Ansible Solution Brief
22 February 2016

EOS+ Consulting Services

Introduction
Technical Solution
    A History of the Arista + Ansible Solution
    Conceptual Overview
    Connection Methods
        Understanding connection: local
        CLI Connection
        eAPI Connection
    Module Overview
        About the eos_eapi Module
        About the eos_command Module
        About the eos_config Module
        About the eos_template Module
    Creating a Workflow
    Best Practices
        Data Models
        Roles
    Sample Roles
Getting Started
Support from Arista

Introduction

Arista Networks and Ansible have partnered to bring the strength and agility of Ansible to your network. Arista is built on extensibility and continues to leverage third-party integrations like Ansible to help you get the most out of your switch. The following document describes some of the new Ansible modules and provides tips to help build a comprehensive configuration management solution.

Technical Solution

A History of the Arista + Ansible Solution

Arista has worked hard to provide comprehensive modules to the Ansible community for a few years now. This initial effort was packaged under the arista.eos role, which has certainly done its fair share of heavy lifting! However, as of Ansible 2.1, a set of redesigned modules ship with Ansible in an effort to simplify and accelerate this powerful integration. We strongly recommend users implement their configuration management and compliance solutions with these new modules.

Conceptual Overview

The vast majority of network engineers feel comfortable with the look and feel of the CLI, and by extension the running configuration. Therefore, in this new set of modules, a concerted effort has been made to allow your playbooks and templates look and feel like the syntax so many are already familiar with. By leveraging the syntax of the running configuration, Ansible users can quickly determine which bits and pieces of the configuration need to change, and they know exactly how to do it.

Configuration management is just one piece of a complete Ansible solution. Ansible 2.1 provides elegant ways to monitor the health and status of your switch along with the ability to make corrective actions automatically.

Connection Methods

Understanding connection: local

Those familiar with linux server administration using Ansible know that following a successful SSH connection, Ansible copies the module and corresponding variables to the server, where the module is executed. For Arista switches, the model is slightly different. Playbooks will indicate to Ansible that the module should be run locally on the Ansible Control Host using the *connection: local* argument. The Ansible Control Host then reaches out to the Arista switch via eAPI (http/s) or CLI (ssh).

CLI Connection

Ansible can communicate with your Arista switch using an existing EOS user. The module will connect to your switch over SSH much the same way you would do so manually. Depending upon your EOS configuration, the *authorize* feature may need to be used in order to send an *enable* command to change modes.

eAPI Connection

Ansible can also leverage the built-in eAPI to read the running configuration and send various commands. This requires that eAPI is enabled on your switch; running over HTTP or HTTPS. Again, a standard EOS user can be used to authenticate commands that are sent via Ansible.

Whether you connect using CLI or eAPI, very little configuration is required on your switch to get up and running. A simple EOS user and enabling of eAPI is the most that is needed. This follows the Ansible tenet of an agentless architecture, and requires no additional packages be installed on your Arista switch.

Module Overview

About the eos_eapi Module

The eos_eapi module allows rapid configuration of eAPI. This module allows you to enable/disable eAPI's various features using simple key/value arguments.

```
name: Enable eAPI with no HTTP, HTTPS at port 9443, local HTTP at port 80, and socket
enabled
eos_eapi:
  state: started
  http: false
  https_port: 9443
  local_http: yes
  local_http_port: 80
  socket: yes
  provider: {{ provider }}
```

About the eos_command Module

The eos_command module provides a set of features to help run any EOS command and intelligently analyze the output. Some examples help illustrate its power:

Example 1

```
eos_command:
  commands: "{{ lookup('file', 'commands.txt') }}"
```

This task would allow you to read in a file of commands and execute each one independently. The returned object would have the associated output from each command.

Example 2

```
eos_command:
  commands:
    - show version | json
    - show interfaces | json
    - show version
  waitfor:
    - "result[2] contains '4.15.0F'"
    - "result[1].interfaces.Management1.interfaceAddress[0].primaryIp.maskLen eq 24"
    - "result[0].modelName == 'vEOS'"
```

This task utilizes the *waitfor* feature. This is very helpful when some state on the switch is changing and you want to give it some time to complete. By using the *waitfor* argument, the module will continue running the command until the *waitfor* condition is met (up until a configurable timeout). This module provides a ton of functionality and the ability to make intelligent decisions if certain state criteria are not met within a certain period of time.

About the eos_config Module

The eos_config module is the easiest and most straightforward way to modify the configuration on your switch. Simply include commands that you expect to be present in the running-config and the module will reach out to the switch and check to see if the command is absent or present. Here's a simple example:

Example 1

```
eos_config:
  lines: ['hostname spine.01.ny.us']
```

This task would ensure that the switch hostname was set to the specified string. Pretty simple.

