IT Automation with

Agenda

- Introduction to Ansible control structures
- Ansible conditionals
- Ansible handlers
- How errors are handled in Ansible?

Ansible loops

Introduction

- Loops in ansible as in traditional programming languages allows iterating over a set of items avoiding repeating the task several times.
- Several loop structures exist and you should know at least the most common ones.
- Starting in Ansible 2.5 a new simpler and more intuitive keywoord ‘loop’ was included.
- Loops are actually a combination of things with_ + lookup(), so any lookup plugin can be used as a source for a loop.

Ansible loops

Simple Loops

- Simple or Standard loops are the simplest to use.
- They iterate over a set of items that can be single items or lists of hash/dictionaries.
- To iterate over a set of items you should use *with_items* loop. It takes the the list of elements you will iterate over as a value

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  with_items:
     - testuser1
     - testuser2
```

VS

```
- name: add user testuser1
  user:
    name: "testuser1"
    state: present
    groups: "wheel"
- name: add user testuser2
  user:
    name: "testuser2"
    state: present
    groups: "wheel"
```

Ansible loops

Simple Loops

- Conditional structures (*when*) can be used within loops, however it will evaluate each item separately.
- You can pass already defined variables to with_items, e.g:

```
with_items: "{{ somelist }}"
```

- Lastly items you iterate over with ‘with_items’ don’t have to be simple list of strings, you can also iterate over a list of hases by referencing subkeys:

```
- name: add several users
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  with_items:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Ansible loops

Nested Loops

- Nested loops takes two or more lists and loops run inside loops.
- The '*with_nested*' keyword is used for this type of loops.

```
- name: give users access to multiple databases
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  with_nested:
    - [ 'alice', 'bob' ]
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

```
- name: here, 'users' contains the above list of employees
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  with_nested:
    - "{{ users }}"
    - [ 'clientdb', 'employeedb', 'providerdb' ]
```

Ansible loops

Looping over files

- You can iterate over file's content by using *with_file* loop.
- Item will be set to the content of each file in sequence, e.g:

```
---
- hosts: all

  tasks:

    # emit a debug message containing the content of each file.
    - debug:
        msg: "{{ item }}"
      with_file:
        - first_example_file
        - second_example_file
```

```
TASK [debug msg={{ item }}] ***************************************************
ok: [localhost] => (item=hello) => {
    "item": "hello",
    "msg": "hello"
}
ok: [localhost] => (item=world) => {
    "item": "world",
    "msg": "world"
}
```

Ansible loops

Lookups, what are they?

- Lookup plugins allow access to outside data sources, e.g: environment variables or even key value stores.
- Lookups occur **on the local computer**, not on the remote computer.
- They are executed in the directory containing the role or play, as opposed to local tasks which are executed with the directory of the executed script.

```
vars:
  motd_value: "{{ lookup('file', '/etc/motd') }}"
tasks:
  - debug:
      msg: "motd value is {{ motd_value }}"
```

Ansible loops

New structure

- As of Ansible 2.5 a new keyword ‘loop’ was introduced as a way to make loops in ansible easier.
- ‘*loop*’ main idea is to abstract out some of the magic of what *with_** really is.
- For simple loops such as old *with_items* you just replace any of those with ‘*loop*’ keyword:

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
     - testuser1
     - testuser2
```

Ansible loops

New structure

- As of Ansible 2.5 a new jinja2 function was introduced named *query* for invoking lookup plugins.
- The difference between '*lookup*' and '*query*' is basically that '*query*' will always return a list.
- The default behavior of '*lookup*' is to return a string of comma separated values.
- '*lookup*' can be explicitly configured to return a list using '*wantlist=True*'

```
lookup('dict', dict_variable, wantlist=True)

query('dict', dict_variable)
```

```
loop: "{{ query('nested', ['alice', 'bob'], ['clientdb', 'employeedb', 'providerdb']) }}"

loop: "{{ lookup('nested', ['alice', 'bob'], ['clientdb', 'employeedb', 'providerdb'], wantlist=True) }}"
```

