Install Ansible and Make a Playbook

https://www.youtube.com/embed/mi9HPGap0R0

With our continuing efforts in building an MSP on open source it's important to realize we will need tools to help us automate many of the day to day operations we'll be performing for our clients as well as on our own systems. This brings us to Ansible.

Ansible is an incredibly powerful and robust tool made specifically for automating workflows and actions. It works on Linux, MacOS, and Windows, and can help reduce your task load by thousands, or even tens of thousands of individual actions. We will be rolling out Ansible during our series on building an MSP as a tool to help us deal with a growing business where we are supporting our clients and their multitudes of devices. We will employ Ansible's power to push out updates, agents for the various software services we'll employ, and so much more.

This is just an introduction to ansible, and will get you started with it, but it is highly recommended that you also check out other series on ansible. I'll link to one below that will take you a bit deeper. You should, of course start getting comfortable with the ansible documentation. This is one of the most well documented tools in the open source world. The answers are there for you, just waiting to be read and put to use.

What You'll Need

- A machine to use as an Ansible controller (the server that you run Ansible commands from).
- An SSH Public / Private key pair
- SSH Access to any target machine (client machine)
- About thirty minutes of your time

SSH Key Pair Setup

For this series, I'll be installing Ansible itself on a Ubuntu 22.04 LTS Incus (LXD) container. This is a dedicated virtual machine I'm creating for the sole purpose of running Ansible. You can do the same, or use your main laptop or desktop machine. Keep in mind, that as we grow our use of Ansible, we will want to start scheduling certain tasks to run, so we'll want a machine that is always on and ready.

To setup an SSH key pair, we can run the following command:

`ssh-keygen -t ed25519 -C "ansible"`

The command above tells ssh to generate a key pair using the ed25519 encryption. This encryption is similar in strength to RSA 256, but is a much shorter key, and uses less computer power to generate.

When you submit the command, you'll be prompted on where to store the command, and what to call it. This is up to you, but I highly recommend, you keep the key pair in

`/home/<your user>/.ssh/`

as this is where OpenSSH looks for the keys automatically. As for the name, name it something that makes sense for it's purpose. In my case I will name it 'ansible'. So my entier path and name will look like

`/home/<your user>/.ssh/ansible`

Next, you'll be prompted for a passphrase for this key. We want to leave this blank so that we won't be prompted each time for our passphrase as ansible goes to login to the machines it's taking action on.

Just press Enter to keep the passphrase blank. Next, confirm that you want it blank (empy) by simply pressing Enter again.

SSH will now generate a secure public and private key pair for you. If you do an `ls` on this folder (/home/<your user>/.ssh) you'll see that two new files haave been added. They'll be called `ansible` and `ansible.pub`. The file ending in .pub is your public key file and can be shared out to all of the machines you want to access with ansible. the other file is your private key file and should never be shared with anyone, nor sent to public machines. The `ansible` file is your key to access any host with the `ansible.pub` file on it.

Send Our New Key to Target Machines

In order for ansible to run, we need to share our new key to target machines. We can do this wil the command

`ssh-copy-id`

This command allows us to specify which key to share, and to which host and for which user we share it.

I'll be adding 3 hosts to start, but it's not required. You can do just one if you prefer. My hosts will have the IPs

192.168.10.20, 192.168.10.21, and 192.168.10.22

Don't worry, your hosts don't have to have consecutive IP addresses.

Let's copy and SSH key over to the first host machine.

`ssh-copy-id -i /home/brian/.ssh/ansible brian@192.168.10.20`

`-i` tells ssh that we are going to specify the file to send. We then tell it the path and key to send. You may notice I just put 'ansible' and not 'ansible.pub'. This is ok, as the `ssh-copy-id` command knows to send the public key, not the private key.

We press enter, then will be prompted for our password to access this machine via SSH. Enter the password, and you should get a message confirming that 1 key has been copied.

You can not attempt to access the machine using the new key with the ssh command as follows:

`ssh -i /home/brian/.ssh/ansible brian@192.168.10.20`

You should be logged onto the machine without being prompted for any password.

You have now copied your ssh key to another machine. Repeat the above process for however many machines you want to have ansible accessing.

Install Ansible

Ansible is already packaged in most distribution repositories, but the Ubuntu 22.04 version lags behind a bit, so let's make sure we get the latest version possible. To do this, we'll add the ansible ppa repository instead.

`sudo add-apt-repository ppa:ansible/ansible`

When you submit the above command, you'll be prompted to press Enter, so press Enter, then you should see your system attempt to update the package cache. Let's runt he update one more time just to be sure.

`sudo apt update -y`

Now we can install the latest version of ansible with

`sudo apt install ansible -y`

Creating an Ansible Inventory

Ansible uses an inventory file to know what machines you want it to access and perform actions on. This file can be yaml or plain text. We'll be starting with the plain text file for now, as I think it's a little bit easier to grasp initially. Feel free, however, to check out the documentation if you are a yaml expert.

