UniLaSalle Amiens

DEVOPS TOOLS (IAC)

HOW TO CREATE AND MAINTAIN INFRASTRUCTURE AS CODE

CHAPTER 1
INTRODUCTION TO DEVOPS

UniLaSalle
Amiens

SUMMARY

UniLaSalle Amiens

Word origin

Contraction of the word developer and operation (sysadmin, netadmin, dbadmin)

Definition from the web

DevOps (a portmanteau of "development" and "operations") **is the combination of practices and tools** designed to **increase an organization's ability to deliver applications and services** faster than traditional software development processes

In summary:

- DevOps is a methodology to achieve software development
- DevOps setup practises and tools
- These practises and tools enhance delivery (fiability, speed)

Infrastructure

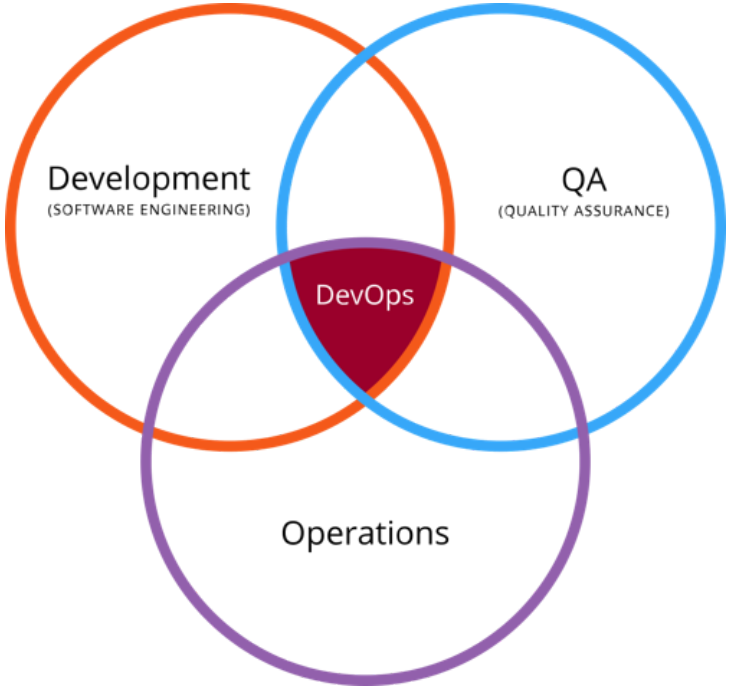
IT infrastructure provides all the necessary compute, storage, networking and software components necessary to deliver a service.

- Every infrastructure component is unique and special.
  - Built at different times with slightly different processes.
  - Built by different people with different levels of experience.
  - "Just make it work."
- Infrastructure is rarely replaced and is fanatically supported throughout its lifecycle.
- Infrastructure changes are carefully controlled by a Change Advisory Board (CAB).

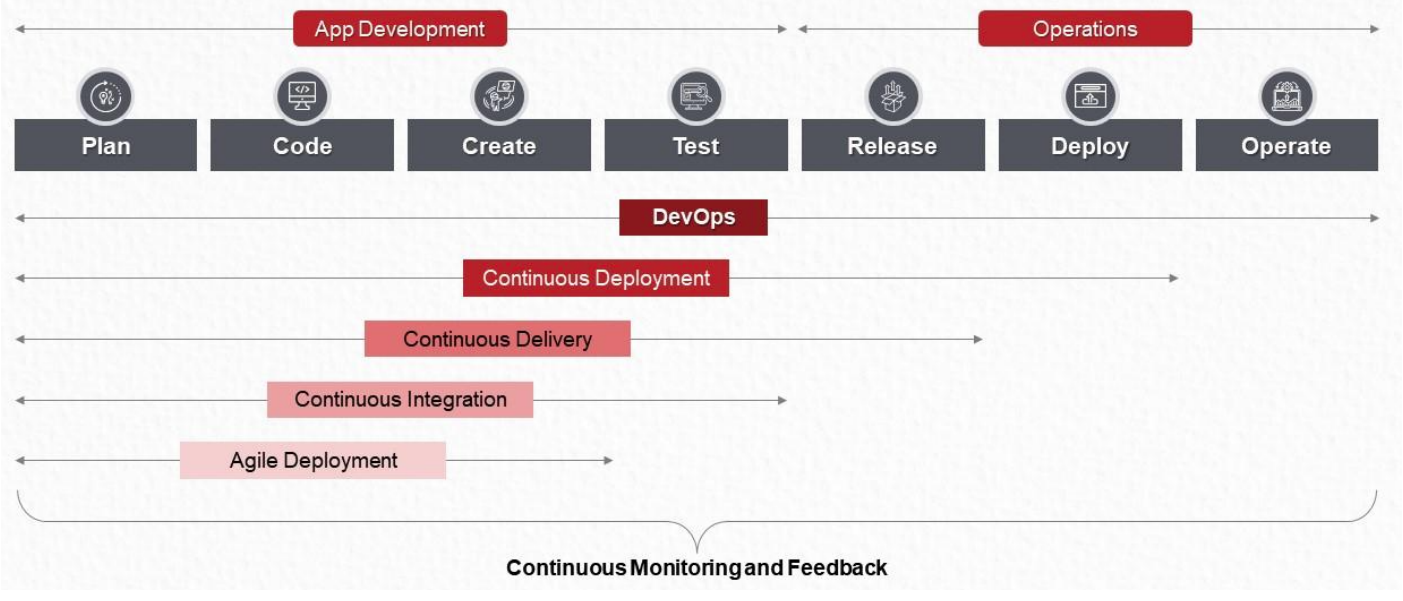
Role of IT operations staff:

- Gatekeepers for all IT services in the organization.
  - "The Office of No."
- Often partitioned into specialist teams:
  - Network engineers
  - Storage engineers
  - Security analysts
  - DBAs
- Ops team focused on preventing infrastructure failure.
  - "Keep the lights on."

- Faster time to market
- Higher ROI
- Improved collaboration
- Better efficiency
- Preventive or early correction of issues


Development
(SOFTWARE ENGINEERING)
QA
(QUALITY ASSURANCE)
DevOps
Operations


code
plan
build
Dev
release
Ops
deploy
Operate
test
monitor


App Development
Operations
Plan
Code
Create
Test
Release
Deploy
Operate
DevOps
Continuous Deployment
Continuous Delivery
Continuous Integration
Agile Deployment
Continuous Monitoring and Feedback

- **Plan:** Organize and schedule tasks
- **Code:** Code development and review
- **Build:** Build the source code
- **Test:** Implement code tests (unitary, integration, non regression)
- **Release:** Prepare code for deployment
- **Deploy:** Set up code in production
- **Operate:** Maintain the infrastructure
- **Monitor:** Watch code performance, errors...

- **Version control:** GitHub, GitLab
- **Containers:** Docker, Kubernetes
- **Monitoring:** Prometheus, Grafana, Sensu, Datadog, Splunk
- **Configuration management:** Chef, Puppet, *Ansible*, SaltStack, Helm
- **CI/CD:** Jenkins, Travis CI, GitLab
- **Tests:** Selenium, Dynatrace
- **Infrastructure:** *Terraform*, Vagrant, Packer,

- Information Technology Infrastructure Library v4 (ITIL v4)
  - **Change management governance**
  - Value chain
- Site Reliability Engineer (SRE) from Google
  - **Dedicated job** into a team
  - Software Engineer doing operational tasks
  - Error budget → Interruption allowed for project
- DevOps
  - **Multidisciplinary teams** (Ops and Dev in the same team)

All methodologies tend to be similar in some points

CHAPTER 2
PART 1: TERRAFORM

UniLaSalle
Amiens

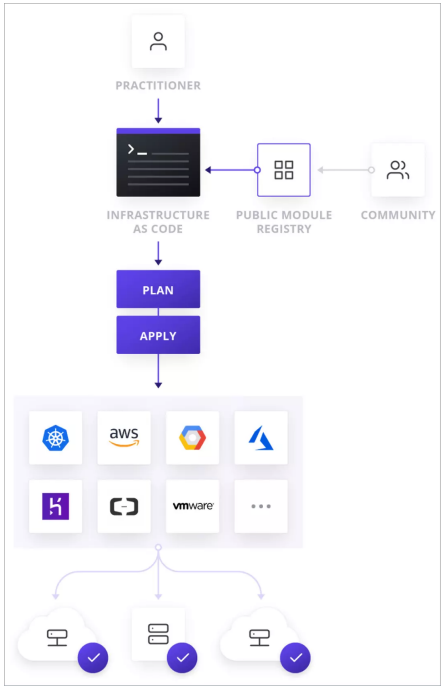
What is Terraform ?

