Linux Academy

Manging AWS with Ansible

Study Guide

Stosh Oldham

stosh@linuxacademy.com

July 20, 2019

Contents

Ansible and Amazon Web Services

Essential Configuration

In this section, we'll go over some of the specifics of configuring an Ansible control host to work with a provided AWS account. This includes both key areas of configuration for Ansible, and configurations in the AWS console. We will also talk about some utilities that will help you maximize automation.

Ansible Configuration

First, we need to go over some notes with regard to Ansible configuration.

Using AWS Free Tier for Practice

Before we get too deep into configuration, it is important that you are aware of the fact that AWS can have significant costs affiliated with it. However, presuming you have a new AWS account, you can manage to interact with most elements we will cover without accruing any charges on your AWS account.

About Free Tier

Most of the topics we cover in this course involve working with EC2 instances. All new AWS accounts are allowed up to 750 hours of EC2 compute resources per month, for the first 12 months of services. Additionally, a limited amount of S3 storage is also available for free in the first year of an AWS account's life. Naturally, these limits are subject to change. See https://aws.amazon.com/free/ for the most up-to-date information regarding AWS free tier.

The Sandbox

Bear in mind that Linux Academy offers the Cloud Playground, which gives students access to a sandbox AWS environment, and is included as part of Linux Academy subscriptions!

Check out Linux Academy's Cloud Playground Introduction for more information on how it works and how to use it.

Ansible Configurations

There are really only a couple of additional requirements to using your Ansible control node for managing an AWS account. The first is ensuring that you have installed Python Boto on the control server. AWS modules require this package in order to properly function. Note that certain modules may require different versions of Boto (version 2 or 3 as of this writing).

Installing Boto

The package may be installed using your distribution's package manager. For example, on CentOS:

```
sudo yum install python-boto sudo yum install python-boto3
```

You may also use `pip` (the Python package installer) if you prefer.

`sudo pip install boto` `sudo pip install boto3`

Our Target Server

Typically, we write playbooks that target the server where we wish to perform configuration. For example:

```
- hosts: webservers
  become: yes
  tasks:
    - ....
```

It is important to be aware that any plays performing the AWS configuration generally execute against the control node. So the AWS configuration plays will target `localhost` in many of the scenarios we will be looking at. To that end, it is important to configure `localhost` for use by Ansible as part of any setup. Of course, if we want to perform a task on a particular EC2 instance, we would target that node instead.

Inventory Considerations

Paying homage to the fact that most modules manipulating the AWS console are run with the control node as the target host, the inventory will stay pretty simple. Given that we create new EC2 instances within Ansible, it may be practical to modify our inventory if we wish to connect directly to those instances and perform tasks targeting the instances using Ansible. We will look at the `add_host` module once we start working with EC2 instances in the course.

Where Things Get Dynamic

It is worth pointing out that this scenario may also leverage a dynamic inventory. There is a stock Python script provided in the Ansible documentation that can get you started if you want to pursue this. Noted that the dynamic inventory is more valuable in scenarios where new instances are being created outside of the scope of Ansible.

Documentation References
Inventory Script Example

Accessing the AWS Console with Ansible

Now that our control node is configured for use with the AWS console, we need to provide a means of authentication for Ansible to log in.

Configuring EC2 SSH Access for Ansible

The focus of this course is about managing an AWS environment with Ansible which will include a number of EC2 instances. As discussed earlier, a number of plays will target the control host in order to interact with the AWS console. However, it can be also useful to manage EC2 instances directly using Ansible.

Standard Configuration

There is nothing particularly special about using Ansible against instantiated EC2 instances. In fact, it works by connecting over SSH to the instance using a pre-configured user, much like connecting to any other server. We can use the AWS generated keypair to do this. We will talk more on working with AWS keypairs through Ansible a bit later in the course.

Working with `ssh-agent`

`ssh-agent` is a helper tool that is part of the SSH suite. It is used to perform public key authentication with SSH. It is automatically invoked when a shell session is created on login. However, there are occasionally cases where working directly with the tool is necessary.

Using the Tool

Authenticating with a remote user via Ansible provides such a case. It is conceivable that there may be more than one key pair associated with instances in a given AWS environment. If you need to use a authentication key pair that does not belong to Ansible, you may use `ssh-agent` to start a new Bash shell and then add the key pair for the new shell's session. The following example assumes the key pair is stored in `keyfile.pem`:

```
ssh-agent bash ssh-add keyfile.pem
```

Understanding AWS Console Access

Typical AWS Console access involves navigating to the AWS sign-in page and supplying credentials. From that point, you are able to click through a series of menus and modules to interact with various AWS components.

Why Are We Clicking?

Most of what we will look at in the web console will be for configuring our Ansible IAM user. To verify whether actions of our playbooks are creating new EC2 instances or manipulating certain properties of them.

We will be covering the noted type of activities at a high level through the console, but much of what we focus on is programmatic access to the console through Ansible. It is important to understand that the method of access for Ansible is not quite like that of the web console. Instead of using a username and password, we must create and utilize a set of keys to allow automated console access.

The AWS CLI interface

We can utilize key authentication using the AWS Python module known as `awscli`. This may be installed using pip as follows:

```
sudo pip install awscli
```

Once installed, we can run `awscli configure` in order to supply access keys. This provides another avenue of AWS console access.

Configuring IAM Users for Ansible

Before we can really get to work with Ansible, we need to take a quick look at how to create a new IAM user with the web console.

