Dell PowerEdge: Getting Started with Redfish Ansible Modules

August 2022

H19306

White Paper

Abstract

This document provides guidelines for getting started with Ansible and Redfish Ansible modules for PowerEdge systems. Techniques, practices, and the Ansible foundation described in this paper apply to other Dell Ansible modules and Ansible in general.

Dell Technologies

Copyright

Published in the USA in August 2022 H19306.

Contents

Executive summary

Overview

Infrastructure as Code (IaC) is a popular paradigm for managing IT infrastructures that range from small to extremely large and from simple to complex. The promise of IaC is to use code as a way of creating repeatable tasks with repeatable results across a wide variety of IT components. To deliver on its promises, IaC uses tools such as Puppet, Chef, Terraform, and Ansible. This white paper focuses on Ansible because it is widely used, requires no specific agents, and can be installed on a wide variety of platforms.

Ansible is a popular open-source software solution for configuration management and automation of IT resources. Many vendors, including Dell Technologies, provide Ansible collections and modules to enable the management of your server, networking, and storage hardware. You also can combine functionalities from multiple vendors to complement Dell Technologies products, provide end-to-end configuration management, and deliver IaC.

This paper introduces Ansible and provides guidelines and examples for how to make the best use of the Dell Ansible modules for the Dell PowerEdge platform. The information in this paper is additional to the product guides and release notes. As module functionality is updated periodically, we will update this paper accordingly.

Audience

This white paper is intended for storage administrators, site reliability engineers, automation engineers, and DevOps engineers who want information about how to get started with Ansible, integrate PowerEdge servers into their infrastructure, and manage resources as code using Ansible. It is assumed that readers are familiar with PowerEdge architecture.

Revisions

Date	Description
August 2022	Initial release

We value your feedback

Dell Technologies and the authors of this document welcome your feedback on this document. Contact the Dell Technologies team by email.

Authors: Bertrand Sirodot

Note: For links to other documentation for this topic, see References.

Introduction to Ansible

Overview

Ansible is an open-source-provisioning, configuration-management, and application-deployment tool that enables IaC. Red Hat acquired Ansible in October 2015.

Ansible is popular for two primary reasons:

- It uses the prevalent YAML language to create human-readable templates with which users can program repetitive tasks to occur automatically, without the need to learn an advanced language.
- It does not require any agent or specific infrastructure; instead, it uses standard tools such as SSH to run its programs.

Both attributes make it easy to deploy and get started with Ansible.

In Ansible terminology, a list of tasks or scripts is called a playbook. A playbook describes the target end state of the configuration of the IT component being managed by Ansible. Playbooks are written in YAML and offer a repeatable, reusable, and simple configuration-management and multi-machine deployment system. Ansible does not use an internal database to store the content of the playbooks, making it easy to back them up. Most organizations store their playbooks within their version control tool.

Ansible is an extensible framework where the open-source community and vendors can create modules, also known as task plug-ins or library plug-ins. The plug-ins are reusable, stand-alone scripts that can be used by the Ansible API or by Ansible Playbooks. Modules are focused on offering functionalities for a specific configuration element. For instance, Ansible supports a user module, which allows playbooks to manage user configuration needs, such as changing passwords, creating users, changing a user's group ownership, or even automatically generating SSH keys for the user.

Modules pertaining to the same component can be aggregated in a collection. Dell Technologies offers various collections for each of its products, such as PowerEdge, PowerMax, PowerFlex, and so on. Ansible collections are available on the Ansible Galaxy website: https://galaxy.ansible.com. The Dell Technologies collections are available at https://galaxy.ansible.com/dellemc.

You can deploy Ansible in multiple ways, depending on your needs. In the following sections, we review a few of the most popular installation approaches. Later, we deploy Ansible as a Python package. This method offers the benefit of not requiring the installation of specific software, which makes it a great option for those who want to test Ansible. It is installed in the same way as any Python package, using the Python package installer, pip.

Note: This paper assumes some level of familiarity with Python and pip.

Requirements for using Ansible with PowerEdge

To manage a PowerEdge server with Ansible, you need the following components, as shown in Figure 1:

- **Integrated Dell Remote Access Controller (iDRAC)**—iDRAC is native to PowerEdge and provides the Redfish API interface for managing PowerEdge servers.
- **Authorized user for iDRAC**—To prevent unauthorized access to the iDRAC, Redfish requires authentication before Ansible performs any action. Ansible employs the authorized user while performing actions against the iDRAC.
- **Linux server**—This server can be either a virtual or physical machine, with Python 3.x or later installed. For exact versions, see the Ansible release notes. Currently, Ansible can manage Windows hosts but cannot be installed on Windows.
- **Dell OpenManage Python SDK**—This SDK is a Python library that automates the life cycle management of PowerEdge servers and modular infrastructure. It is available on Dell GitHub at https://github.com/dell/omsdk.
- **Dell OpenManage Ansible collection**—This Dell Ansible collection contains modules to automate and orchestrate the deployment and update of PowerEdge servers and modular infrastructure. It is available on Ansible Galaxy at https://galaxy.ansible.com/dellemc/openmanage.

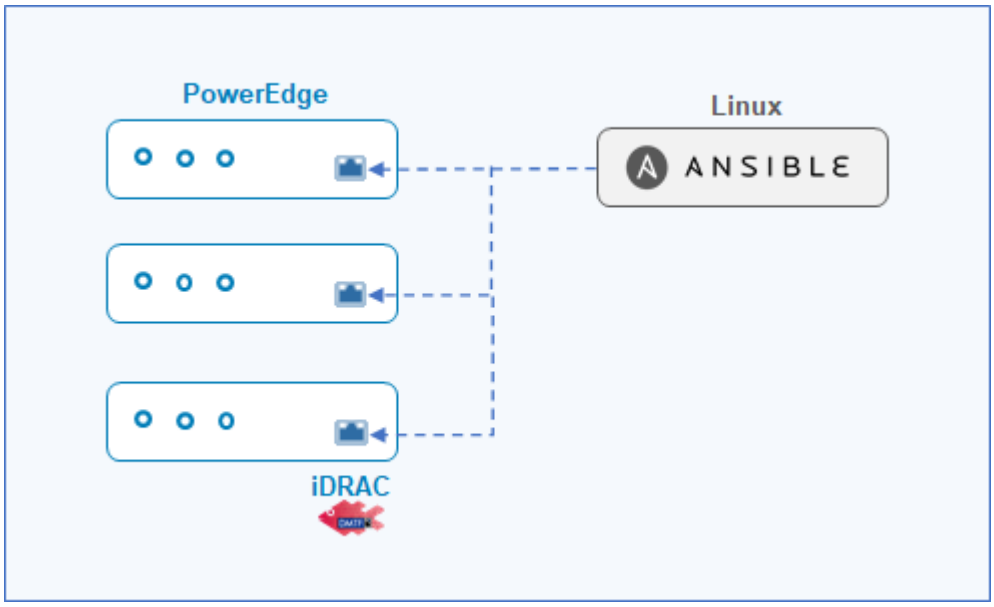


Figure 1. Components for managing PowerEdge server with Ansible

Obtaining official Dell content for Ansible

Ansible Galaxy is an online hosted repository where vendors and community members can upload content for extending Ansible base functionality to provide a simple method for delivering Ansible content.

All officially supported Dell modules are packaged as collections and are available on the Ansible Galaxy site at https://galaxy.Ansible.com/dellemc.

Ansible installation methods

There are multiple approaches to installing Ansible in your environment. A few of the most popular options are:

- Operating system native packages

  Ansible is available as a native package that can be installed onto various operating systems. Ansible cannot be installed on Windows but can manage Windows hosts. To run Ansible on Windows, you can use Windows Subsystem for Linux (WSL), which allows you to run a Linux environment inside Windows.

  Note: How to use WSL is outside the scope of this paper.

- Basic install on a Linux server with Python and Ansible Python using Python package installer (pip)
- Python virtual environments

This paper focuses on installation using Python virtual environments. That option is the easiest to use when testing Ansible because uninstalling Python is as easy as removing the directory for the virtual environment.

Installing Ansible and OpenManage collection in a virtual environment

To install Ansible on a Linux server in a directory called `PEAnsibleRedfish`, follow these steps. For examples, see Figure 2, Figure 3, Figure 4, and Figure 5.

Note: If you do not want to use virtual environments, you can skip steps 1 and 2. If you later decide to use virtual environments, you can activate the virtual environment upon login with an environment variable in the user's .bash_profile.

