Ansible

Info

Ansible Homepage | Ansible Documentation | jamlab-ansible | Jeff Geerling's Ansible Guide | Fast Ansible Guide

Ansible is a software tool that provides simple but powerful automation for cross-platform computer support. It is primarily intended for IT professionals, who use it for application deployment, updates on workstations and servers, cloud provisioning, configuration management, intra-service orchestration, and nearly anything a systems administrator does on a weekly or daily basis. Ansible doesn't depend on agent software and has no additional security infrastructure, so it's easy to deploy.

For the best guide for deep diving into using Ansible check out Jeff Geerling's Ansible Guide if you like video format or Fast Ansible Guide if you prefer text.

For configuration management it made sense to go with something simple to ease bootstrapping and favoring mutability for fastest development. Running a whole platform like Puppet did not make sense because of bootstrapping and resource overhead. Ansible is simple to write, understand and manage if written well from the get-go. I also tried SaltStack, but in the end it had too many shortcomings, check out the conclusions of the Ansible User's Guide to Saltstack page.

Also knowing Ansible I knew how slow it can be. There's two ways of solving this: using push mode with a central management (with homebrew solutions or AWX/Ansible Tower) with parallel playbook execution for each host OR pull mode where each host essentially configures itself. Running AWX/Ansible Tower has the same problem of bootstrapping and resource overhead. Homebrew parallel push system spikes the central management resource usage when executed and requires you to be on two hosts (central management host and the host being configured) when developing. It is quite evident that pull mode is the more scalable, resource efficient and easier for swift changes, although because of it's outside-in nature it is less secure. I've tried and used both, but went back to push mode using ansible-parallel.

I settled on the following requirements:

- Easy to bootstrap (i.e. couple of commands excluding secrets)
- Scalable (execution time does not depend on the number of hosts)
- Simple to modify and manage (DRY monorepo for all hosts)
- No single point of failure in the form of a centralized configuration bastion

The solution was jamlab-ansible: Homelab push-mode configuration management with Ansible.

Ansible Best Practices

Idempotency

The most important thing about using Ansible is that all tasks should be idempotent. It means that each time any task is run, the result of it should be the same regardless of any state on the machine it is run on. For example if you want to install some package on a host with ansible and use the ansible.builtin.shell module for it with some command. Maybe it will succeed the first time but give an error when the package is already installed.

Instead of ansible.builtin.shell module we should use purpose built Ansible modules if they exist since they will make sure that the result is idempotent. However you can make shell tasks idempotent as well with some workarounds. For example consider the following very common trick of registering outputs from tasks:

YAML

```yaml
```

```
# First we check if a directory for CNI exists.
- name: Check if cni exists
ansible.builtin.stat:
    path: /opt/cni/bin/bridge
register: r_cni # Then we register the output of this task
in a variable called "r_cni" (using r_ prefix is an old
convention)

# Check if newer CNI exists IF the directory in the last
task did exist, check "when" key at the bottom of this task
- name: Check if newer cni exists
ansible.builtin.shell: |
    latest_tag=$(curl -s
https://api.github.com/repos/containernetworking/plugins/relea
| jq -r ".tag_name")
    current_ver=$(/opt/cni/bin/bridge 2>&1 | cut -d " " -f
4)
    case "$current_ver" in ${latest_tag} ) echo "latest";;
*) echo "outdated";; esac
register: r_cni_ver # We register the output of our commands
which in this case is either "lastest" or "outdated" we will
use this for handling the cases in the next task
when: r_cni.stat.exists # We only run this task if the
output of the last task says that the directory did exist

# Get the latest CNI if the output of the last task was not
"latest", check "when" key at the bottom of this task
- name: Get latest cni
ansible.builtin.shell: |
    latest_tag=$(curl -s
https://api.github.com/repos/containernetworking/plugins/relea
| jq -r ".tag_name")

latest_url=https://github.com/containernetworking/plugins/relea
plugins-linux-amd64-${latest_tag}.tgz
    wget -P /tmp $latest_url
    mkdir -p /opt/cni/bin
    tar -C /opt/cni/bin -xzf /tmp/"${latest_url##*/}"
    rm /tmp/"${latest_url##*/}"
when: not r_cni.stat.exists or r_cni_ver.stdout != "latest"
# We run this task if CNI directory does not exist or when
the output of the last task was not "latest"
```

Readability

The second most important thing about using Ansible is always being explicit. For example when using modules, it is better to write "ansible.builtin.shell" instead of "shell". That is because external modules and community modules can also be used, but it should be obvious which module is used.

Also it should be immediately obvious where variables come from and what is the variable override precedence. This why it is not native behavior in Ansible to combine dicts and lists from different "variables" or "defaults" files. Instead the variables will follow a precedence and overwrite the one before it. Usually this follows the pattern of (weakest to strongest precedence): global variables, group variables, host variables. So a list from global variables will be overwritten if a list with same name exists in host variables for example.

