Working with dynamic inventory

If your Ansible inventory fluctuates over time, with hosts spinning up and shutting down in response to business demands, the static inventory solutions described in How to build your inventory will not serve your needs. You may need to track hosts from multiple sources: cloud providers, LDAP, Cobbler, and/or enterprise CMDB systems.

Ansible integrates all of these options via a dynamic external inventory system. Ansible supports two ways to connect with external inventory: Inventory Plugins and inventory scripts.

Inventory plugins take advantage of the most recent updates to the Ansible core code. We recommend plugins over scripts for dynamic inventory. You can write your own plugin to connect to additional dynamic inventory sources.

You can still use inventory scripts if you choose. When we implemented inventory plugins, we ensured backwards compatibility via the script inventory plugin. The examples below illustrate how to use inventory scripts.

If you would like a GUI for handling dynamic inventory, the Red Hat Ansible Tower inventory database syncs with all your dynamic inventory sources, provides web and REST access to the results, and offers a graphical inventory editor. With a database record of all of your hosts, you can correlate past event history and see which hosts have had failures on their last playbook runs.

Inventory script example: Cobbler

Ansible integrates seamlessly with Cobbler, a Linux installation server originally written by Michael DeHaan and now led by James Cammarata, who works for Ansible.

While primarily used to kickoff OS installations and manage DHCP and DNS, Cobbler has a generic layer that can represent data for multiple configuration management systems (even at the same time) and serve as a 'lightweight CMDB'.

To tie your Ansible inventory to Cobbler, copy this script to `/etc/ansible` and `chmod +x` the file. Run `cobblerd` any time you use Ansible and use the `-i` command line option (e.g. `-i /etc/ansible/cobbler.py`) to communicate with Cobbler using Cobbler's XMLRPC API.

Add a `cobbler.ini` file in `/etc/ansible` so Ansible knows where the Cobbler server is and some cache improvements can be used. For example:

```
[cobbler]

# Set Cobbler's hostname or IP address
host = http://127.0.0.1/cobbler_api

# API calls to Cobbler can be slow. For this reason, we cache the results of an API
# call. Set this to the path you want cache files to be written to. Two files
# will be written to this directory:
#   - ansible-cobbler.cache
#   - ansible-cobbler.index

cache_path = /tmp

# The number of seconds a cache file is considered valid. After this many
# seconds, a new API call will be made, and the cache file will be updated.

cache_max_age = 900
```

First test the script by running `/etc/ansible/cobbler.py` directly. You should see some JSON data output, but it may not have anything in it just yet.

Let's explore what this does. In Cobbler, assume a scenario somewhat like the following:

```
cobbler profile add --name=webserver --distro=CentOS6-x86_64
cobbler profile edit --name=webserver --mgmt-classes="webserver" --ksmeta="a=2 b=3"
cobbler system edit --name=foo --dns-name="foo.example.com" --mgmt-classes="atlanta" --ksmeta="c=4"
cobbler system edit --name=bar --dns-name="bar.example.com" --mgmt-classes="atlanta" --ksmeta="c=5"
```

In the example above, the system 'foo.example.com' are addressable by ansible directly, but are also addressable when using the group names 'webserver' or 'atlanta'. Since Ansible uses SSH, it contacts system foo over 'foo.example.com', only, never just 'foo'. Similarly, if you try "ansible foo" it wouldn't find the system... but "ansible 'foo*'" would, because the system DNS name starts with 'foo'.

The script provides more than host and group info. In addition, as a bonus, when the 'setup' module is run (which happens automatically when using playbooks), the variables 'a', 'b', and 'c' will all be auto-populated in the templates:

```
# file: /srv/motd.j2
Welcome, I am templated with a value of a={{ a }}, b={{ b }}, and c={{ c }}
```

Which could be executed just like this:

```
ansible webserver -m setup
ansible webserver -m template -a "src=/tmp/motd.j2 dest=/etc/motd"
```

Note

The name 'webserver' came from Cobbler, as did the variables for the config file. You can still pass in your own variables like normal in Ansible, but variables from the external inventory script will override any that have the same name.

So, with the template above (`motd.j2`), this would result in the following data being written to `/etc/motd` for system 'foo':

```
Welcome, I am templated with a value of a=2, b=3, and c=4
```

And on system 'bar' (bar.example.com):

```
Welcome, I am templated with a value of a=2, b=3, and c=5
```

And technically, though there is no major good reason to do it, this also works too:

```
ansible webserver -m shell -a "echo {{ a }}"
```

So in other words, you can use those variables in arguments/actions as well.

Inventory script example: AWS EC2

If you use Amazon Web Services EC2, maintaining an inventory file might not be the best approach, because hosts may come and go over time, be managed by external applications, or you might even be using AWS autoscaling. For this reason, you can use the EC2 external inventory script.

You can use this script in one of two ways. The easiest is to use Ansible's `-i` command line option and specify the path to the script after marking it executable:

```
ansible -i ec2.py -u ubuntu us-east-1d -m ping
```

The second option is to copy the script to */etc/ansible/hosts* and *chmod +x* it. You must also copy the ec2.ini file to */etc/ansible/ec2.ini*. Then you can run ansible as you would normally.