Let's make our new file with

`nano inventory`

Note that Ansible can deal with machines collected into groups, and that a machine can be in more than one group, and that you can even group other groups as well.

We'll start with two groups for our example, but use whatever groups make sense for your needs. The organization for our inventory file will be like the following:

```
[group_name]
machine_alias_1 ansible_host=<machine_ip_1>
machine_alias_2 ansible_host=<machine_ip_2>

[group_name_2]
machine_alias_3 ansible_host=<machine_ip_3>
```

The items in square brackets are the group names we assign, and then below any group name we list the machines that are part of that gorup. We list the machines with an alias name, any name that describes what machine it is. This can be the machine hostname, or just an alias we want for the machine. After the alias, we put a space, then indicate the machines ip or fqdn for ansible to connect to the machine with. We identify it with the key `ansible_host=` then enter the ip or fqdn for the machine. Let's look at an actual example of our inventory file now.

```
[home_server]
dashy ansible_host=192.168.10.20
vaultwarden ansible_host=192.168.10.21

[home_desktops]
brian_studio ansible_host=192.168.10.22
```

Above, you'll see two groups home_server, and home_desktops. The home_server group has two machines in it, and one machine is held under the home_desktops group.

> NOTE: Ansible doesn't allow spaces in group names or aliases, nor hyphens (-), so you need to use underscores (_) in names where you want to separate words.

We can save our 'inventory' file with CTRL + O, then press Enter to confirm, and exit the nano editor with CTRL + X.

Create an Ansible Playbook

Playbooks, are pretty much exactly what they sound like. They are the instructions that you want ansible to execute on a target machine, or set of machines. You can target individual machines (by alias), or group(s) of machines, and even the entire inventory if you want.

Playbooks are written in yaml (yes, the space dependent, very picky file format), but it does make plays easier to read, and as you get more accustomed to yaml, easier to update and modify as needed.

While stiill in our ansible_projects folder, we'll create a new file called update_ubuntu_servers.yml.

`nano update_ubuntu_server.yml`

This will hold the plays we need in order to update our servers and desktops that are running Ubuntu as a base operating system.

In the file, we'll start by stating which target machines from our inventory file we want to perform an update on.

```
---
- hosts: all
```

In the above file, we start with three hyphens on the first line to indicate the beginning of our yaml file. below that we tell ansible that we want to run this on all hosts in our inventory file. Now, if you have machines with differing operating systems, you could group them by OS, but there are some smart checks we can do to make sure we don't try to do `apt updte` on an OpenSuse, Redhat, or other non-apt system as well.

To run the updates in apt, we need to be a sudo user. In ansible this uses the 'become' parameter, so let's add that to our file.

```
---
- hosts: all
  become: true
```

It's 'become' as in "become super user" or root.

Now that we've got the initial portion set, we can start creating 'tasks' for ansible to perform on these systems. The first task is to update the package cache, and the next task is to run the update on any packages that need it, although we list them in the reverse order in the yaml file.

```
---
- hosts: all
```

```
  become: true
  tasks:

    - name: Update packages on Ubuntu systems
      become: true
      apt:
        upgrade: dist
        update_cache: yes
```

Given our additions above, we can see that we give the task a name, this helps us know what task is being performed. This can be any string really, so call the task whatever makes sense.

Next, we again tell it this needs to be done as a super user with `become: true`.

And finally, we add the 'apt' module, and give it two actions to complete. `upgrade`, and we define that we want it to do a distribution level upgrade with `upgrade: dist`. Next we tell it that it needs to update the cahce of packages with `update_cache: yes`.

We can save this file and run it as is, and presuming you've setup your target machines properly, it will indeed update the cache of packages, and then run the upgrade procedure.

But, what about kernel updates? This usually includes the need for a reboot as well. I think we should go ahead and include that as part of our instruction set. to do that we just add another `- name` block under our `tasks:` section.

First, let's have Ansible check to see if a reboot is required.

```
---
- hosts: all
  become: true
  tasks:

    - name: Update packages on Ubuntu systems
      become: true
      apt:
        upgrade: dist
        update_cache: yes

    - name: Check if a reboot is needed
      become: true
      stat:
        path: /var/run/reboot-required
```

```
      register: reboot_required
```

In our next task section, we give it a descriptive name so we'll know what we've asked ansible to do. Then, again, we tell it to do this task as a super user. Next, we call the `stat` module. This can pull status information for us. We tell the 'stat' module to look in `/var/run/reboot-required` with the `path:` line, and finally we save the information in a variable with `register: reboot_required`. The variable name is 'reboot_required', and the stat command will check to see if that path exists or not.

