**YML – First program**

---

- hosts: droplets

tasks:

- name: Installs nginx web server

apt: pkg=nginx state=installed update\_cache=true

notify:

- start nginx

handlers:

- name: start nginx

service: name=nginx state=started

**Basics: task, package**

**Syntax**

**Variable**

**Condition**

**YAML Basics**

For Ansible, nearly every YAML file starts with a list. Each item in the list is a list of key/value pairs, commonly called a “hash” or a “dictionary”. So, we need to know how to write lists and dictionaries in YAML.

There’s another small quirk to YAML. All YAML files (regardless of their association with Ansible or not) can optionally begin with --- and end with .... This is part of the YAML format and indicates the start and end of a document.

All members of a list are lines beginning at the same indentation level starting with a "- " (a dash and a space):

--- begin

#comment line

Data: list

-d1 : item

-d2

…. End

---

*# A list of tasty fruits*

fruits:

- Apple

- Orange

- Strawberry

- Mango

...

A dictionary is represented in a simple key: value form (the colon must be followed by a space):

*# An employee record*

martin:

name: Martin D'vloper

job: Developer

skill: Elite

More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

*# Employee records*

- martin:

name: Martin D'vloper # key = value

job: Developer

skills:

- python # value

- perl

- pascal

- tabitha:

name: Tabitha Bitumen

job: Developer

skills:

- lisp

- fortran

- erlang

Dictionaries and lists can also be represented in an abbreviated form if you really want to:

---

martin: {name: Martin D'vloper, job: Developer, skill: Elite}

fruits: ['Apple', 'Orange', 'Strawberry', 'Mango']

Ansible doesn’t really use these too much, but you can also specify a boolean value (true/false) in several forms:

create\_key: yes

needs\_agent: no

knows\_oop: True

likes\_emacs: TRUE

uses\_cvs: false

Values can span multiple lines using | or >. Spanning multiple lines using a | will include the newlines. Using a > will ignore newlines; it’s used to make what would otherwise be a very long line easier to read and edit. In either case the indentation will be ignored. Examples are:

include\_newlines: |

exactly as you see

will appear these three

lines of poetry

ignore\_newlines: >

this is really a

single line of text

despite appearances

Let’s combine what we learned so far in an arbitrary YAML example. This really has nothing to do with Ansible, but will give you a feel for the format:

---

*# An employee record*

name: Martin D'vloper

job: Developer

skill: Elite

employed: True

foods:

- Apple

- Orange

- Strawberry

- Mango

languages:

perl: Elite

python: Elite

pascal: Lame

education: |

4 GCSEs

3 A-Levels

BSc in the Internet of Things

That’s all you really need to know about YAML to start writing *Ansible* playbooks.

**Gotchas**

While YAML is generally friendly, the following is going to result in a YAML syntax error:

foo: somebody said I should put a colon here: so I did

windows\_drive: c:

...but this will work:

windows\_path: c:\windows

You will want to quote hash values using colons followed by a space or the end of the line:

foo: "somebody said I should put a colon here: so I did"

windows\_drive: "c:"

...and then the colon will be preserved.

Further, Ansible uses “{{ var }}” for variables. If a value after a colon starts with a “{”, YAML will think it is a dictionary, so you must quote it, like so:

foo: "**{{** variable **}}**"

If your value starts with a quote the entire value must be quoted, not just part of it. Here are some additional examples of how to properly quote things:

foo: "**{{** variable **}}**/additional/string/literal"

foo2: "**{{** variable **}}**\\backslashes\\are\\also\\special\\characters"

foo3: "even if it's just a string literal it must all be quoted"

Not valid:

foo: "E:\\path\\"rest\\of\\path

The same applies for strings that start or contain any YAML special characters [] {} : > | .

Boolean conversion is helpful, but this can be a problem when you want a literal *yes* or other boolean values as a string. In these cases just use quotes:

non\_boolean: "yes"

other\_string: "False"

YAML converts certain strings into floating-point values, such as the string *1.0*. If you need to specify a version number (in a requirements.yml file, for example), you will need to quote the value if it looks like a floating-point value:

version: "1.0"

# Including and Importing

**Topics**

* Including and Importing
  + Includes vs. Imports
  + Importing Playbooks
  + Including and Importing Task Files
  + Including and Importing Roles

## Includes vs. Imports

As noted in Creating Reusable Playbooks, include and import statements are very similar, however the Ansible executor engine treats them very differently.

* All import\* statements are pre-processed at the time playbooks are parsed.
* All include\* statements are processed as they encountered during the execution of the playbook.

Please refer to Creating Reusable Playbooks for documentation concerning the trade-offs one may encounter when using each type.

Also be aware that this behaviour changed in 2.4; prior to that Ansible version only include was available, and it behaved differently depending on context.

New in version 2.4.

## Importing Playbooks

It is possible to include playbooks inside a master playbook. For example:

---

- import\_playbook: webservers.yml

- import\_playbook: databases.yml

The plays and tasks in each playbook listed will be run in the order they are listed, just as if they had been defined here directly.