- Open source solution developed by HashiCorp since 2014
- Not only system but also solutions
- Multi-vendor (providers): AWS, GCP, Azure, Alibabacloud, vRA, Cisco

How it works

- Resources are described in configuration files
- Main commands:
  - terraform **init**: Create required files and download providers, modules or backends
  - terraform **plan**: Create a plan of changes (do not alterate infrastructure)
  - terraform **apply**: Apply a plan and change infrastructure
  - terraform **destroy**: Destroy all resources
- Relies on a file containing the infrastructure state at all moment

How it works (Scheme)



"init" command

- Initialize backend
- Install required resources
  - Modules
  - Providers

Example

```
$ terraform init

Initializing the backend...

Initializing provider plugins...

Terraform has been successfully initialized!
```

"plan" command

- Plan the changes
- `terraform plan -out tf.plan` writes plan in a *tf.plan* file
  - Read the current state to make sure the terraform state is up-to-date
  - Construct dependencies between resources
  - Compare configurations
  - Propose change actions

Useful arguments

Setting	Command-line option
`-out`	Save plan to a file
`-refresh=false`	TF will not sync before compare
`-target=ADDRESS`	Tells Terraform to focus on specific resources and on any object they depend on
`-destroy`	Speculative destroy plan (useful with –target)

"apply" command

- Apply the changes
- `terraform apply tf.plan` apply a *tf.plan* file
  - Apply actions proposed in the tf.plan
  - Update the tfstate file
  - It is better to always apply a previously tf.plan file, otherwise terraform will make a plan of your whole configuration before applying it

Useful arguments

Setting	Command-line option
`-auto-approve`	Skips interactive approval of plan before applying (only if no plan file was given)
`-parallelism=n`	limits the number of concurrent operations, default is 10

Other commands

- `terraform fmt` to format your code
- `terraform taint` to force resource recreation on the next apply
- `terraform state` to manage state
- `terraform refresh` to refresh the state with remote resource actual configuration
- And more to see using `terraform help`...

Variables

- Each input variable must be declared using a block:
- Optional arguments:
  - *default:* a default value
  - *type:* the value type (string,number,bool, list, map, …)
  - *description:* to explain the purpose of this variable
  - *validation:* a block to define validation rule
  - *sensitive:* limit Terraform output

Example

```
variable "a_variable" {
  type = "string"
  description = "I am a variable containing a default value"
  default = "I have a default value"
}
```

Outputs

- Each output variable must be declared using a block:
- Optional arguments:
  - *description:* to explain the purpose of this variable
  - *sensitive:* to mark an output as containing sensitive information (limits Terraform's output)

Example

```
output "msg" {
  value = "I use a variable content: ${var.a_variable}"
  description = "Test of a sensitive variable"
  sensitive = true
}
```

Workspaces

- Each terraform code has an associated backend
  - Backend defines how operations are executed and where persistent data such as the Terraform state are stored
  - Persistent data stored in the backend belongs to a workspace
  - By default, backend has only one workspace, called “default” and is not removable
- Certain backends (like local, Amazon S3, Postgres, GCP bucket, …) (https://www.terraform.io/docs/language/state/workspaces.html) support multiple named workspaces
- It then allows multiple states to be associated with a single configuration

Example

```
resource "aws_instance" "example" {
  tags = {
    Name = "web - ${terraform.workspace}"
  }
  # ... other arguments
}
```

Providers

- Providers are Terraform "plugins" to interact with cloud providers
- You must declare which providers are required, and Terraform will install them
  - https://registry.terraform.io/browse/providers
- Each provider adds a list of dedicated resource(s) and data(s)

Example

```
terraform {
  required_providers {
    vra = {
      source = "vmware/vra"
      version = "~> 0.3.6"
    }
  }
}
```

Resources pt. 1

- Resources are the most important stuff to understand!!!
- Each resource block describes one or more infrastructure objects (virtual network, VM, …)
- Resource blocks can include lots of parameters but not all of them are mandatory
- Every Terraform provider has its own documentation:
  - Example of GCP: https://registry.terraform.io/providers/hashicorp/google/latest/docs
  - Example of VRA: https://registry.terraform.io/providers/vmware/vra/latest/docs
- Each resource block corresponds to an object in the infrastructure and has an identifier in the Terraform state

Resources pt. 2

- We declare a resource of a given type ("aws_instance") with a given resource key ("web")
- This resource key must be unique!
- Resource attributes :
  - Are used to access information `<RESOURCE TYPE>.<RESOURCE KEY>.<ATTRIBUTE>`
  - Help to configure and make implicit dependencies

Resources pt. 3

```
resource "aws_vpc" "this" {
  cidr_block           = local.address_space
  enable_dns_support   = true
  enable_dns_hostnames = true

  tags = merge({ Name = "VPC_${var.project}_${terraform.workspace}" }, local.global_tags)
}

resource "aws_vpc_dhcp_options" "dhcp_options" {
  domain_name          = var.route53_zone_domain
  domain_name_servers = ["AmazonProvidedDNS"]

  tags = merge({ Name = "dopt-${var.project}-${terraform.workspace}" }, local.global_tags)
}

resource "aws_vpc_dhcp_options_association" "dns_resolver" {
  vpc_id          = aws_vpc.this.id
  dhcp_options_id = aws_vpc_dhcp_options.dhcp_options.id
}

resource "aws_vpc_ipv4_cidr_block_association" "extend" {
  vpc_id     = aws_vpc.this.id
  cidr_block = local.public_address_space
}
```

Provider: Data

- DATA is a special type of resource used only for looking up information (READ-ONLY)
- DATA exports attributes which can be used as follow: data...

```
# Find the latest available AMI that is tagged with Component = web
data "aws_ami" "web" {
  filter {
    name   = "state"
    values = ["available"]
  }

  filter {
    name   = "tag:Component"
    values = ["web"]
  }

  most_recent = true
}
```

Locals

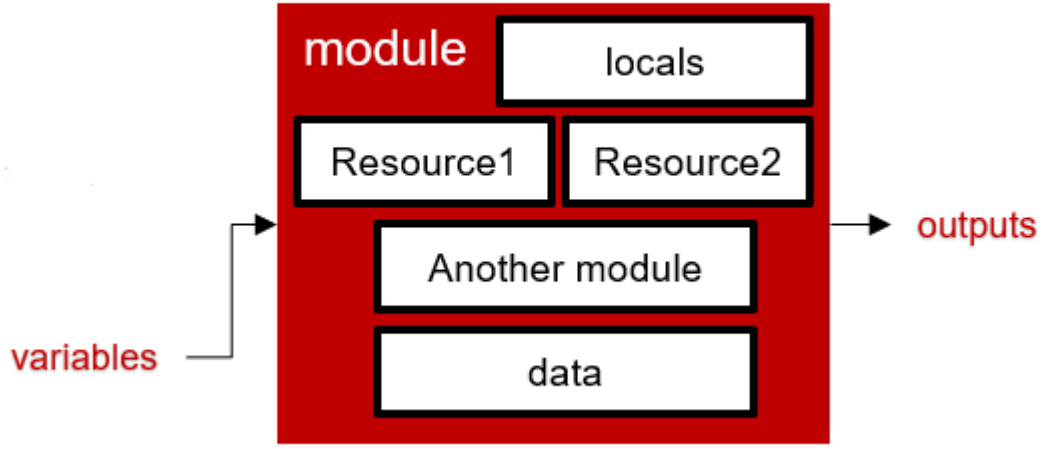
- Variables can’t be computed -> locals is the solution
  - https://www.terraform.io/docs/language/values/locals.html
- Note: Local values are created by a locals block (plural), but you reference them as attributes on an object named local (singular)

