

AUTOMATE ANSIBLE

앤서블 기본

LAB

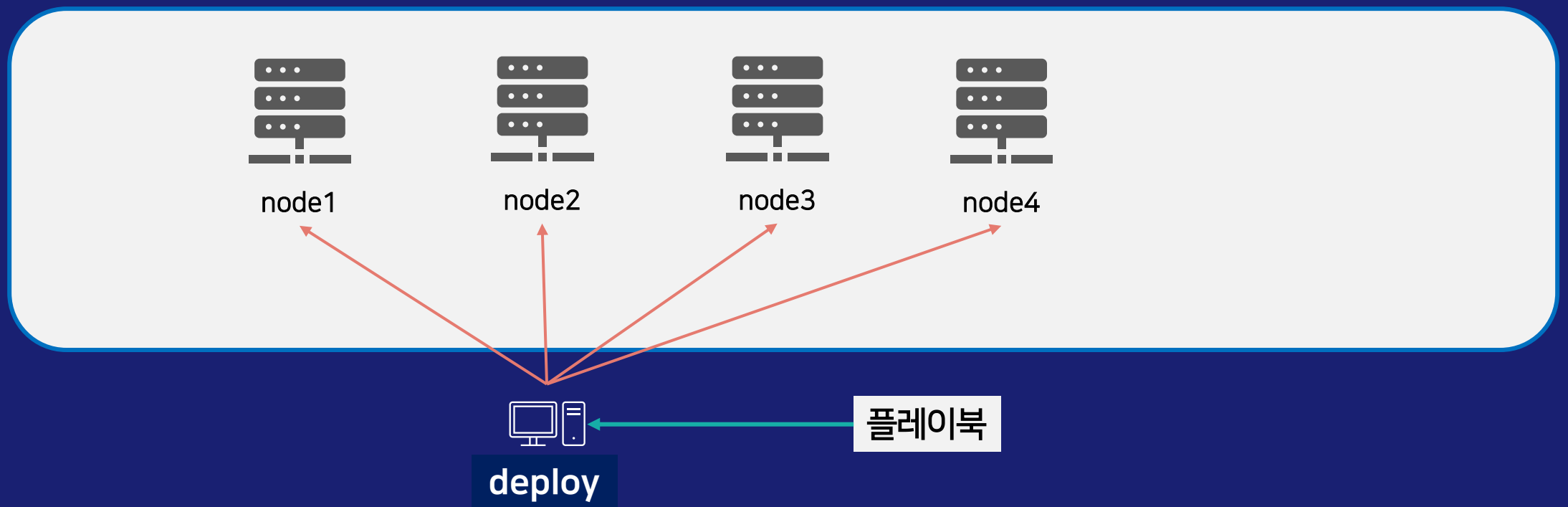
랩 및 교육대상

앤서블 기초

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

랩 다이어그램



소개

앤서블 기능

앤서블

앤서블은 본래 Ansible Community, Company에서 제작하였고 도중에 레드햇이 인수하였음. 앤서블은 총 두 가지 릴리즈를 유지하고 있음.

- Ansible Core
- Ansible Engine(이전 앤서블 네이밍)

앤서블

코어는 앤서블 핵심 모듈로 구성이 되어 있으며, 그 이외 확장으로 `posix`, `collection`, `community`로 통해서 확장이 가능하다. 현재 앤서블은 앤서블 코어로 통합 및 배포가 되고 있다.

각 확장은 "제 3자 제공" 혹은 "커뮤니티"로 나누어져 있고, 주요 코어 모듈 혹은 확장 모듈 경우 "Redhat" 기여 혹은 제공을 하고 있음. 코드상 차이는 크게 없음.

1. 기본적으로 많이 사용하는 모듈(제공) `ANSIBLE CORE`에서는 `POSIX/BUILT-IN`이다.
2. 그 이외 나머지 기능들은 `collection`(`community`, `vender`)로 확장.

조금 혼돈스럽기는 하지만, 앤서블 코어를 엔진(engine)이라고 표현하기도 한다. `core`, `engine` 둘 다 같은 기능이다. 다만, 현재는 `ansible-engine`에서 `ansible-core`로 전환을 하고 있다. 마이그레이션 시, 변경이 되어야 하는 명령어가 있음.

앤서블 프로그램

또한 앤서블 코어는 두 가지 릴리즈 방식이 있다.

`ansible-core`

- 앤서블 코어는 앤서블 인터프리터 + 코어 모듈

`ansible-project`

- 앤서블 코어 + 추가적인 컬렉션 구성

`ansible-navigator`

- 앤서블 통합 도구. 해당 명령어로 플레이북 관리 및 실행이 가능하다. 다만, 현재 레드햇 배포판 제외하고 다른 모든 리눅스 배포판에서는 PIP를 통해서 설치해야 한다.

앤서블 프레임 워크 월드(?)

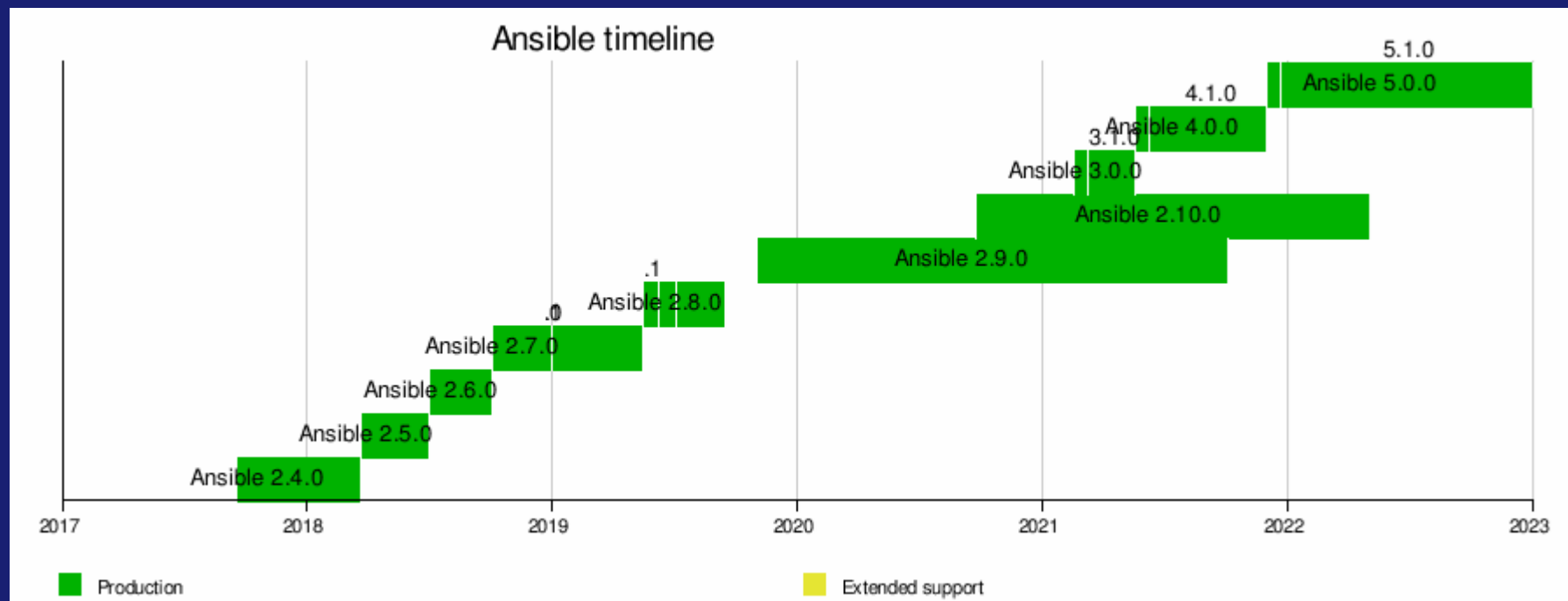
앤서블 AAP(Ansible Automate Platform)

앤서블 기반으로 모든 플랫폼에 대해서 자동화를 제공 합니다. 앤서블 AAP의 핵심 기술은 YAML기반의 자동화 선언기술 및 앤서블 네비게이션 그리고, 갤럭시(역할(role))기반으로 표준화 및 확장이 가능합니다.

앤서블 Galaxy

앤서블 갤럭시는 오토메이션 개발자(autodev)가 작성한 표준 자동화 사양 모듈을 갤럭시라는 네트워크에 올려서 배포 및 활용이 가능하다. 앤서블 갤럭시는 내외부망 관계 없이 http/https/git기반으로 사용이 가능하다.

앤서블 EOL



앤서블 EOL

Version	Support	End Of Life	Control Node Python	Target Python / PowerShell
2.16	GA: 06 Nov 2023 Critical: 20 May 2024 Security: Nov 2024	May 2025	Python 3.10 - 3.12	Python 2.7 Python 3.6 - 3.12 Powershell 3 - 5.1
2.15	GA: 22 May 2023 Critical: 06 Nov 2023 Security: 20 May 2024	Nov 2024	Python 3.9 - 3.11	Python 2.7 Python 3.5 - 3.11 PowerShell 3 - 5.1
2.14	GA: 07 Nov 2022 Critical: 22 May 2023 Security: 06 Nov 2023	20 May 2024	Python 3.9 - 3.11	Python 2.7 Python 3.5 - 3.11 PowerShell 3 - 5.1
2.13	GA: 23 May 2022 Critical: 07 Nov 2022 Security: 22 May 2023	EOL 06 Nov 2023	Python 3.8 - 3.10	Python 2.7 Python 3.5 - 3.10 PowerShell 3 - 5.1

앤서블 릴리즈



앤서블



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 타워

앤서블 타워는 두 가지 버전으로 구성이 되어있다. 첫 번째는 앤서블 AWX 업 스트리밍 버전이 있으며 레드햇에서 제공하는 서브스크립션 버전.

1. 제일 큰 차이점은 버전 픽스가 되지 않는 롤링 업데이트 버전이다.
2. 기술지원이 되지 않음.

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

앤서블

앤서블 사용하기 전에 준비를 해야 될 부분은 다음과 같다. YAML 작성 시 사용할 에디터. 아무거나 좋다! 정말로!

- 에디터(vi, vim, neovim, nano)
- YAML 문법
- ansible.cfg 및 directory work frame
- ansible, ansible-playbook명령어 사용 방법

앤서블

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

거의 대다수 리눅스는 기본적으로 vi는 설치가 되어 있으나, vim설치가 되어 있지 않는 경우가 있다. 이러한 이유로 vim설치 이외 몇 가지 기능을 추가적으로 구성 및 활성화 해야 한다.

사용하기 어려운 경우에는 nano를 사용하여도 된다. vi/vim를 사용하는 경우, 새로운 에디터인 **neovim+ale**를 사용을 권장한다. 아래 명령어로 **neovim/ale**기반으로 에디터 환경을 구성한다.

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

앤서블

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

앤서블

%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, 보안키 기반으로 접근을 권장한다.

SSH

두번째 서버의 아이피 주소를 아래에 넣어주세요.(RHV 대시보드에서 구성이 되어 있는 두번째 서버의 아이피 주소)

```
deploy# ssh-keygen -t rsa -N '' -f ~/.ssh/id_rsa  
deploy# ssh-copy-id root@192.168.90.X
```

혹은, 아래 슬라이드와 같이 파일을 작성 후, 공개키 배포가 가능합니다.

SSH

```
deploy# vim sendsshkey.yaml
---
- hosts: all
  ansible_user: root
  ansible_password: centos
  tasks:
    - authorized_key:
        user: root
        key: "{{ lookup('file', '/home/' + lookup('env', 'USER') +
'{{ k8s_public_rsa_locate }}"
        key: "{{ lookup('file', '/root/' + '.ssh/id_rsa.pub') }}"
...

```

에디터

이 문서는 아래와 같은 조건으로 YAML파일 작성이 되었습니다.

- 들여쓰기 2칸
- 탭은 빈 공간(space)으로 전환
- 자동 들여쓰기 활성화
- nano, vim사용가능

vim editor

vi/vim를 위해서 기능 강화를 합니다.

```
deploy# touch .vimrc
```

```
deploy# sudo yum install vim vim-enhanced neovim yamllint -y
```

혹은 centos 8버전 이후를 사용한다면

```
deploy# sudo yum install vim-ansible
```

vim + ale

windows

```
> curl.exe https://webi.ms/vim-ale | powershell
```

linux

```
# curl -sS https://webi.sh/vim-ale | sh
```

mac

```
# curl -sS https://webi.sh/vim-ale | sh
```

.vimrc

```
$ vi ~/.vimrc  
  
set ts=2  
  
set sts=2  
  
set sw=2  
  
set expandtab  
  
set number  
  
syntax on  
  
filetype indent plugin on
```

nano

나노 에디터를 사용하는 경우, 아래 명령어로 나노 에디터 설정 및 구성이 가능하다.

```
$ sudo yum install nano wget curl  
$ sudo curl  
https://raw.githubusercontent.com/scopatz/nanorc/master/install.sh | sh
```

혹은, 수동으로 구성 시, 아래와 같이 "~/.nanorc"파일을 생성한다.

nano

```
syntax "YAML" "\.ya?ml$"
header "^(---|===)" "%YAML"
color magenta "^\\s*[\\$A-Za-z0-9_-]+\\:"
color brightmagenta "^\\s*@[\\$A-Za-z0-9_-]+\\:"
color white ":[\\s.+$"
icolor brightcyan " (y|yes|n|no|true|false|on|off)$"
color brightred " [[:digit:]]+(\\. [[:digit:]]+)?"
color red "\\[" "\\]" ":[\\s+ [>]" "^\\s*- "
color green "(^| )!!(binary|bool|float|int|map|null|omap|seq|set|str) "
```

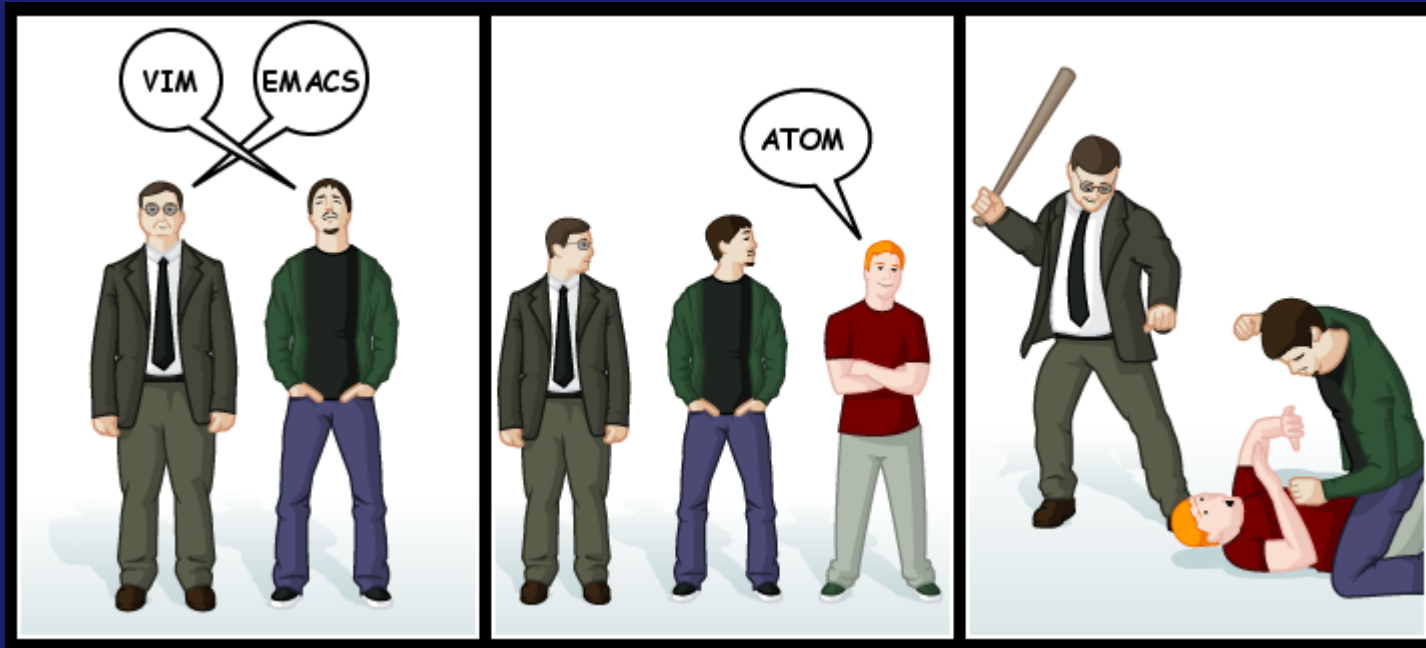
.nanorc

```
color brightwhite "#.*$"
color ,red ":\w.+$"
color ,red ":' .+$"
color ,red ":'\".+$"
color ,red "\s+$"
color ,red "['\\"]^[['\\"]]*$"
color yellow "['\\"].*['\\"]"
color brightgreen ":( |$)"
set tabsize 2
set tabstopotospacs
```

.nanorc

```
# dnf search epel  
# dnf install epel-release -y  
# dnf install ansible -y  
# pip3 dnf install ansible-navigator --user
```

앤서블 준비



앤서블 준비



SSH KEY

키 배포

비밀번호 접근

키 배포

앤서블은 앞에서 잠깐 언급 하였지만 SSH를 사용해서 managed node에 접근. 배포를 하기 위해서는 다음과 같은 명령어를 통해서 생성 및 진행 해야 한다.

