Ansible

Friday, February 12, 2016 3:34 PM

2/12/2016

Installing (reference)

I used these commands to install on Debian:

- PIP way (version 2.x):
  - sudo apt-get install python-pip
  - **Don't run this step unless something goes wrong below.** You may need to upgrade pip (and you may need to run this as sudo).
    - sudo pip install --upgrade pip
  - pip install ansible --user
    - **DO NOT RUN AS SUDO.** Specifying "--user" will run as the current user.
    - You may need to run this as "python-pip" depending on the platform
    - If you get an error about "Python.h", you may need the Python dev tools:
      - sudo yum install python26-devel
        - Note: the Python version comes from the pip error message, not from "python --version"
- Out-of-date way (version 1.7.2 as of 3/14/2016):
  - sudo apt-get update
  - sudo apt-get install ansible
- Note: if this doesn't work, check the reference page and run the full instructions.

Updating Ansible

Run this:

```
pip install ansible -U --user
```

QuickRef: https://github.com/lorin/ansible-quickref

Echoing / debug-logging values during a playbook (reference)

Just add something like this as a task:

```
- debug:
    msg: "Account server dir: {{ accountServerDir }}"
  tags:
    - migrate
```

Note: spacing is very important, just copy/paste what's above

Note: make sure it has the same tags that you're trying to run with. For example, if you start Ansible with **--tags "start"**, then it will skip your debug task unless you also give it the same tag.

Basics (reference)

- Ansible is agentless, meaning all it needs is SSH in order to perform everything; it will connect to a machine, it will push Ansible Modules to the machine, run what it needs to, then clean up the modules that it pushed.
- Ansible should be used for ensuring that the state of a machine is what you think. It is not necessarily meant to be used as a task runner. For example, you may want to make sure that a MySQL server is running and has certain users/tables set up. Ansible can do this (via a module),

but that's probably all part of setting up an entire *system* instead of just a single endpoint.

- An inventory is a grouping of machines, for example

  [webservers]
  www1.example.com
  www2.example.com

  [dbservers]
  db0.example.com
  db1.example.com
- Testing Ansible:
  - Make sure you have an Inventory set up. I made a basic one in /etc/ansible/hosts that looks like this:

    [testeroni]
    192.168.1.26

    I imagine it's generally not advised to do this because your Inventory is something you'd probably want to save in git and not have to put it in a root-only folder. Instead, put your
  - ansible all -a "/bin/echo hello"
- You should make sure you're running in an ssh-agent (see Debian note) or that your SSH key is "id_rsa", otherwise you'll have to specify the key file every time you launch.
- Project structure
  - This is discussed a bit here.
- You can choose just a subset of certain groups from your inventory if you'd like by specifying the "-l" or "--limit" argument, e.g. --limit "overseers,accountservers"

Configuration file (reference)

If you don't want to specify a bunch of garbage on the command line, then you can create an "ansible.cfg" in the current directory and set variables in there. For example, if I don't want to pass the "-i", then I could put this in the config file:

    hostfile = hosts   <-- note: this option should be changed to "inventory = hosts" after version 1.9.

Inventory (reference)

If you're going to make a custom inventory for a project, you should just name it "hosts". Vim will know to highlight the comments somehow, possibly based on the name of the file.

A very basic sample inventory looks like this:

    [RESTservers]
    192.168.1.2

To use this inventory, pass "-i hosts" to Ansible.

The names between square brackets are group names, e.g. [RESTServers].

You can have groups of groups by using ":children":

    [washington]
    1.2.3.4

    [california]
    1.2.3.5

    [usa:children]

washington
california

You can set variables for entire groups like this:
[usa:vars]
key=value

Note: every host is part of the "all" group by default.

However, setting variables above means you can't use YML format, so instead, you should take your group name ("usa" in this case) and make a group_vars/usa.yml file that simply contains your variable definitions. The folder name "group_vars" is important to Ansible and will be searched along with any other applicable group_vars files, e.g. "california.yml".
Alternatively, you could make a subdirectory in "group_dirs" named after your group and then put a "vars" file in there, e.g. "group_vars/usa/vars".

```
# group_vars/usa.yml
---
foo: 5
hostName: "{{ foo }}"
```

Vault

ansible-vault allows you to encrypt any data for Ansible so that you can check it into a codebase without worrying about data leaking.

First, you should have a "group_vars/<group name>" directory already. From that directory, run "ansible-vault create vault" and type a password to create the vault file.

Edit the file so that you have all of your encrypted variables, and make sure they start with the "vault_" prefix. For example:

```
---
vault_testVar: some value here
vault_foo: some other value
```

Then, you should expose these encrypted variables by modifying group_vars/<group name>/vars to point at the encrypted variables, this time without the "vault_" prefix.

```
---
testVar: "{{ vault_testVar }}"
foo: "{{ vault_foo }}"
```

To run using the vaulted variables:

```
ansible-playbook installoverseer.yml  --ask-vault-pass
        OR
ansible-playbook installoverseer.yml  --vault-password-file ~/.vault_pass.txt
```

To edit the files later:

```
ansible-vault edit <path to vault.yml>
```

To change the password, use

```
ansible-vault rekey <path to vault.yml>
```

It'll then ask for the old/new passwords. Keep in mind that if you're checking your vault into a repository and you have the need to do this, then it VERY LIKELY means that you should

change every single piece of information inside the vault. I.e. let's say you're panicking because the vault password leaked, so you change it. Someone could still go into your version control, get the old vault, and use the old password on that to get all of your private info.

Getting stdout/stderr

To get stderr, simply enable the verbose flag ("-v") upon running. Note: I don't know about "-v", but "-vvvv" DEFINITELY prints out more information than you should be comfortable showing on-stream.

