Appendix B - Ansible Best Practices and Conventions

Ansible's flexibility allows for a variety of organization methods and configuration syntaxes. You may have many tasks in one main file, or a few tasks in many files. You might prefer defining variables in group variable files, host variable files, inventories, or elsewhere, or you might try to find ways of avoiding variables in inventories altogether.

There are few *universal* best practices in Ansible, but this appendix contains helpful suggestions for organizing playbooks, writing tasks, using roles, and otherwise build infrastructure with Ansible.

In addition to this appendix (which contains mostly observations from the author's own daily use of Ansible), please read through the official Ansible Best Practices[206] guide, which contains a wealth of hard-earned knowledge.

Playbook Organization

Playbooks are Ansible's bread and butter, so it's important to organize them in a logical manner for easier debugging and maintenance.

Write comments and use `name` liberally

Many tasks you write will be fairly obvious when you write them, but less so six months later when you are making changes. Just like application code, Ansible playbooks should be documented so you spend less time familiarizing yourself with what a particular task is supposed to do, and more time fixing problems or extending your playbooks.

[206] http://docs.ansible.com/playbooks_best_practices.html

In YAML, write comments by starting a line with a hash (`#`). If the comment spans multiple lines, start each line with `#`.

It's also a good idea to use a `name` for every task you write, besides the most trivial. If you're using the `git` module to check out a specific tag, use a `name` to indicate what repository you're using, why a tag instead of a commit hash, etc. This way, whenever your playbook is run, you'll see the comment you wrote and be assured what's going on.

```
- hosts: all

  tasks:

    # This task takes up to five minutes and is required so we will
    # have access to the images used in our application.
    - name: Copy the entire file repository to the application.
      copy:
        src: [...]
```

This advice assumes your comments actually indicate what's happening in your playbooks! I use full sentences with a period for all comments and `name`s, but it's okay to use a slightly different style. Just be consistent, and remember, *bad comments are worse than no comments at all.*

Include related variables and tasks

If you find yourself writing a playbook over 50-100 lines and configuring three or four different applications or services, it may help to separate each group of tasks into a separate file, and use `import_tasks` or `include_tasks` to place them in a playbook (see Chapter 6 for details about when to use which syntax).

Additionally, variables are usually better left in their own file and included using `vars_files` rather than defined inline with a playbook.

```
- hosts: all

  vars_files:
    - vars/main.yml

  handlers:
    - import_tasks: handlers/handlers.yml

  tasks:
    - import_tasks: tasks/init.yml
    - import_tasks: tasks/database.yml
    - import_tasks: tasks/app.yml
```

Using a more hierarchical model like this allows you to see what your playbook is doing at a higher level, and also lets you manage each portion of a configuration or deployment separately. I generally split tasks into separate files once I reach 15-20 tasks in a given file.

Use Roles to bundle logical groupings of configuration

Along the same lines as using included files to better organize your playbooks and separate bits of configuration logically, Ansible roles supercharge your ability to manage infrastructure well.

Using loosely-coupled roles to configure individual components of your servers (like databases, application deployments, the networking stack, monitoring packages, etc.) allows you to write configuration once, and use it on all your servers, regardless of their role.

You'll probably configure something like NTP (Network Time Protocol) on every single server you manage, or at a minimum, set a timezone for the server. Instead of adding two or three tasks to every playbook you manage, set up a role (maybe call it `time` or `ntp`) to do this configuration, and use a few variables to allow different groups of servers to have customized settings.

Additionally, if you learn to build robust and generic roles, you could share them on Ansible Galaxy so others use them and help you make them even better!

Use role defaults and vars correctly

Set all role default variables likely to be overridden inside `defaults/main.yml`, and set variables likely never to be overridden in `vars/main.yml`.

If you have a variable that needs to be overridden, but you need to include it in a platform-specific vars file (e.g. one vars file for Debian, one for RHEL), then create the variable in `vars/[file].yml` as `__varname`, and use `set_fact` to set the variable at runtime if the variable `varname` is not defined. This way playbooks using your role can still override one of these variables.

