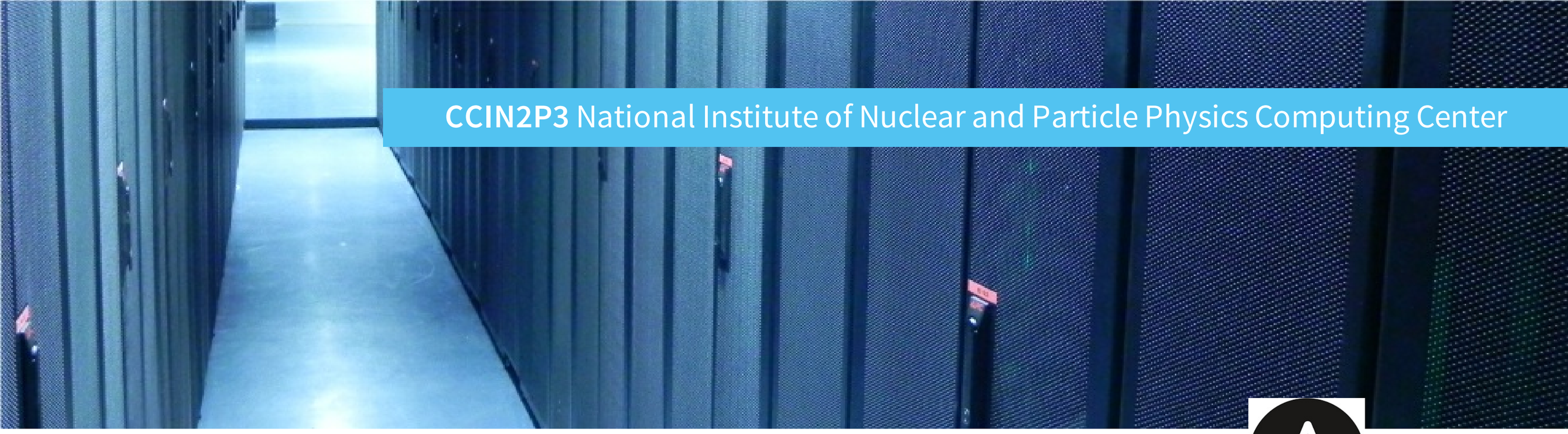


CCIN2P3 National Institute of Nuclear and Particle Physics Computing Center

ANSIBLE

YET ANOTHER AUTOMATION TOOL?

WEBINAIRE RI3 - 2018.03.22

J. BUSSERY & B. GUILLON

CCIN2P3

cnrs

Topics

- Automation: the why and the how
- Ansible 101: learning the basics
- A deeper dive into Ansible

Largely inspired by the book "Ansible: Up & Running".

AUTOMATION: WHY AND HOW?

Automate all the things!

Everyone wants to be a sysadmin

- Ever more complex architectures and environments
- Systems are distributed and heterogeneous
- Devops are trending
- I'm lazy!

Configuration management vs. deployment

Configuration management:

→ *Enforce the state of the system configuration.*

Deployment:

→ *Create, modify and destroy systems or parts of them.*

Here comes Ansible!

Ansible is free and opensource software.

