Ansible on Windows Fundamentals

Reusing Automation Code

Creating Custom Roles

Red Hat

Objective

- Create a custom Ansible role for Windows-based managed hosts

Creating Roles

- Ansible **roles** allow you to make automation code more reusable.
- Provides packaged tasks that can be configured through variables.
- The playbook just calls the role and passes it the right values through its variables.
- Allows you to create generic code for one project and reuse it on other projects.

Benefits of Ansible Roles

- Roles group content, allowing easy sharing of code with others.
- Roles can be written in a way that define the essential elements of a system type: web server, database server, Git repository, and more.
- Roles make larger projects more manageable.
- Different administrators can develop roles in parallel.

Creating Ansible Roles

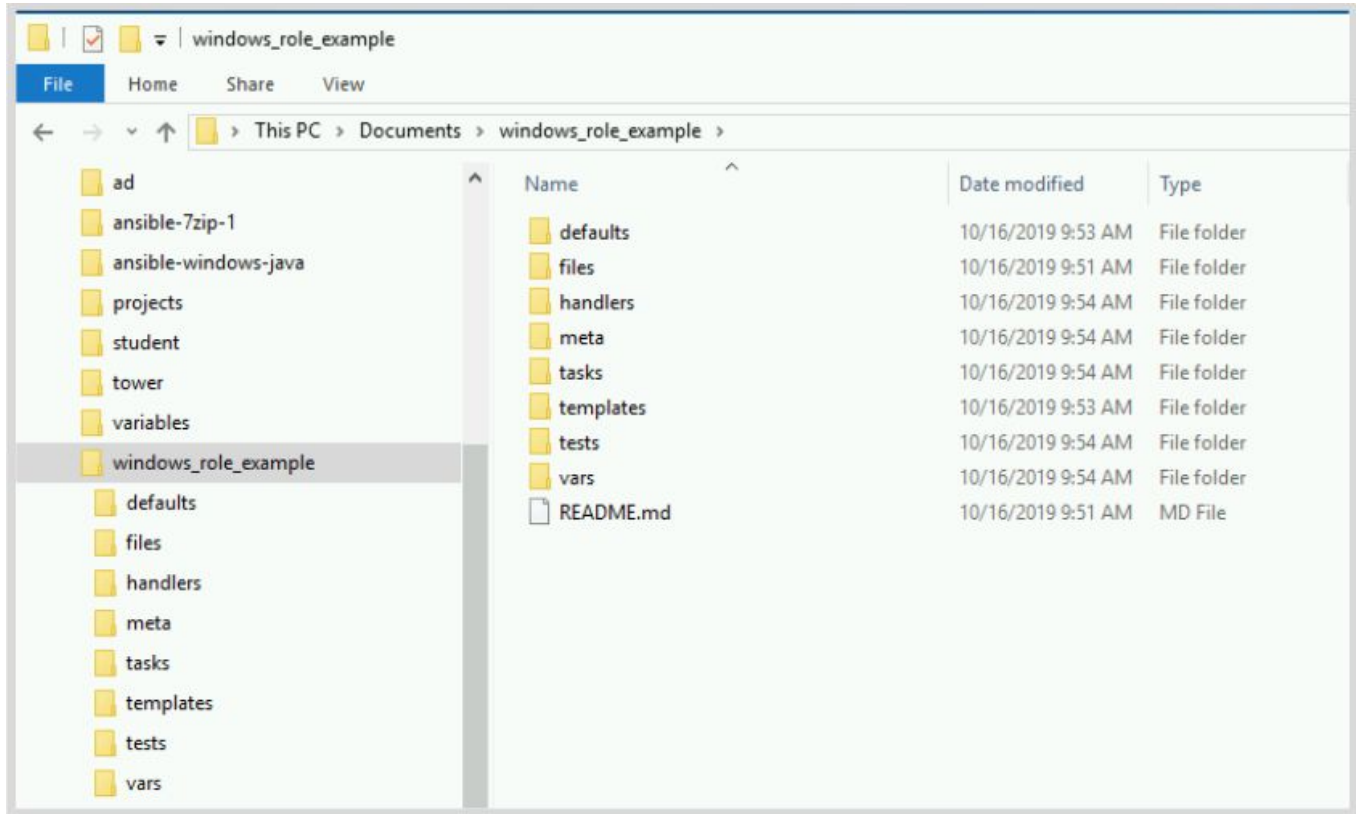
- You can write a role using the same tools you use to write playbooks
- Creating and using a role is a three step process:
  1. Create the role directory structure.
  2. Define the role content.
  3. Use the role in a play.
- One way to create a role is to start by writing a play and then refactoring it into a generic role.

