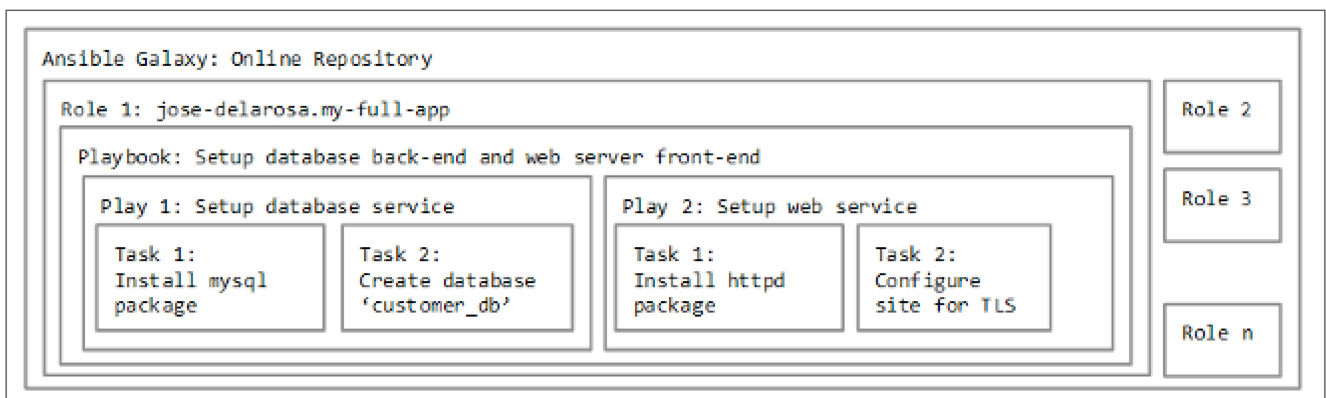


The commands can be placed in a "playbook" and executed against your 100 servers in one shot

```
- name: daily tasks
  hosts: my_100_daily_servers
  tasks:
  - group: name=admin state=present
  - user: name=sysman comment="Sys Admin" group=admin
  - file: path=/opt/tools state=directory owner=sysman
    mode=0755
  - yum: name=httpd state=latest
  - yum: name=* state=latest
  - service: name=httpd state=started enabled=yes
  - file: path=/etc/motd state=absent
```

- **Plays:** A play is made up of tasks. For example, the play: “Prepare a database to be used by a web server” is made up of tasks: 1) Install the database package; 2) Set a password for the database administrator; 3) Create a database; and 4) Set access to the database.
- **Playbook:** A playbook is made up of plays. A playbook could be: “Prepare my website with a database backend,” and the plays would be 1) Set up the database server; and 2) Set up the web server.
- **Roles:** Roles are used to save and organize playbooks and allow sharing and reuse of playbooks. Following the previous examples, if you need to fully configure a web server, you can use a role that others have written and shared to do just that. Since roles are highly configurable (if written correctly), they can be easily reused to suit any given deployment requirements.
- **Ansible Galaxy:** Ansible Galaxy [1] is an online repository where roles are uploaded so they can be shared with others. It is integrated with GitHub, so roles can be organized into Git repositories and then shared via Ansible Galaxy.

These definitions and their relationships are depicted here:



Please note this is just one way to organize the tasks that need to be executed. We could have split up the installation of the database and the web server into separate playbooks and into different roles. Most roles in Ansible Galaxy install and configure individual applications. You can see examples for installing mysql [2] and installing httpd [3].

Tips for writing playbooks

The best source for learning Ansible is the official documentation [4] site. And, as usual, online search is your friend. I recommend starting with simple tasks, like installing applications or creating users. Once you are ready, follow these guidelines:

- When testing, use a small subset of servers so that your plays execute faster. If they are successful in one server, they will be successful in others.
- Always do a dry run to make sure all commands are working (run with `--check-mode` flag).
- Test as often as you need to without fear of breaking things. Tasks describe a desired state, so if a desired state is already achieved, it will simply be ignored.
- Be sure all host names defined in `/etc/ansible/hosts` are resolvable.
- Because communication to remote hosts is done using SSH, keys have to be accepted by the control machine, so either 1) exchange keys with remote hosts prior to starting; or 2) be ready to type in “Yes” to accept SSH key exchange requests for each remote host you want to manage.
- Although you can combine tasks for different Linux distributions in one playbook, it’s cleaner to write a separate playbook for each distro.

In the final analysis

Ansible is a great choice for implementing automation in your data center:

- It’s agentless, so it is simpler to install than other automation tools.
- Instructions are in YAML (though JSON is also supported) so it’s easier than writing shell scripts.
- It’s open source software, so contribute back to it and make it even better!

Links

[1] https://galaxy.ansible.com/
[2] https://galaxy.ansible.com/bennojoy/mysql/
[3] https://galaxy.ansible.com/xcezx/httpd/
[4] http://docs.ansible.com/

Author

Jose is a Linux engineer at Dell EMC. He spends most days learning new things, keeping stuff from breaking, and keeping customers happy.

Adapted from “Tips for success when getting started with Ansible” on Opensource.com, published under a Creative Commons Attribution Share-Alike 4.0 International License at *https://opensource.com/article/18/2/tips-success-when-getting-started-ansible*.

How to use Ansible to patch systems and install applications

BY JONATHAN LOZADA DE LA MATTA

Save time doing updates with the Ansible IT automation engine.

HAVE YOU EVER WONDERED how to patch your systems, reboot, and continue working?

If so, you'll be interested in Ansible [1], a simple configuration management tool that can make some of the hardest work easy. For example, system administration tasks that can be complicated, take hours to complete, or have complex requirements for security.

In my experience, one of the hardest parts of being a sysadmin is patching systems. Every time you get a Common Vulnerabilities and Exposure (CVE) notification or Information Assurance Vulnerability Alert (IAVA) mandated by security, you have to kick into high gear to close the security gaps. (And, believe me, your security officer will hunt you down unless the vulnerabilities are patched.)