Prior to 2.4 only include was available and worked for both playbooks and tasks as both import and include.

New in version 2.4.

## Including and Importing Task Files

Use of included task lists is a great way to define a role that system is going to fulfill. A task include file simply contains a flat list of tasks:

*# common\_tasks.yml*

---

- name: placeholder foo

command: /bin/foo

- name: placeholder bar

command: /bin/bar

You can then use import\_tasks or include\_tasks to include this file in your main task list:

tasks:

- import\_tasks: common\_tasks.yml

*# or*

- include\_tasks: common\_tasks.yml

You can also pass variables into imports and includes:

tasks:

- import\_tasks: wordpress.yml wp\_user=timmy

- import\_tasks: wordpress.yml wp\_user=alice

- import\_tasks: wordpress.yml wp\_user=bob

Variables can also be passed to include files using an alternative syntax, which also supports structured variables like dictionaries and lists:

tasks:

- include\_tasks: wordpress.yml

vars:

wp\_user: timmy

ssh\_keys:

- "**{{** lookup**(**'file'**,** 'keys/one.pub'**)** **}}**"

- "**{{** lookup**(**'file'**,** 'keys/two.pub'**)** **}}**"

Using either syntax, variables passed in can then be used in the included files. These variables will only be available to tasks within the included file. See Variable Precedence: Where Should I Put A Variable? for more details on variable inheritance and precedence.

Task include statements can be used at arbitrary depth.

Includes and imports can also be used in the handlers: section; for instance, if you want to define how to restart apache, you only have to do that once for all of your playbooks. You might make a handlers.yml that looks like:

*# more\_handlers.yml*

---

- name: restart apache

service: name=apache state=restarted

And in your main playbook file:

handlers:

- include\_tasks: more\_handlers.yml

*# or*

- import\_tasks: more\_handlers.yml

# Variables

**Topics**

* Variables
  + What Makes A Valid Variable Name
  + Variables Defined in Inventory
  + Variables Defined in a Playbook
  + Variables defined from included files and roles
  + Using Variables: About Jinja2
  + Jinja2 Filters
  + Hey Wait, A YAML Gotcha
  + Information discovered from systems: Facts
  + Turning Off Facts
  + Local Facts (Facts.d)
  + Ansible version
  + Fact Caching
  + Registered Variables
  + Accessing Complex Variable Data
  + Magic Variables, and How To Access Information About Other Hosts
  + Variable File Separation
  + Passing Variables On The Command Line
  + Variable Precedence: Where Should I Put A Variable?
  + Variable Scopes
  + Variable Examples
  + Advanced Syntax

While automation exists to make it easier to make things repeatable, all of your systems are likely not exactly alike.

On some systems you may want to set some behavior or configuration that is slightly different from others.

Also, some of the observed behavior or state of remote systems might need to influence how you configure those systems. (Such as you might need to find out the IP address of a system and even use it as a configuration value on another system).

You might have some templates for configuration files that are mostly the same, but slightly different based on those variables.

Variables in Ansible are how we deal with differences between systems.

To understand variables you’ll also want to dig into Conditionals and Loops. Useful things like the **group\_by** module and the when conditional can also be used with variables, and to help manage differences between systems.

It’s highly recommended that you consult the ansible-examples github repository to see a lot of examples of variables put to use.

For best practices advice, refer to Variables and Vaults in the Best Practices chapter.

## What Makes A Valid Variable Name

Before we start using variables it’s important to know what are valid variable names.

Variable names should be letters, numbers, and underscores. Variables should always start with a letter.

foo\_port is a great variable. foo5 is fine too.

foo-port, foo port, foo.port and 12 are not valid variable names.

YAML also supports dictionaries which map keys to values. For instance:

foo:

field1: one

field2: two

You can then reference a specific field in the dictionary using either bracket notation or dot notation:

foo['field1']

foo.field1

These will both reference the same value (“one”). However, if you choose to use dot notation be aware that some keys can cause problems because they collide with attributes and methods of python dictionaries. You should use bracket notation instead of dot notation if you use keys which start and end with two underscores (Those are reserved for special meanings in python) or are any of the known public attributes:

add, append, as\_integer\_ratio, bit\_length, capitalize, center, clear, conjugate, copy, count, decode, denominator, difference, difference\_update, discard, encode, endswith, expandtabs, extend, find, format, fromhex, fromkeys, get, has\_key, hex, imag, index, insert, intersection, intersection\_update, isalnum, isalpha, isdecimal, isdigit, isdisjoint, is\_integer, islower, isnumeric, isspace, issubset, issuperset, istitle, isupper, items, iteritems, iterkeys, itervalues, join, keys, ljust, lower, lstrip, numerator, partition, pop, popitem, real, remove, replace, reverse, rfind, rindex, rjust, rpartition, rsplit, rstrip, setdefault, sort, split, splitlines, startswith, strip, swapcase, symmetric\_difference, symmetric\_difference\_update, title, translate, union, update, upper, values, viewitems, viewkeys, viewvalues, zfill.

