Engineering. Simplicity

DAY ONE: AUTOMATING JUNOS®

WITH ANSIBLE

Create Ansible *playbooks*

to automate your network tasks. What once took hours will take minutes. This *Day One* book shows you how.

By Gopal Das

DAY ONE: AUTOMATING JUNOS® WITH ANSIBLE

Network automation is expanding rapidly. Many network engineers are looking into automa- tion but they do not have a background in programming. Ansible helps – it minimizes the programming aspects of automation – but getting started building real-world solutions can be confusing. Many other Ansible training resources focus on automating server tasks, not network tasks. It’s time for a *Day One* guide that helps you set up an Ansible environment that can manage hundreds of Junos networking devices and accomplish realistic network management tasks. *Day One: Automating Junos with Ansible* is the newest book on net- work automation for network engineers. It includes a set-up guide, tutorials, and showcase scenarios whose Ansible scripts you can download on GitHub, all while discussing real-world requirements like secure authentication.

*“This is a no-nonsense tutorial that gets you automating for real – really fast. Reading through the material is like sitting down with a tutor who’s eager to share easy-to-follow real-world examples, complete with practical tips handed down from lessons learned through the author’s past experi- ence. Skip the pie-in-the-sky, 30,000-foot, hand-waving you might get from other automation books and get started automating Junos with Ansible today!”*

*Jarrod Shields, Senior Network Engineer, Juniper Networks*

*“This is a true Day One book providing enough background and guidance to bring a beginner into the world of Network Automation. Even advanced users will be impressed with the thorough screen- shots, CLI outputs, and playbook samples used to educate the audience with the tips and tricks to accomplish another level of automating with Ansible and Junos OS.”*

*Jessica Garrison, Network Automation Architect, Juniper Networks*

IT’S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN HOW TO:

* Install Ansible and PyEZ
* Understand Ansible’s file/folder structure
* Create a device inventory with multiple groups
* Create playbooks to execute commands on Junos devices
* Create playbooks to update Junos device configuration, in either “set” or “config” format
* Create templates of device configuration fragments, assemble the fragments, and apply the resulting config to devices
* Create custom Ansible modules for tasks not supported by existing modules

ISBN 978-1-941441-61-9

9

Juniper Networks Books are singularly focused on network productivity and efficiency. Peruse the complete library at [www.juniper.net/books.](http://www.juniper.net/books)

Automating Junos® with Ansible

# Day One: Automating Junos® with Ansible by Gopal Das

Chapter 1: Introduction: Automation and Ansible 9

[Chapter 2: Installing Ansible 16](#_TOC_250003)

Chapter 3: Understanding JSON and YAML 26

Chapter 4: Running a Command – Your First Playbook 35

Chapter 5: Junos, RPC, NETCONF, and XML 62

[Chapter 6: Using SSH Keys 78](#_TOC_250002)

Chapter 7: Generating and Installing Junos Configuration Files 89

Chapter 8: Data Files and Inventory Groups 126

Chapter 9: Backing Up Device Configuration 162

Chapter 10: Gathering and Using Device Facts 190

Chapter 11: Storing Private Variables – Ansible Vault 213

[Chapter 12: Roles 223](#_TOC_250001)

[Chapter 13: Repeating Tasks 239](#_TOC_250000)

Chapter 14: Custom Ansible Modules 270

Appendix: Using Source Control 287



**© 2018 by Juniper Networks, Inc.**

**All rights reserved.** Juniper Networks and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. The Juniper Networks Logo and the Junos logo, are trademarks of Juniper Networks, Inc. All other trademarks, service marks, registered trademarks, or registered service marks are the property of their respective owners. Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

**© 2018 by Juniper Networks Pvt Ltd.**

**All rights reserved for scripts located at** [https://github.com/](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible) [Juniper/junosautomation/tree/master/ansible/Automat-](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible) [ing\_Junos\_with\_Ansible.](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible)

**Script Software License**

**© 2018 Juniper Networks, Inc. All rights reserved. Licensed under the Juniper Networks Script Software License (the “License”). You may not use this script file except in compliance with the License, which is located at** <http://www.juniper.net/support/legal/scriptlicense/>**. Unless required by applicable law or otherwise agreed to in writing by the parties, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,**

**either express or implied.**

**Published by Juniper Networks Books**

Author: Sean Sawtell

Technical Reviewers: Jarrod Shields, Khelil Sator, Diogo Montagner, Victor Gonzalez, Jessica Garrison, Sravya Sukhavasi, Supratik Sharma

Editor in Chief: Patrick Ames Copyeditor: Nancy Koerbel CodeMaster: Amish Anand

ISBN: 978-1-941441-61-9 (print)

Printed in the USA by Vervante Corporation. ISBN: 978-1-941441-62-6 (ebook)

Version History: v1, December 2018 2 3 4 5 6 7 8 9 10

<http://www.juniper.net/dayone>

**About the Author**

**Sean Sawtell** has been with Juniper Networks since 2002, and has been a Network Engineer with Juniper’s internal network team since 2004. Sean’s focus today is on network automation. In 2014 Sean earned a Master of Science degree in Computer Science, and subsequently was an adjunct professor for two years teaching the CS curriculum. Before joining Juniper, Sean taught Microsoft and Novell courses and held MCSE, MCI, CNE, and CNI certifica- tions.

**Author’s Acknowledgments**

This project ended up being bigger than I expected. My sincere gratitude to Patrick for his guidance, and to everyone who provided suggestions and corrections from the first outline through the final edits. This book is better due to your contributions.

Thanks to my parents and family members for their love and support, for encouraging my interest in computers starting in high school (a long time ago in a state far, far away...), and for teaching me the value of hard work and ongoing learning. I don’t say it enough, but I love you all.

I am blessed to work with a wonderful group of people at Juniper Networks--it is hard to imagine a team more willing to share information, teach and support each other. Thanks to my managers for supporting my decision to write this book, and thanks to Juniper for giving me the opportunity to explore new technologies and play with some really amazing toys.

Hat tip to Red Hat and Ansible for creating this fantastic automation platform, and to everyone who has contributed to Juniper’s Galaxy modules and PyEZ. This book could not exist without the frameworks you built.

Finally, thanks to you, the reader, for choosing this book. I hope you enjoy reading it as much as I enjoyed writing it. Best wishes for your automation journey. Don’t panic, and remember your towel.

*Feedback? Comments? Error reports?* Email them to [dayone@juniper.net.](mailto:dayone@juniper.net)

## Welcome to Day One

This book is part of the *Day One* library, produced and published by Juniper Net- works Books.

*Day One* books were conceived to help you get just the information that you need on day one. The series covers Junos OS and Juniper Networks networking essen- tials with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow.

You can obtain publications from either series in multiple formats:

* Download a free PDF edition at <http://www.juniper.net/dayone>.
* Get the ebook edition for iPhones and iPads from the iBooks Store. Search for

*Juniper Networks Books* or the title of this book.

* Get the ebook edition for any device that runs the Kindle app (Android, Kindle, iPad, PC, or Mac) by opening your device’s Kindle app and going to the Amazon Kindle Store. Search for *Juniper Networks Books* or the title of this book.
* Purchase the paper edition at Vervante Corporation ([www.vervante.com](http://www.vervante.com/)) for between $15-$40, depending on page length.
* Note that most mobile devices can also view PDF files.

## Target Audience

This book is written for network administrators and network engineers who are starting to build and use network automation to make their jobs easier, and is fo- cused on how to use the Ansible automation platform to configure and manage Junos-based devices. However, once you’ve learned how to use Ansible, you can also leverage that knowledge to automate the administration of servers or network gear from other vendors.

Many of the examples in this book are drawn from the author’s experience manag- ing Juniper’s own internal network. It considers real-world concerns such as secu- rity, complying with corporate policy, and synchronizing the automation development work between team members.

## This Book’s GitHub Site

Go to: [https://github.com/Juniper/junosautomation/tree/master/ansible/](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible) [Automating\_Junos\_with\_Ansible](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible).

## What You Need to Know Before Reading This Book

* You should be comfortable managing and configuring Junos devices and using the Junos command line.
* You should be comfortable with the terminal or command line of your computer’s operating system. You should have some familiarity with UNIX/ Linux operating systems.
* You should have, or should obtain, a programmer’s text editor1, such as Atom (<https://atom.io/>) or Sublime Text (<http://www.sublimetext.com/>), or an IDE (Integrated Development Environment) such as PyCharm ([https://www. jetbrains.com/pycharm/](https://www.jetbrains.com/pycharm/)).Whatever your chosen editor, see if it has (built-in or via an installable module) JSON and YAML syntax highlighting .
* You should have at least one Junos device with which you can work the examples in this book. Two or more devices of different classes would be better, as some features are configured differently on different classes of device; for example, VLAN configuration is different on MX devices (bridge domains) vs. EX devices. Your test device(s) should be free from any change control processes that your company may require for production equipment.
* You do not need to be a programmer, and you do not need experience with any particular programming language or with Ansible. Ansible is designed so that little programming is needed, and this book explains the required con- cepts as we work through the examples. However, if you have some program- ming experience you may wish to skip the paragraphs which explain things like conditionals and loops; it’s okay, the author won’t be offended.
* The exception to the prior point is Chapter 14, which discusses custom modules. Writing custom modules requires some programming, typically in the Python language, and Chapter 14 assumes you have Python experience. If you do not have Python experience you can skip Chapter 14, but the author encourages you to follow the examples by rote to get a general understanding of the topic even if the Python code seems obscure.

1 A programmer’s text editor, or programmer’s editor, is a text editor with features geared toward writing programs and making the life of a programmer a little easier. A text editor is like a word proces- sor except that it handles only plain text – no boldface, no fonts, no tables, etc. – and saves files con- taining only ASCII or ANSI text with no formatting information.

Syntax highlighting is a feature of programmer’s editors that shows comments, programming language keywords, and other aspects of a program, in different colors. This helps a programmer quickly identify what text is a comment, a string, a reserved word in the programming language, etc. The color coding

is applied by the editor, but is not saved in the file as formatting information (the file is plain text).

## What You Will Learn by Reading This Book

* Some of the categories of automation and why network engineers should embrace automation (beyond "it’s fun!").
* The author has a sense of humor and will occasionally use it.
* How to install Ansible on different operating systems. Some of the differences between Juniper’s Ansible modules and Ansible’s core modules.
* Some occasions when you should check with your company’s Information Security or other teams about your automation work.
* How to create, run, and debug Ansible playbooks.
* How to gather data from Junos devices using Ansible. Examples include gathering device attributes (reboot time, Junos version, hardware model, etc.) and downloading device configuration.
* How to configure Junos devices using Ansible. The book includes several examples, starting small and growing to more complex configurations. Later examples introduce roles and show why roles are a powerful feature of Ansible.
* What YAML and JSON are, and how are they used with Ansible.
* What RPC, NETCONF, and XML are, and how they relate to Junos automa- tion.
* How to use SSH keys for making secure device access easier.
* Why network automation engineers should use source control and a short introduction to Git and GitHub.
* How to create custom Ansible modules for those things that Ansible does not already do, or does not do the way you want it done.

## Book Organization and Approach to Learning

This book takes a hands-on approach to learning. As you read most of the chap- ters, you will work with Ansible or Junos, creating solutions to problems that you may have faced as a network engineer. Chapters 1 and 3 are exceptions; these chapters present important background information and are more theoretical in nature.

You will work through a number of examples based on real-world needs, learning concepts as you go. The objective is to introduce ideas or techniques and immedi- ately use them, so that theory is quickly reinforced by experience. Later examples often build on earlier examples, adding new techniques to accomplish more

complex tasks or otherwise improve our results. This reflects a natural approach to learning, building a knowledge foundation then layering new knowledge onto the foundation.

Because the author is part of a team that manages a production network, and as- sumes you do as well, the examples reflect real-world concerns that are sometimes overlooked in training material. For example, security will be discussed in several contexts, and all over-the-network communication with our devices will be via SSH rather than insecure protocols like FTP or Telnet. The Appendix discusses the importance of using some type of source control system for archiving your work and sharing it with coworkers (a topic of great importance to working program- mers but often overlooked even in university-level computer science curricula) and shows examples with Git and GitHub.

Sometimes this book does something “wrong” to show problems and how to re- solve them. These examples provide an opportunity to show troubleshooting pro- cesses and illustrate why one approach might work better than another.

References to web sites or other supporting material are included at *the end* of each chapter. The discussions in this book are not exhaustive explorations of each topic, but an effort to familiarize you with the most useful tools and techniques for automating your network with Ansible. Should you wish to dig further into any of these topics, the references are good starting points for those explorations.

Finally, keep in mind that this book is not the end of your journey into automa- tion. This book covers a lot of territory, but it is not a comprehensive discussion of Ansible and networking; you may find useful some features of Ansible not covered here. New tools are introduced every year, and new features are added to existing tools. Best practices may change to take advantage of new features or to address changing requirements. Self-guided, ongoing learning is every bit as important in the automation field as in the networking field.

# Chapter 1

Introduction: Automation and Ansible

Let’s start with some background about automation. What is automation? What business needs can be helped with automation? What is Ansible and how does An- sible support a network engineer’s automation needs?

## What is Automation?

According to the online Merriam-Webster dictionary ([https://www.merriam-web-](https://www.merriam-webster.com/dictionary/automation) [ster.com/dictionary/automation](https://www.merriam-webster.com/dictionary/automation); accessed June 22, 2017), *automation* is:

1. the technique of making an apparatus, a process, or a system operate automati- cally;
2. the state of being operated automatically;
3. automatically controlled operation of an apparatus, process, or system by mechanical or electronic devices that take the place of human labor.

Automation is essentially having a computer or machine to do something that a person would otherwise need to do manually. In the context of network automa- tion, this means a computer (which may be the control plane of a network device) is gathering data from—or making changes to—a network device, tasks that a net- work engineer would otherwise need to accomplish.

The type of automation discussed in most of this book involves processes that are initiated by a person but that then execute with little or no further human input.

Automation can go beyond manually-initiated processes to include having the net- work respond automatically to events, for example, taking some action to mitigate problems without human intervention. This book takes you up to that level, and although event-driven automation is outside its scope, the topics discussed herein create a foundation upon which you could build an event-driven environment.

## Why Use Automation?

Automation is faster and more efficient than manual operations. A computer can establish an SSH session to a router faster than a person can type *ssh myrouter* or click the appropriate bookmark in their SSH client. Having established that con- nection, a computer can issue commands and gather the results more quickly than a person typing at a keyboard and reading a screen. (This author still gets a little excited when he launches an automated process and watches it run on dozens of devices in less time than it would take him to finish the first device.)

People make mistakes – we fat-finger commands while typing, forget steps, or do things out-of-order. Automation avoids those problems – once the automation is implemented, it should perform the process the same way each time.

If a person needs to execute the same command on 100 different network devices, he will quickly get bored and lose focus, possibly leading to mistakes. Automation does not get bored or lose focus.

Automation does not require sleep or food; it can be on-the-job 24 hours a day, monitoring the network and possibly responding to network events while people are doing other work or sleeping.

The network engineer who understands how to automate her network will im- prove her job security – her employer will appreciate that, for example, she can make changes more quickly and with greater accuracy than engineers who do things manually.

However, be careful while developing your automation—an automated process that does the wrong thing can quickly do that “wrong thing” on dozens or hun- dreds of devices and potentially cause major problems.

Despite the benefits, there are times when automation may not be appropriate. For example, if you need to make a single change on a single router, it will probably take more time to develop automation than to make the change manually. Auto- mated solutions usually require an initial investment of time and effort to develop and debug the automation. The development investment is repaid by the time sav- ings that result from using the automation on numerous occasions or with large numbers of devices. A one-time change on a single device may not save enough time to warrant the development effort for an automated solution.

## Business Scenarios for Network Automation

Companies are leveraging automation in a variety of ways. The following are some general examples; there are likely many variations on these scenarios, and probably additional scenarios that are not mentioned.

*Gathering data:* Automation can quickly query devices to gather data about them, such as device model or other hardware information, Junos version, interface sta- tus or error counts, routing tables, etc. This data may be used for planning up- grades, troubleshooting problems, or other needs.

*Configuring devices:* Automation can quickly push configuration changes to de- vices. Changes might be minor, like adding the IP address for a new DNS server, or might be a significant change across multiple hierarchies of the Junos configura- tion, like creating a firewall filter and applying it to one or more interfaces.

*Auditing configuration compliance:* Automation can check the configuration of the network devices to ensure that they meet standards (for example, no *public* SNMP community) and can adjust the configuration to bring non-compliant de- vices into compliance.

*NOOB setup:* Automation can make it easier to configure NOOB (new-out-of- box) devices by putting an initial configuration on the device via a console connec- tion, or by configuring a ZTP (zero-touch-provisioning) server with appropriate configuration files and Junos images.

*Responding to problems:* A network device can automatically gather trouble- shooting information in response to an event, such as uploading a r*equest support information* report when the chassis detects a hardware failure. Or a device can automatically disable an interface when the error rate on the interface exceeds a threshold.

## Off-box Versus On-box Automation

There are many ways of building automation, and many places where automation can run. One important difference within the realm of Junos automation is wheth- er the automation runs on the Junos device itself or on a separate system.

*On-box automation* runs on the Junos device itself. Traditionally this automation is implemented by event-options settings in the Junos configuration, or by scripts written in the SLAX programming language and installed on the device. Recent Junos versions are adding support for on-box Python scripts. These on-box scrips can help with configuration compliance and responding to problems.

*Off-box automation* runs from a computer or system other than the Junos device itself. These solutions must communicate with the Junos device, either over the network or via the device’s serial console. Off-box automation can be implement- ed in a variety of languages or with a variety of platforms.

Sometimes on-box and off-box automation work together. For example, consider Juniper’s Service Now management application (part of Junos Space). Service Now relies on Juniper’s AI-Scripts (Advanced Insight Scripts), a collection of SLAX scripts installed on Junos devices that can detect problems and report them

to Service Now. The AI-Scripts are usually installed on the network devices by the Service Now application. It’s an off-box management platform installing on-box scripts that report problems back to the off-box system: ([http://www.juniper.net/](http://www.juniper.net/us/en/products-services/network-management/junos-space-applications/service-now/) [us/en/products-services/network-management/junos-space-applications/](http://www.juniper.net/us/en/products-services/network-management/junos-space-applications/service-now/)

[service-now/](http://www.juniper.net/us/en/products-services/network-management/junos-space-applications/service-now/)).

## What is Ansible?

Ansible is an automation platform, a framework for executing a series of opera- tions that accomplish defined tasks. It is commonly used “to provision, deploy, and manage compute infrastructure across cloud, virtual, and physical environ- ments.” (<https://www.ansible.com/webinars-training/introduction-to-ansible>; ac- cessed June 22, 2017). Ansible was written by Michael DeHaan and initially released in 2012. Ansible was purchased by Red Hat in 2015.

Building automation with Ansible requires little traditional programming knowl- edge – the programming required for many common operations is already done and made available to you in the form of *modules*. At the risk of over-simplifying, you create a *playbook* describing the automation you need by piecing together a series of modules. This process is the focus of most of this book, so rest assured that we will discuss it in some detail.

Ansible includes a large selection of modules with the platform. While Ansible dif- ferentiates between three categories of included modules (core, curated, and com- munity), this book refers to the included modules collectively as *core modules*. ([http://docs.ansible.com/ansible/list\_of\_all\_modules.html)](http://docs.ansible.com/ansible/list_of_all_modules.html)

There are also modules developed by the Ansible user community and made avail- able via Ansible Galaxy (<https://galaxy.ansible.com/)>. This book refers to these as *Galaxy modules*. As you will see when installing Ansible in Chapter 2, you can install Galaxy modules using the ansible-galaxy command. Once installed, these modules are available for use in your automation solutions.

Starting in 2014, Juniper Networks published Galaxy modules that enable Ansible to manage Junos devices (<http://junos-ansible-modules.readthedocs.io/>or [https://](https://galaxy.ansible.com/Juniper/junos/) [galaxy.ansible.com/Juniper/junos/)](https://galaxy.ansible.com/Juniper/junos/). Supported operations include executing com- mands, downloading configuration, making configuration changes, rolling back configuration changes, and upgrading Junos. (One of the maintainers of these modules is Stacy Smith, co-author of the excellent book, *Automating Junos Ad- ministration, O’Reilly Media, 2016*.) Not surprisingly, these modules use Juniper’s suggested techniques for how off-box automation communicates with Junos de- vices. This book discusses the recommended approach in Chapter 5, *Junos, RPC, NETCONF, and XML* but the short version is that the modules use the Junos API (application programming interface). These modules rely on a library called PyEZ, also written by Juniper, for establishing the connection to the Junos device, issuing the appropriate API request, and receiving the results of the API request.

Starting in 2016 with version 2.1, Ansible added core modules that work with Ju- nos devices ([http://docs.ansible.com/ansible/list\_of\_network\_modules.](http://docs.ansible.com/ansible/list_of_network_modules) html#junos). Supported operations are broadly similar to Juniper’s Galaxy mod- ules, but with many differences in details such as module names and how to use the modules (which means playbooks written for one set of modules need to be

re-written to use the other set of modules). In addition, the Ansible 2.3 versions of these modules do not use Juniper’s PyEZ library, which means Ansible had to write their own code for accessing the Junos API and receiving the results. At this time, Juniper recommends their Galaxy modules over Ansible’s core Junos modules.

## Overview of Ansible Terminology

Let’s briefly introduce some of the terms that Ansible uses. These terms will all be discussed in more detail later in the book.

*Playbook*: The file you create that defines your desired automation process by call- ing a series of modules. Ansible executes the playbook, calling the modules that implement the tasks needed to perform the desired automation.

