Amazon Web Services Guide¶

Introduction¶

Ansible contains a number of modules for controlling Amazon Web Services (AWS). The purpose of this section is to explain how to put Ansible modules together (and use inventory scripts) to use Ansible in AWS context.

Requirements for the AWS modules are minimal.

All of the modules require and are tested against recent versions of boto. You'll need this Python module installed on your control machine. Boto can be installed from your OS distribution or python's "pip install boto".

Whereas classically ansible will execute tasks in its host loop against multiple remote machines, most cloud-control steps occur on your local machine with reference to the regions to control.

In your playbook steps we'll typically be using the following pattern for provisioning steps:

```
- hosts: localhost
  connection: local
  gather_facts: False
  tasks:
    - ...
```

Authentication¶

Authentication with the AWS-related modules is handled by either specifying your access and secret key as ENV variables or module arguments.

For environment variables:

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

For storing these in a vars_file, ideally encrypted with ansible-vault:

```
---
ec2_access_key: "--REMOVED--"
ec2_secret_key: "--REMOVED--"
```

Note that if you store your credentials in vars_file, you need to refer to them in each AWS-module. For example:

```
- ec2
  aws_access_key: "{{ec2_access_key}}"
  aws_secret_key: "{{ec2_secret_key}}"
  image: "..."
```

Provisioning¶

The ec2 module provisions and de-provisions instances within EC2.

An example of making sure there are only 5 instances tagged 'Demo' in EC2 follows.

In the example below, the "exact_count" of instances is set to 5. This means if there are 0 instances already existing, then 5 new instances would be created. If there were 2 instances, only 3 would be created, and if there were 8 instances, 3 instances would be terminated.

What is being counted is specified by the "count_tag" parameter. The parameter "instance_tags" is used to apply tags to the newly created instance.:

```
# demo_setup.yml

- hosts: localhost
  connection: local
  gather_facts: False

  tasks:

    - name: Provision a set of instances
      ec2:
         key_name: my_key
         group: test
         instance_type: t2.micro
         image: "{{ ami_id }}"
         wait: true
         exact_count: 5
         count_tag:
            Name: Demo
         instance_tags:
            Name: Demo
      register: ec2
```

The data about what instances are created is being saved by the "register" keyword in the variable named "ec2".

From this, we'll use the add_host module to dynamically create a host group consisting of these new instances. This facilitates performing configuration actions on the hosts immediately in a subsequent task.:

```
# demo_setup.yml

- hosts: localhost
  connection: local
  gather_facts: False

  tasks:

    - name: Provision a set of instances
      ec2:
         key_name: my_key
         group: test
         instance_type: t2.micro
         image: "{{ ami_id }}"
         wait: true
         exact_count: 5
         count_tag:
            Name: Demo
         instance_tags:
            Name: Demo
      register: ec2

   - name: Add all instance public IPs to host group
     add_host: hostname={{ item.public_ip }} groups=ec2hosts
     loop: "{{ ec2.instances }}"
```

With the host group now created, a second play at the bottom of the same provisioning playbook file might now have some configuration steps:

```
# demo_setup.yml

- name: Provision a set of instances
  hosts: localhost
  # ... AS ABOVE ...

- hosts: ec2hosts
  name: configuration play
  user: ec2-user
  gather_facts: true

  tasks:

     - name: Check NTP service
       service: name=ntpd state=started
```

Host Inventory¶

Once your nodes are spun up, you'll probably want to talk to them again. With a cloud setup, it's best to not maintain a static list of cloud hostnames in text files. Rather, the best way to handle this is to use the ec2 dynamic inventory script. See Working With Dynamic Inventory (../user_guide/intro_dynamic_inventory.html#dynamic-inventory).

This will also dynamically select nodes that were even created outside of Ansible, and allow Ansible to manage them.

See Working With Dynamic Inventory (../user_guide/intro_dynamic_inventory.html#dynamic-inventory) for how to use this, then return to this chapter.

Tags And Groups And Variables¶

When using the ec2 inventory script, hosts automatically appear in groups based on how they are tagged in EC2.