For example, if you need to have a variable like `package_config_path` that is defaulted to one value on Debian, and another on RHEL, but may need to be overridden from time to time, you can create two files, `vars/Debian.yml` and `vars/RedHat.yml`, with the contents:

```
---
# Inside vars/Debian.yml
__package_config_path: /etc/package/package.conf
```

```
---
# Inside vars/RedHat.yml
__package_config_path: /etc/package/configfile
```

Then, in the playbook using the variable, include the platform-specific vars file and define the final `package_config_path` variable at runtime:

```
---
# Include variables and define needed variables.
- name: Include OS-specific variables.
  include_vars: "{{ ansible_os_family }}.yml"

- name: Define package_config_path.
  set_fact:
    package_config_path: "{{ __package_config_path }}"
  when: package_config_path is not defined
```

This way, any playbook using role can override the platform-specific defaults by defining `package_config_path` in its own variables.

YAML Conventions and Best Practices

YAML is a human-readable, machine-parseable syntax that allows for almost any list, map, or array structure to be described using a few basic conventions, so it's a great fit for configuration management. Consider the following method of defining a list (or 'collection') of widgets:

```
widget:
  - foo
  - bar
  - fizz
```

This would translate into Python (using the `PyYAML` library employed by Ansible) as the following:

```
translated_yaml = {'widget': ['foo', 'bar', 'fizz']}
```

And what about a structured list/map in YAML?

```
widget:
  foo: 12
  bar: 13
```

The resulting Python:

```
translated_yaml = {'widget': {'foo': 12, 'bar': 13}}
```

A few things to note with both of the above examples:

- YAML will try to determine the type of an item automatically. So `foo` in the first example would be translated as a string, `true` or `false` would be a boolean, and `123` would be an integer. Read the official documentation for further insight, but for our purposes, declaring strings with quotes (`''` or `""`) will minimize surprises.
- Whitespace matters! YAML uses spaces (literal space characters—*not* tabs) to define structure (mappings, array lists, etc.), so set your editor to use spaces for tabs. You can use either a tab or a space to delimit parameters (like `apt: name=foo state=present`—either a tab or a space between parameters), but it's preferred to use spaces everywhere, to minimize errors and display irregularities across editors and platforms.
- YAML syntax is robust and well-documented. Read through the official YAML Specification[207] and/or the PyYAMLDocumentation[208] to dig deeper.

YAML for Ansible tasks

Consider the following task:

```
- name: Install foo.
  apt: name=foo state=present
```

All well and good, right? Well, as you get deeper into Ansible and start defining more complex configuration, you might start seeing tasks like the following:

[207] http://www.yaml.org/spec/1.2/spec.html
[208] http://pyyaml.org/wiki/PyYAMLDocumentation

```
- name: Copy Phergie shell script into place.
  template: src=templates/phergie.sh.j2 dest=/opt/phergie.sh \
  owner={{ phergie_user }} group={{ phergie_user }} mode=755
```

The one-line syntax (which uses Ansible-specific `key=value` shorthand for defining parameters) has some positive attributes:

- Simpler tasks (like installations and copies) are compact and readable. `apt: name=apache2 state=present` and `apt-get install -y apache2` are similarly concise; in this way, an Ansible playbook feels very much like a shell script.
- Playbooks are more compact, and more configuration is be displayed on one screen.
- Ansible's official documentation follows this format, as do many existing roles and playbooks.

However, as highlighted in the above example, there are a few issues with this `key=value` syntax:

- Smaller monitors, terminal windows, and source control applications will either wrap or hide part of the task line.
- Diff viewers and source control systems generally don't highlight intra-line differences as well as full line changes.
- Variables and parameters are converted to strings, which may or may not be desired.

Ansible's shorthand syntax is troublesome for complicated playbooks and roles, but luckily there are other ways to write tasks which are better for narrower displays, version control software and diffing.

Three ways to format Ansible tasks

The following methods are most often used to define Ansible tasks in playbooks:

Shorthand/one-line (`key=value`)

Ansible's shorthand syntax uses `key=value` parameters after the name of a module as a key:

```
- name: Install Nginx.
  yum: name=nginx state=present
```

For any situation where an equivalent shell command would roughly match what I'm writing in the YAML, I prefer this method, since it's immediately obvious what's happening, and it's highly unlikely any of the parameters (like `state=present`) will change frequently during development.

Ansible's official documentation generally uses this syntax, so it maps nicely to examples you'll find from Ansible, Inc. and many other sources.