*Module:* A program that accomplishes a specific task, like copying a file, installing software, or rebooting a device.

*Task:* Within a playbook, a task is a call to execute a module that does a specific job, like copy a file or configure a device. Tasks usually include one or more *argu- ments*, data that adds detail to what the module should do, such as the name of the file to copy, or the IP address of the device to configure).

*Play:* Within a playbook, a play is a collection of tasks. A playbook will have one or more plays. If a playbook contains multiple plays, it is likely that the plays have different requirements: for example, they may execute on different hosts.

*Fact:* As a playbook executes, Ansible learns about the hosts involved. The learned data are called facts and may be referenced by name in the playbook.

*Variable:* Data about a host or group declared by the user. Like facts, variables can be referenced by name. The difference is that variables are declared by the user (there are several approaches that the book will discuss) not discovered by Ansible.

*Role:* A way of organizing desired behavior into reusable units. Roles consist of tasks, variables, and other elements that can be incorporated into multiple playbooks.

*Template:* A file containing some static text, such as device configuration com- mands, but with some places where Ansible will “fill-in-the-blank” with data spe- cific to each device, such as the hostname or an IP address. Templates can be used to generate configuration files that contain device-specific settings, or to format and save facts gathered from devices in a human-friendly format. Templates are written using the Jinja2 language.

*Inventory:* The list of devices that Ansible knows about, possibly with some pre- set variables (data) about each device, such as the device’s management IP address. Inventory is typically stored in a single file named inventory, or in a set of files in a directory called inventory.

*Groups:* Within the inventory of devices, you can define groups that describe col- lections of devices you can refer to by name, for example, a group called *routers* would provide an easy way to refer to all routers and exclude firewalls and switches.

*group\_vars and host\_vars:* Directories in which you can place files containing variables (data) about groups or hosts. The files in the group\_vars and host\_vars di- rectories let you define more variables, or variables containing more complex data, than would be practical within the inventory file itself.

MORE? For more terms, or to see Ansible’s definitions of these terms, please refer to the Ansible glossary: <http://docs.ansible.com/ansible/glossary.html>.

## Ansible vs. Ansible Tower vs. AWX

Ansible is a no-cost, open-source automation platform. It is a command-line tool; you work with Ansible in your operating system’s terminal or shell.

Red Hat also offers Ansible Tower, a paid commercial software product that builds on the underlying Ansible automation platform. Tower features a WebUI and adds role-based authentication, integration with Git repositories, paid sup- port, and other features intended to make Ansible more accessible to an IT team. (<https://www.ansible.com/products/tower>)

In late 2017, Red Hat released as open-source the AWX Project, “the upstream project from which the Red Hat Ansible Tower offering is ultimately derived.” [<https://www.ansible.com/products/awx-project/faq>, retrieved Jan. 8, 2018.] Like Tower, AWX offers a WebUI, role-based authentication, etc. Unlike Tower, AWX is available at no cost but Red Hat provides no paid support for it.

This book focuses on the command-line Ansible platform. However, many play- books developed in Ansible can likely migrate to AWX or Tower with few changes should your organization choose to adopt one of them.

## Where Can Ansible Help?

No automation tool will satisfy every automation need. But let’s review the auto- mation scenarios mentioned in the *Business Scenarios for Automation* section of this chapter and see if, and how, Ansbile can help with each scenario.

*Gathering data:* Ansible can collect a pre-defined set of facts about Junos devices. It can also run nearly any Junos operation-mode command and collect the results, which can be saved in files, either in separate files for each device, or all collected in a single file. There are also modules to send information by email, IRC, and oth- er communication/notification technologies.

*Configuring devices:* Ansible’s modules (Galaxy and core) include some specific configuration tasks. More powerful, however, is its ability to use a template to cre- ate any Junos configuration you desire. Ansible “fills in” the template with device- specific values and uploads the resulting configuration to the Junos device.

*Auditing configuration compliance:* Ansible can download a Junos device’s con- figuration, or a specific hierarchy of the configuration, and save it to a file. The author is not aware of a module to parse the saved configuration to check compli- ance, but a Python programmer could write such a module, or you could have An- sible call the shell and run grep to search the saved files. Alternately, Ansible can gather operational data from a device, such as a list of BGP peers, and confirm that list matches some pre-defined expectations. Finally, Ansible can apply standard/ expected configurations to a device as mentioned above, thus ensuring the device is in compliance after the configuration is applied.

*NOOB setup:* As noted above, Ansible can generate and apply device configura- tion; this can include a minimalist configuration intended to, for example, put an IP address on the management interface and set the root password, thus making the device available on the network. Juniper’s Galaxy module can apply this con- figuration via a serial connection. For those who use ZTP, Ansible is not part of the ZTP process itself, but Ansible can generate a dhcpd.conf file for the DHCP server and initial configurations for the devices, and it can copy these files, and a Junos image file, into the necessary locations on the DHCP and file server(s).

*Responding to problems:* Ansible is not designed as an event-driven platform; ar- ranging an Ansible playbook to run automatically in response to external events would require an event framework outside of Ansible itself. However, Ansible can use scp to upload SLAX scripts to a Junos device that enable it to respond to events, and it can make the necessary configuration settings in Junos to run those scripts when the event occurs.

# Chapter 2 Installing Ansible

This chapter discusses the system requirements for using Ansible to manage Junos devices, and how to install Ansible on MacOS and Linux systems.

## System Requirements

The computer running Ansible and executing playbooks is called the *control ma- chine*. The systems being managed by an Ansible control machine are called *man- aged nodes*.

An Ansible control machine that will manage Junos devices requires:

* A non-Windows operating system. MacOS, Linux, and other UNIX-type oper- ating systems work well.
* Python 2.6 or 2.7.
* An SSH client, typically OpenSSH. This is usually installed by default on Linux/ UNIX systems and MacOS.
* Juniper’s Galaxy modules and Juniper’s PyEZ Python library.

The control machine will communicate with the Junos devices using the NET- CONF protocol running over SSH. By default, the NETCONF service on Junos uses TCP port 830. NETCONF is discussed in Chapter 5.

Windows users should consider running a Linux distribution in either a virtual machine (VM) or a Docker container. If you create a new Linux VM or container for this purpose, keep in mind that you do not need a GUI for Ansible; you can use a Linux distribution intended for servers and avoid the significant overhead of a desktop GUI. If you wish to use a Docker container, you might consider this image from Docker Hub: <https://hub.docker.com/r/juniper/pyez-ansible/>.

If you are running a non-Windows system but you wish to separate your automa- tion environment from your host OS, you may wish to use a VM or Docker con- tainer similar to the Windows users.

## Software Versions Used While Writing This Book

The author used the following versions of Ansible and related modules while de- veloping and testing the examples in this book:

* Ansible 2.3.2.0 and 2.4.2.0
* Juniper.junos (Juniper’s Galaxy module) 1.4.2 and 1.4.3
* Pip 9.0.1
* PyEZ (junos-eznc) 2.1.5, 2.1.6, and 2.1.7
* Python 2.7.13

You do not need to use the same versions. All were current versions during the time the author started writing, but some were upgraded before the book was complete, and most have probably been upgraded (again?) before you read this. However, keep in mind that the maintainers of these open-source projects do oc- casionally change module names or arguments or the like. If a playbook is throw- ing errors that do not seem to be a typo or inaccessible device, check to see if the versions of the programs you have installed might have changed something.

NOTE As this book is going through the final editing process, the maintainers of Juniper’s Galaxy modules are preparing version 2.0.0 of those modules. The new version is a major rewrite of these modules and includes significant changes to module names and arguments. Version 2.0.0 is expected to be backwards compat- ible with playbooks written with/for version 1.4.3, so the information in this book should work with the updated modules. If you wish to ensure you are using version 1.4.3 as you work through this book, see the tip at the end of the *Installing Ansible on MacOS* section of this chapter for how to install a specific version.

## Ansible’s Installation Instructions

Ansible’s web site has a page that discusses installing Ansible on a wide variety of systems: <http://docs.ansible.com/ansible/intro_installation.html>. Please look over this page before proceeding. Keep in mind, however, that this page only discusses basic Ansible installations; we also need the PyEZ Python library and the Juniper. junos Ansible Galaxy modules in order to administer Junos devices with Ansible.

The remainder of this chapter discusses in some detail how to install Ansible on MacOS and Linux systems, including several suggestions, particularly for MacOS, that are not discussed on Ansible’s web page.

## Installing Ansible on MacOS

This section discusses installing Ansible on MacOS (or OS X) using the optional Homebrew package manager.

### Command-line Developer Tools

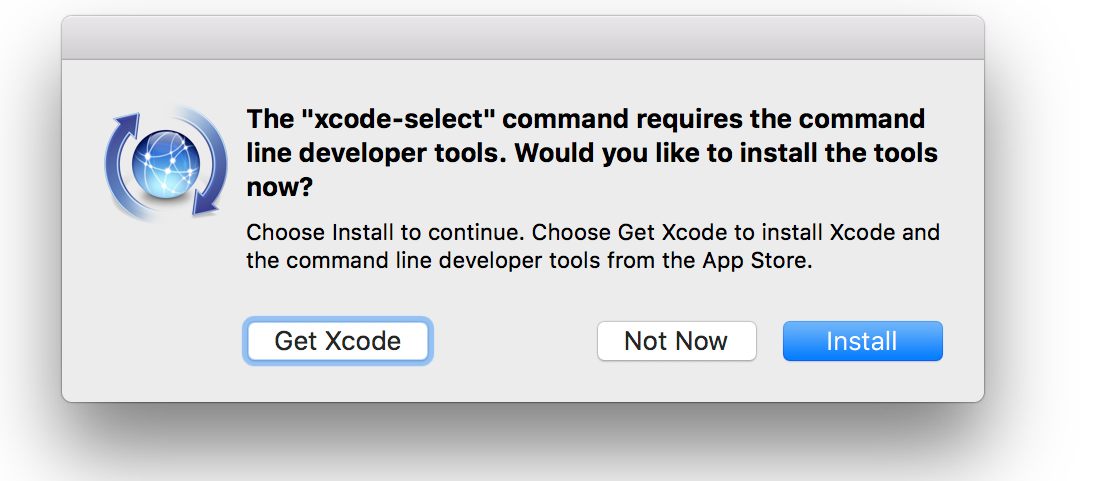
Before you install Ansible on MacOS, you need to install Apple’s command-line developer tools. Some of the software we need to install with Ansible needs to be compiled during installation, and Apple’s command-line developer tools include the necessary compiler and related files.

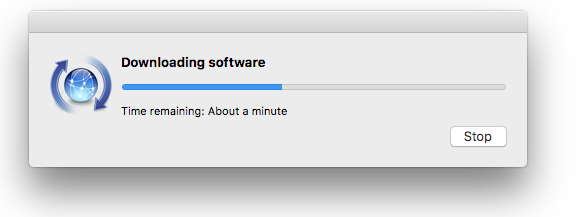
To install the command-line tools (whether or not you have installed the complete XCode environment), open a Terminal window and enter the command xcodese- lect install as shown here:

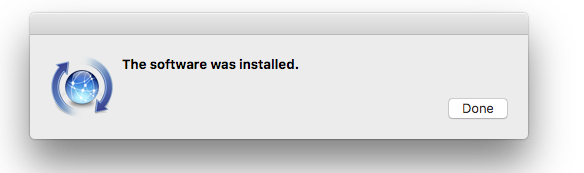
mbp15:~ sean$ **xcode-select --install**

xcode-select: note: install requested for command line developer tools mbp15:~ sean$

MacOS will display a dialog box, similar to the following, to confirm your choice

to install the command-line developer tools. Click *Install* to continue.

Click *Agree* in the *Command Line Tools License Agreement* dialog box that ap- pears next. An installation status dialog box should appear as the software is downloaded and installed. Click *Done* when the installation is complete.



### Homebrew and Python

Recent versions of MacOS include a Python interpreter. While it is possible to in- stall Ansible on MacOS using the system-installed Python interpreter (the process is similar to installing on Linux as shown later in this chapter), the author has found this leads to challenges when updating PyEZ. MacOS includes a Python li- brary that is also used by PyEZ, but MacOS locks the library so it cannot be al- tered. This may not a problem when you first install PyEZ, but when you later attempt to upgrade PyEZ and its dependencies with a pip install --upgrade junos- eznc command, the upgrade will fail because pip will not be able to upgrade the locked library.

One way to avoid this problem is to install the Homebrew ([https://brew.sh/)](https://brew.sh/) pack- age manager1 and use it to install a new Python environment2. The Homebrew- installed Python environment, including the PyEZ library you install in that Python environment, will exist in parallel with the MacOS-installed environment, giving the former a level of independence from the latter. The parallel installation of Python means we will be able to upgrade all libraries without running into problems with the locked system library.

An additional benefit is that the Homebrew-installed Python interpreter will likely be the most recent version, while the MacOS interpreter is probably a little older. For example, on the author’s newly installed MacOS Sierra system, the system- installed Python is version 2.7.10, a few revisions behind the current release.

Take note of the current version and location of the Python interpreter on your system:

mbp15:~ sean$ **python --version**

Python 2.7.10

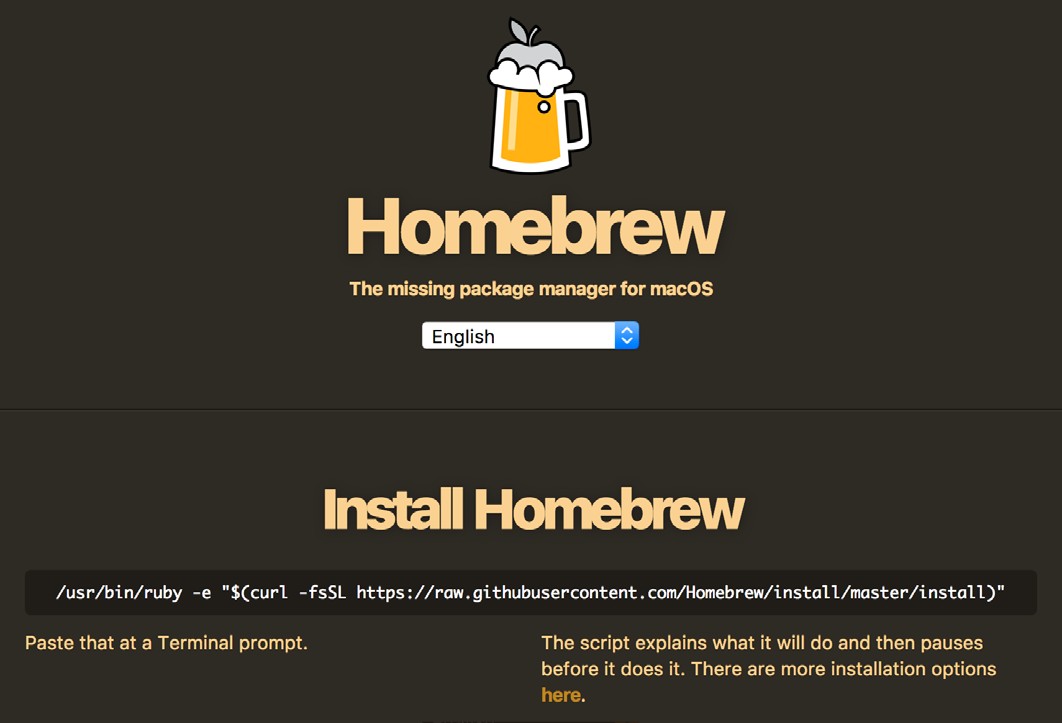
mbp15:~ sean$ **which python**

/usr/bin/python

1. There are other package managers for MacOS, such as MacPorts ([https://www.macports. org/)](https://www.macports.org/), which may accomplish the same goal. The author has not worked with these other package man- agers, but if you already have one of them installed you may wish to see if the package manager you already know has an Ansible package rather than converting to Homebrew.
2. Another option is to create a Python Virtual Environment. There are several tools that can do this, among them virtualenv (https://pypi.python.org/pypi/virtualenv).

To install Homebrew, you need to be logged into your Mac as an Admin user; if your account is a Standard account, use System Preferences to add administrative privileges.

Open Terminal and enter the one-line installation command shown on the Home- brew web page (<https://brew.sh/>):



The following was the command when the author wrote this chapter, but please visit the Homebrew web site to ensure you are using the current installer:

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

Follow the prompts and enter your password when asked. The full output from the script is rather long so the following shows a small subset:

mbp15:~ sean$ **/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/ master/install)"**

==> This script will install:

/usr/local/bin/brew

/usr/local/share/doc/homebrew

...

Press RETURN to continue or any other key to abort **<< press Enter or Return >>**

==> /usr/bin/sudo /bin/chmod u+rwx /usr/local/bin Password: **<< enter your password >>**

==> /usr/bin/sudo /bin/chmod g+rwx /usr/local/bin

...

==> Next steps:

* Run `brew help` to get started
* Further documentation: [http://docs.brew.sh](http://docs.brew.sh/)

Because the Homebrew installer updates your system’s path, it is a good idea to exit the Terminal and re-launch it.

Search for Python *formulas* (Homebrew’s equivalent to packages). As this is being written the python formula is current 2.7.x version of Python, while the python3 for- mula is the current Python 3.x interpreter:

mbp15:~ sean$ brew search python

|  |  |  |  |
| --- | --- | --- | --- |
| app-engine-python | gst-python | micropython | python3 |
| boost-python | ipython | python | wxpython |
| [boost-python@1.59](mailto:boost-python@1.59) | ipython@5 | python-markdown | zpython |

Install the formula for the Python 2.7.x interpreter:

mbp15:~ sean$ **brew install python**

==> Downloading https://homebrew.bintray.com/bottles/python-2.7.13\_1.sierra.bottle.tar.gz Already downloaded: /Users/sean/Library/Caches/Homebrew/python-2.7.13\_1.sierra.bottle.tar.gz

==> Pouring python-2.7.13\_1.sierra.bottle.tar.gz

==> /usr/local/Cellar/python/2.7.13\_1/bin/python2 -s setup.py --no-user- cfg install --force --verbose --s

==> /usr/local/Cellar/python/2.7.13\_1/bin/python2 -s setup.py --no-user- cfg install --force --verbose --s

==> /usr/local/Cellar/python/2.7.13\_1/bin/python2 -s setup.py --no-user- cfg install --force --verbose --s

==> Caveats

This formula installs a python2 executable to /usr/local/bin.

If you wish to have this formula's python executable in your PATH then add the following to ~/.bash\_profile:

export PATH="/usr/local/opt/python/libexec/bin:$PATH"

Pip and setuptools have been installed. To update them pip2 install --upgrade pip setuptools

You can install Python packages with pip2 install <package>

They will install into the site-package directory

/usr/local/lib/python2.7/site-packages

See: https://docs.brew.sh/Homebrew-and-Python.html

==> Summary

/usr/local/Cellar/python/2.7.13\_1: 3,528 files, 48MB

Now check to see if the newly-installed interpreter is the default:

mbp15:~ sean$ **which python**

/usr/local/bin/python

mbp15:~ sean$ **python --version**

Python 2.7.13

If the path and version are unchanged from what you saw prior to installing Homebrew, it may be that Homebrew failed to create a symlink (symbolic link, also called a soft link) named python in your /usr/local/bin/ directory. You can manually create this link if needed. Start by changing to that directory and check- ing to see if there is a symlink called python or python2:

mbp15:~ sean$ **cd /usr/local/bin/**

mbp15:bin sean$ **ls -l python**

ls: python: No such file or directory mbp15:bin sean$ **ls -l python2**

lrwxr-xr-x 1 sean admin 35 Jun 3 21:19 python2 -> ../Cellar/python/2.7.13/bin/python2

Assuming you get results similar to those shown above, then create a new symlink called python to the same target as the python2 symlink:

mbp15:bin sean$ **ln -s ../Cellar/python/2.7.13/bin/python2 python**

Return to your home directory and confirm that the Homebrew-installed Python interpreter is now the default.

Check also that the default pip, the Python package manager, is the Homebrew- installed version in /usr/local/bin/:

mbp15:~ sean$ **which pip**

/usr/local/bin/pip

If which pip returns no results or says something other than /usr/local/bin/pip, you may need to create a symlink for the Homebrew-installed pip also. Follow the above instructions but look for pip2 instead of python2.

### PyEZ, Ansible, and Galaxy Modules

Now that you have Python and pip installed and working, you can proceed with installing Ansible and the other required libraries.

Start by using pip to install PyEZ, which will also install a number of PyEZ’s de- pendencies (some output is omitted for brevity):

mbp15:~ sean$ **pip install junos-eznc**

Collecting junos-eznc

Downloading junos\_eznc-2.1.5-py2.py3-none-any.whl (149kB) 100% |████████████████████████████████| 153kB 408kB/s

Collecting lxml>=3.2.4 (from junos-eznc)

Downloading lxml-3.8.0-cp27-cp27m-macosx\_10\_6\_intel.macosx\_10\_9\_intel.macosx\_10\_9\_x86\_64. macosx\_10\_10\_intel.macosx\_10\_10\_x86\_64.whl (7.8MB)

100% |████████████████████████████████| 7.8MB 131kB/s

...

Successfully installed MarkupSafe-1.0 PyYAML-3.12 asn1crypto-0.22.0 bcrypt-3.1.3 cffi-1.10.0 cryptography-2.0.3 enum34-1.1.6 idna-2.6 ipaddress-1.0.18 jinja2-2.9.6 junos-eznc-2.1.5 lxml-3.8.0 ncclient-0.5.3 netaddr-0.7.19 paramiko-2.2.1 pyasn1-0.3.2 pycparser-2.18 pynacl-1.1.2 pyserial-3.4 scp-0.10.2 six-1.10.0

mbp15:~ sean$

Now use pip to install Ansible (some output is omitted for brevity):

mbp15:~ sean$ **pip install ansible**

Collecting ansible

Downloading ansible-2.3.2.0.tar.gz (4.3MB)

100% |████████████████████████████████| 4.3MB 192kB/s

Requirement already satisfied: jinja2 in /usr/local/lib/python2.7/site-packages (from ansible)

...

