About Ansible

Ansible is an IT automation tool. It can configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates.

Ansible's main goals are simplicity and ease-of-use. It also has a strong focus on security and reliability, featuring a minimum of moving parts, usage of OpenSSH for transport (with an accelerated socket mode and pull modes as alternatives), and a language that is designed around auditability by humans–even those not familiar with the program.

Installation

RPMs for RHEL7 are available from the Extras channel.

RPMs for RHEL6 are available from yum for EPEL 6 and currently supported Fedora distributions.

Ansible will also have RPMs/YUM-repo available at <*https://releases.ansible.com/ansible/rpms/*.

Ansible version 2.4 can manage earlier operating systems that contain Python 2.6 or higher.

You can also build an RPM yourself. From the root of a checkout or tarball, use the `make rpm` command to build an RPM you can distribute and install.

```
$ git clone git://github.com/ansible/ansible.git
$ cd ./ansible
$ make rpm
$ sudo rpm -Uvh ./rpm-build/ansible-*.noarch.rpm
```

Latest Releases Via Apt (Ubuntu)

Ubuntu builds are available in a PPA here.

To configure the PPA on your machine and install ansible run these commands:

```
$ sudo apt-get update
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

Inventory

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible’s inventory, which defaults to being saved in the location `/etc/ansible/hosts`. You can specify a different inventory file using the `-i <path>` option on the command line.

Not only is this inventory configurable, but you can also use multiple inventory files at the same time and pull inventory from dynamic or cloud sources or different formats (YAML, ini, etc), as described in Dynamic Inventory. Introduced in version 2.4, Ansible has inventory plugins to make this flexible and customizable.

Hosts and Groups

The inventory file can be in one of many formats, depending on the inventory plugins you have. For this example, the format for `/etc/ansible/hosts` is an INI-like (one of Ansible’s defaults) and looks like this:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

Host Variables

As described above, it is easy to assign variables to hosts that will be used later in playbooks:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

Group Variables

Variables can also be applied to an entire group at once:

The INI way:

```
[atlanta]
host1
host2

[atlanta:vars]
ntp_server=ntp.atlanta.example.com
proxy=proxy.atlanta.example.com
```

Playbooks

Playbooks are Ansible's 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. If Ansible modules are the tools in your workshop, playbooks are your instruction manuals, and your inventory of hosts are your raw material.

At a basic level, playbooks can be used to manage configurations of and deployments to remote machines. At a more advanced level, they can sequence multi-tier rollouts involving rolling updates, and can delegate actions to other hosts, interacting with monitoring servers and load balancers along the way.

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

YAML

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a “hash” or a “dictionary”. So, we need to know how to write lists and dictionaries in YAML.

There’s another small quirk to YAML. All YAML files (regardless of their association with Ansible or not) can optionally begin with `---`and end with `...`. This is part of the YAML format and indicates the start and end of a document.

All members of a list are lines beginning at the same indentation level starting with a `"- "` (a dash and a space):

```
---
# A list of tasty fruits
fruits:
    - Apple
    - Orange
    - Strawberry
    - Mango
...
```

A dictionary is represented in a simple `key: value` form (the colon must be followed by a space):

```
# An employee record
martin:
    name: Martin D'vloper
    job: Developer
    skill: Elite
```

Modules

Modules (also referred to as “task plugins” or “library plugins”) are the ones that do the actual work in ansible, they are what gets executed in each playbook task. But you can also run a single one using the ‘ansible’ command.

Let’s review how we execute three different modules from the command line:

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

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

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

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

Which can be abbreviated to:

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

Another way to pass arguments to a module is using yaml syntax also called ‘complex args’

```
- name: restart webserver
  service:
    name: httpd
    state: restarted
```

See all lists of the Modules available in Ansible here.

Handlers

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

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

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

```
- name: template configuration file
  template: src=template.j2 dest=/etc/foo.conf
  notify:
     - restart memcached
     - restart apache
```

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

Here's an example handlers section:

```
handlers:
    - name: restart memcached
      service: name=memcached state=restarted
    - name: restart apache
      service: name=apache state=restarted
```

Loops

Often you'll want to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached.

Standard Loops

To save some typing, repeated tasks can be written in short-hand like so:

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  with_items:
     - testuser1
     - testuser2
```

If you have defined a YAML list in a variables file, or the 'vars' section, you can also do:

```
with_items: "{{ somelist }}"
```

The above would be the equivalent of:

```
- name: add user testuser1
  user:
    name: "testuser1"
    state: present
    groups: "wheel"
- name: add user testuser2
  user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

The yum and apt modules use with_items to execute fewer package manager transactions.

Note that the types of items you iterate over with ‘with_items’ do not have to be simple lists of strings. If you have a list of hashes, you can reference subkeys using things like:

```
- name: add several users
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  with_items:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Also be aware that when combining *when* with *with_items* (or any other loop statement), the *when* statement is processed separately for each item. See The When Statement for an example.

Loops are actually a combination of things *with_* + *lookup()*, so any lookup plugin can be used as a source for a loop, ‘items’ is lookup.

