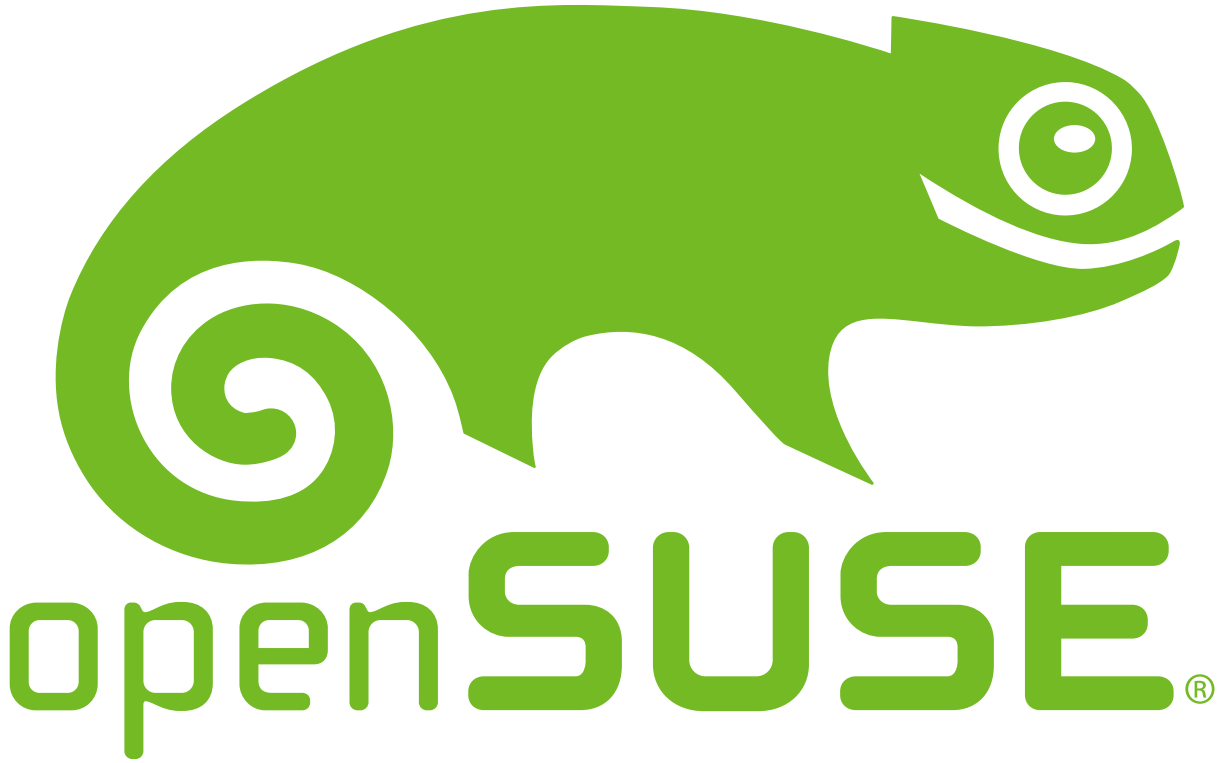


introduction to Ansible

Aug 2023

andrea.manzini@suse.com

A little background

Automation

is about taking manual processes and placing technology around them to make them repeatable. Automation is the key to speed, consistency, scalability and repeatability.

Think about car factories in the 1900 vs modern robot-automated industries!

Benefits of Automation

What does automation enable:

- Scalability
- Reliability
- Repeatability
- Consistency
- Auditability
- Security

Some widely-known configuration Management Tools

listed in no specific order:

- Salt
- Puppet
- Chef
- Ansible

And many others; recently, Ansible creator Michael DeHaan shared about a new OSS project he's working on.

What is Ansible?

Ansible is a tool for:

- Configuration Management
- Deploying software
- Orchestration
- Provisioning

ANSIBLE

andrea.manzini@suse.com

Ansible features

- Based on Python
- Agentless (only needs Python on remote host)
- Only requires SSH
- Push based
- Long history and big community of contributors

A bit of history

- The term "ansible" was coined by Ursula K. Le Guin in her 1966 novel Rocannon's World, and refers to fictional instantaneous communication systems. It was also used in the science fiction novel Ender's Game
- The Ansible tool was developed in 2012 by Michael DeHaan, the author of the provisioning server application Cobbler and co-author of the Fedora Unified Network Controller (Func) framework for remote administration.
- Ansible, Inc. (originally AnsibleWorks, Inc.) was the company founded in 2013 by DeHaan, Timothy Gerla, and Saïd Ziouani to commercially support and sponsor Ansible.
- Red Hat acquired Ansible Inc. in October 2015. Legal note: Ansible is a registered trademark of Red Hat / IBM and is GPLv2 licensed without copyright assignment.