가급적이면, 앤서블 사용자는 "root"계정이 아닌, 일반 계정에서 실행을 권장.

비공개키 생성

```
$ adduser ansible
$ echo ansible | passwd --stdin ansible
$ usermod -aG wheel
$ vi /etc/sudoers.d/ansible
> %wheel  ALL=(ALL)        ALL
> %wheel  ALL=(ALL)        NOPASSWD: ALL
$ ssh-keygen -t rsa -N '<PASSPHASE>' -f '<SSH_KEY_DIRECTORY>'
$ sshpass -p <PASSWORD> ssh-copy-id <USER>@<HOST>
```

비공개키 생성

"ssh private"키 생성에 관련된 앤서블 모듈은 아직 없기 때문에 생성을 원하는 경우에는 'command', 'shell'와 같은 모듈을 통해서 생성해야 한다.

혹은, 커뮤니티 모듈인 "ansible-galaxy collection install community.crypto"를 사용해야 한다.

```
- name: Generate an OpenSSH rsa
  community.crypto.openssh_keypair:
    path: /tmp/id_ssh_rsa
    size: 2048
```

키 배포

혹은 'authorized_keys'기능을 사용해서 ssh공개키 배포가 가능하다. 아래 예제 코드가 있다. 이 코드 내용은 앤서블 웹 사이트에서 확인이 가능하기 때문에, 자세한 내용은 아래 링크에서 확인이 가능하다.

https://docs.ansible.com/ansible/latest/collections/ansible/posix/authorized_key_module.html

키 배포

```
- hosts: all
  tasks:
    - authorized_key:
      user: testuser
      key: "{{ lookup('file', '/home/' + lookup('env', 'USER') + '/.ssh/id_rsa.pub') }}"
```

위치

형식(파일)

비밀번호 접근

만약, 특정한 이유로 보안키로 서버나 혹은 네트워크 장비에 접근이 안되는 경우, 다음과 같은 방법으로 원격 서버 혹은 장비에 접근이 가능하다. 아래 코드는 플레이 북에서 작성한다.

```
become: false  
become_password: rocky
```

아래 코드는 인벤토리에 설정하는 내용이다.

```
localhost ansible_user=ansible ansible_password=ansible  
ansible_become_password=ansible
```


앤서블 명령어

ansible

ansible-playbook

ansible-navigator

ansible ad-hoc

앤서블에서 모듈 기반으로 사용하는 명령어. 일반적으로 ad-hoc처리시 많이 사용한다. 보통 쉘 스크립트를 앤서블 모듈 기반으로 처리하고 싶은 경우 많이 사용한다. 하지만, systemd로 들어 오면서 점점 쉘 스크립트를 사용하는 상황이 작아지면서 ansible명령어는 사용하는 영역은 작아지고 있다.

```
$ ansible -m shell -a 'cmd="ls -l"'
```

ansible-doc

앤서블 문서를 확인하는 방법은 두 가지가 있다.

1. 앤서블 온라인 문서
2. 앤서블 문서 명령어

앤서블은 모듈에 대한 맨-페이지를 제공하지 않기 때문에, 모듈 사용 방법을 확인하기 위해서는 'ansible-doc'명령어를 통해서 확인해야 한다.

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

ansible-playbook/navigator

앤서블 플레이북을 하나 이상 실행 시, 해당 명령어를 사용한다. 앤서블은 보통 하나 이상의 YAML기반으로 구성이 되어 있으며, 이를 실행해주는 프로그램은 'ansible-playbook'명령어. 이를 보통 **실행자(launcher)**라고 부르며, YAML및 Jinja2로 구성이 되어 있는 구성 파일을 메모리에 불러와서 컴파일 후 작업을 수행 및 실행한다.

'ansible-navigator'는 앤서블 런타임을 컨테이너 형태로 패키징이 되어 있는, 일종의 런타임이다.

이 런타임(컨테이너 이미지)을 통해서 앤서블 실행에 필요한 라이브러리 및 도구를 제공한다. 이 컨테이너는 루트리스 형태로 동작하며, 사용자가 명시한 데이터를 바인딩하여 실행한다. 단, 네비게이터는 레드햇 계열에서는 RHEL을 제외하고는 RPM패키지로 제공하지 않는다. (2024년 기준)

```
$ pip3 install 'ansible-navigator[ansible-core]'  
$ ansible-playbook -i hosts main.yaml  
$ ansible-navigator run main.yaml
```

ansible-tower

앤서블 타워는 웹 기반으로 플레이북/인벤토리/사용자를 관리 할 수 있는 도구이다. 대규모로 사용하는 개인 및 업체를 위한 도구이다. 앤서블 타워는 현재 두 가지 버전으로 구성되어 있다.

1. 앤서블 오토메이션 플랫폼(구: 앤서블 타워(레드햇 판매 버전))
2. 앤서블 AWS(커뮤니티 버전)

두 개의 제일 큰 차이점은, 버전 갱신 방식이 롤링 업데이트(rolling update), 릴리즈 업데이트(version release update)차이가 있다. 이 과정에서는 앤서블 AWX에서 다루지는 않는다.

앤서블 문법

문법 및 간단한 인벤토리

표준 데이터 파일

현재 오픈소스는 데이터 형식에 대해서 표준화를 진행하고 있다. 현재, 오픈소스에서 다음과 같이 자원 표준화를 진행하고 있다.

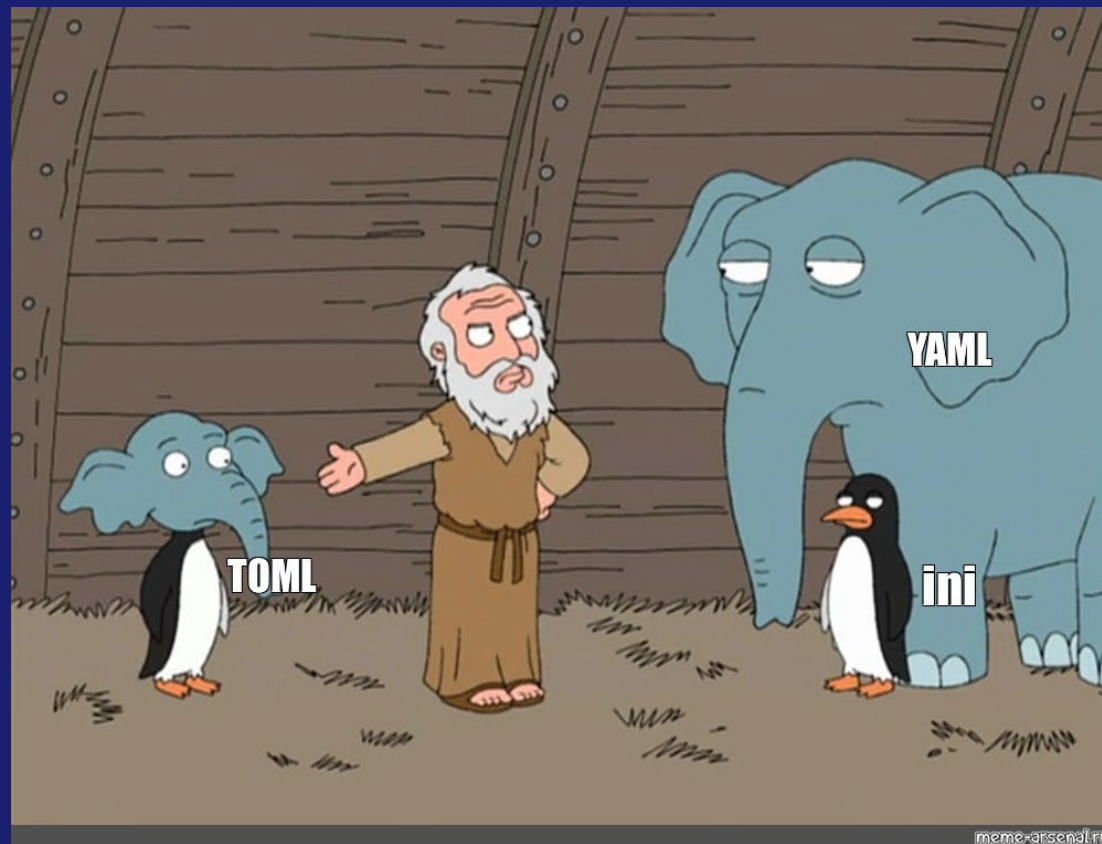
1. TOML
2. YAML
3. JSON

앤서블은 TOML/YAML/JSON를 전부 사용하고 있다. 자동화를 위한 작업 데이터 셋은 YAML으로 작성이 되며, 앤서블 내부적으로 데이터 핸들링은 JSON으로 데이터를 다룬다.

인벤토리와 같은 설정형식은 기존 INI에서 TOML으로 변환하고 있다. TOML를 앤서블 코어 2.8이후부터 지원한다.

https://docs.ansible.com/ansible/latest/collections/ansible/builtin/toml_inventory.html

YAML/JSON/TOML



YAML



YAML?

```
- name: simple playbook
  hosts: all
  become: false
  tasks:
    - name:
      module
      args1:
      args2:
```

YAML??

YAML를 형식(format)이 정해져 있다. 다만, 문제가 들여쓰기(indent)가 조금은 명확하지 않는 부분이 있다. 예를 들어서 리스트 혹은 딕셔너리 선언 기호인 "-" 경우에는 다음과 같은 문제가(?) 있다.

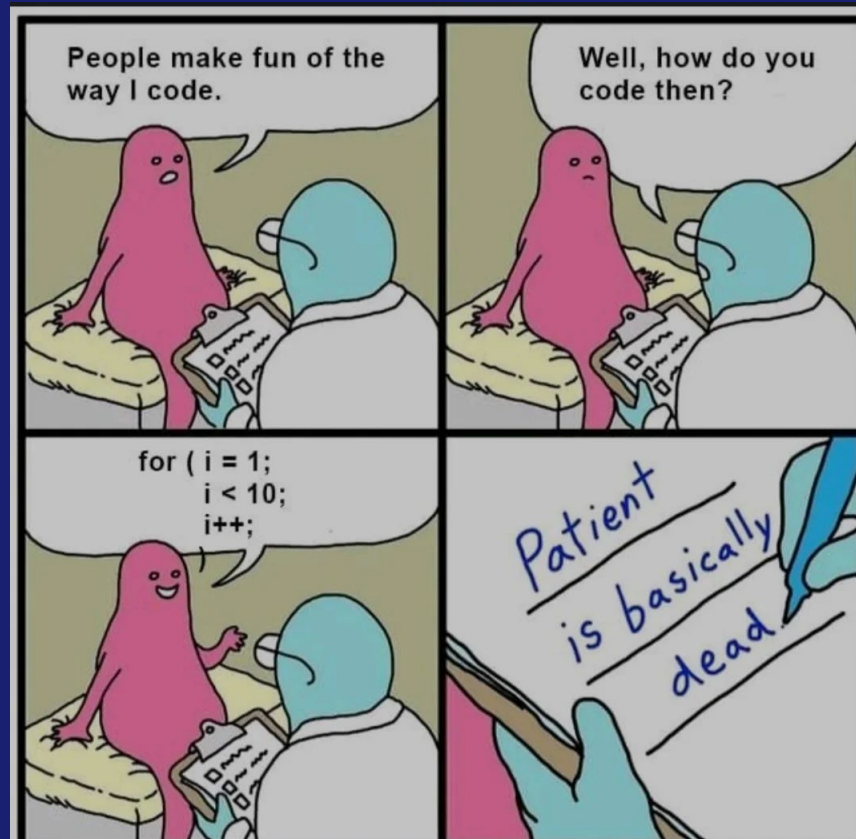
```
tasks:  
- test  
  parameter
```



```
tasks:  
  - test  
    parameter
```

위의 두 개는 들여쓰기가 다르지만, 결과적으로 같은 파싱(parsing)결과가 나온다. 결국 사람마다 다른 형식으로 작성하기 때문에, 적절하게 작업자끼리 형식에 대한 합의가 필요하다.

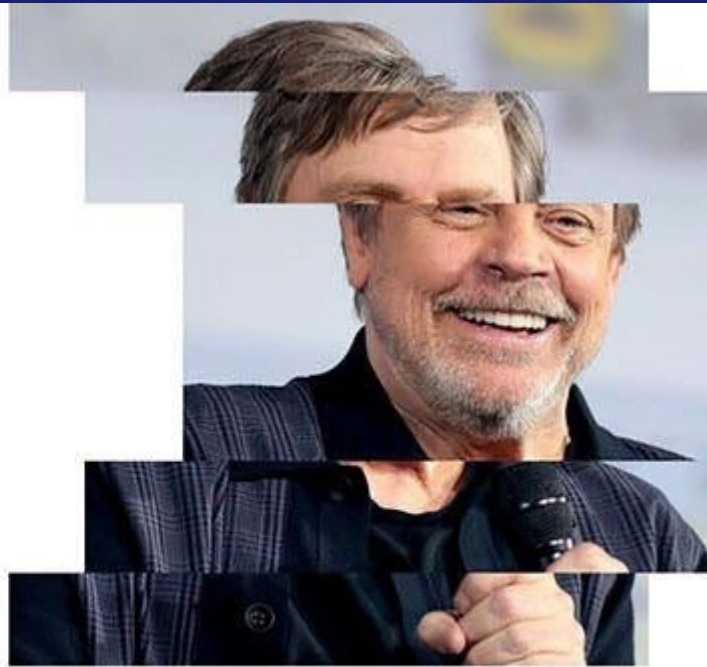
STYLE DEAL WITH DEATH



그러면..YAML?



Mark Hamill



Mark Yaml

TAB/SPACE

혼용은 금지! 결과는...?