Managing IAM users through the IAM service menu is performed. It is possible to create users, group, and roles among other security related objects.

A Note on Permission Assignment

For the purposes of this course, we will only be interacting with the users menu to create a new IAM user for Ansible to use for console access. We will also perform some direct permission assignment for the sake of brevity. It is advisable to adequately plan permission and account security for any public facing architecture.

If you are looking for more on IAM, be sure to check out the IAM courses offered at Linux Academy.

Configuring IAM Access Keys

The AWS modules handle the intricacies of communicating with AWS. But to start that communication, there must be a means of logging into AWS with Ansible. This may be achieved by providing an Access Key as well as a Secret Key.

Where to Get Your Keys

The keys are created in the IAM service menu for a given user. This is typically accomplished by select `programmatic access` when creating your IAM user. This will automatically generate the access key as well as the secret key.

You will be provided authentication keys precisely one time after user creation in the web console. It is important to store the keys in a secure location to which you can maintain access. It is impossible to recover a secret key if it is lost. The only option is to generate a new key which may incur some overhead in management.

How to Use the Keys

In terms of providing the keys to Ansible, there are a few options. Passing the keys to the AWS Ansible modules is a common way is to provide the values as a shell variables like so:

```
export AWS_ACCESS_KEY_ID='ABC123'
export AWS_SECRET_ACCESS_KEY='XYZ123'
```

Most all modules will support using the noted environment variables. While it is possible to include the noted commands with the appropriate keys in their login scripts, this does come with an inherent security risk. The Access and Secret Keys can provide full access to an AWS account. Storing these values in plain text should be avoided. Frankly, the environment variable approach has been known to not work correctly in many versions of Ansible, even in more recent versions.

Alternatively, the values may be supplied as Ansible variables. If you choose to use this approach, you may use the Ansible vault to encrypt the values, which provides enhance security in case you wish to leave the key values on a file system. We will cover how to do this in a later section.

Understanding IAM Permissions with Regard to Ansible

Any IAM user starts with no permissions to any service. Part of user creation allows for permission assignment. You may also assign permissions to an IAM account later, so long as you are using another account with appropriate IAM permissions.

A Note on Permission Assignment (again)

Permissions may be applied through groups, roles, or even directly to the user. While best practice may suggest otherwise, in this course, we will make use of direct permission assignment for brevity.

As stated earlier, it is not the goal of this course to explain the intricacies of IAM. Linux Academy offers full courses that cover this topic in much more detail. For the purposes of this course, we will be assigning certain permissions for the sake working with certain Ansible modules.

How to Avoid HTTP 403

A common cause of error when working with AWS via Ansible is inadequate permission for the IAM user. These errors tend to manifest as HTTP 403 Forbidden status codes. It is crucially important to ensure that the `ansible` IAM user has the access to perform the necessary functions.

Conversely, it is just as important to provide only the required permission needed by a given user, and no more. It can be risky to provide full administrative access to a service account. It is important to appropriately plan user account security, and apply due security consideration when engineering infrastructure.

Securing Keys with Ansible Vault

Even when we restrict the permissions applied to our `ansible` user, having the login credentials fall into the wrong hands can lead to many undesirable effects. A bad actor who can log into your AWS environment might spin up numerous instances for a bot net, or some kind of compute intensive operation which can mean running up an astronomical AWS bill! This is but one of many reasons it is important to secure login credentials and reduce permissions to only what is required.

Maximum Security

We have spoken some on the topic of permission. When it comes to login credentials, it is rather straightforward: do not leave the credentials in an non-secure location. That may be difficult when you need to export environment variables on your system containing these credentials so that Ansible can use them.

You can partially protect credentials in those situations using minimal file system permissions (0600 for keys typically). But Ansible ships with some tooling that can provide more robust security. We can use `ansible-vault` to encrypt files containing keys, so that they are unusable to a person who does not have the vault password.

The best part is that the encryption is very simple. We create an Ansible variable file containing our credential information like so:

```
---
# File: awskeys.yml
# AWS Console Keys - TOP SECRET

AWS_ACCESS_KEY_ID: 123accesskey
AWS_SECRET_ACCESS_KEY: 456secretkey
```

Once the file exists, we can encrypt the file with the following command:

```
ansible-vault encrypt awskeys.yml
```

The tool will ask you to provide and then confirm a password for the vault.

At this point, we can include the variable file in our playbook like normal and we only need to supply the `--ask-vault-pass` flag to `ansible-playbook`. We will be prompted for our password to decrypt the file during playbook execution.

Key Modules

Now that our control node is configured and we have covered some basic security, it is time to start looking at the modules that we can use to shape the cloud!

EC2 Modules

Creating compute instances is at the heart of running any kind of infrastructure. In this section, we are going to take a look at a number of Ansible modules that allow us to work with EC2 instances.

Creating and Working with EC2 Instances

The `ec2` module is the primary means of interacting with EC2 instances.

Module: `ec2`

What does it do?

The `ec2` module is somewhat multipurpose, as it is used to both instantiate and to control EC2 instances.

The key functions of the EC2 module are: - Creating new EC2 instances - Terminating EC2 instances - Starting and stopping EC2 instances

Of course, in order for the module to function correctly, the Ansible control node must be appropriately authenticated to AWS.