To make a successful API call to AWS, you must configure Boto (the Python interface to AWS). You can do this in several ways available, but the simplest is to export two environment variables:

```
export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'
```

You can test the script by itself to make sure your config is correct:

```
cd contrib/inventory
./ec2.py --list
```

After a few moments, you should see your entire EC2 inventory across all regions in JSON.

If you use Boto profiles to manage multiple AWS accounts, you can pass `--profile PROFILE` name to the `ec2.py` script. An example profile might be:

```
[profile dev]
aws_access_key_id = <dev access key>
aws_secret_access_key = <dev secret key>

[profile prod]
aws_access_key_id = <prod access key>
aws_secret_access_key = <prod secret key>
```

You can then run `ec2.py --profile prod` to get the inventory for the prod account, although this option is not supported by `ansible-playbook`. You can also use the `AWS_PROFILE` variable - for example: `AWS_PROFILE=prod ansible-playbook -i ec2.py myplaybook.yml`

Since each region requires its own API call, if you are only using a small set of regions, you can edit the `ec2.ini` file and comment out the regions you are not using.

There are other config options in `ec2.ini`, including cache control and destination variables. By default, the `ec2.ini` file is configured for **all Amazon cloud services**, but you can comment out any features that aren't applicable. For example, if you don't have `RDS` or `elasticache`, you can set them to `False`:

```
[ec2]
...

# To exclude RDS instances from the inventory, uncomment and set to False.
rds = False

# To exclude ElastiCache instances from the inventory, uncomment and set to False.
elasticache = False
...
```

At their heart, inventory files are simply a mapping from some name to a destination address. The default `ec2.ini` settings are configured for running Ansible from outside EC2 (from your laptop for example) – and this is not the most efficient way to manage EC2.

If you are running Ansible from within EC2, internal DNS names and IP addresses may make more sense than public DNS names. In this case, you can modify the `destination_variable` in `ec2.ini` to be the private DNS name of an instance. This is particularly important when running Ansible within a private subnet inside a VPC, where the only way to access an instance is via its private IP address. For VPC instances, *vpc_destination_variable* in `ec2.ini` provides a means of using which ever boto.ec2.instance variable makes the most sense for your use case.

The EC2 external inventory provides mappings to instances from several groups:

Global

All instances are in group `ec2`.

Instance ID

These are groups of one since instance IDs are unique. e.g. `i-00112233` `i-a1b1c1d1`

Region

A group of all instances in an AWS region. e.g. `us-east-1` `us-west-2`

Availability Zone

A group of all instances in an availability zone. e.g. `us-east-1a` `us-east-1b`

Security Group

Instances belong to one or more security groups. A group is created for each security group, with all characters except alphanumerics, converted to underscores (_). Each group is prefixed by `security_group_`. Currently, dashes (-) are also converted to underscores (_). You can change using the replace_dash_in_groups setting in ec2.ini (this has changed across several versions so check the ec2.ini for details). e.g. `security_group_default` `security_group_webservers` `security_group_Pete_s_Fancy_Group`

Tags

Each instance can have a variety of key/value pairs associated with it called Tags. The most common tag key is 'Name', though anything is possible. Each key/value pair is its own group of instances, again with special characters converted to underscores, in the format `tag_KEY_VALUE` e.g. `tag_Name_Web` can be used as is `tag_Name_redis-master-001` becomes `tag_Name_redis_master_001` `tag_aws_cloudformation_logical-id_WebServerGroup` becomes `tag_aws_cloudformation_logical_id_WebServerGroup`

When the Ansible is interacting with a specific server, the EC2 inventory script is called again with the `--host HOST` option. This looks up the HOST in the index cache to get the instance ID, and then makes an API call to AWS to get information about that specific instance. It then makes information about that instance available as variables to your playbooks. Each variable is prefixed by `ec2_`. Here are some of the variables available:

- ec2_architecture
- ec2_description
- ec2_dns_name
- ec2_id
- ec2_image_id

- ec2_instance_type
- ec2_ip_address
- ec2_kernel
- ec2_key_name
- ec2_launch_time
- ec2_monitored
- ec2_ownerId
- ec2_placement
- ec2_platform
- ec2_previous_state
- ec2_private_dns_name
- ec2_private_ip_address
- ec2_public_dns_name
- ec2_ramdisk
- ec2_region
- ec2_root_device_name
- ec2_root_device_type
- ec2_security_group_ids
- ec2_security_group_names
- ec2_spot_instance_request_id
- ec2_state
- ec2_state_code
- ec2_state_reason
- ec2_status
- ec2_subnet_id
- ec2_tag_Name
- ec2_tenancy
- ec2_virtualization_type
- ec2_vpc_id

Both `ec2_security_group_ids` and `ec2_security_group_names` are comma-separated lists of all security groups. Each EC2 tag is a variable in the format `ec2_tag_KEY`.