Successfully built ansible pycrypto

Installing collected packages: pycrypto, ansible Successfully installed ansible-2.3.2.0 pycrypto-2.6.1 mbp15:~ sean$

TIP By default, pip installs the current released version of a module. To ask pip

to install a specific version of a module, include the version number like this:

module==version

For example: pip install junos-eznc==2.1.5

Finally, use the ansible-galaxy command to install Juniper’s Galaxy modules. Be- cause ansible-galaxy needs to modify some Ansible-related files in the system direc- tory /etc/, you may need to sudo this command:

mbp15:~ sean$ **sudo ansible-galaxy install Juniper.junos**

Password: **<< enter your local password >>**

* downloading role 'junos', owned by Juniper
* downloading role from https://github.com/Juniper/ansible-junos-stdlib/archive/1.4.2.tar.gz
* extracting Juniper.junos to /etc/ansible/roles/Juniper.junos
* Juniper.junos (1.4.2) was installed successfully mbp15:~ sean$

TIP By default, ansible-galaxy installs the current released version of a module. To ask ansible-galaxy to install a specific version of a module, include the version number like this: module,version. For example:

sudo ansible-galaxy install Juniper.junos,1.4.3

## Installing Ansible on Linux

Due to the variety of Linux distributions and their package managers, it is impos- sible to write a single set of step-by-step instructions for installing Ansible on Linux. The general process on most distributions should be similar to that de- scribed below, but exact commands will change depending on the Linux distribu- tion and possibly even the version of that distribution.

The commands that follow were tested with Ubuntu Linux Server versions 14.04 (Trusty Tahr) and 16.04 (Xenial Xerus), and should work with minimal modifica- tion on other Debian-based distributions. Users of Red Hat or other non-Debian Linux flavors should alter these instructions to use the appropriate package man- ager and package names for their distribution or version.

Start by updating the package manager’s data files:

sudo apt-get update

Install SSH (OpenSSH client), software build tools, and related packages:

sudo apt-get install openssh-client build-essential sudo apt-get install libffi-dev libxslt-dev libssl-dev

Install the Python language and the Python package manager (pip):

sudo apt-get install python python-dev python-pip

Install Git (this is optional, but the Appendix uses Git):

sudo apt-get install git

Now use pip to install Ansible and PyEZ (junos-eznc):

sudo pip install ansible junos-eznc

Finally, add Juniper’s Galaxy modules to Ansible:

sudo ansible-galaxy install Juniper.junos

## Quick Ansible Test

Now that you have Ansible installed on your system, let’s run a quick test. This test will display *facts* (information) about your system by running an Ansible mod- ule called setup.

For this test, it is normal to get a few [WARNING] lines at the beginning of the output because we have not yet set up an inventory file. The exact warnings differ be- tween Ansible 2.3.x and 2.4.x.

From your command prompt, run the following command (the output is truncated for space reasons):

mbp15:~ sean$ **ansible localhost -m setup**

[WARNING]: Host file not found: /etc/ansible/hosts

[WARNING]: provided hosts list is empty, only localhost is available localhost | SUCCESS => {

"ansible\_facts": { "ansible\_all\_ipv4\_addresses": [

"192.0.2.1",

"198.51.100.10"

...

],

"ansible\_date\_time": { "date": "2017-08-16",

"day": "16",

"epoch": "1502906921",

"hour": "14",

"iso8601": "2017-08-16T18:08:41Z", "iso8601\_basic": "20170816T140841091761",

"iso8601\_basic\_short": "20170816T140841", "iso8601\_micro": "2017-08-16T18:08:41.091832Z",

"minute": "08",

"month": "08",

"second": "41",

"time": "14:08:41",

"tz": "EDT",

"tz\_offset": "-0400",

...

"weekday": "Wednesday", "weekday\_number": "3",

"weeknumber": "33",

"year": "2017"

},

"module\_setup": true

},

"changed": false

}

The ansible command is one of several provided with Ansible; it lets you run an Ansible module (command) against one or more hosts without creating a play- book. The argument m setup says run the setup module, and the argument localhost says to run the module on the local system.

The output is in JSON format; see Chapter 3 for a discussion of JSON.

## References

Ansible’s installation instructions: <http://docs.ansible.com/ansible/latest/intro_installation.html>

Ansible’s setup module: <http://docs.ansible.com/ansible/latest/setup_module.html>

Juniper’s Galaxy modules on GitHub: <https://github.com/Juniper/ansible-junos-stdlib>

Juniper’s Galaxy modules documentation, version 1.4.3:

<http://junos-ansible-modules.readthedocs.io/en/1.4.3/>

PyEZ on GitHub:

<https://github.com/Juniper/py-junos-eznc>

Juniper TechLibrary Ansible documentation: [https://www.juniper.net/documentation/en\_US/release-independent/junos-ansible/](https://www.juniper.net/documentation/en_US/release-independent/junos-ansible/information-products/pathway-pages/index.html) [information-products/pathway-pages/index.html](https://www.juniper.net/documentation/en_US/release-independent/junos-ansible/information-products/pathway-pages/index.html)

# Chapter 3

Understanding JSON and YAML

This chapter introduces you to the JSON and YAML formats for representing data and data structures. Ansible uses both of these formats – playbooks are YAML data files, and output is often shown in JSON format.

This is a theory chapter. Don’t worry, we’ll keep it short.

## What are JSON and YAML?

JSON and YAML are both standards for storing, transmitting, or displaying com- puter data or data structures using a text-based human-readable format.

*Data structures* are techniques for organizing data, usually within a computer’s memory. There are many different data structures available to computer scientists, but most of them (or at least the most widely used) can be represented as:

* + *List* – a collection of data elements (items)
  + *Dictionary* – a set of key:value data pairs
  + a combination of lists and dictionaries

Both lists and dictionaries are discussed in more detail below.

This book uses the names *list* and *dictionary* for these data structures because they are the names used by Python, the programming language in which Ansible is writ- ten. Other programming languages have different names for equivalent or similar data structures.

JSON and YAML are often called *data serialization languages* because the process of translating a data structure into a format that can be stored or transmitted is called *serialization*.

## Data

Lists

Dictionaries

Before we discuss collections of data (lists and dictionaries), we need to under- stand the types of data that might appear in the collections. There are four basic data types used by JSON and YAML:

Numbers: A *number* is a numeric value: 21, -5, 3.14.

Strings: A *string* is a sequence of characters enclosed in quotation marks: “Hello” or “My puppy’s name is Puddles” or “I think, therefore I am.” These examples use double-quotes ("string") but some environments may use single-quotes ('string').

Boolean: A *boolean* is a true-or-false value, typically represented by the words true or false. Note that true or false are not quoted – "true" is a string while true is a Boolean value.

Null: A *null* or *nil* value is used to represent the absence of any assigned value, and is represented by the word null (not quoted).

A *list*, sometimes called an *array*, is an ordered collection of zero or more *elements* (an entry in the list, datum). *Ordered* in this context means the sequence of the ele- ments within the list is preserved; the order of elements in the list will not change unless you, the user, make it change. (*Ordered* does not mean the list elements are automatically placed in alphabetical order or numerical order.)

Lists are denoted by square brackets ([ ]) – in other words, all the elements of a single list must be contained within a pair of square brackets. Elements within the list are separated by commas. For example:

[9, 2, 7, 32, 5]

["Sean", "Jackie", "Bridget", "James"]

Accessing a single element of a list is often done by *index*, or position; for example, the third name in the list of names above is "Bridget". However, computer scientists like to number things from 0. As a result, the element at index 0 of the name list above is "Sean", and element 3 of that list is "James".

Lists can contain a mix of data types, including other lists or dictionaries. A list within a list is sometimes called a *nested list*:

["Hello", 5, true, ["nested", "list"], null]

A *dictionary*, also called an *associative array*, is a collection, like a list, but the ele- ments are key:value data pairs. The *value* is the data we need to store, and the *key* is a label, a way to identify and locate the associated value.

Dictionaries are denoted by curly braces ({ }), with each key and associated value joined by a colon (:), and key:value pairs separated by commas. For example:

{"name": "France", "capital": "Paris", "population": 67000000}

The elements of a dictionary are accessed by key, not by index; in the dictionary above, the *name* value is *France*.

Keys are normally strings.

Keys should be unique within the dictionary. If you have the dictionary {"things": 5, "widgets": 10, "things": 15} and you asked for the value associated with the key *things*, would you get 5 or 15? Many programming languages enforce unique keys in their implementation of lists.

Dictionaries are *unordered* -- the key:value pairs are not guaranteed to be in any particular sequence. This is not a concern when accessing values by key. However, when you print or display a complete dictionary, the key:value pairs are likely to be displayed in an order other than the order in which they were added to the dictionary.

Values can be any of the data types, including lists or dictionaries. For example, a set of daily low- and high-temperature data could be represented using a diction- ary, where the keys are the names of the week days and the values are two-element arrays with the low and high temperatures:

{"Monday": [67, 90], "Tuesday": [70, 91], "Wednesday": [65, 83]}

However, that data might be easier to understand if we replace each list of tem- peratures with a dictionary; the keys in the nested dictionary can help describe the numbers:

{"Monday": {"low": 67, "high": 90}, "Tuesday": {"low": 70, "high": 91}, "Wednesday":

{"low": 65, "high": 83}}

Notice that the above data has several occurrences of the keys *low* and *high*. At first glance, one might think we have duplicate keys, but this is not the case. Each day (key) has its own dictionary (value) containing temperature data, and the *low* and *high* keys are unique within each day’s dictionary.

## JSON

JSON (JavaScript Object Notation) is based on a subset of the JavaScript pro- gramming language commonly used in web browsers, and was originally designed to provide a way of exchanging data between browser and web server. Today, JSON is supported by many programming languages and is used for a wide range of data serialization tasks.

A file containing JSON-formatted data will typically have “.json” as the file’s extension.

Everything discussed above applies to JSON-formatted data, with a few caveats:

* + Strings must be denoted with double-quotes (" ").
  + Lists are called *arrays*.
  + Dictionaries are called *objects*. A dictionary’s keys must be strings.
  + Keys are not required to be unique under the JSON specification. However, the dictionary or equivalent data structure in many programming languages en- forces unique keys, so reading JSON data which contains duplicate keys may result in missing data or errors.

JSON was created to make it easy for computers to serialize and transfer data. It is plain text and thus human-readable, but how human-readable can vary by how the data formatted. The following is valid JSON but, unless you are a computer, good luck figuring out the nested lists and dictionaries:

{"test1":{"sum":215,"avg":71,"values":[62,74,79]},"test3":{"sum":142,"avg":47,"values":[2,46,94]},"t est2":{"sum":259,"avg":86,"values":[94,73,92]}}

The good news is that JSON ignores *whitespace* characters (space, tab, newline or linefeed, carriage return), which means you can add whitespace as needed to im- prove the human-readability of the JSON data without damaging the computer- readability of that data. This is the same data as shown above, but formatted for human consumption:

{

"test1": {

"sum": 215,

"avg": 71,

"values": [ 62,

74,

79

]

},

"test3": {

"sum": 142,

"avg": 47,

"values": [ 2,

46,

94

]

},

"test2": {

"sum": 259,

"avg": 86,

"values": [ 94,

73,

92

]

}

}

Should you encounter some poorly formatted JSON and wish to reformat it, check if your programmer’s editor has an option to do so. You can also use your Python interpreter and the json.tool module. For example, assuming that file info.json contains the JSON data you want to “prettify,” use this command:

python -m json.tool < info.json

## YAML

test1:

avg: 71

sum: 215 values:

YAML (today, short for *YAML Ain’t Markup Language* but originally was *Yet Another Markup Language*) is a data serialization language intended to be easy for humans to read and modify. It is a bit harder to learn YAML than JSON (and much harder to explain), but once learned it is easier to work with YAML-format- ted data.

A file containing YAML-formatted data will typically have either “.yaml” or “.yml” as the file’s extension.

YAML data files should have three hyphens ("---") on the first line of the file. (The reason is outside the scope of this book, but is related to subdividing the data into multiple “documents” within the file). The examples in this section will not show the "---", but you will see "---" at the start of Ansible playbooks and other YAML files through much of the rest of the book.

YAML is a superset of JSON, meaning that valid JSON is valid YAML. Within YAML data you may see brackets ("[ ]") denoting lists and braces ("{ }") denoting dictionaries. However, one of the human-readability benefits of YAML over JSON comes from using an alternative representation for lists and dictionaries, a repre- sentation based in part on the layout of the text, which we will discuss below.

YAML dictionaries require unique keys; converting key:value data with duplicate keys to YAML may result in errors or missing data. Keys may be strings or other data types, but this book will use only strings for keys.

YAML does not require quotes around most strings, though both single- and dou- ble-quotes are supported. However, a string that starts with a character that might be mistaken for JSON formatting, such as a left-bracket ("[") or left-brace ("{"), must be quoted, because brackets and braces normally represent the start of a list or dictionary, respectively.

Because the layout of text of YAML data is important, an example may help set the stage. We will discuss the details in a few paragraphs; this example is just to provide an overview. The last example of JSON data from the previous section can be represented in YAML as follows:

- 62

- 74

- 79

test2:

avg: 86

sum: 259 values:

- 94

- 73

- 92

test3:

avg: 47

sum: 142 values:

- 2

- 46

- 94

* element
* hello
* world

- 10

- 0

- - 1

- 2

- - 3

- 4

- - 5

- 6

Notice how the YAML data has less punctuation (no quotes, no braces, no brack- ets, and no commas) than the JSON equivalent. The addition of leading hyphens ("-") to indicate list entries is intuitive as it looks a bit like a bulleted list.

Indentation in a YAML document is important. Unlike JSON, where indentation is optional and used only for human readability, indentation in YAML data helps to show what elements belong together in a single collection (list or dictionary). A change of indentation level indicates a change of collection. Take a look at the *test1* key above and how its value, another dictionary, is indented. When we get to the key *test2* the indentation level moves back out, indicating we are returning to the original dictionary of which *test1* is an element.

YAML indentation should be done with spaces, never with tab characters. The number of spaces for each indentation level is not defined, but the author prefers two spaces for each level of indentation – two spaces provides enough visual dis- tinction between levels but avoids excessive indentation with deeply nested data structures.

Each element in a list starts with a hyphen and a space ( - ), like this: The list ['hello', 'world', 10] in YAML is:

A change of indentation means a different list. For example, to represent the nested lists [0, [1, 2], [3, 4, [5, 6]], 7] in YAML:

- 7

Notice how elements of nested lists are indented more than the elements of the parent list. The double hyphen on some lines shows that we are starting a new list that is an element of the parent list.

Dictionary entries have no special notation other than the colon and space be- tween the key and the value:

title: Automating Junos with Ansible author: Sean Sawtell

A series of key:value pairs at the same indentation level will be part of the same

dictionary, but a change of indentation means a different dictionary. For example,

{'k1': 'v1', 'k2': {'k2a': 'v2a', 'k2b': 'v2b'}, 'k3': 'v3'} becomes:

k1: v1 k2:

k2a: v2a k2b: v2b

k3: v3

To create a list with an element which is a dictionary, include the leading hyphen only on the first key:value pair that will be part of the nested dictionary. For ex- ample, this list with nested dictionaries…

['e0', {'k1a': 'v1a', 'k1b': 'v1b'}, {'k2b': 'v2b', 'k2a': 'v2a'}, 'e3', {'k4a': 'v4a'}, 'e5']

…is represented by the following YAML:

* e0
* k1a: v1a k1b: v1b
* k2a: v2a k2b: v2b
* e3
* k4a: v4a
* e5

- k1: v1

- k2: v2

k1: v1 k2:

* v2a

Notice how k1a has a leading hyphen indicating it is a new array element, but k1b does not have a hyphen making it part of the same dictionary as k1a, and thus part of the same element of the parent list.

A common mistake when defining a dictionary within a list is to put a hyphen in front of each key:value pair in the dictionary. This YAML…

…creates a list with two elements, each consisting of a single-entry dictionary:

[{'k1': 'v1'}, {'k2': 'v2'}]

To create a dictionary that contains a list as one of the values, put the nested list’s elements on separate lines, each starting with a hyphen, after the line containing the key. For example, the YAML for the dictionary {'k1': 'v1', 'k2': ['v2a','v2b','v2c'], 'k3': 'v3'} is:

* v2b
* v2c k3: v3

k1: v1 k2:

* v2a
* v2b
* v2c k3: v3

The author’s personal preference is to indent the list entries one extra level (two extra spaces) because, to his eyes, having the hyphen of the list entries at the same visual indentation level as the associated key does not provide quite enough visual separation from the surrounding key:value pairs. The extra indentation does not affect the meaning of the YAML provided the indentation is consistent across all elements of the list:

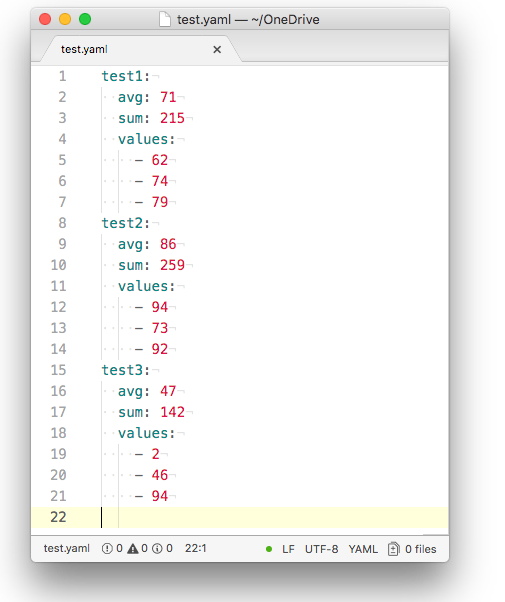
## Text Editor Tips

As you can see, correct indentation is critical for YAML data. Your text editor can help with this!

Most programmer’s editors provide the following settings, though the names of the settings vary between editors:

* + Use spaces instead of tab characters when you hit the tab key. This means you can still use the tab key for quickly indenting lines, but the file will contain YAML-friendly spaces not tab characters.
  + The “tab size” or number of spaces represented by a tab; in other words, each press of the tab key indents this many spaces.
  + Display non-printing characters, including spaces. When the editor displays a symbol for each space, it is easy to see how many spaces a line is indented.
  + Display vertical lines at each indentation level. This helps ensure not only that different lines are indented the same amount, but that the indentation amount is a multiple of the defined tab size, e.g. that a line is not indented five spaces when it should be indented either four or six spaces. (This feature is not as com- mon as the others, and may not be in your chosen editor.)

The following screen capture shows the Atom editor, configured for a two-space tab size and with the above display settings enabled, showing some YAML data from this chapter.



## References

JSON home page and specification: <http://json.org/> YAML home page and specification: <http://yaml.org/>

# Chapter 4

Running a Command – Your First Playbook

This chapter begins to explore Ansible, playbooks, and related files. In it, you will write a short playbook that will execute a command on Junos devices and display the command’s results. You will learn about the structure and content of a play- book, how to prompt for input, how to display output, one way to send a com- mand to a Junos device, and a little about debugging playbooks.

The author is using a Virtual SRX and an EX-2200-C for the examples in this book. You can use any Junos device you have available, preferably a lab or test de- vice. The examples in this book use only two Ansible-managed devices in order to keep the example output short, but if you have more devices you can run the play- book against all of them if you wish.

## The (manual) Command

Assume you need to find out the uptime for your network devices. You might need this information because you want to see if any devices have rebooted since the last scheduled maintenance window.

The Junos CLI command for this is “show system uptime” and, when run manu- ally, it will look something like this (the exact output will differ based on device hardware and uptime):

sean@vsrx1> **show system uptime** Current time: 2017-07-26 20:54:18 UTC Time Source: LOCAL CLOCK

System booted: 2017-07-26 19:19:26 UTC (01:34:52 ago)

Protocols started: 2017-07-26 19:19:27 UTC (01:34:51 ago)

Last configured: 2017-07-26 19:29:26 UTC (01:24:52 ago) by sean

8:54PM up 1:35, 2 users, load averages: 0.16, 0.12, 0.04

The remainder of this chapter shows you how to create an Ansible playbook to run this command across several devices and report the results.

## Playbook Directory and Files

You should create a subdirectory to hold your Ansible playbooks and related files, and you should change to that directory before running a playbook contained there. This chapter, and the next few chapters, will assume you are using subdirec- tory aja in your home directory:

$ **pwd**

/Users/sean

$ **mkdir aja**

$ **cd aja**

$ **pwd**

/Users/sean/aja

In order to run a basic Ansible playbook you need three files: the Ansible configu- ration file, ansible.cfg; the inventory file, which we name inventory, that contains the list of devices that Ansible might access or manage; and the playbook file con- taining the Ansible playbook in YAML format. For this example we name the playbook uptime.yaml.

In future chapters, we will build on this set of files, but these three will suffice for the example in this chapter.

### File: ansible.cfg

Start with the Ansible configuration file. There are many configuration settings that can be placed in this file, but two settings will suffice for now. Create file an- sible.cfg in your ~/aja directory and enter following lines in the file:

[defaults]

inventory = inventory host\_key\_checking = False

The line inventory = inventory tells Ansible to look in the file inventory (in the cur- rent directory) for the list of devices which Ansible will manage.