1. Create the Python virtual environment (venv) in the following directory:
   `PEAnsibleRedfish: python3 -m venv PEAnsibleRedfish`
2. Activate the Python venv:
   `source ./PEAnsibleRedFish/bin/activate`
3. Install pip, the Python package manager:
   `python3 -m pip install --upgrade pip`
4. Install Python Wheel:
   `pip install wheel setuptools`
5. Install the OpenManage Python SDK:
   `pip install omsdk`
6. Install Ansible:
   `pip install Ansible`
7. Install Ansible Lint:
   `pip install Ansible-lint`
8. Install OpenManage collection:
   `ansible-galaxy collection install dellemc.openmanage`

```
bertrand@W10780S2X2:~$ python3 -m venv PEAnsibleRedfish
bertrand@W10780S2X2:~$ source ./PEAnsibleRedfish/bin/activate
(PEAnsibleRedfish) bertrand@W10780S2X2:~$ python3 -m pip install --upgrade pip
Collecting pip
  Using cached pip-21.3.1-py3-none-any.whl (1.7 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 20.0.2
    Uninstalling pip-20.0.2:
      Successfully uninstalled pip-20.0.2
Successfully installed pip-21.3.1
(PEAnsibleRedfish) bertrand@W10780S2X2:~$ pip install wheel setuptools
Collecting wheel
  Using cached wheel-0.37.1-py2.py3-none-any.whl (35 kB)
Requirement already satisfied: setuptools in ./PEAnsibleRedfish/lib/python3.8/site-packages (44.0.0)
Installing collected packages: wheel
Successfully installed wheel-0.37.1
(PEAnsibleRedfish) bertrand@W10780S2X2:~$
```

Figure 2. Preparing your virtual environment

```
(PEAnsibleRedfish) bertrand@W10780S2X2:~$ pip install omsdk
Collecting omsdk
  Using cached omsdk-1.2.488-py2.py3-none-any.whl (830 kB)
Requirement already satisfied: ipaddress>=0 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from omsdk) (1.0.23)
Requirement already satisfied: future>=0.16.0 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from omsdk) (0.18.2)
Requirement already satisfied: PyYAML>=3.12 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from omsdk) (6.0)
Requirement already satisfied: requests>=2.12.3 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from omsdk) (2.27.1)
Requirement already satisfied: pysnmp-mibs>=0 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from omsdk) (0.1.6)
Requirement already satisfied: pysnmp>=4.3.0 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from pysnmp-mibs>=0->oms
dk) (4.4.12)
Requirement already satisfied: certifi>=2017.4.17 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from requests>=2.12
.3->omsdk) (2021.10.8)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from requests>=2
.12.3->omsdk) (1.26.8)
Requirement already satisfied: idna<4,>=2.5 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from requests>=2.12.3->om
sdk) (3.3)
Requirement already satisfied: charset-normalizer~=2.0.0 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from request
s>=2.12.3->omsdk) (2.0.10)
Requirement already satisfied: pysmi in ./PEAnsibleRedfish/lib/python3.8/site-packages (from pysnmp>=4.3.0->pysnmp-mibs>
=0->omsdk) (0.3.4)
Requirement already satisfied: pyasn1>=0.2.3 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from pysnmp>=4.3.0->pysn
mp-mibs>=0->omsdk) (0.4.8)
Requirement already satisfied: pycryptodomex in ./PEAnsibleRedfish/lib/python3.8/site-packages (from pysnmp>=4.3.0->pysn
mp-mibs>=0->omsdk) (3.13.0)
Requirement already satisfied: ply in ./PEAnsibleRedfish/lib/python3.8/site-packages (from pysmi->pysnmp>=4.3.0->pysnmp-
mibs>=0->omsdk) (3.11)
Installing collected packages: omsdk
Successfully installed omsdk-1.2.488
(PEAnsibleRedfish) bertrand@W10780S2X2:~$
```

Figure 3. Installing omsdk package

```
(PEAnsibleRedfish) bertrand@W10780S2X2:~$ pip install Ansible
Collecting Ansible
  Using cached ansible-5.2.0.tar.gz (37.9 MB)
  Preparing metadata (setup.py) ... done
Collecting ansible-core~=2.12.1
  Using cached ansible-core-2.12.1.tar.gz (7.4 MB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: PyYAML in ./PEAnsibleRedfish/lib/python3.8/site-packages (from ansible-core~=2.12.1->Ansi
ble) (6.0)
Collecting cryptography
  Using cached cryptography-36.0.1-cp36-abi3-manylinux_2_24_x86_64.whl (3.6 MB)
Collecting jinja2
  Using cached Jinja2-3.0.3-py3-none-any.whl (133 kB)
Collecting packaging
  Using cached packaging-21.3-py3-none-any.whl (40 kB)
Collecting resolvelib<0.6.0,>=0.5.3
  Using cached resolvelib-0.5.4-py2.py3-none-any.whl (12 kB)
Collecting cffi>=1.12
  Using cached cffi-1.15.0-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (446 kB)
Collecting MarkupSafe>=2.0
  Using cached MarkupSafe-2.0.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_12_x86_64.manylinux2010_x86
_64.whl (30 kB)
Collecting pyparsing!=3.0.5,>=2.0.2
  Using cached pyparsing-3.0.7-py3-none-any.whl (98 kB)
Collecting pycparser
  Using cached pycparser-2.21-py2.py3-none-any.whl (118 kB)
Building wheels for collected packages: Ansible, ansible-core
  Building wheel for Ansible (setup.py) ... done
  Created wheel for Ansible: filename=ansible-5.2.0-py3-none-any.whl size=62990786 sha256=e4e3de7e88d8ad59503302fd177ab5
c8eae81524327b9a29b9cd410be95ebb3d
  Stored in directory: /home/bertrand/.cache/pip/wheels/d8/ee/28/bc3d171cb674cdc6ab434127534b87e5e89956f9e6c0a71af2
  Building wheel for ansible-core (setup.py) ... done
  Created wheel for ansible-core: filename=ansible_core-2.12.1-py3-none-any.whl size=2073413 sha256=22c9c839d357912a9862
b9c7ad10fbf32c5f455aad3e7e0adbe85ba5f5d1f6dd
  Stored in directory: /home/bertrand/.cache/pip/wheels/34/bd/63/4f3348987a1079c559b4f10f5a8460784d8ac803d46e762d87
Successfully built Ansible ansible-core
Installing collected packages: pycparser, pyparsing, MarkupSafe, cffi, resolvelib, packaging, jinja2, cryptography, ansi
ble-core, Ansible
Successfully installed Ansible-5.2.0 MarkupSafe-2.0.1 ansible-core-2.12.1 cffi-1.15.0 cryptography-36.0.1 jinja2-3.0.3 p
ackaging-21.3 pycparser-2.21 pyparsing-3.0.7 resolvelib-0.5.4
(PEAnsibleRedfish) bertrand@W10780S2X2:~$
```

Figure 4. Installing Ansible

```
(PEAnsibleRedfish) bertrand@W10780S2X2:~$ pip install Ansible-lint
Collecting Ansible-lint
  Using cached ansible_lint-5.3.2-py3-none-any.whl (115 kB)
Collecting tenacity
  Using cached tenacity-8.0.1-py3-none-any.whl (24 kB)
Collecting wcmatch>=7.0
  Using cached wcmatch-8.3-py3-none-any.whl (42 kB)
Requirement already satisfied: packaging in ./PEAnsibleRedfish/lib/python3.8/site-packages (from Ansible-lint) (21.3)
Collecting rich>=9.5.1
  Downloading rich-11.1.0-py3-none-any.whl (216 kB)
     |████████████████████████████████| 216 kB 1.7 MB/s
Requirement already satisfied: pyyaml in ./PEAnsibleRedfish/lib/python3.8/site-packages (from Ansible-lint) (6.0)
Collecting ruamel.yaml<1,>=0.15.37
  Using cached ruamel.yaml-0.17.20-py3-none-any.whl (109 kB)
Collecting enrich>=1.2.6
  Using cached enrich-1.2.7-py3-none-any.whl (8.7 kB)
Collecting colorama<0.5.0,>=0.4.0
  Using cached colorama-0.4.4-py2.py3-none-any.whl (16 kB)
Collecting pygments<3.0.0,>=2.6.0
  Using cached Pygments-2.11.2-py3-none-any.whl (1.1 MB)
Collecting commonmark<0.10.0,>=0.9.0
  Using cached commonmark-0.9.1-py2.py3-none-any.whl (51 kB)
Collecting ruamel.yaml.clib>=0.2.6
  Using cached ruamel.yaml.clib-0.2.6-cp38-cp38-manylinux1_x86_64.whl (570 kB)
Collecting bracex>=2.1.1
  Using cached bracex-2.2.1-py3-none-any.whl (12 kB)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in ./PEAnsibleRedfish/lib/python3.8/site-packages (from packagin
g->Ansible-lint) (3.0.7)
Installing collected packages: pygments, commonmark, colorama, ruamel.yaml.clib, rich, bracex, wcmatch, tenacity, ruamel
.yaml, enrich, Ansible-lint
Successfully installed Ansible-lint-5.3.2 bracex-2.2.1 colorama-0.4.4 commonmark-0.9.1 enrich-1.2.7 pygments-2.11.2 rich
-11.1.0 ruamel.yaml-0.17.20 ruamel.yaml.clib-0.2.6 tenacity-8.0.1 wcmatch-8.3
(PEAnsibleRedfish) bertrand@W10780S2X2:~$
```

Figure 5. Installing Ansible-lint

Upgrading your Ansible environment

Ansible playbooks are usually forward-compatible; however, retesting your playbooks with the latest software versions before releasing them to production is always advisable. Adopting continuous integration/continuous development (CI/CD) pipeline methodologies and tools can be useful and is generally recommended.