Backend

- By default, the state is written in a local file terraform.tfstate
- The state must be kept in a safe place:
  - Only authorized for authorized persons (secrets inside)
  - Prevents corruption
  - Keeps revision history
- Only certain backends support multiple workspaces
  - S3, Postgres, ...

Modules

- Modules are containers for multiple resources that are used together
- Modules are the main way to package and reuse resource configurations with Terraform
- Modules can be called multiple times within the same configuration
- Modules are downloaded and stored locally when Terraform initiates



How to use modules

- Using module blocks
  - Source argument tells Terraform where to find the module
  - Include contents of that module into the configuration with specific values for its input variables
- Module can declare output values to export certain values to be accessed outside of that module `module.<MODULE_NAME>.<OUTPUT_NAME>`

Example

```
module "servers" {
  source = "./my-source"
  servers = 5
}
```

State management

- `terraform import` allows you to import previously created resources and to add it in the terraform state
- `terraform state list` shows you all the resources present in the state
- `terraform state pull > {{nameyouwant}}` manually downloads the terraform state file
- `terraform state push {{nameyouwant}}` manually uploads a local terraform state file
- `terraform state mv {{SOURCE}} {{DESTINATION}}` used to continue tracking resources renamed or moved to a module
- `terraform state rm` removes the track of a resource in the terraform state without deleting it

State lock

- On each state READ or WRITE, the state is locked by terraform
- If you break terraform gracefully (Ctrl + C) on your local linux machine, it will unlock it
- If you break terraform hard (multiple Ctrl +C), close the shell, cancel gitlab pipeline, etc., the state will remain locked
- Unlocking could be done manually on cloud console or using terraform force-unlock LOCK_ID

CHAPTER 2
PART 2: ANSIBLE

UniLaSalle
Amiens

SUMMARY

UniLaSalle
Amiens

Definition

Ansible is an open-source software provisioning, configuration management, and application-deployment tool. It runs on many Unix-like systems, and can configure both Unix-like systems as well as Microsoft Windows. It includes its own declarative language to describe system configuration.

Introduction

- **Cross platfom support**:
  - **Agentless** : using Openssh or WinRm, no agent on host.
  - Support for Linux, Windows, UNIX, and network devices.
  - Physical, virtual, cloud, and container environments.
- **Human-readable automation**:
  - Simple.
  - Ansible Playbooks, written as YAML text files.
- **Perfect description of applications**:
  - Every change can be made by Ansible Playbook.
  - Every aspect of your application environment can be described and documented.
- **Easy to manage in version control**:
  - Ansible Playbooks and projects are plain text.
  - They can be treated like source code and placed in your existing version control system

Introduction

- **Support for dynamic inventories**:
  - List of machines that Ansible manages can be dynamically.
  - Updated from external sources.
  - Servers all the time, regardless of infrastructure or location.
- **Orchestration that integrates easily with other systems:**:
  - HP SA, Puppet, Jenkins, Red Hat Satellite, …
  - Other systems that exist in your environment.
- **DevOps oriented**:
  - Automation language that can be read and written across IT.
  - Can automate the application life cycle and continuous delivery pipeline from start to finish.

Introduction

- Two types of machines :
- Control nodes :
  - Ansible installation and execution.
  - Copies of ansible project files.
- Managed hosts:
  - Hosts to manage.

Introduction

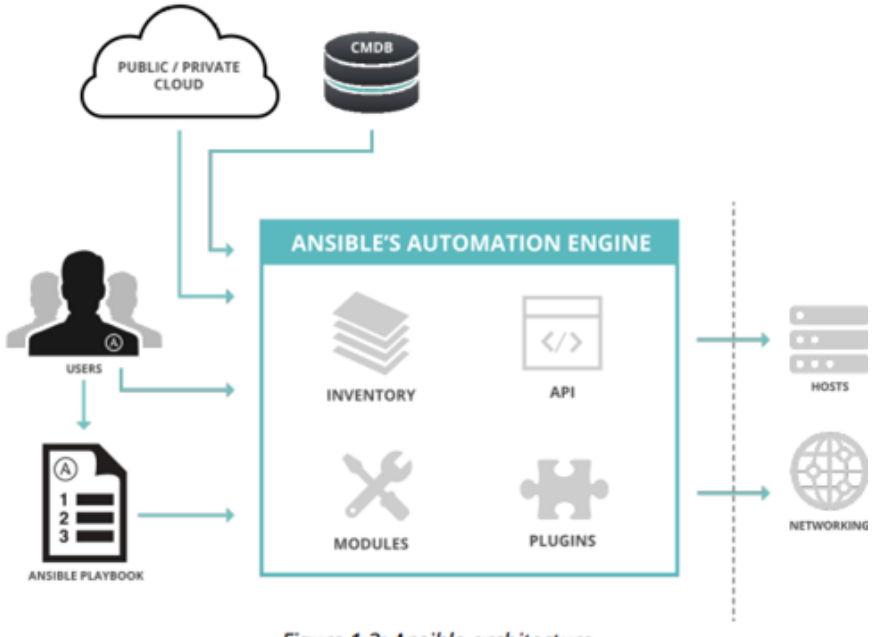


Figure 1.2: Ansible architecture

Concepts

- **Modules:**

  Modules (also referred to as “task plugins” or “library plugins”) 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 target node, and collects return values.

Concepts

- **Tasks:**
  - **Runs a module**: Module generally ensures that some particular thing about the machine is in a particular state: File exists,particular permissions , contents, mounted file system…

If the system is **not in that state**, the task should put it in that state.

If the system **is already in that state**, it should do nothing.

If a **task fails, Ansible's default behavior** is to abort the rest of the playbook.

- This property is called **IDEMPOTENCE**

Concepts

- **Play:** A play is a set of tasks that should be run in sequential order and on a given set of servers

```
- hosts: webservers #Target set of server
  tasks: #List of tasks
  - name: ensure apache is at the latest version
    yum: # Yum module
      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
    service:
      name: httpd
      state: started
```

