Ansible

- Managing Remote Hosts
- Creating Roles
- Creating Playbooks

Managing Remote Hosts

Basic Requirements

On this page, I'll describe how to configure Ansible to manage a remote host. In the context of this page, a `controller` is the ansible node that executes commands on remote hosts, while a `client` is a host which accepts commands from some controller.

The requirements for running / creating an ansible controller for a set of clients is as follows -

- **controller**
  - has ansible
  - create ssh key as the ansible user
    - `ssh-copy-id <worker>`
    - should be able to ssh with no password - `ssh <host / IP>` as ansible user
      - If the above does not work, create `/home/USER/.ssh/config` and add `IdentityFile /path/to/Private.key`, this will pass the key automatically when connecting as USER.
      - Ensure the host you are connecting to has the connecting key within the `~/.ssh/authorized_keys` file.
      - restart sshd.service - `sudo systemctl restart sshd.service`
- **client**
  - has ansible
  - has a known password, but can sudo without one.
    - `<user> ALL=(ALL:ALL) NOPASSWD:ALL` within sudoers

Creating a Controller

This section will configure a new user to be our Ansible controller -

- **controller**
  - has ansible
  - create ssh key as the ansible user
  - `ssh-copy-id <worker>`
  - should be able to ssh with no password - `ssh <host / IP>` as ansible user
    - If the above does not work, create `/home/USER/.ssh/config` and add `IdentityFile /path/to/Private.key`, this will pass the key automatically when connecting as USER.
    - Ensure the host you are connecting to has the connecting key within the `~/.ssh/authorized_keys` file.
    - restart sshd.service - `sudo systemctl restart sshd.service`

First, install Ansible -

```
sudo apt-add-repository --yes --update ppa:ansible/ansible && sudo apt update -y
sudo apt install software-properties-common -y && sudo apt install ansible -y
```

Creating Controller Ansible User

On the controller we plan to use to manage remote hosts, create a user that will carry out all Ansible commands.

```
sudo adduser username
[sudo] password for admin:
ssh-rsa
AAAeAB3NXyXeAAADAQABAAABXwxAQDXndHlHw2DxXMk1thdTsSJWoRxXXGl5jXXMGaRta1sdprzg/sXJAdding
user `username' ...
Port 22
Adding new group `username' (1000) ...
Adding new user `username' (1000) with group `username' ...
Creating home directory `/home/username' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for username
Enter the new value, or press ENTER for the default
    Full Name []:
    Room Number []:
    Work Phone []:
    Home Phone []:
    Other []:
Is the information correct? [Y/n] y
```

Controller Sudo Configuration

Now that we created our user, we need to configure sudo, add `user ALL=(ALL:ALL) ALL` to the following file -

Add or edit our custom sudoers config to allow for sudo with no password

```
sudo visudo -f /etc/sudoers.d/mySudoers
```

Add the following line -

```
kansible ALL=(ALL:ALL) NOPASSWD:ALL
```

```
# Add our new user to sudo group
admin@server:~$ sudo vigr
You have modified /etc/group.
You may need to modify /etc/gshadow for consistency.
Please use the command 'vigr -s' to do so.
admin@server:~$ sudo vigr -s
You have modified /etc/gshadow.
You may need to modify /etc/group for consistency.
Please use the command 'vigr' to do so.
```

Secure the new user's User / Group ID's by defining a custom user and group ID

```
sudo usermod -u 61182 username
sudo groupmod -g 61181 username
```

Change file permissions created when we added the user. Here we are just updating user files to reflect new IDs. Errors are ok

Change all files to reference the correct group -

```
sudo find / -group 1000 -exec chgrp -h username {} \;


find: ‘/proc/18580/task/18580/fd/6’: No such file or directory
find: ‘/proc/18580/task/18580/fdinfo/6’: No such file or directory
find: ‘/proc/18580/fd/5’: No such file or directory
find: ‘/proc/18580/fdinfo/5’: No such file or directory
```