Structured map/multi-line (`key: value`)

Define a structured map of parameters (using `key: value`, with each parameter on its own line) for a task:

```
- name: Copy Phergie shell script into place.
  template:
    src: "templates/phergie.sh.j2"
    dest: "/home/{{ phergie_user }}/phergie.sh"
    owner: "{{ phergie_user }}"
    group: "{{ phergie_user }}"
    mode: 0755
```

A few notes on this syntax:

- The structure is all valid YAML, and functions similarly to Ansible's shorthand syntax.
- Strings, booleans, integers, octals, etc. are all preserved (instead of being converted to strings).
- Each parameter *must* be on its own line; multiple variables can't be chained together (e.g. `mode: 0755, owner: root, user: root`) to save space.
- YAML syntax highlighting works slightly better for this format than `key=value`, since each key will be highlighted, and values will be displayed as constants, strings, etc.

Folded scalars/multi-line (>)

Use the > character to break up Ansible's shorthand `key=value` syntax over multiple lines.

```
- name: Copy Phergie shell script into place.
  template: >
    src=templates/phergie.sh.j2
    dest=/home/{{ phergie_user }}/phergie.sh
    owner={{ phergie_user }} group={{ phergie_user }} mode=755
```

In YAML, the > character denotes a *folded scalar*, where every line that follows (as long as it's indented further than the line with the >) will be joined with the line above by a space. So the above YAML and the earlier `template` example will function exactly the same.

This syntax allows arbitrary splitting of lines on parameters, but it does not preserve value types (`0775` would be converted to a string, for example).

While this syntax is often seen in the wild, I don't recommend it except for certain situations, like tasks using the `command` and `shell` modules with extra options:

```
- name: Install Drupal.
  command: >
    drush si -y --site-name="{{ drupal_site_name }}"
    --account-name=admin
    --account-pass=admin
    --db-url=mysql://{{ domain }}:1234@localhost/{{ domain }}
    --root={{ drupal_core_path }}
    creates={{ drupal_core_path }}/sites/default/settings.php
  notify: restart apache
  become_user: www-data
```

Sometimes the above is as good as you can do to keep unwieldy tasks formatted in a legible manner.

Using | to format multiline variables

In addition to using > to join multiple lines using spaces, YAML allows the use of | (pipe) to define literal scalars, to define strings with newlines preserved.

For example:

```
extra_lines: |
  first line
  second line
  third line
```

Would be translated to a block of text with newlines intact:

```
first line
second line
third line
```

Using a folded scalar (>) would concatenate the lines, which might not be desirable. For example:

```
extra_lines: >
  first line
  second line
  third line
```

Would be translated to a single string with no newlines:

```
first line second line third line
```

Using `ansible-playbook`

Generally, running playbooks from your own computer or a central playbook runner is preferable to running Ansible playbooks locally (using `--connection=local`), since

Ansible and all its dependencies don't need to be installed on the system you're provisioning. Because of Ansible's optimized use of SSH for remote communication, there is usually minimal difference in performance running Ansible locally or from a remote workstation (barring network flakiness or a high-latency connection).

Use Ansible Tower

If you are able to use Ansible Tower to run your playbooks, this is even better, as you'll have a central server running Ansible playbooks, logging output, compiling statistics, and even allowing a team to work together to build servers and deploy applications in one place.

Specify `--forks` for playbooks running on > 5 servers

If you are running a playbook on a large number of servers, consider increasing the number of `forks` Ansible uses to run tasks simultaneously. The default, `5`, means Ansible will only run a given task on 5 servers at a time. Consider increasing this to 10, 15, or however many connections your local workstation and ISP can handle—this will dramatically reduce the amount of time it takes a playbook to run.

Set `forks=[number]` in Ansible's configuration file to set the default `forks` value for all playbook runs.

Use Ansible's Configuration file

Ansible's main configuration file, in `/etc/ansible/ansible.cfg`, allows a wealth of optimizations and customizations for running playbooks and ad-hoc tasks.

Read through the official documentation's Ansible Configuration File[209] page for customizable options in `ansible.cfg`.

[209] http://docs.ansible.com/intro_configuration.html

Summary

One of Ansible's strengths is its flexibility; there are often multiple 'right' ways of accomplishing your goals. I have chosen to use the methods I outlined above as they have proven to help me write and maintain a variety of playbooks and roles with minimal headaches.

It's perfectly acceptable to try a different approach; as with most programming and technical things, being *consistent* is more important than following a particular set of rules, especially if the ruleset isn't universally agreed upon. Consistency is especially important when you're not working solo—if every team member used Ansible in a different way, it would become difficult to share work very quickly!