Mastering

ANSIBLE

Section 05

Advanced playbook

More about handlers

- In the previous section we saw how we can use handlers to list tasks that can be triggered from other tasks using the `notify` parameter.
- You can also have more than one handler notified at the same time since notify accepts a list of tasks. For example, the previous lab could have something like this:

```
- name: Enable caching
  # Change a configuration file using lineinfile for example
  notify:
    - restart Apache
    - restart nginx
```

- The same way, handlers can call other handlers. So for example you can force nginx (which might be acting as a reverse proxy) to get restarted whenever apache is. This can be done as follows:
  handlers:

```
-name: restart apache
 service: name=httpd state=restarted
 notify:
   - restart nginx
```

- Note that handlers will only run when they get notified. They always run at the end of the playbook and they can never run more than once.

More ways to add variables

- We've seen before how we can add variables to the playbook either by putting them in the file itself under the `vars` section, or by placing them in a separate file(s) and pointing to it using `vars_file`.
- If you have only one or two variables that need to be used only once, you needn't add them to the playbook; they can be put on the command line itself. For example, `ansible-playbook myvars.yml --extra-vars "name=ahmad"`
- The same method can be applied if you need to use a variables file only once. Consider the following: `ansible-playbook myvars.yml --extra-vars "@vars.json"` where `vars.json` is a JSON file containing the needed variables. Ansible can use a YML or a JSON file for variables.
- Finally, you may prompt the user for variables-input. This can be done through vars_prompt clause. The following lab demonstrates a use-case for this.

LAB: asking for the database password

- At the end of this section, we'll discuss how Ansible can store sensitive information (like passwords) in Ansible's Vault. However, what if the person who configures Ansible is different from the one who owns the password? And what if this password cannot be shared by any means? The only solution for this is to ask the password owner for the password whenever the playbook runs.

- Let's write a playbook that connects to the database using a password that its entered by the user. The playbook should look as follows:

```
- hosts: db
  vars_prompt:
    - name: dbpass
      prompt: "Enter the database password"
  tasks:
    - name: Upload create.sql
      become: yes
      copy:
        src: /vagrant/create.sql
        dest: /tmp/
    - name: connect to mysql
      mysql_db:
        name: appdb
        login_user: root
        login_password: '{{ dbpass }}'
        state: import
        target: /tmp/create.sql
```

- Notice that - by default - any text entered by the user will not by echoed to the screen. If you want to override this behavior add private: no in the vars_prompt stanza.

- Notice also that any variables defined in vars_prompt will be overridden by command line variables (through `--extra-vars`). Ansible will not prompt the user for variables if the playbook is not run interactively (like through a cron job). For non-sensitive variables using prompt, you can use the `default` parameter to specify a default value.

Conditionals

- In some situations, you may need to execute a task only if a condition is true. For example, I may not want to upgrade a system that hosts an application which is tightly coupled with the specific kernel version.

- In this case, you can use the when clause. It accepts the traditional logical operations like ==, !=, >, <, >=, <=. In our case, we can write the task as follows:

```
- hosts: web
  tasks:
    - name: Upgrade the kernel
      become: yes
      yum:
        name: kernel
      when: ansible_kernel != "3.10.0-514.21.1.el7.x86_64"
```

- Notice here the use of the special variable ansible_kernel. This is one of several variables that automatically created by Ansible to reflect different aspects of the remote system. You don't quote those variables with curly braces. For a list of possible variables you can run `ansible hostname -m setup`

- The `when` clause can be used with multiple conditions at the same time when added as a list. For example:

```
when:
 - ansible_kernel != "3.10.0-514.21.1.el7.x86_64"
 - ansible_os_family == "RedHat"
```

Storing command output in variables

- Sometimes you may want to examine the result of a specific command to decide the way the rest of the playbook will be executed.
- Let's say you need to ensure that the /etc/hosts file on the web server contains an entry for the database server. If it doesn't, we are going to use a DNS server for name resolution.
- This can be done using a playbook as follows:

```
- hosts: web
  vars:
    dns_server: 192.168.33.100
  tasks:
    - name: Print the output of /etc/hosts and store it in a variable
      shell: cat /etc/hosts
      register: hosts_file_contents

    - 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'
      when: hosts_file_contents.stdout.find('192.168.33.30') == -1
```