Change all files to reference the correct user -

```
sudo find / -user 1000 -exec chown -h username {} \;


find: ‘/proc/18611/task/18611/fd/6’: No such file or directory
find: ‘/proc/18611/task/18611/fdinfo/6’: No such file or directory
find: ‘/proc/18611/fd/5’: No such file or directory
find: ‘/proc/18611/fdinfo/5’: No such file or directory
```

That's it! Further customization for managing our remote servers will take place in defining hosts in the Ansible inventory, creating playbooks, and defining / applying roles. For now, we should configure a remote host to accept commands from this new Ansible controller

Creating Ansible Clients

Below, we configure a user to authenticate with on the remote host we want to admin, known as our Ansible client -

- **client**
  - has ansible
  - has a known password, but can sudo without one.
    - `<user> ALL=(ALL:ALL) NOPASSWD:ALL` within sudoers

To create an Ansible client, you'll need a user with a known password that can sudo without one. Also, we will need to install our publickey from the controller we created above into this users `~/.ssh/authorized_keys` file so Ansible can ssh and sudo on this worker with only a private key.

First, install Ansible on the remote client -

```
sudo apt-add-repository --yes --update ppa:ansible/ansible && sudo apt update -y
sudo apt install software-properties-common -y && sudo apt install ansible -y
```

Creating Ansible User for Remote Client

To speed this up, I used a script I wrote to create a user with a custom userID, and configure sudo.

Get it here, or manually create the user as I did above for the Ansible controller.

If we run the script with no arguments, we see the help text -

```
sudo ./adduser.sh ansible
Illegal number of parameters.
Usage: sudo ./adduser.sh <username> <groupid>

Available groupd IDs:
60001.....61183 Unused | 65520.............65533  Unused
65536.....524287 Unused | 1879048191.....2147483647  Unused
```

So we can add a user with the following command -

```
sudo ./adduser.sh ansible 524280

Adding user `ansible' ...
Adding new group `ansible' (524280) ...
Adding new user `ansible' (524280) with group `ansible' ...
Creating home directory `/home/ansible' ...
```

```
Copying files from `/etc/skel' ...

Enter 1 if ansible should have sudo privileges. Any other value will continue and make no changes
1

Configuring sudo for ansible...

Enter 1 to set a password for ansible, any other value will exit with no password set
1

Changing password for ansible...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

Configure Sudo for Remote Client

Now, we need to configure the Sudoers file to allow our user to sudo without the password, even though we did configure a password during user setup.

```
sudo visudo -f /etc/sudoers.d/mySudoers
```

Assuming your username is `ansible`, add the following line to this file. -

```
ansible ALL=(ALL:ALL) NOPASSWD:ALL
```

> Be sure to either run the `sudo visudo -f /etc/sudoers.d/mySudoers` or append the line above to the end of the default sudoers file if you ran only `sudo visudo` - This is a sequential configuration so the order of the statements is important, and we want to ensure that nothing overrides our choice to disable sudo passwords on this user

Now the ansible user can sudo with no prompt for password! Now we just need to add our controller's SSH key to the .ssh/authorized_keys file within the new ansible user's home directory.

Login as the user, and add the publickey that Ansible will pass for authentication.

```
sudo -iu username
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.
username@server:~$ mkdir .ssh
username@server:~$ sudo vim .ssh/authorized_keys
```

Verify sshd_config, and restart sshd.service

```
sudo vim /etc/ssh/sshd_config
sudo systemctl restart sshd.service
```

Once you have added your key to the authorized_keys file, determine if you have or plan to have any custom PAM configurations on your host, and if so - add the following module to bypass any future changes.

Adding Listfile Module (PAM)

If you have or plan to have any custom PAM configurations on your host, you will need to change PAM sshd authentication configuration as follows to allow our user to bypass other modules

```
sudo vim /etc/pam.d/sshd
```

In `/etc/pam.d/sshd`, we can add the following line to allow for a list of users past any other modules configured on the server. Be sure to add this line at the top of our configuration file, so it is handled before any other module.

