Getting Started with Ansible

Formatting, Management, Usage, and Examples

Joshua Hegie

October 9, 2020

Text Conventions

- Ansible keywords and module names will be formatted like this: `keyword`
- References to filesystem paths will be formatted like this: `path`
- Commands and arguments will be formatted like this: `command`
- External links will be formatted like this: URL

YAML Rules

YAML has the following strict formatting rules:

- Files *must* begin with the line "---" and optionally end the same
- Indentations are *exactly* 2 spaces
- A hyphen/dash is always followed by a single space
- A colon (:) is always followed by a space
- A line beginning with a single quote (' ') or double quote (" ") *must* end with the same quote

YAML Rules for Variables

- Variables are encloses in double curly braces (*e.g.* `{{ variable }}`)
- When using a variable in a module's options, if the variable is the beginning of the argument, the whole argument has to be double quoted.
  - `name: "{{ cool_var }}"`
  - `name: "{{ cool_var }}/subdirectory"`
  - `name: topdir/{{ cool_var }}`

Simplification with vim

```
1 autocmd FileType yaml setlocal ai si ts=2 sw=2 sts=-1 et
```

Listing 1: vimrc configuration for YAML files

This sets the following for only YAML files when put in your .vimrc:

- ai: Autoindent; Indent to the depth of the previous line
- si: Smartindent; Attempt to determine the actual indentation for new lines based on context
- ts=2: Tab stop indicates each tab character represents as 2 spaces
- sw=2: Shift width of 2 spaces for autoindent
- sts=-1: Soft tab stop indicates how much whitespace to delete on backspace, in this case a full tab stop
- et: Expand tabs to spaces

Also useful in vim is enabling syntax highlighting by adding `syntax on` to .vimrc

Making it even easier

```
augroup yaml
  autocmd BufNewFile *.yml 0r ~/.vim/templates/skel.yml
augroup end
```

Listing 2: vimrc YAML empty file trigger

```
---
- name: Playbook description
  hosts:
  become: yes
  tasks:
    - name: Task description
```

Listing 3: Contents of YAML skeleton file referenced in Listing 2

Some BASH Shortcuts

```
1 alias ansible-syn="ansible-playbook --syntax-check"
2 alias ansible-check="ansible-playbook -C"
```

Listing 4: Shortened commands for syntax checking and testing

The aliases in Listing 4 can be added to either ∼/.bashrc or ∼/.bash_profile to provide easier access to the syntax check and non-destructive playbook check functionality of the ansible-playbook command.

Anatomy of a play

```
1  ---
2  - name: Play desription
3    hosts: localhost
4    gather_facts: no
5    become: no
6    tasks:
7      - name: Task description
8        debug:
9          msg: "Did a thing"
10
11     - name: Another task description
12       debug:
13         msg: "Did another thing"
```

Listing 5: Example play

From Listing 5 the lines break down as:

- Line 2: Name of the overall play, or collection of tasks
- Lines 3-5: Global options for the play
- Line 6: Indicator that the next indentation will be a list of tasks
- Lines 7-13: A pair of tasks using the debug module

Anatomy of a playbook

```
1  ---
2  - name: This is a play
3    hosts: localhost
4    gather_subset:
5      - min
6    become: no
7    tasks:
8      - name: Task 1 of play 1
9        debug:
10         msg: "{{ ansible_hostname }}: play 1"
11
12 - name: This is also a play
13   hosts: localhost
14   gather_subset:
15     - min
16   become: no
17   tasks:
18     - name: Task 1 of play 2
19       debug:
20         msg: "{{ ansible_hostname }}: play 2"
```

Listing 6: Example playbook with 2 plays

Listings 5 and 6 are both examples of valid playbooks. Listing 5 is a playbook with one play, and Listing 6 is a playbook with two.