Use cases

Using Ansible with the `ec2` module can take automation a step further! Instead of orchestrating software deployment and performing configuration management, all new stacks may be provisioned using Ansible. Ansible is able to create all new compute nodes using the `ec2` module, and then can customize each node with the more traditional Ansible modules such as `template`, `lineinfile`, and `package`!

Key arguments

The key parameters of the module are as follows:

- **key_name:** Which keypair to use for the instance
- **instance_type:** Type of instance to provision
- **image:** AMI to use when provisioning a new instance
- **wait:** Stops the module from returning until a target state is reached:
  - By default, Ansible will return as soon as the console command is issued.
- **wait_timeout:** How long the module will wait for work to complete, set to 300 seconds by default
- **count:** How many instances to create
- **vpc_subnet_id:** VPC to which a new instance will be provisioned
- **assign_pulbic_ip:** If yes, assigns a public IP to the instance

Example

```
tasks:
  - name: Launch instance
    ec2:
       key_name: "{{ keypair }}"
       instance_type: "{{ instance_type }}"
       image: "{{ image }}"
       wait: true
       region: "{{ region }}"
       vpc_subnet_id: subnet-xxxxxxx
       assign_public_ip: yes
    register: ec2
```

Documentation References
`ec2` module

As you start using Ansible to create EC2 instances, you will want to maintain a "handle" of sorts, in order to work with the new hosts in a generic way. An easy way to achieve this is by using the `add_host` module to update your inventory as hosts are created.

Module: `add_host`

What does it do?

The `add_host` module can update the in-memory inventory during play execution.

Use cases

If you are running plays that involve creating new hosts, such as new EC2 instances, the `add_host` module can update your inventory as the new hosts are created, allowing you to work with the host during play execution.

Key arguments

- **hostname:** The IP or DNS host name of your new host
- **groupname:** The Ansible group in which you would like your host
- You may also set host level variables as parameters for this module!

Example

```
- name: Add new instance to host group
  add_host:
    hostname: "{{ item.public_ip }}"
    groupname: launched
  loop: "{{ ec2.instances }}"
```

Documentation References
`add_host` module

Gathering Facts on EC2 Instances

One of Ansible's core strengths is the ability to maintain system runtime information through facts. When working with AWS, Ansible can generate facts for various aspects of AWS using specific modules. We are going to look at a handful of these fact modules throughout the course. While they each have slightly different attributes, you will find that the generally behave in a similar way. We will highlight the differences as we work through the material. Let's start with the `ec2_instance_facts` module.

Module: `ec2_instance_facts`

What does it do?

This module collects information on existing EC2 instances from the AWS web API using Boto.

Use cases

It can often be handy to know certain things about your instances as plays are running. For example, you may want to return the instance ID of all currently stopped instances to issue a start command. Or, perhaps, you want to check out CPU use all of your **t3.medium** instances that are running, to see if it's safe to shut down a few.

Key arguments

- **filters:** You can provide key value pairs that allow for a targeted fact collection.

Example

```
tasks:
  - name: Get instance facts
    ec2_instance_facts:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      filters:
        instance-state-name: running
    register: ec2_facts
  - debug:
      var: ec2_facts
```

Documentation References
`ec2_instance_facts` module
DescribeInstances API

EC2 Facts from Metadata

It is possible to access the AWS metadata from a particular EC2 instance with Ansible. The `ec2_metadata_facts` module can do this. But you must execute the module on the EC2 instance, as opposed out Ansible control node. This is unlike many of the AWS modules.

Module: `ec2_medtadata_facts`

What does it do?

This module fetches data from the instance metadata endpoint in EC2, as described in AWS's Instance Metadata and User Data page. The module must be called from within the EC2 instance itself.

Use cases

The data contained in the AWS metadata somewhat heavily overlaps with typical Ansible facts, as it contains information such as network addresses and block device information. The areas where you may find value are specific AWS configurations, such as the AMI ID used to create the instance, the instance security group, and information regarding IAM roles affiliated with the instance.

Of course, a lot of that information is returned in the `ec2_instance_facts` module as well. But if your play is executing against the instance and not your control node, you may not have readily configured access to the AWS console from that instance. It can be useful to keep facts on a per instance basis to have handy during playbook execution.

Key arguments

There are currently no arguments for this module. Facts gathered by the module may be used like any regular Ansible facts, once they are collected.

Example

```
- ec2_metadata_facts:

- debug:
    msg: "User data from this instance is: {{ ansible_ec2_user_data }}"
```

Documentation References
`ec2_metadata_facts` module

Working with AMIs

AMIs are a large part of EC2 configuration. Ansible has a few modules that allow us to automate management of AMIs.

Module: `ec2_ami`

What does it do?

As you might expect, the `ec2_ami` module is used to create and destroy AMIs.

Use cases

This module may be particularly handy as part of an AMI build management workflow where you can use Ansible to update your AMIs as new software versions arise.

Key arguments

- **instance_id:** Reference a specific EC2 instance for creating an AMI.
- **image_id:** Reference an existing AMI to deregister.
- **name:** Assign this name to the new AMI or reference an existing AMI.
- **state:** This will be *present* or *absent*, which corresponds to *register* or *deregister*. Present is the default.

- **device_mapping:** This takes a list of dictionaries defining custom mapping configuration for EBS devices.