```
auth sufficient pam_listfile.so item=user sense=allow file=/etc/authusers
```

Now we can add our user to the pam_userlist.so configured in the changes made above

```
sudo vim /etc/authusers
```

In this `/etc/authusers` file, we simply list users that can bypass further PAM configurations -

```
user
otheruser
thirduser
```

Updating hosts

Be sure you add your hose IP and port to your `/etc/ansible/hosts` file, syntax is seen below -

```
[group]
www.domain.com
sub.domain.com:22
0.0.0.0
127.0.0.1:22
```

```
[othergroup]
sub.domain.com:22
127.0.0.1:22

[nginx-server]
sub.domain.com:22

[docker-host]
127.0.0.1:22

[dev]
sub.domain.com:22
```

That's it! Now just `sudo apt install ansible` and ssh to your Ansible controller to test out the configuration.

Testing Ansible client

From this point, the user is fully configured to bypass all security settings only if the ansible controller is attempting to connect, allowing full sudo access. To test this, run the following command and look for similar output -

```
ansible dev -m ping
The authenticity of host '159.203.190.63 (159.203.190.63)' can't be established.
ECDSA key fingerprint is SHA256:jDxFV7KA00wNIdpG40ppvW2RobNXyPeltdi4jL3h78s.
Are you sure you want to continue connecting (yes/no)? yes
worker.domain.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

This test says that the host was not changed (`"changed": false`), and the server accepted our connection (`"ping":"pong"`)

Creating Roles

Ansible Galaxy

Ansible has a built in tool `ansible-galaxy` which allows us to quickly create a set of folders and files that are needed in the creation of an Ansible role.

Simply run `ansible-galaxy init rolename --offline` and a folder will be created within your current directory that contains the basic structure of an Ansible role. Within this directory, we can easily pick and choose which components we will need for our role.

Creating NGINX Roles

To begin, we will create a simple role for installing and configuring a simple nginx server. Navigate within your role, which we will assume is simply called `nginx-role`

Define Tasks

Within `nginx-role/tasks/main.yml` we include the following -

```
---
# tasks file for /etc/ansible/roles/nginx
- import_tasks: install.yml
- import_tasks: configure.yml
- import_tasks: service.yml
```

This task assumes that within the `nginx-role/tasks/` directory we also have the files `install.yml`, `configure.yml`, and `service.yml` - See the below snippets for examples of how these files could look, depending on your scenario.
Within the `nginx-role/tasks/` directory, create the following files -

Create a `nginx-role/tasks/install.yml` task for installing nginx and any other required packages if needed

```
---
- name: Install nginx Package
apt: name=nginx state=latest
```

Create a `nginx-role/tasks/configure.yml` task for templating various configuration files needed to configure an nginx webserver

```
---
- name: Copy nginx configuration file
  template: src=files/nginx.conf dest=/etc/nginx/nginx.conf
- name: Copy index.html file
  template: src=files/index.html dest=/var/www/html
  notify:
    - restart nginx
```

Create a task `nginx-role/tasks/service.yml` for starting the nginx service

```
---
- name: Start and enable nginx service
  service: name=nginx state=restarted enabled=yes
```

Now we have defined all the tasks that Ansible needs to carryout in order to create a new nginx host. All thats left to do is ensure that the tasks we created above have all the resources we said would be available when the role is ran on a host.