Badly formatted example

```
1  ---
2  - name: Badly formatted tasks
3    hosts: localhost
4    become: no
5    gather_subset:
6      - min
7    tasks:
8      - name: Create a file in /tmp
9        file: path=/tmp/foo.{{ ansible_user_id }} mode=0644 owner={{
       ansible_user_id }} state=touch
10
11     - name: Remove a file in /tmp
12       file: path=/tmp/foo.{{ ansible_user_id }} state=absent
```

Listing 7: Poorly formatted playbook

- More compact view
- Difficult to parse module options
- With shell text editor, difficult to update an option

Cleaned up formatting

```
1  ---
2  - name: Cleanly formatted tasks
3    hosts: localhost
4    become: no
5    gather_subset:
6      - min
7    tasks:
8      - name: Create a file in /tmp
9        file:
10         path: /tmp/foo.{{ ansible_user_id }}
11         mode: 0644
12         owner: "{{ ansible_user_id }}"
13         state: touch
14
15     - name: Remove a file in /tmp
16       file:
17         path: /tmp/foo.{{ ansible_user_id }}
18         state: absent
```

Listing 8: Cleaner version of Listing 7

- Less compact view
- Easier to visually parse
- Easier to edit

Task Types

- Tasks: Individual calls to modules. Can be used in the sections: tasks, pre_tasks, post_tasks
- Handlers: Actions that changes in tasks can notify. Can be used in the section "handlers"
- Roles: Collections of tasks, handlers and variables that perform a unified function. Can be used in the section "roles"

Play Task Flow

Each play will process tasks in the following order[1]

1. pre_tasks
2. handlers notified by pre_tasks
3. roles
4. tasks
5. handlers notified by roles and tasks
6. post_tasks
7. handlers notified by post_tasks

[1] An example playbook that demonstrates this flow is located in the `Flow` directory of the examples repo.

Defining What Constitutes a Task

All actions in Ansible plays, handlers, and roles are tasks. A task is best defined as a call to an Ansible module to put the target hosts into a defined state. Example modules that a task might invoke include, but are not limited to: `yum`, `firewalld`, `service`, and `debug`.

Finding Modules

There exists a module for almost every change you might want to make. These can be looked up at on-line here or with the command `ansible-doc -l`.

Example Tasks

```
1  - name: Print a message
2    debug:
3      message: "This is a text output"
4
5  - name: Install httpd
6    yum:
7      name: httpd
8      state: present
9
10 - name: Touch a file
11   file:
12     path: /tmp/foo.{{ ansible_user_id }}
13     state: touch
```

Listing 9: Examples of tasks

Purpose of Handlers

Handlers are tasks which only run when notified by other tasks[2], only when the notifying task has caused a change. Handlers run, not when notified, but at the end of specific phases. A typical use case would be restarting a service if a configuration file is changed, but any action can be taken inside of the task.

[2] Handlers *always* run in the order they are notified in

Defining Idempotence

The definition of idempotence is: "The property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application".

Idempotence in Ansible

```
1 - name: Install httpd package
2   shell: yum install -y httpd
```

Listing 10: Non-idempotent module

```
1 - name: Install httpd
2   yum:
3     name: httpd
4     state: present
```

Listing 11: Idempotent module

Listing 10 *is not* idempotent, because whether or not httpd needs to be installed, a command is run. Listing 11 *is* idempotent, because it checks for the state of the system before taking action. Idempotence is important for handlers, because it is not desirable to always invoke a handler (*e.g.* restarting services) if no change was made. Using the `shell` or `command` modules make it not possible to invoke Ansible in non-destructive check mode (`-C`), and risk unnecessarily restarting services.

Idempotence and Handlers

Handlers are not run unless a task notifies them, since they're actions that might cause outages. Notifications are not sent unless a task makes a change, which makes it important to only send notifications from tasks utilizing idempotent to prevent handlers from running unnecessarily. Examples of handlers will follow in Listings 12 and 13.

Note: The name passed to "notify" is the exact name of the handler task

Playbook with a Handler

```
1  ---
2  - name: Handler Examples
3    hosts: localhost
4    gather_subset:
5      - min
6    become: no
7    tasks:
8      - name: Touch a file
9        file:
10         path: /tmp/foo.{{ ansible_user_id }}
11         state: "{{ item }}"
12       loop:
13         - touch
14         - absent
15       notify: file handler
16
17   handlers:
18     - name: file handler
19       debug:
20         msg: "Notified of a change"
```

Listing 12: An example playbook with a handler that will always fire

Playbook with a Handler

```
---
- name: Handler that never fires
  hosts: localhost
  gather_subset:
    - min
  become: no
  tasks:
    - name: Ensure a file doesn't exist
      file:
        path: /tmp/foo.{{ ansible_user_id }}.{{ 1500 | random }}
        state: absent
      notify: file handler

  handlers:
    - name: file handler
      debug:
        msg: "This will never fire"
```

Listing 13: An example playbook with a handler that will never fire

Purpose of a Role

Roles are collections of tasks, variables, files and templates that aggregately perform a specific unified task. Some of the functionality of a role can be achieved by including additional playbooks as a task action, but that approach fails to be portable and quickly becomes unwieldy. Roles are, instead, created, and interacted with, using the `ansible-galaxy` command.