The Idempotency Concept

Configuring systems using shell script can be simple and effective, but:

- require to follow complex logic
- hard to grasp env variables scoping rules
- portability issues between distro and/or operating systems
- they are not repeatable (e.g. **idempotent**).

With Ansible we solve this problem by writing the final destination state we want to reach; the tool makes only the necessary changes. Let's see an example.

Idempotency example #1

We want to add a specific user to a system. Imperative way:

```
$ adduser / useradd -b -u -d -G ...
[...other stuff...]
$ adduser / useradd -b -u -d -G ...
ERROR: user 'adam' already exists
```

Declarative way: we **state** that the user must exist in the system

```
# Ensure the user Adam exists in the system
    - name: Add the user 'Adam' with a specific uid and a primary group of 'sudo'
      ansible.builtin.user:
        name: adam
        comment: Adam Engineer
        uid: 1077
        group: sudo
        createhome: yes
        home: /home/users/adamlis
```

Idempotency example #2

SAP HANA DEPLOYMENT EXTRACT (source)

from

```
echo "vm.swappiness=60" >> /etc/sysctl.d/90-sap_hana.conf
echo "kernel.msgmni=32768" >> /etc/sysctl.d/90-sap_hana.conf
...
sysctl -p /etc/sysctl.d/90-sap_hana.conf
```

to

```
- name: setting kernel tunables
  sysctl: name={{ item.name }} value={{ item.value }} state=present
          sysctl_set=yes reload=yes
  with_items:
    - { name: kernel.msgmni, value: 32768 }
    - { name: vm.swappiness, value: 60 }
```

installing Ansible

for experimental purpose in our workshop we can use any machine or also a development container:

```
$ toolbox enter
```

When you have shell access to container, installation is simple:

```
# zypper install ansible
```

This need to be run only on the "main" node. Ansible by default works via ssh connection sending/pushing commands to other machines.

Ansible Hello World

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

this means that Ansible is correctly installed and working. `-m` stands for "use this module". The `ping` module does not **change** anything on the host, it simply reply back to test the communication.

Modules

Modules are discrete units of code that can be used from the command line or in a playbook task. Ansible executes each module, usually on the remote managed node, and collects return values. In Ansible 2.10 and later, most modules are hosted in collections.

