IT Automation with

ANSIBLE

Agenda

- Ansible playbooks basics
- Running Ansible Playbooks
- Ansible variables
- Creating reusable Ansible Playbooks
- Ansible tags

Ansible playbooks

Basics

- Playbooks are expressed in YAML format and have a minimum of syntax which intentionally tries to not be a programming language, but rather a model of a configuration or a process.
- Playbook files are normally saved with **.yml** extension although this is not mandatory
- Each playbook is composed of one of more ‘plays’ in a list.
- A play is an ordered set of tasks which sould be run against host(s) selected from your inventory.
- Playbooks can declare configurations, but they can also orchestrate steps of any manual ordered process.
- They are preferably stored stored in VCS such as Git.
- Each playbook is composed of one or more plays which in turn contain one or more tasks.

Ansible playbooks

Basics

- All playbooks start with a dash **“-”** followed by a space and the “**hosts**” key to indicate the managed nodes this play will apply to.
- An optional “**name**” key can also be specified to identify each play in case there are many of them.
- The user that will be used to perform tasks in the managed nodes can be specified through the *remote_user* option.
- Only the space character can be used for identation; tab characters are not allowed!
- If you use the *vim* editor you may want to use the following trick to your vim config (usually in $HOME/.vimrc):

  autocmd FileType yaml setlocal ai ts=2 sw=2 et

Ansible playbooks

Privileges escalation and tasks

- Privileges escalation can be done at play and task level.
- Play level:

```
---
- hosts: webservers
  remote_user: yourname
  become: yes
```

Task level:

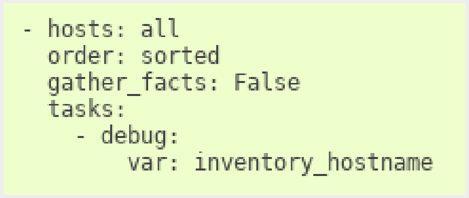
```
---
- hosts: webservers
  remote_user: yourname
  tasks:
    - service:
        name: nginx
        state: started
      become: yes
      become_method: sudo
```

- A “**tasks**” key contains the tasks that will be executed in the managed hosts, one at a time.
- It’s **important** to be aware that all keys: **name, vars, hosts** and **tasks** should be at the same identation level.
- Each task should have a name, which is included in the playbook output.

Ansible playbooks

Controll the order in which hosts are run

- As of Ansible 2.4 is possible to specify the order in which hosts are run.
- The default is to follow the order supplied by the inventory.



```
- hosts: all
  order: sorted
  gather_facts: False
  tasks:
    - debug:
        var: inventory_hostname
```

- Possible values for order are:
  - **inventory**: the default. The order is ‘as provided’ by the inventory.
  - **reverse_inventory**: This reverses the order ‘as provided’ by the inventory
  - **sorted:** Hosts are alphabetically sorted by name.
  - **reverse_sorted:** Hosts are sorted by name in reverse alphabetical order.
  - **shuffle:** Hosts are randomly ordered each run.

Ansible playbooks

Tasks and modules

- When running the playbook, which runs top to bottom, hosts with failed tasks are taken out of the rotation for the entire playbook. If things fail, simply correct the playbook file and rerun.
- Every task must contain a key with the name of a module and a value with the arguments to that module.

```
- name: install nginx
  apt: name=nginx update_cache=yes
```

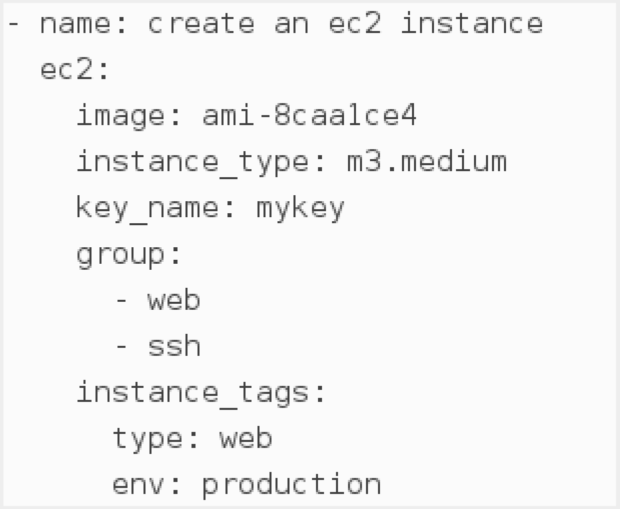
```
- name: install package with pip
  pip:
    name: "{{ item.name }}"
    version: "{{ item.version }}"
    virtualenv: "{{ venv_path }}"
```