Role Layout

```
1  Example
2  Example/defaults
3  Example/defaults/main.yml
4  Example/files
5  Example/handlers
6  Example/handlers/main.yml
7  Example/meta
8  Example/meta/main.yml
9  Example/README.md
10 Example/tasks
11 Example/tasks/main.yml
12 Example/templates
13 Example/tests
14 Example/tests/inventory
15 Example/tests/test.yml
16 Example/vars
17 Example/vars/main.yml
```

Listing 14: Layout of a role provisioned with `ansible-galaxy init`

Components of a Role

The items initialized are as follows:

- defaults: Variables that need to be defined for the role to run, but can be overridden
- files: Static files to be deployed by the role
- handlers: Handlers to be invoked by the role
- meta: Meta data about the role
- tasks: Tasks for the roll to run
- templates: Jinja2 templates for the role to deploy
- tests: Tests to use to validate that the role is functioning
- vars: Static variables for the role, not intended to be overridden

Invoking a Role

When invoked by a playbook the `tasks/main.yml` playbook is called, which sources `defaults/main.yml` and `vars/main.yml`. If there are additional variables files, for instance an encrypted vault, the `include_vars` module can be invoked. Similarly, if there are additional playbooks that need to be run `import_tasks` or `include_tasks` can be used to incorporate them[3].

Inside of a role most operations are automatically done from the correct directory (*e.g.* the `file` module uses the files directory as its root), to prevent having hard coded paths.

[3] `import_tasks` preserves tags, `include_tasks` does not

Lists and Dicts

The primary structures for Ansible are lists and dictionaries (dicts). Both store data in an orderly fashion, but that data is accessed differently. In a list the data is accessed by an index, but in a dict you can use the key name to access the data you want directly.

Examples

```
1 list:
2   - item1
3   - item2
4   - item3
```

Listing 15: A simple array

A list is a zero (0) indexed array, see Listing 15 that can be directly indexed into (*e.g.* list[0]).

```
1 dictionary:
2   first:
3     name: Person A
4     email: persona@place.tld
5   second:
6     name: Person B
7     email: personb@place.tld
```

Listing 16: A simple dictionary

A dictionary is more complex data structure that is addressable by keyword. In Listing 16, the primary keys are `first` and `second`, with the fields `name` and `email`. To access the name "Person A", the call would be `dictionary.first.name`

Accessing data in a task

Most commonly, a list will be accessed using the `loop` keyword.

```
---
- name: Example With a Loop
  hosts: localhost
  become: no
  gather_facts: no
  vars:
    list:
      - Thing one
      - Thing two
  tasks:
    - name: Loop over the list
      debug:
        var: item
      loop: "{{ list }}"
```

Listing 17: A simple loop with a predefined list

Accessing Loop Data

Listing 17 iterates over a static list. The variable `item` is the default iterator used by Ansible. Loops can only be applied to individual tasks, and can be used in tasks, handlers, and standard play tasks.

More Complicated Data Structure

```
1  ---
2  - name: More complex example
3    hosts: localhost
4    gather_facts: no
5    become: no
6    vars:
7      complex:
8        - name: Person A
9          email: persona@place.tld
10       - name: Person B
11         email: personb@place.tld
12   tasks:
13     - name: Iterate over a more complex structure
14       debug:
15         msg: "{{ item.name }}: {{ item.email }}"
16       loop: "{{ complex }}"
```

Listing 18: A loop that iterates over a list of dictionaries

Common Global Keywords

The following keywords are typically placed at a global, for the play, scope, but can be used on individual tasks as well:

- `hosts`: Names of hosts or host groups to operate on
- `gather_facts`: Whether to gather facts at the start of the play. Valid responses: yes, no, true, false
- `gather_subset`: Which facts should be gathered. Takes a *list* of valid groups to collect.
- `become`: Whether to run as the user running Ansible or another user. Valid responses: yes, no, true, false
- `become_user`: Which user to switch to on hosts being targeted by the play
- `force_handlers`: Run notified handlers, even if a task fails.
- `ignore_errors`: Do not stop play execution if a task fails. Discussed more in the Error Handling section.

