

앤서블

최국현

2022-10-11

랩 접속하기

앤서블 기초

앤서블 기초

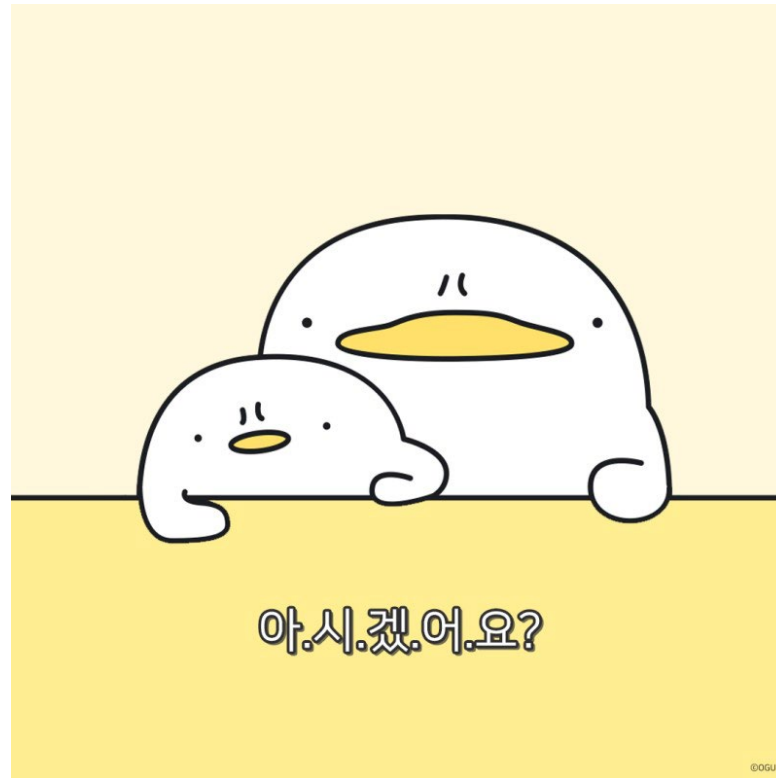
교육목적

이번 앤서블 교육은 다음 대상으로 제작 및 구성.

1. 앤서블 처음 사용하는 사용자
2. 실무까지는 아니어도 어떠한 방식으로 동작하는지 궁금하신 사용자
3. 간단하게 YAML 형태로 동작 학습
4. 전체적인 용어 및 기능 학습

앤서블 기초

교육목적



앤서블

앤서블 소개

앤서블은 본래 `Ansible Community, Company`에서 제작하였고 릴리즈 도중에 레드햇이 인수하였음.

앤서블은 총 두 가지 릴리즈를 유지하고 있음.

`Ansible Core`

`Ansible Engine`

앤서블

앤서블 코어

코어는 앤서블 앤서블 핵심 모듈로 구성되어 있으며, 그 이외 확장으로 `posix`, `collection`, `community`로 통해서 확장이 가능하다.

기본적으로 많이 사용하는 모듈(제공) `core module`, `posix`이다.
그 이외 나머지 기능들은 `collection`, `community`로 확장.

조금 혼돈 스럽기는 하지만 앤서블 코어를 **엔진(engine)**이라고 표현하기도 한다. **core**, **engine** 둘 다 같은 기능이다.

앤서블

앤서블 코어

또한 앤서블 코어는 두 가지 릴리즈 방식이 있다.

ansible-core(another name is ansible-base)

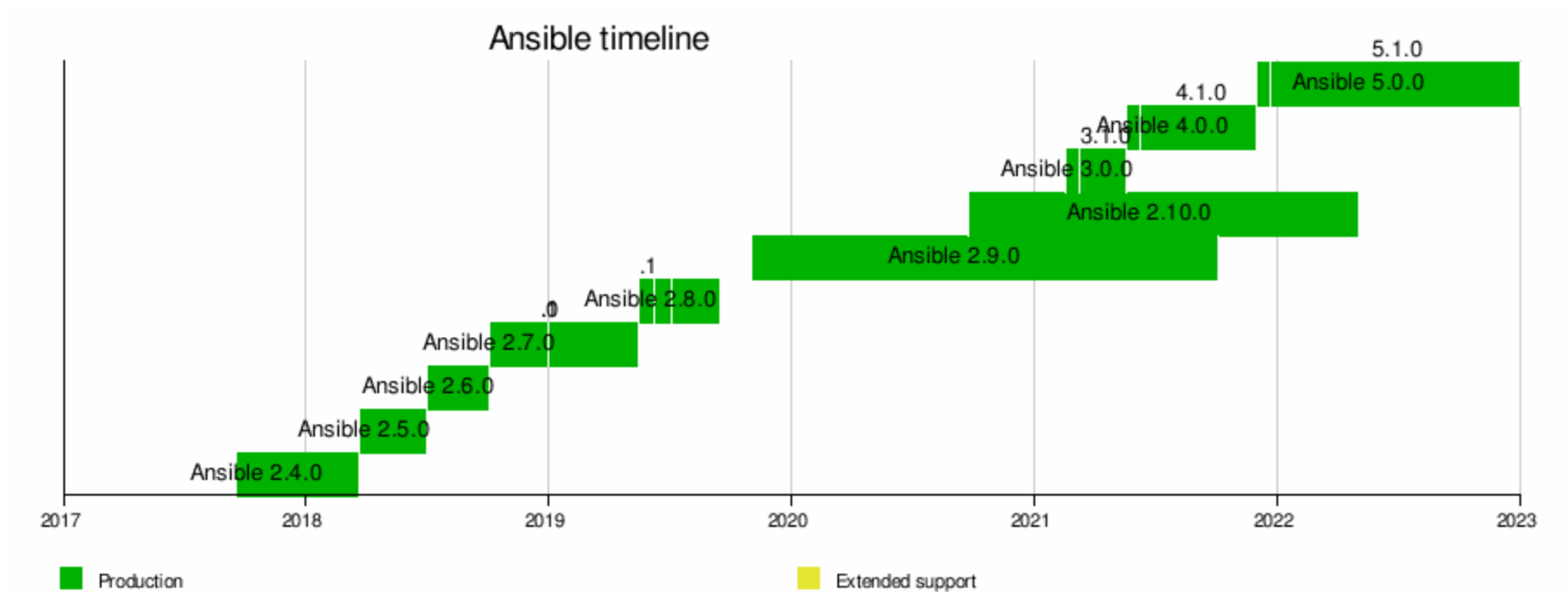
앤서블 코어는 앤서블 인터프리터 + 코어 모듈

ansible-project

앤서블 코어 + 추가적인 컬렉션 구성

앤서블

타임라인

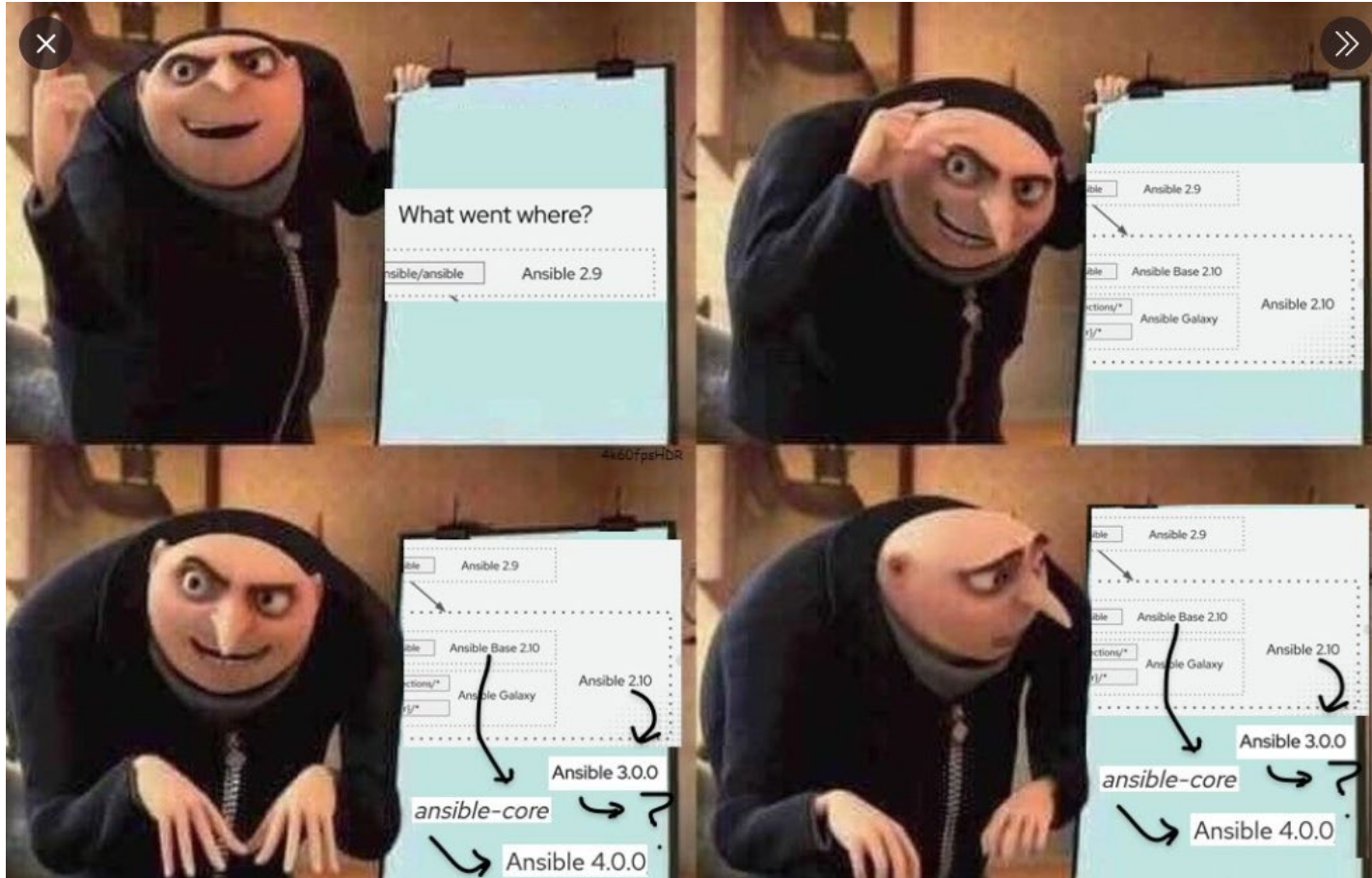


앤서블

릴리즈



앤서블



Andre Sencioles
@asenci

Is it just me struggling to keep up with Ansible changes? 😓

Red Hat Ansible @ansible · Feb 18

Ansible 3.0.0 has been released! This version of the #Ansible #community package marks the end of the restructuring of the Ansible ecosystem. To learn more about how and why we got to 3.0.0, take a look at "Announcing the Community Ansible 3.0.0 Package" [ansible.com/blog/announcing...](https://ansible.com/blog/announcing-the-community-ansible-3.0.0-package) pic.twitter.com/jdUiHlaKJR

[Show this thread](#)

5:48 PM · Feb 18, 2021 · Twitter Web App

9 Likes



앤서블

- ▶ The ansible==4.0.0 package on PyPI will depend on ansible-core>=2.11
- ▶ ansible==3.0.0 that was released today depends on ansible-base>=2.10.5,<2.11.
- ▶ ansible-core doesn't become 4.0.0, the next version will be 2.12.