Ansible can reduce the time it takes to patch systems by running packaging modules [2]. To demonstrate, let's use the yum module [3] to update the system. Ansible can install, update, remove, or install from another location (e.g., `rpmbuild` from continuous integration/continuous development). Here is the task for updating the system:

```
- name: update the system
  yum:
    name: "*"
    state: latest
```

In the first line, we give the task a meaningful `name` so we know what Ansible is doing. In the next line, the `yum module` updates the CentOS virtual machine (VM), then `name: "*"` tells yum to update everything, and, finally, `state: latest` updates to the latest RPM.

After updating the system, we need to restart and reconnect:

```
- name: restart system to reboot to newest kernel
  shell: "sleep 5 && reboot"
```

```
  async: 1
  poll: 0

- name: wait for 10 seconds
  pause:
    seconds: 10

- name: wait for the system to reboot
  wait_for_connection:
    connect_timeout: 20
    sleep: 5
    delay: 5
    timeout: 60

- name: install epel-release
  yum:
    name: epel-release
    state: latest
```

The `shell module` puts the system to sleep for 5 seconds then reboots. We use `sleep` to prevent the connection from breaking, `async` to avoid timeout, and `poll` to fire & forget. We pause for 10 seconds to wait for the VM to come back and use `wait_for_connection` to connect back to the VM

as soon as it can make a connection. Then we `install epel-release` to test the RPM installation. You can run this playbook multiple times to show the `idempotent`, and the only task that will show as changed is the reboot since we are using the `shell` module. You can use `changed_when: False` to ignore the change when using the `shell` module if you expect no actual changes.

So far we've learned how to update a system, restart the VM, reconnect, and install a RPM. Next we will install NGINX using the role in Ansible Lightbulb [4].

```
- name: Ensure nginx packages are present
  yum:
    name: nginx, python-pip, python-devel, devel
    state: present
  notify: restart-nginx-service

- name: Ensure uwsgi package is present
  pip:
    name: uwsgi
    state: present
  notify: restart-nginx-service

- name: Ensure latest default.conf is present
  template:
    src: templates/nginx.conf.j2
    dest: /etc/nginx/nginx.conf
    backup: yes
  notify: restart-nginx-service

- name: Ensure latest index.html is present
  template:
    src: templates/index.html.j2
    dest: /usr/share/nginx/html/index.html

- name: Ensure nginx service is started and enabled
  service:
    name: nginx
    state: started
    enabled: yes

- name: Ensure proper response from localhost can be received
  uri:
    url: "http://localhost:80/"
    return_content: yes
  register: response
  until: 'nginx_test_message in response.content'
  retries: 10
  delay: 1
```

And the handler that restarts the nginx service:

```
# handlers file for nginx-example
  - name: restart-nginx-service
    service:
      name: nginx
      state: restarted
```

In this role, we install the RPMs `nginx`, `python-pip`, `python-devel`, and `devel` and install `uwsgi` with PIP. Next, we use the `template` module to copy over the `nginx.conf` and `index.html` for the page to display. After that, we make sure the service is enabled on boot and started. Then we use the `uri` module to check the connection to the page.

Here is a playbook showing an example of updating, restarting, and installing an RPM. Then continue installing nginx. This can be done with any other roles/applications you want.

```
  - hosts: all
    roles:
      - centos-update
      - nginx-simple
```

This was just a simple example of how to update, reboot, and continue. For simplicity, I added the packages without variables [5]. Once you start working with a large number of hosts, you will need to change a few settings:

- async & poll [6]
- serial [7]
- forks [8]

This is because on your production environment you might want to update one system at a time (not fire & forget) and actually wait a longer time for your system to reboot and continue.

Links

[1] https://www.ansible.com/overview/how-ansible-works
[2] https://docs.ansible.com/ansible/latest/list_of_packaging_modules.html
[3] https://docs.ansible.com/ansible/latest/yum_module.html
[4] https://github.com/ansible/lightbulb/tree/master/examples/nginx-role
[5] https://docs.ansible.com/ansible/latest/playbooks_variables.html
[6] https://docs.ansible.com/ansible/latest/playbooks_async.html
[7] https://docs.ansible.com/ansible/latest/playbooks_delegation.html#rolling-update-batch-size
[8] https://docs.ansible.com/ansible/latest/intro_configuration.html#forks

Author

Jlozadad is a Ansible Consultant. I'm from Carolina, Puerto Rico and I love IT & Gaming. Spend Most of my time playing video games, looking at open source software and laughing.

Adapted from "How to use Ansible to patch systems and install applications" on Opensource.com, published under a Creative Commons Attribution Share-Alike 4.0 International License at *https://opensource.com/article/18/3/ansible-patch-systems*.

A sysadmin's guide to Ansible: **How to simplify tasks**

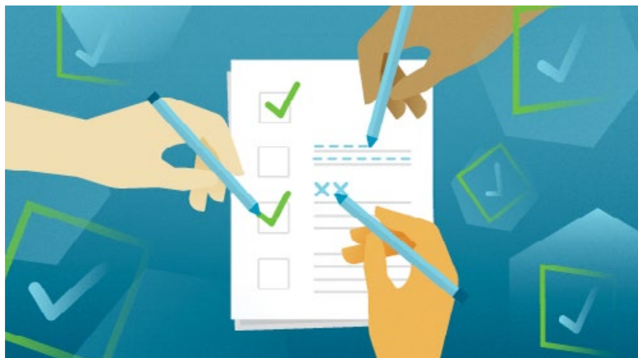
BY JONATHAN LOZADA DE LA MATTA

There are many ways to automate common sysadmin tasks with Ansible. Here are several of them.

IN THE PREVIOUS CHAPTER, I discussed how to use Ansible to patch systems and install applications. In this chapter, I'll show you how to do other things with Ansible that will make your life as a sysadmin easier. First, though, I want to share why I came to Ansible.