Example 2

```
eos_config:
  lines:
    - 10 permit ip 1.1.1.1/32 any log
    - 20 permit ip 2.2.2.2/32 any log
    - 30 permit ip 3.3.3.3/32 any log
    - 40 permit ip 4.4.4.4/32 any log
    - 50 permit ip 5.5.5.5/32 any log
  parents: ['ip access-list test']
  before: ['no ip access-list test']
  match: exact
```

In the second example, we use some more of the eos_config features to intelligently modify and ACL. Here, we tell the eos_config module to look at a specific ACL, *ip access-list test* using the *parents* argument. We also tell the module that the entries must *match: exactly*, otherwise

execute all of the *lines*. Finally, we tell the module to run a negation command before running any configuration commands so that we start with a clean ACL. The eos_config module is great for small configuration changes, but it doesn't allow you to work with larger sets of configuration text. That's where eos_template comes in.

About the eos_template Module

The eos_template module allows you to pass in static running-configuration text or it will execute a jinja template to generate the desired configuration. This becomes extremely powerful since jinja provides a simple logic structure as well as advanced variable substitution.

The basic usage of eos_template would look something like:

Example 1

```
name: Provide a static startup configuration
eos_template:
  src: startup_config.txt
```

where startup_config.txt is a standard startup-config file that the switch typically reads in. In this case, the module will read every line in the text file and compare it to the current configuration on the switch to determine which commands need to be run.

The power of eos_template is in the name, template. The module can be passed a jinja template which it will execute first and then compare the resultant configuration against the running-configuration on the switch.

Example 2

```
name: Configure Ethernet Interfaces
eos_template:
  src: interfaces.j2
with_items: {{ intfs }}
```

Given the above task with the following host_vars and template:

```
# host_vars
intfs:
  - name: Ethernet1
    mtu: 1500
    desc: uplink to spine
  - name: Ethernet2
    mtu: 1000
    desc: peering link to DC
```

```
# interfaces.j2
interface {{ item.name }}
   description {{ item.desc }}
   mtu {{ item.mtu }}
```

This helps illustrate how powerful the templating module is. You can now simply keep a logical, simple, data model in a database or in group/host_vars and then allow Ansible to perform the templating and variable substitution. This makes multivendor environments very easy to manage.

Creating a Workflow

Once you get the hang of these new modules you'll want to fit the solution into a seamless workflow. This can be accomplished using Ansible Tower along with other open source tools.

Consider the following scenario:

All of your playbooks and variables reside in a Git version-controlled repository. As part of the daily adds, removals and updates, an admin wants to add a new vlan to a group of switches. She creates a new branch in the Git repo and adds the new vlan in the appropriate group_vars file. Then the change it committed via git and a pull request is generated to merge her change into the master branch. In your git repo you can create a trigger which launches a Jenkins job to run. This Jenkins job clones your repo and checks out your test branch. It then spins up some vEOS devices and runs the playbook against them to look for any errors. You could even use the eos_command module to run a series of checks to make sure the switch is operating as expected. Once the success is reported, the git branch can be merged into master and the playbook can be confidently run in production.

Best Practices

Data Models

Before you set out to create scores of playbooks that manage your network, it's important to take time to plan. Part of this planning includes the creation of a data model. A well thought out data model will help you keep track of your environment and even allow you to effortlessly manage multiple systems and vendor devices. This data model should be self-describing and easy to read such that anyone can review your group/host_vars and understand what each variable does. Once you have a comprehensive data model, it will be easy to create templates to implement EOS running-configuration commands.

Roles

It's easy to go overboard when you first start using Ansible to create a playbook for each and every system and requirement. This can lead to repeated tasks in dozens of playbooks. This quickly becomes unmanageable! The solution is to use Ansible roles to logically group associated tasks together. For example you can have a bgp role that will configure bgp on any Arista device. The tasks can be structured in a way that only certain ones are run for spine config versus leaf config. By using roles, you can quickly update scores of playbooks by only changing tasks in one place.

Sample Roles

Arista is already creating sample roles for the Ansible community. These roles are hosted at galaxy.ansible.com/arista and showcase how easy it is to use these new modules.

Getting Started

Check out **ansible.com/ansible-arista-networks** to view the latest tutorials and getting started documents.

Support from Arista

Arista Networks proudly provides both best-effort as well as comprehensive professional services around Ansible implementations. If you need a hand, don't hesitate to reach out to the EOS+ Consulting Service group at **ansible-dev@arista.com**.