앤서블

앤서블 소개

앤서블 타워

앤서블 타워는 두 가지 버전으로 구성이 되어있다.

첫 번째는 앤서블 **AWX** 업 스트리밍 버전이 있으며 레드햇에서 제공하는 서브스크립션 버전.

제일 큰 차이점은 버전 픽스가 되지 않는 롤링 업데이트 버전이다.

타워의 큰 특징은 웹 대시보드에서 API기반으로 앤서블 호출, 구성 및 관리가 가능하며 사용자 단위로 작업 할당 및 추적이 가능하다.

앤서블

앤서블 소개

앤서블 사용하기 전에 준비를 해야 될 부분은 다음과 같다.

YAML 작성 시 사용할 에디터. 아무거나 좋다! 정말로!

- YAML 문법
- `ansible.cfg` 및 `directory work frame`
- `ansible`, `ansible-playbook` 명령어 사용 방법

앤서블

editor

리눅스 콘솔에서 작성 시 사용하는 대표적인 에디터는 **vi/vim**에 디터가 있다.

거의 대다수 리눅스는 기본적으로 **vi**는 설치가 되어 있으나, **vim** 설치가 되어 있지 않는 경우가 있다.

이러한 이유로 **vim**설치 이외 몇 가지 기능을 추가적으로 구성 및 활성화 해야 한다.

사용하기 어려운 경우에는 **nano**를 사용하여도 된다.

앤서블

editor

다른 편집기를 원하는 경우, 다른 에디터를 사용해서 교육을 진행 하셔도 됩니다. 😊

앤서블

YAML

YAML은 HTML처럼 사양만 지정이 되어 있고, 여기에서 필요한 방법을 가져와서 자신이 사용하는 언어 기반으로 **포매팅/파싱/프로세싱** 하면 된다.

앤서블

<https://yaml.org/>

```
%YAML 1.2
---
YAML: YAML Ain't Markup Language™

What It Is:
  YAML is a human-friendly data serialization
  language for all programming languages.
|
YAML Resources:
  YAML Specifications:
    - YAML 1.2:
      - Revision 1.2.2      # Oct 1, 2021 *New*
      - Revision 1.2.1      # Oct 1, 2009
      - Revision 1.2.0      # Jul 21, 2009
    - YAML 1.1
    - YAML 1.0

  YAML Matrix Chat: '#chat:yaml.io'      # Our New Group Chat Room!
  YAML IRC Channel: libera.chat#yaml      # The old chat
  YAML News: twitter.com/yamlnews
  YAML Mailing List: yaml-core           # Obsolete, but historical

  YAML on GitHub:                                     # github.com/yaml/
    YAML Specs: yaml-spec/
    YAML 1.2 Grammar: yaml-grammar/
    YAML Test Suite: yaml-test-suite/
    YAML Issues: issues/
```

앤서블 준비

앤서블 설치

SSH

앤서블은 기본적으로 두 가지 접근 방식을 제공한다.

1. SSH 비공개/공개키 접근 방법
2. 사용자 아이디 및 비밀번호 접근

이 둘 중 하나를 사용하면 된다. 일반적으로 첫번째 방식을 더 많이 사용한다.

SSH

```
$ ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa
```

```
$ ssh-copy-id <ID>@<HOST>
```

SSH

- hosts: all

tasks:

- authorized_key:
 - user: "{{ ansible_user }}"
 - key: "{{ lookup('file', '/home/' + lookup('env', 'USER') + '{{ k8s_public_rsa_locate }}') }}"

vim editor

```
$ touch .vimrc
```

```
$ sudo yum install vim vim-enhanced neovim  
yamllint -y
```

혹은 centos 8버전 이후를 사용한다면

```
$ sudo yum install vim-ansible
```

vim editor

VIM

```
# mkdir -p ~/.vim/pack/git-plugins/start  
# git clone --depth 1 https://github.com/dense-analysis/ale.git  
~/.vim/pack/git-plugins/start/ale
```

NeoVim

```
# mkdir -p ~/.local/share/nvim/site/pack/git-plugins/start  
# git clone --depth 1 https://github.com/dense-analysis/ale.git  
~/.local/share/nvim/site/pack/git-plugins/start/ale
```


vim

```
$ vi ~/.vimrc
```

```
set ts=2
```

```
set sts=2
```

```
set sw=2
```

```
set expandtab
```

```
set number
```

```
let g:indentLine_char = '⋮'
```

```
syntax on
```

```
filetype indent plugin on
```

.vimrc

```
set ruler
```

```
set cursorcolumn
```

```
set foldlevelstart=20
```

```
let g:ale_echo_msg_format = '[%linter%] %s [%severity%]'
```

```
let g:ale_sign_error = '✖'
```

```
let g:ale_sign_warning = '⚠'
```

```
let g:ale_lint_on_text_changed = 'never'
```

nano

```
$ touch ~/.nanorc
```

```
$ sudo yum install nano
```

```
$ nano ~/.nanorc
```

nano

```
# Supports `YAML` files
```

```
syntax "YAML" "\.ya?ml$"
```

```
header "^(---|===)" "%YAML"
```

```
## Keys
```

```
color magenta "^\\s*[$A-Za-z0-9_-]+\\: "
```

```
color brightmagenta "^\\s*@[[$A-Za-z0-9_-]+\\: "
```

```
## Values
```

```
color white "\\s.+ $"
```

nano

Booleans

```
icolor brightcyan " (y|yes|n|no|true|false|on|off)$"
```

Numbers

```
color brightred " [[:digit:]]+(\. [[:digit:]]+)?"
```

Arrays

```
color red "\[" "\]" ":\s+[|>]" "^\\s*- "
```

Reserved

```
color green "(^| )!!(binary|bool|float|int|map|null|omap|seq|set|str) "
```

nano

Comments

color brightwhite "#.*\$"

Errors

color ,red ":\w.+\$"

color ,red ":' .+\$"

color ,red ":' .+\$"

color ,red "\s+\$"

nano

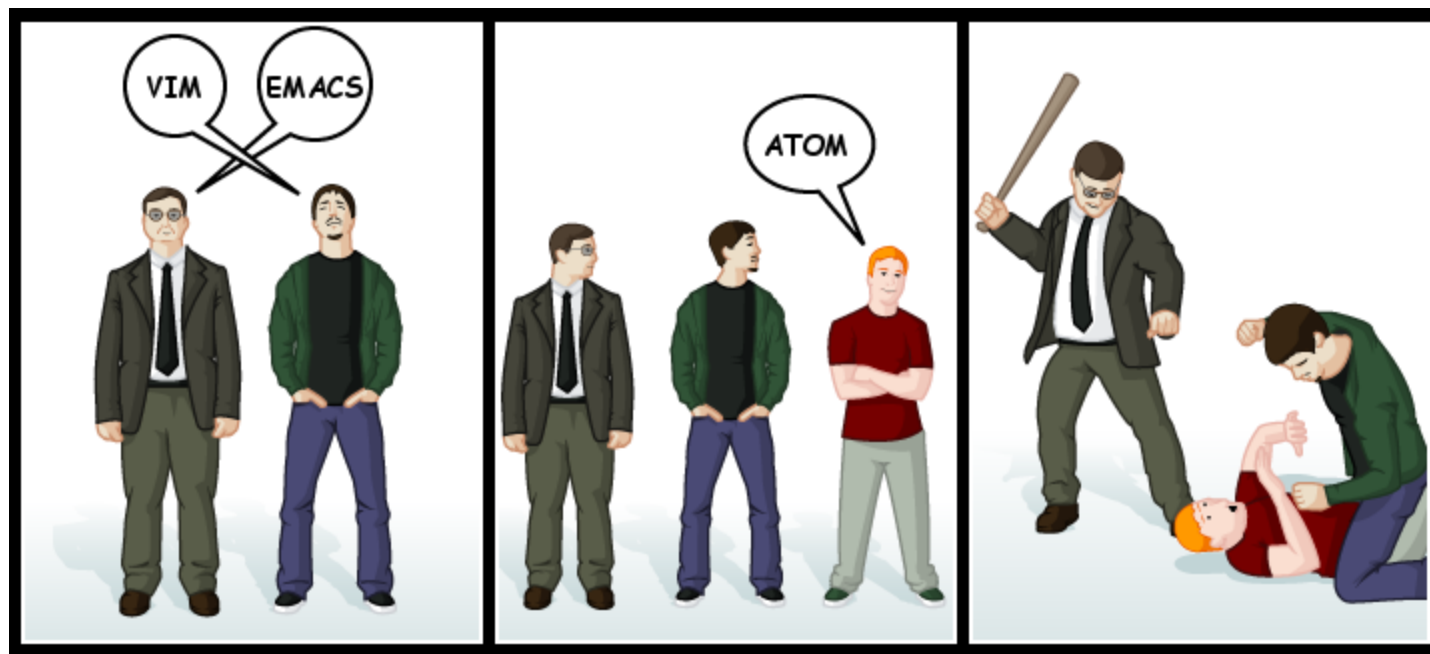
```
## Non closed quote  
color ,red "['\"]^['\"]]*$"
```

```
## Closed quotes  
color yellow "['\"].*['\"]"
```

```
## Equal sign  
color brightgreen ":( |$)"  
set tabsize 2  
set tabtospaces
```

앤서블 준비

Editor War



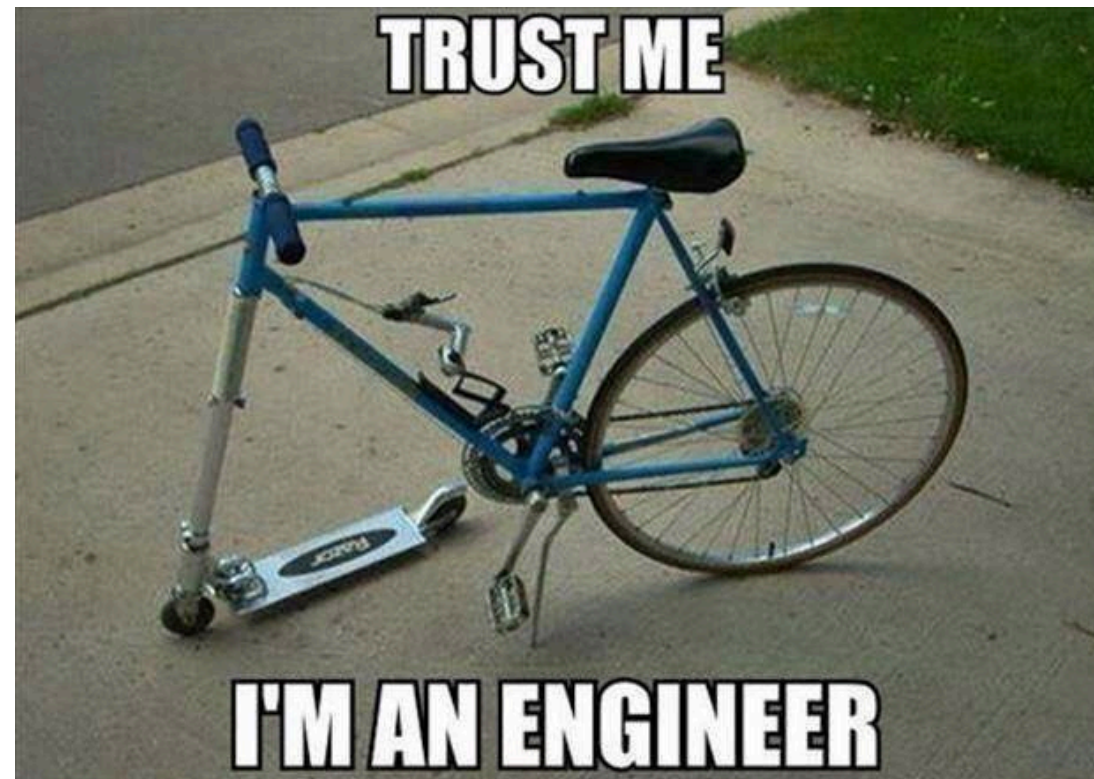
앤서블 준비

Editor War



앤서블 문법

YAML



YAML

앤서블에서 사용하는 문법을 작성하기 위해서는 다음과 같은 조건을 만족해야 한다.