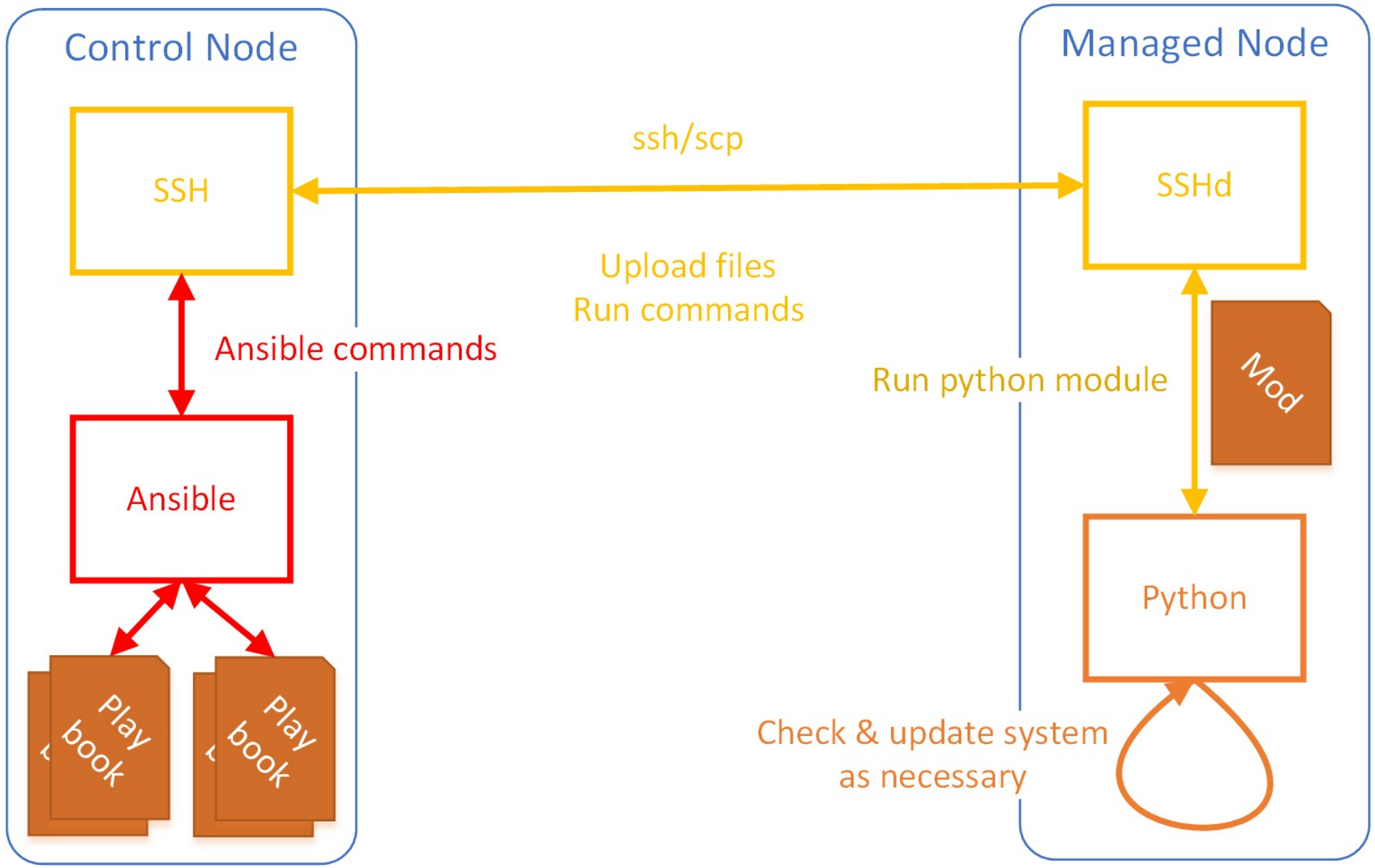
- Created by Michael DeHaan
- First release in 2012
- Support and GUI provided by Ansible Inc.
- Ansible Inc. acquired by RedHat in 2015

How it works

Ansible is:

- Written in Python
- Agentless (kind of)
- Push-based

ANSIBLE ARCHITECTURE



Domain Specific Language

Ansible provides a DSL (Domain Specific Language)

→ To describe configuration and deployment steps

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
```

Packaging logic

Modules are provided by Ansible.

- Logic as a package
- Ensures idempotence of operations
- Hundreds of community driven modules available

Play by the book

Create your plays using modules

→ Tasks call modules.

→ Assemble roles from tasks.

→ Build playbooks with various roles.

→ Run playbooks.

Exemple: Install and configure Nginx, deploy a webapp and set the firewall

Learning Ansible

Ansible has a very smooth learning curve

- Kept simple for simple things
- Yet complex stuff can be done

Scaling Ansible

Ansible scales well in various environments:

- Leverages SSH multiplexing capabilities to reach thousands of nodes
- Can manage Linux *(MacOS?)* and Windows boxes
- Provides GUI and interactivity when needed *(Tower, Semaphore)*
- Industry standard: Rackspace, RedHat, NASA, etc.

ANSIBLE 101: THE BASICS

- Installation
- Ad-hoc mode
- Inventory files
- Tasks
- Core modules
- Variables
- Loops
- Conditions
- Templates
- Playbooks
- More variables

Installing Ansible

Most distros have packaged Ansible

```
$ sudo apt install ansible
```

But get the latest from github or pypi

```
$ sudo pip install ansible
```

Latest stable version as of today

```
$ ansible --version
ansible 2.4.3.0
```

Use python virtualenvs

Using a python `virtualenv` is safer:

- Multiple versions can live on the same host
- No package dependency issues
- One environment per usage

Configure your SSH client

Make sure you can `ssh` easily:

- To the hosts you want to manage
- With the users that have proper permissions

For this, `~/.ssh/config` is your friend.

Ad-hoc mode

The `ansible` command, a.k.a. the "ad-hoc" mode.

- Fire up a single ansible module to targeted hosts
- Quite useful for:
  - one-shot operations
  - tests and debugging
  - monitoring
- But does not scale really well...

Ping & Pong

Let's ping one of our nodes:

```
$ ansible node01 -m ping
node01 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

So cool!