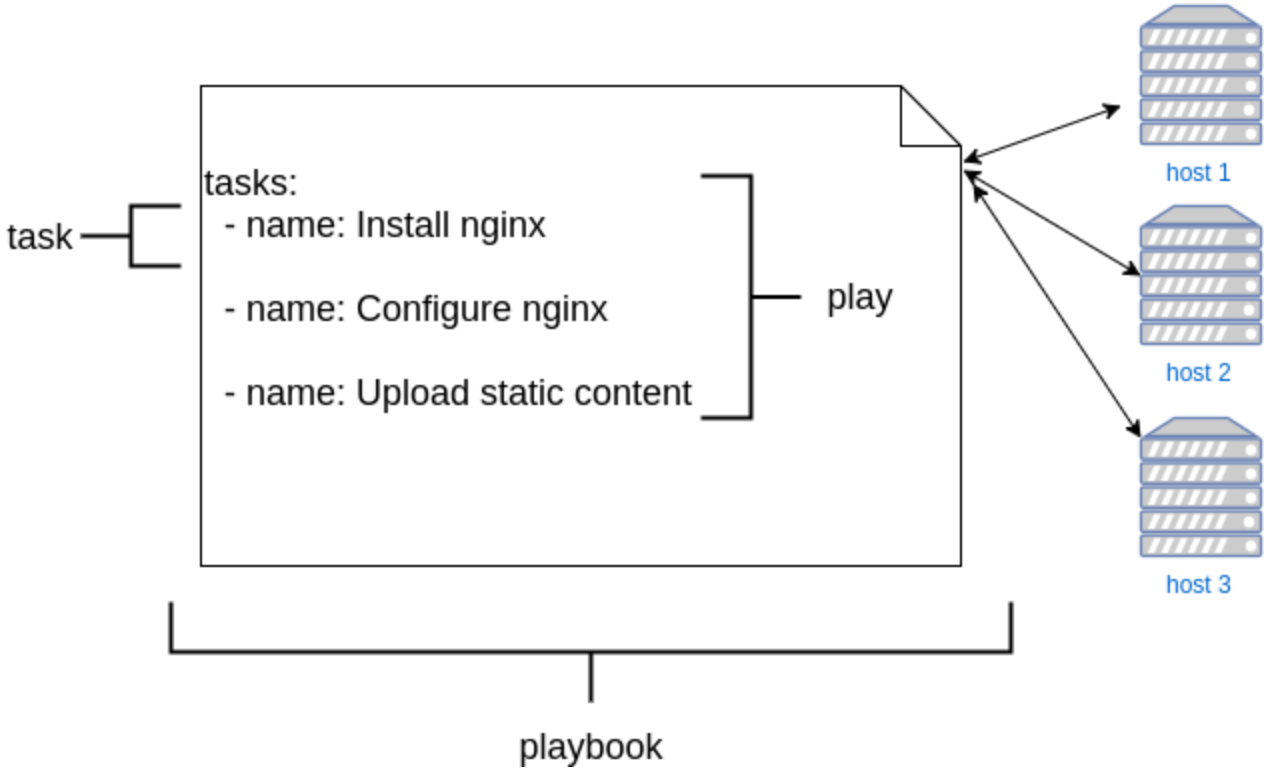
you can inspect the code being executed, for example the `ping` builtin module lives at

`/usr/lib/python3.11/site-packages/ansible/modules/ping.py`

you can also check out the Ansible Collection, it contains hundreds of ready-made modules, and when you can't find the right one, it's rather easy to write your own

Ansible Basic Terminology

- **Task** : A single action to perform, e.g. invoke a module with some parameters
- **Play** : A collection of tasks
- **Playbook** : YAML file containing one or more plays



PLAYBOOK EXAMPLE: INSTALL & CONFIGURE APACHE WEBSERVER

```
# begin of playbook
---
- name: first play to install and start apache
  hosts: localhost
  connection: local
  become: yes
  tasks:
    - name: install apache2 (task1)
      zypper: name=apache2 state=latest
    - name: start apache2 (task2)
      systemd:
        state: started
        name: apache2
- name: second play, includes another play from a file
  ansible.builtin.import_playbook: otherplays.yaml
# end of playbook
```

More Terminology

- **Module** : Blob of Python code which is executed to perform task
- **Inventory**: Array of hosts (stored in a file or specified via command-line) where tasks will be run
- **Role**: A mechanism for reusing and organizing code in Ansible in a standard hierarchy
- **Facts**: Builtin variables related to remote systems (i.e. ipaddress, hostname, cpu, ram, disk, etc.). They are filled-in by the `setup` module which is always run by default. Let's see the facts in our machine:

```
$ ansible localhost -m setup | less
```

running Ansible

There are two ways to run ansible:

1. ad hoc / console

Run a single task, as we'll do next, ideal for interactive usage and experiments

```
ansible <pattern> [options]
```

2. Playbook

Run multiple tasks (a *playbook*) sequentially, ideal for scripting/automation

```
ansible-playbook <pattern> [options]
```

other utilities

- ansible-pull to pull playbooks from a VCS and run them (used tipically in a scheduled cron job / timer)
- ansible-vault to manage secrets, tokens and passwords
- ansible-doc display documentation about installed modules
- ansible-config view the current configuration as it will be used
- ansible-inventory view the current inventory as it will be used

Inventory 1/2

Ansible inventory is the list of hosts where we want to apply our configuration. The simplest inventory is a single file with a list of hosts and groups. The default location for this file is `/etc/ansible/hosts`. You can specify a different inventory file at the command line using the `-i <path>` (even multiple times) option or in `ansible.cfg` configuration using inventory. On the same command line you can also specify a comma-separated list of hosts.

Lets' see some example.