Common Task Keywords

- `loop`: Loop through the list provided. Can be provided a variable or a static list.
- `register`: Store the results of a tasks in the provided variable name.
- `notify`: Notify the specified handler if the tasks returns changed status.
- `tags`: Attach the following tags to the task[4].
- `when`: Only run the task if the specified conditional returns true.
  - Variable names *cannot* be enclosed in curly braces.
  - If multiple conditionals are provided in list format, their results are processed with a logical AND.
  - Spanning operators that can be used:
    - | Resulting text includes newlines and trailing spaces[5].
    - > Resulting text has newlines converted to spaces[6].
- `changed_when`: Only flag the task as changed when the specified conditional returns true.

[4] Will be discussed later
[5] Used for `when`
[6] Used for variables and text blocks

Defining Failure

Ansible has a few ways to allow customization of error handling[7]. The most immediately helpful are `failed_when` and `ignore_errors`.

`failed_when` can only be attached to a task, and is used to set custom conditions for a task to be considered failed. This might be parsing a registered variable for the presence of a text string or specific non-standard return code, or intentionally triggering a failure without the use of the `fail` module.

`ignore_errors`, conversely, allows the error to happen, but does not stop the execution of the play. If this is set to true for a task, only that task will ignore errors. If it's set at the play level, no failed tasks will cause the play to stop.

[7] See here for all methods

The Block Keyword

Ansible has a special module, `block`, that allows groups of tasks to be aggregated. There are a few special caveats to `block`:

- `loop` and `failed_when` are not a valid keywords for a block, but are for tasks inside of the block.
- Tasks inside of a block *must* be indented one level in from the block.
- The keywords `rescue` and `always` *must* be specified after block.

Adding Rescue and Always

`rescue`:

- Intended to remediate failure conditions encountered in a block
- Executed *after* `block`, but *only* if a task inside of the block fails.
- Clears the failed flag for the play, not the task, which is reported as failed at the end of the play.
- Failure inside of `rescue` fails the play.

`always`:

- Runs regardless of failures in `block` or `rescue`
- Does *not* clear any fail states

Block-Rescue Example

```
1  ---
2  - name: Demonstrate Block/Rescue
3    hosts: localhost
4    become: no
5    gather_facts: no
6    vars:
7      blockvar: true
8    tasks:
9      - name: Block of tasks
10       block:
11         - name: Fail on value of blockvar
12           debug:
13             msg: "This always fails if blockvar is true"
14           failed_when: blockvar
15
16         - name: Print a message
17           debug:
18             msg: This always succeeds
19
20       rescue:
21         - name: Clear failure and continue
22           debug:
23             msg: "This only runs if a task in the block fails"
24           failed_when: rescuevar is defined
25
26       always:
27         - name: Always runs
28           debug:
29             msg: "This task always runs"
```

Listing 19: Example of block/rescue structure

Testing Block/Rescue

Using the playbook from Listing 19 the following test cases can be run:

- `ansible-playbook Failure/block-rescue.yml`
  Results in: Task 1 fails, task 2 is not run, task 3 succeeds, task 4 succeeds
- `ansible-playbook Failure/block-rescue.yml -e "blockvar=false"`
  Results in: Task 1 succeeds, task 2 succeeds, task 3 is not run, task 4 succeeds
- `ansible-playbook Failure/block-rescue.yml -e "rescuevar=true"`
  Results in: Task 1 fails, task 2 is not run, task 3 fails, task 4 succeeds

Tags

While most ofter it is desirable to run all tasks in a play or role, there are situations where only a subset of tasks need to be run. In Ansible this is accomplished with tags.

The `tags` keyword can only be applied to tasks, including `block` tasks, and take a list of text tags to apply to the task.