To upgrade an existing environment, create a new virtual environment and install the latest OpenManage SDK and OpenManage Ansible collection to this environment.

With multiple virtual environments configured, you can run playbooks in a validated virtual environment by switching environments. To make the switch, run the `source ./<environmentname>/bin/activate` command.

Basic Ansible commands

This paper uses the following basic Ansible commands:

- `Ansible-galaxy`—Performs various role and collection-related operations, including searching the Ansible Galaxy site for collections and installing them on your system.
- `Ansible-doc`—Provides documentation for usage and supported functionality of Ansible modules. Useful for any storage administrator, site reliability engineer, or Ansible administrator. The EXAMPLES section is particularly helpful.
- `Ansible-playbook`—Runs Ansible playbooks.
- `Ansible-vault`—Encrypts or decrypts files used by Ansible that contain sensitive data.
- `Ansible-lint`—Validates Ansible playbooks for syntax.

Ansible Galaxy

Ansible Galaxy is the Ansible equivalent of an app store. Ansible Galaxy is an online hosted repository where vendors and Ansible community members can upload content for extending Ansible base functionality. Ansible Galaxy provides a simple method for delivering Ansible content. Users can package and distribute playbooks, roles, modules, and plug-ins.

All officially supported Dell modules are packaged and distributed in Ansible collections, which are available on the Ansible Galaxy site at https://galaxy.Ansible.com/dellemc.

YAML—Yet Another Markup Language

Ansible is an end-state-driven automation engine that uses a markup language to describe the end state of the environments it creates or manipulates. Ansible playbooks use a markup language called YAML.

YAML, which is an acronym for Yet Another Markup Language, is descriptive as well as easy to read and understand. Interacting with YAML requires little or no programming skills. Some users might decide to implement conditionals checks once they become more familiar with YAML and Ansible constructs. YAML files can have the extension .yml or .yaml, depending on preference. The examples in this paper use the shorter .yml extension.

When working with YAML, you can use your favorite text editor or integrated development environment (IDE). If you are storing your configuration offerings in source control management, an IDE is recommended for checking versions in and out. IDEs also make variable tracking simpler and take care of indentation once the file type is specified as YAML.

When working with YAML, remember these factors:

- **Everything specified in YAML is a list**. A new list starts with a keyword followed by a colon (`:`). New list items are indicated with a dash (–) on the next line, indented by two spaces. Sub-items for a list value are indented on a new line by two spaces.
- **Spaces are important**. Working with YAML is easier if you set your editor to two spaces per tab.
- **You can save time by using VIM.** You can update your .vimrc file with the following options to work well with YAML and auto-indent the YAML code:

  ```
  autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab
  ```

  When pasting in VIM, you might want to use the F2 option to switch paste mode on and off to avoid incorrect formatting of indents.
- **Variables notation requires variables to be enclosed in curly brackets and double quotes, for example, `"{{ metro_dr_array }}."`** Good practice is to add a space before and after the variable name to make your file easier to read. `Ansible-lint` would fail without white spaces around the variable.

What is a playbook?

An Ansible playbook is a blueprint of automation tasks that run with limited or no human involvement. Playbooks are run on a set, group, or classification of hosts, which together make up an Ansible inventory. The following figure shows a basic playbook.

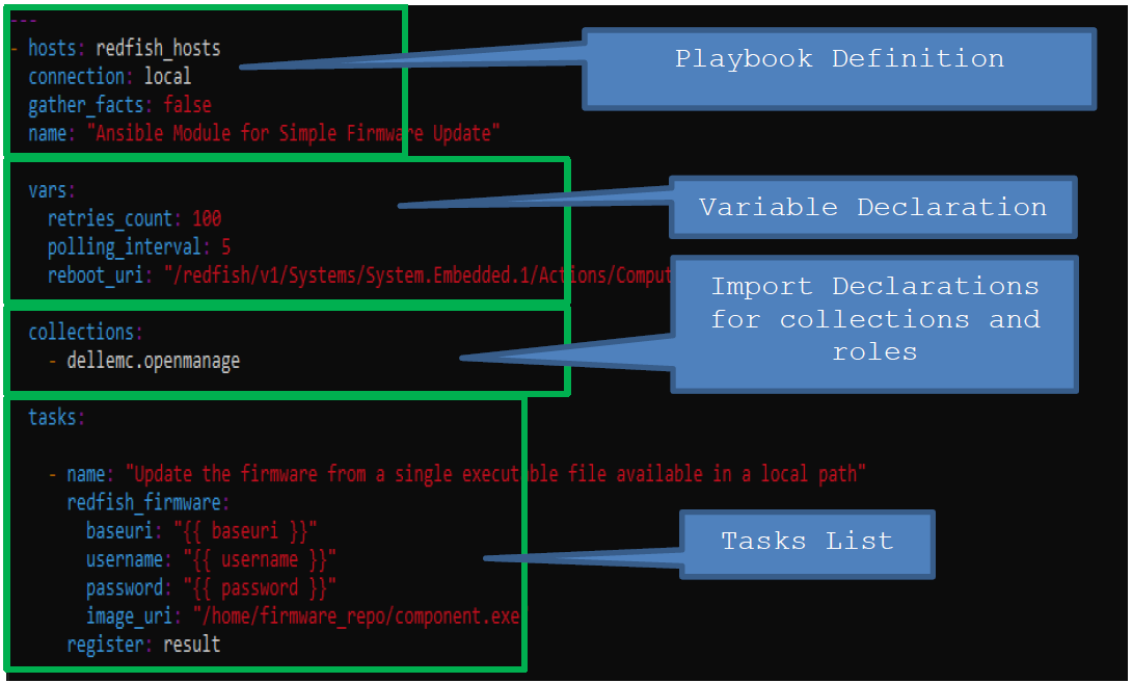


Figure 6. A basic playbook

A playbook usually includes the keywords described in the following table:

Table 1. Playbook keywords

Keyword	Description
`collections`	A keyword that imports any collections. Using this keyword to import the dellemc.openmanage namespace tells the playbook where to look for PowerEdge modules.
`connection`	A keyword that is usually set to `local` because the Ansible server runs the REST calls.
`gather_facts`	An optional keyword and one that is not necessary for running Dell modules. Gathering information about the configuration of the local host is unnecessary because the information will not be used in the playbook. Setting this keyword to `no` reduces runtime slightly,
`hosts`	A keyword that Identifies the host on which the playbook is run. Because Dell playbooks use REST APIs, this keyword is usually set to `localhost`, and the Ansible server runs the REST calls.
`input`	A special keyword known as an anchor, which is used to reference multiple values with a single variable referenced by &uni_connection_vars. Figure 6 shows credentials and connection variables short-handed with this technique. Subsequent tasks can refer to the grouped variables using the `<<:*uni_connection_vars` anchor.
`name`	An identifier for the playbook, which ideally gives a good indication of what the playbook is designed to accomplish. Names can be dynamic and include variables, as in the example, so runtime changes according to the values the playbook is run with.
`vars`	A keyword that details any variables that the playbook needs to run. This keyword is optional because variables can also be passed in at runtime on the command line.
`vars_files`	An optional keyword that details any variable files that provide information for the playbook to run.

Best Practice: Because the YAML file works like a scripting language, you can use comment lines as often as necessary. These lines are ignored during parsing. Comment lines start with the # symbol, which is followed by some text. Comment lines can be used to explain sections of a playbook to make them more easily understandable to others.

