Playbook Roles and Include Statements

Topics

Introduction

While it is possible to write a playbook in one very large file (and you might start out learning playbooks this way), eventually you'll want to reuse files and start to organize things.

At a basic level, including task files allows you to break up bits of configuration policy into smaller files. Task includes pull in tasks from other files. Since handlers are tasks too, you can also include handler files from the 'handlers:' section.

See *Playbooks* if you need a review of these concepts.

Playbooks can also include plays from other playbook files. When that is done, the plays will be inserted into the playbook to form a longer list of plays.

When you start to think about it – tasks, handlers, variables, and so on – begin to form larger concepts. You start to think about modeling what something is, rather than how to make something look like something. It's no longer "apply this handful of THINGS to these hosts", you say "these hosts are dbservers" or "these hosts are webservers". In programming, we might call that "encapsulating" how things work. For instance, you can drive a car without knowing how the engine works.

Roles in Ansible build on the idea of include files and combine them to form clean, reusable abstractions – they allow you to focus more on the big picture and only dive down into the details when needed.

We'll start with understanding includes so roles make more sense, but our ultimate goal should be understanding roles – roles are great and you should use them every time you write playbooks.

See the ansible-examples repository on GitHub for lots of examples of all of this

put together. You may wish to have this open in a separate tab as you dive in.

Task Include Files And Encouraging Reuse

Suppose you want to reuse lists of tasks between plays or playbooks. You can use include files to do this. Use of included task lists is a great way to define a role that system is going to fulfill. Remember, the goal of a play in a playbook is to map a group of systems into multiple roles. Let’s see what this looks like...

A task include file simply contains a flat list of tasks, like so:

```
---
# possibly saved as tasks/foo.yml

- name: placeholder foo
  command: /bin/foo

- name: placeholder bar
  command: /bin/bar
```

Include directives look like this, and can be mixed in with regular tasks in a playbook:

```
tasks:

  - include: tasks/foo.yml
```

You can also pass variables into includes. We call this a ‘parameterized include’.

For instance, if deploying multiple wordpress instances, I could contain all of my wordpress tasks in a single wordpress.yml file, and use it like so:

```
tasks:
  - include: wordpress.yml wp_user=timmy
  - include: wordpress.yml wp_user=alice
  - include: wordpress.yml wp_user=bob
```

If you are running Ansible 1.4 and later, include syntax is streamlined to match roles, and also allows passing list and dictionary parameters:

```
tasks:
 - { include: wordpress.yml, wp_user: timmy, ssh_keys: [ 'keys/one.txt', 'keys/two.txt' ] }
```

Using either syntax, variables passed in can then be used in the included files. We'll cover them in *Variables*. You can reference them like this:

```
{{ wp_user }}
```

(In addition to the explicitly passed-in parameters, all variables from the vars section are also available for use here as well.)

Starting in 1.0, variables can also be passed to include files using an alternative syntax, which also supports structured variables:

```
tasks:

  - include: wordpress.yml
    vars:
        wp_user: timmy
        some_list_variable:
          - alpha
          - beta
```

```
    - gamma
```

Playbooks can include other playbooks too, but that's mentioned in a later section.

> **Note**
> As of 1.0, task include statements can be used at arbitrary depth. They were previously limited to a single level, so task includes could not include other files containing task includes.

Includes 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:

```
---
# this might be in a file like handlers/handlers.yml
- name: restart apache
  service: name=apache state=restarted
```

And in your main playbook file, just include it like so, at the bottom of a play:

```
handlers:
  - include: handlers/handlers.yml
```

You can mix in includes along with your regular non-included tasks and handlers.

Includes can also be used to import one playbook file into another. This allows you to define a top-level playbook that is composed of other playbooks.

For example:

```
- name: this is a play at the top level of a file
  hosts: all
  remote_user: root

  tasks:

  - name: say hi
    tags: foo
    shell: echo "hi..."

- include: load_balancers.yml
- include: webservers.yml
- include: dbservers.yml
```

Note that you cannot do variable substitution when including one playbook inside another.

