Mastering

ANSIBLE

Section 06

Managing large environments

Reusing playbooks

- As the number of servers under your control gets larger and larger, you will need to minimize the time and code you use while managing them.
- Ansible has a number of ways to let you organize and reuse parts of playbooks that you use more often than others.
- For example, let's assume that you have an NTP server in your environment that you want all the machines to point to when syncing time.
- This involves installing the NTP client on the machine, and adding the necessary configuration lines then starting the service. This has to be done on each new machine configuration regardless of its function (web server, database, application server…etc.).
- Using the `include` phrase, you can isolate the code that installs, configures, and starts the NTP client in a separate file that you can call whenever you are configuring a new machine. Examine the following lab.

LAB: configuring an NTP client

- The first thing to do is to write the necessary tasks to install and configure and NTP client. I am going to use time.nist.gov as the NTP server to point to.

- Create a new file, ntp_client.yml and add the following to it:

```
- name: Install NTP client
  yum:
    name: ntp
    state: present
- name: Configure NTP client
  template:
    src: templates/etc/ntp.conf.jn2
    dest: /etc/ntp.conf
- name: Perform initial time synchronization
  command: "ntpdate {{ ntp_server }}"
- name: Start the service
  service:
    name: ntpd
    state: started
    enabled: yes
```

- Now create the main playbook, `generic.yml` and add the following lines to it:

```
- hosts: all
  tasks:
    - include: ntp_client.yml
      become: yes
      vars:
        ntp_server: time.nist.gov
```

- You should be having an `ntp.conf.jn2` template file in `templates/etc/`, containing the desired configuration with the NTP server's name or IP. This file can be found in the project files directory.

- Notice how we referenced the NTP server in the playbook as a variable. This allows you to change it only in once place and without having to edit the `ntp_client.yml` file.

- Now you can drop `nap.conf.jn2` and `ntp_client.yml` in any playbook directory, reference the file using include clause and you're done.

- Include files works the same as any playbook. You can even add include clauses inside the include file to make your Ansible files modular.

Include on demand

- Sometimes you may want to include a set of tasks only when a condition is met, just like when we used the when clause to configure a DNS server entry only when the hosts file did not contain the required information.

- Let's do the same task but using an include file. Create a new file dns.yml and add the following:

```
- name: Use the DNS server if no entry for database server is found in the hosts file
  become: yes
  shell: 'echo nameserver "{{ dns_server }}" > /etc/resolv.conf'
```

- Then change generic.yml to be as follows:

```
- hosts: web
  tasks:
    - name: Print the output of /etc/hosts and store it in a variable
      shell: cat /etc/hosts
      register: hosts_file_contents
    - include: dns.yml
      become: yes
      vars:
        dns_server: 192.168.33.111
      when: hosts_file_contents.stdout.find('192.168.33.30') == -1
```

- Notice here how we are calling the include file only if needed by using the when clause.

Other forms of include

- Included files are not limited to sets of tasks; you can include handlers or complete playbooks.
- For example, if you have a pretty complex handler you can abstract it to a separate file and call it when needed. Let's abstract the logic of `restarting Apache:`

```
  handlers:
    include: restart_apache.yml
```

- The `restart_apache.yml` contains the same code that was in the handlers' section.

```
- hosts: all
  become: yes
  tasks:
   - name: Ensure that NTP is installed
     yum: name=ntp state=present
- include: web.yml
- include: db.yml
```

- The previous playbook contained aspects that would be applied to all hosts: becoming root and ensuring that NTP is installed. Then more specific tasks can be applied to each server by including the corresponding playbook. The included playbook has the same format as any normal one.

Include files and more complex scenarios

- Includes is a great way to organize and abstract your infrastructure code. However, sometimes and even when using them, things can get more complex.
- In our LAMP example, we had instructions for deploying Apache and others for installing PHP 7. For modularity, you could put each set of those in an include file (apache.yml and php7.yml for example).
- But later on, the web server might also need Wordpress, Joomla, and phpMyAdmin. You put each set of tasks in a separate file for organization and usability. However, each time you need to install a web server you'll have to manually reference all those files.
- Additionally, your playbook may depend on other playbook as prerequisites. For example, non of our applications can be deployed without PHP available on the server. So their should be a mechanism informing the user that PHP7 playbook should be executed first before attempting to deploy any PHP-based application.
- This will become a more common need when you want to share your playbook among your friends or colleagues in different environments.
- For such scenarios you'd better use Ansible Roles.