The line host\_key\_checking = False tells Ansible that it should not use SSH host key checking1. Host key checking is desirable from a security perspective, but can be a problem with automated connections. Disabling Ansible’s host key checking al- lows Ansible to connect even if the server’s ID is not in the known\_hosts file (for ex- ample, if you have not previously manually connected to that device and cached its

1 When connecting manually with SSH, the OpenSSH client confirms that the server’s ID matches the ID cached in the user’s ~/.ssh/known\_hosts file. If there is no entry in known\_hosts then SSH will ask the user to confirm that the server’s ID is valid and that the connection should proceed. If the cached ID is dif- ferent from the ID provided by the server, the client displays an error and aborts the connection.

ID) or does not match the cached value in known\_hosts (as can happen, for example, after a routing engine failover). Ansible 2.4 enables host key checking by default, but it was disabled by default in earlier versions; if you are using an earlier version of Ansible, you may be able to omit this setting.

### File: inventory

Ansible needs to have an *inventory*, a list of devices it should work with. There are a few ways of arranging an inventory, but the easiest is to create a single text file.

Inventory data must include a name for each managed device, which will be avail- able to the playbook in a variable called inventory\_hostname. (The author likes to use the device’s hostname for the inventory name, but that is not a requirement.) Inventory can define groups of devices, a topic we will explore in Chapter 8.

Ansible’s default is to use the file /etc/ansible/hosts for inventory data. The author prefers to have the inventory in the directory with the playbook(s) that use it. This keeps all related files together, makes it easier to have different inventory for differ- ent playbooks, and makes it easier to keep the inventory in source control with the playbooks (we will discuss source control in a later chapter).

Create a file called inventory in your ~/aja directory and add a single line for each test device (your names may be different from what is shown here, and you may use fully qualified names if needed, such as bilbo.mycompany.com):

vsrx1 bilbo

Inventory can also include variables, which define additional data about the de- vice. For example, if your playbook needs to know the role of a device in the net- work (is an EX or QFX acting only as a Layer 2 switch, or does it have Layer 3 interfaces and routing features enabled?) you can define a variable to hold that in- formation, such as device\_role=router. Though defining variables in the inventory file is supported, and we will do so for some of our early playbooks, it is not rec- ommended – it can be difficult to manage as the number of devices and variables increases. We will explore a more scalable approach in Chapter 8.

Two variables that are often useful, and which have special meaning to Ansible, are ansible\_host and inventory\_hostname.

The inventory\_hostname variable contains the name of the host as specified in the inventory file. This is often useful within playbooks; for example, if a playbook saves a file related to a host, you may use inventory\_hostname in the filename so it is clear to which host the file relates.

The inventory\_hostname variable is also often used to specify the device to be man- aged by the playbook, but this only works correctly if name resolution works on that name. In other words, Ansible needs to be able to resolve the author’s device

names bilbo and vsrx1 (from the inventory above) into their respective IP addresses in order to establish the SSH sessions to those devices. If you cannot rely on name resolution, as might be the case with new devices not yet added to DNS, or when setting up a new office which does not yet have connectivity back to corporate, you need an alternative.

The ansible\_host variable is how the author prefers to specify the managed device in playbooks (we will see this shortly). Ansible automatically populates ansible\_ host with the inventory name, but instead of relying on that fact we will populate ansible\_host with the IP address of the target host.

(Ansible versions prior to 2.0 used the name ansible\_ssh\_host for the same variable. If you are using an older Ansible version you should use the longer name.)

An inventory file with variables looks something like this:

vsrx1 ansible\_host=192.0.2.10

bilbo ansible\_host=198.51.100.5 device\_role=l2\_switch

Please update your inventory file to include an ansible\_host variable and appropri- ate IP address for at least one of your test devices.

### File: uptime.yaml

Our first playbook is called uptime.yaml and will, when completed, gather and dis- play the device uptime from our network devices. We will build the playbook in a couple of steps, explaining as we go.

*Playbooks* are Ansible’s “scripts,” describing a series of tasks that will be per- formed on or by various hosts or devices. Playbooks contain *plays*; plays contain *tasks*; tasks call Ansible *modules* to carry out operations.

*Play*: Playbooks consist of one or more *plays*. Each play defines a set of hosts or devices on which the play will run, and one or more *tasks* to be performed on each of those hosts. Plays may also declare variables or include other features needed for the tasks in the play. If a playbook contains multiple plays then the tasks within the different plays probably have different requirements, such as a different set of hosts or devices.

*Tasks*: A *task* is a specific command to be executed. Tasks specify the Ansible *mod- ule* (the command) to execute. Tasks usually include *arguments* that provide ad- ditional details about how the module should run, such as the network device to control, or the username and password for connecting to the device.

## Path to the Python Interpreter

In Chapter 2, the author suggested that MacOS users install Homebrew and install

Python and Ansible with the Homebrew environment. There is a downside to this approach: it changes the path to the Python interpreter and any user-installed Py- thon libraries, including Ansible and PyEZ.

Check to see where the active Python interpreter is located. From your system shell, enter the command which python:

mbp15:aja sean$ **which python**

/usr/local/bin/python

On most UNIX-type systems, the default Python interpreter is /usr/bin/python. An- sible assumes this will be the case and relies on that interpreter being present. If the active Python interpreter is different, Ansible may be unable to find user-installed Python libraries.

The author is using Homebrew, and you can see above that his Python interpreter is /usr/local/bin/python, not /usr/bin/python. The playbook in the next section will fail on the author’s system unless Ansible is told where to find the active Python interpreter.

If your Ansible environment contains a variable called ansible\_python\_interpreter, Ansible will read from that variable the path to the Python interpreter instead of using the default. There are a number of places where this variable could be set; one option is to put the variable setting in the inventory file.

If your which python command returned a path *other than* /usr/bin/python, append a new section to the end of your file inventory with the following (use the correct path for your system, as it may be different than the author’s system):

[all:vars] ansible\_python\_interpreter=/usr/local/bin/python

The [all:vars] line introduces a new section in the inventory file containing vari- ables that apply to all hosts. The next line sets the ansible\_python\_interpreter vari- able to correct path for your system (copy whatever which python returned).

## Uptime Version 1.0

Create file uptime.yaml in your ~/aja directory and enter the following:

---

* name: Get device uptime hosts:
* all connection: local gather\_facts: no

tasks:

* debug: var=inventory\_hostname
* debug:

var: ansible\_host

Remember this is a YAML file and thus indentation is important. The following screen capture shows the same playbook with a dot (.) representing each space, so you can easily see the amount of indentation for each line. The screen capture also shows line numbers to make it a bit easier to discuss the playbook’s contents (do *not* enter the line numbers in your file), and “¬” for line endings (newline characters).

Lets talk about this playbook and what each line does. Reference the line numbers shown in the screen capture.

Line 1: YAML documents start with ---.

Line 2: The name: line identifies the first play in the playbook. The name is not nor- mally significant to Ansible, but it helps document what is happening both to the engineer reading the playbook itself, and during execution (we will see the text “Get device uptime” in the output when we run the playbook).

The leading hyphen ("–") means this line (and all subsequent lines until another leading hyphen with the same indentation) is an element in a list, in this case the list of plays within the playbook. This simple playbook has only one play; there is no subsequent line with a leading hyphen with the same indentation, which in this case is no indentation (the hypen is on the left margin).

Lines 3-4: Declares the hosts or devices against which the playbook will run. The keyword all here is a default Ansible group that automatically includes all devices in inventory. This is an array, so you can specify multiple devices from inventory; for example, your playbook could say:

hosts:

* + vsrx1
  + bilbo

Because hosts: is indented at the same level as name: on the previous line, it is part of the same dictionary, which is defining the first play.

Line 5: Ansible was originally built to work with servers and assumes that each managed server can execute Python scripts; the host running Ansible would

convert a play into a Python script, upload the script to the managed server, tell the server to run the script, and accept the results from the server. This approach will not work with network devices.

To manage network devices, we need Ansible to run everything *locally* (on the host running the playbook). The line connection: local tells Ansible that it cannot up- load a Python script to the managed device; instead, it needs to run the module locally (even though the module may itself connect to the device in order to con- trol it in some fashion).

Line 6: When managing servers, Ansible normally gathers facts—such as operat- ing system, version, IP addresses, and more—from each server. This does not work the same way with network devices because the tasks are running locally (per line 5), which means any facts gathered would be for the host running Ansible, not for the network device. The line gather\_facts: no overrides that default behavior; it tells Ansible to not spend time gathering facts we do not need. (Later in the book we will have examples where fact gathering is useful.)

Lines 7, 10: Blank lines are ignored by Ansible but help humans see “sections” within the playbook.

Line 8: The tasks: line introduces a list of one or more tasks to be executed. De- spite the blank line above, this is part of the same play (the same dictionary) as lines 2, 3, 5, and 6, because the indentation is the same and there has been no (non-blank) line between with less indentation.

Line 9: The first task (note the leading – indicating this is a list element). This task calls Ansible module *debug*, which prints information to screen during playbook execution. The argument var=inventory\_hostname tells *debug* it should print the con- tents of the inventory\_hostname variable.

Lines 11-12: Another task calling debug, but showing another way to provide ar- guments to a module. This task asks debug to print the contents of variable ansible\_ host. Note that the argument var: ansible\_host is indented.

Let’s run the playbook and see what happens. The author’s inventory file contains the following lines. Your hostnames and IP addresses may be different, and re- member that the ansible\_python\_interpreter variable is needed only if your system’s Python interpreter is in a non-standard location:

vsrx1 ansible\_host=192.0.2.10 bilbo

[all:vars] ansible\_python\_interpreter=/usr/local/bin/python

The command ansible-playbook tells Ansible to execute the playbook whose name is provided on the command line. Be sure you are in your ~/aja directory (where

your playbook and inventory files are located) then run the playbook:

mbp15:aja sean$ **pwd**

/Users/sean/aja

mbp15:aja sean$ **ansible-playbook uptime.yaml**

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"inventory\_hostname": "vsrx1"

}

ok: [bilbo] => { "inventory\_hostname": "bilbo"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"ansible\_host": "192.0.2.10"

}

ok: [bilbo] => { "ansible\_host": "bilbo"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

vsrx1 : ok=2 changed=0 unreachable=0 failed=0

Let’s discuss the output from the playbook:

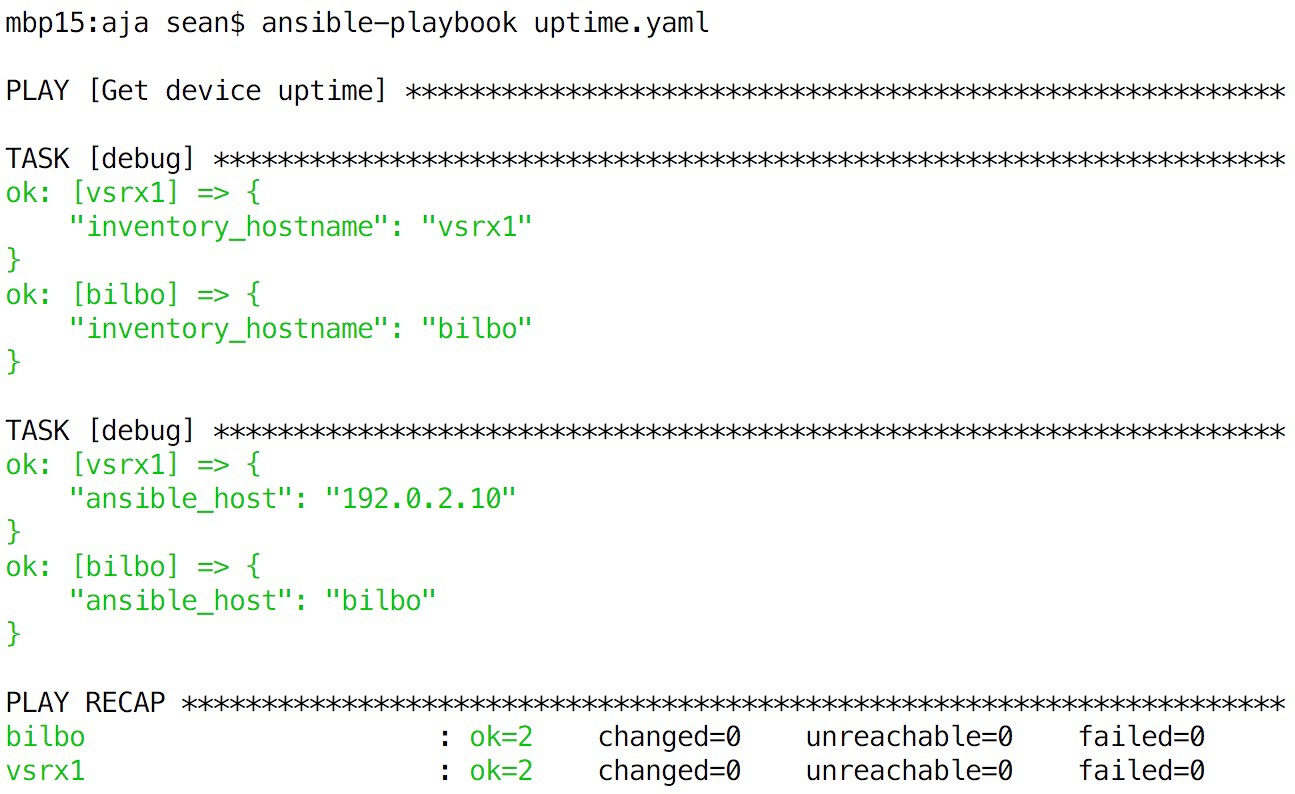
PLAY [Get device uptime] indicates the playbook is starting the play called “Get de- vice uptime.” Note that “Get device uptime” is the value of the name: entry in the play in the playbook; names are usually optional to Ansible but are helpful to humans!

TASK [debug] indicates the playbook is starting a task. Our playbook did not pro- vide names for the tasks, so Ansible displays the module name debug instead.

ok: [vsrx1] and ok: [bilbo] indicate that the task completed successfully for each device. Because the task is debug, which prints information to screen, the output also includes JSON-formatted data showing the value of the requested varible.

The PLAY RECAP section shows a summary of the playbook: for each device, how many tasks completed “ok” (successfully without changing anything), completed “changed” (changed something), or did not complete because the target was un- reachable or there was another failure.

If your terminal shows color, some of the output should have been in green, similar to the following screenshot. Green is good. Tasks that return an ok status will dis- play in green, and in the Play Recap section, devices for which all tasks returned ok will be in green.



As you look at the output, you can see that Ansible runs each task on each device specified in the play. Typically, one task must finish for all devices before Ansible will start the next task, and one play must finish before Ansible will start the next play.

The first TASK [debug] displayed the contents of the inventory\_hostname variable for each device, which is simply the name for the device given in the inventory file.

Each device has a separate set of variables, and different devices will have variables of the same name but containing different data.

The second TASK [debug] displayed the ansible\_host variable for each device. This output is interesting because vsrx1 has an IP address, while bilbo has a hostname. This difference is because of the author’s inventory file, which contains:

vsrx1 ansible\_host=192.0.2.10 bilbo

The inventory line for device vsrx1 assigns a value, an IP address, to the ansible\_ host variable for the device, and we get that IP address in the playbook’s output. Device bilbo does not provide a value for ansible\_host, so Ansible sets it to the same value as inventory\_hostname automatically.

If your debug output includes "VARIABLE IS NOT DEFINED!" instead of a value, check the spelling of the appropriate variable name. The following output illustrates the result of misspelling the ansible\_host variable in the playbook (second task):

mbp15:aja sean$ **ansible-playbook uptime.yaml**

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"inventory\_hostname": "vsrx1"

}

ok: [bilbo] => {

"inventory\_hostname": "bilbo"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"ansible\_hst": "VARIABLE IS NOT DEFINED!"

}

ok: [bilbo] => {

"ansible\_hst": "VARIABLE IS NOT DEFINED!"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

vsrx1 : ok=2 changed=0 unreachable=0 failed=0

## Uptime Version 1.1

Our uptime.yaml playbook runs and displays output, but does not yet communicate with our device to gather any data. Let's fix that!

We will use the Ansible core module junos\_command to communicate with our de- vices and execute the “show system uptime” command. This module needs several arguments: the command to execute, the device to communicate with, and creden- tials for authenticating with the device.

Because we need to authenticate with the devices, our playbook must have a user- name and password. It is poor practice to code those into the playbook; instead, our playbook will prompt for input (ask the user to provide that data).

Modify uptime.yaml so it looks like the following:

---

* name: Get device uptime hosts:
* all connection: local gather\_facts: no

vars\_prompt:

* name: username

prompt: Junos Username private: no

* name: password

prompt: Junos Password private: yes

tasks:

* name: get uptime using ansible core module junos\_command:

commands:

* + show system uptime provider:

host: "{{ ansible\_host }}" port: 22

username: "{{ username }}"

password: "{{ password }}"



Again, let’s discuss the playbook’s contents using line numbers...

Line 8: The vars\_prompt: line introduces a list, each element of which is a diction- ary, that tells Ansible to prompt the user for input and assign that input to specific variables. These variables are associated with the play, not a device, and are avail- able to all devices in the play. Variable names should start with a letter and can contain letters, numerals, and the underscore ("\_") character. Valid variable names include my\_data and Results1; invalid variable names include 2days (starts with a nu- meral) and task-results (contains a hyphen). Variable names are case sensitive: test1 and Test1 are different variables.

Lines 9-11: The first dictionary in the vars\_prompt list. Line 9 tells Ansible to put the user’s input in a variable named username. Line 10 tells Ansible to display “Ju- nos Username” as the prompt for input. Line 11 says the input is not private (the user will be able to see what they type).

Lines 13-15: The second dictionary in vars\_prompt list. This time the input is for a password and is private, meaning Ansible will not display what the user is typing.

Lines 18-26: Defines a task named “get uptime using ansible core module” that calls the Ansible module junos\_command. This task passes two arguments to junos\_ command: commands, which is a list of Junos commands to execute (our playbook has only one element in the list), and provider, which is a dictionary that describes how

to access the target device.

host in the provider dictionary is the device on which Ansible should execute the commands; this is assigned the value of the device’s ansible\_host variable.

* + - port in the provider dictionary is the TCP port which Ansible should use for the SSH connection.
    - username and password in the provider dictionary are the credentials for accessing the device; these are assigned the values provided by the user in the vars\_prompt portion of the playbook.

As you can see in lines 23, 25, and 26, Ansible uses {{ variable\_name }} to say “put the value of variable variable\_name here.” However, YAML considers { } to be a dic- tionary, which would result in an error because {{ variable\_name }} is not a valid dictionary. To make YAML happy, we need to put quotes around the {{ }} so YAML sees it as a string.

Let's run the playbook!

mbp15:aja sean$ ansible-playbook uptime.yaml Junos Username: sean

Junos Password: <enter the device password>

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=0 unreachable=0 failed=0

vsrx1 : ok=1 changed=0 unreachable=0 failed=0

Ansible says it worked...but where are the uptimes?

## Uptime Version 1.2

We saw previously that we can use the debug module to display the contents of a variable, but what variable contains the devices’ uptimes?