5.5. White Space Characters

YAML recognizes two *white space* characters: space and tab.

```
[31] s-space ::= #x20 /* SP */  
[32] s-tab ::= #x9 /* TAB */  
[33] s-white ::= s-space | s-tab
```

YAML

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

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

YAML

그래서 YAML상단에는 다음과 같은 형태로 키워드 명령어를 사용한다.

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

작업이름

대상서버 이름

앤서블 내장 키워드

YAML

서버 접근 시, 사용하는 hosts키워드는 다음과 같은 미리 예약된 옵션 혹은 값이 있다. 일반적으로 많이 사용하는 값은 아래와 같다.

1. `localhost`: 127.0.0.1와 같은 자기 자신 루프 백(loopback)
2. `all` : 인벤토리에(inventory)등록된 모든 호스트
3. `[group]`: 특정 그룹에만 적용하는 명령어 키워드.그룹명은 반드시 중괄호와 같이 사용한다.

YAML INVENTORY

```
[PoC:children]
web
db
[web]
servera.node.example.com
serverb.node.example.com
[db]
serverc.node.example.com
serverd.node.example.com
[web:vars]
hostname=test.example.com
manager=tang
```

TOML INVENTORY

```
[PoC]
children = [ "test", "db" ]
vars = {hostname=test.example.com, manager=tang }

[web.hosts]
servera.node.example.com = {}
serverb.node.example.com = {}

[db.hosts]
serverc.node.example.com = {}
serverd.node.example.com = {}

[web.vars]
hostname=test.example.com
manager=tang
```

인벤토리 확인

앤서블에서 인벤토리가 올바르게 구성이 되었는지 아래와 같이 명령어로 렌더링 및 확인이 가능하다.

```
$ ansible-inventory -i <INVENTORY_FILE> --graph
$ ansible-inventory -i <INVENTORY_FILE> --list group
$ ansible-inventory -i <INVENTORY_FILE> --list host
$ ansible-inventory -i <INVENTORY_FILE> --list host --vars
```

전역 키워드 선언

전역 키워드는 "hosts:", "tasks:" 사이에 작성한다. 일반적으로 많이 사용하는 명령어는 아래와 같다. 이 이외 키워드 명령어는 보통 "ansible.cfg"이나 혹은 매뉴얼을 통해서 확인이 가능하다. 일반적으로 많이 사용하는 전역 키워드는 아래와 같다.

```
- hosts:
  vars:
    - var1
    - var2
  become:
  remote_user:
  task
```

tasks:

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

```
tasks:
```

```
- name:
```

```
tasks:
```

```
- name:
```

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

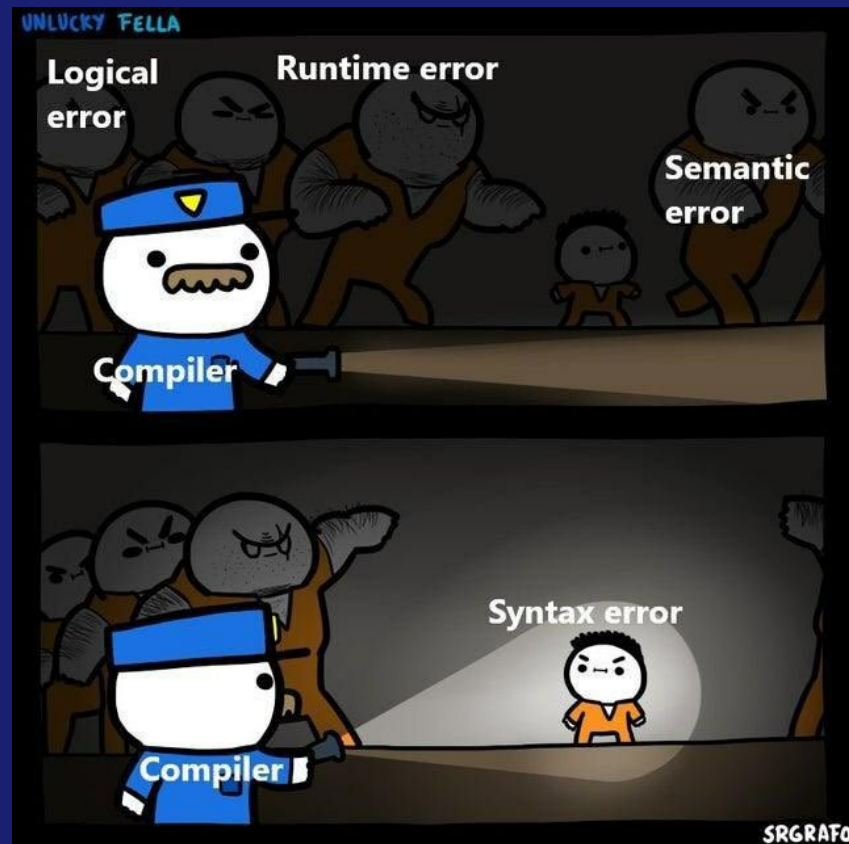
- name

YAML에서 작업 및 혹은 플레이북 작성 시, 각각 모듈 혹은 플레이북에 "name:" 키워드를 사용하여 작성 및 구성을 권장한다.

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

위와 같은 방법으로 명시한 모듈이나 플레이북에 어떠한 목적으로 동작하는지 간단하게 적는다. 이전에는 CJK를 지원하지 않았지만, 지금은 문제없이 지원하고 있다.

YAML



연습문제

아래와 같은 조건으로 인벤토리 및 간단한 INI, YAML형태로 "inventory-test.ini", "test-payload.yaml"를 작성한다.

1. 인벤토리 파일에 두 개의 그룹을 생성한다.
 - "webserver", "database" 그룹을 생성한다.
 - "webserver" 그룹에는 "nodea, nodeb"를 추가, "database" 그룹에는 "nodec", "noded"를 추가한다.
2. "test-payload.yaml"파일에는 다음과 같이 내용을 추가한다.
 - "the test payload"라는 이름으로 작업(task)를 생성한다.
 - 위의 작업 이름에 모듈은 아래처럼 작성한다.

```
debug:  
  msg: "Hello the first payload"
```

기본 모듈 및 패키지

설명 및 시도하기

MODULE

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

1. `ansible.core.*`
2. `ansible.posix.*`
3. `ansible.builtin`

앤서블에서 제공하는 핵심 기능. 일반적인 핵심 기능은 POSIX모듈에서 제공한다. 여기에 "copy", "file", "fetch"와 같은 자주 사용하는 기능이 포함이 되어 있다.

linux(rhel)-system-roles

앤서블에서 제공하는 확장 기능. 리눅스 배포판에 상관없이 사용이 가능하다. 일반적으로 리눅스 배포판에는 핵심 기능만 역할(role)로 제공하며, 추가 기능은 "linux-system-roles"에서 추가로 내려받기가 가능하다.

아래 링크에서 확인이 가능하다.

https://galaxy.ansible.com/ui/repo/published/fedora/linux_system_roles/content/

현재 기준으로 약 40개의 역할을 제공하고 있다.

ansible.collection

앤서블 확장 기능. 코어 기능에서 확장된 기능이며, 이걸 보통 컬렉션(collection)이라고 부른다. 컬렉션에는 밴더사에서 제공하는 모듈 기능도 포함이 되어 있다. 이전 앤서블 엔진 버전에서는 컬렉션 개념이 없었지만, 앤서블 코어로 넘어 오면서 추가가 되었다.

컬렉션은 FQCN(fully-qualified collection name)라고 부르며, 마치 도메인 시스템의 FQDN(fully-qualified domain name)과 비슷하게 말 장난이다. 😊

예를 들어서 대표적으로 많이 사용하는 "Linux System Role"도 컬렉션에 포함 되어 있다.

컬렉션은 'ansible-galaxy'통해서 검색 및 설치가 가능하다. 하지만, 버전에 따라서 갤럭시에서 컬렉션 검색이 되지 않기 때문에 직접 웹 사이트에서 직접 검색을 권장한다.

ansible.community

커뮤니티는 검증이 되지 않는 사용자 혹은 특정 그룹에서 만들어서 배포하는 Role 혹은 Module이다.

커뮤니티에 보면 보통 자신들이 만들어서 사용하는 플레이북 혹은 사용자화 된 Roles를 업로드 많이 한다. 커뮤니티도 컬렉션처럼 갤럭시를 통해서 설치가 가능하다.

```
# ansible-galaxy collection list
```

```
# ansible-galaxy collection install community.general
```

ansible.builtin

앤서블 코어에서 같이 배포하는 기본 모듈이다. 모든 앤서블 버전에서 동일하게 사용이 가능하며, 호출 이름은 "ansible.builtin.*"으로 시작한다. 아래 명령어로 빌트인 명령어 확인이 가능하다.

```
# ansible-doc --list | head
```

혹은 다음 명령어로 모든 모듈(명령어) 출력이 가능하다.

모듈 명령어 목록 출력

```
- hosts: localhost
vars:
  ansible_doc: /usr/share/doc/ansible-core
  modules_all: "{{ (out.stdout|from_yaml).keys()|list }}"
  modules_str: |
    {% for i in modules_all %}
    {% set arr = i.split('.') %}
    - {collection: {{ arr[:~1]|join('.')|d('ansible.builtin', true) }}, module: {{ arr[-1] }}}
    {% endfor %}
  modules_grp: "{{ modules_str|from_yaml|groupby('collection') }}"
  collections_list: "{{ modules_grp|map('first') }}"
  modules_lists: "{{ modules_grp|map('last')|map('map', attribute='module') }}"
  collections: "{{ dict(collections_list|zip(modules_lists)) }}"
```


모듈 명령어 목록 출력

```
tasks:
  - command: "{{ ansible_doc }}" -t module -l -j"
    register: out
  - debug:
      var: out.stdout_lines
      when: false
  - debug:
      var: out.stdout
      when: false
  - debug:
      var: modules_all
      when: false
  - debug:
      var: modules_str
      when: false
```

모듈 명령어 목록 출력

```
- debug:
    var: collections.keys()|list|to_yaml
    when: true
- debug:
    var: collections['ansible.builtin']|to_yaml
    when: true
- debug:
    var: collections['community.digitalocean']|to_yaml
    when: true
- debug:
    var: collections['community.general']|to_yaml
    when: true
```

모듈 및 문서 위치

앤서블 모듈 설치 위치

1. /lib/python3.8/site-packages/ansible_collections
2. /usr/lib/python3.9/site-packages/ansible/plugins/
3. /usr/lib/python3.9/site-packages/ansible/modules/

앤서블 표준 컬렉션 모듈 목록

<https://docs.ansible.com/ansible/latest/collections/community/general/index.html#plugin-index>

앤서블 문서 위치

```
# man -k ansible
```

모듈

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

- name: copy an issue file to remote server

copy:

src: /etc/hostname

dest: /tmp/hostname.bak

remote_src: yes

모듈

패키지 설치시 사용하는 모듈. 패키지 관련 모듈은 운영체제 및 배포판 별로 다르게 제공한다. 일반적으로 레드햇 계열은 다음과 같은 패키지 관리자 제공한다.

dnf/yum

레드햇 계열 배포판에서 사용하는 네트워크 패키지 관리자. 패키지 설치 및 삭제 자동으로 의존성부분을 확인한다. 다만, 온라인 상태가 아닌 경우, 올바르게 동작하지 않는다.

zypper

수세에서 사용하는 패키지 매니저. 수세는 기본적으로 RPM를 사용하지만, 레드햇과 다른 네트워크 패키지 매니저를 사용한다.

RPM

단일 패키지 관리는 RPM명령어를 통해서 사용한다. 레드햇 계열은 RPM기반으로 관리가 되며, 패키지에 문제가 있을 때, 해당 명령어로 장애처리가 가능하다.

모듈

yum:

name: <package_name>

state: present

dnf:

name: <package_name>

state: present

zypper:

name: <package_name>

state: present

package:

name: <package_name>

state: present

모듈

앤서블에서는 저-수준 패키지 관리자 명령어인, rpm에 대해서는 직접적인 관리 모듈을 제공하지 않는다. 관리하기 위해서는 다음 명령어로 관리가 가능하다.

```
- name: remove the httpd RPM package  
  shell: rpm -e httpd
```

모듈

"file:", 디렉터리/파일/링크와 같은 자원을 생성하거나 혹은 소유권 및 퍼미션 변경 시 사용하는 명령어. 많은 옵션이 있기 때문에 간단하게 설명만 한다.

1. path: 변경이 될 디렉터리 및 파일 대상.
2. perm: 디렉터리 및 파일의 퍼미션.
3. owner: 파일 및 디렉터리의 소유자.
4. group: 파일 및 디렉터리의 그룹 소유자.
5. mode: 심볼릭 및 혹은 8진수 기반으로 퍼미션 설정.
6. state: 링크/디렉터리 혹은 비어 있는 파일(touch)를 생성하는 옵션.

모듈

```
- file:
    path: /tmp/test
    owner: foo
    group: foo
    mode: 0644
- ansible.builtin.file:
    path: /tmp/test
    owner: foo
    group: foo
    mode: 0644
```

모듈

'debug'모듈은 사용자 혹은 시스템 변수 출력이나 혹은 모듈의 특정 정보를 확인 시 많이 사용하는 모듈이다.

```
- debug:  
  msg: "{{ username }}"
```

```
- debug:  
  msg: "{{ ansible_facts['distribution'] }}"
```

```
- debug:  
  msg: "Hello Ansible World!"
```

모듈

관리가 되고 있는 노드에 직접 명령어를 실행한다. 앤서블에서 두 개의 모듈을 지원하는데, 기능은 비슷하지만, 동작 방식이 조금 다르다.

shell

사용자가 사용하는 셸 기반으로 명령어를 실행한다. 기존에 사용하던 셸 변수를 그대로 사용이 가능하기 때문에 스크립트 실행 시, 많이 사용한다.

command

간단하게 명령어를 실행 시 사용한다. 위의 모듈 shell과 다르게 셸 환경 변수를 사용하지 않기 때문에, 간단한 명령어 실행 시 사용한다.

모듈

현재 리눅스는 2가지 초기화 관리자(initiator program)를 가지고 있다. 이전의 System V의 스크립트 기반, 두 번째는 systemd기반으로 관리 및 운영하는 서비스 관리자.

1. /bin/init
2. /bin/systemd

최근 리눅스는 systemd기반으로 사용하고 있기 때문에, systemd모듈을 사용한다. 단, 이전 시스템과 호환성을 유지하기 위해서 service모듈을 권장한다.

```
- systemd:
  name: httpd.service
  state: restarted
- service:
  name: httpd
  state: restarted
```

모듈

사용자 추가 모듈이다. 관리자 계정 이외, 사용자를 추가하기 위해서는 이 모듈로 사용자를 추가한다. 배포판 상관없이 사용이 가능하며, 다만 윈도우 서버 경우에는 모듈이 다르다.

```
- user:
  name: testuser1
  shell: /bin/bash
  state: present
  uid: 11000
  group: 11000
  groups: adm, wheel
  append: true
  password: <ENCRYPTED_PASSWORD>
  password: '$6$i6NhgY1fDqM5bIUy$C0xENqViz'      ## dnf install mkpasswd
```

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
```

연습문제

설치한 앤서블에서 간단하게 다음 모듈을 실행해본다. 테스트 용도로 만든 인벤토리 파일을 사용해서 작업을 수행한다.

1. 'file'모듈을 사용하여 `"/tmp/test"`라는 파일을 생성한다.
 - `"path"` 옵션을 사용한다.
 - 사용자 및 그룹을 `"ansible"`으로 변경한다.
2. 사용자 `"test1"`, `"test2"`를 생성한다.
 - 암호는 `test`으로 통일한다.
 - 그룹은 `adm`, `wheel`에 가입한다.
3. 패키지를 설치한다.
 - `httpd`, `vsftpd`패키지를 설치한다.
 - `squid`패키지를 설치한다.
4. 서비스를 동작하도록 구성한다.
 - `httpd/vsftpd/squid` 서비스를 시작한다.
5. 설치된 패키지 및 사용자 제거한다.
 - 위에서 설치한 패키지를 전부 제거한다.
 - 위에서 생성한 사용자를 전부 제거한다.
6. `'ansible-doc'`명령어를 통해서 `'dnf'`명령어에 대한 도움말을 확인한다.
 - 도움말을 확인하여 간단하게 `"httpd"`패키지를 설치한다.

ad-hoc

단일 실행

ad-hoc

앤서블은 YAML형태 말고 애드 혹(ad-hoc)방식이 있다. 이 방식은 마치 셸 스크립트 실행하는 방식과 비슷하게 모듈+인자 형태로 구성이 되어 있다.

아래는 간단한 ad-hoc사용 방식이다.