최소 한 칸 이상의 띄어쓰기(권장은 2칸)

탭 사용시 반드시 빈 공간으로 전환

블록 구별은 -(대시)로 반드시 처리

YAML

```
-name: simple playbook
  hosts: all
  become: true

  tasks:
  module:
    args1:
    args2:
```

YAML

앤서블 블록 구별을 보통 "-"로 구별한다. 시작 블록은 보통 다음과 같은 형식으로 많이 사용한다.

- name:
 <keyword>
 <module>
- name:
 <keyword>
 <module>

YAML

그래서 YAML상단에는 다음과 같은 형태로 키워드 명령어를 사용한다.

- name: <작업이 수행 시 출력되는 이름>
 hosts: all ## 대상서버 이름.
 become: true ## 앤서블 내장 키워드

YAML

`hosts` 키워드는 다음과 같은 미리 예약된 옵션이 있다.

`localhost`: 127.0.0.1와 같은 자기 자신 루프 백(loopback)

`all` : 인벤토리에(`inventory`)등록된 모든 호스트

`[group]`: 특정 그룹에만 적용하는 명령어 키워드

`inventory`, `group` 이런 부분은 너무 깊게 들어가지 말기!

YAML

인벤토리 구조는
`sample-inventory` 파일 참고

전역 키워드 선언

예제

```
become:  
remote_user:
```

맨 상단에 있는 키워드 및 옵션은 보통 전역 키워드(global keyword)라고 생각하면 된다.

여기에서 적용된 옵션 및 명령어는 모든 플레이북에 적용이 되며 기존에 적용이 되어 있는 `ansible.cfg`의 내용을 오버라이드(override)가 된다.

tasks:

모든 작업이 시작되는 구간.

tasks 구간에는 여러 **모듈(module)**이 모여서 하나의 작업 워크플로우(**workflow**)를 구성한다.

여러 개의 워크 플로우가 구성이 되면 이것을 플레이북 혹은 플레이북 작업(**playbook tasking**)이라고 부른다.

- name

그래서 권장하는 방법은 작성시 각각 모듈에 **name:** 키워드를 사용하여 작성 및 구성을 권장한다.

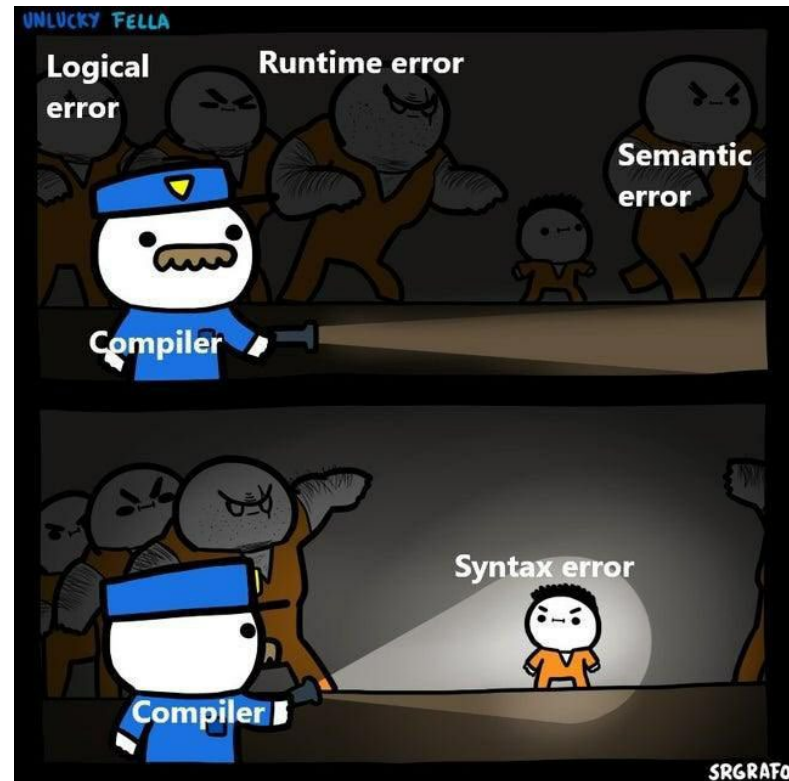
```
- name: this is the first module task  
  ping:
```

위와 같은 방법으로 명시한 모듈에 어떻게 사용할 것인지 명시한다.

YAML

`sample-yaml.yaml` 파일 참고

YAML



앤서블 모듈 및 패키지

MODULE

앤서블 모듈은 확장이 가능하며 확장 기능을 제공하는 프로그램.
현재 앤서블 모듈은 두 가지로 나누어서 기능 제공 하고 있음

ansible.core.*

ansible.posix.*

앤서블에서 제공하는 핵심 기능. 일반적인 핵심 기능은 POSIX 모듈에서 제공한다. 여기에 `copy`, `file`, `fetch`와 같은 자주 사용하는 기능이 포함되어 있다.

CORE MODULE

엔서블에서 제공하는 핵심 기능.

일반적인 핵심 기능은 CORE모듈에서 제공한다.

여기에 `copy`, `file`, `fetch`와 같은 자주 사용하는 기능이 포함되어 있다.

ansible.posix

앤서블에서 제공하는 핵심 기능.

많이 사용하는 리눅스 배포판에서 사용이 가능함.

이를 통해서 배포판 상관없이 공통적인 작업 수행이 가능하다.

ansible.collection

앤서블 확장 기능. 코어 기능에서 확장된 기능이며, 이걸 보통 **컬렉션(collection)**이라고 부른다. 컬렉션에는 벤더사에서 제공하는 모듈 기능도 포함이 되어 있다.

예를 들어서 대표적으로 많이 사용하는 **Network Manager, Linux System Role**도 컬렉션에 포함 되어 있다.

컬렉션은 'ansible-galaxy' 통해서 설치가 가능하다.

ansible.community

커뮤니티는 검증이 되지 않는 사용자 혹은 특정 그룹에서 만들어서 배포하는 **Role** 혹은 **Module**이다.

커뮤니티에 보면 보통 자신들이 만들어서 사용하는 플레이북 혹은 사용자화 된 **Roles**를 업로드 많이 한다.

커뮤니티도 컬렉션처럼 갤럭시를 통해서 설치가 가능하다.

Modules

`/lib/python3.8/site-packages/ansible_collections`

MODULE CASE

제일 많이 사용하는 모듈 **copy**기반으로 기능을 구현하면 다음과 같다.

- name: copy an issue file to remote server

copy:

src: /tmp/issue

dest: /etc/issue

MODULE DOCUMENT

모듈에 대한 자세한 옵션을 보기 위해서는 다음과 같은 명령어로 실행한다.

```
$ ansible-doc <MODULE NAME>
```

사용 가능한 모듈 목록을 확인하기 위해서는 아래 명령어로 목록 확인이 가능하다.

```
$ ansible-doc -l
```

ad-hoc

ad-hoc

앤서블은 YAML 형태 말고 ad-hoc방식이 있다.

이 방식은 마치 셸 스크립트 실행하는 방식과 비슷하게 **모듈+인자 형태**로 구성이 되어 있다.

아래는 간단한 ad-hoc사용 방식이다.

```
$ ansible <host>, -m <module> -a "arg1=<value> arg2=<value>:"
```

ad-hoc

애드훅은 웹 스크립트에서 같이 사용하거나 혹은 몇몇 웹 스크립트 기능을 표준화 모듈 기반으로 사용하기 위해서 사용한다.

자주 사용하지는 않지만 애드훅 기반으로 구성하는 경우 아래와 같은 방식으로 구성을 한다.

ad-hoc

```
$ vi adhoc_replace_shell_functions.sh
```

```
#!/bin/bash
```

```
if $(ansible localhost, -m ping)
then
```

```
    ansible all, -m copy -a "dest=/var/www/html/index.html, content='This is
World'"
```

```
else
```

```
    ansible all, -m copy -a "dest=/root/README.md, content='This file is
wrong'"
```

```
fi
```

ADHOC 연습문제

ad-hoc기반으로 다음과 같은 작업을 처리한다.

파일명은 `adhoc-exam.sh`으로 정한다.

1. "Hello SKT"라는 `default.html`파일을 `/var/www/html/`에 생성.
2. 웹 서버 패키지가 설치가 안되어 있으면 `yum`모듈을 사용해서 설치
3. 방화벽에 `http`, `https`서비스가 등록이 안되어 있으면 `http`, `https service`를 등록.
모듈은 `firewalld`를 사용.
4. 문제가 없으면 `uri`모듈을 통해서 웹 페이지 접근 및 접속이 잘 되는지 확인
5. 동작이 안되는 부분이 있으면 올바르게 동작하도록 수정

ADHOC 연습문제

`materials/ adhoc-exam.sh`

앤서블 인벤토리

INVENTORY

앤서블 인벤토리는 다음과 같은 형식을 가지고 있다.

[인벤토리 이름]

<호스트>

[인벤토리 이름:children]

<그룹이름>

INVENTORY

인벤토리는 위의 내용을 기준으로 다음과 같은 내용을 가지고 있다.

- 호스트 이름
- 아이피 주소
- 호스트에서 사용하는 변수

INVENTORY

[test]

node1.example.com package=httpd

[db]

10.10.5.2 package=mariadb

[was]

was1.example.com package=wildfly


인벤토리 변수

```
graph RL; A[인벤토리 변수] --> B[node1.example.com package=httpd]; A --> C[10.10.5.2 package=mariadb]; A --> D[was1.example.com package=wildfly];
```

INVENTORY

인벤토리 파일 기반으로 실행 시 다음과 같이 실행한다.

```
$ ansible-playbook -i hosts <PLAYBOOK>
```



인벤토리 옵션

ansible.cfg

인벤토리 파일은 일반적으로 **inventory**라는 이름으로 구성함. 다른 이름으로 변경을 원하는 경우 **ansible.cfg**에서 변경이 가능함.

혹은 **"-i"**으로 임의적으로 선택 가능.