As the playbook is constructed right now, the uptime values are lost. We need to assign the results of the junos\_command module to a variable, which we can do by adding register: uptime to that play (uptime is the name of the variable in which the output will be stored):

* name: get uptime using ansible core module junos\_command:

commands:

- show system uptime provider:

host: "{{ ansible\_host }}" port: 22

username: "{{ username }}"

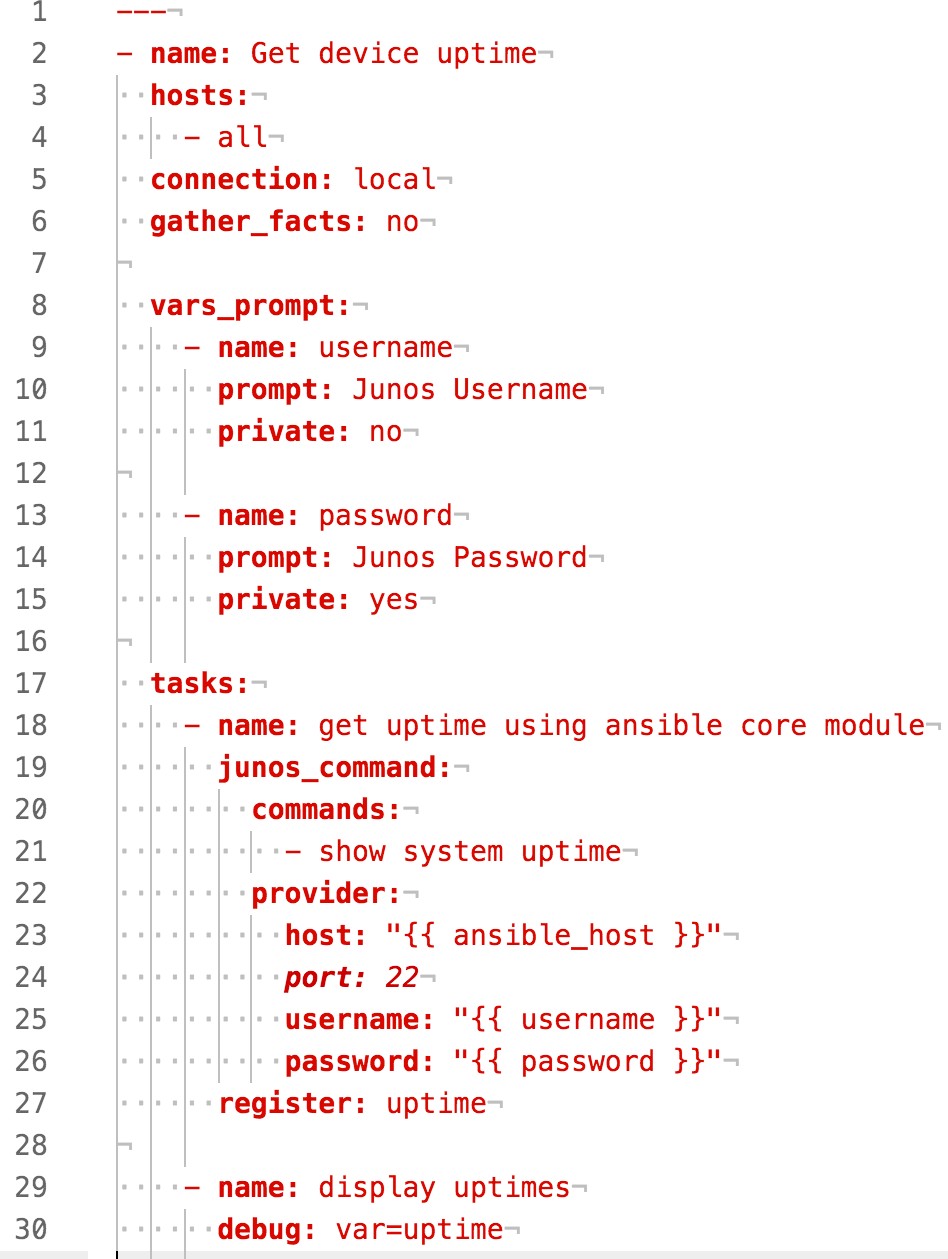
password: "{{ password }}" register: uptime

We then need to add a call to the debug module to display the contents of the new

uptime variable. This time let’s give that task a name:

* name: display uptimes debug: var=uptime

The complete modified playbook (lines 27 – 30 were added):



Let’s run the playbook:

mbp15:aja sean$ **ansible-playbook uptime.yaml**

Junos Username: **sean**

Junos Password: **<enter the device password>**

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=0 unreachable=0 failed=0

vsrx1 : ok=1 changed=0 unreachable=0 failed=0

mbp15:aja sean$ **ansible-playbook uptime-1.2.yaml**

Junos Username: sean Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [vsrx1] => {

"uptime": {

"changed": false, "stdout": [

"Current time: 2017-07-27 08:10:43 UTC\nTime Source: LOCAL CLOCK \nSystem booted: 2017- 07-27 00:54:01 UTC (07:16:42 ago)\nProtocols started: 2017-07-27 00:54:02 UTC (07:16:41 ago)\

nLast configured: 2017-07-

27 00:59:07 UTC (07:11:36 ago) by sean\n 8:10AM up 7:17, 1 user, load averages: 0.17, 0.04, 0.01"

],

"stdout\_lines": [ [

"Current time: 2017-07-27 08:10:43 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2017-07-27 00:54:01 UTC (07:16:42 ago)",

"Protocols started: 2017-07-27 00:54:02 UTC (07:16:41 ago)",

"Last configured: 2017-07-27 00:59:07 UTC (07:11:36 ago) by sean",

" 8:10AM up 7:17, 1 user, load averages: 0.17, 0.04, 0.01"

]

]

}

}

ok: [bilbo] => {

"uptime": {

"changed": false, "stdout": [

"fpc0:\n \

nCurrent time: 2010-01-01 07:53:05 UTC\nTime Source: LOCAL CLOCK \nSystem booted: 2010-01- 01 00:10:04 UTC (07:43:01 ago)\nProtocols started: 2010-01-01 00:16:47 UTC (07:36:18 ago)\

nLast configured: 2010-01-

01 00:15:01 UTC (07:38:04 ago) by root\n 7:53AM up 7:43, 0 users, load averages: 0.14, 0.09, 0.04"

],

"stdout\_lines": [ [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-01 07:53:05 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:10:04 UTC (07:43:01 ago)",

"Protocols started: 2010-01-01 00:16:47 UTC (07:36:18 ago)",

"Last configured: 2010-01-01 00:15:01 UTC (07:38:04 ago) by root",

" 7:53AM up 7:43, 0 users, load averages: 0.14, 0.09, 0.04"

]

]

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

vsrx1 : ok=2 changed=0 unreachable=0 failed=0

Notice that the debug task now has a name: TASK [display uptimes].

Notice that the output is JSON format, and that the uptime variable contains a dic- tionary with a number of values, including stdout (the complete Junos command output as a single string) and stdout\_lines (a list where each element is one line of Junos output).

## Uptime Version 1.3

We do not need the Junos command’s output twice. Can we have debug display just the stdout\_lines part of the uptime dictionary? Yes, we can, by referencing that element of the uptime dictionary.

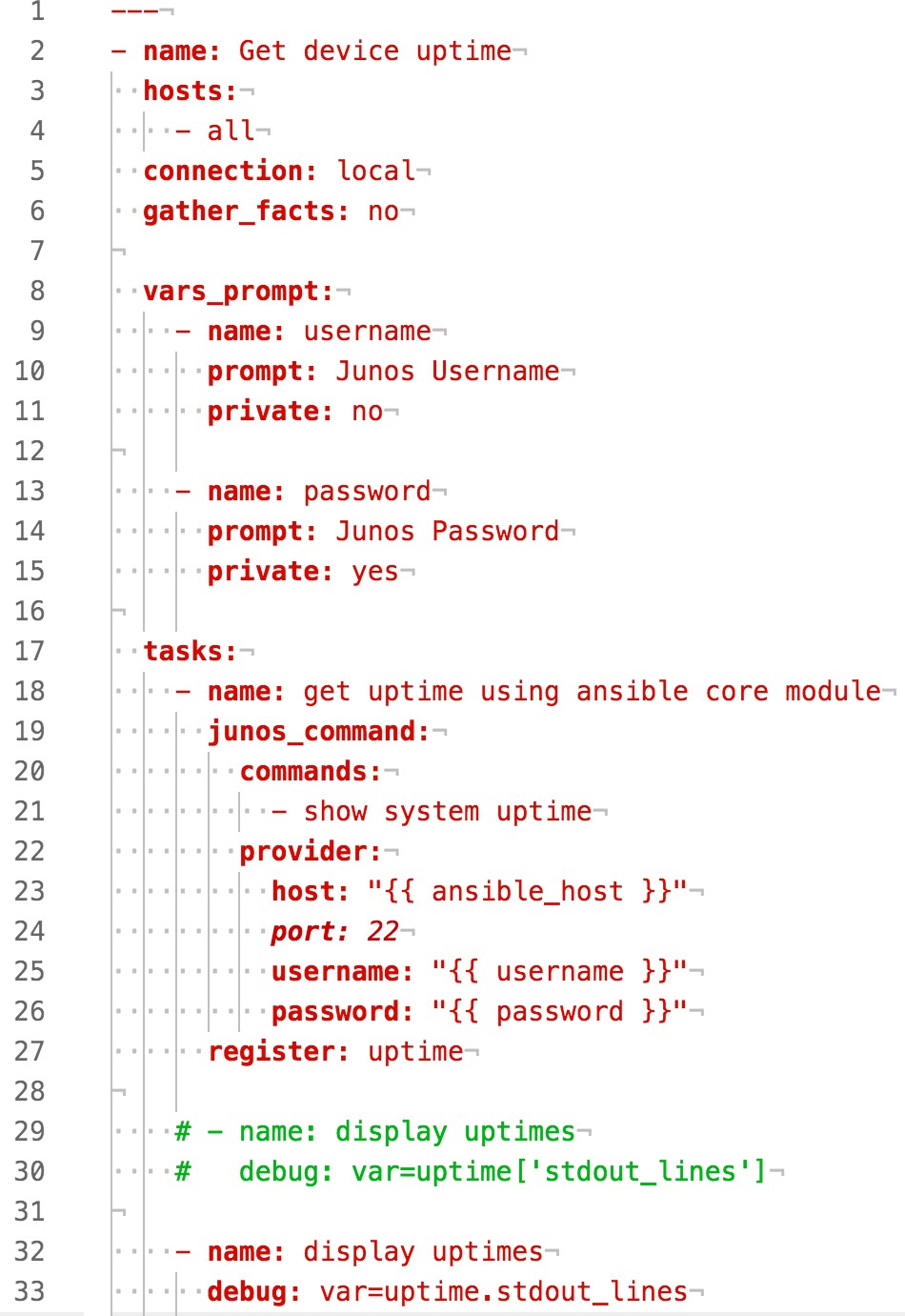
The standard approach to reference a specific dictionary entry, very like what a Python programmer would do, is to put the key for the desired dictionary entry in square brackets after the variable name:

debug: var=uptime['stdout\_lines']

Ansible supports a shortcut, however: use a period to join the variable name and the key for the desired dictionary entry:

debug: var=uptime.stdout\_lines

The modified playbook is shown below. Both approaches discussed above are shown (see lines 30 and 33) but the first approach is commented out. Any line whose first non-space character is a hash or pound symbol ( # ) is a comment and is ignored by Ansible:



Run the playbook again. This time the output for the “display updates” task should look something like the following; notice how much shorter this is, while still providing the information we wanted:

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"uptime.stdout\_lines": [ [

"Current time: 2017-07-27 08:13:19 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2017-07-27 00:54:01 UTC (07:19:18 ago)",

"Protocols started: 2017-07-27 00:54:02 UTC (07:19:17 ago)",

"Last configured: 2017-07-27 00:59:07 UTC (07:14:12 ago) by sean",

" 8:13AM up 7:19, 1 user, load averages: 0.01, 0.02, 0.00"

]

]

}

ok: [bilbo] => { "uptime.stdout\_lines": [

[

"fpc0:",

"---------------------------------------------------------------------", "Current time: 2010-01-01 07:55:41 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:10:04 UTC (07:45:37 ago)",

"Protocols started: 2010-01-01 00:16:47 UTC (07:38:54 ago)",

"Last configured: 2010-01-01 00:15:01 UTC (07:40:40 ago) by root",

" 7:55AM up 7:46, 0 users, load averages: 0.18, 0.10, 0.05"

]

]

}

Uncomment lines 29 and 30 (delete the leading # and the space after it), and com- ment out lines 32 and 33 (add a leading #). Run the playbook again. The output should be essentially the same.

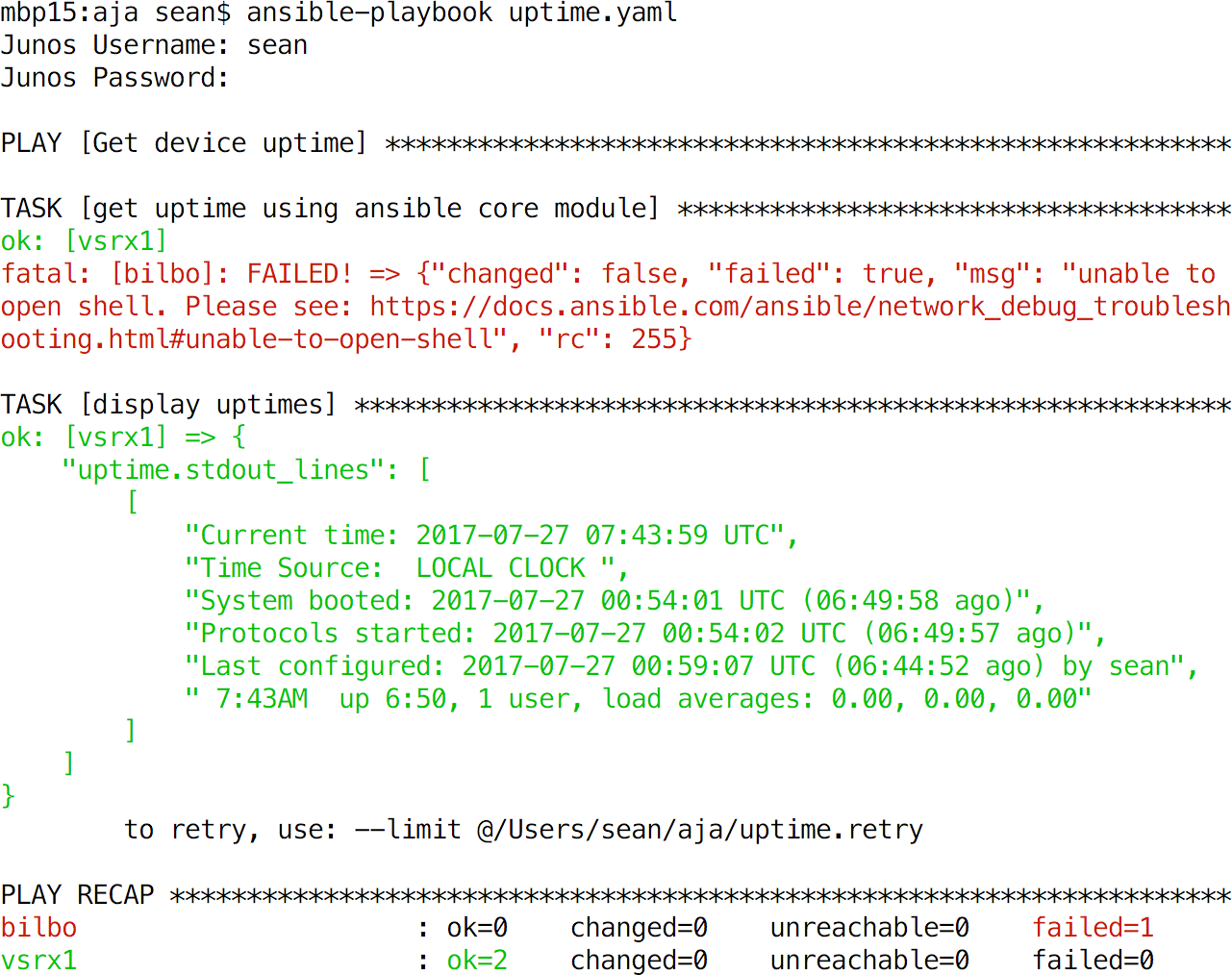
## Errors During Playbook Execution

What happens when problems occur during playbook execution? For purposes of this section we are focusing on problems external to the playbook, such as un- reachable devices or authentication errors, not syntax or other errors within the playbook.

Ansible tracks errors separately for each device. When an error occurs related to a particular device, Ansible stops processing that device; subsequent tasks will not execute for it. However, if other devices have *not* had errors, tasks for those de- vices may be executed. (It is possible, and occasionally useful, to have Ansible ig- nore errors and continue processing a device despite errors, by adding the argument ignore\_errors: yes to the task.)

### Unreachable Device

Unplug the network cable from one of your test devices – for this example, the switch *bilbo* was disconnected – then run the playbook again. The output should look something like the following screen capture.



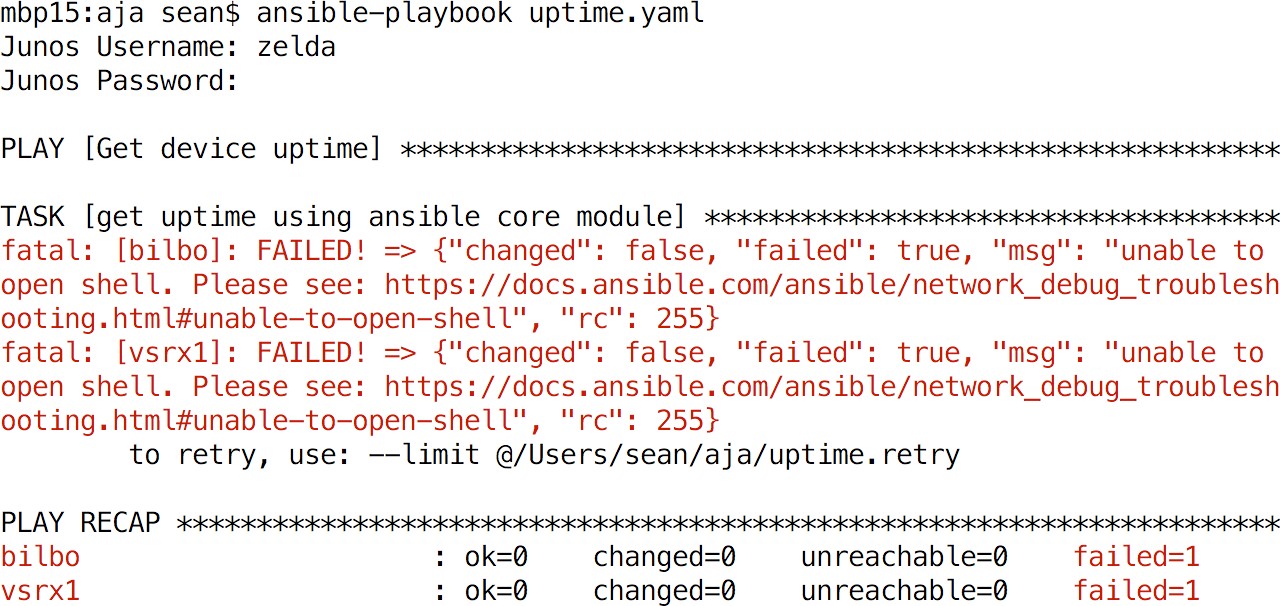
The results for TASK [get uptime using ansible core module] show that *vsrx1* succeed- ed – ok: [vsrx1] – but *bilbo* failed – fatal: [bilbo] followed by an error message. In addition, color terminals display fatal task results in red, and also show red for that device in the PLAY RECAP section of output.

The Ansible junos\_command module produces the generic error message “unable to open shell” seen above for a variety of connection problems. In this example, we know the problem is that the device is unreachable, but the link in the error mes- sage provides information about enabling detailed logging, if needed.

The results for TASK [display uptimes] contains results for only *vsrx1*. Because *bilbo* had an error in the previous task, Ansible stopped processing that device and thus has no results for *bilbo* for subsequent tasks. You can see this in the PLAY RECAP sec- tion – *bilbo* has only one (failed) task, while *vsrx1* has two (ok) tasks.

### Authentication error

Re-connect your network device and give it a moment to restore communication, then run the playbook again. This time, enter invalid credentials at the username and password prompts:



Note that the results for TASK [get uptime using ansible core module] show both de- vices failed: fatal: [vsrx1] and fatal: [bilbo], each followed by the same generic message seen above, though in this case we know the problem is authentication. Notice that TASK [display uptimes] never executed (it does not appear in the out- put). Because there were no devices without errors after the first task, there were no devices against which to execute the second task.

## Limiting Devices

It is often desirable to run a playbook against a subset of the devices in inventory. For example, your inventory for your production network may contain hundreds of devices across dozens of physical locations, but you want to run the playbook against only the Boston devices.

One approach to doing this is to edit the hosts: list in the playbook itself, replacing the default group all with one or more devices:

* name: Get device uptime hosts:
* vsrx1
* newdevice

...

The problem with this approach is that it requires regularly updating the play- book, and doing so in a way that may not be obvious to someone else who might use the playbook and expect it to work on all, or a different subset of, your devic- es. Also, if a playbook contains multiple plays affecting the devices, you would need to make a similar update in each play.

A better approach is to leave the playbook alone, with hosts: set to – all, and use the --limit command line argument to tell Ansible to run against limited set of devices:

mbp15:aja sean$ **ansible-playbook uptime.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"uptime.stdout\_lines": [ [

"Current time: 2017-07-27 09:51:16 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2017-07-27 00:54:01 UTC (08:57:15 ago)",

"Protocols started: 2017-07-27 00:54:02 UTC (08:57:14 ago)",

"Last configured: 2017-07-27 00:59:07 UTC (08:52:09 ago) by sean",

" 9:51AM up 8:57, 1 user, load averages: 0.00, 0.01, 0.00"

]

]

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=2 changed=0 unreachable=0 failed=0

Notice how Ansible ran against only *vsrx1*, not against *bilbo*, even though both devices are in inventory.

The --limit argument can accept multiple inventory names, and can accept wild- cards. In Chapter 8, we will discuss inventory groups, and --limit can accept group names also. Two command-line examples:

ansible-playbook uptime.yaml --limit=vsrx1,newdevice ansible-playbook uptime.yaml --limit='v\*'

When using --limit, it is sometimes helpful to verify which devices Ansible will

manage before running the playbook (and possibly missing devices or managing some you did not expect). You can do this by adding the --list-hosts argument; this causes Ansible to display which devices it will manage, but not actually run the playbook. For example:

mbp15:aja sean$ **ansible-playbook uptime.yaml --limit='v\*' --list-hosts**

playbook: uptime.yaml

play #1 (all): Get device uptime TAGS: [] pattern: [u'all']

hosts (1): vsrx1

## Repeating a Playbook for Devices with Errors

When a playbook encounters an error for a device during a task, it records that device in a “retry” file, a file whose name matches the playbook but with the ex- tension .retry instead of .yaml. By default, “retry” files are stored in the playbook directory.

Earlier in this chapter we forced some errors using the uptime.yaml playbook, so you should have an uptime.retry file:

mbp15:aja sean$ **ls \*.retry**

uptime.retry

Disconnect one or more of your devices – the author disconnected *bilbo* – and re- run the uptime.yaml playbook. Display the contents of uptime.retry:

mbp15:aja sean$ **ansible-playbook uptime.yaml**

Junos Username: **sean**

Junos Password: **<enter password>**

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

fatal: [bilbo]: FAILED! => {"changed": false, "failed": true, "msg": "unable to open shell. Please see: https://docs.ansible.com/ansible/network\_debug\_troubleshooting.html#unable-to-open-shell", "rc": 255}

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"uptime.stdout\_lines": [ [

"Current time: 2017-07-27 08:33:20 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2017-07-27 05:26:26 UTC (03:06:54 ago)",

"Protocols started: 2017-07-27 05:26:26 UTC (03:06:54 ago)",

"Last configured: 2017-07-27 08:33:08 UTC (00:00:12 ago) by sean",

" 8:33AM up 3:07, 1 user, load averages: 0.04, 0.05, 0.05"

]

]

}

to retry, use: --limit @/Users/sean/aja/uptime.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=0 changed=0 unreachable=0 failed=1

vsrx1 : ok=2 changed=0 unreachable=0 failed=0

mbp15:aja sean$ **cat uptime.retry**

Bilbo

Observe that the uptime.retry file lists the inventory\_hostname for the device, bilbo, which recorded a fatal result for a task.

