Ansible Hands-On

DevOps Brasília

March 9, 2016

Deploy an application manually is usually time-consuming

- Most of the applications rely on different services to work correctly
- These services usually run on a distributed set of computing resources and communicate using various networking protocols
- Wire up these services by hand is time-consuming, error-prone, and it makes difficult to implement continuous delivery, for instance.
- Therefore, one way to deal with this problem is to use configuration management tools like Ansible, Chef, Puppet, Salt, among others.

What do we mean by configuration management?

- Writing the states for the servers, and then, using a tool to enforce that the servers are in the required state:
  1. the right packages are installed
  2. the configuration files contain the expected values and the correct permissions
  3. the right services are running
  4. . . .

What can we expect from configuration management tools?

- they can help us on implementing continuous delivery, i.e., on implementing the blue-green deployment approach[1].
- on dealing with deployment orchestration. In other words, when there are multiple servers involved and the tasks must happen in a specific order. For instance, a database must be set up before bringing up the application servers.
- they can provision new servers. In the context of IaaS cloud, this means to spinning up new virtual machine instance.
- they help on guarantee the *idempotence* property.

And Ansible, what is it Good For?

- For describing the state of the servers through its DSL
- For doing deployment as well as configuration management
- For performing actions on multiple servers with a simple state model
- For control the order that the actions must happen in
- For talking to the public clouds API (e.g., AWS EC2, Google Compute Engine, Azure), as well as any cloud that supports the OpenStack API

How Ansible works

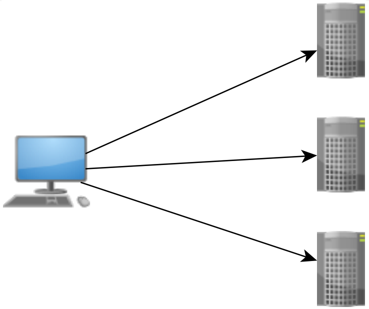

Figure 1: Using Ansible to perform actions on three remote servers

1. it makes an SSH connection for each server
2. it executes the first task on all the three nodes simultaneously:
    1. generate a Python script that represents the task
    2. copy the script to the servers
    3. execute the script on the nodes
    4. wait for the script to complete on all the hosts

What are good Ansible's characteristics?

- Easy-to-read syntax: its script (i.e., playbook) is built on top of the YAML format
- Nothing to install on the remote servers
- Push-based: it is the developer who controls when the changes happens to the servers
- Built-in modules: there are many modules to perform the tasks. Modules are idempotent
- A very thin layer of abstraction: we don't need to learn a new package manager
- It has a low learning curve

Setup Environment

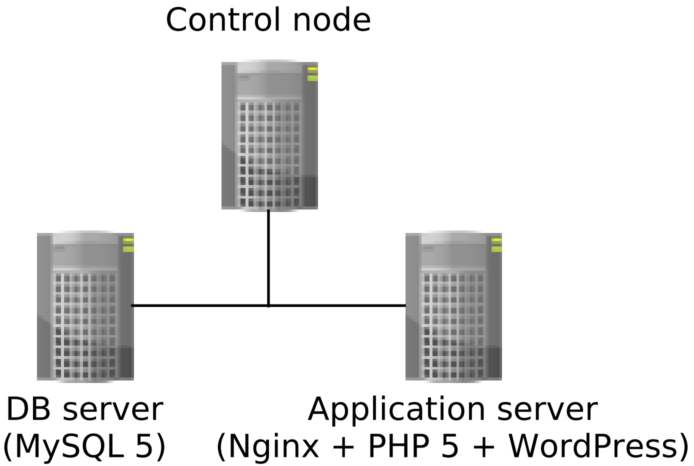

Figure 2: We will use Ansible to deploy WordPress with services running on two different nodes

Starting with Ansible

1. Provision the virtual machines and connect to the control node

   ```
   vagrant up
   vagrant ssh
   ```

2. Create an inventory file to inform Ansible what are the remote servers, as well as how to connect to them.
3. Write the configuration scripts
4. Push the configuration scripts to the remote nodes

Ansible's inventory file

```
[defaults]
hostfile = hosts
remote_user = vagrant
private_key_file = ~/.ssh/id_rsa
host_key_checking = False
nocows = 1
```

```
[dbservers]
mysqlserver ansible_ssh_host=10.100.100.11 ansible_ssh_port=22

[webservers]
nginx ansible_ssh_host=10.100.100.12 ansible_ssh_port=22
```

- By default, Ansible looks for an inventory file (*ansible.cfg*) in:
  1. *ANSIBLE_CONFIG_ENVIRONMENT* variable
  2. *./ansible.cfg*
  3. *$HOME/.ansible.cfg*
  4. /etc/ansible/ansible.cfg

Ansible's hello world

```
ansible all -i hosts -m ping
```

or only

```
ansible all -m ping
```

where,

- **all** is the target hosts
- **hosts** is the host lists
- **ping** is the module (i.e., action) to execute

Default structure of an Ansible's project

- /playbooks
  - files
    - static file$_1$
    - static file$_2$
    - ⋮
    - static file$_n$
  - templates
    - template file$_1$.j2
    - template file$_2$.j2
    - ⋮
    - template file$_n$
  - ansible.cfg .2 hosts .2 playbook$_1$.yml
  - playbook$_2$.yml
  - ⋮
  - playbook$_n$.yml

Ansible's script: Playbook

Playbook

A playbook is the term that Ansible uses for describe a configuration management script. In practice, it is a list of plays.