Creating the Role's Directory Structure

- Each role has its own directory with a standardized folder structure.
- The top-level directory defines the name of the role itself.
- Files are organized into subdirectories named according to the purpose of each file in the role, such as **tasks** and **handlers**.
- For Windows-based users, it can be simplest to manually create the directory structure.
- On Linux, the **ansible-galaxy init *rolename*** command can create the "skeleton" directory for you.

Creating the Role Skeleton

- Two different views of the structure of an example role (**windows_role_example**):




```
windows_role_example
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```


Red Hat

Directory Structure

Directory	Function
defaults	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 by plays.
files	Static files that are referenced by role tasks.
handlers	The handler definitions used by the role.
meta	The **main.yml** file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
tasks	Tasks performed by the role, similar to a play's tasks section.
templates	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	Defines values of variables used internally by the role. These variables have high precedence (therefore difficult to override), and are not intended to be changed by the play.

Starting From a Playbook

- At right is an example of a playbook to create a shared folder for a local group on all systems in the inventory group **windows_group**
- The name of the group and its shared folder are hard coded into the play
- We will convert this into a role that can create any shared folder and any local group name

```
- name: Play to create shared folder
  hosts: windows_group
  tasks:
    - name: Create local group
      win_group: LocalUsers
        description: Access to C:SharedFolder

    - name: Shared folder exists
      win_file:
        path: C:\SharedFolder
        state: directory

    - name: Set ACL of shared folder
      win_acl:
        path: C:\SharedFolder
        rights: FullControl
        state: present
        type: allow
        user: LocalUsers

    - name: Remove parent inheritance on folder
      win_acl_inheritance:
        path: C:\SharedFolder
        reorganize: yes
        state: absent
```


Red Hat

Use Variables as Parameters

- The play has now been rewritten so that variables control the name of the shared group and shared directory to create

```
- name: Play to create shared folder
  hosts: windows_group
  vars:
    sharedgroup: LocalUsers
    shareddir: C:\SharedFolder
  tasks:
    - name: Create local group
      win_group: "{{ sharedgroup }}"
        description: Access to {{ shareddir }}

    - name: Shared folder exists
      win_file:
        path: "{{ shareddir }}"
        state: directory

    - name: Set ACL of shared folder
      win_acl:
        path: "{{ shareddir }}"
        rights: FullControl
        state: present
        type: allow
        user: "{{ sharedgroup }}"

    - name: Remove parent inheritance on folder
      win_acl_inheritance:
        path: "{{ shareddir }}"
        reorganize: yes
        state: absent
```


Red Hat

Defining the Role Content

- Create a new directory for your role with the directories you need for this role.
- We will only need **meta**, **tasks**, and **defaults** directories, and a **README.md** file, in the role's directory for this example.
- We will put this in a **roles** directory in the same place as the existing playbook for now so that we can test it later.

```
project/
├── playbook.yml
└── roles
    └── shared_directory
        ├── defaults
        │   └── main.yml
        ├── meta
        │   └── main.yml
        ├── README.md
        └── tasks
            └── main.yml
```

Defining the Role's Tasks

- Copy the tasks from your playbook into the **tasks\main.yml** file.
- Lines that start with **#** are comments.
- Indentation just needs to be consistent.

```
# tasks file for shared_directory role

    - name: Create local group
      win_group: "{{ sharedgroup }}"
        description: Access to {{ shareddir }}

    - name: Shared folder exists
      win_file:
        path: "{{ shareddir }}"
        state: directory

    - name: Set ACL of shared folder
      win_acl:
        path: "{{ shareddir }}"
        rights: FullControl
        state: present
        type: allow
        user: "{{ sharedgroup }}"

    - name: Remove parent inheritance on folder
      win_acl_inheritance:
        path: "{{ shareddir }}"
        reorganize: yes
        state: absent
```

Defining the Role's Defaults

- Copy the variables from your playbook into the **defaults\main.yml** file.
- This will set the default values for the role if no settings are specified.
- These variables can be overridden with different values when you call the role from a play.