Example

```
tasks:
  - name: Create new AMI
    ec2_ami:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      instance_id: i-xxxxxxx
      name: MyNewAMI
      wait: yes
      state: present
```

Documentation References
`ec2_ami` module

If you find yourself needing to copy an AMI to another region, you may use the `ec2-ami-copy` modules.

Module: `ec2-ami-copy`

What does it do?

This module copies a provided AMI to a specified region.

Use cases

You are only allowed to use a specific AMI in the region where it exists. If you are creating EC2 instances in different regions and would like to use the same AMI, you must copy the AMI to each region where it will be used.

Key arguments

- **region:** The region you are copying to
- **source_image_id:** The image ID of the AMI to be copied
- **source_region:** The region the AMI is being copied from
- **encrypted:** Set to “Yes” if the resulting AMI should be encrypted

Example

```
- ec2_ami_copy:
    source_region: us-east-1
    region: eu-west-1
    source_image_id: ami-xxxxxxx
    encrypted: yes
```

Documentation References
`ec2_ami_copy` module

Working with AMI Facts

Just like we can gather facts for our EC2 instances, we can also gather facts for AMIs.

Module: `ec2_ami_facts`

What does it do?

The AMI facts module works much like the EC2 facts modules, whereby it gathers information on the noted object. An important note about the AMI facts module is that it searches all public AMIs as well as private by default. I have found that, as of this writing, if you do not filter the results, the module will segfault and crash. The filter parameters helps you pare down what is returned. You can also search your AMIs only by providing the "owner" parameter.

Use cases

It is a handy way to get an image ID reference for AMIs meeting your filter criteria.

Key arguments

- **filters:** Allows for filtering of results
- **owner:** Only searches for AMIs belonging to a specific owner
- **image_ids:** A specific image ID to target

Example

```
tasks:
  - name: Get AMI facts
    ec2_ami_facts:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      filters:
        description: "*redhat*"
    register: ami_facts
  - name: Output information
    debug:
      var: ami_facts
```

Documentation References
`ec2_ami_facts` module

Working with EC2 Key Pairs

We can use Ansible to generate or delete key pairs for our EC2 instances using the `ec2_key` module.

Module: `ec2_key`

What does it do?

This creates or destroys AWS key pairs for EC2 instances. It is important to note that the ***private_key*** is returned by the module only when the key is first created.

Use cases

You can build an automated key refresh workflow using this module, which can enhance the security of your environment. You can also use this module to create unique key pairs for newly deployed instances.

Key arguments

- **name:** The name of the key pair
- **force:** Stops an existing keypair from being overwritten:
  - By default, an existing keypair will be overwritten if it is recreated. Setting force to “no” prevents that.
- **state:** Two states:
  - When ***present***, create the key pair.
  - When ***absent***, delete the key pair.

- **key_material:** Allows you to import an existing key pair.

Example

```
tasks:
  - name: remove old key
    ec2_key:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      name: new_keypair
      state: absent
  - name: create a new ec2 keypair
    ec2_key:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      name: new_keypair
    register: keypair
  - name: write new private key to file locally
    lineinfile:
      create: yes
      path: /home/ansible/new_private_key.pem
      line: "{{ keypair.key.private_key }}"
      mode: 0600
```

Documentation References
ec2_key_module

Managing CloudWatch Metric Alarms

With AMIs, Key pairs, and EC2 instance creation all covered, it is time to think about monitoring with CloudWatch. While we will not be going deep into all that is possible with AWS CloudWatch, we will briefly cover the `ec2_metric_alarm` module in Ansible.

Module: `ec2_metric_alarm`

What does it do?

Can create new CloudWatch alarms based on existing metrics.

Use cases

As you deploy new services, you are probably going to want to deploy certain metric alarms for appropriate monitoring. You can use this module to attach alarms to new EC2 instances as your plays create them.

Key arguments

There are several arguments required to make the metric alarm module actually work. We have broken them down into a few groups to hopefully make them easier to understand.

Basic Alarm Properties

- **name:** A name for the alarm
- **state:** Can be one of two:
  - Use *present* to register a new alarm, or *absent* to deregister an existing alarm.
- **metric:** What metric to sound an alarm about:
  - Note that the metric must already exist.
- **dimensions:** What the alarm is applied to, such as an EC2 instance ID
- **namespace:** Which category the alarm will appear under in CloudWatch

Alarm Detail Configuration Properties

- **statistic:** How our measured value is calculated:
  - It can be *Minimum*, *Maximum*, *Sum*, *Average*, *SampleCount*, or *Percentile*.
- **comparison:** How to compare the calculated value against the threshold value
- **period:** How often a metric aggregation is calculated
- **evaluation_periods:** The number of data points required to alarm
- **threshold:** The bound for triggering the alarm
- **unit:** The threshold's unit of measurement

Alarm Response Properties

- **alarm_actions:** A list of Amazon Resource Names of actions to take when the alarm status is "alarm"
- **ok_actions:** A list of Amazon Resource Names of actions to take when the alarm status is "ok"

Example

```
tasks:
  - name: Create a new CPU based Alarm
    ec2_metric_alarm:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      state: present
      name: "cpu-high"
      metric: "CPUUtilization"
      namespace: "AWS/EC2"
      statistic: Average
      comparison: ">="
      threshold: 90.0
      period: 300
      evaluation_periods: 3
      unit: "Percent"
      dimensions: {'InstanceId':'i-xxxxxxx'}
```

Documentation References
`ec2_metric_alarm` module

Working with EC2 Volumes

If you need to add additional EBS volumes to your EC2 instance, or manipulate existing EBS volumes, you can do so by using the `ec2_vol` module in Ansible.