> **Note**
> You can not conditionally path the location to an include file, like you can with ‘vars_files’. If you find yourself needing to do this, consider how you can restructure your playbook to be more class/role oriented. This is to say you cannot use a ‘fact’ to decide what include file to use. All hosts contained within the play are going to get the same tasks. (‘*when*‘ provides some ability for hosts to conditionally skip tasks).

Roles

New in version 1.2.

Now that you have learned about tasks and handlers, what is the best way to organize your playbooks? The short answer is to use roles! Roles are ways of automatically loading certain vars_files, tasks, and handlers based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

Roles are just automation around ‘include’ directives as described above, and really don’t contain much additional magic beyond some improvements to search path handling for referenced files. However, that can be a big thing!

Example project structure:

```
site.yml
webservers.yml
fooservers.yml
roles/
   common/
     files/
     templates/
     tasks/
     handlers/
     vars/
     defaults/
     meta/
   webservers/
     files/
     templates/
     tasks/
     handlers/
     vars/
     defaults/
     meta/
```

In a playbook, it would look like this:

```
---
- hosts: webservers
  roles:
     - common
     - webservers
```

This designates the following behaviors, for each role ‘x’:

- If roles/x/tasks/main.yml exists, tasks listed therein will be added to the play
- If roles/x/handlers/main.yml exists, handlers listed therein will be added to the play
- If roles/x/vars/main.yml exists, variables listed therein will be added to the play
- If roles/x/meta/main.yml exists, any role dependencies listed therein will be added to the list of roles (1.3 and later)
- Any copy tasks can reference files in roles/x/files/ without having to path them relatively or absolutely
- Any script tasks can reference scripts in roles/x/files/ without having to path them relatively or absolutely
- Any template tasks can reference files in roles/x/templates/ without having to path them relatively or absolutely
- Any include tasks can reference files in roles/x/tasks/ without having to path them relatively or absolutely

In Ansible 1.4 and later you can configure a roles_path to search for roles. Use this to check all of your common roles out to one location, and share them easily between multiple playbook projects. See *The Ansible Configuration File* for details about how to set this up in ansible.cfg.

> **Note**
> Role dependencies are discussed below.

If any files are not present, they are just ignored. So it’s ok to not have a ‘vars/’ subdirectory for the role, for instance.

Note, you are still allowed to list tasks, vars_files, and handlers “loose” in playbooks without using roles, but roles are a good organizational feature and are highly recommended. If there are loose things in the playbook, the roles are evaluated first.

Also, should you wish to parameterize roles, by adding variables, you can do so, like this:

```
---

- hosts: webservers
  roles:
    - common
    - { role: foo_app_instance, dir: '/opt/a',  port: 5000 }
    - { role: foo_app_instance, dir: '/opt/b',  port: 5001 }
```

While it’s probably not something you should do often, you can also conditionally apply roles like so:

```
---

- hosts: webservers
  roles:
    - { role: some_role, when: "ansible_os_family == 'RedHat'" }
```

This works by applying the conditional to every task in the role. Conditionals are covered later on in the documentation.

Finally, you may wish to assign tags to the roles you specify. You can do so inline::

```
---

- hosts: webservers
  roles:
    - { role: foo, tags: ["bar", "baz"] }
```

If the play still has a ‘tasks’ section, those tasks are executed after roles are applied.

If you want to define certain tasks to happen before AND after roles are applied, you can do this:

```
---

- hosts: webservers

  pre_tasks:
    - shell: echo 'hello'

  roles:
    - { role: some_role }

  tasks:
    - shell: echo 'still busy'

  post_tasks:
    - shell: echo 'goodbye'
```

Note
If using tags with tasks (described later as a means of only running part of a playbook), be sure to also tag your pre_tasks and post_tasks and pass those along as well, especially if the pre and post tasks are used for monitoring outage window control or load balancing.

Role Default Variables

New in version 1.3.

Role default variables allow you to set default variables for included or dependent roles (see below). To create defaults, simply add a *defaults/main.yml* file in your role directory. These variables will have the lowest priority of any variables available, and can be easily overridden by any other variable, including inventory variables.

Role Dependencies

New in version 1.3.