```
# defaults file for shared_directory role

sharedgroup: LocalUsers
shareddir: C:\SharedFolder
```


Red Hat

Documenting the Role

- Create a **meta\main.yml** file.
- This will include some basic information about this role. A simple example is at right.
- Look at other roles and the Ansible documentation for more complex examples.
- You may also create a **README.md** file in Markdown format as documentation for your role. See examples from roles at https://galaxy.ansible.com/

```
galaxy_info:
  author: your name
  description: your role description
  company: your company (optional)

  # This is an open source role
  license: MIT

  # You may specify minimum supported version
  # of Ansible that works for this role
  min_ansible_version: 2.8
```

Using a Role in a Playbook

```
- name: Play to create shared folder
  hosts: windows_group

  roles:
    - shared_directory
```

- An easy way to call a role in a play is to list it in a roles section.
- This assumes the role's directory has been copied into the playbook's **roles** directory.
- This play calls the **shared_directory** role from the example in this presentation.
- Because no variables are specified, the role is applied with its default values.
- This combination does exactly what the original playbook did.
- Note that there are no tasks on this play. It can have tasks, but the roles run first.

Using a Role with Custom Parameters

```
- name: Play to create shared folder
  hosts: windows_group

  roles:
    - shared_directory

    - role: shared_directory
      vars:
        sharedgroup: DifferentGroup
        shareddir: C:\TestDirectory
```

- In this example play, the role is called twice.
- The first time it is called with its default options and creates the default directory and group.
- The second time it overrides the role's default variables and creates a different directory and group.

Using a Role in a Playbook

```
- name: Play to create shared folder
  hosts: windows_group

  tasks:
    - name: Execute role
      include_role:
        name: shared_directory