I started using Ansible because it made patching systems easier. I could run some ad-hoc commands here and there and some playbooks someone else wrote. I didn't get very in depth, though, because the playbook I was running used a lot of `lineinfile` modules, and, to be honest, my `regex` techniques were nonexistent. I was also limited in my capacity due to my management's direction and instructions: "You can run this playbook only and that's all you can do."

After leaving that job, I started working on a team where most of the infrastructure was in the cloud. After getting used to the team and learning how everything works, I started trying to find ways to automate more things. We were spending two to three months deploying virtual machines in large numbers—doing all the work manually, including the lifecycle of each virtual machine, from provision to decommission. Our work often got behind schedule, as we spent a lot of time doing maintenance. When folks went on vacation, others had to take over with little knowledge of the tasks they were doing.



Diving deeper into Ansible

Sharing ideas about how to resolve issues is one of the best things we can do in the IT and open source world, so I went looking for help by submitting issues in Ansible [1] and asking questions in roles others created [2].

Reading the documentation (including the following topics) is the best way to get started learning Ansible.

- Getting started [3]
- Best practices [4]
- Ansible Lightbulb [5]
- Ansible FAQ [6]

If you are trying to figure out what you can do with Ansible, take a moment and think about the daily activities you do, the ones that take a lot of time that would be better spent on other things. Here are some examples:

- **Managing accounts in systems:** Creating users, adding them to the correct groups, and adding the SSH keys… these are things that used to take me days when we had a large number of systems to build. Even using a shell script, this process was very time-consuming.
- **Maintaining lists of required packages:** This could be part of your security posture and include the packages required for your applications.
- **Installing applications:** You can use your current documentation and convert application installs into tasks by finding the correct module [7] for the job.
- **Configuring systems and applications:** You might want to change `/etc/ssh/sshd_config` for different environments (e.g., production vs. development) by adding a line or two, or maybe you want a file to look a specific way in every system you're managing.
- **Provisioning a VM in the cloud:** This is great when you need to launch a few virtual machines that are similar for your applications and you are tired of using the UI.

Now let's look at how to use Ansible to automate some of these repetitive tasks.

Managing users

If you need to create a large list of users and groups with the users spread among the different groups, you can use `loops`. Let's start by creating the groups:

```
- name: create user groups
  group:
    name: "{{ item }}"
  loop:
    - postgresql
    - nginx-test
    - admin
    - dbadmin
    - hadoop
```

You can create users with specific parameters like this:

```
- name: all users in the department
  user:
    name:  "{{ item.name }}"
    group: "{{ item.group }}"
    groups: "{{ item.groups }}"
    uid: "{{ item.uid }}"
    state: "{{ item.state }}"
  loop:
    - { name: 'admin1', group: 'admin', groups: 'nginx', uid:
        '1234', state: 'present' }
    - { name: 'dbadmin1', group: 'dbadmin', groups: 'postgres',
        uid: '4321', state: 'present' }
    - { name: 'user1', group: 'hadoop', groups: 'wheel', uid:
        '1067', state: 'present' }
    - { name: 'jose', group: 'admin', groups: 'wheel', uid:
        '9000', state: 'absent' }
```

Looking at the user `jose`, you may recognize that `state: 'absent'` deletes this user account, and you may be wondering why you need to include all the other parameters when you're just removing him. It's because this is a good place to keep documentation of important changes for audits or security compliance. By storing the roles in Git as your source of truth, you can go back and look at the old versions in Git if you later need to answer questions about why changes were made.

To deploy SSH keys for some of the users, you can use the same type of looping as in the last example.

```
- name: copy admin1 and dbadmin ssh keys
  authorized_key:
    user: "{{ item.user }}"
    key: "{{ item.key }}"
    state: "{{ item.state }}"
    comment: "{{ item.comment }}"
  loop:
    - { user: 'admin1', key: "{{ lookup('file', '/data/test_
        temp_key.pub'), state: 'present', comment: 'admin1 key' }
    - { user: 'dbadmin', key: "{{ lookup('file',
        '/data/vm_temp_key.pub'), state: 'absent',
        comment: 'dbadmin key' }
```

Here, we specify the `user`, how to find the key by using `lookup`, the `state`, and a `comment` describing the purpose of the key.

Installing packages

Package installation can vary depending on the packaging system you are using. You can use Ansible facts [8] to determine which module to use. Ansible does offer a generic module called package [9] that uses `ansible_pkg_mgr` and calls the proper package manager for the system. For example, if you're using Fedora, the package module will call the DNF package manager.

The package module will work if you're doing a simple installation of packages. If you're doing more complex work, you will have to use the correct module for your system. For example, if you want to ignore GPG keys and install all the security packages on a RHEL-based system, you need to use the yum module. You will have different options depending on your packaging module [10], but they usually offer more parameters than Ansible's generic package module.

Here is an example using the package module:

```
- name: install a package
  package:
    name: nginx
    state: installed
```

The following uses the yum module to install NGINX, disable `gpg_check` from the repo, ignore the repository's certificates, and skip any broken packages that might show up.

```
- name: install a package
  yum:
    name: nginx
    state: installed
    disable_gpg_check: yes
    validate_certs: no
    skip_broken: yes
```

Here is an example using Apt [11]. The Apt module tells Ansible to uninstall NGINX and not update the cache:

```
- name: install a package
  apt:
    name: nginx
    state: absent
    update_cache: no
```

You can use `loop` when installing packages, but they are processed individually if you pass a list:

```
  - name:
      - nginx
      - postgresql-server
      - ansible
      - httpd
```

NOTE: Make sure you know the correct name of the package you want in the package manager you're using. Some names change depending on the package manager.