- **Playbook:** A set of one or more plays

Concepts

- **Inventory:**
  - List of managed hosts
  - Organizes system into logical groups
  - Groups of groups
  - Variables (more on that later)
  - Can be either static or dynamic
- **Example** :

```
mail.example.com

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

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

Ansible philosophy

- **Complexity Kills Productivity**: **Simpler is better.**

  Tools should be simple to use.

  Automation is simple to write and read.

- **Optimize For Readability**:

  The Ansible automation language: easy for humans to read.

  Simple, declarative, text-based files.

  Ansible Playbooks can clearly document your workflow automation.

- **Think Declaratively**:

  Ansible: desired-state engine.

  Ansible's goal is to put your systems into the desired state.

  Only making changes that are necessary.

  Not scripting language.

Ansible philosophy

- **Combines and unites orchestration with configuration**:

  Management, provisioning, and application deployment in one easy-to-use platform.

- **Configuration Management**:

  Centralizing configuration file management and deployment.

- **Application Deployment**:

  Application definition with Ansible, deployment management with Ansible Tower.

  Manage the entire application life cycle from development to production.

- **Provisioning**:

  Help streamline the process of provisioning systems.

  Whether PXE booting and kickstarting bare-metal servers or virtual machines,

  Or creating virtual machines or cloud instances from templates.

Ansible philosophy

- **Continuous Delivery**: CI/CD pipeline requires coordination and buy-in from numerous teams.

  Ansible Playbooks keep your applications properly deployed (and managed).

- **Security and Compliance**: Security policy is defined in Ansible.

  Scanning and remediation can be integrated into other automated processes.

- **Orchestration**: Configurations alone don't define your environment.

  Define how multiple Configurations interact.

  Ensure the disparate pieces can be managed as a whole.

Installation

Control Nodes:

- Only needs to be installed on the control node.

Minimal requirements:

- The control node should be a Linux or UNIX system.
- Python 2 or 3 needs to be installed. Reference

Managed Hosts :

- Linux and UNIX managed hosts need to have Python 2 (version 2.4 or later).
- Ssh daemon configuration and reachable.
- Windows managed hosts: Powershell 3.0 and .NET 4.0 at least + WinRM listener activated

Configuration files

Configuration file (ordered) :

- $ANSIBLE_CONFIG
- $(pwd)/ansible.cfg (*Recommended practice*)
- ~/.ansible.cfg
- /etc/ansible/ansible.cfg

Find out what file is used :

```
$ ansible --version
ansible 2.13.5
  config file = /etc/ansible/ansible.cfg
```

Tip (Will highlight any non default value) :

```
$ ansible-config dump
ACTION_WARNINGS(default) = True
AGNOSTIC_BECOME_PROMPT(default) = True
ALLOW_WORLD_READABLE_TMPFILES(default) = False
...
```

Configuration options

ansible.cfg (ini format) :

```
[defaults]
inventory = ./inventory # The location of the Ansible inventory
remote_user = someuser # The user used for the CONNECTION
ask_pass = false # Does the CONNECTION require a password ?

[privilege_escalation]
become = true # Enable privilege escalation
become_method = sudo # Method used to escalate
become_user = root # Which user to escalate to
become_ask_pass = false #  Does the ESCALATION require a password ?
```

Reference :

```
$ ansible-config list
```

Building a STATIC inventory

Ini formatted text file :

```
[usa]
washington1.example.com
washington2.example.com
[canada]
ontario01.example.com
ontario02.example.com
[north-america:children]
canada
usa
```

Can be tested with :

```
$ ansible-inventory --graph
```

Building a STATIC inventory

Good to know :

- Two groups always exist :
  - **all** : Refers to all hosts **explicitly listed** in the inventory
  - **ungrouped** : Every host that is **NOT** a member of any other group, except **all**
- Ranges can be used in a python style [START:END:STEP(default=1)]
  - **192.168.[4:7].[0:255]** : All IPv4 addresses in the 192.168.4.0/22 network (192.168.4.0 through 192.168.7.255).
  - **server[01:20].example.com** : All hosts named server01.example.com through server20.example.com
  - **[a:c].dns.example.com** : Hosts named a.dns.example.com,b.dns.example.com, and c.dns.example.com.
  - **2001:db8::[a:f]** : All IPv6 addresses from 2001:db8::a through 2001:db8::f

DYNAMIC inventory

- **Dynamic inventory script** : Executable programs that collects information from some external source. Output the inventory in JSON format.
- **Contributed scripts** : Not included in the ansible package or officially supported by Red Hat. Ansible GitHub site at https://github.com/ansible-collections/community.general/tree/main/plugins/inventory.
- **Write your own dynamic inventory script ?** : See the Ansible Developer Guide: https://docs.ansible.com/ansible/latest/dev_guide/developing_inventory.html#developing-inventory

Running Ad Hoc commands :

- **Execute a single task (module)** :

```
ansible host-pattern -m module [-a 'module arguments'] [-i inventory]
```

- **Useful commands** :

```
ansible-doc -l #Lists all the modules that are installed on the system.
ansible-doc “name” #View the documentation of particular modules by name.
```

You can find all the modules on Ansible website :
https://docs.ansible.com/ansible/2.9/modules/modules_by_category.html

- **Shell vs Command**
  - *Command* : Allows administrators to quickly execute remote commands on managed hosts. **No access to shell environement variables.**
  - *Shell* : **Access to shell environment variables and shell operations.**

Why are Command and Shell modules evil ?

- Modules, in general, offer various benefits :
  - Idempotence
  - Check-mode
  - Diff output
  - Sanity checks
  - Return values

All of these properties are not available by default for the command and shell module. We will see how to work around that later

Command line options

You can use command line options to override configuration.

Setting	Command-line option
inventory	-i, --inventory
user	-u, --user
become	-b, --become
become_method	--become-method
become_user	--become-user
become_ask_pass	-K, --ask-become-pass

Full reference :

```
ansible --help
```

What is a playbook ?

Text file that **contains a list of one or more plays** to run in order written in YAML.

Adhoc vs playbook

- **Ad hoc** :

```
ansible -m user -a "name=newbie uid=4000 state=present" servera.lab.example.com
```

- **Playbook** :

```
---
- name: Configure important user
  hosts: servera.lab.example.com
  tasks:
    - name: newbie exists with UID 4000
      user:
        name: newbie
        uid: 4000
        state: present
```

Running playbook

- You can run playbook using ansible-playbook binary.
  - Executed on the control node.
  - The name of the playbook passed as an argument.

```
ansible-playbook yourplay.yml
```

- Ansible playbooks should be idempotent
  - You can run them safely multiple times
- Syntax verification with the --syntax-check option

```
ansible-playbook --syntax-check yourplay.yml
```

Execute a dry run

- Use the -C or --check option :

Report what changes would have occurred if the playbook were executed. No actual changes to managed hosts.

```
ansible-playbook -C yourplay.yml
```

- Monitor changes by using the --diff option :

Prints a diff of every file changed

```
ansible-playbook --diff yourplay.yml
```

- Combine the two to be sure of what will be delivered :

```
ansible-playbook --diff -C yourplay.yml
```

Implementing multiple plays

```
---
# This is a simple playbook with two plays

- name: first play
  hosts: web.example.com
  tasks:
  - name: first task
    yum:
      name: httpd
      state: present
  - name: second task
    service:
      name: httpd
      enabled: true

- name: second play
  hosts: database.example.com
  tasks:
  - name: first task
    yum:
      name: mariadb
      state: present
```

Remote Users and Privilege Escalation in Plays

- **User attribute** : User that is used to connect to hosts can be defined by the remote_user parameter.

```
remote_user: remoteuser
```

- **Privilege Escalation Attributes** :

```
- name: first play
  hosts: web.example.com
  become: true
  become_method: sudo
  become_user: privileged used
  tasks:
  - name: first task
    yum:
      name: httpd
      state: present
  - name: second task
    service:
      name: httpd
      enabled: true
```

A few notes on YAML (Comments and strings)

- Comments :

```
# This is a YAML comment
some data # This is also a YAML comment
```

- Strings :

```
string: this is a string
string2: 'this is another string'
string3: "this is yet another a string"
```

A few notes on YAML (Multilines)

- The | character (Newline characters within the string are to be preserved.)