Example of the ansible.cfg

```
$ cat ansible.cfg
[defaults]
inventory = <INVENTORY_FILE>
host_key_checking = False
remote_user = <SSH_USER>
ansible_ssh_port = <SSH_PORT>
ansible_ssh_user = <SSH_USER>
ansible_ssh_pass = <SSH_PASS>
forks = <PROCESSOR_FORK_NUMBER>
```

예제

위의 내용은 아래 파일 참고

```
materials/sample-inventory  
materials/sample-inventory-2  
materials/sample-ansible.cfg
```

연습문제

ansible.cfg 및 인벤토리를 다음처럼 구성한다.

1. **ansible.cfg**에는 다음과 같은 내용이 구성이 되어야 한다.
2. "**inventory_cloud**"라는 파일을 기본 인벤토리로 불러와야 한다.
3. **ssh** 접근 포트는 **8273**번을 사용한다.
4. 접근 시 사용하는 사용자는 **ansible**, 사용자의 비밀번호는 **ansible**이다.
5. **inventory_cloud**라는 설정 파일에는 다음과 같은 서버를 등록한다.

10.10.10.1

10.20.30.5

192.168.90.*

연습문제

```
# vi ansible.cfg
```

```
[defaults]  
inventory = inventory_cloud  
ansible_ssh_port = 8273  
ansible_ssh_user = ansible  
ansible_ssh_pass = ansible
```

```
# vi inventory_cloud  
10.10.10.1  
10.20.30.5  
192.168.90.*
```

연습문제

inventory_cloud에는 다음과 같이 설정 및 구성한다.

1. **web**, **db**라는 두개의 그룹이 존재하고 각각 그룹에 서버 하나씩 할당한다.
2. **web**에는 **web.lab.example.com**, **db**에는 **db.lab.example.com** 할당
3. 할당된 노드에는 **nodename**이라는 이름의 변수에 **web.example.com**, **db.example.com**라는 호스트 이름을 할당
4. **together**라는 그룹에 **web**, **db**가 동시에 사용이 가능하도록 한다

연습문제

```
# vi inventory_cloud
```

```
[web]
```

```
web.lab.example.com
```

```
[db]
```

```
db.lab.example.com
```

```
[together:children]
```

```
web
```

```
db
```

```
# ansible -i inventory_cloud --list-hosts
```

SSH KEY

키 배포

앤서블은 앞에서 잠깐 언급 하였지만 SSH를 사용해서 managed node에 접근.

배포를 하기 위해서는 다음과 같은 명령어를 통해서 생성 및 진행 해야한다.

키 배포

```
$ ssh-keygen -t rsa -N '<PASSPHASE>' -f  
'<SSH_KEY_DIRECTORY>'
```

```
$ ssh-copy-id <USER>@<HOST>
```

키 배포

혹은 **authorized_keys** 기능을 사용해서 ssh 공개키 배포가 가능하다.

키 배포

- hosts: all

tasks:

- authorized_key:

user: "{{ ansible_user }}"

key: "{{ lookup('file', '/home/' + lookup('env',
'USER') + 'k8s_public_rsa_locate') }}"

형식(파일)

위치

앤서블 테스트

pre/main/post

앤서블 태스크는 기본적으로 **tasks** 기본으로 동작한다.

하지만, 여러가지 이유로 전/후 작업 처리가 필요한 경우 **pre_tasks, task, post_tasks** 총 3개의 작업으로 나누어진다.

기본적으로 많이 사용하는 작업 방법은 다음과 같다.

pre/main/post

materials/example-tasks-1.yaml

pre/main/post

pre_tasks:

- name: updates all of packages

yum:

name: "*"

state: latest

본 작업이 발생하기 전에 작업한다. **pre_tasks** 보통 **tasks**(혹은 **role**)과 수행이 되며 단독으로 사용하지 않는다.

pre/main/post

tasks:

- name: install a vsftpd package

yum:

name: vsftpd

state: latest

보통 **pre_tasks** 이후에 본 작업이 수행이 되며, 단독 실행이 가능하다.

pre/main/post

post_tasks:

- name: start and enable the vsftpd service

service:

name: vsftpd

state: started

enabled: yes

task작업이 수행이 된 다음에 후 처리 작업을 보통 "post_tasks"에서 수행한다.

역시 "post_tasks"만 단독으로 사용하는 경우는 거의 없다.

pre/main/post

pre_tasks:

- name: updates all of packages

yum:

name: *

state: latested

첫번째 작업 수행

tasks:

- name: install a vsftpd package

yum:

name: vsftpd

state: latested

주 작업 수행

post_tasks:

- name: start and enable the vsftpd service

service:

name: vsftpd

state: started

enabled: yes

후 처리 작업 수행

pre/main/post

다중 작업은 `roles`과 함께 사용이 가능하다.

"tasks:"보다는 **"roles:"**키워드와 함께 더 많이 사용한다.
이 부분에 대해서는 뒤에서 더 다루도록 한다.

`pre_tasks:`

`roles:`

- `{ role: vsftpd }` 혹은 `vsftpd`

`post_tasks:`

`tasks`가 `role`로 변경.
한 개 이상의 `role` 조합으로 사용이 가능하다.

pre/main/post

tasks-roles/tasks-roles.yaml

pre/main/post

```
- hosts: web
  pre_tasks:
    - name: this is the hello message from ansible
      debug:
        msg: "This message shows from pre tasks"
```

roles:

```
- { role: vsftpd }
```

tasks가 role로 변경.
한 개 이상의 role 조합으로 사용이 가능하다.

post_tasks:

```
- name: this is the by message from ansible
  debug:
    msg: "This message shows from post tasks"
```


앙서블 변수

VAR

앤서블도 다른 프로그램 언어처럼 변수를 사용한다. 다만, 우리가 알고 있는 변수와는 다른 방식으로 사용한다.

앤서블 변수는 다음과 위치에서 선언이 가능하다.

VAR

- `inventory`
- `group_vars`
- `host_vars`
- `role vars`

VAR IN INVENTORY

Inventory var는 다음처럼 보통 선언한다.

```
[server]
```

```
test.lab.example.com var1=hello var2=world
```

var1은 변수 이름이며, "=" 변수 대입 연산자 "hello"는 영문 소문자로 변수값 할당.

VAR IN INVENTORY

앤서블에서 사용하는 모든 변수는 **public type**으로 구성되어 있다.

별도의 **private** 변수는 없으며 **ansible runtime** 동작이 완료되면 변수는 그 즉시 **heap/stack** 메모리에서 제거가 된다.

하지만, 앤서블은 파이썬 메모리 구조에 영향 받는다.

GROUP VARS

앤서블 변수에서 제일 많이 사용하는 변수는 **group_vars**, **host_vars** 두 개의 변수 디렉터리 프레임 워크가 있다.

이들은 다음과 형태로 구성이 가능하다.

GROUP VARS

두 가지 형태로 구현 할 수 있다.

1. 파일

2. 디렉터리 + 파일

보통은 2번 "디렉터리 + 파일" 형태를 많이 사용한다.

GROUP VARS

group_vars는 다음처럼 구성하고 선언한다.

```
$ mkdir group_vars
```

```
$ mkdir server    ## 이 부분은 파일형태로도 구성이 가능하다.
```

```
$ cd server
```

```
$ vim hostname
```

```
node1: node1.example.com
```


HOST VARS

두 가지 형태로 구현 할 수 있다.

1. 파일

2. 디렉터리 + 파일

보통은 2번 "디렉터리 + 파일" 형태를 많이 사용한다.

HOST VARS

host_vars는 다음처럼 구성하고 선언한다.

```
$ mkdir host_vars
```

```
$ mkdir db1.example.com ## 이 부분도 파일형태로도 구성이 가능
```

```
$ cd db1.example.com
```

```
$ vim db1.example.com
```

```
db1: db1.example.com
```

예제

`materials/group_host_vars/tasks-roles.yaml`

VAR IN ROLE

`materials/var_roles/hostname.yaml`

VAR IN ROLE

role에서 변수 작성 및 구성은 다음과 형태로 한다.

1. `roles/<ROLE NAME>/vars`
2. `roles/<ROLE NAME>/defaults`
3. `main.yaml` 생성
4. 변수값 선언

role에서 `/vars` 혹은 `/defaults` 밑에 반드시 최소 하나의 `main.yaml`이 존재해야 한다.

VAR IN ROLE

```
$ mkdir roles/hostname/vars
```

```
$ vim roles/hostname/vars/main.yaml  
node1: node1.lab.example.com
```

ROLE with VAR

플레이북에서 `role`를 호출하면서 일시적으로 기존 변수 값을 `override`가 필요하다. 그 경우 아래처럼 사용이 가능하다.

```
$ vim var_roles/hostname.yaml
```

```
- hosts: localhost
```

```
roles:
```

```
- { role: hostname, node1: node1-1.lab.example.com }
```

연습문제

VAR IN ROLE

materials/practice-exam-variable

연습문제

앞에서 학습한 내용을 가지고 간단하게 앤서블 기반으로 플레이북 만들어 본다.

1. ansible adhoc 기반으로 host1에서 host2로 파일을 보낸다.

copy모듈로 hello.txt라는 파일을 생성하여 hosts2의 /tmp/hello.txt에 저장한다.

메시지 내용은 "Hello World"

연습문제

앞에서 학습한 내용을 가지고 간단하게 앤서블 기반으로 플레이북 만들어 본다.

2. ansible inventory 파일 생성한 후 host2에 파일을 보낸다.

inventory에는 호스트 그룹은 web, db로 구성한다.

web에는 control.example.com db는 manage.example.com

copy.yaml를 만들어서 /tmp/issue파일을 원격 서버의 /etc/issue에 보낸다.

/tmp/issue파일에는 "Hello SKT ansible remote"라는 메시지가 포함이 되어 있어야 한다.

플레이북 1

PLAYBOOK

앞에서 배운 기본적인 지식을 통해서 간단한 플레이북을 작성해본다.

현재 우리가 가지고 있는 서버는 총 2대의 서버를 가지고 있다.

- 1번 제어서버
- 2번 관리서버

PLAYBOOK

각각 서버를 인벤토리에 등록 후 텍스트 파일을 생성 및 전달을 한다.

이를 구성하기 위해서 다음과 같이 디렉터리를 생성한다.

PLAYBOOK

```
$ mkdir playbook1
```

```
$ cd playbook1
```

```
$ touch inventory
```

```
$ touch ansible.cfg
```

```
$ mkdir roles
```

PLAYBOOK

```
$ tree -L 2 .
```

```
.  
├── ansible.cfg  
├── inventory  
└── roles
```

```
1 directory, 2 files
```


PLAYBOOK

```
$ vim inventory
```

```
[node1]
```

```
192.168.90.11 → control.example.com
```

```
[node2]
```

```
192.168.90.12 -> manage.example.com
```

PLAYBOOK

```
$ echo "Hello SKT World" > welcome.html
```

```
$ vim copy.yaml
```

```
- hosts: node1
```

```
  tasks:
```

```
    - name: install a httpd package on {{ inventory_hostname }}
```

```
      yum:
```

```
        name: httpd
```

```
        state: latest
```

PLAYBOOK

실행!