```
$ ansible -i <INVENTORY> -m <module> -a "arg1=<value> arg2=<value>  
node1.example.com,
```

애드혹은 셸 스크립트에서 같이 사용하거나 혹은 몇몇 셸 스크립트 기능을 표준화 모듈 기반으로 사용하기 위해서 사용한다.

ad-hoc

간단하게 테스트나 혹은 사용시에 보통 다음과 같이 명령줄에서 실행한다.

```
$ ansible -i hosts -m copy -a "src=/etc/hosts dest=/tmp" node1.example.com  
$ ansible -m ansible.builtin.debug -a "msg=hello" localhost  
$ ansible -m setup localhost,
```

ad-hoc shell script

```
$ vi adhoc_replace_shell_functions.sh
#!/bin/bash
if $(ansible localhost, -m ping)
then
    ansible localhost, -m copy -a "dest=/var/www/html/index.html,
content='This is World'"
else
    ansible localhost, -m copy -a "dest=/root/README.md, content='This file
is wrong'"
fi
```

ADHOC 연습문제

파일명은 'adhoc-exam.sh'으로 정한다. "node4.example.com" 대상으로 아래 작업을 수행한다.

1. "Hello World"라는 "default.html"파일을 "/var/www/html/"에 생성.
 1. 'copy:'모듈에서 'content'를 사용해서 위의 내용 생성.
2. 웹 서버 패키지가 설치가 안되어 있으면 "yum"모듈을 사용해서 설치.
3. 방화벽에 "http", "https"서비스가 등록이 안되어 있으면 "http", "https" 서비스에 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.

앤서블 인벤토리

추가 내용

앤서블 인벤토리

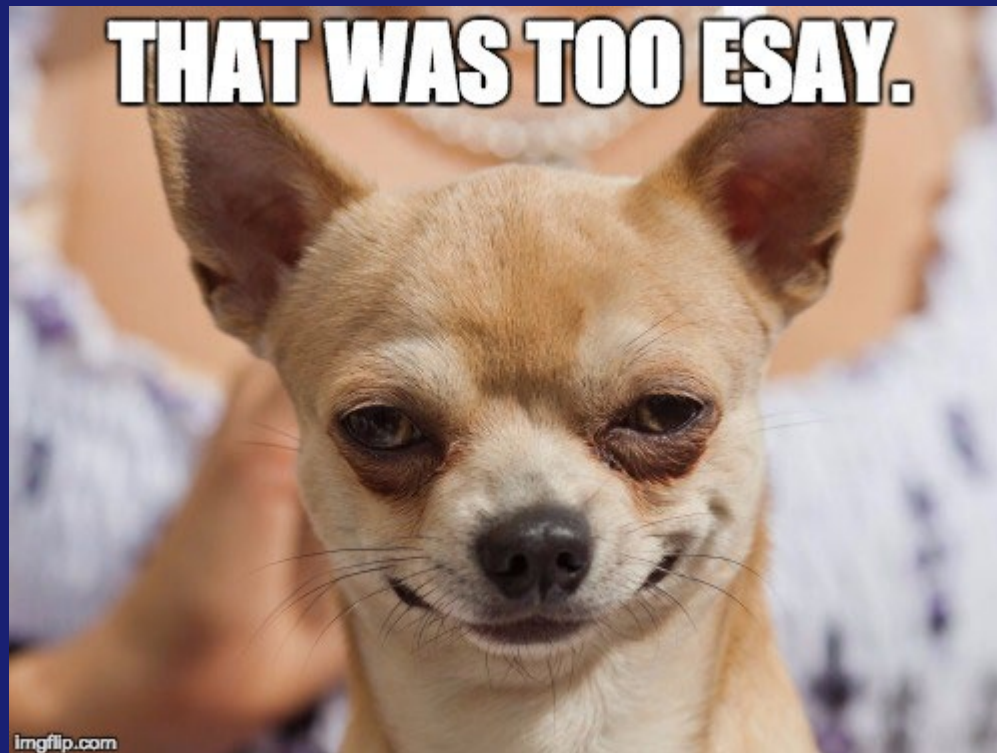
앞에서 간단하게 이야기 하였지만, 앤서블 인벤토리는 총 3가지 형식으로 지원한다.

1. TOML: Tom's Obvious Markup Language
2. INI: Initialization
3. YAML: YAML Ain't Markup Language

현재 최신 앤서블 코어 버전은 TOML으로 구성을 권장하고 있다. 다만, 버전에 따라서 다르며 앤서블 2.8부터 지원한다. 그 이전 버전들은 "YAML", "INI"기반으로 권장한다.

단, 'ansible-inventory' 프로그램은 TOML을 아직 지원하지 않는다.

왜 하필 INI?



Inventory changed

현재 사용하는 인벤토리는 "INI"형식을 사용하고 있다. 레드햇 및 앤서블 커뮤니티에서는 현재 "YAML"형태의 인벤토리를 사용을 권장하고 있다.

https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html

위의 부분은 변경이 되었다. 앤서블은 다음과 같은 순서로 인벤토리 파일을 확인한다. "inventory" 플러그인 설정이 안되어 있는 경우, 기본 값은 "auto"이며, 나머지는 아래와 같은 순서대로 파싱을 시도한다.

1. host_list
2. script
3. auto
4. yaml
5. ini
6. toml

INI

```
mail.example.com
[webservers]
foo.example.com
bar.example.com
[webservers:vars]
http_port=8080
http_name=www.example.com
[dbservers]
one.example.com
two.example.com
three.example.com
```

YAML

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
      vars:
        http_port: 8080
        http_name: www.example.com
```

TOML

```
[ungrouped.hosts]
```

```
mail.example.com
```

```
[webserver.hosts]
```

```
foo.example.com
```

```
bar.example.com
```

```
[webserver.vars]
```

```
http_port = 8080
```

```
http_name = www.example.com
```

인벤토리 분류

앤서블 인벤토리는 다음과 같은 형식을 가지고 있다. 그룹 이름은 **대괄호**로 구별 및 분류한다.

```
[그룹 이름]
```

```
<호스트>
```

```
[그룹 이름:children]
```

```
<그룹이름>
```

인벤토리 와일드 카드

인벤토리에는 와일드 카드 방식이 사용이 가능하다.

```
192.168.10.*
```

```
192.168.[10-20].[1-100]
```

```
*.example.com
```

이와 같은 방식으로 와일드 카드 선언이 가능하다.

인벤토리 변수

[test]

node1.example.com package=httpd

[db]

10.10.5.2 package=mariadb

[was]

was1.example.com package=wildfly

인벤토리 변수

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

인벤토리 오버라이드 변수

인벤토리에서 몇몇 앤서블 변수는 오버라이드가 가능하다. 아래는 특정 서버에 접근 시, 사용하는 리모트 유저 정보를 변경한다.

```
[test]
```

```
node1.example.com package=httpd ansible_user=slack ansible_port=8899
```

```
[db]
```

```
10.10.5.2 package=mariadb
```

```
[was]
```

```
was1.example.com package=wildfly
```

인벤토리 정보

인벤토리는 위의 내용을 기준으로 다음과 같은 내용을 가지고 있다.

- 호스트 이름
- 아이피 주소
- 호스트에서 사용하는 변수
- 앤서블 오버라이드 변수

인벤토리 옵션

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.

```
$ cat ansible.cfg
[defaults]
inventory = <INVENTORY_FILE>
host_key_checking = False
remote_user = <SSH_USER>
remote_port = <SSH_PORT>
[privilege_escalation]
become=true
```

연습문제

ansible.cfg에는 다음과 같은 내용이 구성이 되어야 한다. 디렉터리를 "inventory_lab"으로 생성 후 진행한다.

1. inventory_cloud라는 파일을 기본 인벤토리 파일로 불러와야 한다.
2. ssh 접근 포트는 8822번을 사용한다.
 - 위의 설정은 노드 2/3번에 각각한다.
 - /etc/sshd/sshd_config.d/
 - 02-port.conf
3. 접근 시 사용하는 사용자는 ansible, 사용자의 비밀번호는 ansible이다.
 - 리모트 접근 사용자는 ansible로 한다.
 - ansible.cfg
 - 각 노드에는 해당 사용자가 구성이 되어 있어야 됨.

연습문제

4. sudo접근이 가능하도록 wheel그룹 설정을 각 노드에 올바르게 한다.
 - `/etc/sudoers.d/nopasswd`
 - `grep -I wheel /etc/sudoers`
5. inventory_cloud라는 설정 파일에는 다음과 같은 서버를 등록한다. 등록되는 그룹의 이름은 noexist_server으로 명시한다.
 - 10.10.10.1
 - 10.20.30.5
 - 192.168.90.*

연습문제

"inventory_cloud"에는 다음과 같이 설정 및 구성한다.

1. web, db라는 두개의 그룹이 존재하고 각각 그룹에 서버 하나씩 할당한다.
2. web에는 web.lab.example.com, db에는 db.lab.example.com 할당
3. 할당된 노드에는 nodename이라는 이름의 변수에 web.example.com, db.example.com라는 호스트 이름을 할당
4. together라는 그룹에 web, db가 동시에 사용이 가능하도록 한다

연습문제

```
# vi ansible.cfg
[defaults]
inventory = inventory_cloud
ansible_ssh_port = 8877
ansible_ssh_user = ansible
ansible_ssh_pass = ansible
```

연습문제

```
# vi inventory_cloud  
[noexist_group]  
10.10.10.1  
10.20.30.5  
192.168.90.*
```

연습문제

```
# vi inventory_cloud
[web]
web.lab.example.com
[db]
db.lab.example.com
[together:children]
web
db
# ansible -i inventory_cloud --list-hosts
```


앤서블 태스크

tasks(pre_tasks, post_tasks)

role

delegate_to

작업

앤서블에서 사용하는 작업 관리 방법에 대해서 전체적으로 알아본다. 크게 두 가지 형태로 작업을 구별 및 구분하여 사용/분리가 가능하다.

tasks

앤서블 작업이 수행이 되는 영역. 모든 작업은 Top-Down형태로 수행이 된다. 기본작업은 모듈기반으로 각각 작업이 수행된다.

roles

역할은 작업의 독립적인 형태이며, 오브젝트 개념으로 보면, 각각 함수(function)들이 모인, 클래스(class)와 같은 개념적인 역할을 구성한다. 이 부분은 뒤에서 더 다루도록 한다.

작업/역할

Playbook

tasks

module:

args1: value
args2: value

pre/main/post

앤서블 태스크는 기본적으로 `tasks` 기본으로 동작한다. 하지만, 여러가지 이유로 전/후 작업 처리가 필요한 경우 `pre_tasks`, `task`, `post_tasks` 총3개의 작업으로 나뉘어진다.

`pre_tasks`:

본 작업(tasks)가 실행 되기전에 특정 작업을 수행한다.

`tasks`:

본 작업이 수행이 되는 영역. 일반적으로 제일 많이 사용하는 작업 지시자다.

`post_tasks`:

본 작업이 완료 후, 후속 작업이 필요한 경우 사용한다.

pre/main/post

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

pre_tasks:

- name: updates all of packages

yum:

name: "*"

state: latest

pre/main/post

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

tasks:

- name: install a vsftpd package

yum:

name: vsftpd

state: present

pre/main/post

task작업이 수행이 된 다음에 후 처리 작업을 보통 "post_tasks"에서 수행한다. 역시 "post_tasks"만 단독으로 사용하는 경우는 거의 없다.

```
post_tasks:  
  - name: start and enable the vsftpd service  
    service:  
      name: vsftpd  
      state: started  
      enabled: true
```

pre/main/post

```
pre_tasks:  
  - name: updates all of packages
```

```
    yum:
```

```
      name: *
```

```
      state: present
```

```
tasks:
```

```
  - name: install a vsftpd package
```

```
    yum:
```

```
      name: vsftpd
```

```
      state: present
```

첫번째 작업 수행

주 작업 수행

pre/main/post

```
post_tasks:
- name: start and enable the vsftpd service
  service:
    name: vsftpd
    state: started
    enabled: yes
```

후속 작업 수행

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

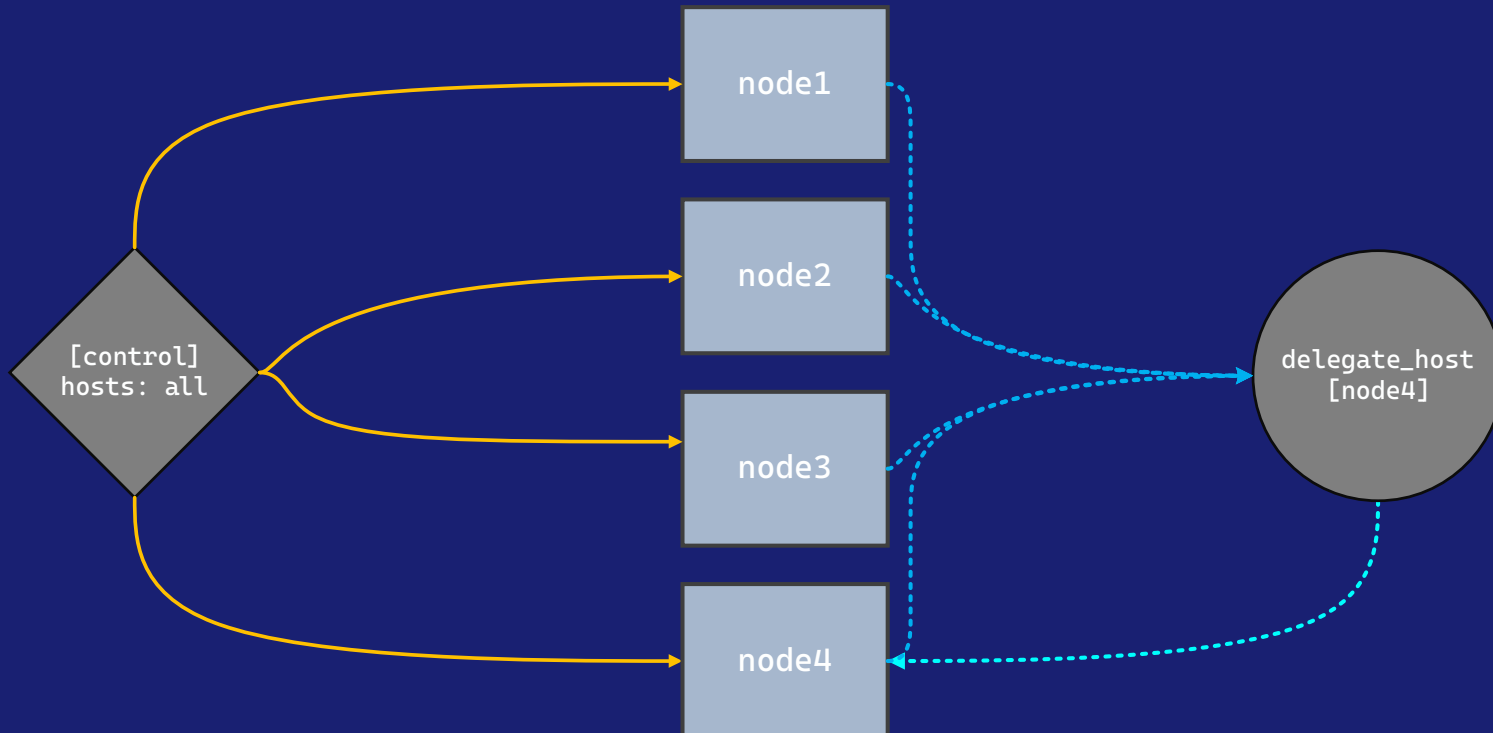
tasks가 role로 변경.
한 개 이상의 role 조합으로
사용이 가능하다.