```
include_newlines: |
  Example Company
  123 Main Street
  Atlanta, GA 30303
```

- The > character (Newline characters are converted into spaces)

```
fold_newlines: >
  This is
  a very long,
  long, long, long
  sentence.
```

A few notes on YAML (Arrays and dictionnaries)

- Arrays (Lists):

```
multiline_style_list:
  - servera
  - serverb
  - serverc

inline_style_list: [servera, serverb, serverc]
```

- Dictionnaries:

```
multiline_style_dict:
  name: svcrole
  svcservice: httpd
  svcport: 80

inline_syte_dict: {name: svcrole, svcservice: httpd, svcport: 80}
```

The multiline syntax is recommended

Variables

Variables can be useful for dynamic configuration, execution control etc...

- **Naming Variables** :

Variable names should be letters, numbers, and underscores. Variables should always start with a letter.

`foo_port` is a great variable. `foo5` is fine too.

`foo-port`, `foo port`, `foo.port` and `12` are not valid variable names.

- **Variables scope**:

A variable can be defined on three different scopes :

- Global
- Play
- Host

Defining variables in playbooks

- **Using vars directive** :

```
- hosts: all
  vars:
    user: joe
    home: /home/joe
```

- **Using vars_files directive** :

```
- hosts: all
  vars_files:
    - vars/users.yml
```

```
$ cat vars/users.yml

user: joe
home: /home/joe
```

- **Overriding Variables from the Command Line** :

```
ansible-playbook main.yml -e "package=apache"
```

Using variables

Use the double curly braces.

```
vars:
  user: joe

tasks:
  # This line will read: Creates the user joe
  - name: Creates the user {{ user }}
    user:
      # This line will create the user named Joe
      name: "{{ user }}"
```

When a variable is used as the first element to start a value, quotes are mandatory

```
name: "{{ user }}" # Valid
name: {{ user }} # Invalid
```

Using variables

With the following inventory content:

```
users:
  bjones:
    first_name: 'Bob'
  acook:
    home_dir: '/home/acook'
```

Syntax to access to variables:

```
# Returns 'Bob'
users.bjones.first_name
# Returns /users/acook
users.acook.home_dir
```

Or (python dictionnary)

```
# Returns 'Bob'
users['bjones']['first_name']
# Returns '/users/acook'
users['acook']['home_dir']
```

Host variables and group variables

This is the recommended way to set variables

- **Group variables**: Apply to all hosts in a group or its children groups
- **Host variables**: Apply to a single host

```
[datacenter1]
demo1.example.com
demo2.example.com
[datacenter2]
demo3.example.com
demo4.example.com
[datacenters:children]
datacenter1
datacenter2
```

```
$ cat ~/project/group_vars/datacenter1
package: httpd
$ cat ~/project/group_vars/datacenter2
package: apache
```

Recommended directory layout

```
inventories/
   production/
      hosts               # inventory file for production servers
      group_vars/
         group1.yml       # here we assign variables to particular groups
         group2.yml
      host_vars/
         hostname1.yml    # here we assign variables to particular systems
         hostname2.yml

   staging/
      hosts               # inventory file for staging environment
      group_vars/
         group1.yml       # here we assign variables to particular groups
         group2.yml
      host_vars/
         stagehost1.yml   # here we assign variables to particular systems
         stagehost2.yml
```

Registered variables

Allows to capture the output of a module

```
- name: Installs a package and prints the result
  hosts: all
  tasks:
    - name: Install the package
      yum:
        name: httpd
        state: installed
      register: install_result

    - debug: var=install_result
```

Debug module is used to dump the value of a given variable

Facts

Facts are values automatically retrieved by Ansible at the start of a play

To get a list of available facts for a system:

```
ansible demo1.example.com -m setup
```


```
{
    "ansible_all_ipv4_addresses": [
        "REDACTED IP ADDRESS"
    ],
    "ansible_all_ipv6_addresses": [
        "REDACTED IPV6 ADDRESS"
    ],
    "ansible_apparmor": {
        "status": "disabled"
    },
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "11/28/2013",
    "ansible_bios_version": "4.1.5",
    ...
}
```


Set `gather_facts: no` in a play to disable it

Custom facts

- Stored locally on each managed host
- `/etc/ansible/facts.d` directory
- **Example:**
  - File: `custom.fact` (INI or JSON format)

```
{
  "packages":{
    "web_package": "httpd",
    "db_package": "mariadb-server"
  },
  "users": {
    "user1": "joe",
    "user2": "jane"
  }
}
```

`ansible_local['custom']['users']['user1']` is `joe`

Magic variables

Set of variables automatically defined by Ansible

`group_names` : Lists all groups the current managed host is in

`groups` : Lists all groups in the inventory

`inventory_hostname` : Name of the current host as defined in the inventory (not as discovered by Ansible)

See all magic variables here

Includes

Useful for long and complex playbooks.

Allows to cut tasks and variables in smaller pieces for more readability

- **Include tasks**:

```
tasks:
- name: Include tasks to install the database server
  include_tasks: tasks/db_server.yml
```

- **Include variables**:

```
tasks:
- name: Include the variables from a YAML or JSON file
  include_vars: vars/variables.yml
```

Loops

- Sometimes you want to repeat a task multiple times. In computer programming, this is called a loop.
- **Simple loops** : Use the **loop** keyword

Before :

```
- name: Postfix is running
  service:
    name: postfix
    state: started
- name: Dovecot is running
  service:
    name: dovecot
    state: started
```

After :

```
- name: Services are running
  service:
    name: "{{ item }}"
    state: started
  loop:
    - postfix
    - dovecot
```

Loops over lists

Iterating over a variable :

```
vars:
  mail_services:
    - postfix
    - dovecot

tasks:
- name: Services are running
  service:
    name: "{{ item }}"
    state: started
  loop: "{{ mail_services }}"
```

Iterating over a list of dictionaries :

```
- name: Users exists
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - name: jane
      groups: wheel
    - name: joe
      groups: root
```

Nested loops

Iterating over a list of dictionaries :

```
tasks:
  - name: All DB users have privileges
    mysql_user:
      name: "{{ item[0] }}"
      priv: "{{ item[1] }}:ALL"
      append_privs: yes
      password: redhat
    loop: "{{ ['joe', 'jane'] | product(['clientdb', 'employeedb', 'providerdb']) | list }}"
```

Older syntax (still works):

```
with_nested:
  - ['joe','jane']
  - ['clientdb', 'employeedb','providerdb']
```

Other types of loops

Loop keyword	Description
`with_file`	Takes a list of control node file names. **item** is set to the **content** of each file in sequence.
`with_fileglob`	Takes a file name globbing pattern. **item** is set to each file in a directory on the control node that matches that pattern, in sequence, non-recursively
`with_sequence`	Generates a sequence of items in increasing numerical order. Can take start and end arguments which have a decimal, octal or hexadecimal integer value.
`with_random_choice`	Takes a list. **item** is set to one of the list items at random.

Refer to the Full reference for more

Conditionals

Sometimes you will want to skip a particular step on a particular host. That's what conditionals are for.

- Use the **when** statement :
  - When the expression after the `when` is **true**, the task is **RUN**
  - When the expression after the `when` is **false**, the task is **SKIPPED**
  - The expression may include operators like and, or, not. Just like in vanilla python

Examples :

```
when: item.mount == "/" and item.size_available > 300000000
```

```
when: ansible_distribution == "RedHat" or ansible_distribution == "Fedora"
```

Combining loops and conditionals

You can combine the `when` statement with a loop statement. In that case, the expression after the `when` will be **processed for each item** inside the loop.

Example :

```
- name: Install mariadb-server if enough space
  yum:
    name: mariadb-server
    state: latest
  loop: "{{ ansible_mounts }}"
  when: item.mount == "/" and item.size_available > 3000000000
```

CAREFUL : The expression in a `when` statement does not use curly braces for variables

Handlers

Often when you change a configuration, you may want to restart or reload the service to take the changes into account. This is what handlers are for

- **Definition** : A handler is a task that will only run when another task triggered it. It will be run at the end of the play that triggered it.
  - Use the `handlers` statement (play level) to declare your handlers
  - Use the `notify` statement to run the handler when the task reports `changed`

Example :