Role dependencies allow you to automatically pull in other roles when using a role. Role dependencies are stored in the *meta/main.yml* file contained within the role directory. This file should contain a list of roles and parameters to insert before the specified role, such as the following in an example *roles/myapp/meta/main.yml*:

```
---
dependencies:
```

```
  - { role: common, some_parameter: 3 }
  - { role: apache, port: 80 }
  - { role: postgres, dbname: blarg, other_parameter: 12 }
```

Role dependencies can also be specified as a full path, just like top level roles:

```
---
dependencies:
   - { role: '/path/to/common/roles/foo', x: 1 }
```

Role dependencies can also be installed from source control repos or tar files, using a comma separated format of path, an optional version (tag, commit, branch etc) and optional friendly role name (an attempt is made to derive a role name from the repo name or archive filename):

```
---
dependencies:
  - { role: 'git+http://git.example.com/repos/role-foo,v1.1,foo' }
  - { role: '/path/to/tar/file.tgz,,friendly-name' }
```

Roles dependencies are always executed before the role that includes them, and are recursive. By default, roles can also only be added as a dependency once - if another role also lists it as a dependency it will not be run again. This behavior can be overridden by adding *allow_duplicates: yes* to the *meta/main.yml* file. For example, a role named ‘car’ could add a role named ‘wheel’ to its dependencies as follows:

```
---
dependencies:
- { role: wheel, n: 1 }
- { role: wheel, n: 2 }
```

```
- { role: wheel, n: 3 }
- { role: wheel, n: 4 }
```

And the *meta/main.yml* for wheel contained the following:

```
---
allow_duplicates: yes
dependencies:
- { role: tire }
- { role: brake }
```

The resulting order of execution would be as follows:

```
tire(n=1)
brake(n=1)
wheel(n=1)
tire(n=2)
brake(n=2)
wheel(n=2)
...
car
```

> **Note**
> Variable inheritance and scope are detailed in the *Variables*.

Embedding Modules In Roles

This is an advanced topic that should not be relevant for most users.

If you write a custom module (see *Developing Modules*) you may wish to distribute it as part of a role. Generally speaking, Ansible as a project is very interested in taking high-quality modules into ansible core for inclusion, so this shouldn't be the norm, but it's quite easy to do.

A good example for this is if you worked at a company called AcmeWidgets, and wrote an internal module that helped configure your internal software, and you wanted other people in your organization to easily use this module – but you didn't want to tell everyone how to configure their Ansible library path.

Alongside the 'tasks' and 'handlers' structure of a role, add a directory named 'library'. In this 'library' directory, then include the module directly inside of it.

Assuming you had this:

```
roles/
   my_custom_modules/
       library/
          module1
          module2
```

The module will be usable in the role itself, as well as any roles that are called *after* this role, as follows:

```
- hosts: webservers
  roles:
    - my_custom_modules
    - some_other_role_using_my_custom_modules
    - yet_another_role_using_my_custom_modules
```

This can also be used, with some limitations, to modify modules in Ansible's core

distribution, such as to use development versions of modules before they are released in production releases. This is not always advisable as API signatures may change in core components, however, and is not always guaranteed to work. It can be a handy way of carrying a patch against a core module, however, should you have good reason for this. Naturally the project prefers that contributions be directed back to github whenever possible via a pull request.

Ansible Galaxy

Ansible Galaxy is a free site for finding, downloading, rating, and reviewing all kinds of community developed Ansible roles and can be a great way to get a jumpstart on your automation projects.

You can sign up with social auth, and the download client 'ansible-galaxy' is included in Ansible 1.4.2 and later.

Read the "About" page on the Galaxy site for more information.

See also

Ansible Galaxy
How to share roles on galaxy, role management

YAML Syntax
Learn about YAML syntax

Playbooks
Review the basic Playbook language features

Best Practices
Various tips about managing playbooks in the real world

Variables
All about variables in playbooks

Conditionals
Conditionals in playbooks

Loops
Loops in playbooks

About Modules
Learn about available modules

Developing Modules
Learn how to extend Ansible by writing your own modules

GitHub Ansible examples
Complete playbook files from the GitHub project source

Mailing List
Questions? Help? Ideas? Stop by the list on Google Groups

Previous Next