Re-connect your test device(s).

How can we use the “retry” file? We can re-run the playbook for only the failed devices. To do this we use the --limit option and reference the “retry” file, prefix- ing the filename with an “at sign” ( @ ), like this:

mbp15:aja sean$ **ansible-playbook uptime.yaml** [**--limit=@uptime.retry**](mailto:--limit%3D@uptime.retry)

Junos Username: **sean**

Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"uptime.stdout\_lines": [ [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2016-02-07 10:27:16 UTC",

"System booted: 2016-01-20 07:10:48 UTC (2w4d 03:16 ago)",

"Protocols started: 2016-01-20 07:14:01 UTC (2w4d 03:13 ago)",

"Last configured: 2016-02-07 07:45:52 UTC (02:41:24 ago) by sean",

"10:27AM up 18 days, 3:16, 0 users, load averages: 0.17, 0.07, 0.01"

]

]

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

Observe that only the device(s) listed in the “retry” file is (are) processed.

Ansible provides a reminder about this capability in the playbook output – look back at the output with the failure on *bilbo* and note the line, just before the PLAY RECAP section, that says "to retry, use: --limit @/Users/sean/aja/uptime.retry."

With only one failure out two test devices, it would be easy to just use

--limit=bilbo for the repeat run. Consider, however, what it would be like with 100 devices, a dozen of which fail and need to be re-tried. Referencing a single .retry file is much faster and less error-prone than manually finding and “--limiting” the failed devices in a long list of results.

## Debugging Playbooks

Debugging a playbook is part skill and part art. This section provides a few tips that can help, but practice and experience are the best teachers. Google or Bing are often a big help also.

### Syntax and semantic errors

A *syntax error* is when the “grammar” of the playbook is incorrect; for example, a colon (:) is missing or the indentation of a line is incorrect. A *semantic error* is when the syntax is valid but something still does not make sense; for example, the playbook tries to read the value of a variable that has not yet been assigned a value.

Usually, syntax errors will be detected and reported and the playbook will abrupt- ly terminate. Semantic errors may or may not be detected and reported; sometimes the playbook will complete but the results will not be what you expected.

Let’s introduce a couple of errors into the playbook. Introduce each of the follow- ing errors one at a time, and reverse each before proceeding to the next one. The line numbers refer to the screen capture of uptime version 1.3 from earlier in this chapter.

Missing hosts

Delete the hosts: dictionary, lines 3 and 4, from the playbook. When you run the playbook, you should get something like this:

mbp15:aja sean$ ansible-playbook uptime.yaml Junos Username: sean

Junos Password:

ERROR! the field 'hosts' is required but was not set

Incorrect indentation

Remove two spaces from the beginning of lines 13-15, the password prompt. The vars\_prompt section of the playbook should look like this:

vars\_prompt:

* name: username

prompt: Junos Username private: no

- name: password

prompt: Junos Password private: yes

When you run the playbook, you should get something like this:

mbp15:aja sean$ ansible-playbook uptime.yaml ERROR! Syntax Error while loading YAML.

The error appears to have been in '/Users/sean/aja/uptime.yaml': line 13, column 3, but maybe elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

- name: password

^ here

Unmatched (missing) quotation mark

Delete the quotation mark from the end of line 23; the revised line should be:

host: "{{ ansible\_host }}

When you run the playbook, you should get something like this:

mbp15:aja sean$ **ansible-playbook uptime.yaml**

ERROR! Syntax Error while loading YAML.

The error appears to have been in '/Users/sean/aja/uptime.yaml': line 25, column 22, but may be elsewhere in the file depending on the exact syntax problem. The offending line appears to be:

port: 22

username: "{{ username }}"

^ here

We could be wrong, but this one looks like it might be an issue with missing quotes. Always quote template expression brackets when they start a value. For instance:

with\_items:

- {{ foo }}

Should be written as: with\_items:

- "{{ foo }}"

This error message is almost right: the problem is missing quotes, but the error message identifies line 25 as the likely problem, not line 23. Ansible cannot always identify the exact location of a syntax error, even if it correctly identifies the nature of the error.

In the author’s experience, Ansible’s error messages are usually pretty good. The exact wording of error messages may change in different versions of Ansible, so what you see when you perform these examples might be a bit different from what is shown above.

### Verbose mode

Ansible offers a “verbose mode” when running a playbook that provides more information about what is happening. To enable verbose mode, add the com- mand-line argument –v to the ansible-playbook command, like this:

mbp15:aja sean$ ansible-playbook uptime.yaml --limit=bilbo -v

*Using /Users/sean/aja/ansible.cfg as config file*

Junos Username: sean Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] *=> {"changed": false, "stdout":*

*["fpc0:\n \nCurrent time:*

*2010-01-03 02:38:54 UTC\nTime Source: LOCAL CLOCK \nSystem booted: 2010 01-01 00:10:04 UTC (2d 02:28*

*ago)\nProtocols started: 2010-01-01 00:16:47 UTC (2d 02:22 ago)\nLast configured: 2010 01-01 00:15:01*

*UTC (2d 02:23 ago) by root\n 2:38AM up 2 days, 2:29, 0 users, load averages: 0.03, 0.06, 0.02"], "stdout\_lines": [["fpc0:", "------------------------------------------------------------------------*

*--", "Current time: 2010-01-03 02:38:54 UTC", "Time Source: LOCAL CLOCK ", "System booted: 2010-01-01 00:10:04 UTC (2d 02:28 ago)", "Protocols started: 2010-01-01 00:16:47 UTC (2d 02:22 ago)", "Last*

*configured: 2010-01-01 00:15:01 UTC (2d 02:23 ago) by root", " 2:38AM up 2 days, 2:29, 0 users, load*

*averages: 0.03, 0.06, 0.02"]]}*

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"uptime.stdout\_lines": [ [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-03 02:38:54 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:10:04 UTC (2d 02:28 ago)",

"Protocols started: 2010-01-01 00:16:47 UTC (2d 02:22 ago)",

"Last configured: 2010-01-01 00:15:01 UTC (2d 02:23 ago) by root",

" 2:38AM up 2 days, 2:29, 0 users, load averages: 0.03, 0.06, 0.02"

]

]

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

The emphasized text above (*emphasis* added by the author) highlights the addi- tional details provided by –v: the name of the config file and the data returned by the junos\_command module.

You can get still more detail by using –vv or –vvv; each “v” adds a little more “ver- bosity” to the playbook output.

### Verbosity argument to debug module

Recall that our original version of the uptime.yaml playbook used the debug module to display the values of the ansible\_host and inventory\_hostname variables. We re- moved those calls to debug because we really did not need them, but it might be nice to have the value ansible\_host displayed during any future troubleshooting because we pass that value to the junos\_command module. However, if we add the debug calls back in the way we had them originally, the playbook would *always* dis- play the variable’s contents, even when we were *not* troubleshooting, which means most of the time we would be getting useless information.

We can ask debug to display the variable’s data only when we have enabled verbose mode as described above. Add lines 18-21 in the screen capture below into your uptime.yaml playbook:



The number given with the verbosity argument specifies the minimum number of “v” that needs to be specified when enabling verbose mode before the debug mod- ule will print the variable’s value: verbosity: 1 displays the value with -v, -vv or

-vvv, while verbosity: 3 displays the value only with –vvv.

Now run the playbook with verbose mode (–v):

mbp15:aja sean$ ansible-playbook uptime.yaml --limit=bilbo -v Using /Users/sean/aja/ansible.cfg as config file

Junos Username: sean Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"ansible\_host": "bilbo"

}

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {"changed": false, "stdout":

["fpc0:\n \nCurrent time:

2010-01-03 02:56:32 UTC\nTime Source: LOCAL CLOCK \nSystem booted: 2010 01-01 00:10:04 UTC (2d 02:46

ago)\nProtocols started: 2010-01-01 00:16:47 UTC (2d 02:39 ago)\nLast configured: 2010 01-01 00:15:01

UTC (2d 02:41 ago) by root\n 2:56AM up 2 days, 2:46, 0 users, load averages: 0.10, 0.04, 0.01"], "stdout\_lines": [["fpc0:", "------------------------------------------------------------------------

--", "Current time: 2010-01-03 02:56:32 UTC", "Time Source: LOCAL CLOCK ", "System booted: 2010-01-01 00:10:04 UTC (2d 02:46 ago)", "Protocols started: 2010-01-01 00:16:47 UTC (2d 02:39 ago)", "Last

configured: 2010-01-01 00:15:01 UTC (2d 02:41 ago) by root", " 2:56AM up 2 days, 2:46, 0 users, load

averages: 0.10, 0.04, 0.01"]]}

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"uptime.stdout\_lines": [ [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-03 02:56:32 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:10:04 UTC (2d 02:46 ago)",

"Protocols started: 2010-01-01 00:16:47 UTC (2d 02:39 ago)",

"Last configured: 2010-01-01 00:15:01 UTC (2d 02:41 ago) by root",

" 2:56AM up 2 days, 2:46, 0 users, load averages: 0.10, 0.04, 0.01"

]

]

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=3 changed=0 unreachable=0 failed=0

Notice the TASK [debug] section displaying the contents of ansible\_host. Now run the playbook *without* enabling verbose mode (no –v argument). Notice that while the TASK [debug] section is present, it says it is skipping the device:

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

### Logging

[defaults]

Ansible can log the results of playbooks to a log file, including some information that is not displayed on screen. You can enable this feature by adding the log\_path parameter to the ansible.cfg file, like this (adjust the path and filename as needed):

inventory = inventory host\_key\_checking = False log\_path = ~/aja/ansible.log

Now when you run the playbook, Ansible will log the normal output and also sta- tus information from the junos\_command module:

2017-08-03 16:10:36,651 p=2133 u=sean | PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2017-08-03 16:10:36,689 p=2133 u=sean | TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2017-08-03 16:10:36,713 p=2133 u=sean | skipping: [bilbo]

2017-08-03 16:10:36,716 p=2133 u=sean | TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2017-08-03 16:10:37,080 p=2141 u=sean | creating new control socket for host bilbo:22 as user sean 2017-08-03 16:10:37,081 p=2141 u=sean | control socket path is /Users/sean/.ansible/pc/fbbc5f62b5 2017-08-03 16:10:37,081 p=2141 u=sean | current working directory is /Users/sean/aja

2017-08-03 16:10:37,081 p=2141 u=sean | using connection plugin netconf 2017-08-03 16:10:37,130 p=2141 u=sean | network\_os is set to junos

2017-08-03 16:10:37,130 p=2141 u=sean | ssh connection done, stating ncclient

2017-08-03 16:10:37,371 ncclient.transport.ssh Connected (version 2.0, client OpenSSH\_6.9)

2017-08-03 16:10:37,907 ncclient.transport.ssh Authentication (publickey) failed.

2017-08-03 16:10:38,048 ncclient.transport.ssh Authentication (password) successful!

2017-08-03 16:10:38,076 ncclient.transport.ssh Channel closed. (subsystem request rejected) 2017-08-03 16:10:40,343 ncclient.transport.session initialized: session-id=4377 | server\_ capabilities=<dictionary-keyiterator object at 0x10344f7e0>

2017-08-03 16:10:40,343 p=2141 u=sean | ncclient manager object created successfully

2017-08-03 16:10:40,344 p=2141 u=sean | connection established to bilbo in 0:00:03.262504 2017-08-03 16:10:41,083 p=2141 u=sean | incoming request accepted on persistent socket 2017-08-03 16:10:41,083 p=2141 u=sean | socket operation is CONTEXT

2017-08-03 16:10:41,085 p=2141 u=sean | socket operation is EXEC

2017-08-03 16:10:41,085 p=2141 u=sean | socket operation completed with rc 0

2017-08-03 16:10:42,000 p=2141 u=sean | incoming request accepted on persistent socket 2017-08-03 16:10:42,000 p=2141 u=sean | socket operation is EXEC

2017-08-03 16:10:42,001 ncclient.operations.rpc Requesting 'ExecuteRpc'

2017-08-03 16:10:43,897 p=2141 u=sean | socket operation completed with rc 0 2017-08-03 16:10:43,921 p=2133 u=sean | ok: [bilbo]

2017-08-03 16:10:43,924 p=2133 u=sean | TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2017-08-03 16:10:43,950 p=2133 u=sean | ok: [bilbo] => {

"uptime.stdout\_lines": [ [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-03 03:18:25 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:10:04 UTC (2d 03:08 ago)",

"Protocols started: 2010-01-01 00:16:47 UTC (2d 03:01 ago)",

"Last configured: 2010-01-01 00:15:01 UTC (2d 03:03 ago) by root",

" 3:18AM up 2 days, 3:08, 0 users, load averages: 0.09, 0.04, 0.02"

]

]

}

2017-08-03 16:10:43,951 p=2133 u=sean | PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2017-08-03 16:10:43,952 p=2133 u=sean | bilbo : ok=2 changed=0 unreachable=0 failed=0

2017-08-03 16:11:13,899 p=2141 u=sean | shutting down control socket, connection was active for 0:00:36.817827 secs

2017-08-03 16:11:13,899 ncclient.operations.rpc Requesting 'CloseSession'

The logged information can be helpful when faced with one of those generic “un- able to open shell” messages we saw earlier. Run the playbook again but this time provide invalid credential to the username and password prompts. When you re- view the log file, you should see messages similar to this:

2017-08-03 16:18:33,262 ncclient.transport.ssh Authentication (password) failed.

...

AuthenticationError: AuthenticationException('Authentication failed.',)

Those messages tell you the problem was authentication, not, for example, lack of connectivity to the managed device.

CAUTION Ansible does not automatically clear or delete the log file, which means that over time it can grow quite large. Consider enabling logging only when needed to troubleshoot a problem, or remember to delete the file occasionally.

## References

Ansible’s core modules for Junos: <http://docs.ansible.com/ansible/latest/list_of_network_modules.html>

Ansible glossary: <http://docs.ansible.com/ansible/glossary.html>

More about Ansible’s inventory file: <http://docs.ansible.com/ansible/latest/intro_inventory.html>

More about Ansible’s configuration file: <http://docs.ansible.com/ansible/latest/intro_configuration.html>

More about prompting for input: <http://docs.ansible.com/ansible/latest/playbooks_prompts.html>

More about variables: <http://docs.ansible.com/ansible/latest/playbooks_variables.html>

This book’s GitHub site: [https://github.com/Juniper/junosautomation/tree/master/ansible/](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible) [Automating\_Junos\_with\_Ansible](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible)

# Chapter 5

Junos, RPC, NETCONF, and XML

In Chapter 4, you executed a Junos CLI command using an Ansible playbook. That playbook provided an easy introduction to Ansible, but scripting CLI com- mands is not the approach that Juniper recommends for automating Junos operations.

This chapter introduces the preferred approach to automating Junos, which is to call RPC commands using NETCONF. The chapter starts with a little theory, briefly introducing RPC, NETCONF, and XML, then revises the playbook from Chapter 4 to use this preferred approach.

## Junos Management Architecture

Junos includes a management daemon (process), MGD, that is responsible for ex- ecuting commands. Those commands may come from various sources, including the Junos CLI.

Communication with MGD uses *remote procedure calls* (RPC), a mechanism for letting one process (e.g. the CLI) request services from another process (e.g.

MGD). The RPCs for communicating with MGD use XML (eXtensible Markup Language) to organize the request and the response.

### XML

XML is a text-based language for encoding data that is both human- and comput- er-readable. In some respects, XML serves a similar purpose to JSON and YAML, though XML looks quite different. This section briefly discusses the structure of XML data, and the References section contains links for those who wish to investi- gate it further.

When you execute a command at the Junos CLI, the CLI normally takes the XML data from MGD and reformats it in a human-friendly format. For example:

sean@vsrx1> **show system alarms**

1 alarms currently active

Alarm time Class Description

2017-07-21 17:17:36 UTC Minor Rescue configuration is not set

However, you can ask the CLI to display the XML data by appending | display xml to the command. For example:

sean@vsrx1> show system alarms | display xml

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">](http://xml.juniper.net/junos/15.1X49/junos)

<alarm-information [xmlns="http://xml.juniper.net/junos/15.1X49/junos-alarm">](http://xml.juniper.net/junos/15.1X49/junos-alarm)

<alarm-summary>

<active-alarm-count>1</active-alarm-count>

</alarm-summary>

<alarm-detail>

<alarm-time junos:seconds="1500657456"> 2017-07-21 17:17:36

</alarm-time>

<alarm-class>Minor</alarm-class>

<alarm-description>Rescue configuration is not set</alarm-description>

<alarm-short-description>no-rescue</alarm-short-description>

<alarm-type>Configuration</alarm-type>

</alarm-detail>

</alarm-information>

<cli>

<banner></banner>

</cli>

</rpc-reply>

Note that the XML data has a definite structure. (If you are familiar with HTML, you probably noticed that XML is similar.) XML data consists of one or more *ele- ments*, and each element is delimited by *tags*, the names within the angle brackets

< >.

There are three types of tags:

*Opening tags* identify the start of an element. For example, the tag <alarm-summary> indicates the start of an element named alarm-summary. The opening tag must contain the name of the element, but can also contain additional information called *attributes*, such as junos:seconds="1500657456" in the alarm-time tag above.

*Closing tags* identify the end of an element. Closing tags have a slash (/) before the element’s name, such as </alarm-summary>. A closing tag is paired with an opening tag and must have the same element name.

*Empty tags* are a shortcut way of representing an empty element (an element with no contents) and have a slash after the element name, such as <test/>. (The XML above does not contain an empty tag.) Empty tags are a shortcut for an opening and closing tag pair with nothing between them, such as <test></test>.

### NETCONF

The XML above contains an element rpc-reply, which contains elements alarm- information and cli. Element alarm-information contains elements alarm-summary and alarm-details. Element alarm-detail contains several additional elements, each of which contains additional information called CDATA, XML’s term for *character data*, such as the word Minor in the alarm-class element.

Note again the name of the outermost, or *root*, element: rpc-reply. The name rpc- reply is a reminder that this XML data is a response to a RPC (remote procedure call) from the CLI to the MGD process. We will revisit this idea in a few paragraphs.

The Network Configuration Protocol (NETCONF) is a standard protocol for re- mote administration of network devices. In fact, NETCONF is derived, in part, from Juniper’s RPC- and XML-based management architecture.

NETCONF uses XML data encoding for communication between the manage- ment system and the network device being managed, and supports the use of RPCs.

Enable NETCONF on a Junos device with the configuration-mode command:

set system services netconf ssh

By default, Junos will listen on TCP port 830 for NETCONF connections, once the service is enabled. The NETCONF port can be changed if your environment requires using a different port:

set system services netconf ssh port 2222

Junos will also accept NETCONF connections over the standard SSH port 22. This means that if your device already has SSH enabled, it is able to accept NET- CONF connections even without the settings configured above. We will see this work later in this chapter, and take advantage of this fact in a future chapter.

Junos can also limit the number of simultaneous NETCONF connections and the number of new connections accepted per minute:

set system services netconf ssh connection-limit 3 set system services netconf ssh rate-limit 3

On Junos SRX devices, depending on the interface used for management and the current security zone settings, you may also need to permit the netconf system ser- vice on one or more security zones. For example:

set security zones security-zone trust host-inbound-traffic system-services netconf

These were the changes on one of the author’s test systems; please make the ap- propriate changes on your test systems:

[edit]

sean@vsrx1# **show | compare**

[edit system services]

+ netconf {

+ ssh {

+ connection-limit 5;

+ rate-limit 5;

+ }

+ }

[edit security zones security-zone trust]

+ host-inbound-traffic {

+ system-services {

+ netconf;

+ }

+ }

## Finding RPCs

Juniper recommends using RPCs over NETCONF for off-box automation, but doing so requires knowing the RPCs. Fortunately, Junos makes it easy to find the RPC equivalent for most CLI commands: append " | display xml rpc" to the com- mand at the Junos CLI. For example:

sean@vsrx1> **show system alarms | display xml rpc**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">](http://xml.juniper.net/junos/15.1X49/junos)

<rpc>

<get-system-alarm-information>

</get-system-alarm-information>

</rpc>

<cli>

<banner></banner>

</cli>

</rpc-reply>

Look at the rpc element: its contents show the RPC equivalent for the *show system alarms* command, get-system-alarm-information, expressed in XML as opening and closing tags. In this example, the get-system-alarm-information element is empty.

However, a command that includes arguments will contain additional elements containing the arguments. For example:

sean@vsrx1> **show interfaces terse lo0 | display xml rpc**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">](http://xml.juniper.net/junos/15.1X49/junos)

<rpc>

<get-interface-information>

<terse/>

<interface-name>lo0</interface-name>

</get-interface-information>

</rpc>

<cli>

<banner></banner>

</cli>

</rpc-reply>

Here you can see that the CLI command *show interfaces* translates to the RPC get- interface-information and that the command-line arguments become additional elements within the get-interface-information element. The argument *terse* is ex- pressed as the empty tag <terse/>, while the interface name *lo0* is represented as CDATA between the opening tag <interface-name> and its closing tag.

## Revising the Uptime Playbook – Uptime Version 2.0

Let’s revise the uptime.yaml playbook from Chapter 4 to use an RPC.