Starting services

Much like packages, Ansible has different modules to start services [12]. Like in our previous example, where we used the package module to do a general installation of packages, the service [13] module does similar work with services, including with systemd and Upstart. (Check the module's documentation for a complete list.) Here is an example:

```
  - name: start nginx
    service:
      name: nginx
      state: started
```

You can use Ansible's service module if you are just starting and stopping applications and don't need anything more sophisticated. But, like with the yum module, if you need more options, you will need to use the systemd module. For example, if you modify systemd files, then you need to do a `daemon-reload`, the service module won't work for that; you will have to use the systemd module.

```
  - name: reload postgresql for new configuration and reload daemon
    systemd:
      name: postgresql
      state: reload
      daemon-reload: yes
```

This is a great starting point, but it can become cumbersome because the service will always reload/restart. This a good place to use a handler [14].

If you used best practices and created your role using `ansible-galaxy init "role name"`, then you should have the full directory structure [15]. You can include the code above inside the `handlers/main.yml` and call it when you make a change with the application. For example:

```
handlers/main.yml

  - name: reload postgresql for new configuration and reload
          daemon
    systemd:
      name: postgresql
      state: reload
      daemon-reload: yes
```

This is the task that calls the handler:

```
  - name: configure postgresql
    template:
      src: postgresql.service.j2
      dest: /usr/lib/systemd/system/postgresql.service
    notify: reload postgresql for new configuration and reload
            daemon
```

It configures PostgreSQL by changing the systemd file, but instead of defining the restart in the tasks (like before), it calls the handler to do the restart at the end of the run. This is a good way to configure your application and keep it idempotent since the handler only runs when a task changes—not in the middle of your configuration.

The previous example uses the template module [16] and a Jinja2 file [17]. One of the most wonderful things about configuring applications with Ansible is using templates. You can configure a whole file like `postgresql.service` with the full configuration you require. But, instead of changing every line, you can use variables and define the options somewhere else. This will let you change any variable at any time and be more versatile. For example:

```
[database]
DB_TYPE  = "{{ gitea_db }}"
HOST     = "{{ ansible_fqdn}}:3306"
NAME     = gitea
USER     = gitea
PASSWD   = "{{ gitea_db_passwd }}"
SSL_MODE = disable
PATH     = "{{ gitea_db_dir }}/gitea.db
```

This configures the database options on the file `app.ini` for Gitea [18]. This is similar to writing Ansible tasks, even though it is a configuration file, and makes it easy to define variables and make changes. This can be expanded further if you are using group_vars [19], which allows you to define variables for all systems and specific groups (e.g., production vs. development). This makes it easier to manage variables, and you don't have to specify the same ones in every role.

Provisioning a system

We've gone over several things you can do with Ansible on your system, but we haven't yet discussed how to provision a system. Here's an example of provisioning a virtual machine (VM) with the OpenStack cloud solution.

```
  - name: create a VM in openstack
    osp_server:
      name: cloudera-namenode
      state: present
      cloud: openstack
```

```
    region_name: andromeda
    image: 923569a-c777-4g52-t3y9-cxvhl86zx345
    flavor_ram: 20146
    flavor: big
    auto_ip: yes
    volumes: cloudera-namenode
```

All OpenStack modules start with `os`, which makes it easier to find them. The above configuration uses the osp-server module, which lets you add or remove an instance. It includes the name of the VM, its state, its cloud options, and how it authenticates to the API. More information about cloud.yml [20] is available in the OpenStack docs, but if you don't want to use cloud.yml, you can use a dictionary that lists your credentials using the `auth` option. If you want to delete the VM, just change `state:` to `absent`.

Say you have a list of servers you shut down because you couldn't figure out how to get the applications working, and you want to start them again. You can use `os_server_action` to restart them (or rebuild them if you want to start from scratch).

Here is an example that starts the server and tells the modules the name of the instance:

```
- name: restart some servers
  os_server_action:
    action: start
    cloud: openstack
    region_name: andromeda
    server: cloudera-namenode
```

Most OpenStack modules use similar options. Therefore, to rebuild the server, we can use the same options but change the `action` to `rebuild` and add the `image` we want it to use:

```
os_server_action:
  action: rebuild
  image: 923569a-c777-4g52-t3y9-cxvhl86zx345
```

Doing other things

There are modules for a lot of system admin tasks, but what should you do if there isn't one for what you are trying to do? Use the shell [21] and command [22] modules, which allow you to run any command just like you do on the command line. Here's an example using the OpenStack CLI [23]:

```
- name: run an opencli command
  command: "openstack hypervisor list"
```

They are so many ways you can do daily sysadmin tasks with Ansible. Using this automation tool can transform your hardest task into a simple solution, save you time, and make your work days shorter and more relaxed.

Links

[1] https://github.com/ansible/ansible/issues/18006
[2] https://github.com/abaez/ansible-role-user/issues/1
[3] http://docs.ansible.com/ansible/latest/user_guide/intro_getting_started.html
[4] http://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html
[5] https://github.com/ansible/lightbulb
[6] https://docs.ansible.com/ansible/latest/reference_appendices/faq.html
[7] https://docs.ansible.com/ansible/latest/modules/modules_by_category.html
[8] https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#information-discovered-from-systems-facts
[9] http://docs.ansible.com/ansible/latest/modules/package_module.html
[10] http://docs.ansible.com/ansible/latest/modules/list_of_packaging_modules.html
[11] https://docs.ansible.com/ansible/latest/modules/apt_module.html
[12] http://docs.ansible.com/ansible/latest/modules/list_of_system_modules.html
[13] http://docs.ansible.com/ansible/latest/modules/service_module.html#service-module
[14] https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html#handlers-running-operations-on-change
[15] http://docs.ansible.com/ansible/latest/user_guide/playbooks_best_practices.html#directory-layout
[16] https://docs.ansible.com/ansible/latest/modules/template_module.html
[17] https://docs.ansible.com/ansible/latest/user_guide/playbooks_templating.html
[18] https://gitea.io/en-us/
[19] https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-examples
[20] https://docs.openstack.org/python-openstackclient/pike/configuration/index.html
[21] https://docs.openstack.org/python-openstackclient/pike/configuration/index.html
[22] https://docs.ansible.com/ansible/latest/modules/command_module.html
[23] https://docs.openstack.org/python-openstackclient/pike/

Author

Jlozadad is a Ansible Consultant. I'm from Carolina, Puerto Rico and I love IT & Gaming. Spend Most of my time playing video games, looking at open source software and laughing.

