Variables

Topics

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.

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.

Easy enough, let's move on.

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 *Playbook Roles and Include Statements*, 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.

> **Note**
> ansible allows Jinja2 loops and conditionals in templates, but in playbooks, we do not use them. Ansible playbooks are pure machine-parseable YAML. This is a rather important feature as it means it is possible to code-generate pieces of files, or to have other ecosystem tools read Ansible files. Not everyone will need this but it can unlock possibilities.

Jinja2 Filters

> **Note**
> These are infrequently utilized features. Use them if they fit a use case you have, but this is optional knowledge.

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 filters in the

official Jinja2 template documentation.

In addition to those, Ansible supplies many more. See the *Jinja2 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:


```
"ansible_all_ipv4_addresses": [
    "REDACTED IP ADDRESS"
],
"ansible_all_ipv6_addresses": [
    "REDACTED IPV6 ADDRESS"
],
"ansible_architecture": "x86_64",
"ansible_bios_date": "09/20/2012",
"ansible_bios_version": "6.00",
"ansible_cmdline": {
    "BOOT_IMAGE": "/boot/vmlinuz-3.5.0-23-generic",
    "quiet": true,
    "ro": true,
    "root": "UUID=4195bff4-e157-4e41-8701-e93f0aec9e22",
    "splash": true
},
"ansible_date_time": {
    "date": "2013-10-02",
    "day": "02",
    "epoch": "1380756810",
    "hour": "19",
    "iso8601": "2013-10-02T23:33:30Z",
    "iso8601_micro": "2013-10-02T23:33:30.036070Z",
    "minute": "33",
    "month": "10",
    "second": "30",
```

```
    "time": "19:33:30",
    "tz": "EDT",
    "year": "2013"
},
"ansible_default_ipv4": {
    "address": "REDACTED",
    "alias": "eth0",
    "gateway": "REDACTED",
    "interface": "eth0",
    "macaddress": "REDACTED",
    "mtu": 1500,
    "netmask": "255.255.255.0",
    "network": "REDACTED",
    "type": "ether"
},
"ansible_default_ipv6": {},
"ansible_devices": {
    "fd0": {
        "holders": [],
        "host": "",
        "model": null,
        "partitions": {},
        "removable": "1",
        "rotational": "1",
        "scheduler_mode": "deadline",
        "sectors": "0",
        "sectorsize": "512",
        "size": "0.00 Bytes",
        "support_discard": "0",
        "vendor": null
    },
    "sda": {
        "holders": [],
        "host": "SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-MPT Dual Ultra320 
        "model": "VMware Virtual S",
        "partitions": {
            "sda1": {
                "sectors": "39843840",
                "sectorsize": 512,
                "size": "19.00 GB",
                "start": "2048"
```

```
            },
            "sda2": {
                "sectors": "2",
                "sectorsize": 512,
                "size": "1.00 KB",
                "start": "39847934"
            },
            "sda5": {
                "sectors": "2093056",
                "sectorsize": 512,
                "size": "1022.00 MB",
                "start": "39847936"
            }
        },
        "removable": "0",
        "rotational": "1",
        "scheduler_mode": "deadline",
        "sectors": "41943040",
        "sectorsize": "512",
        "size": "20.00 GB",
        "support_discard": "0",
        "vendor": "VMware,"
    },
    "sr0": {
        "holders": [],
        "host": "IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)",
        "model": "VMware IDE CDR10",
