Linux Academy

Ansible Playbooks

Study Guide

Stosh Oldham

stosh@linuxacademy.com

February 6, 2019

Contents

Playbook Basics

Using YAML for Ansible Playbooks

- The goal of this section is to cover YAML specifics with regard to Ansible Playbooks.
- Be sure to check out Linux Academy's *YAML Essentials* course for greater detail on YAML.
- Best practice dictates that YAML files open with 3 hyphens, `---`, and end with 3 periods, `...`, as observed below.

```
---
concentration: DevOps
# List of courses
courses:
- Ansible
- Openshift
- Configuration Management
- Containerized Application Development
...
```

- List members start with a single hyphen followed by a space. Each list item should be at the same indentation level. See the course list above.
- Dictionaries are key value pairs that are designated with a colon and a space.
    - Example:

```
course: title: Ansible level:
professional id: 123456
```

- Multiple Line Values
- There are two characters that can be used to indicate a multi-line value: `|` or `>`
    - `|` will not ignore newlines in the input.
        - Example:

```
ports: |
9001
9002
9003
```

    - Is interpreted as follows:

```
9001
9002
9003
```

    - `>` will ignore newlines in the input.

    - Example:

```
Ports: >
9001
9002
9003
```

  - Is interpreted as: `9001 9002 9003`

- When to Quote:
  - If a colon ends a line or is followed by a space, the values should be wrapped in double quotes.
- Special characters meant to be literals and should always be wrapped in double quotes. YAML Special characters are `[ ] { } : > |`
- It should be noted that variables in Ansible are an exception to the special character rule.
  - As Ansible variables are indicated with curly braces, `{{ variable }}`, they must be wrapped in double quotes to prevent interpretation as a dictionary.
  - Example: `port: "{{ web_port }}"`
- Booleans are automatically converted in Ansible, thus allowing one to use yes, no, true, false, etc.
  - This means if you want something like a literal "yes" or "false", you must use double quotes.
- Floating point numbers are taken as numbers.
  - Sometimes you may prefer them to be a string (as in a version number).
  - In this case, you should use double quotes.

Creating an Ansible Play

- What is a **Play** in Ansible?
  - A set of instructions expressed in YAML.
  - It targets a host or group of hosts.
  - It provides a series of tasks (basically `ad-hoc` commands) that are executed in sequence.
  - The goal of a *play* is to map a group of hosts to a well defined role.
  - *Plays* are kept in files known as *Playbooks*.
- Writing a Playbook:
  - Each Playbook contains one ore more plays.
  - Each play starts by designating a target which may be a host or group of hosts.
    - Example:

```
---
- hosts: webservers
```

- After the target is defined, a number of options may be set.
  - **remote_user** – System user to execute the play (if not the current user).
  - **become** – If yes, Ansible will escalate permission to the **become_user** using **become_method**.
  - **gather_facts** – Whether or not facts should be gathered (default: yes).
    - Example:

```
---
- hosts: webservers
  become: yes
  remote_user: ansible
  gather_facts: yes
```

- The next section of a play is the *tasks* section:
  - This section contains a list of modules that will be executed against the target(s).
  - Each task maps to the use of an Ansible module.
    - Example:

```
---
- hosts: webservers
  become: yes
  remote_user: ansible
  gather_facts: yes
  tasks:
    - name: ensure httpd is installed
      package:
        name: httpd
        state: latest

    - name: ensure httpd is started
      service:
        name: httpd
        state: started
```

    - More detail on tasks and options will be provided as the course progresses!
- Best Practices:
  - Comments are important!
    - Comments are indicated within the playbooks using the hash mark, **#**.
    - Break up plays with white space and lead plays with comments for easier understanding of a playbook.
  - Keep it Simple.
    - Try to keep plays as straightforward as possible.
    - Avoid subtly.
    - Remain consistent in feature applications.
- Be careful with indentation!
  - Many playbook errors are the result of improper indentation.

The `ansible-playbook` Command

- **Playbooks** must be executed using the `ansible-playbook` command.
- The `ansible-playbook` command takes a few basic arguments:
  - The inventory file to use (use `-i` flag).
  - The playbook to execute.
  - Example: `ansible-playbook -i production site.yml`
    - This command executes the playbook `site.yml` using the inventory stored in the file `production`.
  - Notable options:
    - `-K` — (note capital) Asks for the *become* password.
    - `-k` — (note lowercase) Asks for the *connect* password.
    - `-C` — Run in check mode which is an effective dry run of the provided playbook.