Module: `ec2_vol`

What does it do?

This provides functionality for EBS volumes in a similar way to how the `ec2` module does for EC2 instances. The module is able to create and destroy volumes, but it is also able to manipulate existing volumes, and can attach or detach them to or from existing EC2 instances.

Use cases

If you find that a situation calls for an additional EBS volume not on an AMI you regularly use, you can use this module to add the extra volume. You can also use it to move a volume between EC2 instances if it is a data volume.

Key arguments

- **encrypted:** Allows for a new new volume to be encrypted when at rest, particularly important when combined with snapshots:
  - There is more in the next section on snapshots.

- **device_name:** System device name to use for the volume on an EC2 instance:
    - Note that sometimes the OS will override this name.
- **volume_size:** Size of the volume (GiB) to create
- **volume_type:** How to specify what kind of EBS volume to create:
    - *Standard* is the default, which equates to magnetic. Other options are *gp2*, *st1*, and *sc1*.
- **iops:** Sets an integer value for IOPs to associate with the volume
- **snapshot:** Specifies a specific snapshot to use:
    - There is more in the next section on snapshots.
- **delete_on_termination:** A boolean value, where if **true** will delete the volume upon instance termination

Example

```
tasks:
  - name: Attach Volume
    ec2_vol:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      instance: i-xxxxxxx
      id: vol-06fb0ae1104764110
      device_name: /dev/sda1
      delete_on_termination: yes
```

Documentation References
`ec2_vol` module

Working with Volume Snapshots in EC2

While we are on the topic of EBS volumes, a handy feature we have is the ability to snapshot a volume to make a point-in-time copy of it. We can achieve this in Ansible using the `ec2_snapshot` module.

Module: `ec2_snapshot`

What does it do?

The `ec2_snapshot` module takes snapshots of a given volume. There are also `ec2_snapshot_facts` and `ec2_snapshot_copy` modules. We will not go into detail about these, but they work in much the same way previous facts and copy modules have worked with other AWS objects. In particular, the `ec2_snapshot_facts` module collects relevant snapshot facts that may be used during playbook execution. The `ec2_snapshot_copy` module may be used to copy a snapshot to another AWS region.

Keep in mind that snapshots are not free. Having snapshots on your AWS account will begin to accrue costs!

Note that if your target volume is encrypted, your snapshot will be encrypted as well.

Use cases

A key use of snapshots is providing regular backups of data volumes. Even better, we can use the module to restore a snapshot to a particular host, in the event that we require use of the earlier noted backup.

Key arguments

- **volume_id:** Volume to base the snapshot on
- **description:** Plain language description as to why it was needed:
  - Any snapshot should have this
- **device_name:** To provide an *ec2 instance_id* and device name instead of a *specific volume_id* when taking a snapshot
- **snapshot_id:** Used to reference an existing snapshot
- **snapshot_tags:** Dictionary of tags to add to a snapshot
- **state:** Controls creation and deletion of snapshots respectively
  - May be *present* or *absent*.
- **last_snapshot_min_age:** Used for specifying the minimum age of an existing snapshot before taking a new one:
  - This value is in minutes.

Note that there is currently a bug with *last_snapshot_min_age*: See this issue

Example

```
tasks:
  - name: Gather snapshot facts
    ec2_snapshot_facts:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
  - name: Create Volume Snapshot
```

```
    ec2_snapshot:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      instance_id: i-xxxxxxx
      device_name: /dev/sda1
      state: present
      description: "On demand volume backup."
```

Documentation References
`ec2_snapshot` module
`ec2_snapshot_copy` module
`ec2_snapshot_facts` module

Creating and Removing EC2 Tags

Several of the modules we have looked at have had the ability to assign tags on creation. But there is a module that allows us to better control tags for a given EC2 object.

Module: `ec2_tag`

What does it do?

This module is used to manage tags on any EC2 resource.

Use cases

If you are using a role composed of any discrete tasks in the deployment of an environment, you may have a tagging task that will uniformly tag elements of a given system.

Key arguments

- **resource:** The EC2 Resource ID
- **state:** Can be one of *present*, *absent*, or *list*:
  - *present* will ensure the tag is there.
  - *absent* will remove the specified tag.
  - *list* is a special state that shows all tags affiliated with a specific EC2 object.
- **tags:** A dictionary of tags to be added to or removed from the resource in question

Example

```
tasks:
  - name: Gather tags
    ec2_tag:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      state: list
      resource: i-023f10d0235b10ec4
    register: tags
  - name: Display tags
    debug:
      var: tags.tags
  - name: Add tags to resource
    ec2_tag:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      resource: i-023f10d0235b10ec4
      tags:
        system: erp
        version: 1.2
      state: present
```

Documentation References
`ec2_tag` module

Working with VPCs

There are several modules that can assist with managing VPCs, we will be looking at a few key modules.

Working with EC2 VPC Security Groups

Part of managing EC2 instances is dealing with VPC security groups. Security group rules set can be created an managed using `ec2_group`.

Module: `ec2_group`

What does it do?

This module can create, delete, and maintain EC2 VPC security groups, including managing rule sets.

Use cases

As you are working within your VPC, you may need to make firewall changes at the VPC security group level to allow traffic into our out of a given group of EC2 instances.