        "partitions": {},
        "removable": "1",
        "rotational": "1",
        "scheduler_mode": "deadline",
        "sectors": "2097151",
        "sectorsize": "512",
        "size": "1024.00 MB",
        "support_discard": "0",
        "vendor": "NECVMWar"
    }
},
"ansible_distribution": "Ubuntu",
"ansible_distribution_release": "precise",
"ansible_distribution_version": "12.04",
```

```
"ansible_domain": "",
"ansible_env": {
    "COLORTERM": "gnome-terminal",
    "DISPLAY": ":0",
    "HOME": "/home/mdehaan",
    "LANG": "C",
    "LESSCLOSE": "/usr/bin/lesspipe %s %s",
    "LESSOPEN": "| /usr/bin/lesspipe %s",
    "LOGNAME": "root",
    "LS_COLORS": "rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:or=40;31;
    "MAIL": "/var/mail/root",
    "OLDPWD": "/root/ansible/docsite",
    "PATH": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "PWD": "/root/ansible",
    "SHELL": "/bin/bash",
    "SHLVL": "1",
    "SUDO_COMMAND": "/bin/bash",
    "SUDO_GID": "1000",
    "SUDO_UID": "1000",
    "SUDO_USER": "mdehaan",
    "TERM": "xterm",
    "USER": "root",
    "USERNAME": "root",
    "XAUTHORITY": "/home/mdehaan/.Xauthority",
    "_": "/usr/local/bin/ansible"
},
"ansible_eth0": {
    "active": true,
    "device": "eth0",
    "ipv4": {
        "address": "REDACTED",
        "netmask": "255.255.255.0",
        "network": "REDACTED"
    },
    "ipv6": [
        {
            "address": "REDACTED",
            "prefix": "64",
            "scope": "link"
        }
    ],
```

```
    "macaddress": "REDACTED",
    "module": "e1000",
    "mtu": 1500,
    "type": "ether"
},
"ansible_form_factor": "Other",
"ansible_fqdn": "ubuntu2",
"ansible_hostname": "ubuntu2",
"ansible_interfaces": [
    "lo",
    "eth0"
],
"ansible_kernel": "3.5.0-23-generic",
"ansible_lo": {
    "active": true,
    "device": "lo",
    "ipv4": {
        "address": "127.0.0.1",
        "netmask": "255.0.0.0",
        "network": "127.0.0.0"
    },
    "ipv6": [
        {
            "address": "::1",
            "prefix": "128",
            "scope": "host"
        }
    ],
    "mtu": 16436,
    "type": "loopback"
},
"ansible_lsb": {
    "codename": "precise",
    "description": "Ubuntu 12.04.2 LTS",
    "id": "Ubuntu",
    "major_release": "12",
    "release": "12.04"
},
"ansible_machine": "x86_64",
"ansible_memfree_mb": 74,
"ansible_memtotal_mb": 991,
```

```
"ansible_mounts": [
    {
        "device": "/dev/sda1",
        "fstype": "ext4",
        "mount": "/",
        "options": "rw,errors=remount-ro",
        "size_available": 15032406016,
        "size_total": 20079898624
    }
],
"ansible_os_family": "Debian",
"ansible_pkg_mgr": "apt",
"ansible_processor": [
    "Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz"
],
"ansible_processor_cores": 1,
"ansible_processor_count": 1,
"ansible_processor_threads_per_core": 1,
"ansible_processor_vcpus": 1,
"ansible_product_name": "VMware Virtual Platform",
"ansible_product_serial": "REDACTED",
"ansible_product_uuid": "REDACTED",
"ansible_product_version": "None",
"ansible_python_version": "2.7.3",
"ansible_selinux": false,
"ansible_ssh_host_key_dsa_public": "REDACTED KEY VALUE"
"ansible_ssh_host_key_ecdsa_public": "REDACTED KEY VALUE"
"ansible_ssh_host_key_rsa_public": "REDACTED KEY VALUE"
"ansible_swapfree_mb": 665,
"ansible_swaptotal_mb": 1021,
"ansible_system": "Linux",
"ansible_system_vendor": "VMware, Inc.",
"ansible_user_id": "root",
"ansible_userspace_architecture": "x86_64",
"ansible_userspace_bits": "64",
"ansible_virtualization_role": "guest",
"ansible_virtualization_type": "VMware"
```