For instance, if a host is given the "class" tag with the value of "webserver", it will be automatically discoverable via a dynamic group like so:

```
- hosts: tag_class_webserver
  tasks:
    - ping
```

Using this philosophy can be a great way to keep systems separated by the function they perform.

In this example, if we wanted to define variables that are automatically applied to each machine tagged with the 'class' of 'webserver', 'group_vars' in ansible can be used. See Splitting Out Host and Group Specific Data (../user_guide/intro_inventory.html#splitting-out-vars).

Similar groups are available for regions and other classifications, and can be similarly assigned variables using the same mechanism.

Autoscaling with Ansible Pull¶

Amazon Autoscaling features automatically increase or decrease capacity based on load. There are also Ansible modules shown in the cloud documentation that can configure autoscaling policy.

When nodes come online, it may not be sufficient to wait for the next cycle of an ansible command to come along and configure that node.

To do this, pre-bake machine images which contain the necessary ansible-pull invocation. Ansible-pull is a command line tool that fetches a playbook from a git server and runs it locally.

One of the challenges of this approach is that there needs to be a centralized way to store data about the results of pull commands in an autoscaling context. For this reason, the autoscaling solution provided below in the next section can be a better approach.

Read ansible-pull (../cli/ansible-pull.html#ansible-pull) for more information on pull-mode playbooks.

Autoscaling with Ansible Tower¶

Ansible Tower (../reference_appendices/tower.html#ansible-tower) also contains a very nice feature for auto-scaling use cases. In this mode, a simple curl script can call a defined URL and the server will "dial out" to the requester and configure an instance that is spinning up. This can be a great way to reconfigure ephemeral nodes. See the Tower install and product documentation for more details.

A benefit of using the callback in Tower over pull mode is that job results are still centrally recorded and less information has to be shared with remote hosts.

Ansible With (And Versus) CloudFormation¶

CloudFormation is a Amazon technology for defining a cloud stack as a JSON document.

Ansible modules provide an easier to use interface than CloudFormation in many examples, without defining a complex JSON document. This is recommended for most users.

However, for users that have decided to use CloudFormation, there is an Ansible module that can be used to apply a CloudFormation template to Amazon.

When using Ansible with CloudFormation, typically Ansible will be used with a tool like Packer to build images, and CloudFormation will launch those images, or ansible will be invoked through user data once the image comes online, or a combination of the two.

Please see the examples in the Ansible CloudFormation module for more details.

AWS Image Building With Ansible¶

Many users may want to have images boot to a more complete configuration rather than configuring them entirely after instantiation. To do this, one of many programs can be used with Ansible playbooks to define and upload a base image, which will then get its own AMI ID for usage with the ec2 module or other Ansible AWS modules such as ec2_asg or the cloudformation module. Possible tools include Packer, aminator, and Ansible's ec2_ami module.

Generally speaking, we find most users using Packer.

See the Packer documentation of the Ansible local Packer provisioner (https://www.packer.io/docs/provisioners/ansible-local.html) and Ansible remote Packer provisioner (https://www.packer.io/docs/provisioners/ansible.html).

If you do not want to adopt Packer at this time, configuring a base-image with Ansible after provisioning (as shown above) is acceptable.

Next Steps: Explore Modules¶

Ansible ships with lots of modules for configuring a wide array of EC2 services. Browse the “Cloud” category of the module documentation for a full list with examples.

See also

All modules (../modules/list_of_all_modules.html#all-modules)
All the documentation for Ansible modules
Working With Playbooks (../user_guide/playbooks.html#working-with-playbooks)
An introduction to playbooks
Delegation, Rolling Updates, and Local Actions (../user_guide/playbooks_delegation.html#playbooks-delegation)
Delegation, useful for working with loud balancers, clouds, and locally executed steps.
User Mailing List (https://groups.google.com/group/ansible-devel)
Have a question? Stop by the google group!
irc.freenode.net (http://irc.freenode.net)
#ansible IRC chat channel