How to create the most basic inventory

```
 echo hostname.domain.com > inventory.txt
```

Example: we use ansible to check memory usage on all the servers listed

```
$ ansible -i inventory.txt all -a "free -m"
```

alternate inventory on command line

```
$ ansible -i myhost1,myhost2,myhost3 all -a "free-m
```

Let's explain the last command line

- `-i` option lets us pass the inventory (via filename or list of hosts)
- `all` means we want to run the next command on all hosts on the inventory
- `-a` gives the execution `arguments` to the module, which is by default command when not specified . Would be the same to run

```
$ ansible -i inventory.txt all -m command -a "free -m"
```

Inventory 2/2

- hosts in an inventory can be organized into **groups**. Groups can also be nested, e.g. it's possible to create groups of groups.
- Inventory can be enriched with `host_vars` (variables with values associated with a single host) and/or `group_vars` (variables with values associated with a group, all host in that group will get the same variable)
- The inventory can be in `yaml` format or in `ini` format and can be made dynamic, e.g. user provides a script that outputs the list of machines at runtime (there are many already made for most cloud providers, cmdb, etc.)

See documentation at
https://docs.ansible.com/ansible/latest/inventory_guide/intro_inventory.html

SIMPLE INVENTORY EXAMPLE

```
machine-debug.example.suse.de
another_server-1.example.suse.de

[virtual_machines]
openqa-worker1.example.suse.asia
srv01.example.suse.asia
srv02.example.suse.asia
srv03.example.suse.asia
srv04.example.suse.asia
srv05.example.suse.asia

[baremetal]
baremetal1.example.suse.de
baremetal2.example.suse.de

[asia]
openqa-worker1.example.suse.asia
srv0[1-5].example.suse.asia

[europe]
machine-debug.example.suse.de
baremetal[1-2].example.suse.de
another_server-1.example.suse.de
```

INVENTORY EXAMPLE with some GROUP VARS

```
[asia]
openqa-worker1.example.suse.asia
srv0[1-5].example.suse.asia

[europe]
machine-debug.example.suse.de
baremetal[1-2].example.suse.de
another_server-1.example.suse.de

[asia:vars]
ntp_server=time-sync-server.example.suse.com
nfs_path="another-nfs-server.suse.asia:/folder/blabla/pckgs"

[europe:vars]
ntp_server=ntp.suse.de
nfs_path="11.22.33.44:/folder nfs-server.suse.de:/mnt/myfolder"
```

How Ansible talks to hosts ?

By default, Ansible uses native OpenSSH, because it supports *ControlPersist* (a performance feature), Kerberos, and options in `~/.ssh/config` such as Jump Host setup.

By default, Ansible connects to all remote devices with the user name you are using on the control node. If that user name does not exist on a remote device, you can set a different user name for the connection.

If you just need to do some tasks as a privileged user, use privilege escalation:

```
- name: Ensure the httpd service is running
  service:
    name: httpd
    state: started
  become: true
```

Facts 1/2

By default, whenever you run an Ansible playbook, Ansible first gathers some information (“facts”) about each host in the play.

```
$ ansible-playbook playbook.yml
PLAY [group] ***************************************************************
GATHERING FACTS ************************************************************
ok: [host1]
ok: [host2]
ok: [host3]
```

Facts 2/2

Facts can be extremely helpful when you’re running playbooks; you can use gathered information like host IP addresses, CPU type, disk space, operating system information, and network interface information to change when certain tasks are run, or to change certain information used in configuration files.

to see all available facts on your pc:

```
$ ansible localhost -m ansible.builtin.setup
```

Local Facts

Another way of defining host-specific facts is to place a `.fact` file in a special directory on remote hosts, `/etc/ansible/facts.d/`. These files can be either JSON or INI files, or you could use *executables/scripts* that return JSON. As an example, create the file `/etc/ansible/facts.d/settings.fact` on a remote host, with the following contents:

```
[users]
owner=jane
normal=jim,john
```

Next, use Ansible's setup module to display the new facts on the remote host:

```
$ ansible hostname -m setup -a "filter=ansible_local"
```

Beyond Ad-Hoc commands: playbooks

The Ad-Hoc `ansible` command we used till now are limited to a single task, but most of the time the activity to perform is simply not a single command. We can group many tasks together to form a **play**, and many plays to form a **playbook** (and again many playbooks to form a **role**).