Key arguments

- **vpc_id:** The VPC where the new group will exist
- **rules:** A list of inbound rules:
    - If no rules are provided, no rules will be set.
- **rules_egress:** Like rules, except for outbound traffic

Example

```
- name: Create new Security Group
  ec2_group:
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    region: us-east-1
    name: demoSG
    description: An example EC2 Security Group
    vpc_id: vpc-ed1b7f97
    rules:
      - proto: tcp
        from_port: 80
        to_port: 80
        cidr_ip: 0.0.0.0/0
      - proto: tcp
        from_port: 22
        to_port: 22
        cidr_ip: 10.0.0.0/8
      - proto: tcp
        from_port: 443
        to_port: 443
    rules_egress:
      - proto: tcp
        from_port: 80
        to_port: 80
        cidr_ip: 0.0.0.0/0
```

Documentation References
`ec2_group` module
`ec2_group_facts` module

Configuring a VPC

The primary module for creating a new VPC is `ec2_vpc_net`. There are also modules for VPC subnets, VPC internet gateways, VPC NAT gateways, and VPC route tables. In short, you can interact with the VPC service with Ansible in much the same way we did with the EC2 service. Granted, the use case for spinning up entire an entire VPC is a bit more narrow than EC2 instances, but the modules are certainly worth their salt.

Module: `ec2_vpc_net`

What does it do?

It creates a new VPC with the provided configuration values.

Use cases

The creation of additional VPCs depends heavily on a company's specific needs. Generally speaking, most organizations will not be running a significant number of VPCs. Each new VPC adds a fair amount of network complexity and overhead, which is only going to be worthwhile in certain situations. Given that disclaimer, a fair use case for VPC manipulation with Ansible is configuration management. With certain facets of VPC configuration, Ansible can help audit configuration.

Key arguments

- **name:** The name of the VPC
- **cidr_block:** The network size for your new VPC
- **region:** The region in which your VPC will be based
- **tags:** Tags for the VPC
- **tenancy:** Default or dedicated as it pertains to VPCs

Example

```
- name: create a VPC with dedicated tenancy
  ec2_vpc_net:
    name: NewVPC
    cidr_block: 10.10.0.0/16
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    region: us-east-1
    tags:
      client: LA
    tenancy: dedicated
```

Documentation References
`ec2_vpc_net` module
`ec2_vpc_subnet` module
`ec2_vpc_igw` module
`ec2_vpc_nat_gateway` module
`ec2_vpc_route_table` module

Facts for VPCs

For each an every VPC modules, there exists a corresponding facts modules as well. We will cover the core `ec2_vpc_net_facts` here.

Module: `ec2_vpc_net_facts`

What does it do?

This modules collects facts about VPCs affiliated with a provided region and IAM account.

Use cases

You can return information such as the tenancy type, vpc_id, and network configuration information on the VPCs under your account. You can use related modules such as `ec2_vpc_igw_facts` to collect facts on additional VPC resources.

Key arguments

- **vpc_ids:** You can target certain VPCs by ID for fact collection.

Example

```
- name: Get VPC facts
  ec2_vpc_net_facts:
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    region: us-east-1
  register: vpc_facts
- debug:
    var: vpc_facts
```

Documentation References
ec2_vpc_net_facts
ec2_vpc_subnet_facts
ec2_vpc_igw_facts
ec2_vpc_nat_gateway_facts
ec2_vpc_route_table_facts

S3 Modules

Storage tends to be pretty straightforward on the surface, but it can quickly get complicated. The aim in this section is to get started by looking at basic storage configuration in AWS with Ansible.

Working with S3 Objects

While a lot of working with AWS means working with EC2, there are other components of the cloud. One such component is storage in the form of AWS S3. We can work with S3 using the `aws_s3` module. It is important to remember that AWS S3 buckets must follow a strict DNS compliant naming convention, and must be uniquely named. Errors in naming your bucket do not present well in the module output.

Module: `aws_s3`

What does it do?

You can use this module to create, delete, and otherwise manage S3 buckets and their content.

Use cases

Using public facing storage may be a necessary component of an application. Being able to create and manipulate S3 buckets within Ansible can assist in fully automating and managing such an application.

Key arguments

- **bucket:** The S3 bucket name
- **mode:** The action being taken:
  - This can be ***get*** (download), ***put*** (upload), ***delobj*** (delete object), ***delete*** (remove bucket), and ***create*** (make bucket).
- **permission:** Permission to use when creating a new bucket:
  - This is set to ***private*** by default.
- **src:** Local file to upload
- **object:** An item to place in a bucket
- **object:** The S3 object location

Example

```
tasks:
  - name: Create new S3 bucket
    aws_s3:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      bucket: la-ansible-demo-bucket
      mode: create
  - name: Add file to bucket
    aws_s3:
```

```
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    region: us-east-1
    bucket: la-ansible-demo-bucket
    mode: put
    object: /testfile.txt
    src: /home/ansible/testfile.txt
```

Documentation References
`aws_s3` module

IAM Modules

A key use case for Ansible is controlling system configuration, which inevitably means managing user accounts and security to an extent.

Managing IAM Users and Groups

IAM accounts are a necessary construct of AWS, and Ansible allows us to manage a lot of the aspects of IAM via a single module called `iam`. However, we do have the `iam_group` and `iam_role` modules that provide specific functionality for groups and roles respectively.