    - name: Role with non-default parameters
      include_role:
        name: shared_directory
      vars:
        sharedgroup: DifferentGroup
        shareddir: C:\TestDirectory
```

- As an alternative, you can call the role as a task at any time by using the **include_role** module.
- This syntax lets you mix roles with normal tasks in the play.

Deploying Roles with Ansible Galaxy

Objectives

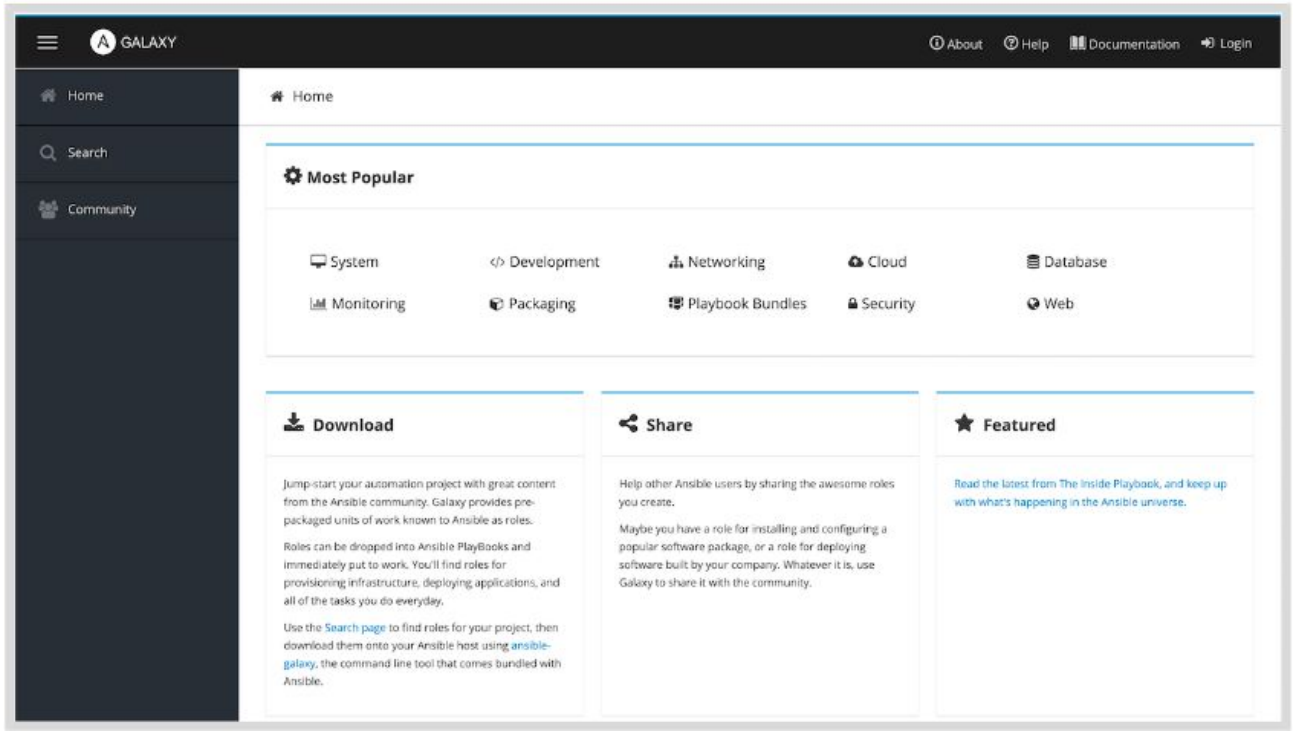
- Explain the key components and functionality of Ansible Galaxy
- Create a playbook that uses a community role from Ansible Galaxy
- Configure **roles\requirements.yml** so Red Hat Ansible Tower automatically downloads roles needed by the playbook from Ansible Galaxy or a Git repository

Obtaining and Using Roles

- Normally, roles are kept in their own Git repository separately from the playbook
- This helps avoid each playbook having a private copy of the role that has local edits
- But the role has to be available to the playbook when the playbook is run
- You might want to use your own roles, or reuse roles written by the open source community

About Ansible Galaxy

- https://galaxy.ansible.com
- Public library of Ansible content
- Written by a community of Ansible administrators and users
- Searchable database
- Links to documentation and videos for new Ansible user and role developers.
- Not officially supported by Red Hat, roles may have varying quality levels.

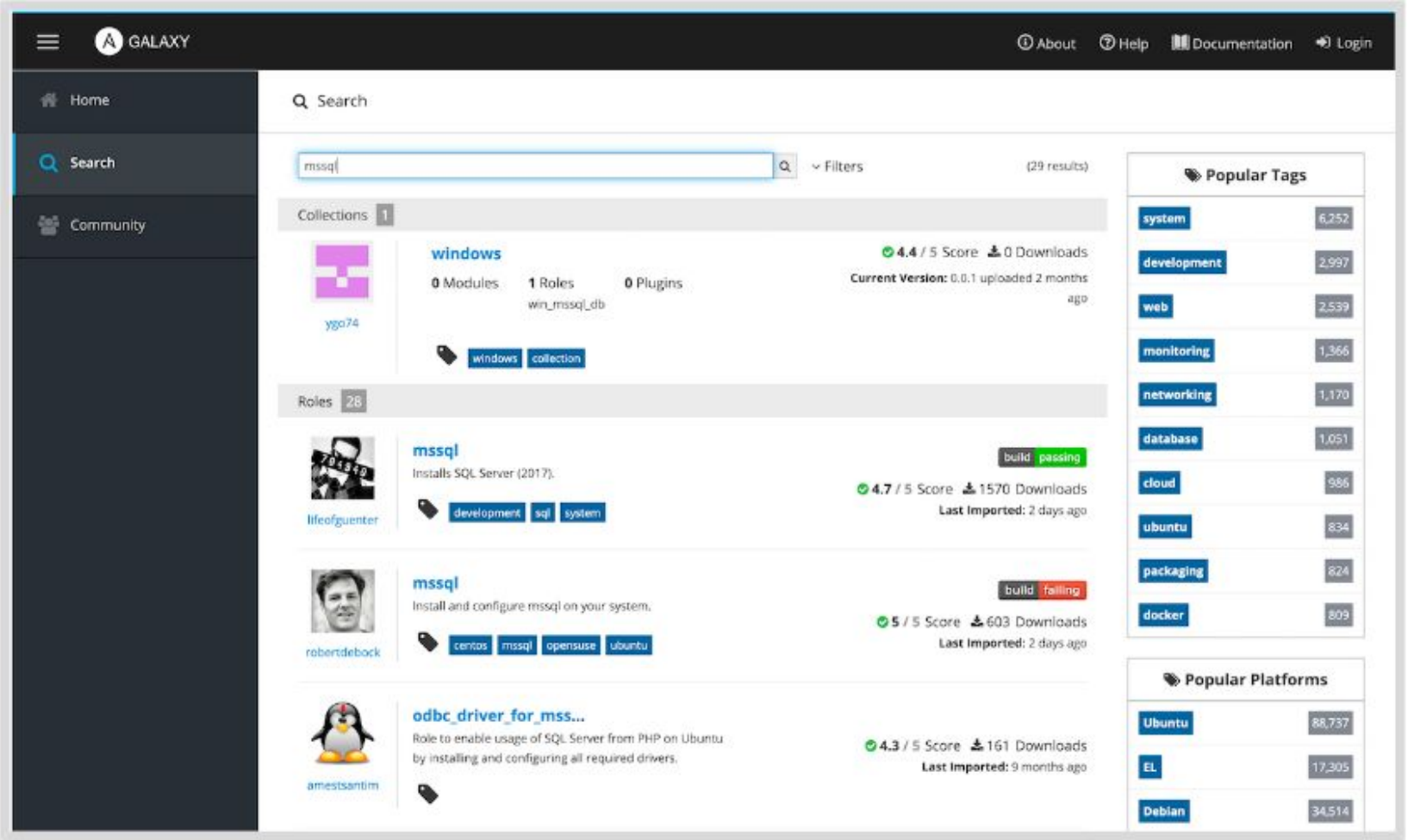


Red Hat

Introducing Ansible Galaxy

- Getting Help with Ansible Galaxy
  - **Documentation** tab on the Ansible Galaxy website home page
  - Provides information about downloading and implementing roles from Ansible Galaxy