Idempotency and Ansible

Ansible is a declarative state engine at its core. Through tasks, playbooks describe the state to which the system is to be configured. The logic behind the modules that are run by the playbook tasks works to achieve the specified state.

A key principle of Ansible that makes it different from scripting is the concept of Idempotency. The official Ansible documentation describes Idempotency as follows: “An operation is idempotent if the result of performing it once is exactly the same as the result of performing it repeatedly without any intervening actions.” Thus, if you run a playbook with the same set of inputs, you should not expect it to make any changes on the system. Dell modules for OpenManage (and other Dell Technologies products) are written to conform with this type of idempotency.

Getting started with Ansible

Introduction

A significant part of Ansible's popularity is because it does not need a dedicated client or agent to perform its tasks. Instead, Ansible relies on default tools, primarily SSH, to run its tasks, so some configuration must be done to ensure that Ansible can access the right tool in the right way.

In the following sections, we look at how to properly configure the destination hosts for Ansible so that Ansible can perform actions on them. Most of these configurations are not necessary if you only want to run playbooks to perform actions using Redfish. However, we include the information for reference or for your use if you want Ansible to manage both the hardware and software layer on your server. If you only want to run playbooks for Redfish, proceed to Building your inventory file.

Setting up SSH for Ansible

SSH is a critical piece in any Ansible-managed and controlled infrastructure because Ansible relies on it to perform actions on the hosts. Not using dedicated clients is what makes Ansible easy to use and implement, but it creates the requirement of having a robust and properly configured SSH infrastructure.

The biggest part of getting SSH ready for Ansible is enabling key-based authentication, which allows Ansible to access the remote hosts without the need for a password being entered.

If you are going to use Ansible solely to manage PowerEdge iDRACs through Redfish, you do not have to set up SSH. Redfish uses its own authentication method and does not rely on SSH. However, you must set up SSH key-based authentication to run playbooks against any Windows or Linux server.

In SSH, the source and the destination of the SSH session conduct key-based authentication by sharing public and private keys to authenticate the session, instead of using a password. Enable key-based authentication from the server, workstation, or desktop from which you will run Ansible by following these steps:

1. If you are going to use a dedicated user, such as `ansible`, to run the playbooks, `su` to that user.
2. Create your private/public key pair by running `ssh-keygen`, and then follow the prompts.

   We recommend that you enter a passphrase when prompted. The passphrase protects the private key and limits the possibilities of that private key being compromised.
3. Copy your public key to each server you want to run Ansible playbooks against by running the command `ssh-copy-id <username>@<remote host>`.

   This command copies the public key to `<remote host>` and establishes key-based authentication for user `<username>`.

Building your inventory file

After you have set up key-based authentication for SSH, the next part of getting ready to run Ansible playbooks is to define an inventory file. In Ansible, the inventory file contains the list of hosts against which playbooks can be run. The challenge is that certain playbooks might only be run against specific hosts and not every host, so how is that achieved within the inventory file?

Fortunately, the Ansible inventory file supports the creation of host groups with associated variables. Ansible supports two formats for its inventory file: INI and YAML.

Here is an example of an inventory file in INI format:

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

And here is the same inventory file in YAML format:

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
```

The inventory file format is flexible enough that a host can be part of multiple groups. This capability allows system administrators to build groups based on application, location, or deployment environment (dev, test, prod, and so on). Hosts can be a member of all the groups simultaneously.

Groups can also be defined as a hierarchy, where a group can be made of one or more groups. For instance, the group `awesomeapp` can be made of groups `webservers`, `databases`, and `applications`. For example:

```
[awesomeapp]
webservers
databases
applications
```

with group `webservers` defined as follows:

```
[webservers]
webserver01
webserver02
```

Similarly, for groups `databases` and `applications`:

```
[databases]
dbsrv01
dbsrv02
[applications]
appsrv01
appsrv02
```

Hosts can also be defined using ranges. For instance, the following example defines a group called `webservers`, containing all the servers called `webserver01` to `webserver50`:

```
[webservers]
webserver[01:50]
```

Two types of variables can be in the inventory file: host variables and group variables. Host variables allow customization for a specific host, whereas group variables allow the definition of variables to be applied to every host in the group.

The following example shows the definition of a host variable:

```
[webservers]
webserver01 http_port=80
webserver02 http_port=8080
```

This example allows a system administrator to customize the port for HTTP requests for each host. For a host where SSH is running on a port other than port 22, it is possible to specify which port Ansible should use by defining the host like this: `webserver01:8022`. This specification instructs Ansible to use port 8022 to connect to this web server.

You can also define group-level variables, as follows:

```
[webservers:vars]
ntp_server=ntpserver.acme.com
username=username
password=password
```

These variables are used for all servers in the `webservers` group. With iDRAC, you can define the username and password to be used to connect to iDRAC, as previously shown. Because this method poses some security risk, ensure that the permissions on the inventory file are as closed as possible.

Ansible defines a multitude of variables for the inventory file, which we do not fully address in this paper. For more information about this topic, see the following Ansible documentation: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#inventory-basics-formats-hosts-and-groups

Running your first playbook

Now that we have established key-based authentication for SSH and defined the inventory file, we can safely test that Ansible is working. To do that, we could run a playbook, but the fastest and easiest way to test Ansible is to run the ping command:

```
(PEAnsibleRedfish) bertrand@W10780S2X2:~/redfish-ansible-module$ ansible -i ./inventory.yml lab -m ping
192.168.1.58 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
192.168.1.100 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
(PEAnsibleRedfish) bertrand@W10780S2X2:~/redfish-ansible-module$
```

This command looks into the inventory file, searches for a group called `lab`, and runs an Ansible `ping` command against each host. The Ansible ping command tests the connectivity to the host to ensure that it can log in using the SSH key-based authentication. It then checks that Python is installed on the host. This process ensures that all the pieces necessary for running playbooks are properly configured.

Managing PowerEdge servers through Redfish with Ansible

What is Redfish?

The Redfish standard is a suite of specifications that deliver an industry-standard protocol providing a RESTful interface for the management of servers.

Redfish can access iDRAC, which comes with every PowerEdge server. The iDRAC is a baseboard management controller (BMC) that is responsible for providing out-of-band, low-level management functions. Such functions include remotely rebooting a server, changing BIOS settings, upgrading firmware, getting server component information, streaming telemetry, and so on. The functionalities offered by the iDRAC can be accessed in various ways:

- HTML5 web interface
- IPMI
- SNMP
- Redfish
- SSH
- Racadm

In environments where IaC is the standard infrastructure management paradigm, Redfish is gaining increasing traction due to its security features, standard RESTful API interface, and integration with IaC management frameworks such as Ansible.

Dell Redfish Ansible modules

Our GitHub site, https://www.github.com/dell, includes a couple of repositories that are dedicated to managing Dell PowerEdge servers using Redfish:

- iDRAC-Redfish-Scripting
- redfish-ansible-module

This paper focuses on the redfish-ansible-module repository, which includes all the playbooks that we discuss later in this paper. We recommend that you clone the redfish-ansible-module repository, using `git clone https://www.github.com/dell/redfish-ansible-module`, on the server, workstation, or virtual machine from which you are running Ansible.

The playbooks and roles defined in the repository use several community-supported Ansible modules—in particular, the redfish_info, redfish_command, redfish_config, idrac_redfish_info, idrac_redfish_command, and idrac_redfish_config modules. These modules are automatically installed when you install the `dellemc.openmanage` collection.

Running any playbook from this repository requires a couple of configuration items to be set up in the inventory file in the `redfish-ansible-module` directory. The repository comes with an example of the inventory file, but the two configuration items required are:

- A group of hosts, with each host requiring defining of the `baseuri` variable:

  ```
  [myhosts]
  Host01 baseuri=idrac-host01
  Host02 baseuri=idrac-host02
  ```

  The `baseuri` variable must be set to the DNS entry for the iDRAC of the host. The Redfish modules use the variable to connect to the iDRAC.

- iDRAC username and password for authentication. These items are typically set as group variables so they can be used to connect to all the hosts in the group:

  ```
  [myhosts:vars]
  username=<username>
  password=<password>
  ```

  Because the inventory file contains plain text usernames and passwords, take care to set the proper permission to avoid wrongful access.

After these configuration items are set, you can run playbooks. All the playbooks in this repository store their results under the `inventory_files` directory in the home directory of the user running the `ansible-playbook` command. The `inventory_files` directory contains a subdirectory for each of the hosts the playbook runs against, with the result of the playbook being stored in a JSON file.

Getting PowerEdge server inventory

The `redfish-ansible-module` repository comes with a myriad of playbook examples to perform various tasks, such as getting the hardware inventory, power cycling, creating users, enabling lockdown mode, and so on.

Here is the playbook to get the complete inventory of a server:

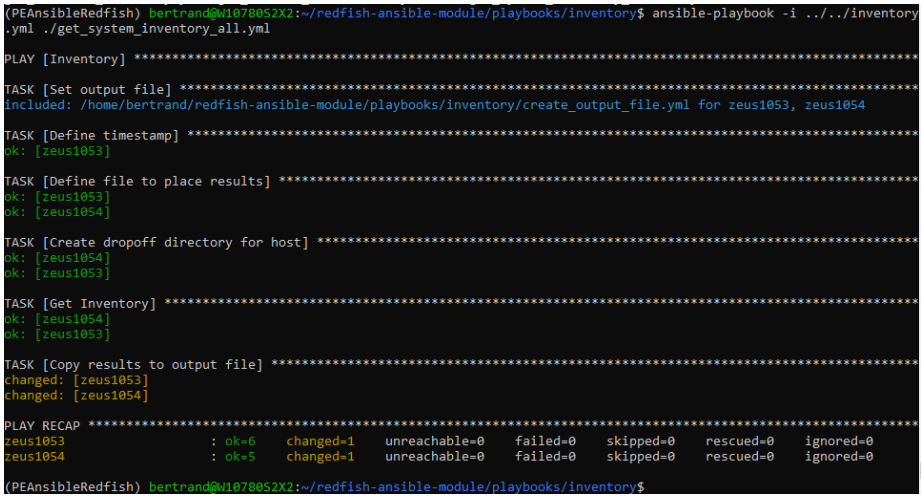
```
---
- hosts: myhosts
  connection: local
  name: Inventory
  gather_facts: False

  vars:
    ansible_python_interpreter: "/usr/bin/env python"
```

```
    datatype: SystemAll


  tasks:

  - name: Set output file
    include_tasks: create_output_file.yml

  - name: Get Inventory
    community.general.redfish_info:
      category: Systems
      command: all
      baseuri: "{{ baseuri }}"
      username: "{{ username}}"
      password: "{{ password }}"
    register: result

  - name: Copy results to output file
    ansible.builtin.copy:
      content: "{{ result | to_nice_json }}"
      dest: "{{ template }}.json"
```

This playbook proceeds as follows:

1. Looks for a list of hosts called `myhosts` in the inventory file supplied in the command line
2. Creates the output file according to the `create_output_file.yml` file
3. Runs the command `all` found in the `community.general.redfish_info` module in the `Systems` category, using the supplied `baseuri`, `username`, and `password` variables
4. Formats the result of the `all` command to JSON and stores it in a file named according to the `template` variable, which is defined in the `create_output_file.yml`

Running the `ansible-playbook -i <inventory file> ./get_system_inventory_all.yml` command creates the following output:



Looking at the output of the command, we can see that our inventory file contains two hosts: zeus1053 and zeus1054. The result of the command against zeus1053 is stored under the `~/inventory_files/zeus1053` in a file called `zeus1053_SystemAll_<timestamp>.json`. Similarly, the result for zeus1054 is stored under the `~/inventory_files/zeus1054` in a file called `zeus1054_SystemAll_<timestamp>.json`.

Each JSON file contains the complete hardware inventory of the server, including the part number of each memory DIMM, the component health status, and virtual disk information.

Changing a BIOS setting

The playbooks interacting with the BIOS are stored in the `redfish-ansible-module/playbooks/bios` directory. That directory includes playbooks to:

- Enable and disable PXE boot
- Get and set BIOS attributes
- Set the boot mode
- Reset default BIOS settings
- Set a one-time boot value, such as booting one time from virtual media to install a server

First, we get the current value of all the BIOS attributes on our servers by running the `get_bios_attributes.yml` playbook:

```
(PEAnsibleRedfish) bertrand@W10780S2X2:~/redfish-ansible-module/playbooks/bios$ ansible-playbook -i ../../inventory.yml
./get_bios_attributes.yml

PLAY [Get BIOS attributes] ***************************************************************************************************

TASK [Define output file] ****************************************************************************************************
included: /home/bertrand/redfish-ansible-module/playbooks/bios/create_output_file.yml for zeus1053, zeus1054

TASK [Define timestamp] ******************************************************************************************************
ok: [zeus1053]

TASK [Define file to place results] ******************************************************************************************
ok: [zeus1053]
ok: [zeus1054]

TASK [Create dropoff directory for host] *************************************************************************************
ok: [zeus1053]
ok: [zeus1054]

TASK [Get BIOS attributes] ***************************************************************************************************
ok: [zeus1054]
ok: [zeus1053]

TASK [Copy results to output file] *******************************************************************************************
changed: [zeus1054]
changed: [zeus1053]

PLAY RECAP *******************************************************************************************************************
zeus1053                   : ok=6    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
zeus1054                   : ok=5    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(PEAnsibleRedfish) bertrand@W10780S2X2:~/redfish-ansible-module/playbooks/bios$
```

All the BIOS attributes are stored in this file for zeus1053:

~/inventory_files/zeus1053/zeus1053_BiosAttributes_<timestamp>.json

Here is an extract from the file:

```
{
    "changed": false,
    "failed": false,
    "redfish_facts": {
```

```
            "bios_attribute": {
                "entries": [
                    [
                        {
                            "system_uri":
"/redfish/v1/Systems/System.Embedded.1"
                        },
                        {
                            "AcPwrRcvry": "Last",
                            "AcPwrRcvryDelay": "Immediate",
                            "AcPwrRcvryUserDelay": 60,
                            "AesNi": "Enabled",
                            "AgesaVersion": "MilanPI-SP3 1.0.0.2",
                            "AmdMaxXgmiSpeed": "16GB",
                            "ApbDis": "Disabled",
                            "AssetTag": "",
                            "AuthorizeDeviceFirmware": "Disabled",
                            "BiosNvmeDriver": "DellQualifiedDrives",
                            "BootMode": "Bios",
                            "BootSeqRetry": "Enabled",
                            "NumLock": "On",
                        }
                    ]
                ],
                "ret": true
            }
        }
}
```


In this section, we change two attributes in the BIOS: `BootMode` and `NumLock`. Changing `BootMode` from `bios` to `uefi` is as simple as running the provided playbook, called `set_bootmode_uefi.yml`, as follows:

```
(PEAnsibleRedfish) bertrand@W10780S2X2:~/redfish-ansible-module/playbooks/bios$ ansible-playbook -i ../../inventory.yml
./set_bootmode_uefi.yml

PLAY [Set boot mode to UEFI and reboot] ****************************************************************************

TASK [Set Bios boot mode to Uefi] **********************************************************************************
changed: [zeus1053]
changed: [zeus1054]

TASK [Create BIOS configuration job (schedule BIOS setting update)] ************************************************
changed: [zeus1053]
changed: [zeus1054]

TASK [Reboot system to apply new BIOS settings] ********************************************************************
changed: [zeus1053]
changed: [zeus1054]

PLAY RECAP *********************************************************************************************************
zeus1053                   : ok=3    changed=3    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
zeus1054                   : ok=3    changed=3    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

(PEAnsibleRedfish) bertrand@W10780S2X2:~/redfish-ansible-module/playbooks/bios$
```

If we run the `get_bios_attributes.yml` playbook again, we can see that the `BootMode` attribute is now set to `Uefi`:


```
{
    "changed": false,
    "failed": false,
    "redfish_facts": {
        "bios_attribute": {
            "entries": [
                [
                    {
                        "system_uri":
"/redfish/v1/Systems/System.Embedded.1"
                    },
                    {
                        "AcPwrRcvry": "Last",
                        "AcPwrRcvryDelay": "Immediate",
                        "AcPwrRcvryUserDelay": 60,
                        "AesNi": "Enabled",
                        "AgesaVersion": "MilanPI-SP3 1.0.0.2",
                        "AmdMaxXgmiSpeed": "16GB",
                        "ApbDis": "Disabled",
                        "AssetTag": "",
                        "AuthorizeDeviceFirmware": "Disabled",
                        "BiosNvmeDriver": "DellQualifiedDrives",
                        "BootMode": "Uefi",
                        "BootSeqRetry": "Enabled",
                        "NumLock": "On",
                    }
                ]
            ],
            "ret": true
        }
    }
}
```


The new value is set, and the change appears after the server reboots.

As you can see, changing the boot mode is easy because of the provided playbook. If you want to change a value for which you do not have a specific playbook, you can create one. For instance, to switch NumLock on and off, we have created a playbook called `set_bios_attribute.yml`:

```
---
- hosts: myhosts
  connection: local
  name: Set BIOS attributes
  gather_facts: False

  vars:
    bios_attributes:
```

```
      SriovGlobalEnable: "Enabled"

  tasks:

  - name: Set BIOS attribute
    community.general.redfish_config:
      category: Systems
      command: SetBiosAttributes
      bios_attributes: "{{ bios_attributes}}"
      baseuri: "{{ baseuri }}"
      username: "{{ username }}"
      password: "{{ password }}"
    register: bios_attribute

  - name: Create BIOS configuration job (schedule BIOS setting update)
    community.general.idrac_redfish_command:
      category: Systems
      command: CreateBiosConfigJob
      baseuri: "{{ baseuri }}"
      username: "{{ username }}"
      password: "{{ password }}"
    when: bios_attribute.changed

  - name: Reboot system to apply new BIOS settings
    community.general.redfish_command:
      category: Systems
      command: PowerReboot
      baseuri: "{{ baseuri }}"
      username: "{{ username }}"
      password: "{{ password }}"
    when: bios_attribute.changed
```

The following section is what allows this playbook to change any BIOS attribute:

```
  vars:
    bios_attributes:
      SriovGlobalEnable: "Enabled"
```

That section creates a variable called `bios_attributes` with a value of `SriovGlobalEnable`: "`Enabled`". Changing a specific BIOS attribute is as simple as changing the value of the `bios_attributes` variable. For instance, to set the NumLock to off, change the preceding section to:

```
  vars:
    bios_attributes:
      NumLock: "Off"
```

After the change, the playbook file looks like this:

```
---
- hosts: myhosts
  connection: local
```

```
name: Set BIOS attributes
gather_facts: False

vars:
  bios_attributes:
    NumLock: "Off"

tasks:

- name: Set BIOS attribute
  community.general.redfish_config:
    category: Systems
    command: SetBiosAttributes
    bios_attributes: "{{ bios_attributes}}"
    baseuri: "{{ baseuri }}"
    username: "{{ username }}"
    password: "{{ password }}"
  register: bios_attribute

- name: Create BIOS configuration job (schedule BIOS setting update)
  community.general.idrac_redfish_command:
    category: Systems
    command: CreateBiosConfigJob
    baseuri: "{{ baseuri }}"
    username: "{{ username }}"
    password: "{{ password }}"
  when: bios_attribute.changed

- name: Reboot system to apply new BIOS settings
  community.general.redfish_command:
    category: Systems
    command: PowerReboot
    baseuri: "{{ baseuri }}"
    username: "{{ username }}"
    password: "{{ password }}"
  when: bios_attribute.changed
```

Changing a BIOS attribute is an asynchronous operation, meaning that the playbook returns before the change is made because the playbook itself does not make the change. Instead, it sets the new value and then creates a BIOS configuration job within the iDRAC. The configuration job commits the new value to the BIOS. The new value is only visible after the configuration job is completed and the server reboots.

As we wait for the job to be completed and the server to reboot, we can see that the `NumLock` attribute is now set to `Off`:

```
{
    "changed": false,
    "failed": false,
    "redfish_facts": {
        "bios_attribute": {
```

```
            "entries": [
                [
                    {
                        "system_uri":
"/redfish/v1/Systems/System.Embedded.1"
                    },
                    {
                        "AcPwrRcvry": "Last",
                        "AcPwrRcvryDelay": "Immediate",
                        "AcPwrRcvryUserDelay": 60,
                        "AesNi": "Enabled",
                        "AgesaVersion": "MilanPI-SP3 1.0.0.2",
                        "AmdMaxXgmiSpeed": "16GB",
                        "ApbDis": "Disabled",
                        "AssetTag": "",
                        "AuthorizeDeviceFirmware": "Disabled",
                        "BiosNvmeDriver": "DellQualifiedDrives",
                        "BootMode": "Uefi",
                        "BootSeqRetry": "Enabled",
                        "NumLock": "Off",
                    }
                ]
            ],
            "ret": true
        }
    }
}
```


This example shows how you can use an Ansible playbook to change BIOS attributes across many servers.

Upgrading the firmware of server components

As shown by vulnerabilities such as Spectre and Meltdown, being able to apply firmware upgrades in a timely fashion is critical to any IT infrastructure. Redfish and Ansible enable the automation of firmware upgrades of any components, such as network cards, PERC RAID cards, the server BIOS, or even the iDRAC itself. However, only a single firmware can be updated at a time, so to upgrade multiple firmware, you must duplicate the upgrade tasks for each of them.

In this example, we update the firmware of the PERC RAID controller by using the following playbook. The playbook assumes that each host in the inventory defines the `baseuri` variable, as previously described.

```
---
- hosts: testhosts
  name: Update firmware of PERC Card
  connection: local
  gather_facts: False

  vars:
    ansible_python_interpreter: "/usr/bin/env python"
```

```
    retries_count: 180
    polling_interval: 5
    username: <iDRAC user>
    password: <iDRAC user password>
    reboot_uri:
"/redfish/v1/Systems/System.Embedded.1/Actions/ComputerSystem.Reset"
    perc_firmware: "/home/msanders/Downloads/SAS-
RAID_Firmware_6MTTK_WN64_52.16.1-4158_A05_01.EXE"

  collections:
    - dellemc.openmanage

  tasks:

    - name: "Upload new PERC firmware"
      redfish_firmware:
        baseuri: "{{ baseuri }}"
        username: "{{ username }}"
        password: "{{ password }}"
        image_uri: "{{ perc_firmware }}"
        validate_certs: no
      register: result

    - name: "Track PERC upload job to completion"
      uri:
        url: "https://{{ baseuri }}{{ result.task.uri }}"
        user: "{{ username }}"
        password: "{{ password }}"
        method: "GET"
        use_proxy: yes
        status_code: 200, 202
        return_content: yes
        validate_certs: no
        force_basic_auth: yes
        headers:
          Content-Type: "application/json"
          Accept: "application/json"
      register: job_result
      until: job_result.json.TaskState == 'Completed' or
job_result.json.TaskState == 'Starting'
      retries: "{{ retries_count }}"
      delay: "{{ polling_interval }}"

    - name: "Reboot the server to update PERC firmware"
      uri:
        url: "https://{{ baseuri }}{{ reboot_uri }}"
        user: "{{ username }}"
        password: "{{ password }}"
        method: "POST"
        body_format: raw
```

```
        body: '{"ResetType": "ForceRestart"}'
        use_proxy: yes
        status_code: 204
        return_content: no
        validate_certs: no
        force_basic_auth: yes
        headers:
          Content-Type: "application/json"
          Accept: "application/json"
      register: reboot_result
      changed_when: reboot_result.status == 204
      when: job_result.json.TaskState == 'Starting' and
job_result.json.Messages.0.Message == 'Task successfully scheduled.'

    - name: "Wait 5mins for PERC firmware to be applied"
      wait_for:
        timeout: 300
      when: job_result.json.TaskState == 'Starting' and
job_result.json.Messages.0.Message == 'Task successfully scheduled.'

    - name: "Track PERC firmware update job"
      uri:
        url: "https://{{ baseuri }}{{ result.task.uri }}"
        user: "{{ username }}"
        password: "{{ password }}"
        method: "GET"
        use_proxy: yes
        status_code: 200, 202
        return_content: yes
        validate_certs: no
        force_basic_auth: yes
        headers:
          Content-Type: "application/json"
          Accept: "application/json"
      register: final_result
      until: final_result.json.TaskState == 'Completed'
      retries: "{{ retries_count }}"
      delay: "{{ polling_interval }}"

    - name: "Fact from PERC firmware upgrade"
      set_fact:
        job_details: "{{ final_result.json }}"
      failed_when: final_result.json.TaskState == "Completed" and
final_result.json.TaskStatus != "OK"
      changed_when: final_result.json.TaskState == "Completed" and
final_result.json.TaskStatus == "OK"
```

A dissection of the playbook, starting with the `vars` section, follows:

```
  vars:
    ansible_python_interpreter: "/usr/bin/env python"
```

```
    retries_count: 180
    polling_interval: 5
    username: <iDRAC user>
    password: <iDRAC user password>
    reboot_uri:
"/redfish/v1/Systems/System.Embedded.1/Actions/ComputerSystem.Reset"
    perc_firmware: "/home/user1/Downloads/SAS-
RAID_Firmware_6MTTK_WN64_52.16.1-4158_A05_01.EXE"
```

The `retries_count` and `polling_interval` variables are used when we have to wait for a task to finish. The variable `retries_count` specifies how many times we want to retry, while `polling_interval` specifies how often we should check to see if the task has finished.

Next, we define the `username` and `password` to be used to log in to the iDRAC.

After each firmware update, the server must be rebooted. If you are performing multiple updates, defining a variable for the reboot URI can avoid significant typing and potential errors, which is why we have defined the `reboot_uri` variable here.

The final variable is `perc_firmware`, which points to the firmware package on the host from which the Ansible playbook will run. Firmware for PowerEdge servers is offered in either a .BIN package or a .EXE package. If you perform updates through the iDRAC web interface, the .BIN package must be used, but for updates performed using Redfish, the .EXE is required.

The required tasks to update the firmware are as follows:

- Task #1:

```
- name: "Upload new PERC firmware"
  redfish_firmware:
    baseuri: "{{ baseuri }}"
    username: "{{ username }}"
    password: "{{ password }}"
    image_uri: "{{ perc_firmware }}"
    validate_certs: no
  register: result
```

  This task starts a job to upload the firmware to the iDRAC. Depending on the size of the package and the speed of the network, this task can take a few minutes; however, it starts the job and returns as soon the job is started.

- Task #2:

```
- name: "Track PERC upload job to completion"
      uri:
        url: "https://{{ baseuri }}{{ result.task.uri }}"
        user: "{{ username }}"
        password: "{{ password }}"
        method: "GET"
        use_proxy: yes
        status_code: 200, 202
        return_content: yes
        validate_certs: no
```

```
        force_basic_auth: yes
        headers:
          Content-Type: "application/json"
          Accept: "application/json"
      register: job_result
      until: job_result.json.TaskState == 'Completed' or
job_result.json.TaskState == 'Starting'
      retries: "{{ retries_count }}"
      delay: "{{ polling_interval }}"
```

Because the previous task starts the upload job but does not wait for the job to be finished before returning, this task performs the necessary work of checking the status of the job. It polls the specified URI at every `polling_interval` until the number of retries equals `retries_count` or until the state of the job running on the iDRAC is either `'Completed'` or `'Starting'`. If the number of retries is reached and the state of the job is neither `'Completed'` nor `'Starting'`, the task fails.

- Task #3:

```
    - name: "Reboot the server to update PERC firmware"
      uri:
        url: "https://{{ baseuri }}{{ reboot_uri }}"
        user: "{{ username }}"
        password: "{{ password }}"
        method: "POST"
        body_format: raw
        body: '{"ResetType": "ForceRestart"}'
        use_proxy: yes
        status_code: 204
        return_content: no
        validate_certs: no
        force_basic_auth: yes
        headers:
          Content-Type: "application/json"
          Accept: "application/json"
      register: reboot_result
      changed_when: reboot_result.status == 204
      when: job_result.json.TaskState == 'Starting' and
job_result.json.Messages.0.Message == 'Task successfully
scheduled.'
```

After the firmware update job is started on the iDRAC, the server must reboot before the Lifecycle Controller performs the update. The next task is to reboot the server.

- Task #4:

```
    - name: "Wait 5mins for PERC firmware to be applied"
      wait_for:
        timeout: 300
     when: job_result.json.TaskState == 'Starting' and
job_result.json.Messages.0.Message == 'Task successfully
scheduled.'
```

Task #4 and task #5 are similar in that they are both waiting for the new firmware to be applied. We could have removed task #4 and started polling the server immediately, but instead we wait for 5 minutes before starting to poll the server. We take this approach because we know that the rebooting of the server and applying the new firmware will take some time. Depending on the server, this wait time could be adjusted or even removed.

- Task #5:

```
- name: "Track PERC firmware update job"
  uri:
    url: "https://{{ baseuri }}{{ result.task.uri }}"
    user: "{{ username }}"
    password: "{{ password }}"
    method: "GET"
    use_proxy: yes
    status_code: 200, 202
    return_content: yes
    validate_certs: no
    force_basic_auth: yes
    headers:
      Content-Type: "application/json"
      Accept: "application/json"
  register: final_result
  until: final_result.json.TaskState == 'Completed'
  retries: "{{ retries_count }}"
  delay: "{{ polling_interval }}"
```

As with task #2, this task polls the server to get the status of the iDRAC job and ensure that it is completed. It polls the specified URI at every `polling_interval` until the number of retries equals `retries_count` or until the state of the job running on the iDRAC is either `'Completed'` or `'Starting'`. If the number of retries is reached and the state of the job is neither `'Completed'` nor `'Starting'`, the task fails.

- Task #6:

```
    - name: "Fact from PERC firmware upgrade"
      set_fact:
        job_details: "{{ final_result.json }}"
      failed_when: final_result.json.TaskState == "Completed"
and final_result.json.TaskStatus != "OK"
      changed_when: final_result.json.TaskState == "Completed"
and final_result.json.TaskStatus == "OK"
```

This task gathers the facts of the previous tasks and show whether the update finished successfully or not.

Operating system deployment

This section describes how to use Ansible and Redfish to automate operating system deployment on a PowerEdge server. The example shows how to connect the virtual media to the server and have the server boot from it.

Note: Details about the installation of the operating system and its automation are operating-system dependent and are outside the scope of this paper.

Note: The playbook assumes that an ISO file containing the operating system installer has been downloaded and is available on an NFS or CIFS share. Downloading the ISO file and the creation of the NFS share are outside the scope of this paper.

In our example, we boot from an Ubuntu installation ISO. We assume that each host entry in the inventory file includes the `baseuri` variable, pointing to the iDRAC IP address for each server. We also assume that each host entry in the inventory file includes `idrac_user` and `idrac_password` variables containing the credentials to authenticate against the iDRAC.

```
---
- hosts: labhosts
  name: Install ubuntu 20.04.4
  gather_facts: False

  vars:
    ansible_python_interpreter: "/usr/bin/env python3"
    idrac_osd_command_allowable_values: ["BootToNetworkISO",
"GetAttachStatus", "DetachISOImage"]
    idrac_osd_command_default: "GetAttachStatus"
    GetAttachStatus_Code:
      DriversAttachStatus:
        "0": "NotAttached"
        "1": "Attached"
      ISOAttachStatus:
        "0": "NotAttached"
        "1": "Attached"
    idrac_https_port:           443
    expose_duration:            1080
    command:                    "{{ idrac_osd_command_default }}"
    validate_certs:             no
    force_basic_auth:           yes
    share_name:                 nfsserver:/home/user1/share
    ubuntu_iso:                 ubuntu-20.04.4-live-server-amd64.iso

  collections:
    - dellemc.openmanage

  tasks:
    - name: find the URL for the DellOSDeploymentService
      ansible.builtin.uri:
        url:
"https://{{ baseuri }}/redfish/v1/Systems/System.Embedded.1"
        user: "{{ idrac_user }}"
        password: "{{ idrac_password }}"
        method: GET
        headers:
          Accept: "application/json"
```

```
          OData-Version: "4.0"
        status_code: 200
        validate_certs: "{{ validate_certs }}"
        force_basic_auth: "{{ force_basic_auth }}"
      register: result
      delegate_to: localhost

    - name: find the URL for the DellOSDeploymentService
      ansible.builtin.set_fact:
        idrac_osd_service_url:
"{{ result.json.Links.Oem.Dell.DellOSDeploymentService['@odata.id'] }}
"
      when:
        - result.json.Links.Oem.Dell.DellOSDeploymentService is
defined

    - block:
        - name: get ISO attach status
          ansible.builtin.uri:
            url:
"https://{{ baseuri }}{{ idrac_osd_service_url }}/Actions/DellOSDeploy
mentService.GetAttachStatus"
            user: "{{ idrac_user }}"
            password: "{{ idrac_password }}"
            method: POST
            headers:
              Accept: "application/json"
              Content-Type: "application/json"
              OData-Version: "4.0"
            body: "{}"
            status_code: 200
            validate_certs: "{{ validate_certs }}"
            force_basic_auth: "{{ force_basic_auth }}"
          register: attach_status
          delegate_to: localhost

        - name: set ISO attach status as a fact variable
          ansible.builtin.set_fact:
            idrac_iso_attach_status: "{{ idrac_iso_attach_status |
default({}) | combine({item.key: item.value}) }}"
          with_dict:
            DriversAttachStatus:
"{{ attach_status.json.DriversAttachStatus }}"
            ISOAttachStatus:
"{{ attach_status.json.ISOAttachStatus }}"

      when:
        - idrac_osd_service_url is defined
        - idrac_osd_service_url|length > 0
```