Ansible loops

Looping over the inventory

- It's possible to loop over the inventory or just a subset of it by using **groups** variable like this:

```
# show all the hosts in the inventory
- debug:
    msg: "{{ item }}"
  loop: "{{ groups['all'] }}"
```

- There is also a specific lookup plugin **inventory_hostnames** that can be used:

```
# show all the hosts in the inventory
- debug:
    msg: "{{ item }}"
  loop: "{{ query('inventory_hostnames', 'all') }}"

# show all the hosts matching the pattern, ie all but the group www
- debug:
    msg: "{{ item }}"
  loop: "{{ query('inventory_hostnames', 'all!www') }}"
```

Demo

Ansible conditionals

Ansible conditional

When statement

- Ansible can use conditionals to execute tasks or plays when certain conditions are met, e.g: checking whether is there enough hard disk space before installing an application, etc.
- Conditions in Ansible are expressed through the **when** statement. It takes a value that will be evaluated similar to an if statement on regular programming languages.
- The when statement uses a raw Jinja2 expression without double curly braces

```
tasks:
  - name: "shut down Debian flavored systems"
    command: /sbin/shutdown -t now
    when: ansible_os_family == "Debian"
    # note that Ansible facts and vars like ansible_os_family can be used
    # directly in conditionals without double curly braces
```

Ansible conditional

When statement

Operator	Example
Equal (value is a string)	ansible_os_family == “Debian”
Equal (value is numeric)	max_memory == 1024
Greater than	min_memory > 512
Less than	min_memory < 256
Less than or equal to	min_memory <= 256
Greater than or equal to	min_memory >= 512
Not equal to	min_memory != 512
Variable exists	min_memory is defined
Variable does not exist	min_memory is not defined
Variable is set to 1, yes or True	available_memory
Variable is set to 0, no or False	not available_memory
First variable’s value is present in second variable’s list	my_user in superusers

Ansible conditional

When statement

- Multiple conditions to be evaluated can be grouped with parentheses:

```
tasks:
  - name: "shut down CentOS 6 and Debian 7 systems"
    command: /sbin/shutdown -t now
    when: (ansible_distribution == "CentOS" and ansible_distribution_major_version == "6") or
          (ansible_distribution == "Debian" and ansible_distribution_major_version == "7")
```

- Multiple conditions that all need to be true (a logical ‘and’) can be specified as a list:

```
tasks:
  - name: "shut down CentOS 6 systems"
    command: /sbin/shutdown -t now
    when:
      - ansible_distribution == "CentOS"
      - ansible_distribution_major_version == "6"
```

Ansible conditional

Conditionals and loops

- Conditionals can be used along with registered variables:

```
- hosts: lamp
  tasks:
    - name: mysql server status
      command: /usr/bin/systemctl is-active mysqld
      ignore_errors: yes
      register: webserver

    - name: Restart nginx if mysql is running
      service:
        name: nginx
        state: restarted
      when: webserver.rc == 0
```

- Conditionals can also be used with loops, however in these cases the when statement will be processed for each item separately:

```
tasks:
    - command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
```

Ansible handlers

Ansible handlers

Basics

- Playbooks have a basic event system that allows to ‘notify’ other tasks whether a change has been performed, e.g: a config file been altered.
- Handlers are simple tasks that act based on a ‘notify’ action triggered by another task.
- They have its own section at the end of the play (just after tasks) as a separate block.
- Handlers are once executed **once**, regardless of how many times they are notified before in the *tasks* block.
- Handlers will be executed **after** all tasks have been executed, and only when the tasks that invoked them have a “*changed*” state

Ansible handlers

Basics

```
- name: template configuration file
  template:
    src: template.j2
    dest: /etc/foo.conf
  notify:
     - restart memcached
     - restart apache
```

```
handlers:
    - name: restart memcached
      service:
        name: memcached
        state: restarted
    - name: restart apache
      service:
        name: apache
        state: restarted
```

- Handlers will be executed **after** all tasks have been executed, and only when the tasks that invoked them have a "*changed*" state.
- Handlers are run in the order they appear under the *handlers* section, NOT in the order they are notified!
- If you are including tasks in your play / playbook, *notify* will only work on handlers that are included statically (*import_**).