Adapted from "A sysadmin's guide to Ansible: How to simplify tasks" on Opensource.com, published under a Creative Commons Attribution Share-Alike 4.0 International License at *https://opensource.com/article/18/7/sysadmin-tasks-ansible*.

Testing Ansible roles **with Molecule**

BY JAIRO DA SILVA JUNIOR

Learn how to automate your verifications using Python.

TEST TECHNIQUES play an important role in software development, and this is no different when we are talking about Infrastructure as Code (IaC).

Developers are always testing, and constant feedback is necessary to drive development. If it takes too long to get feedback on a change, your steps might be too large, making errors hard to spot. Baby steps and fast feedback are the essence of TDD (test-driven development). But how do you apply this approach to the development of ad hoc playbooks or roles?

When you're developing an automation, a typical workflow would start with a new virtual machine. I will use Vagrant [1] to illustrate this idea, but you could use libvirt [2], Docker [3], VirtualBox [4], or VMware [5], an instance in a private or public cloud, or a virtual machine provisioned in your data center hypervisor (oVirt [6], Xen [7], or VMware, for example).

When deciding which virtual machine to use, balance feedback speed and similarity with your real target environment.

The minimal start point with Vagrant would be:

```
vagrant init centos/7 # or any other box
```

Then add Ansible provisioning to your **Vagrantfile**:

```
config.vm.provision "ansible" do |ansible|
  ansible.playbook = "playbook.yml"
end
```

In the end, your workflow would be:

1. `vagrant up`
2. Edit playbook.
3. `vagrant provision`
4. `vagrant ssh` to verify VM state.
5. Repeat steps 2 to 4.

Occasionally, the VM should be destroyed and brought up again (`vagrant destroy -f; vagrant up`) to increase the reliability of your playbook (i.e., to test if your automation is working end-to-end).

Although this is a good workflow, you're still doing all the hard work of connecting to the VM and verifying that everything is working as expected.

When tests are not automated, you'll face issues similar to those when you do not automate your infrastructure.

Luckily, tools like Testinfra [8] and Goss [9] can help automate these verifications.

I will focus on Testinfra, as it is written in Python and is the default verifier for Molecule. The idea is pretty simple: Automate your verifications using Python:

```
def test_nginx_is_installed(host):
    nginx = host.package("nginx")
    assert nginx.is_installed
    assert nginx.version.startswith("1.2")


def test_nginx_running_and_enabled(host):
    nginx = host.service("nginx")
    assert nginx.is_running
    assert nginx.is_enabled
```

In a development environment, this script would connect to the target host using SSH (just like Ansible) to perform the above verifications (package presence/version and service state):

```
py.test --connection=ssh --hosts=server
```

In short, during infrastructure automation development, the challenge is to provision new infrastructure, execute playbooks against them, and verify that your changes reflect the state you declared in your playbooks.

- What can Testinfra verify?
  - Infrastructure is up and running from the user's point of view (e.g., HTTPD or Nginx is answering requests, and MariaDB or PostgreSQL is handling SQL queries).
  - OS service is started and enabled
  - A process is listening on a specific port
  - A process is answering requests
  - Configuration files were correctly copied or generated from templates
  - Virtually anything you do to ensure that your server state is correct
- What safeties do these automated tests provide?
  - Perform complex changes or introduce new features without breaking existing behavior (e.g., it still works in RHEL-based distributions after adding support for Debian-based systems).
  - Refactor/improve the codebase when new versions of Ansible are released and new best practices are introduced.

What we've done with Vagrant, Ansible, and Testinfra so far is easily mapped to the steps described in the Four-Phase Test [10] pattern—a way to structure tests that makes the test objective clear. It is composed of the following phases: *Setup*, *Exercise*, *Verify*, and *Teardown*:

- **Setup**: Prepares the environment for the test execution (e.g., spins up new virtual machines):

  ```
  vagrant up
  ```

- **Exercise**: Effectively executes the code against the system under test (i.e., Ansible playbook):

  ```
  vagrant provision
  ```

- **Verify**: Verifies the previous step output:

  `py.test` (with Testinfra)

- **Teardown**: Returns to the state prior to *Setup:*

  ```
  vagrant destroy
  ```

The same idea we used for an ad hoc playbook could be applied to role development and testing, but do you need to do all these steps every time you develop something new? What if you want to use containers, or an OpenStack, instead of Vagrant? What if you'd rather use Goss than Testinfra? How do you run this continuously for every change in your code? Is there a more simple and fast way to develop our playbooks and roles with automated tests?

Molecule

Molecule [11] helps develop roles using tests. The tool can even initialize a new role with test cases: `molecule init role -role-name foo`

Molecule is flexible enough to allow you to use different drivers for infrastructure provisioning, including Docker, Vagrant, OpenStack, GCE, EC2, and Azure. It also allows the use of different server verification tools, including Testinfra and Goss.

Its commands ease the execution of tasks commonly used during development workflow:

- `lint` - Executes **yaml-lint**, **ansible-lint**, and **flake8**, reporting failure if there are issues
- `syntax` - Verifies the role for syntax errors
- `create` - Creates an instance with the configured driver
- `prepare` - Configures instances with preparation playbooks
- `converge` - Executes playbooks targeting hosts
- `idempotence` - Executes a playbook twice and fails in case of changes in the second run (non-idempotent)
- `verify` - Execute server state verification tools (testinfra or goss)
- `destroy` - Destroys instances
- `test` - Executes all the previous steps

The `login` command can be used to connect to provisioned servers for troubleshooting purposes.

Step by step

How do you go from no tests at all to a decent codebase being executed for every change/commit?