Example with Tags

```
1  ---
2  - name: Demonstrate Tags
3    hosts: localhost
4    become: no
5    gather_subset:
6      - min
7    tasks:
8      - name: Print a message
9        debug:
10         msg: This is an example
11       tags:
12         - debug
13
14     - name: Test for a file
15       file:
16         path: /tmp/foo.{{ ansible_user_id }}.{{ 1500 | random }}
17         state: absent
18       tags:
19         - file
```

Listing 20: A simple playbook with a tagged task

Interpreting Listing 20

Listing 20 provides 2 explicit tags: debug and file

```
1 playbook: Tags/example.yml
2
3   play #1 (localhost): Demonstrate Tags TAGS: []
4       TASK TAGS: [debug, file]
```

Listing 21: Output of `ansible-playbook --list-tags` on Listing 20

To run only the debug task, the playbook can be invoked with:
`ansible-playbook --tags debug Tags/example.yml`[8]

[8] The `--tags` flag takes a comma separated list of tags to execute

Special Considerations for Tags

- Tags applied to a `block` apply to all tasks in that block (`rescue` and `always` included, in addition to tags on the specific tasks.
- The `include_tasks` keyword *does not* preserve tags from the included playbook. If the `include_tasks` task has tags, those tags will apply to all included tasks.
- The `import_tasks` keyword preserves tags from the included playbook. If the `import_tasks` task has tags, those tags will apply to all included tasks, in addition to tags on individual tasks.
- Reserved tags:
  - `always`: Tasks that have the `always` tag will be run regardless of tags specified on the commandline.
  - `never`: Tasks that have the `never` tag will not be run, unless another tag on the task is explicitly called out.

Advanced Tags Example

```
1  ---
2  - name: Advanced tag usage
3    hosts: localhost
4    become: no
5    gather_facts: no
6    tasks:
7      - name: Task always runs
8        debug:
9          msg: This task always runs
10       tags:
11         - always
12
13     - name: This task has to specifically be called out
14       debug:
15         msg: This tags requires the "debug" tag
16       tags:
17         - never
18         - debug
```

Listing 22: Example of tasks tagged with `always` and `never`

Advanced Tags Execution

In Listing 22, if the tag "debug" is not explicitly included (`ansible-playbook --tags debug`), the second task will never be executed. Even if the "debug" tag is included, the first task always runs.

Using the `always` tag is a good way, for example, to ensure that `include_vars` tasks are always run, so that there aren't any missing variables. Similarly, the `never` tag is useful, for example, to attach to debug tasks, allowing normal tasks of a subset to run and optionally have debug outputs provided.

Ansible Commandline Invocation

Ansible, when run from the commandline, will typically be invoked similarly to `ansible-playbook -K play.yml` In this case, the playbook will be executed after prompting for the user's sudo password, which will be used to elevate privileges for tasks. Other common options include:

- `--tags`: Provide a comma separated list of tags. Only tasks with these tags will be executed
- `--ask-vault-pass`: Prompt for a vault password
- `--vault-id`: Similar to `--ask-vault-pass` allows for a password to be prompted for (`--vault-id HPC@prompt`). *Unlike* `--ask-vault-pass`, `--vault-id` can be specified multiple times, allowing vaults to have different passwords.
- `-b`: Escalate privileges on hosts
- `-K`: Prompt for a password to escalate privileges

Additional Useful Commandline Options

- `-l`: Limit the execution to the specified hosts/hostgroups (comma separated). Using this option allows for testing without changing inventories or the `hosts` field in the playbook.
- `-e`: Specify the value for a variable in the format `"variable=value"`
- `-i`: Specify the inventory file to use. Useful in testing roles with the built in tests directory.

Specifying Facts

Fact order of precedence, least to greatest, is as follows:

1. Role defaults
2. Inventory variables
3. Inventory `group_vars/all` followed by `playbook group_vars/all`
4. Inventory `group_vars/*` followed by playbook `group_vars/*`
5. Inventory `host_vars` followed by playbook `host_vars`
6. Discovered facts
7. Play variables: `vars` specified in the play header, followed by prompted variables (`vars_prompt`[9]), followed by loaded variable files (`vars_files`)
8. Role variables
9. Block variables followed by task variables (*i.e.* vars specified on the block or task)
10. Included variables (`include_vars`)
11. Play facts created with `set_facts`
12. Commandline extra variables (`-e`)