The `iam_group` module overlaps with some functionality in terms of user creation, but it does allow for some group specific functionality, like purging users from groups and retrieving managed policies for specific groups. The `iam_role` module we will get into a bit more in its own section.

For the time being, let's just look at `iam`.

Module: `iam`

What does it do?

We can create IAM users, groups, and roles, and perform a number of management tasks using this module.

Use cases

A great use case of the `iam` module is for keeping security keys rotated. Ansible can automate much of the work that comes with keeping keys from getting too old.

Key arguments

- **name:** The name of the IAM resource being created
- **state:** One of three states: ***present***, ***absent***, or ***update***:
  - Note that roles may not be updated with the `iam` module (see `iam_role` module later)

- **iam_type:** The type of IAM resource we want:
    - This can be user, group, or role.
- **access_key_ids:** Keys to be operated on by module
- **access_key_state:** One of these four: *create*, *remove*, *active*, or *inactive*.

Example

```
tasks:
  - name: Create new IAM user
    iam:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      iam_type: user
      state: present
      name: demouser
      password: xxxxxxxxxx
  - name: Create new group with policy and add user
    iam_group:
      aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
      aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
      region: us-east-1
      name: testgroup
      managed_policy:
        - arn:aws:iam::aws:policy/AmazonEC2ReadOnlyAccess
      users:
        - demouser
      state: present
```

Documentation References
`iam` module
`iam_group` module

Working with IAM Roles

While we can do some basic role creation using the `iam_role` module, there is another module geared specifically for role creation called `iam_role`. The key benefit of the `iam_role` module is the ability to create IAM roles based on a policy file.

Module: `iam_role`

What does it do?

Creates and manages IAM roles.

Use cases

This module is particularly handy for creating roles from existing policy json files.

Key arguments

- **name:** Name of the role to create, and is required
- **state:** *Absent* or *present*, which correspond to remove or create
- **assume_role_policy_document:** Required for `state=present`:
  - It is a policy document that grants an entity permission to assume the role.
- **managed_policy:** A list of ARNs to apply to a role

Example

```
- name: Create a role based on the file policy.json with a description
  iam_role:
    name: mynewrole
    assume_role_policy_document: "{{ lookup('file','policy.json') }}"
    description: This is My New Role
```

Documentation References
`iam_role` module
`iam_role_facts` module

Bringing It All Together

Having a powerful script engine, such as Ansible, connected to an API that literally creates infrastructure in the cloud makes for all manner of exciting possibilities. We are going to take a look a spinning up a simple application using some of the tools we have discussed.

Use Case: Deploying a Web Site

Ansible is reasonably well known for it's ability to orchestrate complex deployments. We have surveyed using Ansible with key AWS functionality. We know that we can use Ansible for not only software deployment and configuration management, but for actually deploying infrastructure in the cloud!

Planning High Level Steps

The Scenario

For our example, we will build a relatively simple Web Server image creation workflow. The work flow will deploy a simple web site using AWS and the Apache HTTP Server, with an EBS Volume for the web root. Once deployed, the workflow will create a golden image for future deployment. We will assume that we are using an existing VPC, and that our network configuration and security group have been taken care of and appropriately configured.

High Level Steps

We have two high level tasks so far:

1. Deploy a website on an HTTP Server with dedicated EBS web root.
2. Create an image.

The first step to engineering a deployment is to start with the high level steps and slowly break down those large steps into smaller tasks. We will need to run `httpd`, which means we must have a server. That server requirement can be met with an EC2 instance.

Breaking It Down

So now we have:

1. Deploy an EC2 instance with dedicated EBS web root.
2. Install an HTTP Server and website on the EC2 instance.
3. Create an image.

We will need a local file store for our web content. There are a few ways to satisfy this need. While we might use one of Amazon's more sophisticated storage solutions in a production situation, we will use a standard EBS volume to keep our example simple.

1. Create an EBS Volume for web root.
2. Deploy an EC2 instance using new volume.
3. Install an HTTP Server and website on the EC2 instance.
4. Create an image.

Once we have a Server and a place to store our web content, we will need a way to access our server and make configuration changes. For this, we will need to import a key pair we on new instance.

1. Create an EBS Volume for web root.
2. Import an EC2 key pair.
3. Deploy an EC2 instance using new volume and key.
4. Install an HTTP Server and website on the EC2 instance.
5. Create an image.

Using our key, we can configure the server for use with Ansible by updating our inventory. Then, using Ansible, we can install the Apache HTTP server and push our sample web content to it.

1. Create an EBS Volume for web root.
2. Create new key pair.
3. Deploy an EC2 Instance using new volume and key.
4. Configure Ansible Inventory for the new instance.
5. Install an HTTP Server, and web content, on the EC2 instance with Ansible.
6. Create an image.

We started with two very high level tasks and have broken them down into six more granular tasks. Each task we have now roughly corresponds to a single module in Ansible, with the exception of step five which is going to be a few standard Ansible tasks that are not AWS specific. In the next section, we can look at which modules can perform each task.

Understanding Key Tasks

In this section, we are starting with a list of plain language tasks that we need to translate into Ansible tasks.