Define Handlers

In the tasks above, notice the `notify: -restart nginx` within configure.yml. Here, we have declared that this task makes changes that require nginx to be restarted in order to be applied. So, we create the handler task below to carry out the `restart nginx` task that we have notified of our changes.
To set this up, create the following `nginx-role/handlers/main.yml` configuration

```
---
# handlers file for /etc/ansible/roles/nginx
- name: restart nginx
  service: name=nginx state=restarted
```

Define Templates / Files

Ansible will need to refer to the templates / files we declared in the above tasks -
Add them within the `nginx-role/files/` directory

Create the following `nginx-role/files/nginx.conf`

```
user www-data;
worker_processes auto;
pid /run/nginx.pid;

events { }
```

```
http {
  include mime.types;

  # Basic Server Configuration
  server {
    listen 80;
    server_tokens off;
    server_name {{ domain_name }};

    location / {
      root {{ nginx_root_dir }};
      index {{ index_files }};
    }

    # Uncomment to pass for SSL
    #return 301 https://$host$request_uri;
  }
}
```

Then we can create a custom template at `nginx-role/files/index.html` for our landing page to verify things are working.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to nginx!</title>
    <style>
      body {
          width: 35em;
          margin: 0 auto;
          font-family: Tahoma, Verdana, Arial, sans-serif;
      }
    </style>
 </head>
  <body>
    <h1>Klips!</h1>
    <p>If you see this page, the nginx web server is successfully installed and working. Further configuration is required.</p>
    <p>For online documentation and support please refer to
```

```
    <a href="http://nginx.org/">nginx.org</a>.<br/>
    Commercial support is available at
    <a href="http://nginx.com/">nginx.com</a>.</p>
    <p><em>Thank you for using nginx.</em></p>
  </body>
</html>
```

Define Variables / Defaults

Last, we need to define the Ansible defaults we referenced in the above configurations `{{ variable_name }}` is a variable within Ansible, these can be used to create roles that can be used dynamically or easily reconfigured and reapplied to different scenarios.
Create the following `main.yml` file in `nginx-role/defaults`

```
---
# defaults file for /etc/ansible/roles/nginx
#
domain_name: "localhost"
nginx_root_dir: "/var/www/html/"
index_files: "index.html index.htm"
```

Ansible has a wide range of variables, or facts, that it collects on the hosts within its inventory. To see a complete list of all the facts available for a host, run the following

```
ansible hostname -m setup
```

This will print a ton of information, all of which is available for use within ansible templates by calling a variable corresponding to the fact name. For example, if we wanted the fact `ansible_hostname` and `ansible_fqdn`, we call them as `{{ ansible_hostname }}` or `{{ ansible_fqdn }}`. When these variables are ran within a playbook, ansible will insert the values of these variables depending on the host the task is running on.

Using Ansible Roles

That's it! Now all we need to do is create an inventory / hosts file and run a playbook using our new role -

Create your ansible host file at `/etc/ansible/hosts` with the relevant information for your environment

```
# This is the default ansible 'hosts' file.
#
# It should live in /etc/ansible/hosts
```

```
[group]
www.domain.com
sub.domain.com:22
0.0.0.0
127.0.0.1:22

[othergroup]
sub.domain.com:22
127.0.0.1:22

[nginx-server]
sub.domain.com:22
```

Create the playbook `/etc/ansible/nginx.yml` to kick off our role using the role information and groups entered above within the `hosts` file.

```
---
- hosts: nginx-server
  become: yes
  roles:
  - nginx
```

Now from within `/etc/ansible/`, simply run `ansible-playbook nginx.yml` and our tasks configured above will be carried out on the server defined in the `hosts` file above.

If you are testing using SSL, be sure to use the `--dry-run` argument until your configurations are tested and working correctly.

```
sudo certbot certonly -d domain.com -d www.domain.com --dry-run --standalone --agree-tos -m some-email@domain.com
```

Creating Playbooks

Ad-Hoc Commands

First, we should be sure that ansible is configured correctly, to run commands on a server or a group of servers within the `/etc/ansible/hosts` file, run any of the below commands

```
ansible -m ping hostname
ansible -m ping 134.23.4.5

ansible -a "sudo ls /" hostname
ansible -a "sudo ls /" 134.23.4.5

ansible -a "free -h" hostname
ansible -a "free -h" 134.23.4.5
```

While the above is an example of running bash commands on remote hosts ad-hoc via the commandline, you can also run ansible modules from the commandline in a similar way -