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_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.

> **Note**
> Perhaps “local facts” is a bit of a misnomer, it means “locally supplied user values” as opposed to “centrally supplied user values”, or what facts are – “locally dynamically determined values”.

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.

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"
            }
        }
 }
```

And this data can be accessed in a template/playbook as:

```
{{ ansible_local.preferences.general.asdf }}
```

The local namespace prevents any user supplied fact from overriding system facts or variables defined elsewhere in the playbook.

If you have a playbook that is copying over a custom fact and then running it, making an explicit call to re-run the setup module can allow that fact to be used during that particular play. Otherwise, it will be available in the next play that gathers fact information. Here is an example of what that might look like:

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

```
- name: create directory for ansible custom facts
  file: state=directory recurse=yes path=/etc/ansible/facts.d
- name: install custom impi fact
  copy: src=ipmi.fact dest=/etc/ansible/facts.d
- name: re-read facts after adding custom fact
  setup: filter=ansible_local
```

In this pattern however, you could also write a fact module as well, and may wish to consider this as an option.

Fact Caching

New in version 1.8.

As shown elsewhere in the docs, it is possible for one server to reference variables about another, like so:

```
{{ hostvars['asdf.example.com']['ansible_os_family'] }}
```

With “Fact Caching” disabled, in order to do this, Ansible must have already talked to ‘asdf.example.com’ in the current play, or another play up higher in the playbook. This is the default configuration of ansible.

To avoid this, Ansible 1.8 allows the ability to save facts between playbook runs, but this feature must be manually enabled. Why might this be useful?

Imagine, for instance, a very large infrastructure with thousands of hosts. Fact caching could be configured to run nightly, but configuration of a small set of servers could run ad-hoc or periodically throughout the day. With fact-caching enabled, it would not be necessary to “hit” all servers to reference variables and information about them.

With fact caching enabled, it is possible for machine in one group to reference variables about machines in the other group, despite the fact that they have not been communicated with in the current execution of /usr/bin/ansible-playbook.

To configure fact caching, enable it in ansible.cfg as follows:

```
[defaults]
gathering = smart
fact_caching = redis
fact_caching_timeout = 86400
# seconds
```

You might also want to change the ‘gathering’ setting to ‘smart’ or ‘explicit’ or set gather_facts to False in most plays.

At the time of writing, Redis is the only supported fact caching engine. To get redis up and running, perform the equivalent OS commands:

```
yum install redis
service redis start
pip install redis
```

Note that the Python redis library should be installed from pip, the version packaged in EPEL is too old for use by Ansible.

In current embodiments, this feature is in beta-level state and the Redis plugin does not support port or password configuration, this is expected to change in the near future.

Registered Variables

Another major use of variables is running a command and using the result of that command to save the result into a variable. Results will vary from module to module. Use of -v when executing playbooks will show possible values for the results.

The value of a task being executed in ansible can be saved in a variable and used later. See some examples of this in the *Conditionals* chapter.

While it's mentioned elsewhere in that document too, here's a quick syntax example:

```
- hosts: web_servers

  tasks:

     - shell: /usr/bin/foo
       register: foo_result
       ignore_errors: True

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

Registered variables are valid on the host the remainder of the playbook run, which is the same as the lifetime of "facts" in Ansible. Effectively registered variables are just like facts.

Accessing Complex Variable Data

We already talked about facts a little higher up in the documentation.

Some provided facts, like networking information, are made available as nested data structures. To access them a simple {{ foo }} is not sufficient, but it is still easy to do. Here's how we get an IP address:

```
{{ ansible_eth0["ipv4"]["address"] }}
```

OR alternatively:

```
{{ ansible_eth0.ipv4.address }}
```

Similarly, this is how we access the first element of an array:

```
{{ foo[0] }}
```

Magic Variables, and How To Access Information About Other Hosts

Even if you didn't define them yourself, Ansible provides a few variables for you automatically. The most important of these are 'hostvars', 'group_names', and 'groups'. Users should not use these names themselves as they are reserved. 'environment' is also reserved.

Hostvars lets you ask about the variables of another host, including facts that have been gathered about that host. If, at this point, you haven't talked to that host yet in any play in the playbook or set of playbooks, you can get at the variables, but you will not be able to see the facts.

If your database server wants to use the value of a 'fact' from another node, or

an inventory variable assigned to another node, it's easy to do so within a template or even an action line:

```
{{ hostvars['test.example.com']['ansible_distribution'] }}
```

Additionally, *group_names* is a list (array) of all the groups the current host is in. This can be used in templates using Jinja2 syntax to make template source files that vary based on the group membership (or role) of the host:

```
{% if 'webserver' in group_names %}
   # some part of a configuration file that only applies to webservers
{% endif %}
```

groups is a list of all the groups (and hosts) in the inventory. This can be used to enumerate all hosts within a group. For example:

```
{% for host in groups['app_servers'] %}
   # something that applies to all app servers.
{% endfor %}
```

A frequently used idiom is walking a group to find all IP addresses in that group:

```
{% for host in groups['app_servers'] %}
   {{ hostvars[host]['ansible_eth0']['ipv4']['address'] }}
{% endfor %}
```

An example of this could include pointing a frontend proxy server to all of the app servers, setting up the correct firewall rules between servers, etc. You need to make sure that the facts of those hosts have been populated before though, for

example by running a play against them if the facts have not been cached recently (fact caching was added in Ansible 1.8).

