Red Hat Enterprise Linux Automation with Ansible

FEEDBACK TRANSLATIONS CERTIFICATE OF ATTENDANCE

VIDEO CLASSROOM

Chapter 7. Simplifying Playbooks with Roles ☆

Abstract

Goal	Use Ansible roles to develop playbooks more quickly and to reuse Ansible code.

Objectives	• Describe what a role is, how it is structured, and how you can use it in a playbook. • Write playbooks that take advantage of Red Hat Enterprise Linux System Roles to perform standard operations. • Create a role in a playbook's project directory and run it as part of one of the plays in the playbook. • Select and retrieve roles from Ansible Galaxy or other sources such as a Git repository, and use them in your playbooks. • Obtain a set of related roles, supplementary modules, and other content from content collections, and use them in a playbook.
Sections	• Describing Role Structure (and Quiz) • Reusing Content with System Roles (and Guided Exercise) • Creating Roles (and Guided Exercise) • Deploying Roles with Ansible Galaxy (and Guided Exercise) • Getting Roles and Modules from Content Collections (and Guided Exercise)
Lab	• Simplifying Playbooks with Roles

Describing Role Structure

Objectives

After completing this section, you should be able to describe what a role is, how it is structured, and how you can use it in a playbook.

Structuring Ansible Playbooks with Roles

As you develop more playbooks, you will probably discover that you have many opportunities to reuse code from playbooks that you have already written. Perhaps a play to configure a MySQL database for one application could be re-purposed, with different hostnames, passwords, and users, to configure a MySQL database for another application.

But in the real world, that play might be long and complex, with many included or imported files, and with tasks and handlers to manage various situations. Copying all that code into another playbook might be nontrivial work.

Ansible *roles* provide a way for you to make it easier to reuse Ansible code generically. You can package, in a standardized directory structure, all the tasks, variables, files, templates, and other resources needed to provision infrastructure or deploy applications. Copy that role from project to project simply by copying the directory. You can then simply call that role from a play to execute it.

A well-written role will allow you to pass variables to the role from the playbook that adjust its behavior, setting all the site-specific hostnames, IP addresses, user names, secrets, or other locally-specific details you need. For example, a role to deploy a database server might have been written to support variables which set the hostname, database admin user and password, and other parameters that need customization for your installation. The author of the role can also ensure that reasonable default values are set for those variables if you choose not to set them in the play.

Ansible roles have the following benefits:

- Roles group content, allowing easy sharing of code with others
- Roles can be written that define the essential elements of a system type: web server, database server, Git repository, or other purpose

- Roles make larger projects more manageable
- Roles can be developed in parallel by different administrators
  In addition to writing, using, reusing, and sharing your own roles, you can get roles from other sources. Some roles are included as part of Red Hat Enterprise Linux, in the rhel-system-roles package. You can also get numerous community-supported roles from the Ansible Galaxy website. Later in this chapter, you will learn more about these roles.

Examining the Ansible Role Structure

An Ansible role is defined by a standardized structure of subdirectories and files. The top-level directory defines the name of the role itself. Files are organized into subdirectories that are named according to each file's purpose in the role, such as `tasks` and `handlers`. The `files` and `templates` subdirectories contain files referenced by tasks in other YAML files.

The following `tree` command displays the directory structure of the `user.example` role.

```
[user@host roles]$ tree user.example
user.example/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Table 7.1. Ansible role subdirectories

Subdirectory	Function
`defaults`	The main.yml file in this directory contains the default values of role variables that can be overwritten when the role is used. These variables have low precedence and are intended to be changed and customized in plays.
`files`	This directory contains static files that are referenced by role tasks.
`handlers`	The main.yml file in this directory contains the role's handler definitions.
`meta`	The main.yml file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
`tasks`	The main.yml file in this directory contains the role's task definitions.
`templates`	This directory contains Jinja2 templates that are referenced by role tasks.
`tests`	This directory can contain an inventory and test.yml playbook that can be used to test the role.
`vars`	The main.yml file in this directory defines the role's variable values. Often these variables are used for internal purposes within the role. These variables have high precedence, and are not intended to be changed when used in a playbook.

Not every role will have all of these directories.

Defining Variables and Defaults

Role variables are defined by creating a `vars/main.yml` file with key: value pairs in the role directory hierarchy. They are referenced in the role YAML file like any other variable: `{{ VAR_NAME }}`. These variables have a high precedence and can not be overridden by inventory variables. The intent of these variables is that they are used by the internal functioning of the role.

Default variables allow default values to be set for variables that can be used in a play to configure the role or customize its behavior. They are defined by creating a `defaults/main.yml` file with key: value pairs in the role directory hierarchy. Default variables have the lowest precedence of any variables available. They can be easily overridden by any other variable, including inventory variables. These variables are intended to provide the person writing a play that uses the role with a way to customize or control exactly what it is going to do. They can be used to provide information to the role that it needs to configure or deploy something properly.

Define a specific variable in either `vars/main.yml` or `defaults/main.yml`, but not in both places. Default variables should be used when it is intended that their values will be overridden.

> **IMPORTANT**
>
> Roles should not have site-specific data in them. They definitely should not contain any secrets like passwords or private keys.
>
> This is because roles are supposed to be generic, reusable, and freely shareable. Site-specific details should not be hard coded into them.
>
> Secrets should be provided to the role through other means. This is one reason you might want to set role variables when calling a role. Role variables set in the play could provide the secret, or point to an Ansible Vault-encrypted file containing the secret.

Using Ansible Roles in a Playbook

Using roles in a playbook is straightforward. The following example shows one way to call Ansible roles.