  - Instructions on how to develop and upload roles to Ansible Galaxy
- Browsing Ansible Galaxy for Roles
  - The search tab gives users access to information about the roles published on Ansible Galaxy. You can search for an Ansible role by its name, tags, or other role attributes.
  - Many of the roles on Ansible Galaxy are designed for other operating systems or network devices. Use the **Search** tab to find Microsoft Windows-compatible roles.
  - Results are presented in descending order, based on the **Best Match** score.

Introducing Ansible Galaxy



Red Hat

Some Role Security Considerations

- You do not have to use Ansible Galaxy to store your roles
- You might want to keep certain roles private and store them in a private Git repository
- It is important to never put sensitive data like passwords in a role itself
- Sensitive data should be set through variables passed to the role by the play

Installing Roles Using a Requirements File

- Red Hat Ansible Tower can install a list of roles for a project based on definitions in a text file.
- If your playbook requires specific roles, create a **roles\requirements.yml** file in the project directory
- That file is a YAML list of roles to install
- For each role
  - Use the **name** keyword to override the local name of the role.
  - Use the **version** keyword to specify the version of the role.
  - The **src** attribute specifies the source of the role.
- The **requirements.yml** entry at right downloads and installs version 1.3.2 of the **arillso.chocolatey** role from Ansible Galaxy, but renames it **test.chocolatey** locally (the name you play must use)