pre/main/post + tasks

```
- hosts: web
  pre_tasks:
    - name: this is the hello message from ansible
      debug:
        msg: "This message shows from pre tasks"
  roles:
    - { role: vsftpd }
  tasks:
    - debug:
        msg: "This is Tasks session. This part will run after run to 'roles'."
  post_tasks:
    - name: this is the by message from ansible
      debug:
        msg: "This message shows from post tasks"
```

delegate_to

작업 수행 시, "hosts:"에 명시된 서버 그룹 혹은 서버에 직접 접근이 아니라, 한번 더 다른 서버에 걸쳐서 작업을 수행한다. 이를 "delegate"라고 하며 호출 시 사용하는 지시자는 'delegate_to'를 사용한다.



delegate_to

- hosts:
 - remote1.example.com
 - remote2.example.com

tasks:

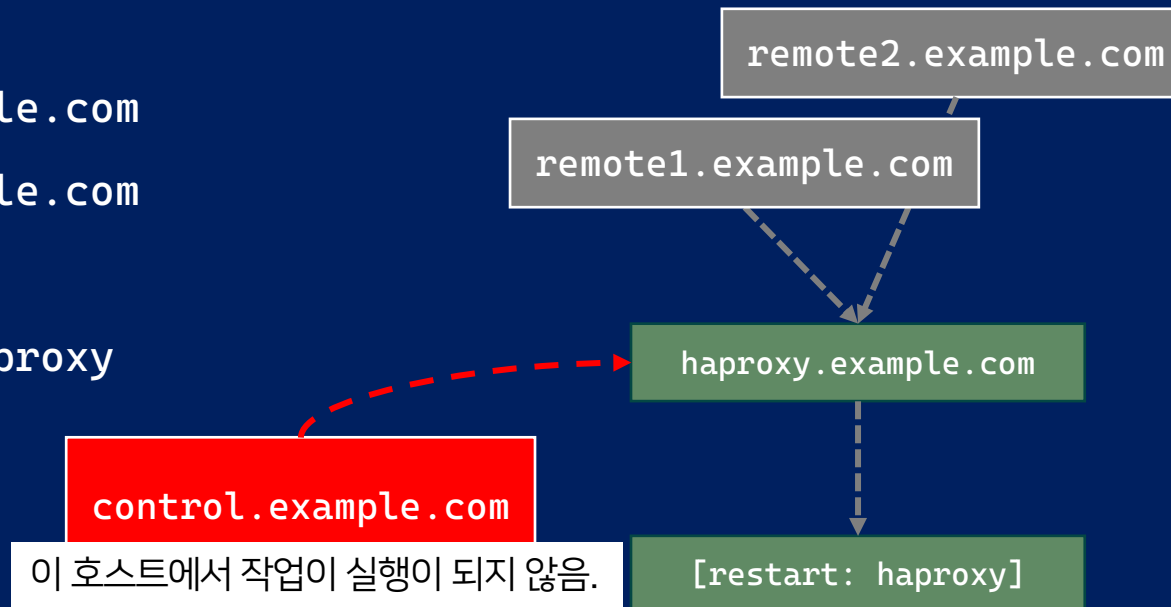
- name: restart haproxy

service:

name: haproxy

state: restart

delegate_to: haproxy.example.com



연습문제

앤서블 작업을 다음 조건으로 구성한다.

pre, post 및 tasks를 사용하여, 전후 작업을 구성한다.

1. 선 작업은 "we are going to install the httpd package"메시지 출력.
 1. /var/www/html/index.html
 2. 만약, 위의 디렉터리 및 파일이 없는 경우, 'file'로 생성 및 구성.
2. 주 작업에서는 아파치 패키지를 최신 패키지로 설치.
3. 후 작업에서는 "the package has been installed"라는 메시지를 출력.
 1. debug, msg.

모든 노드의 호스트 이름 설정한다.

1. node[1-4].example.com으로 올바르게 호스트 이름(FQDN)를 설정한다.
 - 'hostname'모듈을 사용해서 모든 서버에 적절하게 FQDN으로 도메인 및 호스트 네임 구성한다.

대행서버(delegate)서버는 node1가 대행, 다음과 같은 작업을 수행한다.

1. 대행은 반드시 node1가 한다.
2. node1에서 다른 노드에 문제 없이 접근이 가능해야 한다.
3. 각 노드에 설치가 된 squid패키지를 설치한다.

앤서블 변수

변수선언 및 구성

VAR

앤서블도 다른 프로그램 언어처럼 변수를 사용한다. 다만, 우리가 알고 있는 변수와는 조금 다른 방식으로 사용하며, 지원하는 변수 범위가 폭 넓게 지원을 하고 있다. 그래서 맨 처음에 사용하면, 변수 때문에 많이 어려워하는 부분이 많다.

아래는 앤서블에서 지원하는 변수 범위이다.

1. inventory
2. group_vars
3. host_vars
4. role vars
5. 앤서블 시스템 변수(ansible_facts)

VAR IN INVENTORY

Inventory var는 다음처럼 보통 선언한다.

```
[server]  
test.lab.example.com var1=hello var2=world
```

"var1"은 변수 이름이며, "="변수 대입 연산자 "hello"는 영문 소문자로 변수 값 할당.

앤서블에서 사용하는 모든 변수는 **public type**으로 구성이 되어 있다. 앤서블은 **private**변수는 없으며 앤서블 런타임 동작이 완료가 되면 변수는 그 즉시 **heap/stack**메모리에서 제거가 된다.

일반변수

일반 변수는 보통 플레이북이 다음처럼 작성 및 구성한다.

```
---
- name: this is first vars
  hosts: localhost
  vars:
    division: middleware
  tasks:
    - name: Hello world
      command: echo "Hello world"
    - debug:
        msg: "{{ division }}"
```

리스트

앤서블 리스트는 매우 간단하게 선언한다. 리스트의 이름 그리고, 리스트가 가져가는 값을 선언한다.

```
---
- name: this is first dict
  hosts: localhost
  vars:
    fruits_list:
      - apple
      - mango
      - pineapple
  tasks:
    - name: shows your name
      command: echo "{{ dict.firstname }}" "{{ dict.lastname }}" "{{ dict['gender'] }}"
```

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 리스트와 비슷하지만, 키쌍(keypair)형태로 구성이 되어 있다. 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'] }}"
```

그룹변수

설명

GROUP VARS

앤서블 변수에서 제일 많이 사용하는 변수는 "group_vars", "host_vars" 두 개의 변수 디렉터리 프레임 워크가 있다. 이들은 두 가지 형태로 구현 할 수 있다.

1. 파일

2. 디렉터리 + 파일

보통은 2번 "디렉터리 + 파일" 형태를 많이 사용한다.

GROUP VARS

"group_vars"는 다음처럼 디렉터리나 혹은 파일을 생성 후 구성이 가능하다.

```
$ mkdir group_vars
$ mkdir server
$ touch server
$ cd server
$ vim hostname
node1: node1.example.com
```


그룹 및 호스트 변수

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_VARS

디렉터리 혹은 파일은 아래와 같은 조건으로 생성을 고려해야 한다.

1. 그룹변수는 아래와 같이 디렉터리나 혹은 파일 형태로 생성할 수 있다.
2. 단순한 변수 선언은 파일, 복잡하고 많은 양의 변수 선언은 디렉터리 기반으로 구성한다.

`example_group_vars_dir/`

그룹이름으로 디렉터리를 만든 다음에, 임의의 파일 이름으로 변수 선언.

`example_group_vars_file`

디렉터리를 만들지 않고, 해당 그룹 이름으로 파일을 생성 후, 변수 선언.

GROUP_VARS

아래와 같이 생성한다.

```
$ mkdir group_vars
$ vim inventory
[node1]
192.168.90.13 nodename=node1.example.com

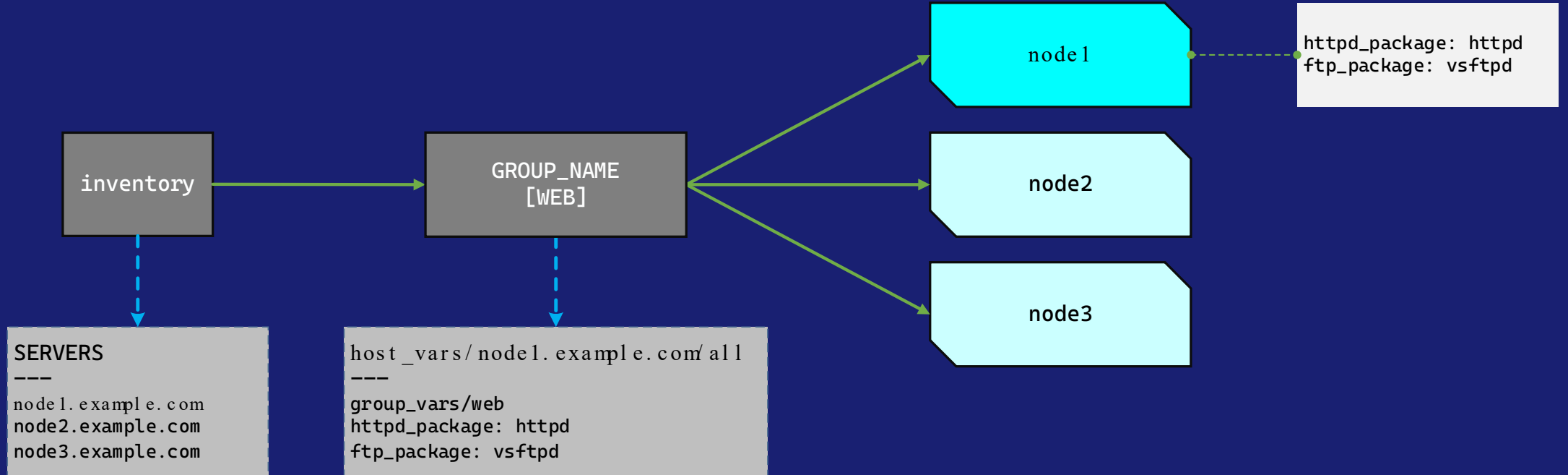
$ mkdir -p group_vars/node1
```

GROUP_VARS

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

```
$ vim all
httpd_package: httpd
ftp_package: vsftpd
```

GROUP_VARS



호스트 변수

설명

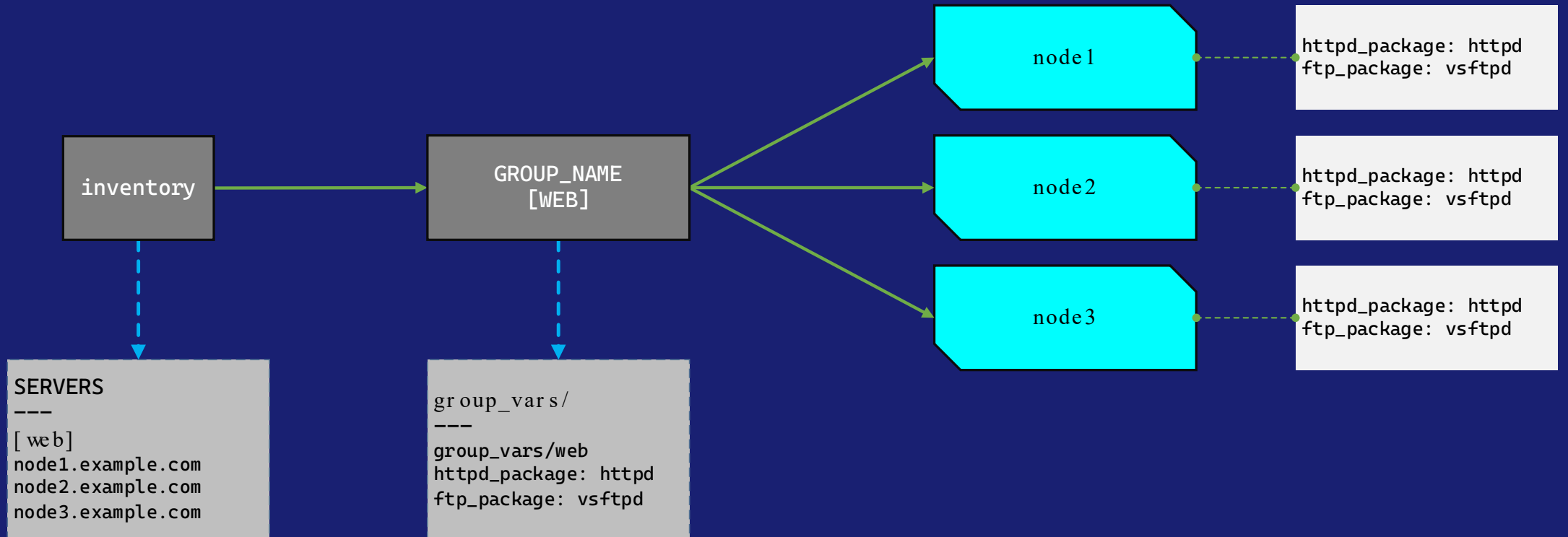
HOST VARS

두 가지 형태로 구현 할 수 있다.

1. 파일
2. 디렉터리 + 파일

보통은 2번 "디렉터리 + 파일" 형태를 많이 사용한다. 혹은, 인벤토리를 통해서 특정 호스트에 대해서 변수 선언이 가능하다.

HOST VARS



HOST VARS

"host_vars"는 다음처럼 구성하고 선언한다.

```
$ mkdir host_vars
$ mkdir db1.example.com
$ touch db1.example.com
$ cd db1.example.com
$ vim db1.example.com
db1: db1.example.com
```

HOST_VARS

192.168.90.13에 서버만 "echoecho"라는 변수에 "This is node1.example.com"이라는 변수를 할당하고 싶으면 다음처럼 작성한다.

```
$ mkdir -p inventory/host_vars/192.168.90.13/  
$ vim inventory/host_vars/192.168.90.13/all  
echoecho: "This is node1.example.com"
```

HOST_VARS

혹은 인벤토리에 다음처럼 선언 및 구성하기도 한다.