- Here we used the `register` clause to store the output of /etc/hosts in a variable called `hosts_file_contents`. Then we used it later with the when clause to examine its contents using Python's `find` method to test whether or not it contains an entry for the database server. Accordingly, we can reference a DNS server for name resolution.
- Notice the use of the `shell` module here. Similar to the `command` module, it allows you to execute arbitrary commands against the remote server but it is more suitable for milt-line commands and scripts.
- Once the variable has the command output, you can access either the standard output or the standard error of the command by using the `stdout` or `stderr` respectively as in the above example.

Manual failure reporting

- In a perfect world, any command that exits successfully should return an exit status of 0. Otherwise, it should return a non-zero exit code (1,2,…etc.) that indicates that it did not finish successfully. Additionally, the exit code number should indicate the type of error that happened.
- However, in the real world, that is not always the case. Sometimes, commands do not exit successfully and they just print out a message indicating that some error has happened.
- When working with automation, the exit status is very important as it indicates whether or not the main script should continue after each of its commands is finished, depending on its exit status.
- Ansible can handle commands that do not return the correct exit status by examining the output of the command (using `register`), and using the `failed_when` phrase to trigger a failure preventing the the rest of the playbook from running. The following lab will examine this.

LAB: handling incorrect exit status

- First, let's write a simple script that requires to be run only by the root user. If another user tries to run it, it will fail. However, it will provide an exit status of 0. It'll print an error message: `"You must be root to run this script. Command failed"`. The script can be found in the project files directory.

- Then let's write a playbook that will upload this script to the remote machine, runs it, and examines its output. If it finds the string "failed" then the rest of the playbook will not continue and an error message will be displayed to the user. Depending on the way the script was written, you may want to check the standard output or the standard error for the error message. The playbook may look as follows:

```
- hosts: web
  tasks:
    - name: Upload the script
      copy:
        src: /vagrant/admin.sh
        dest: /home/vagrant/
        mode: 0755
    - name: Run admin script
      command: /home/vagrant/admin.sh
      register: admin_result
      failed_when: "'failed' in admin_result.stderr"
    - name: Report command ran successfully
      copy:
        dest: /home/vagrant/result.txt
        content: "admin.sh ran successfully"
```

Choosing when to report changes

- I guess you have noticed by now that whenever Ansible runs an arbitrary command on the remote system, it always reports that a change has happened.
- If you want a report about which machines where changes and which were not affected, this behavior may be misleading. For that reason, Ansible provides the `changed_when` clause. It works similarly as `failed_when`.
- Consider the following example: You are using `pip` (Python's own package manager) to install `requests`. You want to ensure that Ansible reports a change only if `pip` installs the package. If the package is already installed, no change should be reported. The problem is, whenever `pip` is run Ansible will report that a change occurred regardless of the command output.
- Actually there is an Ansible module specifically for `pip` that will handle this for you. It can be found on http://docs.ansible.com/ansible/pip_module.html. However, we're acting as if this module does not exist is that we can see the power of `changed_when` clause.
- In fact, `changed_when` is most commonly used with third-party package managers (the ones that are not OS-native) like `pip`, `composer`, `npm`, `gem` and others. In the following lab we'll examine how we can install `requests` on the remote machine, instructing Ansible to report a changed only when `requests` gets actually installed.

LAB: Install requests and report a change only once

- The only prerequisite for this lab is to ensure that pip is installed on the remote machine. Then we use changed_when to examine the output of the pip command. It will report a change when the string "Downloading" appears in the output, indicating a change. The playbook should look as follows:

```
- hosts: web
  tasks:
    - name: Install pip
      become: yes
      yum:
        name: python2-pip
        state: present
    - name: Install requests using pip
      become: yes
      command: pip install requests
      register: pip_output
      changed_when: "'Downloading' in pip_output.stdout"
```

- Now try running this playbook a number of times. You should see that Ansible reports a change on the remote system on the first run only. Subsequent runs do not report any changes; as requests is already installed ('Downloading' does not appear in the command output).