First, we need to know the RPC we plan to call. Log in to one of your Junos de- vices and determine the RPC for the *show system uptime* command as discussed above:

sean@bilbo> **show system uptime | display xml rpc**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/15.1R6/junos">](http://xml.juniper.net/junos/15.1R6/junos)

<rpc>

<get-system-uptime-information>

</get-system-uptime-information>

</rpc>

<cli>

<banner>{master:0}</banner>

</cli>

</rpc-reply>

1|---

The RPC we need to call is get-system-uptime-information.

Ansible’s core modules provide two ways to call an RPC, both added in Ansible 2.3:

* a variation of the junos\_command module we used in Chapter 4
* the junos\_rpc module

This chapter discusses the junos\_rpc module. Either is a viable approach, but the author likes the clarity of keeping junos\_command for CLI commands and using ju- nos\_rpc for RPC commands.

Modify the uptime.yaml playbook so it looks like the following (line numbers added for discussion, do not include the line numbers or ‘|’ separator in your playbook; lines with changes are **boldfaced**):

2|- name: Get device uptime 3| hosts:

4| - all

5| connection: local 6| gather\_facts: no 7|

8| vars\_prompt:

9| - name: username

10| prompt: Junos Username 11| private: no

12|

13| - name: password

14| prompt: Junos Password 15| private: yes

16|

17| tasks:

**18| - name: get uptime using ansible core module 19| junos\_rpc:**

**20| rpc: get-system-uptime-information 21| output: text**

22| provider:

23| host: "{{ ansible\_host }}"

24| port: 22

25| username: "{{ username }}"

26| password: "{{ password }}" 27| register: uptime

28|

**29| - name: display uptimes 30| debug: var=uptime**

The changes made were, by line number:

Before line 18: Removed the debug task with the verbosity option. Line 19: Changed junos\_command module to junos\_rpc.

Lines 20, 21: Changed commands list to rpc argument (not a list). Added output argu- ment, which will be explained below.

Lines 29, 30: Removed the commented-out debug call and adjusted the remaining

debug call to display the full uptime variable, not just the uptime.stdout\_lines

element.

Also note that line 24, port: 22, was left in place so this playbook will use the stan- dard SSH port for NETCONF, not the preferred port 830. This was done simply to illustrate that it works.

Run the playbook; your output should look something like the following:

mbp15:aja sean$ **ansible-playbook uptime.yaml --limit=bilbo**

Junos Username: **sean**

Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"uptime": { "changed": false,

"output": "fpc0:\n \

nCurrent time: 2010-01-03 23:14:25 UTC\nTime Source: LOCAL CLOCK \nSystem booted: 2010 01-01 00:10:04

UTC (2d 23:04 ago)\nProtocols started: 2010-01-01 00:16:47 UTC (2d 22:57 ago)\nLast configured:

2010-01-01 00:15:01 UTC (2d 22:59 ago) by root\n11:14PM up 2 days, 23:04, 0 users, load averages:

0.27, 0.15, 0.06",

"output\_lines": [ "fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-03 23:14:25 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:10:04 UTC (2d 23:04 ago)",

"Protocols started: 2010-01-01 00:16:47 UTC (2d 22:57 ago)",

"Last configured: 2010-01-01 00:15:01 UTC (2d 22:59 ago) by root", "11:14PM up 2 days, 23:04, 0 users, load averages: 0.27, 0.15, 0.06"

],

"xml": "<rpc-reply message-id=\"urn:uuid:dfbce36d-7f35-437e-a19c-156042845bd3\">\n<output>\nfp c0:\n \nCurrent time:

2010-01-03 23:14:25 UTC\nTime Source: LOCAL CLOCK \nSystem booted: 2010 01-01 00:10:04 UTC (2d 23:04

ago)\nProtocols started: 2010-01-01 00:16:47 UTC (2d 22:57 ago)\nLast configured: 2010 01-01 00:15:01

UTC (2d 22:59 ago) by root\n11:14PM up 2 days, 23:04, 0 users, load averages: 0.27, 0.15, 0.06\n</ output>\n</rpc-reply>"

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

The output looks very similar to what we saw from uptime version 1.2. However, notice that the keys in the uptime dictionary are different – if we wanted the equiva- lent of the stdout\_lines key we referenced in uptime 1.3, we would need to refer- ence the output\_lines key instead.

TIP If you got an error message similar to “ImportError: No module named lxml” or “jxmlease is required but does not appear to be installed,” please review Chapter 4’s section, *Path to the Python Interpreter*.

This output is in text format. However, as we discussed earlier in this chapter, RPC calls natively use XML. The reason we got text is the output: text argument on line

21. This argument can be set to text, xml, or json. (Note that JSON output requires Junos support, which was added in Junos version 14.2).

Change line 21 to read output: xml (you could also delete the line because XML is the default output format) and run the playbook again:

mbp15:aja sean$ **ansible-playbook uptime-2.0.yaml --limit=bilbo**

Junos Username: **sean**

Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"uptime": {

"changed": false, "output": [

"<rpc-reply message-id=\"urn:uuid:c3c7b9a2-d8d6-4f09-b3e4-000eb18cb4b4\">",

"<multi-routing-engine-results>", "",

"<multi-routing-engine-item>", "",

"<re-name>fpc0</re-name>", "",

"<system-uptime-information>", "<current-time>",

"<date-time seconds=\"1262560832\">2010-01-03 23:20:32 UTC</date-time>", "</current-time>",

"<time-source> LOCAL CLOCK </time-source>", "<system-booted-time>",

"<date-time seconds=\"1262304604\">2010-01-01 00:10:04 UTC</date-time>", "<time-length seconds=\"256228\">2d 23:10</time-length>",

"</system-booted-time>", "<protocols-started-time>",

"<date-time seconds=\"1262305007\">2010-01-01 00:16:47 UTC</date-time>", "<time-length seconds=\"255825\">2d 23:03</time-length>",

"</protocols-started-time>", "<last-configured-time>",

"<date-time seconds=\"1262304901\">2010-01-01 00:15:01 UTC</date-time>", "<time-length seconds=\"255931\">2d 23:05</time-length>", "<user>root</user>",

"</last-configured-time>", "<uptime-information>",

"<date-time seconds=\"1262560833\">11:20PM</date-time>", "<up-time seconds=\"256259\">2 days, 23:10</up-time>",

"<active-user-count format=\"0 users\">0</active-user-count>", "<load-average-1>0.08</load-average-1>",

"<load-average-5>0.08</load-average-5>", "<load-average-15>0.06</load-average-15>", "</uptime-information>",

"</system-uptime-information>", "</multi-routing-engine-item>", "",

"</multi-routing-engine-results>", "</rpc-reply>"

],

"xml": "<rpc-reply message-id=\"urn:uuid:c3c7b9a2-d8d6-4f09-b3e4-000eb18cb4b4\">\n<multi- routing-engine-results>\n\n<multi-routing-engine-item>\n\n<re-name>fpc0</re-name>\n\n<system-uptime- information>\n<current-time>\n<date-time seconds=\"1262560832\">2010-01-03 23:20:32 UTC</date-time>\ n</current-time>\n<time-source> LOCAL CLOCK </time-source>\n<system-booted-time>\n<date-time seconds=\"1262304604\">2010-01-01 00:10:04 UTC</date-time>\n<time-length seconds=\"256228\">2d 23:10</time-length>\n</system-booted-time>\n<protocols-started-time>\n<date-time seconds=\"1262305007\">2010-01-01 00:16:47 UTC</date-time>\n<time-length seconds=\"255825\">2d 23:03</time-length>\n</protocols-started-time>\n<last-configured-time>\n<date-time seconds=\"1262304901\">2010-01-01 00:15:01 UTC</date-time>\n<time-length seconds=\"255931\">2d 23:05</time-length>\n<user>root</user>\n</last-configured-time>\n<uptime-information>\n<date-time seconds=\"1262560833\">11:20PM</date-time>\n<up-time seconds=\"256259\">2 days, 23:10</up-time>\ n<active-user-count format=\"0 users\">0</active-user-count>\n<load-average-1>0.08</load-average-1>\ n<load-average-5>0.08</load-average-5>\n<load-average-15>0.06</load-average-15>\n</uptime- information>\n</system-uptime-information>\n</multi-routing-engine-item>\n\n</multi-routing-engine- results>\n</rpc-reply>"

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

Now the JSON output contains the XML data returned by Junos (and the JSON dictionary’s keys have changed again!)

If your test device is running Junos 14.2 or newer, try changing line 21 to read output: json and run the playbook again. This time the uptime variable should in- clude an output key with JSON-formatted data:

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"uptime": {

"changed": false, "output": {

"multi-routing-engine-results": [

{

"multi-routing-engine-item": [

{

"re-name": [

{

"data": "fpc0"

}

],

"system-uptime-information": [

{

"attributes": {

"xmlns": ["http://xml.juniper.net/junos/15.1R6/junos"](http://xml.juniper.net/junos/15.1R6/junos)

},

"current-time": [

{

"date-time": [

{

"attributes": {

"junos:seconds": "1262561325"

},

"data": "2010-01-03 23:28:45 UTC"

}

]

}

],

"last-configured-time": [

{

"date-time": [

{

"attributes": {

"junos:seconds": "1262304901"

},

"data": "2010-01-01 00:15:01 UTC"

}

],

"time-length": [

{

"attributes": { "junos:seconds": "256424"

},

"data": "2d 23:13"

}

],

"user": [

{

"data": "root"

}

]

}

],

"protocols-started-time": [

{

"date-time": [

{

"attributes": {

"junos:seconds": "1262305007"

},

"data": "2010-01-01 00:16:47 UTC"

}

],

"time-length": [

{

"attributes": { "junos:seconds": "256318"

},

"data": "2d 23:11"

}

]

}

],

"system-booted-time": [

{

"date-time": [

{

"attributes": {

"junos:seconds": "1262304604"

},

"data": "2010-01-01 00:10:04 UTC"

}

],

"time-length": [

{

"attributes": { "junos:seconds": "256721"

},

"data": "2d 23:18"

}

]

}

],

*<<< output truncated to conserve space >>>*

If you get an error that includes the phrase “No JSON object could be decoded,” the device probably has a Junos version that does not support JSON output.

## Juniper’s Galaxy Modules – Uptime Version 2.1

Let’s revise our uptime playbook to use Juniper’s Galaxy modules instead of An- sible’s core modules. Let’s also use the standard NETCONF port 830, rather than the SSH port.

To tell a play that it should use Juniper’s Galaxy module Juniper.junos you need to add a roles key to the play, like this (see boldfaced lines):

* name: Get device uptime hosts:

- all

**roles:**

* **Juniper.junos** connection: local gather\_facts: no

Juniper’s Galaxy modules normally have different names than Ansible’s similar core modules, but in the case of the modules for executing an RPC, both core and Galaxy are called junos\_rpc. (This name clash is unfortunate and can create a prob- lem, which we discuss in the next section.) However, the Galaxy module’s argu- ments are a little different. The following is the modified playbook (line numbers shown for discussion):

1|---

2|- name: Get device uptime 3| hosts:

4| - all

5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars\_prompt:

11| - name: username

12| prompt: Junos Username 13| private: no

14|

15| - name: password

16| prompt: Junos Password 17| private: yes

18|

19| tasks:

20| - name: get uptime using galaxy module 21| junos\_rpc:

22| rpc: get-system-uptime-information 23| format: xml

24| dest: "{{ inventory\_hostname }}-uptime.xml" 25| host: "{{ ansible\_host }}"

26| port: 830

27| user: "{{ username }}"

28| passwd: "{{ password }}"

29| register: uptime 30|

31| - name: display uptimes 32| debug: var=uptime

Table 5.1 shows the Galaxy module play and the core module play side-by-side for easy comparison. Table 5.2 contrasts the arguments used by the two different ju- nos\_rpc modules in their respective plays, and how they save or return results.

Table 5.1 *Comparing junos\_rpc Plays*

|  |  |
| --- | --- |
| Juniper’s Galaxy junos\_rpc | Ansible’s core junos\_rpc |
| - name: get uptime using galaxy module junos\_rpc:  rpc: get-system-uptime-information format: xml  dest: "{{ inventory\_hostname }}-uptime.xml" host: "{{ ansible\_host }}"  port: 830  user: "{{ username }}"  passwd: "{{ password }}" register: uptime | - name: get uptime using ansible core module junos\_rpc:  rpc: get-system-uptime-information output: text  provider:  host: "{{ ansible\_host }}" port: 22  username: "{{ username }}"  password: "{{ password }}" register: uptime |

Table 5.2 *Comparing junos\_rpc Arguments and Results*

|  |  |
| --- | --- |
| Juniper’s Galaxy junos\_rpc | Ansible’s core junos\_rpc |
| Argument format specifies the desired result type (JSON, text, or XML). | Argument output specifies the desired result type (JSON, text, or XML). |
| Host connection information is contained in arguments to junos\_rpc. | Host connection information is contained within a dictionary called provider, which is an argument to junos\_rpc. |
| Argument user specifies the username for the device connection. | Within the provider dictionary, element username  specifies the username for the device connection. |
| Argument passwd specifies the password for the device connection. | Within the provider dictionary, element password  specifies the password for the device connection. |
| Argument dest specifies the name of the file in which to store the RPC results. | Returns RPC results so they can be stored with register. |

The last difference noted in Table 5.2 might require a bit more explanation. As we saw with our previous versions of the uptime playbook, Ansible’s core junos\_com- mand and junos\_rpc modules return the output from a command or RPC back to the playbook; these results can be stored in a variable using the register option on the task, and later tasks in the playbook can access the output from that variable.

Juniper’s Galaxy junos\_rpc module takes a different approach: the output from the RPC is stored in a file whose filename is provided by the dest argument. The argu- ment dest: "{{ inventory\_hostname }}-uptime.xml" will cause junos\_rpc to create a file whose name is the device’s inventory hostname with “-uptime.xml” appended, such as bilbo-uptime.xml.

Let’s run the new playbook and see the results:

mbp15:aja sean$ **ansible-playbook uptime.yaml**

Junos Username: sean Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using galaxy module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [display uptimes] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"uptime": {

"changed": true, "check\_mode": false, "kwargs": {},

"rpc": "get-system-uptime-information"

}

}

ok: [bilbo] => {

"uptime": {

"changed": true, "check\_mode": false, "kwargs": {},

"rpc": "get-system-uptime-information"

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

vsrx1 : ok=2 changed=1 unreachable=0 failed=0

TIP If you get a fatal error whose message is similar to "Unsupported param- eters for (junos\_rpc) module," please skip forward to the next section of this chapter, "Dueling junos\_rpc modules," for some tips to troubleshoot the problem.

Juniper’s junos\_rpc module returns some data about the RPC that was executed, but not the RPC’s results; as we noted above, those are stored in a file. To view the RPC’s results, display the file:

mbp15:aja sean$ **ls -l \*.xml**

-rw-r--r-- 1 sean staff 1184 Aug 7 10:31 bilbo-uptime.xml

-rw-r--r-- 1 sean staff 1036 Aug 7 10:31 vsrx1-uptime.xml

mbp15:aja sean$ **cat bilbo-uptime.xml**

<multi-routing-engine-results>

<multi-routing-engine-item>

<re-name>fpc0</re-name>

<system-uptime-information>

<current-time>

<date-time seconds="1262813928">2010-01-06 21:38:48 UTC</date-time>

</current-time>

<time-source> LOCAL CLOCK </time-source>

<system-booted-time>

<date-time seconds="1262304604">2010-01-01 00:10:04 UTC</date-time>

<time-length seconds="509324">5d 21:28</time-length>

</system-booted-time>

<protocols-started-time>

<date-time seconds="1262305007">2010-01-01 00:16:47 UTC</date-time>

<time-length seconds="508921">5d 21:22</time-length>

</protocols-started-time>

<last-configured-time>

<date-time seconds="1262750221">2010-01-06 03:57:01 UTC</date-time>

<time-length seconds="63707">17:41:47</time-length>

<user>sean</user>

</last-configured-time>

<uptime-information>

<date-time seconds="1262813929">9:38PM</date-time>

<up-time seconds="509355">5 days, 21:29</up-time>

<active-user-count format="0 users">0</active-user-count>

<load-average-1>0.19</load-average-1>

<load-average-5>0.06</load-average-5>

<load-average-15>0.01</load-average-15>

</uptime-information>

</system-uptime-information>

</multi-routing-engine-item>

</multi-routing-engine-results>

## Dueling junos\_rpc Modules

As we mentioned above, both Ansible and Juniper have a module called *junos\_ rpc*. Ansible added their core junos\_rpc module in May 2017 (Ansible 2.3). Juni- per’s Galaxy modules have included their junos\_rpc module since February 2016 (version 1.3.0 of the Galaxy modules).

Some systems may use the core junos\_rpc module, not the Galaxy junos\_rpc mod- ule, even when the playbook includes the Juniper.junos role. Because the core and Galaxy modules take different arguments, if your system is experiencing this prob- lem, the symptom you will probably see is an “unsupported parameters” error message during playbook execution, similar to the following:

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "Unsupported parameters for (junos\_rpc) module: dest,format,passwd,user. Supported parameters include: args,host,output,password, port,provider,rpc,ssh\_keyfile,timeout,transport, username"}

One fix for this problem is to copy the Galaxy junos\_rpc module to junos\_run\_rpc, then update your playbook to call junos\_run\_rpc instead of junos\_rpc.

If you need to make the module fix yourself, start by locating the modules for the installed Juniper.junos role (your location may differ from what is shown here):

mbp15:~ sean$ **sudo ansible-galaxy info Juniper.junos | grep 'path:'**

path: [u'/etc/ansible/roles']

Change to the Juniper.junos/library/ subdirectory of the role’s path:

mbp15:~ sean$ **cd /etc/ansible/roles/Juniper.junos/library/**

mbp15:library sean$

Copy the module to a new (unique) name:

mbp15:library sean$ **ls \*rpc**

junos\_rpc

mbp15:library sean$ **sudo cp junos\_rpc junos\_run\_rpc**

mbp15:library sean$ **ls \*rpc**

junos\_rpc junos\_run\_rpc

## Uptime Version 2.2

Let’s do a little bit of cleanup on uptime.yaml. Make the following changes:

* + Remove the port: 830 line from the junos\_rpc play; 830 is the default port so we do not need to specify it.
  + Alter the format to text and change the dest filename’s extension to .txt from

.xml.

* + Finally, as the information being captured by register is not terribly interesting, remove the register line and the following debug task.

This is the modified playbook (line numbers added):

1|---

2|- name: Get device uptime 3| hosts:

4| - all

5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars\_prompt:

11| - name: username

12| prompt: Junos Username 13| private: no

14|

15| - name: password

16| prompt: Junos Password 17| private: yes

18|

19| tasks:

20| - name: get uptime using galaxy module 21| junos\_rpc:

22| rpc: get-system-uptime-information

**23| format: text**

**24| dest: "{{ inventory\_hostname }}-uptime.txt"**

25| host: "{{ ansible\_host }}"

26| user: "{{ username }}"

27| passwd: "{{ password }}"

Run the playbook and display the results in the files:

mbp15:aja sean$ **ansible-playbook uptime.yaml**

Junos Username: sean Junos Password:

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using galaxy module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=1 unreachable=0 failed=0

vsrx1 : ok=1 changed=1 unreachable=0 failed=0 mbp15:aja sean$ **cat bilbo-uptime.txt**

fpc0:

--------------------------------------------------------------------------

Current time: 2010-01-06 22:22:46 UTC Time Source: LOCAL CLOCK

System booted: 2010-01-01 00:10:04 UTC (5d 22:12 ago)

Protocols started: 2010-01-01 00:16:47 UTC (5d 22:05 ago)

Last configured: 2010-01-06 03:57:01 UTC (18:25:45 ago) by sean

10:22PM up 5 days, 22:13, 0 users, load averages: 0.12, 0.05, 0.01 mbp15:aja sean$ **cat vsrx1-uptime.txt**

Current time: 2017-07-29 19:03:52 UTC Time Source: LOCAL CLOCK

System booted: 2017-07-29 11:35:53 UTC (07:27:59 ago)

Protocols started: 2017-07-29 11:35:54 UTC (07:27:58 ago)

Last configured: 2017-07-29 16:44:06 UTC (02:19:46 ago) by sean

7:03PM up 7:28, 1 user, load averages: 0.01, 0.03, 0.00

## References

Example Ansible playbooks by another Juniper engineer, Khelil Sator: <https://github.com/ksator/junos-automation-with-ansible>

Juniper’s Galaxy module documentation (version 1.4.3): <http://junos-ansible-modules.readthedocs.io/en/1.4.3/>

NETCONF background: <https://en.wikipedia.org/wiki/NETCONF>

XML tutorial: <https://www.w3schools.com/xml/>

# Chapter 6 Using SSH Keys

The playbooks developed in Chapters 4 and 5 prompt the user for the username and password needed to access the managed network device. This chapter explores an alternative, the use of SSH keys for device authentication.

## What is an SSH Key Pair?

SSH, and NETCONF over SSH, require that the client authenticate to the Junos device, the same credentials used for console access. Basic authentication uses a username and password; the relevant configuration on a Junos device would look something like this:

sean@vsrx1> **show configuration system login**

user sean {

uid 2000;

class super-user; authentication {

encrypted-password "$5$gG...PYdyZp0t4hhb94Jp495FRiICB62"; ## SECRET-DATA

}

}

SSH offers an alternative authentication method based on asymmetric cryptogra- phy, also known as public-key cryptography. The user generates a *key pair*, a matched set of encryption keys. The *public key* needs to be installed on the Junos devices (or other servers); the *private key* is on the user’s client computer(s). As the name implies, the private key must be kept *private*; like a password, the private key should never be shared with anyone because sharing the private key would allow another person to authenticate as you. By contrast, the public key can be shared, such as being placed on multiple Junos devices or servers.