Please note that `with_items` flattens the first depth of the list it is provided and can yield unexpected results if you pass a list which is composed of lists. You can work around this by wrapping your nested list inside a list:

```
# This will run debug three times since the list is flattened
- debug:
    msg: "{{ item }}"
  vars:
    nested_list:
      - - one
        - two
        - three
  with_items: "{{ nested_list }}"

# This will run debug once with the three items
- debug:
    msg: "{{ item }}"
  vars:
    nested_list:
      - - one
```

```
        - two
        - three
  with_items:
    - "{{ nested_list }}"
```

Nested Loops

Loops can be nested as well:

```
- name: give users access to multiple databases
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  with_nested:
    - [ 'alice', 'bob' ]
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

As with the case of ‘with_items’ above, you can use previously defined variables.:

```
- name: here, 'users' contains the above list of employees
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  with_nested:
    - "{{ users }}"
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

Code Reuse: Roles

Roles are ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

Example project structure:

```
site.yml
webservers.yml
fooservers.yml
roles/
   common/
     tasks/
     handlers/
     files/
     templates/
     vars/
```

```
      defaults/
      meta/
    webservers/
      tasks/
      defaults/
      meta/
```

Roles expect files to be in certain directory names. Roles must include at least one of these directories, however it is perfectly fine to exclude any which are not being used. When in use, each directory must contain a `main.yml` file, which contains the relevant content:

- `tasks` - contains the main list of tasks to be executed by the role.
- `handlers` - contains handlers, which may be used by this role or even anywhere outside this role.
- `defaults` - default variables for the role (see Variables for more information).
- `vars` - other variables for the role (see Variables for more information).
- `files` - contains files which can be deployed via this role.
- `templates` - contains templates which can be deployed via this role.
- `meta` - defines some meta data for this role. See below for more details.

Other YAML files may be included in certain directories. For example, it is common practice to have platform-specific tasks included from the `tasks/main.yml` file:

```
# roles/example/tasks/main.yml
- name: added in 2.4, previouslly you used 'include'
  import_tasks: redhat.yml
  when: ansible_os_platform|lower == 'redhat'
- import_tasks: debian.yml
  when: ansible_os_platform|lower == 'debian'

# roles/example/tasks/redhat.yml
- yum:
    name: "httpd"
    state: present

# roles/example/tasks/debian.yml
- apt:
    name: "apache2"
    state: present
```

Roles may also include modules and other plugin types. For more information, please refer to the Embedding Modules and Plugins In Roles section below.

Using Roles

The classic (original) way to use roles is via the `roles:` option for a given play:

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

include

Includes a file with a list of plays or tasks to be executed in the current playbook. Files with a list of plays can only be included at the top level, lists of tasks can only be included where tasks normally run (in play).

```
# include a play after another play
- hosts: localhost
  tasks:
    - debug:
        msg: "play1"

- include: otherplays.yml


# include task list in play
- hosts: all
  tasks:
    - debug:
        msg: task1

    - include: stuff.yml

    - debug:
        msg: task10

# dyanmic include task list in play
- hosts: all
  tasks:
    - debug:
        msg: task1

    - include: "{{ hostvar }}.yml"
      static: no
      when: hostvar is defined
```

Jinja2 Filters

Filters in Jinja2 are a way of transforming template expressions from one kind of data into another. Jinja2 ships with many of these. See builtin filters in the official Jinja2 template documentation.

TIP: YAML syntax requires that if you start a value with `{{ foo }}` you quote the whole line, since it wants to be sure you aren't trying to start a YAML dictionary. This is covered on the YAML Syntax page.

This won't work:

```
- hosts: app_servers
  vars:
      app_path: {{ base_path }}/22
```

Do it like this and you'll be fine:

```
- hosts: app_servers
  vars:
       app_path: "{{ base_path }}/22"
```

Templating

Ansible uses Jinja2 templating to enable dynamic expressions and access to variables. Ansible greatly expands the number of filters and tests available, as well as adding a new plugin type: lookups.

Please note that all templating happens on the Ansible controller before the task is sent and executed on the target machine. This is done to minimize the requirements on the target (jinja2 is only required on the controller) and to be able to pass the minimal information needed for the task, so the target machine does not need a copy of all the data that the controller has access to.

Ansible Galaxy

Ansible Galaxy refers to the Galaxy website where users can share roles, and to a command line tool for installing, creating and managing roles.

s a free site for finding, downloading, and sharing community developed roles. Downloading roles from Galaxy is a great way to jumpstart your automation projects.

You can also use the site to share roles that you create. By authenticating with the site using your GitHub account, you're able to *import*roles, making them available to the Ansible community. Imported roles become available in the Galaxy search index and visible on the site, allowing users to discover and download them.

Learn more by viewing the About page.

The command line tool

The `ansible-galaxy` command comes bundled with Ansible, and you can use it to install roles from Galaxy or directly from a git based SCM. You can also use it to create a new role, remove roles, or perform tasks on the Galaxy website.

The command line tool by default communicates with the Galaxy website API using the server address *https://galaxy.ansible.com*. Since the Galaxy project is an open source project, you may be running your own internal Galaxy server and wish to override the default server address. You can do this using the *–server* option or by setting the Galaxy server value in your *ansible.cfg* file. For information on setting the value in *ansible.cfg* visit Galaxy Settings.

Installing Roles

Use the `ansible-galaxy` command to download roles from the Galaxy website

```
$ ansible-galaxy install username.role_name
```

roles_path

Be aware that by default Ansible downloads roles to the path specified by the environment variable `ANSIBLE_ROLES_PATH`. This can be set to a series of directories (i.e. */etc/ansible/roles:~/.ansible/roles*), in which case the first writable path will be used. When Ansible is first installed it defaults to */etc/ansible/roles*, which requires *root* privileges.

You can override this by setting the environment variable in your session, defining *roles_path* in an *ansible.cfg* file, or by using the *–roles-path* option. The following provides an example of using *–roles-path* to install the role into the current working directory:

```
$ ansible-galaxy install --roles-path . geerlingguy.apache
```