PLAYBOOK

올바르게 동작하면 추가로 다음과 같이 기능을 추가한다.

- name: copy the welcome.html to {{ inventory_hostname }}

copy:

content: "Hello SKT world"

dest: /var/www/html/welcome.html

PLAYBOOK

실행!

PLAYBOOK

```
- name: install a httpd package on {{ inventory  
_hostname }}
```

```
  yum:
```

```
    name: httpd
```

```
    state: latest
```

PLAYBOOK

올바르게 동작하면 추가로 다음과 같이 기능을 추가한다.

- name: enable and start the service

- service:**

- name: httpd

- state: started

- enabled: yes

PLAYBOOK

실행!

PLAYBOOK

- name: copy the welcome.html to {{ inventory_hostname }}
copy:
 - src: welcome.html
 - dest: /var/www/html/welcome.html
- name: enable and start the service
service:
 - name: httpd
 - state: started
 - enabled: yes

플레이북 2

PLAYBOOK 2

지금까지 학습한 내용을 가지고 간단하게 플레이북 생성.

플레이북 2에서는 인벤토리 변수를 사용해서 기능을 더 추가해보도록 하겠다.

PLAYBOOK 2

현재 가상머신은 플레이북 1를 통해서 패키지 및 `welcome.html`파일은 배포 하였지만, 호스트 이름을 아직 수정하지 않았다.

호스트 이름을 변수에 다음처럼 선언한다.

INVENTORY

인벤토리를 다음처럼 수정을 한다.

[node1]

192.168.90.11 **nodename=node1.example.com**

[node2]

192.168.90.12 **nodename=node2.example.com**

PLAYBOOK

그리고 추가적으로 플레이북 파일을 하나 더 생성한다.

```
$ touch hostname.yaml
```

```
$ vim hostname.yaml
```

PLAYBOOK

- name: set up hostname to {{ inventory_hostname }}
- hostname:
- name: "{{ nodename }}"

IMPORT_TASKS

태스크 확장하기 위해서는 다음과 같은 명령어를 사용한다.

```
$ vim main.yaml
```

```
- name: setup hostname
```

```
  import_tasks: hostname.yaml
```


IMPORT_TASKS

```
- name: copy the welcome.html to {{ inventory_hostname }}  
  copy:  
    src: welcome.html  
    dest: /var/www/html/welcome.html  
  
- name: setup hostname  
  import_tasks: nodename.yaml
```

IMPORT_TASKS

실행이 되지 않는다.
이유를 찾아보자.

추가 기능 설명

import/include

"import_" 문법은 "tasks"이외 "role", "playbook"와 같은 추가적인 기능들이 더 있다.

"include"문과 많이 다른 부분은 "include"는 동적으로 동작한다. "import"문법은 정적으로 동작한다.

include_playbook은 deprecated가 되었다.

<https://www.youtube.com/watch?v=7EwCsUFibsU>

include_tasks

"include_" 문법은 위의 "import_"와 동일하다.

다만, `include_`는 동적으로 파일을 불러오며 앤서블 인터프리터가 파싱 하면서 `include`문법을 만나면 그때 파일을 메모리에 적재한다.

특수변수

앤서블에는 특수 변수가 존재한다. 이것을 매직변수라고 부르기도 한다. 우리는 위의 예제에서 다음과 같은 예제를 사용하였다.

1. `inventory_hostname`
2. `ansible_hostname`
3. `ansible_username`
4. `ansible_ssh_user`(deprecated)
5. `ansible_ssh_port`(deprecated)

특수변수

! Note

Ansible 2.0 has deprecated the “ssh” from `ansible_ssh_user`, `ansible_ssh_host`, and `ansible_ssh_port` to become `ansible_user`, `ansible_host`, and `ansible_port`. If you are using a version of Ansible prior to 2.0, you should continue using the older style variables (`ansible_ssh_*`). These shorter variables are ignored, without warning, in older versions of Ansible.

확장변수

이전에 사용한 인벤토리에 다음과 같은 변수를 선언 하였다.

`nodename`

따로 변수 선언문은 없지만, 앤서블에서 이것을 파싱 (parsing)하면 자동적으로 인벤토리의 추가 변수로 인식한다.

변수

GROUP_VAR

example_group_vars/

일반변수

GROUP_VARS

그룹이름으로 변수를 제공 합니다. 해당 그룹에 포함이 되어 있는 서버들은 "group_vars"에 선언된 변수를 전달 받습니다.

HOST_VARS

호스트 이름, 즉 인벤토리에 할당되어 있는 호스트의 이름 혹은 아이피 주소로 변수를 전달한다.

GROUP_VAR

구성은 다음과 같이 한다.

```
$ mkdir group_vars
```

```
$ vim inventory
```

```
[node1]
```

```
192.168.90.13 nodename=node1.example.com
```

```
$ mkdir -p group_vars/node1
```

GROUP_VAR

```
$ vim all
```

```
httpd_package: httpd
```

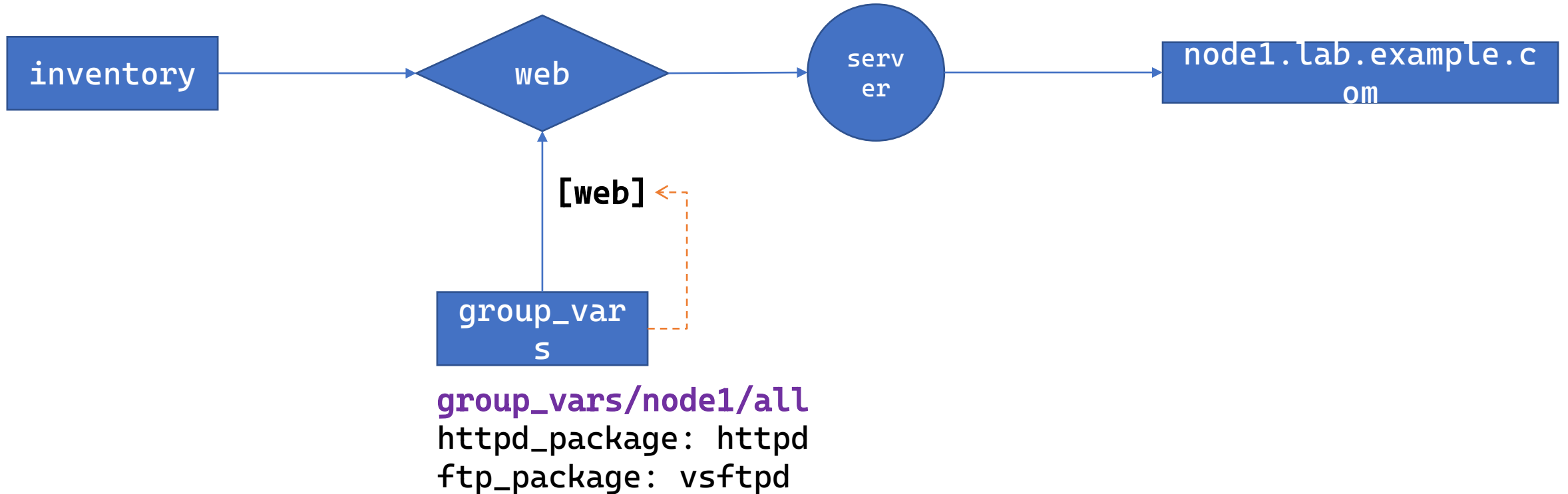
```
ftp_package: vsftpd
```

인벤토리 node1에 할당이 되어 있는 서버는 192.168.90.13
이기 때문에 192.168.90.13서버에는 위의 변수를 할당 받
는다.

GROUP_VAR

동작 방식은 다음과 같다.

GROUP_VAR



HOST_VAR

`example_host_vars/`

HOST_VARS

구성은 다음과 같이 한다.

```
$ mkdir host_vars
```

```
$ vim inventory
```

```
[node1]
```

```
192.168.90.13 nodename=node1.example.com
```

```
$ mkdir -p host_vars/192.168.90.13
```

HOST_VAR

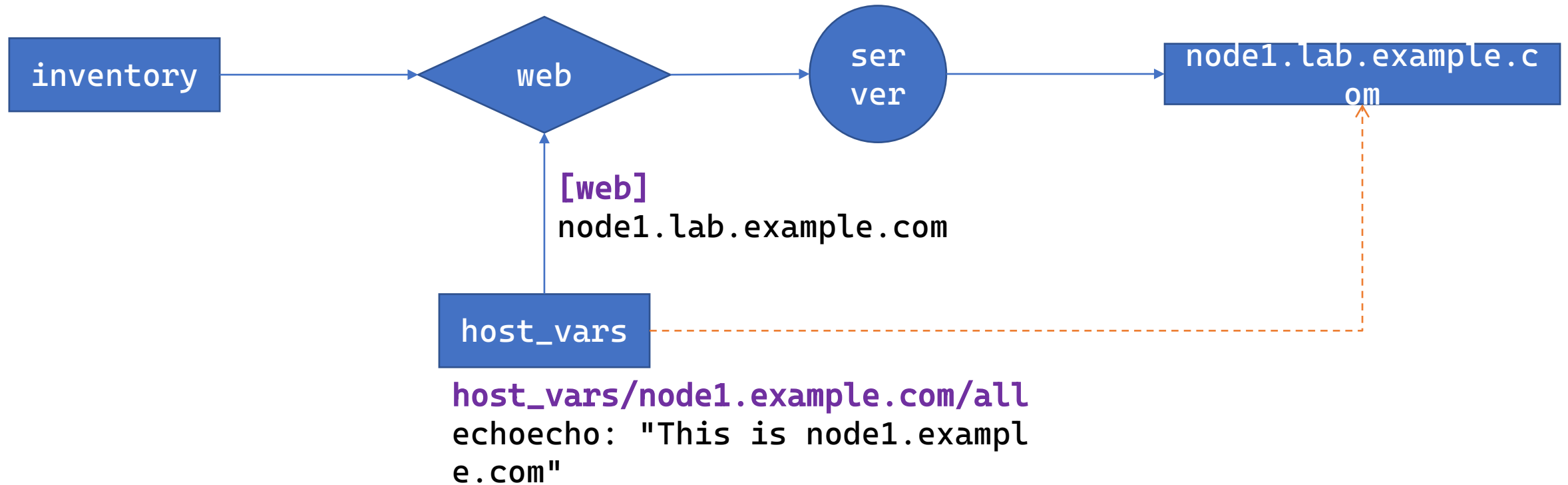
```
$ vim all
```

```
echoecho: "This is node1.example.com"
```

192.168.90.13에 서버만 echoecho라는 변수에 "This is node1.example.com"이라는 문자열 할당한다.

다른 서버에는 echoecho라는 변수가 할당되지 않는다.