FROM A BASIC SHELL SCRIPT

Let's say we want to install and configure a web server. Start with a basic shell script:

```
#!/bin/sh
# Install Apache.
zypper install -y apache2
# Copy the configuration file to use in production.
cp my_httpd.conf /etc/apache2/httpd.conf
# Start Apache and configure it to run at boot.
systemctl enable apache2.service
systemctl start apache2.service
```

to *run* this: `$ sudo ./install-apache.sh`

this script has some issues; can you spot them ?

... TO A (BAD) ANSIBLE PLAYBOOK ...

```
---
- hosts: all
  tasks:
    - name: Install Apache.
      command: zypper install -y apache2
    - name: Copy configuration files.
      command: >
        cp my_httpd.conf /etc/apache2/httpd.conf
    - name: Start Apache and configure it to run at boot.
      command: systemctl enable apache2.service
    - command: systemctl start apache2.service
```

to *run* this: `$ sudo ansible-playbook install-apache.yml`

question: why this is quite bad practice ?

… TO A BETTER ONE

```
---
- hosts: all
  become: yes
  tasks:
    - name: Install Apache.
      package:
        name: apache2
        state: latest
    - name: Copy configuration files
      copy:
        src: my_httpd.conf
        dest: /etc/apache2/httpd.conf
        owner: root
        group: root
        mode: 0644
    - name: Make sure Apache is started now and at boot
      systemd:
        name: apache2
        enabled: true
        state: started
```

Variables

With Ansible, you can execute tasks and playbooks on multiple different systems with a single command. To represent the variations among those different systems, you can create variables with standard YAML syntax, including lists and dictionaries. You can define these variables in your playbooks, in your inventory, in re-usable files or roles, or at the command line. You can also create variables during a playbook run by registering the return value or values of a task as a new variable.

👉 variables can be of many types: boolean, numerical, string, list, dictionary

👉 variables can be defined in many scopes. See variable precedence

👉 TIP: use `debug` module to display variable values during the execution

When to quote variables (a YAML gotcha)

If you start a value with `{{ foo }}`, you must quote the whole expression to create a valid YAML syntax.

```
- hosts: app_servers
  vars:
      app_path: {{ base_path }}/myapp
```

You will see: `ERROR! Syntax Error while loading YAML.` If you add quotes, Ansible works correctly:

```
- hosts: app_servers
  vars:
       app_path: "{{ base_path }}/myapp"
```

Register variables

Sometimes you will want to run a command, then use its return code, stderr, or stdout to determine whether to run a later task. For these situations, Ansible allows you to use register to store the output of a particular command in a variable at runtime.

```
    - name: Run a shell command and register its output as a variable
      ansible.builtin.shell: /usr/bin/foo
      register: foo_result
      ignore_errors: true

    - name: Run a shell command using output of the previous task
      ansible.builtin.shell: /usr/bin/bar
      when: foo_result.rc == 5
```

set_facts vs register variables

they do almost the same thing, but with one difference:

facts are host-specific

when you set a fact using set_facts module, it is specific to the host within which task is currently running. As documentation says : Variables are set on a host-by-host basis just like facts discovered by the setup module. If your playbook has multiple hosts then you can not share a fact set using set_facts from one host to another.

if/then/else conditionals

A task can be conditionally executed with the `when:` keyword.

```
tasks:
  - name: Shut down CentOS 6 and Debian 7 systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] == "6") or
          (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version'] == "7")
```

see more example on the documentation

Loops / Iteration 1/2

Repeated tasks can be written as standard loops over a simple list of strings. You can define the list directly in the task or keep the values in a variable

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item }}"
    state: present
    groups: "developers"
  loop:
     - joe
     - frank
     - "{{ another_big_list_of_users }}"
```

Loops / Iteration 2/2

You can use the until keyword to retry a task until a certain condition is met. Here's an example:

```
- name: Retry a task until a certain condition is met
  ansible.builtin.shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

for details please see documentation

Handlers

Sometimes you want a task to run only when a change is made on a machine. For example, you may want to restart a service if a task updates the configuration of that service, but not if the configuration is unchanged. Ansible uses handlers to address this use case. **Handlers are tasks that only run when notified.**

See documentation example

Beyond the basics

- delegation / local actions (example: send a mail to notify)
- manage pauses with `wait_for` or `prompt`
- error control: `ignore_errors` / `failed_when`
- tags / filtering
- blocks
- import, include
- reboot control
- unit testing

Templating

When we want to refer to some variable content, introduce some logic expressions or provide a file, we can use the **Jinja2** template engine embedded in Ansible.