Ansible handlers

Basics

- As of Ansible 2.2, handlers can also “listen” to generic topics and tasks then notify those topics:

```
handlers:
    - name: restart memcached
      service:
        name: memcached
        state: restarted
      listen: "restart web services"
    - name: restart apache
      service:
        name: apache
        state:restarted
      listen: "restart web services"

tasks:
    - name: restart everything
      command: echo "this task will restart the web services"
      notify: "restart web services"
```

Exercises

Ansible error handling

Ansible error handling

Basics

- Whenever a playbook is run Ansible evaluates each task return code to check it success state. In case a task fails for a certain host, then the rest of the tasks are not executed on that host to avoid inconsistencies.
- Ansible has some features to handle these situations in a similar fashion with exceptions in most programming languages, e.g: Java.
- **Ignoring Task Failure**: the *ignore_errors* can be used within a task in order to ignore a possible error on that task if it is expected and to allow to continue with play tasks on that host:

```
- name: Instalando un paquete que no existe
  apt:
    name: paquetequenoexiste
    state: present
  ignore_errors: yes
```

Ansible error handling

Basics

- **Forcing handlers execution:** after a task execution on a certain node fails any handlers which had been notified previously won't get executed. Ansible has a feature to override this behavior through the *force_handlers* keyword:

```
---
- hosts: webservers
  force_handlers: yes
  tasks:
    - name: example task
      command: /bin/true
      notify: restart apache2
    - name: Instalando un paquete que no existe
      apt:
        name: paquetequenoexiste
        state: present
      ignore_errors: yes
  handlers:
    - name: restart apache2
      service:
        name: apache2
        state: restarted
```

Ansible error handling

Basics

- **Controlling what defines a failure:** In certain situations you may want to control what is considered a failure in a task regardless its output, for instance if the string “FAILED” is in the output. This can be done through the ***failed_when*** keyword:

```
- name: Fail task when the command error output prints FAILED
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

or even based on the command return code as follows:

```
- name: Fail task when both files are identical
  raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

Note: a similar approach for overriding the changed result can be achieved through the changed_when keyword.

Ansible error handling

Blocks

- **Blocks**: a block in Ansible allows you to group a logical group of tasks. Most of what can be applied to a single task can be applied at block level, e.g: a when condition:

```
tasks:
  - name: Install Apache
    block:
      - yum:
          name: "{{ item }}"
          state: installed
        with_items:
          - httpd
          - memcached
      - template:
          src: templates/src.j2
          dest: /etc/foo.conf
      - service:
          name: bar
          state: started
          enabled: True
    when: ansible_distribution == 'CentOS'
    become: true
    become_user: root
```

Ansible error handling

Blocks

- **Blocks for error handling**: blocks also allow error handling features through the following structure:
  - **block**: Defines the main set of task to be run (similar to try block in Java)
  - **rescue**: Defines the set of tasks that will be executed if tasks enclosed within the block section fails (similar to catch block in Java).
  - **always**: Defines the tasks that will always be run no matter what previous error occurred (or not) in the block and rescue section.

```
---
- hosts: webservers
  tasks:
    - block:
        - name: upgrade the database
          command: /usr/local/bin/upgrade_database
      rescue:
        - name: rollback the upgrade
          command: /usr/local/bin/rollback_database
      always:
        - name: restart the database
          service:
            name: oradb
            state: restarted
```

Exercises