```
    - block:
        - name: detach ISO image if attached
          ansible.builtin.uri:
            url:
"https://{{ baseuri }}{{ idrac_osd_service_url }}/Actions/DellOSDeploy
mentService.DetachISOImage"
            user: "{{ idrac_user }}"
            password: "{{ idrac_password }}"
            method: POST
            headers:
              Accept: "application/json"
              Content-Type: "application/json"
              OData-Version: "4.0"
            body: "{}"
            status_code: 200
            validate_certs: "{{ validate_certs }}"
            force_basic_auth: "{{ force_basic_auth }}"
          register: detach_status
          delegate_to: localhost

        - ansible.builtin.debug:
            msg: "Successfuly detached the ISO image"

      when:
        - idrac_osd_service_url is defined and
idrac_osd_service_url|length > 0
        - idrac_iso_attach_status
        - idrac_iso_attach_status.ISOAttachStatus == "Attached" or
          idrac_iso_attach_status.DriversAttachStatus == "Attached"

    - name: boot to network ISO
      dellemc.openmanage.idrac_os_deployment:
        idrac_ip: "{{ baseuri }}"
        idrac_user: "{{ idrac_user }}"
        idrac_password: "{{ idrac_password }}"
        share_name: "{{ share_name }}"
        iso_image: "{{ ubuntu_iso }}"
        expose_duration: "{{ expose_duration }}"
        validate_certs: False
      register: boot_to_network_iso_status
      delegate_to: localhost
```

A dissection of this playbook, except for the self-explanatory variable section, follows:

- Task number 1:

```
        - name: find the URL for the DellOSDeploymentService
          ansible.builtin.uri:
            url:
      "https://{{ baseuri }}/redfish/v1/Systems/System.Embedded.1"
            user: "{{ idrac_user }}"
```

```
        password: "{{ idrac_password }}"
        method: GET
        headers:
          Accept: "application/json"
          OData-Version: "4.0"
        status_code: 200
        validate_certs: "{{ validate_certs }}"
        force_basic_auth: "{{ force_basic_auth }}"
      register: result
      delegate_to: localhost

    - name: find the URL for the DellOSDeploymentService
      ansible.builtin.set_fact:
        idrac_osd_service_url:
"{{ result.json.Links.Oem.Dell.DellOSDeploymentService['@odata.i
d'] }}"
      when:
        - result.json.Links.Oem.Dell.DellOSDeploymentService is
defined
```

The first task is to get the Redfish URI for the operating system deployment service on the PowerEdge server. That URI is Dell specific, which is why it is in the OEM schema within Redfish. The URI is used in subsequent tasks to ascertain if an ISO image is already attached to the server.

- Task #2:

```
    - block:
        - name: get ISO attach status
          ansible.builtin.uri:
            url:
"https://{{ baseuri }}{{ idrac_osd_service_url }}/Actions/DellOS
DeploymentService.GetAttachStatus"
            user: "{{ idrac_user }}"
            password: "{{ idrac_password }}"
            method: POST
            headers:
              Accept: "application/json"
              Content-Type: "application/json"
              OData-Version: "4.0"
            body: "{}"
            status_code: 200
            validate_certs: "{{ validate_certs }}"
            force_basic_auth: "{{ force_basic_auth }}"
          register: attach_status
          delegate_to: localhost

        - name: set ISO attach status as a fact variable
          ansible.builtin.set_fact:
            idrac_iso_attach_status: "{{ idrac_iso_attach_status
| default({}) | combine({item.key: item.value}) }}"
```

```
          with_dict:
            DriversAttachStatus:
"{{ attach_status.json.DriversAttachStatus }}"
            ISOAttachStatus:
"{{ attach_status.json.ISOAttachStatus }}"

      when:
        - idrac_osd_service_url is defined
        - idrac_osd_service_url|length > 0
```

In this task, we use the URI found through task #1 to get the ISO attach status for the variable. To that end, we query the `DellOSDeploymentService.GetAttachStatus` endpoint and determine if an ISO image is already attached.

- Task #3:

```
    - block:
        - name: detach ISO image if attached
          ansible.builtin.uri:
            url:
"https://{{ baseuri }}{{ idrac_osd_service_url }}/Actions/DellOS
DeploymentService.DetachISOImage"
            user: "{{ idrac_user }}"
            password: "{{ idrac_password }}"
            method: POST
            headers:
              Accept: "application/json"
              Content-Type: "application/json"
              OData-Version: "4.0"
            body: "{}"
            status_code: 200
            validate_certs: "{{ validate_certs }}"
            force_basic_auth: "{{ force_basic_auth }}"
          register: detach_status
          delegate_to: localhost

        - ansible.builtin.debug:
            msg: "Successfuly detached the ISO image"

      when:
        - idrac_osd_service_url is defined and
idrac_osd_service_url|length > 0
        - idrac_iso_attach_status
        - idrac_iso_attach_status.ISOAttachStatus == "Attached"
or
          idrac_iso_attach_status.DriversAttachStatus ==
"Attached"
```

This task is optional but could be considered a safety measure. It detaches any ISO image that is already attached to the server, which ensures that the server does not boot from the wrong ISO image. In task #2, we created two facts (also known as variables):

`DriverAttachStatus` and `ISOAttachStatus`. Task #3 runs only if the status of either of these variables is `'Attached'`, meaning that an image is already attached to the server. If an image is attached, we detach it because the next task attaches the required ISO to the server.

- Task #4:

```
- name: boot to network ISO
  dellemc.openmanage.idrac_os_deployment:
    idrac_ip: "{{ baseuri }}"
    idrac_user: "{{ idrac_user }}"
    idrac_password: "{{ idrac_password }}"
    share_name: "{{ share_name }}"
    iso_image: "{{ ubuntu_iso }}"
    expose_duration: "{{ expose_duration }}"
    validate_certs: False
  register: boot_to_network_iso_status
  delegate_to: localhost
```

This task is where we boot the server from the ISO image. This task uses the `dellemc.openmanage` Ansible module. The idrac_os_deployment function within that module performs multiple actions on the iDRAC:

- Attaches the ISO image—an Ubuntu 20.04.4 ISO image in this example—on the `share_name` to the iDRAC. The image is exposed to the server for a duration of `expose_duration`.
- Sets the boot device as being the mounted ISO image.
- Reboots the server.

At the end of these tasks, the server boots from the ISO image and launches the operating system installer that is stored on the image. You can also use Ansible to automate operating system installation. That process is operating-system dependent and outside the scope of this paper.

Getting help

Dell Github

The Dell Github repositories are regularly updated with new and updated playbooks. We recommend that you always clone the latest repository before running critical tasks. If you experience any issue with any of the content in the various repositories, report the issue through Github.

Dell Community pages

The Dell Community pages are a great place to exchange ideas and to get help. Developers and Technical Marketing Engineering team members monitor these pages, which provide the ideal platform for discussions on using the content provided by Dell Technologies.

Dell Customer Support

Should you run into issues with the Ansible modules for Redfish, Customer Support is on hand to troubleshoot any issues and can escalate to development if necessary.

Ansible forum on Google Groups

The Ansible Project on Google Groups is a great resource to help new and experienced Ansible users get the most out of their automation. The forums are an open arena where like-minded individuals can benefit from a global talent pool.

Summary

This paper describes how to install Ansible, create Ansible playbooks, and use Redfish to automate system management activity on PowerEdge servers. You can use those technologies in conjunction to enable IaC within an IT environment. Ansible and Redfish technologies require no additional licensing.

By automating tasks using Ansible and Redfish with PowerEdge servers, customers can lower the resource and time requirements for managing compute infrastructure of any size. Such automation also increases uptime of applications and overall infrastructure by decreasing potential human errors.

With Ansible and Redfish, deploying new PowerEdge servers or updating existing PowerEdge servers takes minutes, instead of days or weeks. Automating deployment and updates with Ansible and Redfish allows system administrators to focus on tasks that add value to their organization, instead of spending cycles keeping the lights on.

References

Dell Technologies documentation

The following Dell Technologies resources provide additional information related to this document. Access to documents depends on your login credentials. If you do not have access to a document, contact your Dell Technologies representative.

- Dell Technologies DevOps Community Pages
- Dell Technologies Developer Portal
- Dell Technologies Redfish API documentation
- Dell Technologies iDRAC documentation
- Dell Technologies DevOps

Red Hat Ansible documentation

See also the following Red Hat Ansible documentation:

- Ansible User Guide
- Ansible Sample Code
- Ansible Forums on Google Groups
- Blog – Introduction to Ansible builder
- Blog – Using Ansible and tower with shared roles

Other resources

The following book is also useful when learning Ansible:

Ansible for DevOps: Server and configuration management for humans – Jeff Geerling, ISBN 978-0-9863934-3-3