Running playbooks locally

- Typically, Ansible works by being installed on control machine and communicating with one more remote machines for configuration management.
- But sometimes, perhaps for testing purposes, you may want to run it on the local machine. This is as easy as setting the `hosts` clause to `127.0.0.1`.
- However, this will not prevent Ansible from initiating an SSH session with localhost needlessly. To avoid that, add the following flag to the end of ansible-playbook command: `--connection=local`. Let's run our current playbook against the client machine. Issue the following command: `ansible-playbook /vagrant/myvars.yml --connection=local`

Adding variables to the inventory file

- So far we've seen several places/ways to use variables in Ansible. They can be placed inside the playbook, in a separate file, or passed as a command-line argument at execution time.
- But sometimes you may wish to limit variables to a specific host or set of hosts (group). One way to do this is to add them directly in the inventory file.
- For example, you may want to define a specific application user for the webserver, and another application user for the database server. Since both of them will be using the same variable name (appuser), you can do it inside the inventory file as follows:

```
[web]
192.168.33.20 appuser=cidev
[db]
192.168.33.30 appuser=dbuser
```

- Then we can create a generic playbook that will ensure that there exists an application user on both servers, each of which is having the appropriate name.
- You can also add variables to whole groups. Let's say that you have more than one web server and you need to have the civdev user created on all the servers. You can add the variable to the group as follows:

```
[web:vars]
appuser=cidev
```

The group_vars and host_vars directories

- Placing variables inside the inventory file might be convenient when using very few variables. However, as the number grows you will quickly find that it's less practical.
- Ansible allows you to create two special directories: `group_vars` for placing group-level variables, and `host_vars` for placing host-level ones.
- Those directories can exist in your current directory, under `/etc/ansible` (where the inventory file exists by default), or under the directory where you place your inventory file in case you are not using the default one.
- For example, inside `group_vars` you can create a `web.yml` file (after the name of the group) and add whatever variables inside. They will be available to any machine in the web group.
- Similarly you can create a `webserver.yml` file under host_vars directory to contain variables specific to the webserver machine.
- Finally, if you want to apply variables to all groups, you can add them to an `all.yml` file under the `group_vars`.

Gathering facts

- A few moments ago you're briefly introduced to the setup module. When using it, it displayed different aspects of the running system like CPU, Memory, IP address and other facts.
- Ansible grabs those facts whenever it is run. You can notice this action in the "Gathering facts" message that gets displayed when you run an Ansible command against a remote system(s).
- However, sometimes you may not need to use this information. Perhaps you are just configuring NTP on a couple of hundreds of servers. Spending one-second gathering facts on each server costs you more than two minutes needlessly.
- Instead, you can disable fact-gathering by setting `gather_facts: no` in the playbook.

Storing sensitive information

- When provisioning our LAMP stack, we needed to store some sensitive data like the database root password and the application user's password.
- This data was stored in plain text format. Anyone who has a copy of the project files can easily gain access to this data, which is not recommended for obvious security reasons.
- You have the option to use a third party key management service like Vault by HashiCorp (www.vaultproject.io), and other well known KMS services, or you can use the built-in vault provided by Ansible.
- Ansible Vault works by encrypting the playbook (or any YML file). You'll need a password for opening and working with the file. This password can be entered interactively or saved in a securely-stored file on the system. In the following lab we'll see how we can encrypt our variables file.

LAB: Encrypt variables file

- In the previous section we've used a variables file to store the usernames and passwords of Mariadb. Since this is not secure, let's encrypt this file.
- Enter the following command to encrypt the file: `ansible-vault encrypt /vagrant/db_vars.yml`
  It will ask you for a password and a confirmation. Make sure you enter a strong password.
- Now if you try to view the contents of the file you'll find that it is no more readable.
- What about working with the file? You can do pretty much to the file without having to decrypt it let's see:
  - View the file contents by running: `ansible-vault view /vagrant/db_vars.yml`
  - Edit the file in-place by running: `ansible-vault edit /vagrant/db_vars.yml`
  - Change the current password for the file by running: `ansible-vault rekey /vagrant/db_vars.yml`
  - Decrypt the file by running `ansible-vault decrypt /vagrant/db_vars.yml`
- You can even create new encrypted file from the beginning by running `ansible-vault create encrypted.yml`

- But about using the encrypted variables file while running the playbook? Enter the following command `ansible-playbook /vagrant/lamp.yml` you will receive an error message that Ansible could not decrypt the file.