Understanding Playbook Tasks

- As covered earlier, tasks are essentially the use of Ansible modules within a play.
- Tasks are presented in list form (each list element starts with a hyphen `-`) beginning with the name property.
- The name property is simply a plain English statement describing what the task does.
- The module to be used is provided on the next line followed by a colon.
- If applicable, each argument that is provided to the module follows line by line in the format `argument: value`.
  - Example:

```
tasks:
- name: install elinks
  package:
    name: elinks
    state: latest
```

- Best Practices:
  - Name your tasks!
    - The provided name is displayed on task execution which provides insights to those running the plays.
    - The name also serves as basic documentation within the playbook.

Essential Playbook Syntax

Using Variables in Playbooks

- Typical uses of variables:
  - Customize configuration values.
  - Hold return values of commands.
  - Ansible has many default variables for controlling Ansible's behavior.
- Variable names should be letters, numbers, and underscores.
- Variables should always start with a letter.
- Examples of valid variable names:
  - `foobar`
  - `foo_bar`
  - `foo5`
- Examples of invalid variable names:
  - `foo-bar`
  - `1foobar`
  - `foo.bar`
- Variables can be scoped by group, host, or within a playbook.
- Variables may be used to store a simple text or numeric value.
  - Example: `month: January`
- Variables may also be used to store simple lists.
  - Example:

```
colors:
- red
- blue
- yellow
```

- Additionally, variables may be used to store python style dictionaries.
  - A dictionary is a list of key value pairs.
  - Example:

```
person:
  name: sam
  age: 4
  favorite_color: green
```

- Variables may be defined in a number of ways:
  - Via command line argument.
  - Within a variables file.
  - Within a playbook.
  - Within an inventory file.
- How to defined variables via the command line:
  - Use the `--extra-vars` or `-e` flag defined within a playbook.
  - CLI Example: `ansible-playbook service.yml -e "target_hosts=localhost target_service=httpd"`
- Defining variables within a playbook:
  - Playbook Example:

```
---
- hosts: webservers
become: yes
vars:
  target_service: httpd
  target_state: started
tasks:
  - name: Ensure target service is at target state
    service:
      name: "{{ target_service }}"
      state: "{{ target_state }}"
```

- Note: Variables are referenced using double curly braces.
- It is good practice to wrap variable names or statements containing variable names in double quotes.
  - Example: `hosts: "{{ my_host_var }}"`
- Variables may also be stored in files and included using the `vars_file` directive.
  - Example variable file:

```
# file: /home/ansible/web_vars.ini
target_service: httpd
target_state: started
```

  - Example Playbook:

```
---
- hosts: webservers
become: yes
vars_files:
  /home/ansible/web_vars.ini
tasks:
  - name: Ensure target service is at target state
    service:
      name: "{{ target_service }}"
      state: "{{ target_state }}"
```

- The `register` module is used to store task output in a dictionary variable.
    - It essentially can save the results of a command.
    - Several attributes are available: return code, stderr, and stdout.
    - Example:

```
- hosts: all
  tasks:
  - shell: cat /etc/motd
    register: motd_contents

  - shell: echo "motd contains the word hi"
    when: motd_contents.stdout.find('hi') != -1
```

Working with Templates

- Templates are files with Ansible variables inside that are substituted on play execution.
- Templates use the template module.
    - Module parameters
        - **src** – Template file to use.
        - **dest** – Where the resulting file should be on the target host.
        - **validate** – A command that will validate a file before deployment.
        - Can also manipulate result file properties (owner, permissions, etc).
        - Example:

```
---
- hosts: webservers
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest

  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
```

- Notes regarding template files:

- They are essentially text files that have variable references.
- They use Jinja2 templating.
- They tend to be identified by using the file extension `.j2`.
- A typical use case is a skeleton configuration file where variables may be used for simple customizations (such as IP addresses or host names).

Using Ansible Facts

- Ansible facts are simply various properties regarding a given remote system.
- The `setup` module can retrieve facts.
  - The `filter` parameter takes regex to allow you to prune fact output.
- Facts are gathered by default in Ansible playbook execution.
  - The keyword `gather_facts` may be set in playbook to change fact gathering behavior.
- It is possible to print Ansible facts in files using variables.
- Facts may be filtered using the setup module `ad-hoc` by passing a value for the `filter` parameter.
- It is possible to use `{{ ansible_facts }}` for conditional plays based on facts.