Step 1: Create a Volume

We will start at the top of the list. We need to create an EBS Volume for our web root. We can use the `ec2_vol` module to take care of this (step 1):

```
- ec2_vol:
    volume_size: 2
    name: web_volume
    device_name: /dev/xvdf
```

Step 2: Import the Key

Now we will import our key pair into AWS (step 2):

```
- name: Create AWS key pair using Ansible's key.
  ec2_key:
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    name: ansible_keypair
    key_material: "{{ lookup('file', '/home/ansible/.ssh/id_rsa.pub') }}"
```

Steps 3 and 4: Create an EC2 Instance and Update the Inventory

With our volume and key pair ready, we can create the EC2 instance (step 3). We can also go ahead and knock out updating our Ansible inventory here (step 4). Note that we do disable strict host key checking in our inventory entry to facilitate automation.

```
- name: Provision instance
  ec2:
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    ec2_region: us-east-1
    instance_type: t2.micro
    keypair: ansible-keypair
    image: '{{ BASE_AMI }}'
    assign_public_ip: yes
    vpc_subnet_id: '{{ DEFAULT_VPC_SUBNET }}'
    wait: true
    count: 1
    instance_tags:
      name: web-server
      image: yes
  register: ec2
- name: Attach volume
  ec2:
    aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
    aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
    ec2_region: us-east-1
    instance: '{{ ec2.instances }}'
    name: web_volume
- name: Update Ansible inventory
  add_host:
    hostname: '{{ item.public_ip }}'
    groupname: webservers
    ansible_ssh_common_args: "-o StrictHostKeyChecking=no"
  loop: '{{ ec2.instances }}'
```

Step 5: Configure the Instance

Now that we have an instance running and we can reach it with Ansible, we need to configure the system volume on the host and install our target software. We should note that our base AMI allows for `ec2-user` to sudo without passwords, which will allow us to operate as root using our Ansible key pair, so long as our remote_user is set to ec2-user:

```
- name: Create new partition
  become: yes
  parted:
    device: /dev/xvdf
    number: 1
    state: present
  register: parted_output
- debug:
    var: parted_output

- name: Format new partition
  become: yes
  filesystem:
    fstype: ext4
    dev: /dev/xvdf1

- name: Mount web root volume
  become: yes
  mount:
    path: /var/www
    src: /dev/xvdf1
    fstype: ext4
    state: mounted

- name: Install HTTPD
  become: yes
  yum:
    name: httpd
    state: present
  - name: Copy sample web content
    copy:
      src: index.html
      dest: /var/www/html/index.html
```

Step 6: Build the AMI

Now with our build finished, we can create an AMI for future clones:

```
- hosts: webservers
  remote_user: ec2-user
  tasks:
    - name: Collect instance ID
      ec2_metadata_facts:
      set_fact:
        target_image_host: '{{ ansible_ec2_instance_id }}'
- hosts: localhost
  tasks:
    - name: Create new AMI
      ec2_ami:
        aws_access_key: '{{ AWS_ACCESS_KEY_ID }}'
        aws_secret_key: '{{ AWS_SECRET_ACCESS_KEY }}'
        region: us-east-1
        instance_id: 'hostvars[groups['webservers'][0]]['target_image_host']'
        name: webcontentAMI
        state: present
```

Role Design

Having all of our tasks defined is most of the work. When it comes to putting the proverbial pen to paper, we need to simply lay the tasks out in an appropriately organized set of task files. This can be accomplished, to varying degrees, using roles. We can lay out a couple of example approaches.

Infrastructure Roles

The Concept

One approach would be to develop some roles related to hardware specification. The idea here is that we would build Ansible roles that create certain hardware configurations, such as a particular size of EC2 instance with certain storage parameters.

Use Cases

The uses cases here are somewhat limited, as different systems are going to call for different resources. A large benefit to cloud infrastructure is the ability to be very efficient in resource allocation.

The primary reason you might be able to use this approach is if you are provisioning smaller transient environments, for development or perhaps QA. In these cases, the environments are small enough that they are not very expensive. Such environments would also need to be regularly scrapped and re-provisioned, which makes a reasonably compelling use case for such a role.

Application Roles

The Concept

Ansible documentation lays out a great way to incorporate roles in your overall infrastructure. You can design roles for certain applications, such as a `base` role and perhaps a `web tier` role, and then apply necessary roles to certain groups of servers. It is not too difficult to apply our AWS knowledge to this scenario, whereby we build the infrastructure into the existing role rather than targeting bare metal servers.

Use Cases

The use case here remains the same as core Ansible. We are using roles to organize configuration just as we normally do. The exception is that we can simply extend our use case to include infrastructure management as well as system configuration. How many servers you will have in a particular application server cluster becomes a configuration value instead that is implemented dynamically rather than being built out beforehand.

A Web Site Role

In our web site example, we would move each of our steps into a task file that would be part of our role. The affiliated artifacts, such as files and base configuration, could be stored in the appropriate role directories (namely `files` and `vars`).

Our role has been constructed to produce an AMI that would be part of the actual production deployment. We could use the role in a playbook to rebuild our golden AMI as necessary. We would have the flexibility of modifying configuration values if we wanted to use a different base AMI, or provide a larger web root volume.

Naturally there are many ways that the AWS modules can be used to solve problems and build systems. This is just a very simple example of one way you might use the technologies together.

Conclusion

You have reached the end of the "Managing AWS with Ansible" course! Hopefully you found the content informative and engaging. There are many other facets of AWS covered on LinuxAcademy.com that you can combine with your new Ansible knowledge to further your skill set! We also have plenty of DevOps content to further automation! Be sure to check out both the AWS and DevOps categories on LinuxAcademy.com for more information.