```
$ mkdir host_vars
$ vim inventory
[node1]
192.168.90.13 nodename=node1.example.com
$ mkdir -p host_vars/192.168.90.13
$ touch host_vars/192.168.90.13
```

연습문제

"practice_groupvars/"라는 디렉터리 생성 후, 아래에 다음과 같이 구성한다. 그룹 이름은 "middleware", "openstack", "kubernetes" 그리고 "storage"으로 구성한다.

1. middleware그룹에는 middleware라는 파일에 software: tomcat 선언
2. openstack그룹에는 version라는 파일에 release: rocky 선언
3. kubernetes그룹에는 release라는 파일에 version: 1.23.00 선언
4. storage그룹에는 nfs라는 파일에는 version: 4.2, ceph라는 파일에는 version: 5.0으로 선언

앤서블 시스템 변수

키워드 변수

시스템 변수

앤서블 시스템 변수는 여러가지가 있다. 시스템 변수의 대다수는 또한, 키워드 명령어와 관련이 있다. 이러한 이유로 어떠한 부분은 앤서블 설정 파일에서 변경이 되며, 혹은 어떠한 변수는 YAML에서 변수 형태로 값이 변경이 된다.

모든 시스템 변수는 별도로 값을 설정하지 않으면, 하드코드 수준에서 기본값이 정해져 있기 때문에, 기본 코드 값으로 시스템 변수는 동작하게 된다.

https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html

hosts

hosts는 앤서블 키워드이며, 싱글 플레이북 혹은 멀티 플레이 북에서 하나 혹은 여러 개 사용이 가능하다. hosts에는 하나 이상의 값 할당이 가능하다.

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

remote_user

'remote_user'는 앤서블에서 관리 노드에 접근 시에, 사용하는 계정이다. 명령어로 비교하면 다음과 같다.

```
- name: this is the first playbook
  hosts: remote.example.com
  remote_user: ansible
  become: true
```

```
$ ssh ansible@remote.example.com
```


ansible_connection

보통 인벤토리에 선언하는 시스템 변수(키워드)이다. 인벤토리에 선언된 이름과 실제로 접속이 되는 서버 이름이 다른 경우 아래와 같이 선언한다. hostname, IP Address상관 없다.

```
inventory_file
```

```
---
```

```
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. "ansible_user" -> "remote_user"를 대신한다. 동작이 되는 영역은 인벤토리 파일에서만 동작한다.

```
inventory
```

```
---
```

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

ansible_port

앤서블 리모트 서버에 접근 시 사용하는 포트 번호가 다른 경우 다음처럼 임시로 포트번호 변경이 가능하다. 앤서블 설정 파일에서는 "remote_port"라는 옵션을 통해서 변경이 가능하다.

```
inventory
```

```
---
```

```
192.168.90.11 ansible_port=8080
```

```
ansible.cfg
```

```
---
```

```
[defaults]
```

```
remote_port=8080
```

ansible_ssh_pass

앤서블 서버에 접근 시, ssh 공개키가 아닌 비밀번호로 접근해야 되는 경우 아래와 같은 옵션으로 접근이 가능하다. 비밀번호는 가급적이면 변수, 그리고 앤서블 볼트를 통해서 암호화를 권장한다.

```
inventory
```

```
---
```

```
192.168.90.11 ansible_ssh_pass=centos
```

ansible_facts

이러한 앤서블 매직 변수는 다음 소스코드에서 확인이 가능하다. 긴 내용이기 때문에 여유 시간이 있을 때, 전체적인 키워드 명령어를 한번 훑어본다.

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

앤서블에서 제일 활발하게 호출하여 사용하는 변수는 따로 있다. 바로 "ansible_facts"라는 앤서블 내부 변수이다. 이 변수는 "ansible.builtin.setup"모듈을 통해서 호스트의 정보를 플레이북 시작 전에 수집 후, 앤서블 데이터를 컴파일 한다. 버전에 따라서 다르지만, "ansible.builtin.gather_facts"모듈을 통해서도 수집이 가능하다.

모든 데이터는 "리스트", "딕셔너리"형태로 관리 노드에 전달이 되며, 이 정보 기반으로 데이터를 생성 및 컴파일을 진행한다.

ansible_facts

사용하는 방법은 두 가지가 있다.

첫 번째 변수 호출 방법은 JSON형태로 부르며, 두번째 호출 방법은 이전 앤서블 템플릿 변수 호출 방식이다.

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

두번째 호출 방법은 아래처럼 속성 접근이 가능하다.

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

ansible_facts

변수를 사용하기전에 테스트 및 필요한 도구. 앤서블 네비게이터를 사용하는 경우, 별도로 아래 패키지를 사용하지 않아도 된다. 아래는 "setup" 모듈을 통해서 시스템 정보를 수집 및 확인한다.

```
$ ansible localhost, -m setup  
$ yum install python3-jmespath
```

특정한 팩트 정보를 수집하기 위해서, 아래와 같이 필터를 걸어서 확인이 가능하다.

```
$ ansible localhost -m setup -a 'filter=ansible_dist*'
```

앤서블 팩트 예제(NIC/BLOCK)

```
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. 이전에 이 부분을 해결하기 위해서 "~/.ssh/config"를 통해서 해결 하였다.

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

위의 설정을 넣어주면 "Fingerprint 경고"를 무시하고 진행한다. 보통 오류가 발생하면, 아래와 같이 화면에 출력이 된다.

```
{"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, 아이디 및 비밀번호를 가지고 로그인에 필요한 경우, 다음처럼 "ansible.cfg"에 구성이 가능하다.

```
[defaults]
ansible_user = root
ansible_pass = centos
remote_user = root
```

ansible_ssh_{{ user, password }}

"ansible_ssh_*"키워드는 사용은 가능하나, 다음 키워드로 대체하여 사용하는 것을 권장한다. 버전이 업그레이드 되면서 많은 키워드 명령어는 제거 혹은 변경이 되기 때문에, 릴리즈 시, 해당 부분을 꼭 확인해야 한다.

```
ansible_ssh_user → ansible_user  
ansible_ssh_host → ansible_host  
ansible_ssh_port → ansible_port
```

EXTRA-VARS

확장변수는 "--extra-vars"라는 옵션이 있다. 이 옵션을 통해서 기존에 구성된 변수 값 변경이 가능하다.

vars:

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

```
package_name: httpd
```

EXTRA-VARS

아래 명령어를 변수로 표현하면 아래와 같다.

```
$ ansible-playbook -e config_dir="/etc/vsftpd/" -e package_name=vsftpd  
main.yaml  
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. inventory file or script group vars
3. role defaults (defined in `role/defaults/main.yml`)
4. inventory group_vars/all
5. inventory group_vars/*
6. playbook group_vars/*
7. inventory file or script host vars
8. inventory host_vars/*
9. playbook host_vars/*

변수 우선순위에 대한 자세한 정보는 아래의 링크에서 확인이 가능하다.

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

연습문제

앞에서 학습한 내용을 가지고 간단하게 앤서블 기반으로 플레이북 만들어 본다.

1. ansible adhoc 기반으로 node1에서 node2로 파일을 보낸다.

- copy모듈로 hello.txt라는 파일을 생성하여 node2의 /tmp/hello.txt에 저장한다.
- 메시지 내용은 "Hello World"
- ansible inventory 파일 생성한 후 node2에 파일을 보낸다.
- inventory에는 호스트 그룹은 web, db로 구성한다.

2. web에는 control.example.com db는 manage.example.com 호스트 이름 사용. 실제 접근 시, node3/4로 접근하게 한다.

- copy.yaml를 만들어서 /tmp/issue파일을 원격 서버의 /etc/issue에 보낸다.
- /tmp/issue파일에는 "Hello an ansible remote"라는 메시지가 포함이 되어 있어야 한다.

연습문제

확장 명령어

EXTEND KEYWORD

import/include

IMPORT

이 키워드는 "tasks"이외 "role", "playbook"와 같은 추가적인 기능들이 더 있다. "import" 키워드는 정적으로 동작한다. 즉, 실행 시 모든 내용을 메모리에 불러온 후, 플레이북 작업을 수행한다. 동작 방식은 정적으로 동작한다.

INCLUDE

"include_" 키워드는 "tasks", "role"를 지원하면 "playbook"영역은 지원하지 않는다. "include", "import"의 다른 부분은 "include"는 해당 지시자를 만나면, 그 순간 파일을 메모리에 불러오며, 동적으로 동작한다.

아래는 앤서블에서 지원하는 정적/동적 파일 불러오기 기능이다. 보통 이를 통해서 파일 기능 분류를 하기도 한다.

1. import_tasks
2. import_playbook
3. include_tasks
4. include_role

"include_playbook"은 더 이상 지원하지 않는다. <https://github.com/ansible/ansible/issues/76684>

include_tasks

"include_" 문법은 위의 "import_"와 기능적으로는 동일하다.

다만, "include_"는 동적으로 파일을 불러오며, 앤서블 인터프리터가 파싱 하면서 "include_"문법을 만나면, 해당 파일을 메모리에 불러온다.

키워드 변수

앤서블에는 특수한 변수가 존재한다. 보통 **매직변수**라고 부른다. 위의 예제에서 다음과 같은 매직변수를 사용하였다.

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

일반 팩트

앤서블 팩트(Fact)는 앞에서 다루었던, 앤서블 시스템 변수 혹은 사용자 정의 변수이다. 이 내용은 이미 앞서 "변수"에서 다루었다.

앤서블 시스템 변수는 앤서블 인벤토리에 등록된 서버에 접근하여 각각 서버의 정보를 setup모듈을 통해서 수집한다. 사용자가 사용자화를 통해서 별도로 설정 및 구성이 가능하다.

앤서블 명령어로 확인이 가능하다.

```
# ansible localhost, -m ansible.builtin.setup
```

확장 팩트를 생성하기 위해서 아래 슬라이드와 같이 생성한다.

확장 팩트

팩트는 사용자가 별도로 시스템에 값을 추가적으로 설정이 가능하다. 보통 다음과 같이 디렉터리를 생성 후, 조회한 `/etc/ansible/facts.d/`

예를 들어서 소유권 및 용도에 대한 팩트 추가가 필요한 경우, 아래처럼 팩트를 작성한다.

```
[deployment]
mgmt_name=choigookhyun
mgmt_type=k8s_master
mgmt_division=infra
```

위의 정보 기반으로 검색 시, 다음과 같은 명령어로 확인이 가능하다.

```
# ansible <hostname> -m ansible.builtin.setup -a "filter=ansible_local"
```

오래된 키워드 변수

앤서블 사이트에 보면 종종 오랫동안 사용한 키워드 변수 혹은 명령어를 변경 혹은 제거를 한다. 한번씩 꼭 확인한다.

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.

조건문/레지스터

condition

register

debug and debugging

조건문

when

when

'when'이라는 조건문을 사용한다. 'when'은 "module", "roles"에서 함께 많이 사용한다. 그 이외 조건에서 사용이 가능하면 언제든지 붙여서 사용이 가능하다.

'when'명령어는 일반 언어에서 'if'명령어와 동일하다. 다만, 'if'명령어처럼 복잡하게 조건식이 들어가지 않으며, 기본적으로 반환 값(return count)가 참(true)이면 동작한다. 다만, 특정 상황에서는 "when"명령어가 전부 적용되지 않는다.

아래는 쉘 스크립트와 비교한 내용이다.

```
bash
$ rc=true
if [[ $rc -eq 0 ]] ; then
    echo "run a module"
fi
```

```
ansible
- name: check to the httpd package installed
  shell: rpm -qa httpd
  register: res_httpd_pkg
- name: remove a httpd package
  package:
    name: httpd
    state: absent
  when: res_httpd_pkg
```

when

간단하게 'when'명령어를 사용하기 위해서 아래와 같이 간단하게 작성한다.

```
- hosts: node1.example.com
  vars:
    httpd_conf_update: true
  tasks:
  copy:
    src: httpd.conf
    dest: /etc/httpd/conf/httpd.conf
  when: httpd_conf_update
```

when

```
vars_file:
```

```
- vars.yaml
```

외부 변수파일을 불러옴

```
$ vi vars.yaml
```

```
httpd_conf_update: yes
```

```
tasks:
```

```
copy:
```

```
src: httpd.conf
```

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

```
when: httpd_conf_update == "yes"
```

"httpd_conf_update"의 변수가 "yes"로 되어 있으면 참으로 판단하여 수행이 됨.

when

"ansible_facts"를 그냥 네이티브로 사용하기도 하는데 보통은 when을 통해서 조건 비교를 많이 한다.

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

위의 조건을 아래처럼 적용이 가능하다.

```
$ 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"
```


버전 조건문

앤서블 2.11이후부터, 버전 형식에 대한 조건문이 추가가 되었다. 버전 비교 시 사용한다.

```
installedVersion is version(softwareVersion, '<=')
```

버전

loose

loose 버전은 버전 레이블링 형식이 우리가 아는 상식(?)내에서 자유롭게 보통 표현을 한다. 보통 형식이 없이 작성하는 경우 loose를 사용한다. 버전은 보통 아래와 같은 형식으로 작성한다.

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/Patch형태로 구성하고 숫자 앞에 0이 붙으면 안된다. 리눅스 커널이 이 방식을 따른다.

V1.1

V1.2

V3

V2.4

V2.4.0

다만, 아래 링크를 참고한다. 넘버링 방식은 계속 변경이 되기 때문에, 항상 넘버링 정책을 확인해야 한다.

https://www.reddit.com/r/linux4noobs/comments/sr5fn8/what_do_linux_kernel_version_numbers_mean_nowadays/?rdt=46166&onetap_auto=true&one_tap=true

일반 조건문

일반 조건문은 다음처럼 지원한다.

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

위의 조건문을 적용하면 아래처럼 사용이 가능하다.

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

조건문 확인

```
- name: version test examples
```

```
  assert:
```

```
    that:
```

```
      - "'1.0' is version_compare('1.0', '==')" # old name
      - "'1.0' is version('1.0', '==')"
      - "'1.0' is version('2.0', '!=')"
      - "'1.0' is version('2.0', '<')"
      - "'2.0' is version('1.0', '>')"
      - "'1.0' is version('1.0', '<=')"
      - "'1.0' is version('1.0', '>=')"
      - "'1.0' is version_compare('1.0', '==', strict=true)" # old name
      - "'1.0' is version('1.0', '==', strict=true)"
```

조건문 모듈

특정 변수에 따라서 메시지 혹은 참 조건을 만들기 위해서 위에서 사용한 'assert'라는 명령어 사용이 가능하다. 'that'이라는 옵션을 사용하면, 'if'문법에서 조건을 'and'연산자로 처리하는 것과 비슷하게 동작한다.

```
- ansible.builtin.assert: { that: "ansible_os_family != 'RedHat'" }
- ansible.builtin.assert:
  that:
    - "'foo' in some_command_result.stdout"
    - number_of_the_counting == 3

- name: 'msg' shows the prompt message
  ansible.builtin.assert:
    that:
      - my_param <= 100
      - my_param >= 0
    msg: "'my_param' must be between 0 and 100"
```

앤서블 tests

앤서블에서 제공하는 "subset"명령어는 다음처럼 수정 및 변경이 되었다. 사용 시, 참고한다.

```
issubset    -> subset  
issuperset  -> superset
```

subset

말 그대로 변수가 하위에 포함이 되었는지 확인 시 사용하는 명령어. 보통 배열 비교 시 많이 사용한다.

superset

특정 배열의 값이 다른 배열에 포함이 되었는지 확인한다. 'subset'과 다른 부분은 'superset'의 비교 대상 배열에 모든 값이 포함이 되어 있으면, "참"이다.

test(superset/subset)

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

```
vars:
  a: [1,2,3,4,5,6,7,8,9,10]
  b: [2,3,4,5]
tasks:
  - debug:
      msg: "A include B"
      when: a is superset(b)

  - debug:
      msg: "B include in A"
      when: b is subset(a)
```

앤서블 tests

앤서블에서 'tests'를 명령어를 통해서 Jinja2 템플릿 기반으로 변수 값 확인이 가능하다. 이는 실제로 앤서블에서 해당 값을 직접 메모리에 할당하기 전에, 미리 템플릿으로 올바르게 선언이 되었는지 확인한다.

Jinja2로 판단이 되는 문법은 보통 다음과 같다.