Conditional Execution in Playbooks

- Ansible playbooks are capable of making actions conditional.
- The `when` keyword is used to test a condition within a playbook.
  - Jinja2 expressions are used for conditional evaluation.
  - Example using facts:

```
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
```

  - It is also possible to use module output conditionally:

```
- hosts: web_servers
tasks:
  - shell: /usr/bin/foo
    register: foo_result
    ignore_errors: True

  - shell: /usr/bin/bar
    when: foo_result.rc == 5
```

Using Loops in Ansible

- The `loop` keyword may be used to more concisely express a repeated action.
  - Example:

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

- `loop` may also operate with a list variable.
  - Example:

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

- It is also possible to combine loops and conditionals:
  - Example:

```
- name: install software on debian systems
  apt:
    name: "{{ item }}"
    state: latest
  loop: "{{ packages }}"
  when: ansible_os_family == "Debian"
```

Working with Handlers in Ansible

- Ansible provides a mechanism that allows an action to be flagged for execution when a task performs a change.
  - By only executing certain tasks on change, plays are more efficient.
  - This mechanism is known as a `handler` in Ansible.
  - A `handler` may be called using the `notify` keyword.
  - No matter how many times a `handler` is flagged in a play, it is only ran one time at the final phase of play execution.

- `notify` will only flag a `handler` if a task block makes changes.
- Example:

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

- The calls made in the notify section correspond to handler definitions within the play.
- A `handler` may be defined similarly to tasks:
- Example:

```
handlers:
- name: restart memcached
  service:
    name: memcached
    state: restarted
  listen: "restart cache service"

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

Advanced Playbook Syntax

Executing Selective Parts of a Playbooks

- Ansible allows for both plays and tasks to be tagged.
- By tagging a play or task, you may run a playbooks in such a way as to only run plays or tasks with a particular tag.
- Alternatively, you may also skip certain tags during execution.
- Note: Tasks can be tagged the same.
- Example:

```
tasks:
- name: install software
  yum:
    name: "{{ item }}"
```

```
    state: installed
  loop:
    - httpd
    - memcached
  tags:
    - packages

- name: install conf file
  template:
    src: templates/src.j2
    dest: /etc/foo.conf
  tags:
    - configuration
```

- You specify which tags to run or not run via arguments to the `ansible-playbook` command.
- Run certain tag CLI syntax: `ansible-playbook all playbook.yml --tags "pacakges"`
- Skip tag CLI syntax `ansible-playbook all playbook.yml --skip-tags "configuration"`

Working with Sensitive Data using Ansible Vault

- The `ansible-vault` command is used to encrypt files and work with those files.
- It can take a number of sub-commands:
  - **encrypt** to protect a file: `ansible-vault encrypt <file>`
  - **rekey** to change the password of an already encrypted file: `ansible-vault rekey <file>`
  - **view** to output the contents of an encrypted file: `ansible-vault view <file>`
  - **edit** to edit an encrypted file:`ansible-vault edit <file>`
  - **unencrypt** will unencrypt an encrypted file: `ansible-vault unencrypt <file>`
  - **encrypt_string** will encrypt a string: `ansible-vault encrypt_string 'encrypted text goes here'`
- An file encrypted with `ansible-vault` is called a **vault** in Ansible parlance.
- The primary use case for **vaults** is for encrypting variable files to protect sensitive information such as passwords.
- It is also possible to encrypt task files or even arbitrary files such as binaries if desired.
- The `ansible-vault` password file is simply a file that contains a password. There is no special formatting.
- The recommended way to provide a vault password from the CLI is to use `--vault-id`.
  - `--vault-id` may be passed a vault file or a prompt flag (–vault-id@prompt) to collect credentials to unencrypt a target vault.
  - Prior to –vault-id, Ansible could only take a single vault password for a playbook.

- Multiple Vault IDs may be provided and Ansible will try each sequentially to unencrypt as needed.
- Vault IDs also allow for the application of labels to encrypted strings.
- Example:
  - `ansible-vault encrypt_string --vault-id test@my-vault-file 'some secret text' > file.txt`
  - The label 'test' is applied using the password from the vault file `my-vault-file`.
  - In order for Ansible to use the vault-id during playbook execution you must pass `--vault-id test@my-vault-file` with the `ansible-playbook` command.
- Example:
  - Let us say you have a playbook called `site.yml` that makes use of the vault `file.txt`.
  - In order for Ansible to access the `file.txt` vault, you must specify the password file for the vault using the correct vault id.
  - Run: `ansible-playbook --vault-id test@my-vault-file site.yml`
  - Using this command, Ansible will try the password from `my-vault-file` on any string labeled with 'test' before trying any other passwords or vault files.
- You may also specify `@prompt` instead of `label@password_file` to have Ansible prompt for the password.
- Labels are not strictly required.
  - You may use only a password file.
  - Generally, this is not ideal but may have niche use cases.
  - Note: Password files may also be executable (such as a python script).
- Note: when debugging plays, it is possible that sensitive information may be displayed in verbose logs.
- You can set `no_log` for a module to censor log output to avoid accidentally exposing sensitive information during play execution.

Error Handling in a Playbook

- A playbook may be ran against specific hosts and groups out of what is designated within the playbook.
  - Example: `ansible-playbook <playbook> --limit <hostname>`
  - May alternatively specify a list file using `--limit @filename`
  - May also use after playbook failure.
    - When a playbook fails to execute on any host, a file is created containing the names of each host where the playbook failed.
    - This file may be used with the `--limit` flag to execute only against hosts where the playbook failed.
- Ansible may be configured to continue execution even after an error occurs.
- Example: `ignore_errors: yes`
- When set for a task, playbooks will not halt on that task failing.

- Ansible allows failure conditions to be defined.
  - Use the `failed_when` keyword to do this.
  - Allows you to specify the failure condition for a given task.
    - Example:

```
- 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
```

- There is also `changed_when`.
  - This keyword allows overriding what Ansible considers changed.
  - Using a Jinja2 expression on output to create the rule.
  - Example:

```
- name: Run foo process
  shell: /usr/local/foo
  register: foo_result
  changed_when: "foo_result.rc != 2"
```

- The `debug` module may be used to help troubleshoot plays.
  - Use to print detail information about in progress plays.
  - Handy for troubleshooting.
  - Debug takes two primary parameters that are mutually exclusive:
    - **msg** - A message that is printed to STDOUT
    - **var** - A variable whose content is printed to STDOUT.
  - Example:

```
- debug:
  msg: "System {{ inventory_hostname }} has uuid {{ ansible_product_uuid }}"
```

- Error handling may also be dealt with using block groups in Ansible.
  - There are 3 key blocks that may be used to organize tasks:
    - **block** - Group tasks into a 'block'.
    - **rescue** - A special block that is executed when the preceding block fails.
    - **always** - A special block that is always executed after the preceding block.
  - Example:

```
tasks:
- name: Attempt and gracefully roll back demo
  block:
    - debug:
      msg: 'I execute normally'
```

```
    - command: /bin/false
    - debug:
      msg: 'I never execute, due to the above task failing'
rescue:
  - debug:
    msg: 'I caught an error'
  - command: /bin/false
  - debug:
    msg: 'I also never execute :-('
always:
  - debug:
    msg: "this always executes"
```

Asynchronous Tasks within a Playbook

- Some operations may require a significant amount of time to execute.
  - By default, all playbook block tasks ran against a single host use a single SSH session.
  - Ansible provides the **async** feature to allow an operation to run asynchronously such that the status may be checked.
  - This can prevent interruption from SSH timeouts for long running operations.
  - You may configure a few key values for an asynchronous task:
    - A timeout for an operation (default is unlimited).
    - A poll value for who often Ansible should check back.
    - Note: A poll value of 0 will have Ansible not check back on a task.
  - Example:

```
- name: 'Install docker-io (async)'
  yum:
    name: docker-io
    state: installed
  async: 1000
  poll: 25
```

Delegating Playbook Execution

- Certain tasks may need to be executed on specific hosts.
  - This is referred to as delegation in Ansible.
  - By delegating a task, the task will only run on the host or group to which it was delegated.
  - In order to delegate, use the `delegate_to` keyword.
  - Example:

```
- hosts: webservers
  tasks:
  - name: take out of load balancer pool
    command: /usr/bin/take_out_of_pool {{ inventory_hostname }}
    delegate_to: 127.0.0.1

  - name: Update software package
    yum:
      name: acme-web-stack
      state: latest
```

  - In the example, the task, 'take out of load balancer pool', would be ran specifically on 127.0.0.1 an no other host in webservers.
  - You may also use DNS names over IP addresses if preferred.
  - Delegating to localhost may also be expressed in shorthand using `local_action: <module_name> [arg1=val1] ... [argN=valN]` for a given task.

- Parallelism in playbooks:
  - It is possible to control number of host acted upon at once time by Ansible.
  - This may be done using *forks*, which are parallel Ansible processes that execute playbook tasks.
    - The number of forks can be set using `-f` flag with the `ansible` or `ansilbe-playbook` commands.
    - The default number of forks is 5 but can be set in `ansible.cfg`.

- The `serial` keyword may be used to control forks in playbook.
  - You may provide as integer count or as percentage.
  - You may provide a step up approach (can mix and match count with percentage)
  - It is possible to use `max_fail_percentage` to allow a certain percentage to fail (Ansible will still pass play).
    - Note: serial can only be as parallel as the number of set forks will allow.
    - Example:

```
---
- hosts: webservers
  max_fail_percentage: 10
  serial:
    - 1
    - 5
    - "30%"
  tasks:
    -name: Install apache
      yum: name=httpd state=latest
```

Using `run_once`

- There are scenarios where a specific task needs to be ran only a single time in a given playbook and not on each host.
- This may be achieved using the `run_once` keyword.
    - Example:

```
- name: db upgrade task
  command: /opt/application/upgrade_db.py
  run_once: true
```

- This may be leveraged with `delegate_to` for greater control over which host executes the command.
- It should also be noted that when used with `serial` that `run_once` will execute for *each* serial batch.

Overview of Ansible Roles

- Roles provide a way of automatically loading certain var_files, tasks, and handlers based on a known file structure.
- Roles also make sharing of configuration templates easier.
- Roles require a particular directory structure.

```
base_directory/
<role 1>
  tasks/
  handlers/
  files/
  tempaltes/
  vars/
  defaults/
  meta
<role 2>
  tasks/
  defaults/
  meta/
```

- Role definitions must contain at least one of the noted directories
    - **tasks** - Contains the main list of tasks to be executed by the role.
    - **handlers** - Contains handlers, which may be used by this role or even anywhere outside this role.
    - **defaults** - Default variables for the role (see Variables for more information).
    - **vars** - Other variables for the role (see Variables for more information).
    - **files** - Contains files which can be deployed via this role.

    - **templates** - Contains templates which can be deployed via this role.
    - **meta** - Defines some meta data for this role. See below for more details.
- Unused directories need not exist.
- With the exception of **templates** and **files**, each directory must include a `main.yml` if that directory is being used.
- `main.yml` serves as the entry point for the role.
- Files in the **tasks**, **templates**, and **files** directories may be referenced without path within the role.
- To invoke a role in a playbook, you must use the **role** keyword.
    - Example using two roles in a playbook using various keywords as needed:

```
---
- hosts: webservers
  roles:
    - common
    - role: foo_app_instance
      vars:
        dir: '/opt/a'
        app_port: 5000
    - role: foo_app_instance
      vars:
        dir: '/opt/b'
        app_port: 5001
```

- When a role is called in a playbook, Ansible looks for the role definition in `${PWD}/roles/<role_name>`
    - If `${PWD}/roles` does not contain the sought role, then `/etc/ansible/roles` is checked.
    - The default role location may be changed in `ansible.cfg`.
    - The full path to a role may also be specified with the **role** keyword to use a non-default path.
        - Example:

```
---
- hosts: webservers
  roles:
    - role: '/path/to/my/roles/common'
```

    - A role with a given set of parameters will only be applied once even if called multiple times in a play.
        - If a role is called with different parameters, it will be ran again.
        - A role may have `allow_duplicates: true` defined in `meta/main.yml` within the role.
            - This will also allow the role to be applied more than once.
    - Roles may have dependent roles defined in `meta/main.yml` using the dependencies keyword.
        - Parameters may also be included in the dependency list.
        - Dependent roles are applied prior to the role dependent on them.

- Be careful of role duplication with dependencies.
- Example:

```
---
dependencies:
  - role: common
    vars:
      some_parameter: 3
 - role: apache
   vars:
     apache_port: 80
 - role: postgres
   vars:
     dbname: blarg
     other_parameter: 12
```

- Roles and variables
  - There are three primary ways (aside from conventional variable use such as inventory) to interact with variables within a role: vars directory, defaults directory, and parameters.
  - Each way has a different level of precedence.
  - The vars directory defined within the role has the highest level of precedence. (it will override inventory variables as well)
  - The defaults directory has the lowest level of precedents and provides a 'default' value.
  - Parameters are passed inline to the role and sit between vars and defaults in terms of precedents.
  - Example of passing a parameter

```
roles:
- role: apache
  vars:
    http_port: 8080
```

  - Variables defined within a role may be accessed across roles.
  - You may still pass variables on the command line with the -e flag for use in a role. (These variables override all others in terms of precedents.)
  - Best practice dictates that you properly namespace your variables when working with a role to avoid conflicts.
  - An easy way to do this is to prepend your role name to all variable names within the role. (Example: `webserver_timeout` instead of just `timeout`)