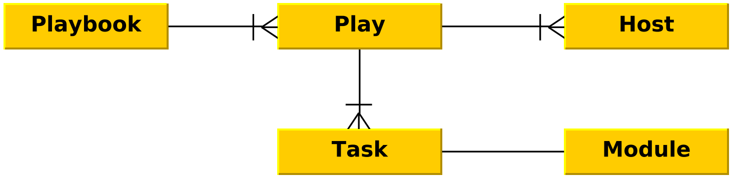

Figure 3: Representing Ansible's playbook elements

Ansible's script: Playbook

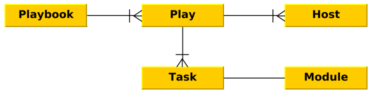

Figure 4: Representing Ansible's playbook elements

What is a play?

A play associates an unordered set of hosts with an ordered list of tasks. A play must have:

- a set of hosts to configure
- a lits of tasks to be executed in the hosts

Additionally, a play may also have:

- a **name**: a comment that describes what the play is about
- **become** and **become_method**: tell Ansible if the tasks must be executed as root
- **vars** a list of variables and values to be used in the play.

Ansible's script: Playbook

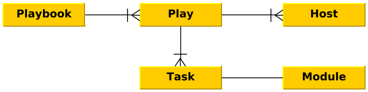

Figure 5: Representing Ansible's playbook elements

What is a task?

A task is the action to execute on the host. Every task must contain:

- a **key** with the name of a module
- a **value** with the arguments of the module.

```
tasks:
  - name: install ngix
    apt: name=nginx update_cache=yes cache_valid_time=3600
```

Playbooks support the usage of variables

- Variables can be used in tasks and in templates files.
- A variable's value can be: string, boolean, lists, and dictionaries.
- To reference a variable, we use the brace {{ var_name }} notation.
- Ansible uses Jinja2[2]template engine to evaluate variables in playbooks and in template files.

```
vars:
    cert_file: /etc/nginx/ssl/nginx.crt
tasks:
  - name: copy the TLS certificate
    copy: src=nginx.crt dest={{ cert_file }}
```

- Ansible also allows us to put the variables into one or more files, and reference them through the **vars_files** section.

```
vars_files:
    - nginx.yml
tasks:
  - name: copy the TLS certificate
    copy: src=nginx.crt dest={{ cert_file }}
```

Playbooks support the usage of variables

- Variables can be defined at runtime, using the **register** clause when executing a task.
- In this case, the type of the variable is always a dictionary, and its keys depend of the modules.
- the debug task can be used to know the value of a variable.

```
tasks:
  - name: register the output of whoami command
    command: whoami
    register: login_user
  - debug: msg="Logged as user: {{ login_user.stdout }}"
```

Built-in variables

Name	Description
hostvars	a dictionary whose keys are Ansible hostnames and values are dictionaries that map variables names to values.
inventory_hostname	name of the current host as defined in the inventory file.
group_names	a list of all groups that the current node is member of
groups	a dictionary whose keys are Ansible group names and values are list of hostnames that are member of the group.
play_hosts	a list of the hostnames of the current play.
ansible_version	a dictionary with the Ansible's version.

- Ansible can also provide information about the node, such as IP addresses, memory size, disk, operating system, etc.

```
---
 - name: collect the name of the user and the facts about the
     node
   hosts: webservers
   gather_facts: yes
   tasks:
     - name: register the output of whoami command
       command: whoami
       register: login_user
     - debug: msg="Logged as user {{ login_user.stdout }} and my
        IP address is {{ hostvars[groups['webservers'][0]]['
        ansible_eth1']['ipv4']['address'] }}"
```

Using Handler to notify a new state

- A handler is similar to a tasks, but it only runs if it has be notified by a task.
- A task only fires a notification only if the node's state has changed.
- Handlers only run after all of the tasks have finished, and they only run once, even if they are notified multiple times.
- Handlers always run in the order that they appear in the play, and not in the notification order.

```
tasks:
  - name: copy the TLS key
    copy: src=files/nginx.key dest={{ key_file }} owner=root
        mode=0600
    notify: restart nginx
handlers:
  - name: restart nginx
    service: name=nginx state=restarted
```