```
variable is test_name  
result is failed
```

SOOOOOO BORED

..... zzz



연습문제

레지스터/조건문

register

when/failed_when

레지스터 설명

레지스터는 앤서블 모듈에서 실행된 결과를 메모리에 저장하는 기능이다. 레지스터 명령어는 'register', 사용법은 다음과 같다.

```
- <MODULE>  
  <ARGS>:  
  <ARGS>:  
    register: <REGISTER_NAME>
```

레지스터는 모듈 실행 후 출력된 표준 출력 및 오류 내용을 JSON형태로 메모리에 저장한다. 사용자가 원하는 내용을 호출 및 출력해서 사용이 가능하다.

일반적인 언어에서는 'echo', 'print'와 같은 키워드를 앤서블에서는 'debug'모듈을 통해서 출력 및 확인이 가능하다.

레지스터

```
$ 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

레지스터 사용은 아래와 같다.

```
$ vi register2.yaml
vars:
  result: true
tasks:
- name: register a variable
  command: /bin/false
  register: result
  ignore_errors: true

- name: the result is going to failed
  command: /bin/somedothing
  when: result is failed
```


register

또한 이와 같은 방식으로 사용이 가능하다.

```
vars:  
  result: true  
tasks:  
  - name: when the result is true  
    command: /bin/true  
    when: result is succeeded  
  
  - name: when the result is true  
    command: /bin/something_else  
    when: result is skipped
```

register

레지스터 내용을 출력하고 싶은 경우 다음처럼 debug모듈을 사용하면 된다.

- **debug:**
 msg: "{{ VARIABLE_NAME }}"
- **debug:**
 var: "{{ VARIABLE_NAME }}"

레지스터 반환 값

성공은 "succeeded"로 표현

success -> succeeded

실패는 "failed"라고 표현.

fail -> failed

무시는 "skipped".

skip -> skipped

위의 조건들은 'when'명령어에서 사용이 가능하다.

Boolean

앤서블에서 불린 값을 사용하기 위해서는 다음과 같이 사용한다.

```
yes / on / 1 / true
```

위의 모든 내용은 참을 나타낸다. 현재 앤서블은 위의 내용에 대해서 투표를 시작하였다. 결과는 대다수가 "true"를 선호를 하고 있다. "on", "1", "yes"와 같은 표현식 가급적이면 사용하지 않는다.

레지스터에서 값을 다룰 때, 위와 같은 리턴 변수(return value)에 대해서 확인을 꼭 해야 한다.

Boolean

앤서블에서 불린 값(Boolean value) 을 사용하기 위해서는 다음과 같이 사용한다.

yes / on / 1 / true

반대는

no / off / 0 / false

위의 모든 내용은 참을 나타낸다. 레지스터는 보통 'true', 'false'나 혹은 '0', '1'으로 반환한다.

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

혹은 부정형으로 위의 내용을 앤서블에서 적용할 수 있다.

```
vars:
```

```
  test1: true
```

```
  test2: yes
```

```
- name: Boolean false
```

```
  shell: echo "This is true"
```

```
  when: not test1
```

failed_when

혹은 부정형으로 위의 내용을 앤서블에서 적용할 수 있다. "RC"값이 "0"이 아닌 경우, 참이기 때문에 동작한다. 아래는 변수 "test1", "test2"에 "RC"값을 "1"로 하였기 때문에 'failed_when'은 참이다.

```
vars:
  test1: false
  test2: no

- name: Boolean false
  shell: echo "This is true"
  failed_when: test1
```


changed_when

변경이 된 경우에 해당 값이 동작이 된다. "RC 값"은 보통 "2"를 반환한다.

```
- hosts: localhost
  tasks:
    - name: 전달 값이 2가 아닌 경우
      ansible.builtin.shell: echo "누가 앤서블 쉽다고 했는가!"
      register: res_changed
      changed_when: "res_changed != 2"

    - name: RC값을 확인하자
      ansible.builtin.debug:
        msg: "{{ res_changed }}"

    - name: 앤서블에서 '변수'처리와 '문자'처리 조건 따증남
      ansible.builtin.shell: echo "앤서블 진짜 싫음-_- "
      changed_when: False
```

failed/changed

혹은 특정 조건에서 "실패(fail)"로 출력 및 종료를 원하는 경우 다음과 같이 작성이 가능하다.

```
- hosts: localhost
  vars:
    value: true
  tasks:
    - name: 우리는 실패 합시다. 까르르륵.
      ansible.builtin.fail:
        msg: 실패했습니다. 하지만, 당신은 {{ value }} xx!!
      when: value
```



연습문제

대화모드

PROMPT

대화모드

앤서블은 일반적으로 비대화 모드 상태에서 동작한다.

하지만, 여러가지 이유로 대화모드로 구성을 필요한 경우가 있다. 많은 사용자들은 앤서블 VS 테라폼 기준으로 비교하였을 때, 많이 나오는 조건 중 하나가 "앤서블 대화형 모드 지원"이다.

하지만, 거의 사용하지는 않는다. 앤서블 구조상 질의 형태 작업 구성에는 적합하지 않다.

대화모드

대화형 모드의 시작은 변수 구성으로 시작한다. 하지만, 조금 특이하다. 다른 언어와 다르게 변수 선언 시 인터프리터를 명시해야 한다. 대화형 모드 시작 모듈은 'prompt'으로 호출한다. 'private'은 사용자가 입력한 내용을 화면에 출력 여부이다.

vars_prompt:

- name: username
prompt: Put down your name
private: no
- name: password
prompt: Put your password

https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_prompts.html

대화모드

만약, 기본값을 선언한 상태에서 사용하고 싶으면 다음처럼 기본값을 설정한다.

```
vars_prompt:  
- name: nic_name  
  prompt: put the NIC card name  
  default: "ens4"
```


대화모드

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

대화모드

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

암호화 지원 목록

지원하는 암호화 라이브러리는 아래와 같다. 필요에 따라서 선택해서 사용한다.

1. `des_crypt` - DES Crypt
2. `bsdi_crypt` - BSDi Crypt
3. `bigcrypt` - BigCrypt
4. `crypt16` - Crypt16
5. `md5_crypt` - MD5 Crypt
6. `bcrypt` - BCrypt
7. `sha1_crypt` - SHA-1 Crypt

암호화 지원 목록

1. sun_md5_crypt - Sun MD5 Crypt
2. sha256_crypt - SHA-256 Crypt
3. sha512_crypt - SHA-512 Crypt
4. apr_md5_crypt - Apache's MD5-Crypt variant
5. phpass - PHPass' Portable Hash
6. pbkdf2_digest - Generic PBKDF2 Hashes
7. cta_pbkdf2_sha1 - Cryptacular's PBKDF2 hash
8. dlitz_pbkdf2_sha1 - Dwayne Litzenberger's PBKDF2 hash
9. scram - SCRAM Hash
10. bsd_nthash - FreeBSD's MCF-compatible nthash encoding

연습문제

루프

반복

loop

앤서블 루프 명령어 키워드는 다음과 같다. 다만, 현재는 'loop'명령어로 통합을 하고 있다. 그렇다고 해서 'with_' 명령어 사용이 불가능한 것은 아니다.

- loop: 계속 개선 중
- with_*: 이미 완성됨
- until

loop

loop 키워드는 앤서블 2.5에서 추가가 되었다.

하지만, 이 명령어는 'with_*' 명령어를 전부 대체하지는 못하고 있다. 앤서블 매뉴얼에서는 가급적이면 'loop' 명령어를 사용해서 처리한다. 루프의 유일한 단점은, 유틸리티 모듈과 함께 사용해야 효과적으로 활용이 가능하다.

1. We added loop in Ansible 2.5. It is not yet a full replacement for with_<lookup>, but we recommend it for most use cases.
2. We have not deprecated the use of with_<lookup> - that syntax will still be valid for the foreseeable future.
3. We are looking to improve loop syntax - watch this page and the changelog for updates.

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>

with_items/loop

일반적인 'with_items'명령어. 'loop'명령어의 대다수 기능은 'with_items'에서 사용이 가능하다.

```
- hosts: localhost
  tasks:
    - debug:
        msg: "{{ item }}"
      with_items:
        - 1
        - [2,3]
        - 4
```

루프에서 위와 동일하게 구현 및 사용하기 위해서 플러그인(flatten)를 사용해야 한다.

```
- hosts: localhost
  tasks:
    - debug:
        msg: "{{ item }}"
      loop: "{{ [1, [2, 3], 4] | flatten(1) }}"
```

with_fileglob

특정 파일을 루프로 불러오는 경우 다음과 같이 구현 및 구성이 가능하다.

```
- hosts: localhost
  tasks:

    - debug:
        msg: "{{ item }}"
        with_fileglob: '*.yaml'
```

```
- hosts: localhost
  tasks:
    - debug:
        msg: "{{ item }}"
        loop: "{{ lookup('fileglob', '*.txt', wantlist=True) }}"
```

with_dict/loop

딕셔너리로 구성된 배열을 다루는 경우, 아래와 같이 구성이 가능하다.

```
- hosts: localhost
  vars:
    dictionary:
      dict1:
        name: keyname1
        value: keyvalue1
      dict2:
        name: keyname2
        value: keyvalue2
  tasks:
    - name: with dict
      ansible.builtin.debug:
        msg: "{{ item.key }}" - "{{ item.value.name }}"
      with_dict: "{{ dictionary }}"

    - name: loop dict
      ansible.builtin.debug:
        msg: "{{ item.key }}" - "{{ item.value.name }}"
      loop: "{{ dictionary|dict2items }}"
```

with_sequence/loop

순차적으로 루프를 돌리기 위해서는 아래와 같이 구성이 가능하다.

```
- hosts: localhost
  tasks:
    - name: with_sequence
      ansible.builtin.debug:
        msg: "{{ item }}"
      with_sequence: start=0 end=4 stride=2 format=testuser%02x

- hosts: localhost
  tasks:
    - name: with_sequence -> loop
      ansible.builtin.debug:
        msg: "{{ 'testuser%02x' | format(item) }}"
      loop: "{{ range(0, 4 + 1, 2)|list }}"
```

loop

루프는 하나 이상의 값을 반복적으로 모듈에 전달할 때 사용한다.

```
- name: add users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
    - user1
    - user2
```

loop with list

혹은 이걸 리스트를 통해서 전달 받을 수 있다.

```
vars:
  userlist:
    - sktuser1
    - sktuser2
  groups: "wheel"
  tasks:
    - debug
      msg: "{{ item }}"
    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: 'admluser', groups: 'wheel' }
    - { name: 'normaluser', groups: 'adm' }
```


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
- name: dumped the echo register
  debug:
    msg: "{{ echo }}"
```

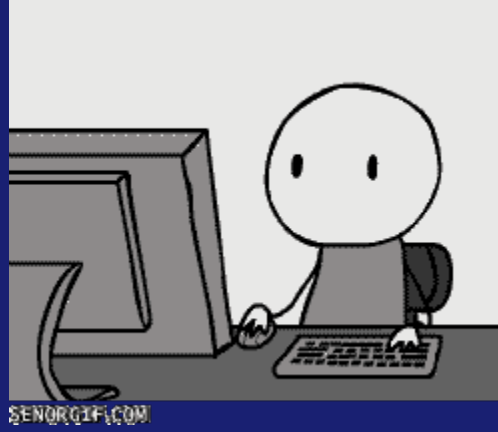
loop

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

complex loop

이러한 방식으로 잘 사용하지는 않지만, 다음과 같은 방식으로 리스트 사용하기도 한다. 다만, 두 번째 리스트 "product"이름으로 리스트를 선언한다.

```
- name: mysql user access
  mariadb_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "marriadb"
  loop: - "{{ ['user1', 'user2'] | product(['sample_db1', 'sample_db2', 'sample_db3']) | list }}"
```



연습문제

디버깅

디버깅

--tags: 특정 플레이만 실행을 원하는 경우, 모듈 작성시 태그를 구성 후 실행한다.

--skip-tags: 특정 플레이 태그만 제외하고 실행을 원하는 경우 이 옵션을 사용한다.

```
- hosts: localhost
  tasks:
    - name: tag1
      debug:
        msg: "tag1 is run"
      tags: tag1
    - name: tag2
      debug:
        msg: "tag2 is run"
      tags: tag2
$ ansible-playbook -i localhost, --tags=tag1 tags.yaml
$ ansible-playbook -i localhost, --skip-tags=tag1 tags.yaml
```


디버깅

`--start-at-task`: 특정 플레이(작업)에서 실행을 원하는 경우 이 옵션을 사용한다.

`--step`: "`--start-at-task`"를 사용하는 경우, 특정 플레이 위치부터 실행하기 때문에, 중간에 멈추고 싶은 경우 이 옵션을 같이 사용한다.

```
- hosts: localhost
  tasks:
    - name: tag1
      debug:
        msg: "tag1 is run"
      tags: tag1
    - name: tag2
      debug:
        msg: "tag2 is run"
      tags: tag2
$ ansible-playbook -i localhost, --start-at-task=tag1 --step
```

연습문제

시스템 관리

네트워크

모든 리눅스 배포판은 더 이상, `ifcfg` 형태의 파일 형태의 네트워크 설정을 더 이상 지원하지 않는다.

- 레드햇 계열은 버전 8버전 이후부터는 더 이상, "`ifcfg-rh`"를 지원하지 않는다.
- 데비안(우분투 포함) 및 수세 리눅스 경우에도 더 이상 "`ifcfg-suse`", "`ifcfg-deb`"를 지원하지 않는다.

현재 대다수 리눅스 배포판은 네트워크를 관리 아래 프로그램 기반으로 관리한다. 보통 기본 값은 "`NetworkManager`" 혹은 "`Netplan`"를 사용한다.

1. `NetworkManager`
2. `systemd-networkd`

nmcli

```
---
- hosts: node1.example.com
  tasks:
    - name: 네트워크 매니저 도구 설치
      package:
        name:
          - NetworkManager-libnm
          - nm-connection-editor
          - libsemanage-python
          - policycoreutils-python
      state: present
```

nmcli

- name: nmcli ipv4 Addr, ipv4 GW and ipv4 DNS

nmcli:

type: ethernet

conn_name: '{{ item.conn_name }}'

ip4: '{{ item.ip4 }}'

gw4: '{{ item.gw4 }}'

dns4: '{{ item.dns }}'

state: present

with_items:

- '{{ nmcli_ethernet }}'

systemd-networkd

```
$ vi systemd-networkd.jinja2
```

```
[Match]
```

```
Name="{{ name }}"
```

```
[Network]
```

```
Address="{{ address }}"
```

```
Gateway="{{ gateway }}"
```

systemd-networkd

```
- hosts: localhost
  vars:
    name: eth0
    address: 10.10.10.1
    gateway: 10.10.10.254
  tasks:
    - name: create systemd-networkd
      template:
        src: systemd-networkd.jinja2
        dest: /etc/systemd/network/eth0.network
```


디스크(파티션)

- name: LVM 파티션 생성(1기가)

parted:

device: /dev/sdb

number: 1

flags: [lvm]

state: present

part_end: 1GiB

디스크(파티션)

- name: 1기가 파티션 생성

parted:

device: /dev/sdb

number: 1

state: present

part_end: 1GiB

디스크(포맷)

```
- name: 파일 시스템 생성(ext4)
  community.general.filesystem:
    fstype: ext4
    dev: /dev/sdb1
    opts: -cc
```

디스크(포맷)

- name: 파일 시스템 생성(xfs), UUID 생성