What are roles?

- They are a way by which Ansible logically groups instructions of common purpose. In our example, a web server in our environment should contain Apache, PHP7, Wordpress, Joomla, and phpMyAdmin.
- A role is no more than a couple of directories: meta and tasks. Ansible will look for and execute a file named main.yml. Those can be placed either in your current directory (where the playbook resides) or in a globally configured roles directory. This can be set in `/etc/ansible/ansible.cfg` by setting `roles_path = /path/to/roles`. You can add multiple places separated by commas and Ansible will look in each path in order.
- Then you can call the role inside your main playbook as:

```
roles:
  - role_name
```

- It is worth noting that there are a lot of Ansible Roles created by the community and are available for free on Ansible Galaxy (covered later).
- You can use `ansible-galaxy init` command followed by the role name to automatically create the required files and directories for your role. The command ensures that the role can then by further shared with the community through Ansible Galaxy.

LAB: creating an application server role

- Since our "web server" will be hosting more than Apache, PHP, and CodeIgniter framework, it's better be called an application server. Create a new directory called appserver and add two subdirectories: tasks and meta. Inside each one there should be an empty main.yml file.

- The meta/main.yml file should only contain the following lines:
  `dependencies: []`
  This means that your playbook does not require any dependencies for the time being.

- Create a bootstrap playbook, let's call it appserver.yml. Add the following lines:

```
- hosts: web
  become: yes
  vars_files:
    - db_vars.yml
  roles:
    - appserver
  pre_tasks:
    - name: Create developers group
      group:
        name: developers
        state: present
    - name: Create cidev user
      user:
        name: cidev
        comment: CodeIgniter user
        group: developers
        state: present
```

- Now head on to tasks/main.yml and add the following instructions:

```
- name: Ensure that Apache is installed
  yum:
    name: httpd
    state: present
- name: Ensure that Apache is started and enabled
  service:
    name: httpd
    state: started
    enabled: yes
- name: Install PHP 7 most common packages
  yum: name={{ item }} state=present
  with_items:
    - mod_php71w
    - php71w-cli
    - php71w-common
    - php71w-gd
    - php71w-mbstring
    - php71w-mcrypt
    - php71w-mysqlnd
    - php71w-xml
- name: Install Rsync
  yum: name=rsync state=present
- name: Deploy and configure CodeIgniter
  synchronize:
      src: /vagrant/CodeIgniter-3.1.4/
      dest: /var/www/html/
```

- Notice here that we have omitted some of the instructions that were in the original lamp.yml playbook like the handlers, and the post-tasks. This is only going to install PHP 7 and deploy CodeIgniter, without any further work.

- Run this playbook as usual by running `ansilble-playbook appserver.yml`. Notice that the tasks now are prefixed by `appserver:`, which is the name of the role.

Completing the example

- Our role missed important parts of the LAMP deployment. That is because we needed to use some of Ansible's functions that are handled differently in Roles. Let's see.

- First, we used handlers. Roles allow you to use handlers when they are added to a file called main.yml in a directory called handlers. Let's create one and add our `handlers:`

```
- name: restart Apache
  service: name=httpd state=restarted
```

- Now we can add the rewrite module configuration part. Add the following to tasks/main.yml:

```
- name: Ensure that mod_rewrite is enabled
  lineinfile:
    path: /etc/httpd/conf.modules.d/00-base.conf
    regexp: '^.*rewrite_module.*$'
    line: 'LoadModule rewrite_module modules/mod_rewrite.so'
    state: present
  notify:
    - restart Apache
```

- Now we need to add the post-tasks section. But notice that this section contained some of the advanced features like templates, files, and it also used variables. Let's see how Ansible Roles handle those functions.