```
---
- hosts: remote.example.com
  roles:
    - role1
    - role2
```

For each role specified, the role tasks, role handlers, role variables, and role dependencies will be imported into the playbook, in that order. Any `copy`, `script`, `template`, or `include_tasks`/`import_tasks` tasks in the role can reference the relevant files, templates, or task files in the role without absolute or relative path names. Ansible looks for them in the role's `files`, `templates`, or `tasks` subdirectories respectively.

When you use a `roles` section to import roles into a play, the roles will run first, before any tasks that you define for that play.

The following example sets values for two role variables of `role2`, `var1` and `var2`. Any `defaults` and `vars` variables are overridden when `role2` is used.

```
---
- hosts: remote.example.com
  roles:
    - role: role1
    - role: role2
      var1: val1
      var2: val2
```

Another equivalent YAML syntax which you might see in this case is:

```
---
- hosts: remote.example.com
  roles:
    - role: role1
    - { role: role2, var1: val1, var2: val2 }
```

There are situations in which this can be harder to read, even though it is more compact.

> **IMPORTANT**
>
> Role variables set inline (role parameters), as in the preceding examples, have very high precedence. They will override most other variables.
>
> Be very careful not to reuse the names of any role variables that you set inline anywhere else in your play, since the values of the role variables will override inventory variables and any play `vars`.

Controlling Order of Execution

For each play in a playbook, tasks execute as ordered in the tasks list. After all tasks execute, any notified handlers are executed.

When a role is added to a play, role tasks are added to the beginning of the tasks list. If a second role is included in a play, its tasks list is added after the first role.

Role handlers are added to plays in the same manner that role tasks are added to plays. Each play defines a handlers list. Role handlers are added to the handlers list first, followed by any handlers defined in the `handlers` section of the play.

In certain scenarios, it may be necessary to execute some play tasks before the roles. To support such scenarios, plays can be configured with a `pre_tasks` section. Any task listed in this section executes before any roles are executed. If any of these tasks notify a handler, those handler tasks execute before the roles or normal tasks.

Plays also support a `post_tasks` keyword. These tasks execute after the play's normal tasks, and any handlers they notify, are run.

The following play shows an example with `pre_tasks`, `roles`, `tasks`, `post_tasks` and `handlers`. It is unusual that a play would contain all of these sections.

```
- name: Play to illustrate order of execution
  hosts: remote.example.com
  pre_tasks:
    - debug:
        msg: 'pre-task'
      notify: my handler
  roles:
    - role1
  tasks:
    - debug:
        msg: 'first task'
      notify: my handler
  post_tasks:
    - debug:
        msg: 'post-task'
      notify: my handler
  handlers:
    - name: my handler
      debug:
        msg: Running my handler
```

In the above example, a `debug` task executes in each section to notify the `my handler` handler. The `my handler` task is executed three times:

- after all the `pre_tasks` tasks execute
- after all role tasks and tasks from the `tasks` section execute
- after all the `post_tasks` execute

Roles can be added to a play using an ordinary task, not just by including them in the `roles` section of a play. Use the `include_role` module to dynamically include a role, and use the `import_role` module to statically import a role.

The following playbook demonstrates how a role can be included using a task with the `include_role` module.

```
- name: Execute a role as a task
  hosts: remote.example.com
  tasks:
    - name: A normal task
      debug:
        msg: 'first task'
    - name: A task to include role2 here
      include_role: role2
```

> **NOTE**
>
> The `include_role` module was added in Ansible 2.3, and the `import_role` module in Ansible 2.4.

> **REFERENCES**
>
> Roles — Ansible Documentation
> (https://docs.ansible.com/ansible/2.9/user_guide/playbooks_reuse_roles.html)

➜ NEXT (/ROL/APP/COURSES/RH294-8.4/PAGES/CH07S02)