Command & Conquer

The `command` module lets you run arbitrary commands:

```
$ ansible node01 -m command -a "cat /etc/redhat-release"
node01 | SUCCESS | rc=0 >>
CentOS Linux release 7.4.1708 (Core)
```

Even cooler!

Breaking the shell

The `shell` module is very similar

- It runs the command through a local shell
- It loads the local user environment
- It might be the only solution for more complex things

But use it *wisely* since it's more dangerous

Inventory file

This file contains the list of nodes you want to manage:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

Either in *INI* or *YAML*.

```
---
all:
  hosts:
    mail.example.com
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

So many nodes, so many inventories

You can have multiple inventory files, for various scopes and use cases.

Use the `-i` option to specify the inventory file location:

```
$ ansible -i ~/path/to/hosts/file node01 -m ping
```

Otherwise, `/etc/ansible/hosts` is used.

Adding info to the inventory

The inventory can contain node specific information:

```
node01 ansible_port=5555 ansible_host=192.0.2.50
```

Basic pattern completion is also supported:

```
[webservers]
www[01:50].example.com

[databases]
db-[a:f].example.com
```

Ansible playbooks

This is a collection of tasks and configuration states

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
```

Running playbooks

Use `ansible-playbook` to run playbooks:

```
$ ansible-playbook ~/path/to/my/playbook
```

This command has a lot of useful options:

- Use `-i` to specify the inventory file
- Use `-l` to limit execution to a subet of nodes
- Use `-b` to run tasks as a specific user
- Use `-e` to overide variables at runtime
- Use `-h` to get the others...

Variables

Ansible modules use parameters.

- You will need variables to cover different scopes
- This is the only way to efficiently reuse code

```
---
- hosts: webservers
vars:
  http_port: 80
  max_clients: 200
[...]
```

```
node01 ansible_port=5555 ansible_host=192.0.2.50
```

Loops

Various keywords to loop over sets of items:

- Lists: `with_items`
- Hashmaps: `with_dict`
- Files: `with_files`
- Data sets: `with_together`
- Sequences: `with_sequence`
- Etc.

You can loop over variables defined elsewhere!

Looping over a list

Using the `with_items` keyword:

```
---
- hosts: webservers
  tasks:
  - name: add several users
    user:
      name: "{{ item }}"
      state: present
      groups: "wheel"
    with_items:
      - testuser1
      - testuser2
```

Looping on a variable

On the variable `somelist` of the playbook.

```
---
- hosts: webservers
vars:
  somelist:
    - testuser1
    - testuser2
tasks:
  - name: add several users
    user:
      name: "{{ item }}"
      state: present
      groups: "wheel"
    with_items: "{{ somelist }}"
```

Conditions

Using the `when` keyword, conditionnaly run tasks:

```
tasks:
- name: "shut down Debian flavored systems"
  command: /sbin/shutdown -t now
  when: ansible_os_family == "Debian"
```

Templates

Ansible leverages the Jinja2 template engine:

- To interpolate variables in Ansible DSL
- To generate files from templates
- To provide variable filters

Generating files from templates

Templates allow for an easy file generation:

```
# SSHD config file
# See the sshd_config(5) manpage for details
Port {{ sshd_port }}
PermitRootLogin {{ sshd_allow_root }}
[...]
```

Using the `template` module:

```
---
- hosts: webservers
vars:
  sshd_config_path: "/etc/ssh/sshd_config"
  sshd_port: "2222"
  sshd_allow_root: "no"
tasks:
  - name: configure ssh daemon
    template:
      src: templates/sshd_config.j2
      dest: "{{ sshd_config_path }}"
```

Variable interpolation

Jinja2 templating works nearly everywhere.

For variables:

```
---
- hosts: dbservers
vars:
  psql_config_dir: "/var/lib/pgsql"
  psql_config_file: "{{ psql_config_dir }}/postgresql.conf"
  psql_hba_file: "{{ psql_config_dir }}/pg_hba.conf"
tasks:
  - [...]
```

For tasks and loops as we have seen earlier...