To get stdout, you can use the **register** option followed by a debug log, but you can only do this if the task isn't failing. If it *is* failing, then you also need "ignore_errors:true".

```
tasks:
  - name: Some test
    command: node foo.js

    register: out
    ignore_errors: true

    - debug: var=out.stdout_lines
```

This prints like this:

```
TASK: [debug var=out.stdout_lines] ******************************************
ok: [192.168.1.26] => {
   "out.stdout_lines": [
      "Running CREATE DATABASE botland IF NOT EXISTS;",
      "Knex:warning - Pool2 - Error: Pool was destroyed",
      "Knex:Error Pool2 - Error: ER_ACCESS_DENIED_ERROR: Access denied for user
'root'@'192.168.1.26' (using password: YES)",
      "Knex:Error Pool2 - Error: ER_ACCESS_DENIED_ERROR: Access denied for user
'root'@'192.168.1.26' (using password: YES)"
   ]
}
```

Bailing out if a variable isn't set

Ansible provides a "fail" module in its core that will simply raise an error if a condition is true.

```
- fail: msg="Bailing out. This play requires the database host to be set."
  when: databaseHost is not defined
```

Roles (reference)

Roles are a means of organizing playbooks in a such a way that variables, tasks, etc. can be automatically loaded based on file structure (sort of like how "group_vars" works). **THEY ARE NOT GROUPS; DO NOT USE THEM AS GROUPS.** If you have a role defined in a playbook, then that means that you *must* have some kind of associated data, e.g. tasks, vars, meta, etc. If you just want to use a role as a tag, then you would use either groups or tags. For example, when I wanted to make an "init_database.yml" script, I wanted it to apply to certain hosts, and I wanted to get variables from my "infra" group, so I only needed to make use of groups for this rather than roles.

If you're able to run Ansible locally, you can use "ansible-galaxy init <role name>" to set up the directory and file structure.

Any variables that you define in a "roles" subdirectory should only be used by that particular role. E.g. don't define a "remoteBranch" variable in the "common" role if you expect to use it from a "git pull" in the "webservers" role.

Playbooks

To run an Ansible playbook, you need to use **ansible-playbook**, *not* ansible.

A playbook is a set of tasks (which are really modules and can be found here).

Tags (reference)

If you want to indicate that certain plays or tasks are only for certain scenarios (or even *not* for certain scenarios), you can use tags to tag the plays/tasks, then later, you can run with something like this:

- ansible-playbook example.yml --tags "configuration,packages"
- ansible-playbook example.yml --skip-tags "notification"
- By default, Ansible runs as though you'd specified "--tags all"

A good example of when to use tags is having a "fast deploy" (which may skip creating a user, doing "npm install", etc.). You would tag your necessary steps as "fast_deploy" and then run only with --tags "fast_deploy"). Of course, you could also just tag them like "git pull" and "start service" and then run with those tags.

Copying general scripts

The only real thing to note here is that "chmod +x" on a script will produce a 755 mask, so you should just supply that manually. Also, /usr/local/bin is shared by all users, so don't put scripts there unless you're okay with that!

```
tasks:
  - name: Test creating shell script
    copy: src=./test.sh dest=/usr/local/bin mode=0755
```

Getting IP address of the current host

databaseHost: "{{ ansible_eth0.ipv4.address }}"

Running as root pre-1.9

Note: post-1.9, I believe there is "become_user", which will let you become a certain user but still give you the option of being root. However, the instructions below are for pre-1.9.

Note: your remote_user has to be a user that you have SSH access for.

Write "sudo: yes" at the playbook level (so at the same level as "hosts"). As soon as you do this, you'll need to consider passing "--ask-sudo-pass" (or simply "-K") to ansible-playbook. If you don't do this, Ansible will error out pretty quickly telling you that you need this argument.

From there on out, you can specify sudo_user (which by default seems to be your remote_user) as sort of a hack to switch users. For example, suppose I want to add a user at the beginning of my playbook, then I want to do everything else as that user afterward. I would do something like this:

```
---

- hosts: botland
  vars:
    deployUser: bldeploy
```

```
remote_user: adam

# Note: this really only needs to be applied to the adduser task, but after
# adding that user, we want to run as that user, so we simply change sudo_user
# for each individual task that needs to run as the new user.
sudo: yes

# Note: sudo is needed for installing pm2 globally
tasks:

 - name: Create a user
   user: name={{ deployUser }}
   sudo: yes

 - name: Create folder for AWS credentials
   file: path={{ homeDir }}/.aws state=directory
   sudo_user: "{{ deployUser }}"
```

Note: the "bldeploy" user that you create above will be accessible to you by doing "sudo su bldeploy" when you're on the machine. You probably won't be running in Bash when you first do that.

Splitting a playbook into multiple files

At any point, you can separate individual tasks or playbooks into their own files. For example, suppose I have this file:

```
# tasks.yml
---
- name: Start Overseer
  command: pm2 start main.js chdir={{ infraDir }}/Overseer
  sudo_user: "{{ deployUser }}"
  environment:
    PATH: "{{ ansible_env.PATH }}:{{ nodeGlobalBinFolder }}"
    REST_SERVER_PORT: 8080
    TCP_SERVER_PORT: 3000
    IS_AWS: "{{ isAws }}"
```

I could use this from another playbook by simply doing:

```
- include: ./tasks.yml isAws=true
```

Any variables that I have set already will be visible by tasks.yml, e.g. infraDir or deployUser.

Any variables you want to set from the command line can be done via the "--extra-vars" argument, e.g.:

```
ansible-playbook install_overseer.yml --extra-vars "restPort=8082 isAws=true"
```

Known hosts

If you run into a problem with being asked about known_hosts on launch, then look into this.