- Modules arguments are strings by default. But they can also be specified as a YAML directory, which is helpful when using modules that support complex arguments

Ansible playbooks

Syntax and plays

- An argument is considered complex if it is a list or a dictionary, e.g:



```
- name: create an ec2 instance
  ec2:
    image: ami-8caa1ce4
    instance_type: m3.medium
    key_name: mykey
    group:
      - web
      - ssh
    instance_tags:
      type: web
      env: production
```

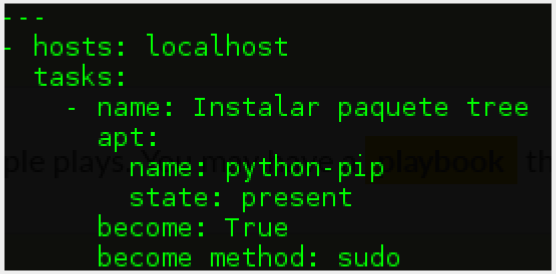
- Multiple plays can be specified within a playbook file, each should have its own "**hosts**" key indicating the target this play will apply to.

Running Ansible playbooks

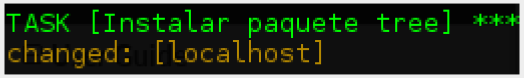
Ansible playbooks

How to run playbooks

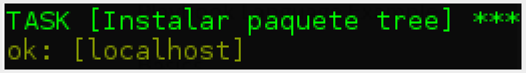
- Basic syntax is: *ansible-playbook file.yml*
- Each task will show outputs status information for each task it executes in the play. If the state of the host matches the arguments of the modules, Ansible takes no action and responds with a ***ok** state.* If there is a difference between the state of the host and the module arguments, Ansible will change the state and return **changed**.
- Tasks in Ansible playbooks are idempotent (mostly), and it is safe to run the playbook multiple times.
- It's possible to execute a dry-run to check what changes (if any) Ansible would have done to the host if ran by specifying the "*-C*" or "*--check*".



```
---
- hosts: localhost
  tasks:
    - name: Instalar paquete tree
      apt:
        name: python-pip
        state: present
      become: True
      become_method: sudo
```



```
TASK [Instalar paquete tree] ***
changed: [localhost]
```



```
TASK [Instalar paquete tree] ***
ok: [localhost]
```

Ansible playbooks

Tips

- It is recommended that you run ***ansible-playbook --‑syntax-check <playbook.yml>*** before executing it in order to verify there are not syntax issues.
- After playbook execution has finished, you can find at the bottom of the screen a summary of the execution itself.

Ansible playbooks
Demo

- Write an Ansible playbook that install an apache2 webserver into the *vagrant1* host (latest version).

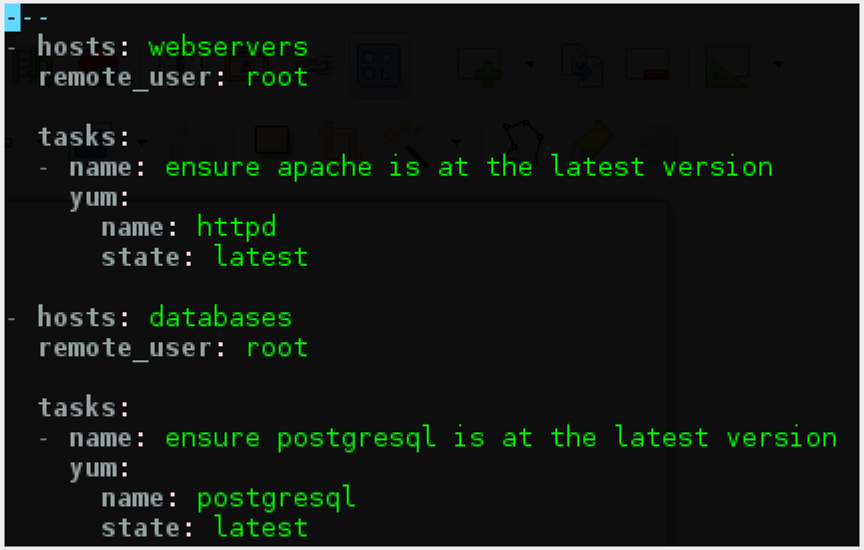
  Once it is installed, replace the default index.html file at /var/www/html directory with a custom one that has the message “Server managed by Ansible”.