Jinja2 filters

Jinja2 provides some bits of logic called *filters*.

- Specify a default value for a variable:

```
{{ my_variable|default('my_variable is not defined') }}
```

- Force a variable to be defined:

```
{{ variable | mandatory }}
```

- Get a random integer, from 0 to supplied end:

```
"{{ 59 |random}} * * * * root /script/from/cron"
  # => '21 * * * * root /script/from/cron'
```

- Extract an IP address from a CIDR string:

```
{{ '192.0.2.1/24' | ipaddr('address') }}
```

More playbooks

One file to rule them all?

```
---
- hosts: webservers, dbservers, more_hosts...
  vars:
    http_port: 80
    max_clients: 200
    more_vars: [...]
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: Adding even more tasks...
```

This is going to get messy...

Use roles!

Roles are:

- An independent set of tasks
- Associated resources

```
roles
├── common       # Common role loaded on every run of the playbook
│   ├── defaults    # Default variable values
│   ├── files       # Files to upload on remote nodes
│   ├── handlers    # Tasks run asynchroneously
│   ├── meta        # Role metadata (author, unit tests, doc ...)
│   ├── scripts     # Scripts run on remote nodes
│   ├── tasks       # Tasks
│   ├── templates   # Templates processed on remote nodes
│   └── vars        # More variables (usualy condition specific)
├── dbserver    # Role to manage database servers
│   └── ...
└── webserver   # Role to manage web servers
    └── ...
```

Simple playbooks

Playbooks are now simpler

→ Just a collection of roles

```
---
- hosts: dbservers
  roles:
    - common
    - dbserver
- hosts: webservers
  roles:
    - common
    - webserver
- hosts: all
  roles:
    - common
    - { role: debian_stock_config,
        when: ansible_os_family == 'Debian' }
```

Roles can be shared back to the community.

More variables

Ansible provides several variable scopes:

- Defaults
- Role or playbook level
- Host level
- Group level

Variable precedence

Variables are preprocessed before the playbook execution. An order of precedence exists, from least to most important:

1. Role or playbook defaults (`defaults` folder)
2. Role or playbook variables (`vars` folder)
3. Group specific variables (`group_vars` folder)
4. Host specific variables (`host_vars` folder)
5. Commande-line override (using the `-e` option)

LET'S TAKE A DEEPER DIVE

- Code management
- Dynamic inventories
- Secrets management
- Privilege escalation
- Even more variables
- Advanced templates
- Writing your own modules
- Error handling & debugging
- Task scheduling
- Playbook interaction
- Privilege delegation
- Advanced playbook organization

Code management

Please, version your code (git, svn, cvs...)

- You benefit from your dev history
- You'll be able to work with others
- You'll be able to share what you did

Ansible configuration files

A lot of configuration scopes are available.

- `ANSIBLE_CONFIG`, an environment variable
- `ansible.cfg`:
  - In the current directory
  - In the home directory
- `/etc/ansible/ansible.cfg`, at system level

Use them. Make your life simpler.

Dynamic inventories

Static inventory files:

- It's a pain to update
- Not necessarily consistent with reality

And then they invented dynamic inventories!

So dynamic

Basically, Ansible eats JSON in the morning

→A dynamic inventory is just any script that produces node descriptions in JSON.

```
$ dyninv --list
{
  "_meta": {
    "hostvars": {
      "node01": {
        "ansible_host": "172.17.??.??",
        "ansible_ssh_host": "172.17.??.??",
        "openstack": {
          "OS-DCF:diskConfig": "MANUAL",
          "OS-EXT-AZ:availability_zone": "nova",
          "OS-EXT-SRV-ATTR:host": "ccosndli0006",
          "OS-EXT-SRV-ATTR:hypervisor_hostname": "ccxxxx.in2p3.fr",
          "OS-EXT-SRV-ATTR:instance_name": "instance-000xxxxxx",
          "OS-EXT-STS:power_state": 1,
          "OS-EXT-STS:task_state": null,
```

Community inventories

Several dynamic inventories are provided by the community:

→LDAP, AWS, Openstack ...

You can write or fork your own (refer to the docs).

So many secrets

Secrets ending up in the code as clear text.