[9] `vars_prompt` is *not* compatible with Ansible Tower

Useful Variable Filters

Filters[10] are operations that can be applied on variables, below are

- `default`: Provides a default for when a variable might not be defined:
  `when: maybevar|default(false)`
  - A special value to `default` is `omit`. Setting the default value to omit will cause the parameter with the undefined variable to be omitted from the task.
- `union`: Join to lists together preserving all elements from both lists.
  - `intersect`: Similar to `union`, but only produces the unique values from each list
  - `difference`: Similar to `union`, but only produces items from the source list not in the intersected list
- `flatten`: Takes a list that might contain further lists and makes it a one dimensional:
  `{{ [1, [2, 3]] | flatten }}`
- `ipaddr`: Test if a variable is a valid IP:
  `{{ '192.168.1.1/32' | ipaddr }}`

[10] A full list of filters can be found at https://docs.ansible.com/ansible/latest/user_guide/playbooks_filters.html

An example list union

```
1  ---
2  - name: Sample tasks with filters
3    hosts: localhost
4    become: no
5    gather_subset:
6      - min
7    vars:
8      list1:
9        - /tmp/foo.{{ ansible_user_id }}
10       - /tmp/bar.{{ ansible_user_id }}
11     list2:
12       - /tmp/{{ ansible_user_id }}.foo
13       - /tmp/{{ ansible_user_id }}.bar
14   tasks:
15     - name: Create files
16       file:
17         path: "{{ item }}"
18         state: touch
19       loop: "{{ list1 | union(list2) }}"
20
21     - name: Clean up files
22       file:
23         path: "{{ item }}"
24         state: absent
25       loop: "{{ list1 | union(list2) }}"
```

Listing 23: Using filters to join lists

An example with defaults

```
1  ---
2  - name: Tasks that use defaults
3    hosts: localhost
4    become: no
5    gather_subset:
6      - min
7    vars:
8      files:
9        - path: /tmp/foo.{{ ansible_user_id }}
10       - path: /tmp/bar.{{ ansible_user_id }}
11         perms: "0644"
12       - path: /tmp/absent
13         state: directory
14   tasks:
15     - name: Set state on files
16       file:
17         path: "{{ item.path }}"
18         mode: "{{ item.perms|default(omit) }}"
19         state: "{{ item.state|default('absent') }}"
20       loop: "{{ files }}"
21
22     - name: Clean up
23       file:
24         path: "{{ item.path }}"
25         state: absent
26       loop: "{{ files }}"
```

Listing 24: Using filters for defaults

Managing Vaults

Some facts need to be encrypted (*e.g.* passwords and authentication tokens). The Ansible solution for this is vaults. A vault is used to store facts, but can only be accessed by using either `--ask-vault-password` or `--vault-id` with the `@prompt` option. Typical invocations of the `ansible-vault` command are:

- `ansible-vault create vault.yml`
- `ansible-vault edit vault.yml`
- `ansible-vault view vault.yml`
- `ansible-vault rekey vault.yml`[11]

[11] The `rekey` option allows the vault's password to be changed without decrypting the file

Utilizing Vaults

```
1  ---
2  - name: Import a vault and reference a variable
3    hosts: localhost
4    become: no
5    gather_facts: no
6    vars_files:
7      - encrypted.yml
8    tasks:
9      - name: Print the vault data
10       debug:
11         msg: 'This variable was encrypted: {{ vault_var }}'
```

Listing 25: An example playbook that loads a vault

Invocation of Listing 25 would be invoked with:

```
ansible-playbook --vault-id example@prompt vaults.yml
```

Creating and Baselining Roles

Before creating a role, the repository for it needs to be provisioned. For local repositories the command will be one of the following:

- Mercurial: `hg init role`
- Git: `git init --bare --shared=true role`[12]

The empty repository, once cloned locally, must be initialized with `ansible-galaxy init --force role`[13] The newly initialized role should be committed to a changeset (`hg addremove && hg commit -m "Role Initialized"` or `git -a -m "Role Initialized"`)

[12] Git requires bare provisioned repositories for users to be able to push change sets
[13] The force option is required because there's a .hg or .git directory in checked out repo

Sane Ignore Files

```
1 .*.sw[a-z]
2 tests/roles/*(^yml)
3 .*.retry
```

Listing 26: A default `.hgignore` or `.gitignore` file for a role

The ignore patterns shown in Listing 26 will exclude vim swap files, and will not pick up any roles installed in the tests directory.