Files, templates and variables

- Files and templates can be used as they are in their locations, or they can be placed inside the roles directory. Placing them inside the directory helps make your roles self-contained.
- Move the templates and files directories to be inside appserver directory. Now you can reference the files and templates the same way as you did before.
- Add the post-tasks section to the appserver.yml file. Execute the playbook and ensure that no errors are reported.
- Now notice how we used the variables file the same way we did before. Actually Roles do have their own variables file. It can be placed in a main.yml file that resides under a directory called vars.
- There is also a directory that can be placed in the roles directory called defaults. The main.yml file inside this directory can also contain variables. But why have two variable locations and what is the difference?
- The defaults/main.yml file contains the default variables; variables that can be easily overridden by those inside vars/main.yml like for example the default installation path for an application. When you are using a role that was not written by you, you can easily override any variables inside it by specifying your own.
- The defaults/main.yml variables are the least in the precedence order, followed by vars/main.yml and then any variables you define in the playbook itself. That way you can customize your role without having to change any of its original content.

Introducing Ansible Galaxy

- Ansible Galaxy refers to the public repository that hosts roles written by the community. Anyone who writes a role that solves a configuration problem can share it with others on Galaxy. It is the same as GitHub but for Ansible.
- You can visit the Galaxy on its web page at galaxy.ansible.com, where you can browse for different roles. Make sure you read the documentation of each role before attempting to use it to make sure that it is suitable for you specially the OS family that it supports.
- Once you choose a role to use, you run `ansible-galaxy install` followed by the role name. Multiple roles can be downloaded at the same time, just separate them by spaces. For example: `ansible-galaxy install apache php7`
- The role will be downloaded by default to/etc/ansible/roles directory so you might want to add sudo to the command if you don't want to change the destination directory (more on that later).
- You also have some useful `ansible-galaxy` subcommands like `list` to view the installed roles, `remove` to delete an installed role and `init` to scaffold a new role that will be contributed to the Galaxy.
- Downloading roles and viewing their content is an excellent way to learn more about Ansible by examining other people's code and, possibly, creating forked versions containing new features, bug fixes, or enhancements. You can contribute to the Galaxy with your enhanced version of the playbook (you'll need to create an account for this. It's free.)

Working with multiple roles

- The ansible-galaxy install command alone is sufficient to install a handful of roles in one go. However, sometimes you may need a dozed or more, some of which as dependencies for one another.
- Additionally, Ansible Galaxy may not be your only source of Ansible roles. You can use GitHub or a similar repository or even your own HTTP server.
- For that reason, Ansible offers a YAML file that you can create and pass on to ansible-galaxy command. The file contains the roles that you need to install, optionally with the source URL, version and other meta data.
- For example, a `reqs.yml` can contain the following:

```
- src: darthwade.wordpress
  version: 1.0
  path: /vagrant
- src: https://github.com/bennojoy/nginx
- src: http://www.example.com/example_role.tar.gz
```

- Then you can use a command like `ansible-galaxy install -r reqs.yml` to download all the roles in the list.

LAB: Installing Wordpress using a community-written role

- In this lab are going to deploy Wordpress 4.7 on our web server but using a ready-made role that is downloaded from Galaxy.

- First, we need to download the role. Issue the following command: `sudo ansible-galaxy install darthwade.wordpress`. We are using sudo since we want this role to be installed in the default directory `/etc/ansible/roles`, which needs admin privileges.

- Once downloaded, we need to create a playbook that would reference the role and also make any necessary prerequisites available. Create a new file, wordpress.yml and add the following:

```
- hosts: db
  tasks:
  - mysql_db:
      name: wordpress
      state: present
      login_user: root
      login_password: adminpassword
  - mysql_user:
      name: wordpress
      password: wordpress
      priv: 'wordpress.*:ALL'
      host: '%'
      login_user: root
      login_password: adminpassword
      state: present
- hosts: web
  become: yes
  vars:
    wp_install_dir: /var/www/html/wordpress
    wp_version: 4.7
    wp_db_name: wordpress
    wp_db_user: wordpress
    wp_db_password: wordpress
    wp_db_host: 192.168.33.30
  pre_tasks:
    - name: Add a www-data user for Wordpress
      user:
        name: www-data
        state: present
  roles:
  - darthwade.wordpress
```

- The prerequisites needed are: create a new database for wordpress, create a new user with the necessary privileges on the database, and create a service account www-data on the web server to own the files.