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
    app_admin_passwd: Str0nG1337p4ssWD!
  remote_user: root
  tasks:
    - [...]
```

→ Not a very good idea... (SCM history & sharing)

The Vault

Cryptography is a good solution to protect secrets.

Handle file encryption using `ansible-vault`:

```
$ ansible-vault --help
Usage: ansible-vault [create|decrypt|edit|encrypt|encrypt_string|rekey|view] [options] [vaultfile

encryption/decryption utility for Ansible data files

Options:
  --ask-vault-pass      ask for vault password
  -h, --help            show this help message and exit

  [...]

 See 'ansible-vault <command> --help' for more information on a specific command.
```

It uses AES256 with a symmetric encryption key.

Encode strings directly

You can also directly encode strings:

```
$ ansible-vault encrypt_string "mypassword"
!vault |
          $ANSIBLE_VAULT;1.1;AES256
          35626362316137653763373333663235393030633061393933137393535666632613
          38333238396461643731623961346364666535643061346 10a623463386337393361 3
          38366136663834623135393665363962616436333533643736343564323030643036 3
          65366234303530373 70a3166326264373464353566333261616532313963333736646
          6334
Encryption successful
```

Decode at runtime

And then put them in playbooks:

```
---
- hosts: webservers
  vars:
    app_admin_passwd: !vault |
      $ANSIBLE_VAULT;1.1;AES256
      35626362316137653763373333663235393030633061393933313739353566663261323
      38333238396461643731623961346364666535643061346 10a623463386337393361313
      38366136663834623135393665363962616436333533643736343564323030643036326
      65366234303530373 70a31663262643734643535663332616165323139633337366464 3
      6334
```

Provide the encryption key at runtime

```
$ ansible-playbook /path/to/playbook --ask-vault-password encr_key
```

Privilege escalation

Ansible uses ssh

- Modules are run as the ssh user
- With the same privileges

When it's not enough:

```
- name: Configure the ssh daemon
  template:
    src: sshd_config.j2
    dest: /etc/ssh/sshd_config
  become: yes
```

```
- name: Configure postgresql ACLs
  template:
    src: pg_hba.conf.j2
    dest: /var/lib/pgsql/pg_hba.conf
  become: yes
  become_user: postgres
```

Even more variables

When a playbook runs, the `setup` module is triggered.


```
"ansible_facts": {
 "ansible_all_ipv4_addresses": [
     "172.17.??.??"
 ],
 "ansible_all_ipv6_addresses": [
     "fe80::f816:3eff:fe9c:3135"
 ],
 "ansible_apparmor": {
     "status": "disabled"
 },
 "ansible_architecture": "x86_64",
 "ansible_bios_date": "04/01/2014",
 "ansible_bios_version": "1.10.2-3.el7_4.1",
 "ansible_cmdline": {
     "BOOT_IMAGE": "/vmlinuz-3.10.0-693.17.1.el7.x86_64",
```


Protip: you can disable it and you can cache it as well.

Ansible facts

Use Ansible facts in your playbooks:

```
---
- hosts: all
  tasks:
    - name: Get node IP address
      debug:
        msg: "My IPv4 is: {{ ansible_facts.ansible_default_ipv4.address }}"
```

Get facts from other nodes included in the same run:

```
---
- hosts: all
  tasks:
    - name: Get proxy IP address
      debug:
        msg: "The proxy IP address is: {{ hostvars.proxy01.ansible_default_ipv4.address }}"
```

Cascading templates

Jinja is a powerful tool:

- Conditions and loops can be used in templates
- Templates can be an assembly of subtemplates

```
# My config file
{% if server_role == "master" %}
Some master config directives...
{% include "templates/master_directives.j2" %}
{% else %}
Some slave config directives...
{% fi %}
```

The Jinja2 documentation is full of interesting tricks.

Write your own modules

The `command` module is not idempotent.

→Write your own modules.

- It's not that difficult
- Saves time on the long run
- Ensures idempotence (if you did)

If you feel like you did a good enough job: share it!

Error handling

Sometimes, modules fail. Sometimes it's expected.