1. `virtualenv` (optional)

The `virtualenv` tool creates isolated environments, while `virtualenvwrapper` is a collection of extensions that facilitate the use of `virtualenv`.

These tools prevent dependencies and conflicts between Molecule and other Python packages in your machine.

```
sudo pip install virtualenvwrapper
export WORKON_HOME=~/envs
source /usr/local/bin/virtualenvwrapper.sh
mkvirtualenv mocule
```

2. Molecule

Install Molecule with the Docker driver:

```
pip install molecule ansible docker
```

Generate a new role with test scenarios:

```
molecule init role -r role_name
```

or for existing roles:

```
molecule init scenario -r my-role
```

All the necessary configuration is generated with your role, and you need only write test cases using Testinfra:

```
import os

import testinfra.utils.ansible_runner

testinfra_hosts = testinfra.utils.ansible_runner.AnsibleRunner(
    os.environ['MOLECULE_INVENTORY_FILE']).get_hosts('all')

def test_jboss_running_and_enabled(host):
    jboss = host.service('wildfly')

    assert jboss.is_enabled

def test_jboss_listening_http(host):
    socket = host.socket('tcp://0.0.0.0:8080')

    assert socket.is_listening

def test_mgmt_user_authentication(host):
    command = """curl --digest -L -D - http://localhost:9990/ \
                 management -u ansible:ansible"""

    cmd = host.run(command)

    assert 'HTTP/1.1 200 OK' in cmd.stdout
```

This example test case for a Wildfly role verifies that OS service is enabled, a process is listening in port 8080, and authentication is properly configurated.

Coding these tests is straightforward, and you basically need to think about an automated way to verify something.

You are already writing tests when you log into a machine targeted by your playbook, or when you build verifications for your monitoring/alerting systems. This knowledge will contribute to building something with the Testinfra API [12] or using a system command.

CI

Continuously executing your Molecule tests is simple. The example above works for TravisCI with the Docker driver, but it could be easily adapted for any CI server and any infrastructure drivers supported by Molecule.

```
---
sudo: required
language: python
services:
  - docker
before_install:
  - sudo apt-get -qq update
  - pip install molecule
  - pip install docker
script:
  - molecule test
```

Visit Travis CI [13] for sample output.

Links

[1] https://github.com/hashicorp/vagrant
[2] https://libvirt.org/
[3] https://www.docker.com/
[4] https://www.virtualbox.org/
[5] https://www.vmware.com/
[6] https://ovirt.org/
[7] https://www.xenproject.org/
[8] https://testinfra.readthedocs.io/en/latest/
[9] https://github.com/aelsabbahy/goss
[10] http://xunitpatterns.com/Four Phase Test.html
[11] https://molecule.readthedocs.io/en/latest/
[12] https://testinfra.readthedocs.io/en/latest/
[13] https://travis-ci.org/jairojunior/ansible-role-jboss/builds/345731738

Author

Jairo da Silva Junior—Developer, speaker at DevOps conferences, open source contributor, occasional writer, and obsessed with tests and automation. Can't live without CLI tools.

Adapted from "Testing Ansible roles with Molecule" on Opensource.com, published under a Creative Commons Attribution Share-Alike 4.0 International License at *https://opensource.com/article/18/12/testing-ansible-roles-molecule*.

Using Ansible for deploying serverless applications

BY RYAN SCOTT BROWN

Serverless is another step in the direction of managed services and plays nice with Ansible's agentless architecture.

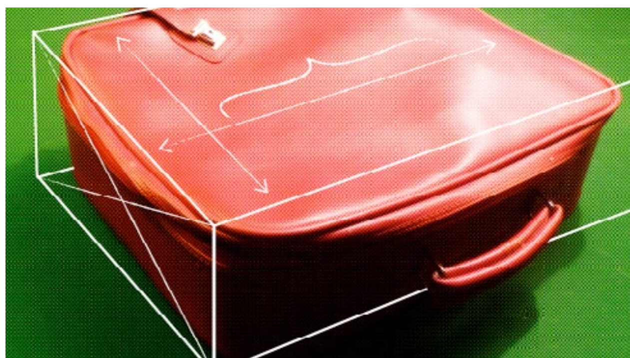
ANSIBLE [1] is designed as the simplest deployment tool that actually works. What that means is that it's not a full programming language. You write YAML templates that define tasks and list whatever tasks you need to automate your job.

Most people think of Ansible as a souped-up version of "SSH in a 'for' loop," and that's true for simple use cases. But really Ansible is about *tasks*, not about SSH. For a lot of use cases, we connect via SSH but also support things like Windows Remote Management (WinRM) for Windows machines, different protocols for network devices, and the HTTPS APIs that are the lingua franca of cloud services.

In a cloud, Ansible can operate on two separate layers: the control plane and the on-instance resources. The control plane consists of everything *not* running on the OS. This includes setting up networks, spawning instances, provisioning higher-level services like Amazon's S3 or DynamoDB, and everything else you need to keep your cloud infrastructure secure and serving customers.

On-instance work is what you already know Ansible for: starting and stopping services, templating config files, installing packages, and everything else OS-related that you can do over SSH.

Now, what about serverless [2]? Depending who you ask, serverless is either the ultimate extension of the continued rush to the public cloud or a wildly new paradigm where everything is an API call, and it's never been done before.



Ansible takes the first view. Before "serverless" was a term of art, users had to manage and provision EC2 instances, virtual private cloud (VPC) networks, and everything else. Serverless is another step in the direction of managed services and plays nice with Ansible's agentless architecture.

Before we go into a Lambda [3] example, let's look at a simpler task for provisioning a CloudFormation stack:

```
- name: Build network
  cloudformation:
    stack_name: prod-vpc
    state: present
    template: base_vpc.yml
```

Writing a task like this takes just a couple minutes, but it brings the last semi-manual step involved in building your infrastructure—clicking "Create Stack"—into a playbook with everything else. Now your VPC is just another task you can call when building up a new region.