```
ansible remotehostname -m fetch -a "src=/home/remoteuser/path/file.txt dest=/home/localuser/ flat=yes"
```

Here, we grab `file.txt` from a remote host and copy it to our local home directory. Where `-m` is selecting which module to use and `-a` is providing the options that you would specify within a normal playbook via a command. Be sure to enclose any module options after `-a` with double quotes or the command will fail. We use `flat=yes` to tell Ansible that we just want the file, and not to rebuild the directory from the remote host. `flat` defaults to `no`, which would result in this command building out the full directory `/home/localuser/www.remotedomain.com/home/remoteuser/path/file.txt` on our local host. See the Ansible documentation for each module for more information on their arguments. Here's a link to the documentation for the fetch module

For the above command, we used the `fetch` module, which may not work for directories and may consume a lot of memory if you are transferring a large file. For example, I experienced issues with this when transferring large database backup files between hosts. If this is your use case, I would recommend checking out the synchronize module documentation.

As an example of an ad-hoc `synchronize` command, I have used this in the past to retrieve fail2ban configurations on a remote host. Note the `mode=pull` parameter that tells ansible that we want to get the files from the remote host and place them at the local destination. By default, `mode` is set to `push`, which would attempt to copy files from our local host and send them to a directory on the

remote host.

```
ansible -m synchronize remotehostname -b -a "src=/etc/fail2ban/filter.d/ dest=/some/local/directory/fail2ban/
mode=pull"
```

Creating Playbooks

Ansible can be configured to carry out tedious or otherwise common tasks on any number of hosts, as we see below in the example playbook where Ansible is being used to backup an instance of Bookstack.

```
---
- hosts: bookstack
  become: yes
  tasks:
    - name: Backup Bookstack container files
      command: tar -cvzf bookstack-backup.tar.gz /home/admin/bookstack
    - name: Fetch backup files from remote host
      command: scp -P 2222 -i /home/username/.ssh/id_rsa /home/admin/bookstack-backup.tar.gz
admin@sub.domain.com:/home/admin/backups/bookstack/
```

Here, we use `scp` instead of Ansible's Fetch module to save memory on the small host that runs the BookStack you are viewing. When fetching large files, memory errors can be encountered so here we have worked around the module using an alternative method for transferring our files.

Here is another example, using ansible to synchronize the fail2ban configurations used between multiple hosts. This allows us to configure one host, which is the local host that runs the playbook, and once we have configured this host correctly we can just run the play and push our changes to a group of hosts. Note that `hostgroup` should be specified in the local `/etc/ansible/hosts` file, or ansible will not be able to run the play. Also notice that the `src` directories in this playbook are relative to the path of the playbook itself. This allows me to store custom fail2ban configurations for different groups of host alongside this playbook to avoid configuring fail2ban to monitor services that don't exist on the system. If you try to monitor a service that does not exist, fail2ban will fail to reload.

```
---
- hosts: hostgroup
  become: yes
  tasks:
    - name: Copy custom fail2ban filters
      synchronize:
        mode: push
        src: fail2ban/filter.d/
```

```
    dest: /etc/fail2ban/filter.d/
- name: Copy custom fail2ban jail.local
  synchronize:
    mode: push
    src: fail2ban/jail.local
    dest: /etc/fail2ban/
- name: Reload fail2ban service
  ansible.builtin.service:
    name: fail2ban
    state: reloaded
- name: Checking status of fail2ban service after restart
  command: systemctl status fail2ban
  register: result
- name: Showing fail2ban status report
  debug:
    var: result
```

> Be careful when synchronizing configurations in this way, hosts can be configured with different services which could result in the fail2ban service failing to reload when it is unable to find the related log files. For this reason, I use a seperate directory to configure fail2ban for hosts with similar filters. In my case, my host that runs the ansible playbooks does not have nginx installed, so copying over configurations for nginx jails will result in fail2ban failing to reload.