```
tasks:
  - name: Checking database cluster status
    command: /usr/pgsql-10/bin/pg_ctl status -D /var/lib/pgsql/10/data
    ignore_errors: yes
    register: psql_status
```

The keyword `ignore_errors` lets you handle it your way.

Debugging

The try & fail method is a good as any.

- The `debug` module is your *print*.
- The `fail` module is your *breakpoint*.

The `ansible` commands can be more verbose:

```
$ ansible -vvvvv /my/playbook
```

Task scheduling

By default, Ansible runs step by step in parallel.

Other strategies exist:

- parallel
- serial
- linear
- free

```
- hosts: all
strategy: free
tasks:
...
```

Or any other custom strategy (plugins)

Interaction

The `prompt` module to interact with the playbook:

```
---
- hosts: all
remote_user: root
vars_prompt:
  - name: "name"
    prompt: "what is your name?"
  - name: "quest"
    prompt: "what is your quest?"
  - name: "favcolor"
    prompt: "what is your favorite color?"
```

You can encrypt on the fly:

```
vars_prompt:
- name: "my_password2"
  prompt: "Enter password2"
  private: yes
  encrypt: "sha512_crypt"
  confirm: yes
  salt_size: 7
```

Privilege delegation

Ansible requires ssh access to the managed nodes

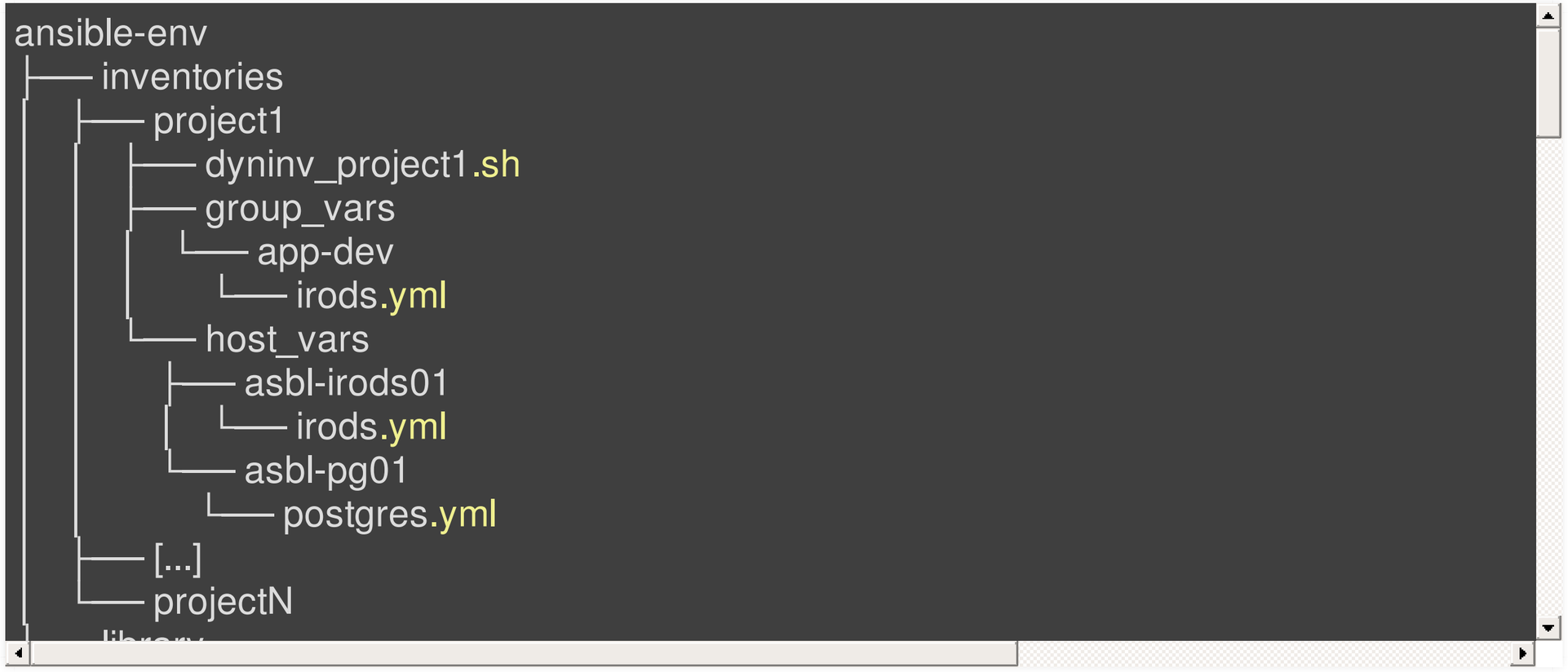
→This can be a problem.

Delegation tools like Ansible Tower, Semaphore or Rundeck could be the solution.

Advanced playbook organization

- A single hierarchy of git submodules
- Independant collections of roles
- Dynamic inventories and multiple variable scopes
- Shared libraries
- Playbooks as collections of roles

Advanced playbook organization



ANY QUESTIONS?

POWERED BY REVEAL.JS