A template contains variables and/or expressions, which get replaced with values when a template is rendered; and tags, which control the logic of the template. The template syntax is heavily inspired by Django and Python.

- most common is `{{ }}` for Expressions (emit the template output)
- there is also {% %} for Statements and `{# #}` for Comments

See template documentation

Templating Example

```
$ cd ansible_examples
$ ansible-playbook -i inventory.txt motd.yml
$ cat /tmp/motd
```

exercise: we want to give some control to the user, who can for example change the destination file or include/exclude IPV6 addresses. How can we achieve that ?

A word about filters

sometimes you will find expression like `{{ foo | bar}}`. Here `bar` is a filter that takes the output of another expression as input and produces another output, *just like unix shell pipes*.

- see the list of built-in filters in the official Jinja2 template documentation
- You can also use Python methods to transform data.
- You can create custom Ansible filters as plugins
- ansible.builtin also provide a lot of filters as plugins

Ansible vault : Keeping secrets secret

If you use Ansible to fully automate the provisioning and configuration of your servers, chances are you will need to use passwords or other sensitive data for some tasks, whether it's setting a default admin password, synchronizing a private key, or authenticating to a remote service.

It's better to treat passwords and sensitive data specially, and there are two primary ways to do this:

1. Use a separate secret management service, such as Vault by HashiCorp, Keywhiz by Square, or a hosted service like AWS's Key Management Service or Microsoft Azure's Key Vault.
2. Use Ansible Vault, which is built into Ansible and stores encrypted passwords and other sensitive data alongside the rest of your playbook.

Ansible Vault

How it works:

Ansible Vault works much like a real-world vault:

1. You take any YAML file you would normally have in your playbook (e.g. a variables file, host vars, group vars, role default vars, or even task includes!), and store it in the vault.
2. Ansible encrypts the vault (‘closes the door’), using a key (a password you set).
3. You store the key (your vault’s password) separately from the playbook in a location only you control or can access.
4. You use the key to let Ansible decrypt the encrypted vault whenever you run your playbook.

What are Ansible roles?

Roles are a way to group multiple tasks together into one container to do the automation in very effective manner with clean directory structures.

Roles are set of tasks and additional files for a certain role which allow you to break up the configurations.

It can be easily reuse the codes by anyone if the role is suitable to someone.

It can be easily modify and will reduce the syntax errors.

an example Ansible Role can be the one to install a WordPress website. It includes a web server, php, a database, the application and all needed configurations.

Ansible galaxy 1/2

Ansible roles are powerful and flexible; they allow you to encapsulate sets of configuration and deployable units of playbooks, variables, templates, and other files, so you can easily reuse them across different servers.

It’s annoying to have to start from scratch every time, though; wouldn’t it be better if people could share roles for commonly-installed applications and services? Enter Ansible Galaxy.

Ansible Galaxy, or just ‘Galaxy’, is a repository of community-contributed Ansible content. There are thousands of roles available which can configure and deploy common applications, and they’re all available through the ansible-galaxy command.

Ansible Galaxy 2/2

Galaxy offers the ability to add, download, and rate roles. With an account, you can contribute your own roles or rate others' roles (though you don't need an account to use roles).

```
$ ansible-galaxy role install geerlingguy.apache \
geerlingguy.mysql geerlingguy.php
```

A LAMP server in seven lines of YAML

```
# file: lamp-setup.yml
- hosts: all
  become: yes
  roles:
    - geerlingguy.mysql
    - geerlingguy.apache
    - geerlingguy.php
    - geerlingguy.php-mysql
```

```
$ ansible-playbook -i path/to/custom-inventory lamp-setup.yml
```

Anatomy of a Role

a role has only 2 mandatory subfolders:

```
role_name/
  meta/
  tasks/
```

If you create a directory structure like the one shown above, with a `main.yml` file in each directory, Ansible will run all the tasks defined in `tasks/main.yml` if you call the role from your playbook using the following syntax:

```
---
- hosts: all
  roles:
    - role_name
```

Role scaffolding

TIP: to easily create the directory structure for a role, we can use

```
$ ansible-galaxy role init role_name
```

Running this command creates an example role in the current working directory, which you can modify to suit your needs. Using the **init** command also ensures the role is structured correctly in case you want to someday contribute the role to Ansible Galaxy.