```
- src: arillso.chocolatey
  name: test.chocolatey
  version: 1.3.2
```

Installing Roles Using a Requirements File

Here are four examples from a **roles\requirements.yml** file:

1. Grabs the latest version of arillso.ntp from Ansible Galaxy

2. Gets a specific version of arillso.ntp from Ansible Galaxy. This is a better practice to avoid unexpected changes.

3. Gets a role from a Git repository and selects a specific commit. It also renames the role locally.

4. Gets a role from a Git repository using SSH and selects the latest version on a specific branch.

```
# from Ansible Galaxy, using the latest version
- src: arillso.ntp

# from Ansible Galaxy, specific version and override name
- src: arillso.ntp
  version: "1.4.3"

# from a Git repo using HTTPS and selecting a specific commit
- src: https://gitlab.example.com/automation/shared_directory.git
  scm: git
  version: 56e00a54
  name: windows_shared_directory

# from a Git repo using SSH and selecting the master branch
- src: git@gitlab.example.com:automation/shared_directory.git
  scm: git
  version: master
```

Retrieving Roles with a Requirements File

- Ansible Tower will automatically retrieve your roles when you launch the job template for your playbook.
- If you are not using Ansible Tower to run playbooks, but are using the Linux command line tool **ansible-playbook**, run **ansible-galaxy install -r roles/requirements.yml** in the playbook directory to update your roles.
- See https://galaxy.ansible.com/docs/using/installing.html for more examples.

Using Ansible to Run PowerShell Desired State Configuration Resources

Objectives

- Describe the key components of a DSC resource needed to configure and run it using Ansible
- Create and run an Ansible Playbook to obtain resources from PowerShell DSC Gallery
- Create and run an Ansible Playbook that uses resources from PowerShell DSC Gallery
- Explain considerations on when to use Ansible modules and roles and when to use PowerShell DSC resources

Desired State Configuration (DSC)

- Desired State Configuration is a system configuration management platform built into PowerShell that uses a declarative model.
- It uses a push-mode execution to send configurations to the target hosts through code.
- This configuration management platform is executed differently than Ansible, and is specific to the Windows platform.
- DSC uses a Local Configuration Manager that runs on all the remote nodes as the DSC execution engine.
- Microsoft fosters a community effort to build and maintain DSC resources for many technologies.
  - These are published each month to the PowerShell Gallery as the DSC Resource Kit
  - These are available from the GitHub repository at https://github.com/PowerShell/DscResources

Getting DSC Resources from PowerShell Gallery

Use **win_psmodule** to get DSC resources from PowerShell Gallery or other repositories:

- At right is a snippet from an example play
- The first task makes sure the xMySql DSC resource is present
- The second task makes sure that both the SpeculationControl and PendingReboot DSC resources are present and up to date
- **win_psmodule** can also make sure that a module is **absent** or that you have the **latest** version
- This ensures that the DSC resources are available on the managed hosts so that you can call them from Ansible

```
tasks:
  - name: Install DSC resource
    win_psmodule:
      name: xMySql
      state: present

  - name: Install latest version of several DSC resources
    win_psmodule:
      name: "{{ item }}"
      state: latest
    loop:
      - SpeculationControl
      - PendingReboot
```

Desired State Configuration (DSC) example Ansible task

```
- name: Create IIS site and add HTTP binding using DSC resource
  win_dsc:
    resource_name: xWebsite
    Ensure: Present
    Name: Ansible
    State: Started
    PhysicalPath: C:\website\MySite
    BindingInfo:
      - Protocol: http
     Port: 8080
     IPAddress: '*'
```

Desired State Configuration (DSC) components for Ansible usage

- The Ansible module `win_dsc` allows Ansible to use existing DSC resources for Windows hosts.
- The minimum requirement to run this module on hosts is PowerShell 5.0 or later.
- You must be familiar with the catalog and purpose of DSC resources to provide the proper instructions in your playbook.
- DSC task execution runs each resource using the **`SYSTEM`** account on the targeted host.
  - To run DSC tasks as a different user, the `win_dsc` module accepts arguments for
    - `PsDscRunAsCredential_username`
    - `PsDscRunAsCredential_password`

Desired State Configuration (DSC) execution user in Ansible

```
win_dsc:
  resource_name: Registry
  Ensure: Present
  Key: HKEY_CURRENT_USER\ExampleKey
  ValueName: TestValue
  ValueData: TestData
  PsDscRunAsCredential_username: '{{ansible_user}}'
  PsDscRunAsCredential_password: '{{ansible_password}}'
  no_log: true
```

Determining usage of DSC resource or an existing Ansible module

- A large overlap exists between DSC resources and the equivalent Ansible modules.
- The table below provides criteria for determining when to utilize each of these implementations.

When to use an Ansible module	When to use a DSC resource
The host does not support PowerShell v5.0	You are familiar and comfortable with the DSC resource
The Ansible module has a feature the DSC resource does not	Reusing authored code that uses a DSC resource
The Ansible module checks for idempotency are better suited	The Ansible module does not support a feature
The Ansible module in question supports diff mode	There is no Ansible module available
There are bugs in a DSC resource	There are bugs in an existing Ansible module