HOST_VAR



예제파일

example_vars/
example_list/
example_dict/

일반변수

일반 변수는 보통 플레이북이 다음처럼 작성 및 구성한다.

```
- name: this is first vars
```

```
  hosts: localhost
```

```
  vars:
```

```
    division: was
```

```
tasks:
```

```
- name: Hello SKT
```

```
  command: echo "Hello SKT"
```

```
- debug:
```

```
  msg: "{{ division }}"
```

리스트

리스트는 변수 앞에 "-"대시가 붙는다. 또한 리스트는 맨 앞에 사용할 리스트의 이름이 붙는다.

```
- name: this is first list
```

```
  hosts: localhost
```

```
  vars:
```

```
    region:
```

```
      - northeast
```

```
      - southeast
```

```
      - midwest
```

```
  tasks:
```

```
    - name: echo the the first element value
```

```
      command: echo "{{ region[0] }}"
```

딕셔너리

딕셔너리는 다음처럼 구성한다. 딕셔너리는 이름 및 값으로 구성이 되어 있으며, 이름/값 구분은 **:(콜론)**으로 되어 있다.

```
- name: this is first dict
```

```
  hosts: localhost
```

```
  vars:
```

```
    dict:
```

```
      firstname: choi
```

```
      lastname: gookhyun
```

```
      gender: male
```

```
  tasks:
```

```
- name: shows your name
```

```
  command: echo "{{ dict.firstname }}" "{{ dict.lastname }}" "{{ dict['gender'] }}"
```

앤서블 시스템 변수

예제파일

`system_keyword/`

hosts

hosts는 **앤서블 키워드**이며, 싱글 플레이북 혹은 멀티 플레이북에서 하나 혹은 여러 개 사용이 가능하다.

hosts에는 하나 이상의 값 할당이 가능하다.

host

- name: this is the first playbook
hosts: control.example.com
- name: this is the second playbook
hosts: manage.example.com

remote_user

- name: this is the first playbook
hosts: control.example.com
remote_user: root
become: true

ansible_connection

보통 인벤토리에 선언하는 시스템 변수(키워드)이다.

인벤토리에 선언된 이름과 실제로 접속이 되는 서버 이름이 다른 경우 아래와 같이 선언한다. **hostname, IP Address** 상관 없다.

inventory

```
192.168.90.11 ansible_connection=local
```

ansible_user

인벤토리 서버들에 접근 시 보통 사용하는 사용자는 **remote_user**를 통해서 접근한다.

만약 다른 사용자로 접근을 해야 되는 경우 아래와 같이 선언을 한다.

inventory

```
192.168.90.11 ansible_connection=localhost ansible_user=testuser
```

ansible_port

앤서블 리모트 서버에 접근 시 사용하는 포트 번호가 다른 경우 다음처럼 임시로 포트번호 변경이 가능하다.

inventory

```
192.168.90.11 ansible_port=8080
```

ansible_ssh_pass

앤서블 서버에 접근 시, `ssh` 공개키가 아닌 비밀번호로 접근해야 되는 경우 아래와 같은 옵션으로 접근이 가능하다.

inventory

```
192.168.90.11 ansible_ssh_pass=centos
```


ansible_*

이러한 앤서블 매직 변수는 다음 소스코드에서 확인이 가능하다.

https://github.com/ansible/ansible/blob/c600ab81ee/lib/ansible/playbook/play_context.py#L46-L55

매직변수

매직 변수 중에서 제일 많이 사용하는 변수는 바로..

아이피 주소 관련된 변수

ansible_facts

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

```
{{ ansible_facts.eth0.ipv4.address }}
```

ansible_facts

앞에서 이야기 하였지만, 여기에서 사용하는 시스템 **리스트 (list)** 혹은 **딕(dict)**는 전부 **setup** 모듈을 통해서 전달 받는다.

대신 facts를 수행하는 경우 수집하는 시간이 있기 때문에 시간이 좀 실제 작업이 되기전까지 딜레이가 발생한다.

앤서블 시스템 변수

ansible_facts

변수를 사용하기전에 테스트 및 필요한 도구.

```
$ ansible localhost, -m setup
```

```
# yum install python3-jmespath
```

ansible_facts

tasks:

- name: Shows NIC interface name

debug:

msg: `"{{ ansible_facts.default_ipv4.interface }}"`

- name: Shows NIC IP Address

debug:

msg: `"{{ ansible_facts.default_ipv4.address }}"`

- name: Shows Block device size

debug:

msg: `"{{ ansible_facts.mounts | json_query('[?mount == `/\`.size_available') }}"`

ansible_ssh_{{ user, password }}

SSH Fingerprint Issue

이를 해결하기 위해서 `ansible.cfg`에 다음처럼 추가한다.

```
[defaults]  
host_key_checking = False
```

위의 설정을 넣어주면 `Fingerprint` 부분은 무시하고 진행한다.

ansible_ssh_{{ user, password }}

SSH Fingerprint Issue

```
{"msg": "Using a SSH password instead of a key is not possible because Host Key checking is enabled and sshpass does not support this. Please add this host's fingerprint to your known_hosts file to manage this host."}
```


ansible_ssh_{{ user, password }}

만약, `ssh` 아이디 및 비밀번호를 가지고 로그인에 필요한 경우, 다음처럼 구성이 가능하다.

```
ansible_user = root
```

```
ansible_pass = centos
```

```
remote_user = root
```

ansible_ssh_{{ user, password }}

https://github.com/ansible/ansible/blob/c600ab81ee/lib/ansible/playbook/play_context.py#L46-L55

ansible_ssh_{{ user, password }}

ansible_ssh_* 키워드는 사용은 가능하나, 다음 키워드로 대체하여 사용하는 것을 권장한다.

ansible_ssh_user → ansible_user

ansible_ssh_host → ansible_host

ansible_ssh_port → ansible_port

EXTRA-VARS

`sample_extra_vars/`

EXTRA-VARS

확장변수는 `--extra-vars`라는 옵션이 있다. 이 옵션을 통해서 추가적으로 변수를 구성하거나 혹은 변수 값 변경이 가능하다.

`vars:`

`config_dir: "/etc/httpd/conf.d/"`

`package_name: httpd`

EXTRA-VARS

```
$ ansible-playbook -e config_dir="/etc/vsftpd/" -  
e package_name=vsftpd main.ayml
```

vars:

config_dir: **"/etc/vsftpd"**

package_name: **vsftpd**

간단한 변수우선 순위

1. command line values (for example, `-u my_user`, these are not variables)
2. role defaults (defined in `role/defaults/main.yml`)
3. inventory file or script group vars
4. inventory `group_vars/all`

변수우선 순위

- 5. inventory group_vars/*
- 6. playbook group_vars/*
- 7. inventory file or script host vars
- 8. inventory host_vars/*
- 9. playbook host_vars/*

변수우선 순위

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#list-variables

조건문/레지스터/Facts

조건문

when

앤서블에서 **when**이라는 조건문을 사용한다.

"**when**"은 **module**, **role/roles**에서 함께 많이 사용한다.

그 이외 조건에서 사용이 가능하면 언제든지 붙여서 사용이 가능하다.

다만 몇몇 조건에서 **when**조건이 적용이 안될 수 있다.

when

```
vars_file:
```

```
  - vars.yaml
```

```
tasks:
```

```
  copy:
```

```
    src: httpd.conf
```

```
    dest: /etc/httpd/conf/httpd.conf
```

```
  when: httpd_conf_update
```

조건문

when

vars_file:

- vars.yaml

외부 변수파일을 불러옴

```
$ vi vars.yaml  
httpd_conf_update: yes
```

httpd_conf_updat의 변수가 "yes"로 되어 있으면 참으로 판단하여 수행이 됨.

tasks:

copy:

src: httpd.conf

dest: /etc/httpd/conf/httpd.conf

when: httpd_conf_update == "yes"

when

when조건에는 `ansible_facts`와 함께 사용이 가능하다.

먼저, `ansible_facts`에 대해서 잠깐 설명하자면, 앤서블에서 `setup`모듈이 동작하면서 원격 서버들의 정보를 수집하는데 이를 `facts`라고 부른다.

when

`ansible_facts`를 그냥 네이티브로 사용하기도 하는데 보통은 `when`을 통해서 조건 비교를 많이 한다.

```
when: (ansible_facts['os_family'] == "Debian") or  
(ansible_facts['os_family'] == "CentOS")
```


when

```
$ vi whenos.yaml
```

```
- hosts: localhost
```

```
  tasks:
```

```
    - debug:
```

```
      msg: "{{ ansible_facts['os_family'] }}"
```

```
      when: ansible_facts['os_family'] == "Debian"
or "CentOS"
```

when

위에서 잠깐 확인 하였지만, 조건문을 사용할 때 **or**, **and**같은 확장 조건문 사용이 가능하다.

and: `true + true = true`

or: `true + false = true`

"`os_family`"가 `Debian`이나 혹은 `CentOS`라는 문자열을 가지고 있으면 참이다.

when multi-condition

여러 개의 조건에 대해서 확인이 필요한 경우는 다음처럼 조건문을 다중으로 사용이 가능하다.

- name: shutdown system
command: poweroff

when:

- ansible_facts['distribution'] == "CentOS"
- ansible_facts['distribution'] == "8"

when multi-condition

```
$ vi multiwhenos.yaml

- hosts: localhost

  tasks:

    - debug:
        msg: "Shutdown OS"

      when:
        - ansible_facts['distribution'] == "CentOS"
        - ansible_distribution_version == "8"
        - ansible_distribution_release == "Stream"

    - debug:
        msg: "No Shutdown OS"

      when:
        - ansible_facts['distribution'] != "CentOS"
        - ansible_distribution_version != "8"
        - ansible_distribution_release != "Stream"
```

<, lt, <=, le, >, gt, >=, ge, ==, =, eq, !=, <>, ne

조건문에는 숨겨진 기능들이 꽤 많다.

정말로 꽤 많다.

작성하다 보니 왜 이것까지 작성하는지 모르겠다.



이런! 어쩌다 이렇게 됐지?

<, lt, <=, le, >, gt, >=, ge, ==, =, eq, !=, <>, ne

```
{{ ansible_facts['distribution_version'] is  
version('12.04', '>=') }}
```

```
{{ your_version_var is version('1.0',  
operator='lt', strict=True) }}
```

여기에 앤서블 2.11에서는 버전타입이 추가가 되었다.

버전

loose

loose 버전은 버전 레이블링 형식이 우리가 아는 상식(?)내에서 자유롭게 보통 표현을 한다.

1996.07.12

3.2.pl0

3.1.1.6

2g6

버전

strict

보통 3자리로 구별이 된다. 일반적으로 많이 사용하는 릴리즈 방식

V1.1

V1.2

V3

V2.4

V2.4.0

버전

semver/semantic

Semantic Version의 약자. **Major/Minor/Patch**형태로 구성하고 숫자 앞에 0이 붙으면 안된다.

<, lt, <=, le, >, gt, >=, ge, ==, =, eq, !=, <>, ne

```
{{ your_version_var is version('4.1.9-rc.1+build.100',  
operator='lt', version_type='semver') }}
```

Theory tests

issubset \rightarrow subset

issuperset \rightarrow superset

Theory tests

vars:

a: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

b: [2, 3, 4, 5]

superset/subset

A영역에 B가 포함이 되어 있는지(혹은 B가 A영역에 포함이 되어 있는지) 검증 및 확인한다.

tasks:

- debug:

 - msg: "A include B"

 - when: a is superset(b)

- debug:

 - msg: "B include in A"

 - when: b is subset(a)

superset/subset

A영역에 B가 포함이 되어 있는지(혹은 B가 A영역에 포함이 되어 있는지) 검증 및 확인한다.

tasks:

- debug:

 - msg: "A include B"

 - when: a is superset(b)

- debug:

 - msg: "B include in A"

 - when: b is subset(a)

레지스터

register

레지스터는 앤서블 모듈에서 실행된 결과를 메모리에 저장하는 기능이다.