  The playbook should start the *apache2* service and make sure it is *enabled* at boot time.

Ansible playbooks

Multiple plays

- A playbook file can contain one or more plays that will apply at different target hosts.
- Plays will be executed in the order they are written in the playbook.



```
---
- hosts: webservers
  remote_user: root

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest

- hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest
```

Ansible variables

Ansible variables

Basic

- While automation exists to make it easier to make things repeatable, all of your systems are likely not exactly alike.
- On some systems you may want to set some behavior or configuration that is slightly different from others. (Such as you might need to find out the IP address of a system and even use it as a configuration value on another system)
- Variables are often used with Conditionals and Loops.
- If a variable exists in more than one location, the one with the higher precedence will win. A link to a variable precedence explanation is available in References section.
- At a basic level, variables can be defined at three scopes: global, host and play
  - **Global scope:**   variables that are set into Ansible configuration directly or from cli.
  - **Host scope:** variables that set at host / group level at inventory or in separate files with host_vars and group_vars.
  - **Play level:** variables that are set in the play header.

Ansible variables

What's a valid name for a variable?

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

  Example:

  - foo_port is valid, *foo5* too
  - *foo-port*, *foo port*, *foo.port* and *12* are not valid variable names.

- YAML also supports dictionaries which map keys to values.
- A specific field in the dictionary can be referenced using brackets or dot notation

  Example:

```
foo:
  field1: one
  field2: two
```

```
foo['field1']
foo.field1
```

Ansible variables

Variables defined at command line level

- Variables defined at command line for commands such as **ansible** and **ansible-playbook** have the highest precedence and will override variables with same name that might be defined at any other level.
- They can be useful in use cases where for a specific run a certain variable wants to be overriden.
- To specify a variable at command line the option *-e* or *--extra-vars* should be used.

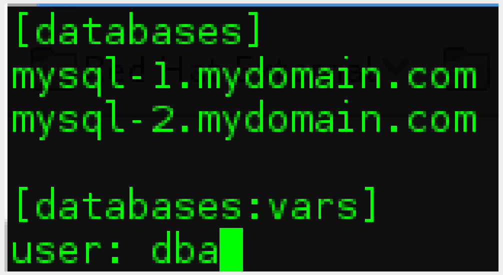
```
$ ansible-playbook -i inventario -e ntp_server=ntp1.example.com
playbook.yml
```

Ansible variables

Variables defined at host/group level

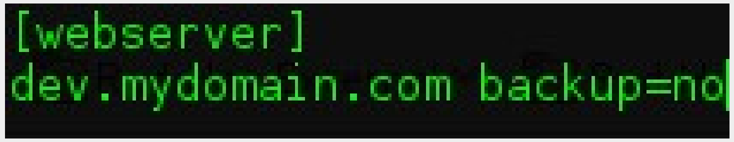
- There are some times where it is desirable to set variables based on what group a host belongs to, e.g: “All database servers should have a *dba* user created”.
- Variables at host / group level can be set directly in the inventory file, although is not recommended approach there might be some code that still use t his method.

   Example:



```
[databases]
mysql-1.mydomain.com
mysql-2.mydomain.com

[databases:vars]
user: dba
```



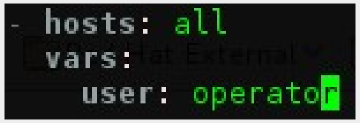
```
[webserver]
dev.mydomain.com backup=no
```

- Note: The recommended approach for defining variables at host / group level is to use **host_vars** and **group_vars** directories instead.