```
tasks:
  - name: copy demo.example.conf configuration template
    copy:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:
      - restart_apache
handlers:
  - name: restart_apache
    service:
      name: httpd
      state: restarted
```

Tags

Tags allow you to run or skip specific parts of a playbook by specifying it on the command line

```
---
- name: Example play using tags
  hosts:
    - servera
    - serverb
  tasks:
    - name: httpd is installed
      yum:
        name: httpd
        state: installed
      tags: webserver
    - name: postfix is installed
      yum:
        name: postfix
        state: latest
```

Tags

Once your tasks or plays are tagged, you can only run or skip them using :

```
ansible-playbook main.yml --tags webserver # Runs only tasks tagged : webserver
```

```
ansible-playbook main.yml --skip-tags webserver # Runs every tasks except those tagged : webserver
```

- There is one special tag that you can apply :

`always` : always run unless explicitly skipped by --skip-tags option

- There are system tags that are available by default :

`tagged` : Any tagged resource

`untagged` : Exclude all tagged ressource

`all` : Select all tasks (that is the default)

Handling errors and changed status

You can override the situations in which Ansible will report a `failed` or a `changed` for a task. To do so, you must use the `failed_when` and `changed_when` directive.

- `failed_when` example :

```
tasks:
  - shell: /usr/local/bin/create_users.sh
    register: command_result
    failed_when: "'Password missing' in command_result.stdout"
```

- `changed_when` example :

```
tasks:
  - shell: /usr/local/bin/upgrade-database
    register: command_result
    changed_when: "'Success' in command_result.stdout"
```

!!! When you use the evil `shell` and `command` modules, you should always make sure you use failed_when and changed_when to allow ansible to properly report the result !!!

Blocks

You can write blocks of tasks in your playbooks, which can have two main benefits :

- Avoid repeating `when` statements
- Better error management (see next slide)

```
- name: block example
  hosts: all
  tasks:
    - block:
      - name: package needed by yum
        yum:
          name: yum-plugin-versionlock
          state: present
      - name: lock version of tzdata
        lineinfile:
          dest: /etc/yum/pluginconf.d/versionlock.list
          line: tzdata-2016j-1
          state: present
      when: ansible_distribution == "Redhat"
```

Handling errors with blocks

Blocks enable you to handle errors kind of like how you would handle exception in code.

- `rescue` **statement** : The tasks in this block will be run if there is a failure in the block statement.
- `always` **statement** : The tasks in this block will be run after those in the block, regardless of failure or not.

```
tasks:
  - block:
      - name: upgrade the database
        shell: /usr/local/lib/upgrade-database
    rescue:
      - name: revert the database upgrade
        shell: /usr/local/lib/revert-database
    always:
      - name: always restart the database
        service:
          name: mariadb
          state: restarted
```

Overview

Jinja 2 is a templating framework in python and the templating engine of choice of Ansible.

Why do I need it ? : When you deliver a configuration file, you may want to make that configuration dynamic. Thanks to templates you can modify the content of files you deliver using variables.

Ansible allows :

- Referencing variables in playbooks with Jinja2
- Jinja2 loops and conditionals in templates
- Loops and conditionals are available in **tasks and playbooks**

```
- name: "[Servicetool] Pull corresponding image"
  docker_image:
    name: 'registry-production.svc.meshcore.net/lgs-platform/awl-c7-elasticsearch7{% if elasticsearch_r
    source: pull
```

Delimiters

- Variables or logic are place between tags
  - Expression or logic : `{% ... %}`
  - Variables : `{{ ... }}` (By now you should be familiar with this notation)
  - Comments : `{# ... #}`
- Example :

```
{# for statement #}
{% for myuser in users if not myuser == "Snoopy"%}
{{loop.index}} - {{ myuser }}
{% endfor %}
```

Jinja2 loops and conditionals

- Loop :

```
{% for user in users %}
    {{ user }}
{% endfor %}
```

- Conditionals :

```
{% if finished %}
  {{ result }}
{% endif %}
```

Jinja2 Filters

- Change output format to JSON or YAML for template expressions:

```
{{ output | to_json }}
{{ output | to_yaml }}
```

- Format expression output in human-readable format:

```
{{ output | to_nice_json }}
{{ output | to_nice_yaml }}
```

- Parse string provided in JSON or YAML format:

```
{{ output | from_json }}
{{ output | from_yaml }}
```

Also available in playbooks :

```
- debug: msg="the execution was aborted"
  when: returnvalue is failed
```

Build a Jinja2 Template

Jinja2 template is composed of two elements:

- Variables
- Expressions

You can therefore use variables or facts in templates.

Example template for motd :

```
Welcome to {{ ansible_hostname }}. Today's date is: {{ ansible_date_time.date }}.
```

Example template for a loadbalancer section

```
{% for myhost in groups['myhosts'] %}
{{ myhost }}
{% endfor %}
```

Actually use the template in playbook

Use the little cousin of the `copy` module, a.k.a `template` module :

```
tasks:
  - name: template render
    template:
      src: /tmp/j2-template.j2
      dest: /tmp/dest-config-file.conf
```

Problems we may encounter

- Datacenters include variety of host types:
  - Web servers
  - Database servers
  - Hosts containing software development tools
- Playbooks require tasks and handlers to manage these
  - Result: large and complex playbooks
- Roles can split playbooks into smaller playbooks and files

Role definition

Roles are ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

- Enable Ansible to load components from external files:
  - Tasks
  - Handlers
  - Variables
- Associate and reference:
  - Static files
  - Templates
- Files defining roles:
  - Given specific names
  - Organized in directory structure
- Roles written as general purpose can be reused

Benefits

- Roles promote easy sharing of content
- Roles can define essential elements of a system type:
  - Web server
  - Database server
  - Git repository
  - Other purposes
- Roles make larger projects more manageable
- Administrators can work on different project roles in parallel

Structure

- Role functionality defined by directory structure
  - Top-level directory: Defines role name
  - Subdirectories: Contain main.yml file
  - files and templates subdirectories: Contain objects referenced by main.yml files

What does it look like ?

```
.
├── defaults
│   └── main.yml
├── files
│   └── atos_ca.pem
├── molecule
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
│   └── requirements.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
│   ├── login.defs.j2
│   ├── nsswitch.conf.j2
│   ├── pam_sshd.j2
│   ├── pam_system-auth.j2
│   └── sssd.conf.j2
└── var
    └── main.yml
```

Subdirectories

Subdirectory	Function
`defaults`	Contains default values that are meant to be overridden
`files`	Static files
`handlers`	Handlers definitions
`meta`	Metadata about the role (author, license, dependencies)
`tasks`	Tasks files
`templates`	Jinja2 templates used by the role
`test (or molecule)`	Contains playbooks and other files to test the role
`vars`	Contains variables NOT meant to be overriden (mostly constants despite what the name suggests

Variables vs defaults

- To define role variables, create vars/main.yml with name/value pairs in hierarchy
  - YAML uses role variables like any other variable: {{ VAR_NAME }}
  - High priority
  - Cannot be overridden by inventory variables
- Use default variables to set default values for included or dependent role variables
  - To define default variables, create defaults/main.yml with name/value pairs in hierarchy
  - Lowest priority of any variables
  - Overridden by any other variable
- Best practice: Define variable in vars/main.yml or defaults/main.yml
- Use default variable when role needs value to be overridden

Use roles in playbook

A very complex syntax is required to use the roles in a playbook

```
---
- hosts: all
  roles:
    - role1
    - role2
```

OR when specifying variables