```
community.general.filesystem:
```

```
  fstype: xfs
```

```
  dev: /dev/sdb1
```

```
  uuid: generate
```

디스크(포맷)

- name: 파일 시스템 재생성(ext4), UUID 재구성

```
community.general.filesystem:
```

```
  fstype: ext4
```

```
  dev: /dev/sdb1
```

```
  uuid: random
```

디스크(마운트)

- name: Mount up device by label

```
ansible.posix.mount:
```

```
  path: /srv/disk
```

```
  src: LABEL=SOME_LABEL
```

```
  fstype: ext4
```

```
  state: present
```

디스크(마운트)

- name: Mount up device by UUID

```
ansible.posix.mount:
```

```
  path: /home
```

```
  src: UUID=b3e48f45-f933-4c8e-a700-22a159ec9077
```

```
  fstype: xfs
```

```
  opts: noatime
```

```
  state: present
```

디스크(마운트)

- name: Mount an NFS volume

ansible.posix.mount:

src: 10.10.1.1:/nfs/shared_data

path: /mnt/shared_data

opts: rw, sync, hard

state: mounted

fstype: nfs

디스크(마운트)

- name: Mount NFS volumes with noauto according to boot option

```
ansible.posix.mount:
```

```
  src: 192.168.1.100:/nfs/ssd/shared_data
```

```
  path: /mnt/shared_data
```

```
  opts: rw,sync,hard
```

```
  boot: false
```

```
  state: mounted
```

```
  fstype: nfs
```

패키지(dnf, yum)

```
- name: install vsftpd
  dnf:
    name: vsftpd
    state: present
- name: install vsftpd
  dnf:
    name: vsftpd
    state: present
```

패키지(package)

```
- name: install httpd  
  package:  
    name: httpd  
    state: present
```

그룹

- name: Ensure group "somegroup" exists
 ansible.builtin.group:
 name: somegroup
 state: present
- name: Ensure group "docker" exists with correct gid
 ansible.builtin.group:
 name: docker
 state: present
 gid: 1750

사용자/비밀번호

```
# dnf install mkpasswd
# echo test | mkpasswd -m SHA-512 -s
$6$/EPJqJSGPi8TqGfn$toTnTAhG5.vPdo81/2Z.CRD5Z0jTmfLv46N565Tb0pz/jfajrZMWqd4GmaMtDaQX3zk78E0x/lCi
TixwVEnZd0
user:
  name: test
  shell: /bin/zsh
  groups: adm, wheel
  uid: 20000
  group: 50000
  append: true
  password: [암호]
```

예외처리

오류

핸들러

예외처리

예외 처리는 너무나 많은 방법 및 조건이 따라온다. 그래서 이 교육에서는 간단하게 앤서블 오류 핸들링 종류 및 사용 방법에 대해서 학습한다.

보통 많이 사용하는 예외처리는 아래와 같다.

1. `until`
2. `block/resuce/always`
3. `failed_*`
4. `ignore_*`
5. `run_once`

until

만약 특정 모듈에서 특정조건을 충족할 때까지 조건 대기하기 위해서는 until이라는 추가 조건을 사용한다.

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

위의 조건은 "I am okay"라는 문자열이 레지스터 result에 있으면 참이 되기 때문에 != -1 성립이 된다. 하지만 위의 명령어가 없기 때문에 올바르지 않는 값을 반환한다. 그래서 최대 2번 그리고 5초 동안 반복하게 된다.

any_errors_fatal

오류가 발생하면, 현재 배치(작업순서)에 걸려있는 모든 호스트에 동일하게 오류 작업을 수행 후, 종료한다. 오류가 발생한 시점 작업만 수행하기 때문에, 그 이후 작업에 대해서는 일반 오류와 동일하게 작업을 수행하지 않습니다.

```
tasks:
  - block:
    - include_tasks: mytasks.yaml
    - debug:
      msg: "It's error"
      any_errors_fatal: true
```

블록(block)

앤서블에서 많이 사용하는 블록 핸들링 오류 처리이다.

```
block  
rescue  
always  
+  
any_errors_fatal
```

블록(block)

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

```
block:
```

```
- yum:
```

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

```
  state: present
```

```
  loop:
```

```
- httpd
```

```
- vsfoo
```

block/rescue/always

앤서블에서 많이 사용하는 블록 핸들링 오류 처리이다. 여러 명령어를 마치 네임스페이스처럼 구성해서 사용이 필요한 경우 아래 블록처럼 구성한다.

- name: start the block session

- block:**

- yum:

- name: httpd

- state: present

- register: result_package

block/rescue

block부분에서 오류가 발생하면 rescue바로 설정된 작업을 차례대로 수행한다. 보통은 block에서 중지된 작업에 대해서 rollback작업을 수행한다.

block/rescue

"block:" 영역의 작업이 실패하면 httpd패키지 제거를 시도한다. 위의 "block"코드를 참고 한다.

```
rescue:
  - yum:
      name: httpd
      state: absent
```

block/rescue/always

```
rescue:
```

```
- yum:
```

```
  name: httpd
```

```
  state: absent
```

```
always:
```

```
- debug:
```

```
  msg: "Always runs it"
```

block/always

"always:"는 필수(mandatory)키워드는 아니다. "block:", "rescue:" 수행이 되면 무조건 "always:" 명시된 작업들은 수행이 된다.

always:

- debug:

 - msg: "Always runs it"

block/always

always:

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

ignore_

이외 예외처리가 사용하는 키워드 명령어.

1. ignore_unreachable
2. ignore_errors
3. failed_when
4. 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: true
    - command: echo "동작 했습니다."
```

ignore_errors

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

```
- hosts: node1, node2
  tasks:
    - command: /bin/noexist_cmd
      ignore_errors: yes
    - command: echo "동작 했습니다."
```

failed_when

실패 조건을 정의한다. 특정 조건이 성립이 되는 경우, 해당 작업은 실패로 간주한다.
이 부분은 위의 조건문에서 이미 다루었다.

```
---
```

```
- name: "FAILED"문자열이 있으면 실패  
  command: ls -al  
  register: command_result  
  failed_when: "'FAILED' in command_result.stderr"
```

failed_when

자주 사용하는 방법은 RC(Return Count)기반으로 확인한다. 이 부분은 위의 조건문에서 이미 다루었다.

```
---
- 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 조건 방식을 "failed_when"에서 동시에 사용이 가능하다.

```
---
- name: Fail the task rc and String existit.
  command: failed-command -h -e -l -l -o
  register: command
  failed_when: command.rc == 1 or "'FAILED' in command.stderr"
```

changed_when

이 부분은 위의 조건문에서 이미 다루었다.

```
- name: shows 'changed' when the return code is not equal to 2
  shell: ls -al
  register: result
  changed_when: result.rc != 2
```


changed_when

오류가 발생 하였을 때, 비프 소리를 만든다. 그리고 "RC"값이 "2"로 반환이 되면 변경상태로 출력한다.

```
- 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

```
$ vi plugin_filters.yaml
---
filter_version: '1.0'
module_rejectlist:
  - docker
  - easy_install
```

run_once

특정 모듈만 한번 실행하는 명령어. 보통 쉘 같은 스크립트에 적용한다. 반복적으로 계속 실행되는 부분을 방지한다. 예를 들어서 메일 발송을 한번만 해야 되는 경우, 메일 발송 모듈에 다음과 같이 적용이 가능하다.

```
- hosts: node2.example.com
  vars:
    mail_recipient: root@example.com
    body: 'Hacking So hot'
  tasks:
    - name: Send summary mail
      local_action:
        module: community.general.mail
        subject: "해킹 당했습니다."
        to: "{{ mail_recipient }}"
        body: "{{ mail_body }}"
      run_once: true
```

핸들러

HANDLER

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: latesed
- 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 설명

- 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

작업이 문제 없이 수행이 되면 "notify"에 명시된 이름을 호출한다. 작업은 handlers에 명시되어있다.

- **handlers:**

- name: restart httpd srv

service:

name: httpd

state: restarted

"notify"에 명시된 이름은 "restart httpd srv" 해당 이름에 명시된 모듈을 실행한다.

연습문제

_roles

역할

역할 설명

roles는 기본적으로 디렉터리 기반의 프레임워크를 가지고 있다. 디렉터리 생성 방법은 두 가지 방법이 있다. 아래는 간단하게 역할에서 사용하는 네임 스페이스 디렉터리를 생성하는 방법이다.

```
# 앤서블 명령어로 생성
$ mkdir roles
$ ansible-galaxy init roles/httpd

# 수동으로 생성
$ mkdir -p roles/httpd/tasks
$ touch roles/httpd/tasks/main.yaml
$ 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

역할에서 템플릿을 배포 및 사용하는 경우, 아래와 같이 템플릿 디렉터리를 생성한다. 모든 자원에 대해서는 각각 디렉터리를 가지고 있다.

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

나머지 부분은 "httpd.conf.j2"파일을 직접 열어서 확인한다.

역할 디렉터리

roles에 제공 혹은 필요한 디렉터리는 다음과 같다.

`roles/httpd/`

`tasks/` : 기본 디렉터리. 이 디렉터리가 없으면 올바르게 tasks를 불러오지 못한다.

`files/` : copy같은 명령어로 복사 시 사용하는 디렉터리.

`templates/` : template모듈로 템플릿 기반으로 파일 생성시 사용.

`vars/` : 변수 디렉터리

`defaults/` : 기본 값을 정의하는 디렉터리

`roles/httpd/`

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

역할 디렉터리

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 패키지 정보를 꼭 작성해주어야 한다.

역할(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

메인 YAML에 위와 같이 작성한 다음에 아래처럼, 각각 파일에 대한 기능을 구현한다.

```
$ 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

특정 역할을, 플레이북에 추가 시 종종 사용하는 명령어. "include_roles"와 "roles"와 비슷하지만 다른 부분이 있다. "include_role"는 역할에서 특정 작업을 불러와서 사용하는 경우가 많다.

```
- 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

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. 변수 값 선언

"role"에서 "/vars" 혹은 "/defaults" 밑에 반드시 최소 하나의 "main.yaml"이 존재해야 한다.

role default vs vars

role에서 사용하는 "defaults"와 "vars"에 대한 오해가 있어서 정리가 필요하다.

정리된 내용은 아래 링크에서 확인이 가능하며, "default", "vars"에 대한 정리는 다음과 같다.

https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

VARs 디렉터리

In general, Ansible gives precedence to variables that were defined more recently, more actively, and with more explicit scope. Variables in the defaults folder inside a role are easily overridden. Anything in the vars directory of the role overrides previous versions of that variable in the namespace. Host and/or inventory variables override role defaults, but explicit includes such as the vars directory or an include_vars task override inventory variables.

VAR/DEFAULT IN ROLE

일반 변수 디렉터리는 역할에서 다음과 같이 구성한다.

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

기본 변수 값 디렉터리는 역할에서 다음과 같이 구성한다.

```
$ mkdir roles/hostname/defaults
$ vim roles/hostname/defaults/main.yaml
node1: nodex.lab.example.com
```

ROLE with VAR

플레이 북에서 "role"를 호출하여 사용 시, 일시적으로 기존 변수 값을 "override"가 필요한 경우가 있다. 기존의 변수 값은 변경을 원하는 경우, 아래처럼 YAML, 인라인 형태로 사용이 가능하다.

```
$ vim var_roles/hostname.yaml
- hosts: localhost
  roles:
  - { role: hostname, node1: node1-1.lab.example.com }
```

```
$ vim var_roles/hostname.yaml
- hosts: localhost
  roles:
  - role: hostname
    vars:
      node1: 'node1-1.lab.example.com'
```


연습문제

플레이북 1

종합 #1

PLAYBOOK

앞에서 배운 기본적인 지식을 통해서 간단한 플레이북을 작성해본다. 현재 우리가 가지고 있는 서버는 총 4/5대의 서버를 가지고 있다.

- 앤서블 컨트롤러 노드
- 1/2/3/4 노드

각각 서버를 인벤토리에 등록 후 텍스트 파일을 생성 및 전달을 한다. 이를 구성하기 위해서 다음과 같이 디렉터리를 생성한다.

디렉터리 구성

다음과 같이 플레이북을 생성 및 구성한다.

```
$ mkdir playbook1  
$ cd playbook1  
$ touch inventory  
$ touch ansible.cfg  
$ mkdir roles
```

플레이북 디렉터리

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

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

```
1 directory, 2 files
```

PLAYBOOK

```
$ vim inventory
```

```
[node1]
```

```
192.168.90.11 → node1.example.com
```

```
[node2]
```

```
192.168.90.12 → node2.example.com
```

PLAYBOOK

```
$ echo "Hello an automation World" > welcome.html
```

```
$ vim copy.yaml
```

```
- hosts: node1
```

```
  tasks:
```

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

```
      yum:
```

```
        name: httpd
```

```
        state: latest
```

PLAYBOOK

올바르게 동작하면 추가로 다음과 같이 기능을 추가한다.

```
- name: copy the welcome.html to {{ inventory_hostname }}  
  copy:  
    content: "Hello an automation world"  
    dest: /var/www/html/welcome.html
```


PLAYBOOK



PLAYBOOK

웹 서버가 설치가 안되어 있으면 아래와 같이 웹 서버 패키지도 같이 설치를 진행한다.

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

PLAYBOOK

웹 페이지 배포 플레이북을 생성한다.

```
- 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.

가상 머신(node1/2/3/4) 플레이북를 통해서 패키지 및 "welcome.html"파일은 배포 하였지만, 호스트 이름을 아직 수정하지 않았다. 호스트 이름을 변수에 다음처럼 선언한다.

INVENTORY

인벤토리를 다음처럼 수정을 한다.

```
[node1]  
10.10.1.1 nodename=node1.example.com
```

```
[node2]  
10.10.1.2 nodename=node2.example.com
```

PLAYBOOK

그리고 추가적으로 플레이북 파일을 더 생성한다.

```
$ touch hostname.yaml
$ vim hostname.yaml
- name: set up hostname to {{ inventory_hostname }}
  hostname:
    name: "{{ nodename }}"
```

메인 파일에 "호스트 이름 변경"에 대한 부분을 통합한다.

```
$ 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
```


플레이북 + 역할

다중 플레이북 구성

역할 설명

이쯤이면 슬슬 혼동이 오는 부분, 바로 roles vs playbook이다.

roles에는 tasks, files, templates같은 기능들이 모여서 하나의 기능을 구현하여 이걸 호스트에 제공하는 기능. 앞서 이야기 하였지만, 일종의 함수와 같은 역할을 한다.

역할 설명

플레이북에서 하나 이상의 역할(roles) 모아서 특정 작업에 대한 프로세스를 구현하고 있는 파일.

예를 들어서 "webserver.yaml", "dbserver.yaml" 이런 식으로 좀 더 추상적인 작업의 워크 플로우를 role기반으로 구성하고 있다. 플레이북은 항상 다음과 같은 내용을 가지고 있다.

1. 어떤 호스트에서 어떠한 작업을 진행할 것인가?
2. 어떠한 역할을 가지고 작업을 할 것인가?
3. YAML에 어떠한 작업(task)나 혹은 역할이 선언이 되었는가?

_PLAYBOOK

통합

include_playbook

앞에서 만든 플레이북을 하나로 통합한다.

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

import_playbook

'import_playbook'은, 이미 작성한 플레이북 파일을 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 불러옴. 장점은 빠르지만, 단점은 불러오는 순간에 문법 검사를 하기 때문에 문법에 문제가 있는 경우 중간에 종료가 됨.

include_playbook

플레이북에 기능을 추가한다. 운영체제의 모든 패키지를 최신으로 업데이트 한다.

```
$ 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
```

테스트 문법

정규식

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/infra/resource/cinder>

이걸 코드로 변경하면 다음과 같이 표현이 가능하다.

is + match/search/regex

vars:

url: https://vms.example.com/users/infra_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("exampleW.com/w+/foo")