Ansible variables

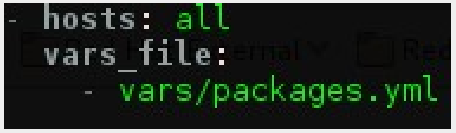
Variables defined at play level

- Variables defined at play level are set before the *tasks* block with a *vars* key, e.g:



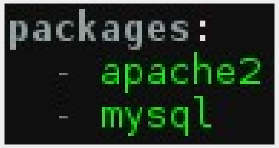
```
- hosts: all
  vars:
    user: operator
```

- Another possibility would be to define the variables in external files and reference these files by using the **vars_file** directive, e.g:



```
- hosts: all
  vars_file:
    - vars/packages.yml
```

- The file referenced should be in YAML format, e.g:

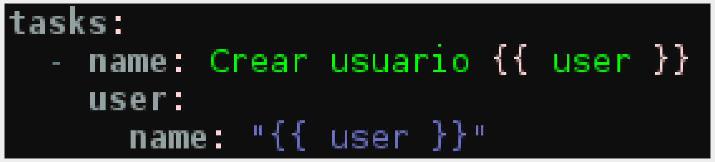


```
packages:
  - apache2
  - mysql
```

Ansible variables

Using variables in playbooks

- Regardless of where variables are defined they can be referenced into a playbook.
- Variable interpolation in Ansible is done by putting the variable name between double curly braces, e.g:



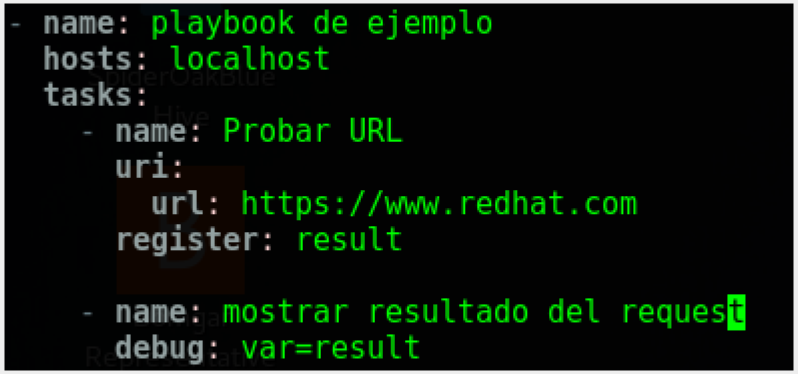
```
tasks:
  - name: Crear usuario {{ user }}
    user:
      name: "{{ user }}"
```

- A **very common** error when interpolating variables in playbooks occurs when users specify a variable name as the first element of a value and they don’t enclose it within double quotes!

Ansible variables

Registered variables

- The output of a command can be saved into a variable through the **register** statement.
- The **register** statement takes a variable as an argument where the result will be kept for either debugging purposes or to achieve something else.
- The content of a registered variable will vary depending on the module that is used on each case.
- In order to check possible values for the results you may use the -v



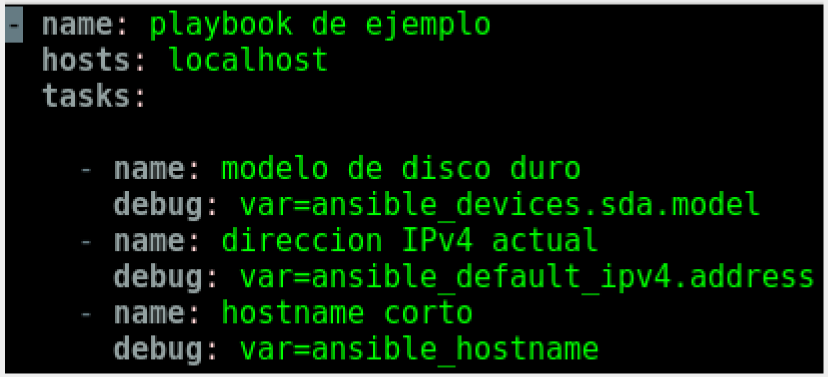
```
- name: playbook de ejemplo
  hosts: localhost
  tasks:
    - name: Probar URL
      uri:
        url: https://www.redhat.com
      register: result

    - name: mostrar resultado del request
      debug: var=result
```