## Variables Defined in Inventory

We’ve actually already covered a lot about variables in another section, so far this shouldn’t be terribly new, but a bit of a refresher.

Often you’ll want to set variables based on what groups a machine is in. For instance, maybe machines in Boston want to use ‘boston.ntp.example.com’ as an NTP server.

See the Inventory document for multiple ways on how to define variables in inventory.

## Variables Defined in a Playbook

In a playbook, it’s possible to define variables directly inline like so:

- hosts: webservers

vars:

http\_port: 80

This can be nice as it’s right there when you are reading the playbook.

## Variables defined from included files and roles

It turns out we’ve already talked about variables in another place too.

As described in Roles, variables can also be included in the playbook via include files, which may or may not be part of an “Ansible Role”. Usage of roles is preferred as it provides a nice organizational system.

## Using Variables: About Jinja2

It’s nice enough to know about how to define variables, but how do you use them?

Ansible allows you to reference variables in your playbooks using the Jinja2 templating system. While you can do a lot of complex things in Jinja, only the basics are things you really need to learn at first.

For instance, in a simple template, you can do something like:

My amp goes to **{{** max\_amp\_value **}}**

And that will provide the most basic form of variable substitution.

This is also valid directly in playbooks, and you’ll occasionally want to do things like:

template: src=foo.cfg.j2 dest=**{{** remote\_install\_path **}}**/foo.cfg

In the above example, we used a variable to help decide where to place a file.

Inside a template you automatically have access to all of the variables that are in scope for a host. Actually it’s more than that – you can also read variables about other hosts. We’ll show how to do that in a bit.

**See also**

**Templating (Jinja2)**

More information about Jinja2 templating

## Jinja2 Filters

Filters in Jinja2 are a way of transforming template expressions from one kind of data into another. Jinja2 ships with many of these. See builtin filtersin the official Jinja2 template documentation.

In addition to those, Ansible supplies many more. See the Filters document for a list of available filters and example usage guide.

## Hey Wait, A YAML Gotcha

YAML syntax requires that if you start a value with {{ foo }} you quote the whole line, since it wants to be sure you aren’t trying to start a YAML dictionary. This is covered on the YAML Syntax page.

This won’t work:

- hosts: app\_servers

vars:

app\_path: **{{** base\_path **}}**/22

Do it like this and you’ll be fine:

- hosts: app\_servers

vars:

app\_path: "**{{** base\_path **}}**/22"

## Information discovered from systems: Facts

There are other places where variables can come from, but these are a type of variable that are discovered, not set by the user.

Facts are information derived from speaking with your remote systems.

An example of this might be the ip address of the remote host, or what the operating system is.

To see what information is available, try the following:

ansible hostname -m setup

This will return a ginormous amount of variable data, which may look like this, as taken from Ansible 1.4 on a Ubuntu 12.04 system

In the above the model of the first harddrive may be referenced in a template or playbook as:

**{{** ansible\_devices.sda.model **}}**

Similarly, the hostname as the system reports it is:

**{{** ansible\_nodename **}}**

and the unqualified hostname shows the string before the first period(.):

**{{** ansible\_hostname **}}**

Facts are frequently used in conditionals (see Conditionals) and also in templates.

Facts can be also used to create dynamic groups of hosts that match particular criteria, see the About Modules documentation on **group\_by** for details, as well as in generalized conditional statements as discussed in the Conditionals chapter.

## Turning Off Facts

If you know you don’t need any fact data about your hosts, and know everything about your systems centrally, you can turn off fact gathering. This has advantages in scaling Ansible in push mode with very large numbers of systems, mainly, or if you are using Ansible on experimental platforms. In any play, just do this:

- hosts: whatever

gather\_facts: no

## Local Facts (Facts.d)

New in version 1.3.

As discussed in the playbooks chapter, Ansible facts are a way of getting data about remote systems for use in playbook variables.

Usually these are discovered automatically by the **setup** module in Ansible. Users can also write custom facts modules, as described in the API guide. However, what if you want to have a simple way to provide system or user provided data for use in Ansible variables, without writing a fact module?

For instance, what if you want users to be able to control some aspect about how their systems are managed? “Facts.d” is one such mechanism.

If a remotely managed system has an /etc/ansible/facts.d directory, any files in this directory ending in .fact, can be JSON, INI, or executable files returning JSON, and these can supply local facts in Ansible. An alternate directory can be specified using the fact\_path play directive.

For instance assume a /etc/ansible/facts.d/preferences.fact:

[general]

asdf=1

bar=2

This will produce a hash variable fact named general with asdf and bar as members. To validate this, run the following:

ansible <hostname> -m setup -a "filter=ansible\_local"

And you will see the following fact added:

"ansible\_local": {

"preferences": {

"general": {

"asdf" : "1",

"bar" : "2"

}

}

}