레지스터 명령어는 `register`, 사용법은 다음과 같다.

- <MODULE>

<ARGS>:

<ARGS>:

`register: <REGISTER_NAME>`

register

레지스터는 모듈 실행 후 출력된 표준 출력 및 오류 내용을 메모리에 저장해서 사용자가 원하는 내용을 출력해서 확인이 가능하다.

다음 모듈을 실행을 한다.

register

```
$ vi register1.yaml
- name: list content of directory
  command: ls /tmp
  register: contents

- name: check contents for emptiness
  debug:
    msg: "Directory is empty"
  when: contents.stdout == ""
```

register

또한 이와 같은 방식으로 사용이 가능하다.

```
$ vi register2.yaml
```

- name: register a variable
command: /bin/false
register: result
ignore_errors: true
- name: run when the "result" var fails
command: /bin/somedothing
when: result is failed

register

또한 이와 같은 방식으로 사용이 가능하다.

- name: run when the "result" var succeeds
command: /bin/somedothing_else
when: result is succeeded
- name: run when the "result" var skipped
command: /bin/somedothing_else
when: result is skipped

register

레지스터 내용을 출력하고 싶은 경우 다음처럼 debug모듈을 사용하면 된다.

- **debug**

 - msg: "{{ VARIABLE_NAME }}"

- **debug:**

 - var: "{{ VARIABLE_NAME }}"

register

success -> **succeeded**

성공은 succeeded로 표현

fail -> **failed**

실패는 failed라고 표현

skip -> **skipped**

무시는 skipped

Boolean

앤서블에서 불린 값을 사용하기 위해서는 다음과 같이 사용한다.

yes / on / 1 / true

위의 모든 내용은 참을 나타낸다.

Boolean

앤서블에서 불린 값을 사용하기 위해서는 다음과 같이 사용한다.

`yes / on / 1 / true`

반대는

`no / off / 0 / false`

위의 모든 내용은 참을 나타낸다.

Boolean

이걸 컨디션으로 사용하기 위해서는 다음처럼 한다.

`vars:`

`test1: true`

`test2: "yes"`

- `name: Boolean true`
`shell: echo "This is true"`
`when: test1 or test2 | bool`

Boolean

vars:

test1: true

test2: "yes"

- name: Boolean false
shell: echo "This is true"
when: not test1

대화모드

Interactive

앤서블은 일반적으로 비대화 모드 상태에서 동작한다.

하지만, 여러가지 이유로 대화모드로 구성을 필요한 경우가 있다.

많은 사용자들은 앤서블 VS 테라폼 비교 하였을 때 대화모드가 안된다고 생각 하기 때문에 테라폼 제품을 사용하는 경우가 많다.

하지만...?!

Interactive

대화형 모드의 시작은 변수 구성으로 시작한다.

하지만, 조금 특이하다. 다른 언어와 다르게 변수 선언 시 인터프리터를 명시해야 한다.

vars_prompt:

- name: username

- prompt: Put down your name

- private: no

- name: password

- prompt: Put your password

Interactive

만약, 기본값을 선언한 상태에서 사용하고 싶으면 다음처럼 **기본값**을 설정한다.

vars_prompt:

- name: nic_name

prompt: put the NIC card name

default: "ens4"

Interactive

입력된 값을 해시 혹은 암호화 하기 위해서 다음처럼 옵션을 구성한다.

vars_prompt:

- name: new_password

prompt: enter new password

private: yes

encrypt: sha512_crypt

confirm: yes

salt_size: 10

Interactive + encrypt

des_crypt – DES Crypt

bsdi_crypt – BSDi Crypt

bigcrypt – BigCrypt

crypt16 – Crypt16

md5_crypt – MD5 Crypt

bcrypt – BCrypt

sha1_crypt – SHA-1 Crypt

Interactive + encrypt

sun_md5_crypt - Sun MD5 Crypt

sha256_crypt - SHA-256 Crypt

sha512_crypt - SHA-512 Crypt

apr_md5_crypt - Apache's MD5-Crypt variant

phpass - PHPass' Portable Hash

pbkdf2_digest - Generic PBKDF2 Hashes

cta_pbkdf2_sha1 - Cryptacular's PBKDF2 hash

dlitz_pbkdf2_sha1 - Dwayne Litzenberger's PBKDF2 hash

scram - SCRAM Hash

bsd_nthash - FreeBSD's MCF-compatible nthash encoding

Interactive

특수문자를 입력 받기 위해서는 다음과 같은 옵션을 사용한다.

vars_prompt:

- name: new_password
prompt: enter new password
private: yes
encrypt: sha512_crypt
confirm: yes
salt_size: 10
unsafe: yes

루프

loop

앤서블 루프 명령어 키워드는 다음과 같다.

loop
with_
until

loop

loop 키워드는 앤서블 2.5에서 추가가 되었다.

하지만, 이 명령어는 with_명령어를 전부 대체하지는 못하고 있지만, 가급적이면 loop명령어를 사용해서 처리한다.

loop

`with_` 키워드는 아직은 구형 명령어(deprecated)으로 전환이 되지 않지만, 이는 호환성 유지를 위해서 존재한다.

loop vs with_

<https://github.com/ansible/ansible/tree/devel/changelogs>

loop

루프는 하나 이상의 값을 반복적으로 모듈에 전달할 때 사용한다.

```
- name: add users
```

```
  user:
```

```
    name: "{{ item }}"
```

```
    state: present
```

```
    groups: "wheel"
```

```
  loop:
```

```
    - sktuser1
```

```
    - sktuser2
```


loop with list

혹은 이걸 리스트를 통해서 전달 받을 수 있다.

```
vars:
```

```
  userlist:
```

- sktuser1
- sktuser2

```
  groups: "wheel"
```

```
loop: "{{ userlist }}"
```

loop with dict

호~~~은~~ 딕셔너리를 다음처럼 핸들링이 가능하다. x 1

- name: add users

user:

name: "{{ item.name }}"

state: present

groups: "{{ item.groups }}"

loop:

- { name: 'sktadm', groups: 'wheel' }

- { name: 'sktuser', groups: 'adm' }

루프

loop with dict

호~~~은~~ 딕셔너리를 다음처럼 핸들링이 가능하다. x 2

- name: add users

user:

name: "{{ item.username }}"

state: present

groups: "{{ item.groups }}"

loop: "{{ user_data | dict2items }}"

vars:

user_data:

username: toor

groups: wheel

loop with register

루프는 레지스터와 함께 사용이 가능하다.

```
- name: register with loop
  shell: "echo {{ item }}"
  loop:
    - "one"
    - "two"
  register: echo
```

loop with register

- name: dumped the echo register

debug:

msg: "{{ echo }}"

loop with register

```
- name: Fail if return code is not 0
  fail:
    msg: "The command ({{ item.cmd }}) is not return 0"
  when: item.rc != 0
  loop: "{{ echo.results }}"
```

loop with register

- name: In loop and looping the result file:
 - path: "/tmp/{{ item }}"
 - state: touch
- loop:
 - one
 - two
- register: echo

complex loop

이러한 방식으로 잘 사용하지는 않지만, 다음과 같은 방식으로 리스트 사용하기도 한다.

```
- name: mysql user access
```

```
  mariadb_user:
```

```
    name: "{{ item[0] }}"
```

```
    priv: "{{ item[1] }}.*:ALL"
```

```
    append_privs: yes
```

```
    password: "marriadb"
```

```
  loop:
```

```
    - "{{ ['sktdb1', 'stkdb2'] | product(['sample_db1', 'sample_db2', 'sample_db3']) | list }}"
```


**예외처리
+ 살짝**

Until

예외 처리는 다양한 방법과 조건이 따라온다.

그래서 이 교육에서는 간단하게 앤서블 오류 핸들링 종류 및 사용방법에 대해서 학습한다.

Until

만약 특정 모듈에서 특정조건을 충족할 때까지 조건 대기하기 위해서는 `until`이라는 추가 조건을 사용한다.

```
- shell: /bin/noexistcmd
  register: result
  until: result.stdout.find("I am okay") != -1
  retries: 2
  delay: 5
```

Until

위의 조건은 "I am okay"라는 문자열이 레지스터 `result`에 있으면 참이 되기 때문에 `!= -1` 성립이 된다.

하지만 위의 명령어가 없기 때문에 올바르게 값을 반환한다.

그래서 최대 2번 그리고 5초 동안 반복하게 된다.

block/rescue/always

앤서블에서 제일 많이 사용하는 블록 핸들링 오류 처리이다.

block

rescue

always

+

any_errors_fatal

block/rescue/always

앤서블에서 제일 많이 사용하는 블록 핸들링 오류 처리이다.

- name: start the block session

- block:

- yum:

- name: httpd

- state: present

- register: result_package

block/rescue/always

rescue:

- yum:

 - name: httpd

 - state: absent

always:

- debug:

 - msg: "Always runs it"

block

block영역은 항상 실행이 되는 부분이다.

이 안에서는 여러 개의 모듈 구성이 가능하며, 이들은 마치 하나의 클래스처럼 같이 실행이 되며 하나의 모듈에서 문제가 발생하는 경우 나머지 모듈 실행은 중지가 된다.

block

httpd, vsfoo 패키지를 설치 시도. vsfoo는 존재하지 않기 때문에 실패.

block:

- yum:

- name: "{{ item }}"

- state: present

- loop:

- httpd

- vsfoo

rescue

`block`부분에서 오류가 발생하면 `rescue`바로 설정된 작업을 차례대로 수행한다. 보통은 `block`에서 중지된 작업에 대해서 `rollback`작업을 수행한다.

rescue

`block`: 작업이 실패하면 `httpd`패키지 제거를 시도한다.

– `yum`:

`name`: `httpd`

`state`: **`absent`**

always

"always:"는 mandatory 키워드 문법은 아니다.

"block:", "rescue:" 수행이 되면 무조건 "always:" 명시된 작업들은 수행이 된다.

always

- debug:

 - msg: "Always runs it"

always example

always:

- debug:
 msg: "the httpd package installed"
 when: result_package.rc
- debug:
 msg: "the httpd package removed"
 failed_when: result_package.rc != 1

ignore_

ignore_unreachable

ignore_errors

failed_when

changed_when

ignore_unreachable

서버는 동작하나, 네트워크나 혹은 관리 서버의 워크로드 인하여 접근이 되지 않는 경우 "ignore_unreachable"라는 키워드를 통해서 접근 오류가 발생하면 무시하고 정상 수행이 가능하다.

- hosts: server_ok, server_error

- tasks:

- command: /bin/true

- ignore_unreachable: yes

- command: echo "This command will be working without problem"

ignore_errors

종류에 상관 없이 모듈에 오류가 발생하면 중지하지 않고 다음 작업으로 진행한다.