Now that we know if a reboot is required, let's tell Ansible to perform the reboot if the variable 'reboot_required' is set to 'true' or 'yes'. To do that, we add another `-name:` section under our `tasks:` in our yaml file.

```
---
- hosts: all
  become: true
  tasks:
    - name: Update packages on Ubuntu systems
      become: true
      apt:
        upgrade: dist
        update_cache: yes

    - name: Check if a reboot is needed
      become: true
      stat:
        path: /var/run/reboot-required
      register: reboot_required

    - name: Rebooting Machine After Upgrade
      become: true
      ansible.builtin.reboot:
        reboot_timeout: 120
      when: reboot_required.stat.exists
```

Here, we name our task 'Rebooting Machine After Upgrade', then tell Ansible it must be run with super user privileges, and finally call on the built in Ansible module to reboot a machine, `ansible.builtin.reboot`.

Below that we add a short waiting time of 120 seconds (2 minutes). This just gives the upgrade time to fully complete before the reboot. Finally, we tell Ansible to only do this reboot when our 'reboot_required' variable shows that the `reboot-required` path exists. The `when` in Ansible is a nice easy way to add some logic to a task.

For example, if your friend asked, "Can you feed my dog?" That's pretyy open ended, and you may not want to feed their dog forever. So it's better for your friend to say, "Can you feed my dog *when* I go on vacation next month?". Now your friend has put a qualifier on when you should feed the dog.

This is the same as in Ansible. We are saying, "when /var/run/reboot-required exists, wait 2 minutes, then reboot the machine." Otherwise no reboot is necessary, so we leave the machine running.

Now we are ready to save our file, and exit the nano editor. We will run this with the `ansible-playbook` command. With the command we want to give Ansible instruction to use our 'ansible' private key, and tell it which host(s) to run the play on. Since we put the 'when' clause in our play, we can, of course just tell it 'all', but later on that may be a bit inefficient.

```
ansible-playbook --key-file ~/.ssh/ansible_key -i inventory --ask-become-pass update_ubuntu_server.yml
```

When you press Enter, you'll see Ansible ask you for your super user password, this is the password on the remote machines, and if entered incorrectly, will generate a list of failed task attempts. Next, Ansible will ensure it can reach each target machine, then it will begin running through each task in the playbook. At the end you'll get a summary of whether a play succeeded or failed, how many changes were made, how many tasks were skipped, and so on. What we are looking for is no tasks skipped, and no tasks failed.

Once we have that result, we have a play we can now use anytime we want in order to keep all of our servers up to date. We can even set this up to run as a cron job, but we need a secure way to feed Ansible our super user password for the tasks that require super user privileges.

Avoiding Fails on Different OSes

We can avoid fails that occur from running plays on machines or operating systems that don't make sense. Such as running apt on a Fedora machine. Again, we can turn to the 'when' operator in Ansible playbooks. The step where Ansible "gathers facts" is an important one, as Ansible grabs a bunch of details about our various machines. We can then use this information to create conditions

```
when: ansible_distribution == 'Ubuntu'
```

and

```
when: ansible_distribution == 'Fedora'
```

We can add this to our existing playbook, and add a section to update Fedora as well.

```
---
- hosts: all
  become: true
  tasks:
    - name: Update packages on Ubuntu systems
      become: true
      apt:
        upgrade: dist
        update_cache: yes
      when: ansible_distribution == 'Ubuntu'

    - name: Update packages on Fedora systems
      become: true
      dnf:
        update_only: true
        update_cache: true
      when: ansible_distribution == 'Fedora'

    - name: Check if a reboot is needed
      become: true
      stat:
        path: /var/run/reboot-required
      register: reboot_required

    - name: Rebooting Machine After Upgrade
      become: true
      ansible.builtin.reboot:
        reboot_timeout: 120
      when: reboot_required.stat.exists
```

In the above, we've added a new task named 'Update packages on Fedora systems', and instead of 'apt', we use 'dnf' as this is the package manager for Fedora, CentOS, Red Hat, etc.

Finally, we added a 'when' clause to the Ubuntu update task, and one to the Fedora update task. This clause will ensure that these tasks only attempt to update systems where the distribution matches the package manager we are using.

You're now setup with a great base for using Ansible to get things built out for automated management of systems and packages. We'll be using this in future videos in this series, so dig in, start watching and reading, and get comfortable with Ansible.

Series for learning Ansible that I suggest:

https://www.youtube.com/embed/3RiVKs8GHYQ?list=PLT98CRl2KxKEUHie1m24-wkyHpEsa4Y70pp=iAQB

Support My Channel and Content

Support my Channel and ongoing efforts through Patreon:

patreon.com/awesomeopensource

Buy me a Beer / Coffee:
https://paypal.me/BrianMcGonagill

Revision #2
Created 2024-05-30 11:45:35 UTC by Brian McGonagill
Updated 2024-05-30 15:14:26 UTC by Brian McGonagill