Additionally, *inventory_hostname* is the name of the hostname as configured in Ansible's inventory host file. This can be useful for when you don't want to rely on the discovered hostname *ansible_hostname* or for other mysterious reasons. If you have a long FQDN, *inventory_hostname_short* also contains the part up to the first period, without the rest of the domain.

play_hosts is available as a list of hostnames that are in scope for the current play. This may be useful for filling out templates with multiple hostnames or for injecting the list into the rules for a load balancer.

delegate_to is the inventory hostname of the host that the current task has been delegated to using 'delegate_to'.

Don't worry about any of this unless you think you need it. You'll know when you do.

Also available, *inventory_dir* is the pathname of the directory holding Ansible's inventory host file, *inventory_file* is the pathname and the filename pointing to the Ansible's inventory host file.

Variable File Separation

It's a great idea to keep your playbooks under source control, but you may wish to make the playbook source public while keeping certain important variables private. Similarly, sometimes you may just want to keep certain information in different files, away from the main playbook.

You can do this by using an external variables file, or files, just like this:

```
---

- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /vars/external_vars.yml

  tasks:

  - name: this is just a placeholder
    command: /bin/echo foo
```

This removes the risk of sharing sensitive data with others when sharing your playbook source with them.

The contents of each variables file is a simple YAML dictionary, like this:

```
---
# in the above example, this would be vars/external_vars.yml
somevar: somevalue
password: magic
```

Note

It's also possible to keep per-host and per-group variables in very similar files, this is covered in Splitting Out Host and Group Specific Data.

Passing Variables On The Command Line

In addition to *vars_prompt* and *vars_files*, it is possible to send variables over the Ansible command line. This is particularly useful when writing a generic release playbook where you may want to pass in the version of the application to deploy:

```
ansible-playbook release.yml --extra-vars "version=1.23.45 other_variable=foo"
```

This is useful, for, among other things, setting the hosts group or the user for the playbook.

Example:

```
---

- hosts: '{{ hosts }}'
  remote_user: '{{ user }}'

  tasks:
     - ...

ansible-playbook release.yml --extra-vars "hosts=vipers user=starbuck"
```

As of Ansible 1.2, you can also pass in extra vars as quoted JSON, like so:

```
--extra-vars '{"pacman":"mrs","ghosts":["inky","pinky","clyde","sue"]}'
```

The key=value form is obviously simpler, but it's there if you need it!

As of Ansible 1.3, extra vars can be loaded from a JSON file with the "@" syntax:

```
--extra-vars "@some_file.json"
```

Also as of Ansible 1.3, extra vars can be formatted as YAML, either on the command line or in a file as above.

Variable Precedence: Where Should I Put A Variable?

A lot of folks may ask about how variables override another. Ultimately it's Ansible's philosophy that it's better you know where to put a variable, and then you have to think about it a lot less.

Avoid defining the variable "x" in 47 places and then ask the question "which x gets used". Why? Because that's not Ansible's Zen philosophy of doing things.

There is only one Empire State Building. One Mona Lisa, etc. Figure out where to define a variable, and don't make it complicated.

However, let's go ahead and get precedence out of the way! It exists. It's a real thing, and you might have a use for it.

If multiple variables of the same name are defined in different places, they win in a certain order, which is:

```
* extra vars (-e in the command line) always win
* then comes connection variables defined in inventory (ansible_ssh_user, etc)
* then comes "most everything else" (command line switches, vars in play, included vars, role vars, etc)
* then comes the rest of the variables defined in inventory
* then comes facts discovered about a system
* then "role defaults", which are the most "defaulty" and lose in priority to everything.
```

> **Note**
> In versions prior to 1.5.4, facts discovered about a system were in the "most everything else" category above.

That seems a little theoretical. Let's show some examples and where you would choose to put what based on the kind of control you might want over values.

First off, group variables are super powerful.