```
---
- hosts: all
  roles:
    - { role: role1 }
    - { role: role2, var1: val1, var2: val2 }
```

Dependencies

Sometimes roles may depend on other roles

Example: Role defining documentation server depends on role that installs and configures web server

Define roles in meta/main.yml in directory hierarchy:

```
---
dependencies:
  - { role: apache, port: 8080 }
  - { role: postgres, dbname: serverlist, admin_user: felix }
```

Order of Execution

- Default: Role tasks execute before tasks of playbooks in which they appear
- To override default, use pre_tasks and post_tasks
  - pre_tasks: Tasks performed before any roles applied
  - post_tasks: Tasks performed after all roles completed

```
---
hosts: remote.example.com
    pre_tasks: - shell: echo 'hello'
    roles:
      - role1
      - role2
    tasks: - shell: echo 'still busy'
    post_tasks:
      - shell: echo 'goodbye'
```

Role creation

- Simple to create roles in Linux
  - No special development tools required
- Three-step process:
  - Create role directory structure
  - Define role content
  - Use role in playbook.
- You can easily create a role with the directory structure with:

```
$> ansible-galaxy init <role-name>
```

How to load roles

- Ansible looks for roles in:
  - roles subdirectory
- Directories referenced by roles_path
  - Located in Ansible configuration file
  - Contains list of directories to search
- Each role has directory with specially named subdirectories

Content example

- tasks/main.yml file manages /etc/motd on managed hosts
  - Uses template to copy motd.j2 to managed host
  - Retrieves motd.j2 from role's templates subdirectory:

roles/motd/tasks/main.yml

```
---
- name: deliver motd file
  template:
    src: templates/motd.j2
    dest: /etc/motd
    owner: root
    mode: 0444
```

- References Ansible facts and system_owner variable:

roles/motd/templates/motd.j2

```
This is the system {{ ansible_hostname }}.

Today's date is: {{ ansible_date_time.date }}.

Only use this system with permission. You can ask {{ system_owner }} for access.
```

Calling role example

- Use motd with different value for system_owner
- someone@host.example.com replaces variable reference when role is applied to managed host:

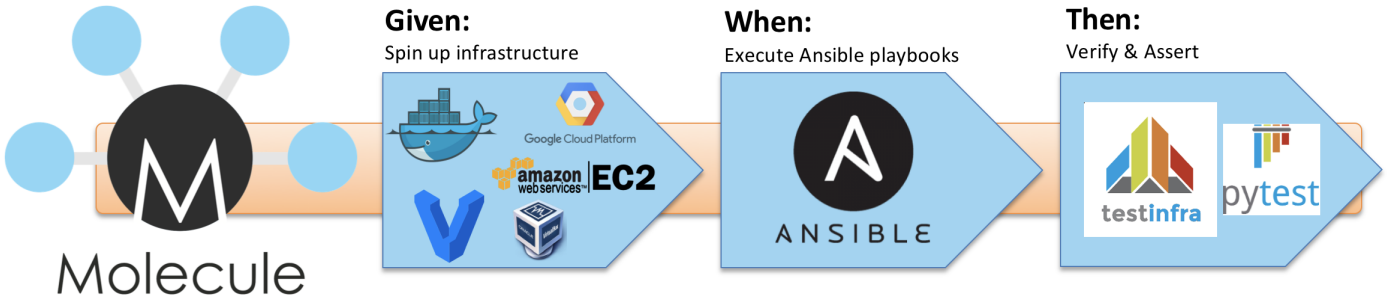
use-motd-role.yml

```
---
- name: use motd role playbook
  hosts: remote.example.com
  user: devops
  become: true
  roles:
    - { role: motd, system_owner: someone@host.example.com }
```

Testing roles

As roles are meant to be generic, they should be tested in various cases to ensure they work consistently.

The de-facto standard for testing Ansible roles is Molecule



SSH is critical to ansible

- Ansible uses SSH protocol to make remote connections to target nodes
- If pipelining not enabled, SSH connection used to:
  - Transfer modules and template files
  - Run remote commands
  - Run playbook plays on managed hosts
- Fast, stable, secure SSH connection critical to Ansible

Connection plugins

- These plugins are the most used by the community:

Setting	Description
paramiko	Python implementation of SSH protocol. Offers backward compatibility for RHEL6 and earlier. No support for OpenSSh ControlPersist
local	Runs commands locally
ssh	Uses OpenSSH-based connection< Supports ControlPersist
docker	Connects to container via `docker exec`
netconf	Provides a persistent connection using the netconf protocol (XML over SSH)
httpapi	Use httpapi to run command on network appliances (API over HTTP)
network_cli	Use network_cli to run command on network appliances (CLI over SSH)

Connection plugins

- More plugins are also used by the community:
  - chroot
  - libvirt_lxc
  - kubectl
- Not based on SSH
- Pluggable and extensible
- More types being added

Connection Type Configuration

ansible.cfg

To specify connection type in `[defaults]`, use transport:

```
[defaults]
# some basic default values...
...output omitted...
#transport = smart
```

- To override default transport value, use -c when:
  - Running ad hoc command using ansible
  - Running playbook using ansible-playbook
- *Best practice*: Leave connection type in ansible.cfg set to smart
  - Configure playbooks or inventory files to choose alternative connection setting

Connection Type Configuration when running playbooks

As we just saw, you can do that on the command line :

```
[student@demo ~]$ ansible-playbook playbook.yml --connection=local
```

OR directly in the playbook you want to run :

```
---
- name: Connection type in playbook
  hosts: 127.0.0.1
  connection: local
```

SSH Connection Configuration

Examples: *ansible.cfg*

```
[ssh_connection]
...output omitted...
control_path = %(directory)s/%%h-%%r
pipelining = False
scp_if_ssh = True
```

Environment settings

Use the **environment** statement :

- At the task level :

```
---
- hosts: devservers
  tasks:
    - name: download a file using demo.lab.example.com as proxy
      get_url:
        url: http://materials.example.copm/file.tar.gz
        dest: ~/Downloads
      environment:
        http_proxy: http://demo.lab.example.com:8080
```

- At the play level :

```
---
- hosts: demohost
  roles:
    - php
    - nginx
  environment:
    http_proxy: http://demo.lab.example.com:8080
```

Delegation

Some tasks must be delegated to different server from the one being managed

Examples:

- Sending notifications
- Waiting for server to be restarted
- Adding server to load balancer/monitoring server
- Making changes to DNS/networking configurations

Delegation helps run tasks to certain classes of hosts

- Example: Those outside current play

```
#task
- name: Running Local Process
  command: ps
  delegate_to: localhost
  register: local_process
```

Delegated facts

By default, any fact gathered by a delegated task are assigned to the inventory_hostname (the current host) instead of the host which actually produced the facts (the delegated to host).