Since cloud providers are the real source of truth when it comes to what's really happening in your account, Ansible has a number of ways to pull that back and use the IDs, names, and other parameters to filter and query running instances or networks. Take for example the cloudformation_facts module that we can use to get the subnet IDs, network ranges, and other data back out of the template we just created.

```
- name: Pull all new resources back in as a variable
  cloudformation_facts:
    stack_name: prod-vpc
  register: network_stack
```

For serverless applications, you'll definitely need a complement of Lambda functions in addition to any other DynamoDB tables, S3 buckets, and whatever else. Fortunately, by using the lambda modules, Lambda functions can be created in the same way as the stack from the last tasks:

```
- lambda:
    name: sendReportMail
    zip_file: "{{ deployment_package }}"
    runtime: python3.6
    handler: report.send
    memory_size: 1024
    role: "{{ iam_exec_role }}"
  register: new_function
```

If you have another tool that you prefer for shipping the serverless parts of your application, that works as well. The open source Serverless Framework [4] has its own Ansible module that will work just as well:

```
- serverless:
    service_path: '{{ project_dir }}'
    stage: dev
  register: sls
- name: Serverless uses CloudFormation under the hood, so you
        can easily pull info back into Ansible
  cloudformation_facts:
    stack_name: "{{ sls.service_name }}"
  register: sls_facts
```

That's not quite everything you need, since the serverless project also must exist, and that's where you'll do the heavy lifting of defining your functions and event sources. For this example, we'll make a single function that responds to HTTP requests. The Serverless Framework uses YAML as its config language (as does Ansible), so this should look familiar.

```
# serverless.yml
service: fakeservice

provider:
  name: aws
  runtime: python3.6

functions:
  main:
    handler: test_function.handler
    events:
      - http:
          path: /
          method: get
```

Links

[1] https://www.ansible.com/
[2] https://en.wikipedia.org/wiki/Serverless_computing
[3] https://aws.amazon.com/lambda/
[4] https://serverless.com/

Author

Ryan is a Senior Software Engineer and spends most of his time on cloud-adjacent Open Source tooling, including Ansible and the Serverless Framework.

Adapted from "Using Ansible for deploying serverless applications" on Opensource.com, published under a Creative Commons Attribution Share-Alike 4.0 International License at *https://opensource.com/article/17/8/ansible-serverless-applications*.

4 Ansible playbooks you should try

BY DANIEL OH

Streamline and tighten automation processes in complex IT environments with these Ansible playbooks.

In a complex IT environment, even the smallest tasks can seem to take forever. Sprawling systems are hard to develop, deploy, and maintain. Business demands only increase complexity, and IT teams struggle with management, availability, and cost.

How do you address this complexity and while meeting today's business demands? There is no doubt that Ansible [1] can improve your current processes, migrate applications for better optimization, and provide a single language for DevOps practices across your organization.

More importantly, you can declare configurations through Ansible playbooks [2], but they 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 ensure the configurations of your remote systems are in spec. Because the Ansible playbooks are configuration, deployment, and orchestration language, they can describe a policy you want your remote systems to enforce or a set of steps in a general IT process.

Here are four Ansible playbooks that you should try to further customize and configure how your automation works.

Managing Kubernetes objects

When you perform CRUD operations on Kubernetes [3] objects, Ansible playbooks enable you to quickly and easily access the full range of Kubernetes APIs through the OpenShift Python client. The following playbook snippets show you how to create specific Kubernetes namespace and service objects:

```
- name: Create a k8s namespace
  k8s:
    name: mynamespace
    api_version: v1
    kind: Namespace
    state: present

- name: Create a Service object from an inline definition
  k8s:
    state: present
    definition:
      apiVersion: v1
      kind: Service
      metadata:
        name: web
        namespace: mynamespace
        labels:
          app: galaxy
          service: web
      spec:
        selector:
```

```
          app: galaxy
          service: web
        ports:
        - protocol: TCP
          targetPort: 8000
          name: port-8000-tcp
          port: 8000

- name: Create a Service object by reading the definition from
        a file
  k8s:
    state: present
    src: /mynamespace/service.yml

# Passing the object definition from a file
- name: Create a Deployment by reading the definition from a
        local file
  k8s:
    state: present
    src: /mynamespace/deployment.yml
```

Mitigate critical security concerns like Meltdown and Spectre

In the first week of January 2018, two flaws were announced: Meltdown and Spectre [4]. Both involved the hardware at the heart of more or less every computing device on the planet: the processor. There is a great in-depth review of the two flaws here [5]. While Meltdown and Spectre are not completely mitigated, the following playbook snippets show how to easily deploy the patches for Windows:

```
- name: Patch Windows systems against Meltdown and Spectre
  hosts: "{{ target_hosts | default('all') }}"

  vars:
    reboot_after_update: no
    registry_keys:
      - path: HKLM:\SYSTEM\CurrentControlSet\Control\Session
              Manager\Memory Management
        name: FeatureSettingsOverride
        data: 0
        type: dword

      - path: HKLM:\SYSTEM\CurrentControlSet\Control\Session
              Manager\Memory Management
        name: FeatureSettingsOverrideMask
        data: 3
        type: dword

      # https://support.microsoft.com/en-us/help/4072699
      - path: HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\
              QualityCompat
        name: cadca5fe-87d3-4b96-b7fb-a231484277cc
        type: dword
        data: '0x00000000'

  tasks:
    - name: Install security updates
      win_updates:
        category_names:
          - SecurityUpdates
      notify: reboot windows system

    - name: Enable kernel protections
      win_regedit:
        path: "{{ item.path }}"
        name: "{{ item.name }}"
        data: "{{ item.data }}"
        type: "{{ item.type }}"
      with_items: "{{ registry_keys }}"

  handlers:
    - name: reboot windows system
      win_reboot:
        shutdown_timeout: 3600
        reboot_timeout: 3600
      when: reboot_after_update
```

You can also find other playbooks for Linux [6].