Ansible variables

Facts

- Ansible facts are variables that are automatically discovered on a managed host. They contain specific information that can be used like any other variables
- They contain specific information that can be used like any other variables in plays, conditionals, loops, etc.
- Typical facts variables include information like IP address of the remote host, hostname, operating system, etc.
- Ansible facts are gathered through **setup** module.



```
- name: playbook de ejemplo
  hosts: localhost
  tasks:

    - name: modelo de disco duro
      debug: var=ansible_devices.sda.model
    - name: direccion IPv4 actual
      debug: var=ansible_default_ipv4.address
    - name: hostname corto
      debug: var=ansible_hostname
```

Ansible variables

Facts

- If you know in advance that you won’t be using any fact data in your playbooks, you might want to turn facts gathering. This is done by adding a **gather_facts: no** setting just below the **hosts:** directive in the play.
- It’s possible to filter that match a certain pattern through the “*filter*” parameter to the **setup** module, e.g:

```
# Display only facts about certain interfaces.
$ ansible all -m setup -a 'filter=ansible_eth[0-2]'
```

Ansible variables

Custom facts

- Custom facts can be defined in a static file, formatted as an INI file or using JSON. They can also be executable scripts which generate JSON output, just like a dynamic inventory script.
- Using custom facts allows a sysadmin to define certain values for managed hosts to conditionally run tasks.
- By default the **setup** module loads custom fact files from */etc/ansible/facts.d* (you can supply an alternative path by modifying the *fact_path* play directive).
- The name of each fact file or script should end in **.fact.** Static custom files should be either in INI or JSON format.
- Dynamic custom facts scripts are also allowed but they should return JSON output (same than an inventory script).
- Custom facts are stored by **setup** module in the *ansible_local* variable

Creating reusable playbooks

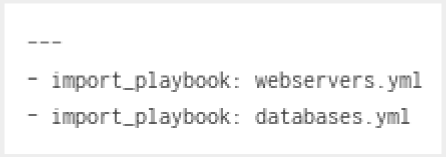
How to reuse code

- While it’s possible to write a playbook in one very large file, eventually you’ll want to reuse files and start to organize things. There are three ways to do this: *includes*, *imports* and *roles*.
- *Includes* and *imports* (added in 2.4) allow users to break up large playbooks into smaller files, which can be used across multiple parent playbooks or even multiple times within the same Playbook.
- *Roles* allow more than just tasks to be packaged together and can include variables, handlers, or even modules and other plugins. Unlike *includes* and *imports*, roles can also be uploaded and shared via Ansible Galaxy.

Creating reusable playbooks

Includes vs imports

- Ansible has two modes of operation for reusable content: *dynamic* and *static.*
- **Differences** (new in Ansible +2.4):
  - All *import** statements are pre-processed at the time playbooks are parsed.
  - All *include** statements are processed as they encountered during the execution of the playbook
- Loops can not be used with imports at all.
- It's possible to include other playbooks into a master playbook by using *import_playbook:*

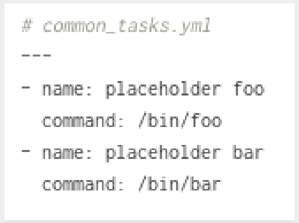


```
---
- import_playbook: webservers.yml
- import_playbook: databases.yml
```

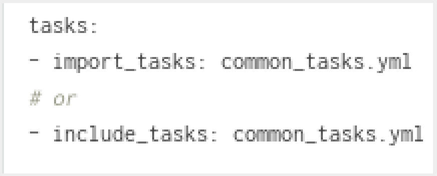
Creating reusable playbooks

Including and importing tasks

- In order to include tasks files two directives can be used: *import_tasks* or *include_tasks.*
- A task include file simply contains a flat list of tasks:

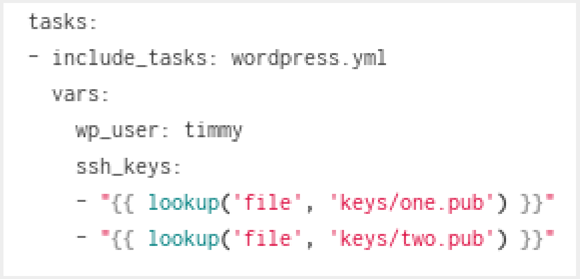


```
# common_tasks.yml
---
- name: placeholder foo
  command: /bin/foo
- name: placeholder bar
  command: /bin/bar
```



```
tasks:
- import_tasks: common_tasks.yml
# or
- include_tasks: common_tasks.yml
```

- You can read contents from files in your filesystem and populate variables using the lookup() filter:



```
tasks:
- include_tasks: wordpress.yml
  vars:
    wp_user: timmy
    ssh_keys:
    - "{{ lookup('file', 'keys/one.pub') }}"
    - "{{ lookup('file', 'keys/two.pub') }}"
```

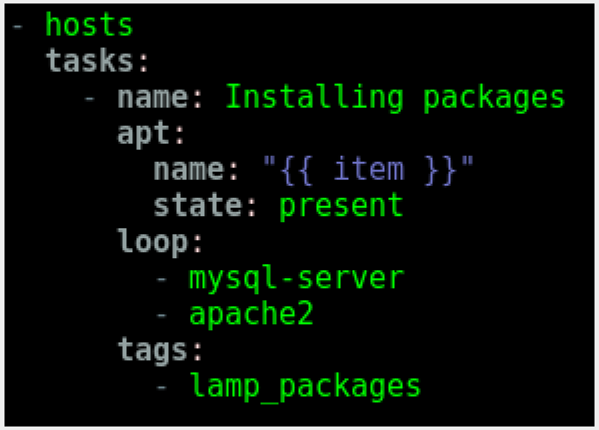
Exercises

Ansible tags

Ansible tags

Basics

- In cases where a large playbook exists, it might be desirable to run just a subset of tasks. Tags can be applied to Ansible resources to achieve this through the ***tag*** keyword.
- The ***tag*** keyword expects a list of tags that will be applied to the resource.



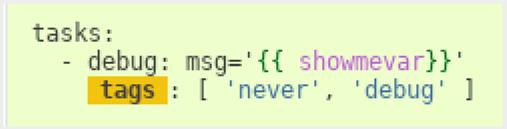
```
- hosts
  tasks:
    - name: Installing packages
      apt:
        name: "{{ item }}"
        state: present
      loop:
        - mysql-server
        - apache2
      tags:
        - lamp_packages
```

- Filtering tasks based on tags can **only** be done from the **ansible-playbook** command through the --***tags*** or ***--skip-tags.***

Ansible tags

Basics

- The ***tag*** keyword expects a list of tags that will be applied to the resource.
- Tags can be reused by applying same tag to more than one task.
- Apart from being applied to tasks, tags can also be set elsewhere so you don't have to write it on every task, e.g: at play level, on a specific role and *import_** statements. However, this does NOT apply to include_tasks or any other dynamic incudes, as the attributes applied to an include will only affect the include itself!
- The ***--list-tags*** option to **ansible-playbook** command will show all tags available.
- The ***--list-tasks*** option to **ansible-playbook** command will show which tags apply to which resources.
- There is a set of "special" tags: ***always*** tag will cause the task is always run unless it is explicitly excluded through the ***--skip-tags*** always. Likewise, there is also the ***never*** tag which will have the opposite behavior, i.e: it will prevent a task to run unless a tag is explicitly requested:



```
tasks:
  - debug: msg='{{ showmevar}}'
    tags: [ 'never', 'debug' ]
```

This task will only run when the ***debug*** or ***never*** Tag is explicitly requested.

Ansible tags

Basics

- Apart from the aforementioned set of special tags, there are also 3 special tags:
  - **tagged:** will run any tagged resource
  - **untagged:** will run only untagged resources (opposite than ‘tagged’).
  - **all:** The default behavior ansible runs as. It will run all tasks.

- Tags could be useful to differentiate tasks for different environments and/or servers types.

References

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