Site wide defaults should be defined as a 'group_vars/all' setting. Group variables are generally placed alongside your inventory file. They can also be returned by a dynamic inventory script (see *Dynamic Inventory*) or defined in things like *Ansible Tower* from the UI or API:

```
---
# file: /etc/ansible/group_vars/all
# this is the site wide default
ntp_server: default-time.example.com
```

Regional information might be defined in a 'group_vars/region' variable. If this group is a child of the 'all' group (which it is, because all groups are), it will override the group that is higher up and more general:

```
---
# file: /etc/ansible/group_vars/boston
ntp_server: boston-time.example.com
```

If for some crazy reason we wanted to tell just a specific host to use a specific NTP server, it would then override the group variable!:

```
---
# file: /etc/ansible/host_vars/xyz.boston.example.com
ntp_server: override.example.com
```

So that covers inventory and what you would normally set there. It's a great place for things that deal with geography or behavior. Since groups are frequently the entity that maps roles onto hosts, it is sometimes a shortcut to set variables on the group instead of defining them on a role. You could go either way.

Remember: Child groups override parent groups, and hosts always override their groups.

Next up: learning about role variable precedence.

We'll pretty much assume you are using roles at this point. You should be using roles for sure. Roles are great. You are using roles aren't you? Hint hint.

Ok, so if you are writing a redistributable role with reasonable defaults, put those in the 'roles/x/defaults/main.yml' file. This means the role will bring along a default value but ANYTHING in Ansible will override it. It's just a default. That's why it says "defaults" :) See *Playbook Roles and Include Statements* for more info about this:

```
---
# file: roles/x/defaults/main.yml
# if not overridden in inventory or as a parameter, this is the value that will be used
http_port: 80
```

if you are writing a role and want to ensure the value in the role is absolutely used in that role, and is not going to be overridden by inventory, you should put it in roles/x/vars/main.yml like so, and inventory values cannot override it. -e however, still will:

```
---
# file: roles/x/vars/main.yml
# this will absolutely be used in this role
http_port: 80
```

So the above is a great way to plug in constants about the role that are always true. If you are not sharing your role with others, app specific behaviors like ports is fine to put in here. But if you are sharing roles with others, putting variables in here might be bad. Nobody will be able to override them with inventory, but they still can by passing a parameter to the role.

Parameterized roles are useful.

If you are using a role and want to override a default, pass it as a parameter to the role like so:

```
roles:
   - { name: apache, http_port: 8080 }
```

This makes it clear to the playbook reader that you've made a conscious choice to override some default in the role, or pass in some configuration that the role can't assume by itself. It also allows you to pass something site-specific that isn't really part of the role you are sharing with others.

This can often be used for things that might apply to some hosts multiple times, like so:

```
roles:
   - { role: app_user, name: Ian    }
   - { role: app_user, name: Terry  }
   - { role: app_user, name: Graham }
   - { role: app_user, name: John   }
```

That's a bit arbitrary, but you can see how the same role was invoked multiple Times. In that example it's quite likely there was no default for 'name' supplied at all. Ansible can yell at you when variables aren't defined – it's the default behavior in fact.

So that's a bit about roles.

There are a few bonus things that go on with roles.

Generally speaking, variables set in one role are available to others. This means if you have a "roles/common/vars/main.yml" you can set variables in there and make use of them in other roles and elsewhere in your playbook:

```
roles:
   - { role: common_settings }
   - { role: something, foo: 12 }
   - { role: something_else }
```

Note

There are some protections in place to avoid the need to namespace variables. In the above, variables defined in common_settings are most

definitely available to ‘something’ and ‘something_else’ tasks, but if “something’s” guaranteed to have foo set at 12, even if somewhere deep in common settings it set foo to 20.

So, that’s precedence, explained in a more direct way. Don’t worry about precedence, just think about if your role is defining a variable that is a default, or a “live” variable you definitely want to use. Inventory lies in precedence right in the middle, and if you want to forcibly override something, use -e.

If you found that a little hard to understand, take a look at the ansible-examples repo on our github for a bit more about how all of these things can work together.

See also

Playbooks
An introduction to playbooks

Conditionals
Conditional statements in playbooks

Jinja2 filters
Jinja2 filters and their uses

Loops
Looping in playbooks

Playbook Roles and Include Statements
Playbook organization by roles

Best Practices
Best practices in playbooks

User Mailing List
Have a question? Stop by the google group!

irc.freenode.net

#ansible IRC chat channel

Previous Next