- That is because there are two different ways to supply the password for Ansible-Vault-encrypted files and you have to choose one. Enter the previous command but this time with `--ask-vault-pass` flag at the end. Now you are asked for the file password. Having entered it correctly, the playbook will run normally as it did before.

- Sometimes you may not want to run playbooks interactively. This means that Ansible should retrieve the password from the file. Make sure that this file is stored in a secure location with the appropriate restrictive permissions.

- First, create a `~/.ansible/pass.txt` file and write the password inside it.

- Now you can reference this file to be used when decrypting files by running the following command: `ansible-playbook /vagrant/lamp.yml --vault-password-file ~/.ansible/pass.txt`

Running playbooks partially

- As your playbooks get larger, you may want to segment it to different parts and run them selectively. This can be done in Ansible using tags.
- For example, let's re-examine the database playbook. We had the following tasks: download and install mariadb, create a database user, configure the database root password, and create the application user and database.
- May be you don't want to run all those tasks. By adding tags to the playbook, you can select which task(s) to run or which to skip.
- Have a look at the `tags.yml` playbook located in the project files directory.
- As you can see, you can place tags as part of the task. You can add one tag on the same line or multiple tags by putting them in a YAML list.
- Once done, you can choose which tasks to run by specifying their tag(s). For example: `ansible-playbook tags.yml --tags "dummy_records"` or `ansible-playbook tags.yml --tags "createdb,dummy_records"`.
- You can also skip tasks by referencing their tags as follows: `ansible-playbook tags.yml --skip-tags "root"` to skip setting the root password; as it's been already set.

Excepting handling

- All modern programming languages have some sort of exception handling. That is, a piece of code that runs whenever an error occurs during execution. This code usually gives a friendly error message to the user.
- It is often used with another piece of code that will always gets executed whether or not an exception was raised. It usually contains instructions for closing network or database connections. Sometimes, exception handling is used to ignore errors and let the program *continue* despite the raised exception.
- In Ansible, we have a similar method of exception handling. It can be applied by using blocks. A block is a logical grouping of tasks so that multiple parameters can be applied to all the tasks within the group.
- Let's create a copy of `tags.yml` playbook, call it `blocks.yml` (available in the project files directory) and organize the task that set the root password and the one that removes anonymous users in a block. We can now add a flag (variable) called `firstrun` and set it to `false`.
- Using the when clause, we can check the value of `firstrun`. When true, the block shall run setting the root password and removing anonymous users.

- You can add any of the parameters you learned about to a block. For example: `become`, `with_items`, `when` … etc. and they will be applied to all the tasks in the block.
- Now let's say an error has occurred while executing the block. This is managed by the `rescue` stanza. You can also add the `always` stanza, where tasks will always run whether or not an execution has happened.
- Back to our example, have a look at `exceptions.yml` file. We need to set the root password (whether or not it has been already set) to a new password and in all cases we need to remove anonymous users from the database.
- This can be done by adding the task that configures the root password in a `block`. If the password was not already set, then the new password is going to be assigned to root and the `rescue` stanza will not be executed.
- If an error occurs (most probably because the root user is already set and we need to use it for logging in) then the `rescue` stanza will be executed. It will first login with the old password and then change it to the new one.
- In all cases, the root password will be changed (whether initially or using the old password). Let's use this new password to login and remove the anonymous users accounts from the database. This task can be placed in the `always` stanza.
- Finally, you may want to just ignore any errors raised when trying to set the root password (if it is already set). This can be done by using `ignore_errors=yes`, which can be added to a single task or a block.