- hosts: servera, serverb
tasks:
 - command: /bin/noexist_cmd
ignore_errors: yes
 - command: echo "This command will be working without problem"

failed_when

실패 조건을 정의한다. 특정 조건이 성립이 되는 경우, 해당 작업은 실패로 간주한다.

- name: Fail the task if "FAILED" string in the output 1
command: ls -al
register: command_result
failed_when: "'FAILED' in command_result.stderr"

failed_when

자주 사용하는 방법은 Return Count기반으로 확인을 많이한다.

- name: Fail the task if "FAILED" string in the output 2

debug:

msg: "Failed Failed Failed!!!"

failed_when:

- result.rc == 0

failed_when

혹은 두 가지 방식을 동시에 사용이 가능하다.

- name: Fail the task rc and String exist.

command: failed-command -h -e -l -l -o

register: command

failed_when: command.rc == 1 or "'FAILED' in command.stderr"

changed_when

특정 작업이 수행이 된 다음에 "변경"으로 표시를 하고 싶은 경우 `changed_when`를 사용한다.

강제로 "changed" 상태로 전환하는 작업이기 때문에 실제로는 변경이 안된 부분도 변경으로 상태 출력한다.

changed_when

– name: shows 'changed' when the return code is not equal to 2

shell: ls -al

register: result

changed_when: result.rc != 2

changed_when

- name: play sounds beep when the command status is "changed"

shell: echo -ne '\007'

register: result

changed_when:

- result.rc == 2
- '"ERROR" in result.stderr'

Rejecting modules

특정 모듈을 사용을 하지 않기 위해서 `avoid(무시)` 목록을 만들 수 있다.

`/etc/ansible.cfg`의 `[defaults]`에서 `plugin_filter_cfg`에 리스트 파일을 명시한다.

```
[defaults]
```

```
plugin_filter_cfg = plugin_filters.yaml
```

Rejecting modules

`plugin_filters.yaml`

`filter_version: '1.0'`

`module_rejectlist:`

- `- docker`
- `- easy_install`

테스트 문법

is + match/search/regex

테스트 문법은 필터 문법을 사용해서 값이 올바르게 구성이 되었는지 확인한다.

앤서블 2.5버전 이후부터 이 기능을 제공하게 되었고 문법은 다음과 같다.

```
variables is test_name 혹은 result is failed
```

is + match/search/regex

예를 들어서 "특정 URL를 통해서 API를 주소를 확인한다."
라는 조건이면 보통 다음과 같이 검증을 한다.

is + match/search/regex

예를 들어서 사용하는 가상 시스템의 URL이 아래와 같다.

<https://vms.example.com/users/skt/resource/cinder>

이걸 코드로 변경하면 다음과 같이 표현이 가능하다.

is + match/search/regex

vars:

url: <https://vms.example.com/users/skt/resource/cinder>

tasks:

- debug:

 - msg: "matched pattern 1"

 - when: url is

 - match(<https://vms.example.com/users/./resources>)

is + match/search/regex

– debug:

msg: "pattern 2"

when: url is search("users/./resources/.*")

is + match/search/regex

- debug:

 - msg: "pattern 3"

 - when: url is search("users")

- debug:

 - msg: "pattern 4"

 - when: url is regex("example\\.com/w+/foo")

다중 플레이북

roles vs playbook

이쯤이면 슬슬 혼동이 오는 부분, 바로 `roles vs playbook`이다.

`roles`를 `tasks`, `files`, `templates`같은 기능들이 모여서 하나의 기능을 구현하여 이걸 호스트에 제공하는 기능.

시작 전 간단한 설명으로 시작하면...

roles vs playbook

`playbook` 경우에는 하나 이상의 `role(s)` 모아서 특정 작업에 대한 프로세스를 구현하고 있는 파일.

예를 들어서 `webserver.yaml`, `dbserver.yaml` 이런 식으로 좀 더 추상적인 작업의 워크 플로우를 `role` 기반으로 구성하고 있다.

roles vs playbook

위의 플레이북은 항상 다음과 같은 내용을 가지고 있다.

1. What Happened to which Host?
2. What Happened with what kind of roles
3. What kind of tasks defined in the YAML

_roles

roles

`roles`는 기본적으로 디렉터리 기반의 프레임워크를 가지고 있다.

디렉터리 생성 방법은 두 가지 방법이 있다.

```
$ ansible-galaxy init httpd
```

```
$ mkdir -p roles/httpd/tasks
```

```
$ touch roles/httpd/tasks/main.yaml
```

roles

위의 디렉터리 `httpd`기반으로 `role`구성을 설명하면 다음처럼 디렉터리 및 파일을 생성 및 구성한다.

```
$ mkdir -p roles/httpd/tasks/
```

```
$ vi roles/httpd/tasks.main.yaml
```

roles

```
$ vi roles/httpd/tasks/main.yaml
```

```
- name: install a httpd package
```

```
  yum:
```

```
    name: httpd
```

```
    state: latest
```

```
- name: template a httpd configuration file
```

```
  template:
```

```
    src: httpd.conf.j2
```

```
    dest: /etc/httpd/conf/httpd.conf
```

roles

```
$ mkdir roles/httpd/templates/  
$ touch roles/httpd/templates/httpd.conf.j2  
# grep -Ev "^#|^$|*#" /etc/httpd/conf/httpd.conf
```


role directory

`roles`에 제공 혹은 필요한 디렉터리는 다음과 같다.

`roles/httpd/`

`tasks/`: 기본 디렉터리.

이 디렉터리가 없으면 올바르게 `tasks`를 불러오지 못한다.

`files/`: `copy`같은 명령어로 복사 시 사용하는 디렉터리.

`templates/`: `template`모듈로 템플릿 기반으로 파일 생성시 사용.

`vars/`: 변수 디렉터리

`defaults/`: 기본 값을 정의하는 디렉터리

`roles/httpd/`

`meta/`: 메타에는 `roles`의 설명 그리고 의존성 같은 부분에 대해서 구성.

role directory

기본적으로 동작에 필요한 디렉터리는 **tasks**만 있어도 된다.

나머지 특정 디렉터리에 대해서는 앤서블 사이트를 참조하여 필요한 경우 그때그때 참조해서 만들면 된다.

다만, 앤서블 갤럭시를 통해서 배포하는 경우 반드시 **meta/**에 메타 정보를 꼭 작성해주어야 한다.

include

"**tasks**"에서 한 개 이상의 YAML파일이 더 필요하다. 그래서 한 개 이상의 **task**파일을 불러와야 하는 경우, 다음처럼 파일을 구성한다.

tasks/main.yaml

- include: packages.yaml
- include: template_config.yaml
- include: hostname.yaml

include

`tasks`에서 한 개 이상의 YAML파일이 더 필요하다. 그래서 한 개 이상의 `task`파일을 불러와야 하는 경우, 다음처럼 파일을 구성한다.

`tasks/main.yaml`

- `include: packages.yaml`
- `include: template_config.yaml`
- `include: hostname.yaml`

roles

```
$ vi roles/httpd/tasks/packages.yaml
```

```
$ vi roles/httpd/tasks/template_config.yaml
```

```
$ vi roles/httpd/tasks/hostname.yaml
```

include_roles

은근히 잘 쓰지 않지만(?), 자주(??) 사용하는 기능.

include_roles

- name: include roles from role directory

include_role:

name: testRole

tasks_from: verify_httpd_srv

include_roles

- name: include roles from role directory

include_role:

name: testRole

tasks_from: verify_httpd_srv

role디렉터리 testRole를 불러온다

tasks디렉터리에 있는
"verify_httpd_srv.yaml"파일을 불러온다

_playbook

include_playbook

- **include_playbook:** prepare_os.yaml
- include_playbook: webserver.yaml
- include_playbook: dbserver.yam

import_playbook

`import_playbook` 말 그대로 `playbook` 불러와서 메모리에 적재하는 키워드 명령어.

미리 메모리에 적재를 하기 때문에 실행 전 문법 검사가 선행되기 때문에, 문법에 문제가 있는 경우 실행이 되지 않는다.

include_playbook

`include_playbook`은 앤서블 인터프리터가 해당문법(`include`)를 만나면 그 순간 이벤트가 발생하여 명시된 파일을 불러 옵니다.

장점은 빠르지만, 단점은 불러오는 순간에 문법 검사를 하기 때문에 문법에 문제가 있는 경우 중간에 종료가 됩니다.

이전 버전에서는 `include`만 존재 하였다.

include_playbook

```
$ vi roles/prepare_os/tasks/main.yaml
```

```
- name: update all of thing
```

```
  yum:
```

```
    name: *
```

```
    state: latest
```

include_playbook

```
$ vi roles/webserver/tasks/main.yaml
```

```
- name: install a httpd package
  yum:
    name: httpd
    state: latest
- name: start and enable the httpd service
  service:
    name: httpd
    enabled: yes
    state: started
```

include_playbook

```
$ vi roles/webserver/tasks/main.yaml
```

```
- name: install a mariadb
  yum:
    name: "{{ item }}"
    state: latest
  loop:
    - mariadb
    - mariadb-server
- name: start and enable the mariadb service
  service:
    name: mariadb-server
    enabled: yes
    state: started
```

include_playbook

```
$ vi prepare_os.yaml
```

```
- name: Updated to all node hosts recently package  
  hosts: all  
  roles:  
    - { role: prepare_os }
```


include_playbook

```
$ vi webserver.yaml
```

```
- name: Updated to all node hosts recently package  
  hosts: webserver  
  roles:  
    - { role: webserver }
```

include_playbook

```
$ vi dbserver.yaml
```

```
- name: install and configuration the database server
  hosts: dbserver
  roles:
    - { role: dbserver }
```

include_playbook

```
$ vi allinone_playbooks.yaml
```

- include_playbook: prepare_os.yaml
- include_playbook: webserver.yaml
- include_playbook: dbserver.yam

is + match/search/regex

- debug:

 - msg: "pattern 3"

 - when: url is search("users")

- debug:

 - msg: "pattern 4"

 - when: url is regex("example\\.com/w+/foo")

정적/동적
핸들러

handler, notify

핸들러(handler)는 특정한 이벤트가 발생하면 **goto**명령어처럼 특정 블록으로 넘어가서 작업 수행 후 다시 본래 작업 위치로 돌아 온다.

문법은 보통 다음처럼 작성한다.

handler, notify

- name: install a httpd package

yum:

name: httpd

state: latested

- name: enabled and start the httpd service

service:

name: httpd

state: started

handler, notify

- name: a copy the index.html file into /var/www/html

 - copy:

 - src: index.html

 - dest: /var/www/html/index.html

 - notify:

 - restart httpd srv

- handlers:

 - name: restart httpd srv

 - service:

 - name: httpd

 - state: restarted

handler, notify

- name: a copy the index.html file into /var/www/html

copy:

src: index.html

dest: /var/www/html/index.html

notify:

- restart httpd srv

- handlers:

- name: restart httpd srv

service:

name: httpd

state: restarted

작업이 문제 없이 수행이 되면
notify에 명시된 이름을 호출한다.

작업은 handlers에 명시가 되어있다.

notif에 명시된 이름은 "restart
httpd srv"
해당 이름에 명시된 모듈을 실행한다.

최종 연습문제 플레이북

첫번째 문제

첫번째

-