Troubleshooting / Debugging tips

- you can dump variable values at runtime using the debug module
- you can run playbooks step-by-step with `--step` option or jump directly at some task with `--start-at-task=<taskname>`
- you can add tags to a playbook and filter/run only on certain tags
- use `check_mode` to syntax check a playbook before running
- use `diff_mode` (when supported) to see what would be change
- inside a playbook you can invoke an interactive debugger

example

```
- name: Determine a system's registration status
  command: SUSEConnect --status
  register: suse_connect_status  # let's save result in a variable
  changed_when: false # this will never changes state of the system
  check_mode: false # and don't run this task in check mode

- name: Parse registration status
  set_fact:
    # uses stdout of previous command (which is in json) format
    # to set value of a fact variable
    sc_status: "{{ suse_connect_status.stdout | from_json }}"
```

[more will follow up]

let's look at real use case

from the project https://sap-linuxlab.github.io/

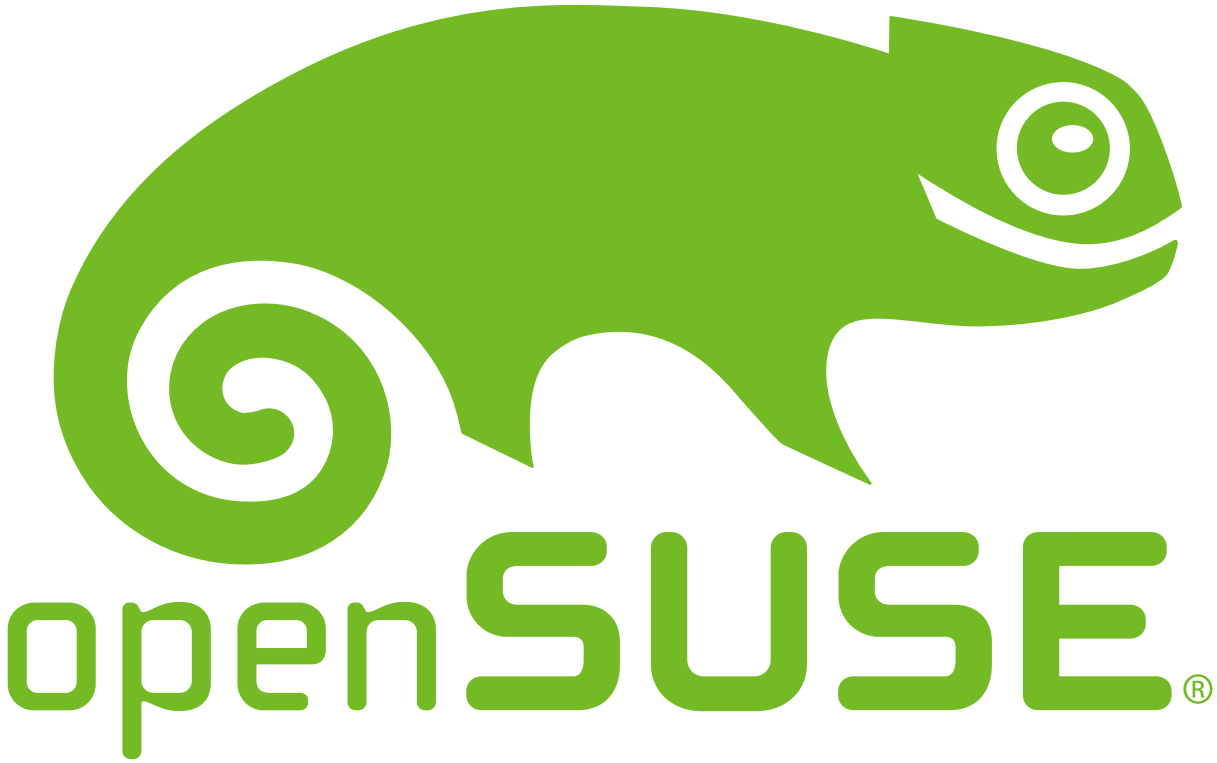
let's find some concrete examples and follow code from the roles in the repo

- SAP General preconfigure
- SAP storage setup
- HANA install

andrea.manzini@suse.com

Thanks!

These slides are Open Source and live in a github repository, feel free to improve them ♥



andrea.manzini@suse.com