Integrating a CI/CD process with Jenkins

Jenkins [7] is a well-known tool for implementing CI/CD. Shell scripts are commonly used for provisioning environments or to deploy apps during the pipeline flow. Although this could work, it is cumbersome to maintain and reuse scripts in the long run. The following playbook snippets show how to provision infrastructure in a Continuous Integration/Continuous Delivery (CI/CD) process using a Jenkins Pipeline [8].

```
---
- name: Deploy Jenkins CI
hosts: jenkins_server
remote_user: vagrant
become: yes

roles:
  - geerlingguy.repo-epel
  - geerlingguy.jenkins
  - geerlingguy.git
  - tecris.maven
  - geerlingguy.ansible

- name: Deploy Nexus Server
hosts: nexus_server
remote_user: vagrant
become: yes
```

```
roles:
  - geerlingguy.java
  - savoirfairelinux.nexus3-oss

- name: Deploy Sonar Server
hosts: sonar_server
remote_user: vagrant
become: yes

roles:
  - wtanaka.unzip
  - zanini.sonar

- name: On Premises CentOS
hosts: app_server
remote_user: vagrant
become: yes

roles:
  - jenkins-keys-config
```

Starting a service mesh with Istio

With a cloud platform, developers must use microservices to architect for portability. Meanwhile, operators are managing extremely large hybrid and multi-cloud deployments. The service mesh with Istio [9] lets you connect, secure, control, and observe services instead of developers through a dedicated infrastructure such as an Envoy sidecar container. The following playbook snippets show how to install Istio locally on your machine:

```
---

# Whether the cluster is an Openshift (ocp) or upstream
  Kubernetes (k8s) cluster
cluster_flavour: ocp

istio:
  # Install istio with or without istio-auth module
  auth: false

  # A set of add-ons to install, for example kiali
  addon: []

  # The names of the samples that should be installed as well.
  # The available samples are in the istio_simple_samples variable
  # In addition to the values in istio_simple_samples,
    'bookinfo' can also be specified
  samples: []

  # Whether or not to open apps in the browser
  open_apps: false

  # Whether to delete resources that might exist from previous
    Istio installations
  delete_resources: false
```

Conclusion

You can find full sets of playbooks that illustrate many of these techniques in the ansible-examples repository [10]. I recommend looking at these in another tab as you go along.

Hopefully, these tips and snippets of Ansible playbooks have provided some interesting ways to use and extend your automation journey.

Links

[1] https://www.ansible.com/
[2] https://docs.ansible.com/ansible/devel/user_guide/playbooks.html
[3] https://kubernetes.io/
[4] https://meltdownattack.com/
[5] https://access.redhat.com/security/vulnerabilities/speculativeexecution
[6] https://github.com/ansible/ansible-lockdown/blob/master/meltdown-spectre-linux.yml
[7] https://jenkins.io/
[8] https://jenkins.io/doc/book/pipeline/
[9] https://istio.io/
[10] https://github.com/ansible/ansible-examples

Author

Daniel Oh—DevOps Evangelist, CNCF Ambassador, Developer, Public Speaker, Writer, Opensource.com Author

Adapted from “4 Ansible playbooks you should try” on Opensource.com, published under a Creative Commons Attribution Share-Alike 4.0 International License at *https://opensource.com/article/18/8/ansible-playbooks-you-should-try*.

GET INVOLVED

If you find these articles useful, get involved! Your feedback helps improve the status quo for all things DevOps.

Contribute to the Opensource.com DevOps resource collection, and join the team of DevOps practitioners and enthusiasts who want to share the open source stories happening in the world of IT.

The Open Source DevOps team is looking for writers, curators, and others who can help us explore the intersection of open source and DevOps. We're especially interested in stories on the following topics:

- DevOps practical how to's
- DevOps and open source
- DevOps and talent
- DevOps and culture
- DevSecOps/rugged software

Learn more about the Opensource.com DevOps team: https://opensource.com/devops-team

ADDITIONAL RESOURCES

The open source guide to DevOps monitoring tools

This free download for sysadmin observability tools includes analysis of open source monitoring, log aggregation, alerting/visualizations, and distributed tracing tools.

Download it now: The open source guide to DevOps monitoring tools

The ultimate DevOps hiring guide

This free download provides advice, tactics, and information about the state of DevOps hiring for both job seekers and hiring managers.

Download it now: The ultimate DevOps hiring guide

The Open Organization Guide to IT Culture Change

In The Open Organization Guide to IT Culture Change, more than 25 contributors from open communities, companies, and projects offer hard-won lessons and practical advice on how to create an open IT department that can deliver better, faster results and unparalleled business value.

Download it now: The Open Organization Guide to IT Culture Change

WRITE FOR US

Would you like to write for Opensource.com? Our editorial calendar includes upcoming themes, community columns, and topic suggestions: https://opensource.com/calendar

Learn more about writing for Opensource.com at: https://opensource.com/writers

We're always looking for open source-related articles on the following topics:

Big data: Open source big data tools, stories, communities, and news.

Command-line tips: Tricks and tips for the Linux command-line.

Containers and Kubernetes: Getting started with containers, best practices, security, news, projects, and case studies.

Education: Open source projects, tools, solutions, and resources for educators, students, and the classroom.

Geek culture: Open source-related geek culture stories.

Hardware: Open source hardware projects, maker culture, new products, howtos, and tutorials.

Machine learning and AI: Open source tools, programs, projects and howtos for machine learning and artificial intelligence.

Programming: Share your favorite scripts, tips for getting started, tricks for developers, tutorials, and tell us about your favorite programming languages and communities.

Security: Tips and tricks for securing your systems, best practices, checklists, tutorials and tools, case studies, and security-related project updates.

Keep in touch!

Sign up to receive roundups of our best articles, giveaway alerts, and community announcements.

Visit opensource.com/email-newsletter to subscribe.