To see the complete list of variables available for an instance, run the script by itself:

```
cd contrib/inventory
./ec2.py --host ec2-12-12-12-12.compute-1.amazonaws.com
```

Note that the AWS inventory script will cache results to avoid repeated API calls, and this cache setting is configurable in ec2.ini. To explicitly clear the cache, you can run the ec2.py script with the `--refresh-cache` parameter:

```
./ec2.py --refresh-cache
```

Inventory script example: OpenStack

If you use an OpenStack-based cloud, instead of manually maintaining your own inventory file, you can use the `openstack_inventory.py` dynamic inventory to pull information about your compute instances directly from OpenStack.

You can download the latest version of the OpenStack inventory script here.

You can use the inventory script explicitly (by passing the *-i openstack_inventory.py* argument to Ansible) or implicitly (by placing the script at */etc/ansible/hosts*).

Explicit use of OpenStack inventory script

Download the latest version of the OpenStack dynamic inventory script and make it executable:

```
wget https://raw.githubusercontent.com/ansible/ansible/stable-2.9/contrib/inventory/openstack_inventory.py
chmod +x openstack_inventory.py
```

Note

Do not name it *openstack.py*. This name will conflict with imports from openstacksdk.

Source an OpenStack RC file:

```
source openstack.rc
```

Note

An OpenStack RC file contains the environment variables required by the client tools to establish a connection with the cloud provider, such as the authentication URL, user name, password and region name. For more information on how to download, create or source an OpenStack RC file, please refer to Set environment variables using the OpenStack RC file.

You can confirm the file has been successfully sourced by running a simple command, such as *nova list* and ensuring it return no errors.

Note

The OpenStack command line clients are required to run the *nova list* command. For more information on how to install them, please refer to Install the OpenStack command-line clients.

You can test the OpenStack dynamic inventory script manually to confirm it is working as expected:

```
./openstack_inventory.py --list
```

After a few moments you should see some JSON output with information about your compute instances.

Once you confirm the dynamic inventory script is working as expected, you can tell Ansible to use the *openstack_inventory.py* script as an inventory file, as illustrated below:

```
ansible -i openstack_inventory.py all -m ping
```

Implicit use of OpenStack inventory script

Download the latest version of the OpenStack dynamic inventory script, make it executable and copy it to */etc/ansible/hosts*:

```
wget https://raw.githubusercontent.com/ansible/ansible/stable-2.9/contrib/inventory/openstack_inventory.py
chmod +x openstack_inventory.py
sudo cp openstack_inventory.py /etc/ansible/hosts
```

Download the sample configuration file, modify it to suit your needs and copy it to */etc/ansible/openstack.yml*:

```
wget https://raw.githubusercontent.com/ansible/ansible/stable-2.9/contrib/inventory/openstack.yml
vi openstack.yml
sudo cp openstack.yml /etc/ansible/
```

You can test the OpenStack dynamic inventory script manually to confirm it is working as expected:

```
/etc/ansible/hosts --list
```

After a few moments you should see some JSON output with information about your compute instances.

Refreshing the cache

Note that the OpenStack dynamic inventory script will cache results to avoid repeated API calls. To explicitly clear the cache, you can run the openstack_inventory.py (or hosts) script with the `--refresh` parameter:

```
./openstack_inventory.py --refresh --list
```

Other inventory scripts

You can find all included inventory scripts in the contrib/inventory directory. General usage is similar across all inventory scripts. You can also write your own inventory script.

Using inventory directories and multiple inventory sources

If the location given to `-i` in Ansible is a directory (or as so configured in `ansible.cfg`), Ansible can use multiple inventory sources at the same time. When doing so, it is possible to mix both dynamic and statically managed inventory sources in the same ansible run. Instant hybrid cloud!

In an inventory directory, executable files will be treated as dynamic inventory sources and most other files as static sources. Files which end with any of the following will be ignored:

```
~, .orig, .bak, .ini, .cfg, .retry, .pyc, .pyo
```

You can replace this list with your own selection by configuring an `inventory_ignore_extensions` list in ansible.cfg, or setting the `ANSIBLE_INVENTORY_IGNORE` environment variable. The value in either case should be a comma-separated list of patterns, as shown above.

Any `group_vars` and `host_vars` subdirectories in an inventory directory will be interpreted as expected, making inventory directories a powerful way to organize different sets of configurations. See Using multiple inventory sources for more information.

Static groups of dynamic groups

When defining groups of groups in the static inventory file, the child groups must also be defined in the static inventory file, or ansible will return an error. If you want to define a static group of dynamic child groups, define the dynamic groups as empty in the static inventory file. For example:

```
[tag_Name_staging_foo]

[tag_Name_staging_bar]

[staging:children]
tag_Name_staging_foo
tag_Name_staging_bar
```

See also

How to build your inventory

All about static inventory files

Mailing List

Questions? Help? Ideas? Stop by the list on Google Groups

irc.freenode.net

#ansible IRC chat channel

Previous Next