```
- hosts: app_servers
  tasks:
    - name: gather facts from db servers
      setup:
      delegate_to: "{{item}}"
      delegate_facts: True
      loop: "{{groups['dbservers']}}"
```

Parallelism

- Ansible supports running tasks in parallel to all hosts
  - Provides more control over playbook execution
- Default: Ansible forks up to five times
  - Runs task on five different machines at once
- Default value set in `ansible.cfg`:

```
[student@demo ~]$ grep forks /etc/ansible/ansible.cfg
#forks = 5
```

- For more than five managed hosts, change forks to match environment

Parallelism (serial keyword)

- To reduce number of machines running in parallel, use serial
  - Sets lower fork number than global value in ansible.cfg
- Primary use case: Control rolling updates
- Example: Website is deployed on 100 web servers
  - Can define number or percentage

```
- name: test play
  hosts: webservers
  serial: 2
  gather_facts: False
  tasks:
  - name: task one
    command: hostname
  - name: task two
    command: hostname
```

Asynchronous

- Asynchronous Tasks
  - Some tasks take long time to complete
- Examples: Downloading large file, rebooting server
- Using parallelism with forks, Ansible:
  - Starts command quickly on managed hosts
  - Polls hosts for status until all are finished
- To run operation in parallel, use async and poll

```
---
- hosts: all
  remote_user: root

  tasks:

  - name: simulate long running op (15 sec), wait for up to 45 sec, poll every 5 sec
    command: /bin/sleep 15
    async: 45
    poll: 5
```

Asynchronous

- `async`: Triggers Ansible to run job in background and check later
  - Value indicates maximum time Ansible waits for command to complete
- `poll`: Sets how often Ansible checks if command has completed
  - Default value: 10 seconds

```
---
name: Long running task
hosts: demoservers
remote_user: devops
tasks:
  - name: Download big file
    get_url: url=http://demo.example.com/bigfile.tar.gz
    async: 3600
    poll: 10
```

Extremely long tasks

Option	Description
Use `wait_for` module	Blocking call, Ansible will wait for a system to be reachable
Set poll to 0	Non blocking call, but Ansible will not check completion or failure of the task
Set async to 0	Blocking call, Ansible waits as long as the job takes

Example : Rebooting a server

```
- name: restart machine
  shell: sleep 2 && shutdown -r now "Ansible updates triggered"
  async: 1
  poll: 0
  become: true
  ignore_errors: true

- name: waiting for server to come back
  wait_for:
    host: "{{ inventory_hostname }}"
    state: started
    delay: 30
    timeout: 300
```

Tip : Since Ansible 2.7, you can use the mighty `reboot` module

```
- name: Restart and wait until the server is rebooted
  hosts: demoservers
  tasks:
    - name: restart machine
      reboot:
```

Task status

To check task status, use async_status module and job id

```
- name: Download big file
  get_url: url=http://demo.example.com/bigfile.tar.gz
    async: 3600
    poll: 0
    register: download_sleeper

- name: Wait for download to finish
  async_status:
    jid: "{{ download_sleeper.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 30
```

10. Troubleshoot

Log files

- Not configured by default
- **log_path** parameter in the default section of the **ansible.cfg** configuration file.
- `ANSIBLE_LOG_PATH` environment variable

Example :

`ansible.cfg`

```
[defaults]
log_path = /home/student/troubleshooting/ansible.log
inventory = /home/student/troubleshooting/inventory
```

Configure **logrotate** to manage Ansible's log file .

Verbosity level

You can modify the output of `ansible` and `ansible-playbook` command

- Verbosity options :
  - **-v** : Output data
  - **-vv** : Output and input data
  - **-vvv** : Output and input data and connection informations
  - **-vvvv** : Extra verbosity options to the connection plug-ins, Users, scripts…

Debug module

Use the `debug` module to output certain variables to the user at runtime

Example:

```
- debug:
    msg: "The free memory for this system is {{ ansible_memfree_mb }}"

- debug:
    var: output
    verbosity: 2
```

The `verbosity` argument specifies that the debug task will be skipped unless `ansible-playbook` is run with superior or equal verbosity level

Managing errors

Several tools are at your disposal to troubleshoot your playbooks

- Syntax check

```
$ ansible-playbook play.yml --syntax-check
```

- Step by step execution

```
$ ansible-playbook play.yml --step
```

- Start at task

```
$ ansible-playbook play.yml --start-at-task="start httpd service"
```

Reminder : Execute a dry run

- Use the -C or --check option :

Report what changes would have occurred if the playbook were executed. No actual changes to managed hosts.

```
ansible-playbook -C yourplay.yml
```

- Monitor changes by using the --diff option :

Prints a diff of every file changed

```
ansible-playbook --diff yourplay.yml
```

- Combine the two to be sure of what will be delivered :

```
ansible-playbook --diff -C yourplay.yml
```

Check mode setting

On tasks you can use the check-mode setting to configure the task behavior while running in check-mode

There are two options:

- Force a task to run in check mode, even when the playbook is called without `--check`. This is called `check_mode: yes`.
- Force a task to run in normal mode and make changes to the system, even when the playbook is called with `--check`. This is called `check_mode: no`.

Integrate tests into your playbooks

- `uri` module:
  - Provides a way to check that a RESTful API is returning the required content.
- `script` module:
  - **Don't abuse it !!**
  - Supports the execution of a script on a managed host.
  - Failing if the return code for that script is non-zero.
- `stat` module:
  - Can check that files and directories not managed directly by Ansible are present.

QUESTION TIME

UniLaSalle
Amiens

PRACTICAL WORK

UniLaSalle
Amiens

- Use of Ansible
- Use of Vagrant to deploy some VMs to interact with
- Make a first use of Ansible
- Discover all seen concepts

- Use of Terraform
- Create some resource and interact with using Ansible
- Combine both tools to deploy a fully working application

Exam

EXAM INFORMATION

UniLaSalle
Amiens

Exam will be an MCQ with some questions being written

- DevOps *(25%)*
  - Definition (written)
  - Some questions about DevOps way of working
- Ansible *(40%)*
  - Principal paradigms
  - Main competitor
  - Command validity
  - Inventory
- Terraform *(35%)*
  - How it works
  - Resources/Modules
  - Differents main commands

- Deploy a fully working application (database, application and loadbalancer) using Terraform (with Docker) and Ansible
  - Only last lab will be graded (last 2 hours) - **75% of grade**
  - Write a report on what you have achieved with explanation on code/your choice - **25% of grade**

- In case of questions, you can reach me:
- By mail
  - alexis.leroux@worldline.com or alexis.leroux@ext.unilasalle.fr
- By SMS
  - +33677084962
- Don't ask for exam subject 😉

JOB TIME

UniLaSalle
Amiens

- Accelerate SysOps agility - Dev web portal
  - Portail web
  - Affichage du statut des sauvegardes dans le OnPrem via les API de CommVault
  - Information NFS pour cartographie.
- SysOps : Automation with a CI/CD pipeline on “Red Hat Entreprise Linux 9”
  - Stage d'automatisation avec pipeline CI/CD pour notre nouvel OS RHEL9 .
- SysOps : Automation with Ansible
  - Ansible Validation Platform
- Elastic stack Industrialization & SELinux Hardening
  - Automatiser notre capacité à déployer nos stacks Elastic.
  - Nous aider dans la sécurisation de nos plateformes avec SELinux
- Kubernetes Automation & SysOps
  - Openshift vs Kubernetes
- Katello tooling & Security report
  - Pipeline CI/CD pour valider notre capacité à faire les campagnes de patch
  - Générer les rapports de vulnérabilités.

- External interconnection as a managed service
  - Définir nos offres d'interconnexions operateur privées sur nos infrastructures Cloud
  - Automatiser le déploiement des configurations associées
- Cloud Security hardening
  - Définir nos architectures Cloud en lien avec les normes PCI et SecNumCloud
  - Automatiser le déploiement des configurations associées
  - Faciliter les audits
- Cloud Multisite Load Balancer
- NetOps & Cloud automation (plusieurs stages de ce type)
  - Développer les nouvelles fonctionnalités nécessaires à nos outils en utilisant Terraform, Ansible, pipeline gitlab CI/CD
- CCAP - Web security anomaly detection improvement
  - Détecter et reporter les anomalies dégradant le niveau de service ou de protection (santé des équipements, niveau de sécurité, cohérence de configuration ainsi que les attaques applicatives)

- Développement d'une pipeline gitlab qui devra être déclenchée automatiquement à la mise à jour de nos standards d'installation. Ce process devra produire une box (image) utilisable par tous les membres de l'équipe sur leurs pc. Ce qui permettra à chacun de pouvoir tester/vérifier les dernières modifications.

- DevOps for Kubernetes Infrastructure - GitOps
- DevOps for Kubernetes Infrastructure – Security and Compliancy