Components of a Role (Review)

The items initialized are as follows:

- defaults: Variables that need to be defined for the role to run, but can be overridden
- files: Static files to be deployed by the role
- handlers: Handlers to be invoked by the role
- meta: Meta data about the role
- tasks: Tasks for the roll to run
- templates: Jinja2 templates for the role to deploy
- tests: Tests to use to validate that the role is functioning
- vars: Static variables for the role, not intended to be overridden[14]

[14] Any variable files (*e.g.* vaults) other than `main.yml` will have to be manually included with `include_vars`

Ansible Projects

An Ansible project is a repository that contains:

- The playbook to run
- Optionally a directory named "roles" that only contains the file `requirements.yml`

Under Ansible Tower, the project repo will be cloned, and if there's a requirements file it will be processed to deploy any roles required by the project.

Example Requirements File

```
1 ---
2 - name: httpd
3   scm: git
4   src: https://github.com/jhegie/httpd
```

Listing 27: An example requirements file

From Listing 27:

- Line 2: The name that the role will be deployed as (what will be called in the playbook)
- Line 3: Which source control system the module is baselined with (supports hg and git)
- Line 4: The path to the repository, http and local filesystem. If no scm is specified, the src target is assumed to be a tar file.[15]

Lines 2-4 can be repeated as needed for additional modules.

[15] There's an implicit assumption in `ansible-galaxy` that it's possible to anonymously clone the repository, either with auth tokens or as an anonymous account

Role Dependencies

Role dependencies can be provided in a role's `meta/main.yml` file, with the `dependencies` variable. By default `dependencies = []` (*aka* an empty list), but if the empty list notation (`[]`) is removed, dependencies can be specified with the same syntax as `requirements.yml`. What makes the dependencies helpful is:

- Dependencies are installed when `ansible-galaxy install` is run
- When the role is invoked by a playbook, the dependencies are run before the role. This allows a role that modifies files deployed by another role to bundle the entire process into a single call.

An example of this can be found in `role/role2` in the examples. That example is intended to be run from the `role/role2/tests` directory and will require `ansible-galaxy install` be invoked to install the role.

Code Possibly Requiring Refactor

There are a number of valid ways to write playbooks, but not all conform to best practices. An example of such a playbook is show in Listing 28:

```
1  ---
2  - name: Bad Playbook
3    hosts: localhost
4    gather_facts: no
5    become: no
6    vars:
7      demovar: This is a variable
8    tasks:
9      - shell: /bin/ls /tmp
10
11     - name: Print a variable
12       debug:
13         msg: "{{demovar}}"
```

Listing 28: A valid, but badly written playbook

Result of running Listing 28

Running the playbook from Listing 28 produces the following results:

```
1
2  PLAY [Bad Playbook] ************************************************************
3
4  TASK [shell] *******************************************************************
5  changed: [localhost]
6
7  TASK [Print a variable] ********************************************************
8  ok: [localhost] => {
9      "msg": "This is a variable"
10 }
11
12 PLAY RECAP *********************************************************************
13 localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

Listing 29: The results of running Listing 28

The ansible-lint Command

The command `ansible-lint` can be used to identify code that might need to be revisited for corrections. For example, running it against the Listing 28, produces the following output:

```
1  301 Commands should not change things if nothing needs doing
2  src/Lint/bad.yml:9
3  Task/Handler: shell /bin/ls /tmp
4
5  305 Use shell only when shell functionality is required
6  src/Lint/bad.yml:9
7  Task/Handler: shell /bin/ls /tmp
8
9  502 All tasks should be named
10 src/Lint/bad.yml:9
11 Task/Handler: shell /bin/ls /tmp
12
13 206 Variables should have spaces before and after: {{ var_name }}
14 src/Lint/bad.yml:13
15         msg: "{{demovar}}"
```

Listing 30: `ansible-lint` output based on Listing 28

5 Accessing Examples and Copyable Links

Examples

- Examples can be cloned via:

  ```
  git clone https://github.com/jhegie/ansible-examples.git
  ```

URLS

Referenced URLS:

- Ansible Module List

  https://docs.ansible.com/ansible/latest/modules/modules_by_category.html

- Error Handling

  https://docs.ansible.com/ansible/latest/user_guide/playbooks_error_handling.html