Jamlab Ansible Architecture

And as per Ansible's own best practices: complexity kills productivity. And I think that a typical ansible monorepo is a bit too complex and usually it is not immediately obvious what goes where.

A typical ansible management repository loops something like the examples from the old best practices doc of Ansible:

Bash

```
production                # inventory file for production servers
staging                   # inventory file for staging environment

group_vars/
   group1                 # here we assign variables to particular groups
   group2                 # ""
host_vars/
   hostname1              # if systems need specific variables, put them here
   hostname2              # ""

library/                  # if any custom modules, put them here (optional)
module_utils/             # if any custom module_utils to support modules, put
them here (optional)
filter_plugins/           # if any custom filter plugins, put them here
(optional)

site.yml                  # master playbook
webservers.yml            # playbook for webserver tier
dbservers.yml             # playbook for dbserver tier

roles/
    common/               # this hierarchy represents a "role"
        tasks/            #
            main.yml      #  <-- tasks file can include smaller files if
warranted
        handlers/         #
            main.yml      #  <-- handlers file
        templates/        #  <-- files for use with the template resource
            ntp.conf.j2   #  <------- templates end in .j2
        files/            #
            bar.txt       #  <-- files for use with the copy resource
            foo.sh        #  <-- script files for use with the script resource
        vars/             #
            main.yml      #  <-- variables associated with this role
        defaults/         #
            main.yml      #  <-- default lower priority variables for this role
        meta/             #
            main.yml      #  <-- role dependencies
        library/          # roles can also include custom modules
        module_utils/     # roles can also include custom module_utils
        lookup_plugins/   # or other types of plugins, like lookup in this case

    webtier/              # same kind of structure as "common" was above, done
for the webtier role
    monitoring/           # ""
    fooapp/               # ""
```

In this structure, each root playbook including the master playbook (`site.yml` in this case) is defined in the project root directory and imports roles from `roles/`, variables from `group_vars/` and `host_vars/`. Then the master playbook runs all the other playbooks that define which roles are run on which hosts or host groups. This introduces a problem where a breaking change in one role will halt the whole run. Also, even with well organized root playbooks, it is never immediately obvious which roles are defined for which root playbooks especially if using hosts in multiple groups or child/parent groups. Furthermore, the root playbooks, `group_vars/` and `host_vars/` are in separate directories which is not a huge deal, but this does require one to verify that root playbooks, variables and roles match when planning changes. This requires extra time of getting familiar with what-goes-where especially when doing changes after a long time. For larger projects usually the roles are managed in and imported from separate repositories. It is a great approach, especially for running tests on the roles. However this increases the time of understanding what-goes-where.

These are small nitpicks and for most use cases following the standard structure works well, but for maximum simplicity I grew very fond of a system for pull mode Ansible we used at CERN. An example structure for this system looks something like this:

Bash

```
ansible.cfg                 # ansible configuration file
hosts                       # inventory file

bin/                        # binaries
    bootstrap.sh            # script for setting up host for the first time
    run-playbooks.sh        # script for running relevant playbooks locally on host

playbooks/                  # "root" playbook directory
    group_base/             # ""
        main.yml            # here we define roles for a particular group
        group_vars/         # ""
            all.yml         # here we assign variables to a particular group
        host_vars/          # ""
            <hostname>.yml  # here we assign variables to a particular host
    host_<hostname>/        # here we define roles for a particular host
        main.yml            # define roles for a particular host
        host_vars/          # ""
            <hostname>.yml  # here we assign variables to a particular host
    function_test/          # ""
        main.yml            # ""

roles/                      # roles directory
    <role>/                 # role name
        defaults/           # ""
            main.yml        # <-- default lower priority variables for this role
        files/              # ""
            file.txt        # <-- files
            template.txt.j2 # <-- files for use with the template resource
        tasks/              # ""
            main.yml        # <-- tasks to run for the role
```

With this system root playbooks are separated into directories with their own variables and are not run from a single master playbook thus each play can run regardless of whether there are errors in other playbooks. Each playbook only defines which roles to run on the host group and nothing else, for example:

YAML

```yaml
```

```
- name: PLAYBOOK FOR GROUP 'GGG'
  hosts: ggg

  roles:

  - { role: pre, tags: [ pre ], when: not (disabled_roles.pre | default(false))
}

  - { role: rrr, tags: [ rrr ], when: not (disabled_roles.rrr | default(false))
}

  - { role: post, tags: [ post ], when: not (disabled_roles.post |
default(false)) }
```

This and it's accompanying variables file make it simple to understand at a glance which roles are run and where the group variables are defined since they are all together in one directory.

For maximum simplicity for managing the playbooks and roles it should be enforced that each host is only part of ONE group. This will ensure that it will always be immediately obvious which playbooks are run for what host when looking at the inventory file.