When the user establishes their SSH session with the server, they use their respec- tive keys to authenticate the connection. No password is necessary. This can be very convenient for scheduled automation tasks as there might not be a person around at the scheduled time to enter a password, but the SSH private key on the client computer is still available.

When the user generates their SSH key pair, they can choose to associate a pass- phrase with the private key. This adds an extra layer of security – the user needs to enter the correct passphrase before the client will initiate the connection to the server, which means an unauthorized person sitting at an authorized user’s com- puter cannot establish the connection. However, a private key with a passphrase is less useful for scheduled automation tasks because a person may not be available to enter the passphrase at the scheduled time.

## Generating a Key Pair

On most UNIX-type systems, you generate an SSH key pair using a program from the OpenSSH collection of programs. MacOS includes the OpenSSH programs. UNIX and Linux systems normally include OpenSSH, or make the OpenSSH cli- ent programs available through their package manager (please take a moment to install it now if needed on your system). This means that generating and using SSH key pairs is consistent across most UNIX-type systems. The discussions in this chapter about generating SSH key pairs, and the discussion about client configura- tion for multiple key pairs, focuses on OpenSSH systems.

Microsoft Windows does not include an SSH client and thus does not include the program needed to generate key pairs. Many third-party SSH clients for Windows provide the ability to generate SSH key pairs, and most can be set up to use mul- tiple key pairs. Check the documentation for your SSH client. Unfortunately, there is too much variation between the various Windows SSH clients to document them here. If your Windows SSH client cannot generate SSH key pairs, but you have access to a UNIX-type system, see if your Windows SSH client can import key pairs generated by OpenSSH. If you do not have access to a UNIX or Linux system, consider installing the Cygwin environment (<http://www.cygwin.com/)>on your Windows system and use Cygwin’s OpenSSH tools.

NOTE This chapter assumes your system does not already have any SSH key pairs installed, and the instructions make no effort to preserve existing key pairs or configuration settings. If you are already using SSH key pairs, please take the necessary precautions to protect those files.

OpenSSH includes the command-line program ssh-keygen for generating SSH key pairs. You can run simply ssh-keygen and it will prompt for answers to a few ques- tions, or you can provide command-line arguments for a variety of options.

Open your shell or terminal and run ssh-keygen. Hit Enter or Return at the “Enter file…” prompt to accept the default filename. Enter your passphrase at the two “passphrase” prompts. The result should look something like this:

mbp15:~ sean$ **ssh-keygen**

Generating public/private rsa key pair.

Enter file in which to save the key (/Users/sean/.ssh/id\_rsa): **<enter>**

Enter passphrase (empty for no passphrase): **<enter passphrase>**

Enter same passphrase again: **<re-enter passphrase>**

Your identification has been saved in /Users/sean/.ssh/id\_rsa. Your public key has been saved in /Users/sean/.ssh/id\_rsa.pub. The key fingerprint is:

SHA256:wAe+lxRhiOZBHvaG5wrTxGmvv5dKgo1HwHyVwESAYXs [sean@mbp15.local](mailto:sean@mbp15.local) The key's randomart image is:

+---[RSA 2048] +

|  |  |
| --- | --- |
| |.+.\*Booo+. | | |
| |oo.+=Ooo . | | |
| | .+EO.B o | | |
| | .\*.= = . | | |
| | o o + S | | |
| | B o . | | |
| | o \* . . | | |
| | . + o | | |
| | ++ | | |

+----[SHA256] +

The single filename prompt is a little misleading because ssh-keygen actually gener- ates two files, the public and private key files. The name entered at the prompt (or the default ~/.ssh/id\_rsa) becomes the private key’s filename, and the public key’s filename will have the extension .pub added:

mbp15:~ sean$ **ls -l ~/.ssh/id\***

-rw------- 1 sean staff 1679 Jul 23 19:36 /Users/sean/.ssh/id\_rsa

-rw-r--r-- 1 sean staff 399 Jul 23 19:36 /Users/sean/.ssh/id\_rsa.pub

The private key does not have a passphrase. We will address this in a later section of this chapter.

## Installing the Public Key on a Junos Device

Let’s install the public key on a Junos device. Display the public key file to the ter- minal or open it in your text editor, whichever you prefer:

mbp15:~ sean$ **cat ~/.ssh/id\_rsa.pub**

ssh-rsaB3NzaC1yc2EAAAADAQABAAABAQDNHawZMgHWTQ+uNKIt4l6I7eZdGgeXPHHx8KQxsOboAlbKuRPHItGITmXbPKOVTXoiY jdkH1LGGBLNcMNJ9pA8skjjOgGfa1VrvtzNp6/1+YY8iRXsSvPN6ZuQgthITUpg1qFNRRFIrP1ygSxhFBPY+ULmgdt5YzPs5k4G0 MnMD5JavffVsEeUzB/HTtT+orT7baf/w4yLi0s0hX6oQL1ycFa9NmU7wZl1qLPzPH8bxusUEYUL/RagSAoK3AAATwobLqggDpCgp Yr+POlxdYVSf9uI0xE7X2G4bpESFchvyEAXw0eYNHjxG5QXEGimctF/9MOE8gjniIJeUsUJzS8b

Copy the key from the “ssh-rsa” through the end.

On the Junos device, enter config mode and navigate to your user account’s configuration:

sean@vsrx1> **configure**

Entering configuration mode

sean@vsrx1# **edit system login user sean**

[edit system login user sean] sean@vsrx1#

Now add the public key using the set authentication ssh-rsa command, placing the public key in quotes and using the paste option of your SSH client or terminal (the key is shown abbreviated below):

[edit system login user sean]

sean@vsrx1# **set authentication ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b** [**sean@mbp15.local"**](mailto:sean@mbp15.local)

[edit system login user sean] sean@vsrx1#

Commit that change. Now, from your computer’s command line or your SSH cli- ent, SSH to the device:

mbp15:~ sean$ **ssh vsrx1**

Enter passphrase for key '/Users/sean/.ssh/id\_rsa': **<enter passphrase>**

--- JUNOS 15.1X49-D90.7 built 2017-04-29 06:51:16 UTC

sean@vsrx1>

Note that you were prompted for the private key’s passphrase, but were not prompted for the device password; the SSH key pair took care of the authentica- tion with the device. Nice!

## Caching Your Private Key Passphrase

Having a passphrase on your private key is a good security precaution. However, it does create a problem when running Ansible playbooks: Ansible will not pause to prompt for the passphrase to unlock the private key, which results in authenti- cation errors connecting to the managed devices. Even if Ansible would stop, you would likely need to enter the passphrase for each device, which would be a head- ache if you were running a playbook on dozens of devices.

MacOS, UNIX, and Linux systems offer a way to cache your passphrase prior to running an Ansible playbook. The passphrase will be retained for a limited time, but during that time it will be provided to subsequent SSH sessions that use that private key, including any NETCONF-over-SSH connections established by an Ansible playbook.

### Linux passphrase caching

Linux and UNIX systems using OpenSSH provide the ssh-agent and ssh-add com- mands, which work together to cache SSH key passphrases. The ssh-agent com- mand launches an authentication agent that can cache SSH passphrases, while the ssh-add command adds passphrases for specific public keys to the agent’s cache.

Start by using ssh-agent to launch a new instance of the command shell as an au- thentication agent client (if you use a shell other than bash, adjust the command accordingly):

sean@ubuntu:~$ **ssh-agent bash**

sean@ubuntu:~$

While the screen has not visibly updated, you are now running within a second instance of the command shell, which is a client of ssh-agent.

Now use ssh-add to cache your passphrase for your private key. When run without arguments ssh-add assumes you wish to cache the passphrase for ~/.ssh/id\_rsa:

sean@ubuntu:~$ **ssh-add**

Enter passphrase for /home/sean/.ssh/id\_rsa: **<enter passphrase>**

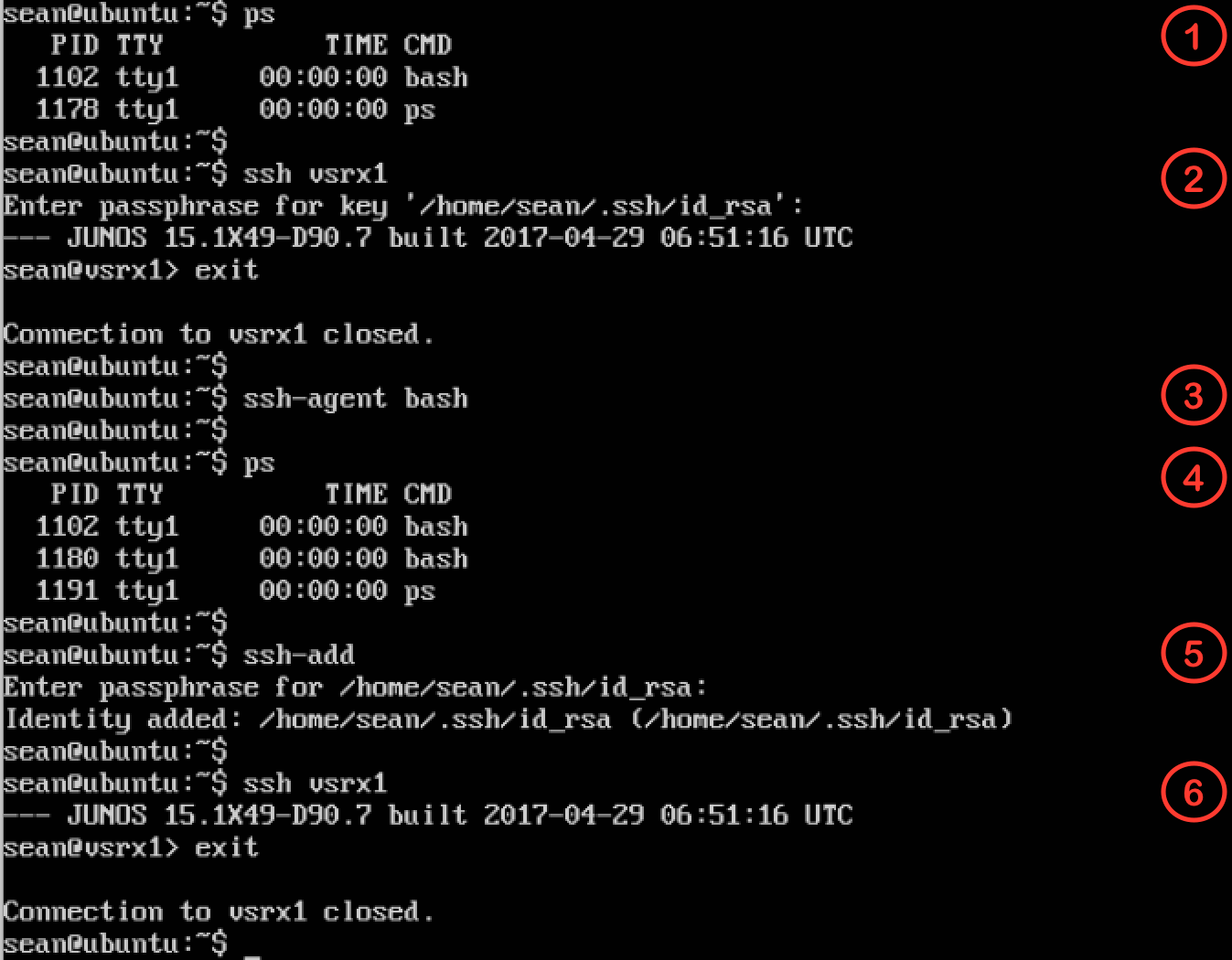
Identity added: /home/sean/.ssh/id\_rsa (/home/sean/.ssh/id\_rsa)

Now you can SSH to hosts that use the matching public key, or use scripts to ac- cess those hosts, without needing to enter your passphrase every time. Keep in mind that the cached passphrase is available only to tasks run within the shell that is a client to ssh-agent, and when you exit that shell the cached passphrase is forgotten.

The following screen capture shows the entire process. Contrast the output of the two ps commands (#1 and #4) to confirm that ssh-agent (#3) launches a second instance of bash. The example also shows that an SSH session (#2) before using

ssh-add (#5) prompts for a passphrase while a similar session (#6) after ssh-add

does not.



### MacOS Passphrase Caching

On some versions of MacOS, including El Capitan (10.11), private key passphrase caching is automatic. When you initiate a manual SSH session (for example,

ssh vsrx1) to a device that uses key-based authentication, MacOS will display a dia- log box similar to the following to prompt for the passphrase for the private key.

The passphrase you enter in the dialog box will be cached until you log out.



With MacOS Sierra (10.12) passphrase caching is not enabled by default, but you can use the ssh-add command discussed above for Linux systems. Curiously, MacOS does not seem to require that you first run ssh-agent:

mbp15:~ sean$ **ssh vsrx1**

Enter passphrase for key '/Users/sean/.ssh/id\_rsa': **<enter passphrase>**

--- JUNOS 15.1X49-D90.7 built 2017-04-29 06:51:16 UTC

sean@vsrx1> exit

Connection to vsrx1 closed. mbp15:~ sean$ **ssh-add**

Enter passphrase for /Users/sean/.ssh/id\_rsa: **<enter passphrase>**

Identity added: /Users/sean/.ssh/id\_rsa (/Users/sean/.ssh/id\_rsa)

mbp15:~ sean$ **ssh vsrx1**

--- JUNOS 15.1X49-D90.7 built 2017-04-29 06:51:16 UTC

sean@vsrx1> exit

Connection to vsrx1 closed.

## Multiple Key Pairs

The ssh client program will look in id\_rsa by default for a private key when estab- lishing a connection to a server, which indicates it accepts key-based authentica- tion. However, ssh can use private keys in other files, which means you can use different key pairs with different servers. For example, assume you need to access a server named Gandalf, but you want to have a unique key pair for that connec- tion because Gandalf is outside your organization (perhaps it is owned by a cus- tomer). Let’s generate another key pair using a different filename, and let’s use command-line arguments this time to illustrate that approach:

mbp15:~ sean$ **ssh-keygen -f ~/.ssh/gandalf -N 'my!passphrase' -C "Seans key for Gandalf"**

Generating public/private rsa key pair.

Your identification has been saved in /Users/sean/.ssh/gandalf. Your public key has been saved in /Users/sean/.ssh/gandalf.pub. The key fingerprint is:

SHA256:AVHrNl9iI85NbKK/l45Pm3B2tXR4Czw9lbmMj6z85DU Seans key for Gandalf The key's randomart image is:

+---[RSA 2048] +

| oo. |

|  |  |
| --- | --- |
| | | . . o| |
| | | o o.| |
| | | . o . +.o| |
| | | S B .=+=o| |
| | | = X +.o=+o| |
| | | . + \*..+.E | |
| | | . \*+++ . .| |
| | | +==o.o | |

+----[SHA256] +

The –f argument provides the private key filename (remember to include the path, or the files will be put in the current directory). The –N argument provides a pass- phrase for the private key. The –C argument includes a comment at the end of the public key file, which can be helpful when you have a number of key pairs for dif- ferent servers or devices. (There are a number of other arguments, including select- ing a key type other than the default RSA or a key bit length other than the default 2048. See the manpage or online documentation for more information.)

Confirm that the key files were created:

mbp15:~ sean$ **ls -l ~/.ssh/g\***

-rw------- 1 sean staff 1766 Aug 7 14:46 /Users/sean/.ssh/gandalf

-rw-r--r-- 1 sean staff 403 Aug 7 14:44 /Users/sean/.ssh/gandalf.pub

And note the comment at the end of the public key file:

mbp15:~ sean$ **cat ~/.ssh/gandalf.pub**

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCdHQ5lDZl6AjY1ko0kknQPek9xL9E8CQVMrF9YbEWOdwaqdPWbQxiT0Qe58Za/ weloMlgihOPpvlutdB6Ou0oZHc4H3IwEuUFxd64+4ZX6PpSLdqpkjluZO6GRVJ7iANLkbVtRH4EB3CIRuxzDmrdaC34YEr7QC+ty NlvxRHjXV7JTHlg1N3mDE5Z0DOR+QOKhiN37oaXh5+V7FkA5zk7rgSShaI4j6rsXWktHyIEtJ8HGViyP/wXADRwqcT8fnoKXqt1W nfB3gVusFyX9EZN0dQiduAOdR+N3UQIiZdfDh5ReRJfiZW6MWthucHHT9xn5cIOjBPPweq4wNvj/4x

mb Seans key for Gandalf

Host gandalf User ssawtell

Give the public key to the administrator of Gandalf and ask them to install the public key in your user account.

Next, you need to let your SSH client know to use the new private key for sessions with Gandalf. OpenSSH does this using a text configuration file ~/.ssh/config.

Create the file ~/.ssh/config with the following lines, or add these lines to the exist- ing file:

IdentityFile ~/.ssh/Gandalf

The Host line specifies the hostname for the server. The User line is the username to use when connecting to that server; this is particularly useful when the username for the server does not match your local username, as illustrated here. The Identi- tyFile line specifies the private key to use when connecting to that server.

Once your public key is installed on the server, you should be able to SSH to Gan- dalf using the alternate key pair:

mbp15:~ sean$ **ssh gandalf**

Enter passphrase for key '/Users/sean/.ssh/gandalf':

Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-83-generic x86\_64)

* Documentation: https://help.ubuntu.com
* Management: https://landscape.canonical.com
* Support: https://ubuntu.com/advantage Last login: Mon Aug 7 11:56:29 2017 from 192.0.2.1 ssawtell@gandalf:~$ **exit**

logout

Connection to gandalf closed.

If you want to cache the passphrase for the private key for Gandalf, specify the path and filename for the private key when using the ssh-add command, like this:

mbp15:~ sean$ **ssh-add ~/.ssh/gandalf**

Enter passphrase for /Users/sean/.ssh/gandalf:

Identity added: /Users/sean/.ssh/gandalf (/Users/sean/.ssh/gandalf)

The ~/.ssh/config file can contain similar entries for multiple servers. There are also many other options that can be specified in the file; see the manpage for ssh\_ config for more information.

NOTE The author has had mixed experience using alternate SSH key pairs with Ansible playbooks. Try to use the default id\_rsa key pair for any devices that you wish to manage with Ansible.

## Security Considerations

Check with your company’s Information Security team before using SSH key- based authentication. They may have restrictions or requirements that you will need to follow; for example, they may require that private keys have passphrases and define a minimum passphrase complexity, or they may require specific proto- cols or key bit lengths, or that the key pair be replaced at regular intervals.

Protect the private key file. It should be stored only on devices you control, and should always have file permissions that prevent anyone but you from reading the file. If you ever suspect the private key has been compromised, generate a new key pair and replace the old pair.

If the private key has a passphrase, keep the passphrase secret just as you would with your logon password.

If you have a private key without a passphrase in order to support scheduled auto- mation tasks, consider making the corresponding account on the Junos devices a read-only account. This mitigates the damage that could be caused should the pri- vate key be compromised. Remember, a private key with no passphrase can be used by anyone who gets the key file, so a *leaked* private key that authenticates to an account that has administrative privileges is a major security concern.

## Playbook Using Key-based Authentication – Uptime 3

Let’s modify the uptime playbook from Chapter 5 to use SSH key-based authenti- cation. This mostly means deleting lines that are no longer needed. Juniper’s Gal- axy modules default to using your local (computer) username and SSH key for device communication.

Be sure your SSH public key is installed on at least one of your test devices, as de- scribed earlier in this chapter. In Chapter 7, we will show how to create a playbook to install your SSH public key on your devices, so if you want to manually install the public key on only a subset of your test environment, that is fine.

Remove the entire vars\_prompt section of the file, and also remove the user and pass- wd arguments from the junos\_rpc task. The resulting playbook should look like this (line numbers added):

1|---

2|- name: Get device uptime 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| tasks:

11| - name: get uptime using ansible core module 12| junos\_rpc:

13| rpc: get-system-uptime-information 14| format: text

15| dest: "{{ inventory\_hostname }}-uptime.txt" 16| host: "{{ ansible\_host }}"

NOTE This playbook example assumes that your username on your computer is the same as your username on the network devices. If your computer username is different from your device username, retain the user argument but set it to your device username, for example:

...

11| - name: get uptime using ansible core module 12| junos\_rpc:

13| rpc: get-system-uptime-information 14| format: text

15| dest: "{{ inventory\_hostname }}-uptime.txt" 16| host: "{{ ansible\_host }}"

**17| user: deviceuser**

If you have not already done so, use ssh-add (and ssh-agent if needed) to cache your private key passphrase:

mbp15:aja sean$ **ssh-add**

Enter passphrase for /Users/sean/.ssh/id\_rsa: **<enter passphrase>**

Identity added: /Users/sean/.ssh/id\_rsa (/Users/sean/.ssh/id\_rsa)

Run the playbook (remember to --limit the hosts if you have not installed your SSH key on all your test systems):

mbp15:aja sean$ **ansible-playbook uptime.yaml**

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=1 unreachable=0 failed=0

vsrx1 : ok=1 changed=1 unreachable=0 failed=0

mbp15:aja sean$ **cat bilbo-uptime.txt**

fpc0:

--------------------------------------------------------------------------

Current time: 2017-08-07 14:15:59 UTC Time Source: LOCAL CLOCK

System booted: 2017-08-01 13:02:52 UTC (6d 01:13 ago)

Protocols started: 2017-08-01 13:09:35 UTC (6d 01:06 ago)

Last configured: 2010-01-07 01:15:11 UTC (395w4d 13:00 ago) by sean 2:15PM up 6 days, 1:13, 0 users, load averages: 0.10, 0.11, 0.11

## References

Asymmetric cryptography: <https://en.wikipedia.org/wiki/Public-key_cryptography>

More about SSH: <https://en.wikipedia.org/wiki/Secure_Shell>

More about ssh-keygen: <https://en.wikipedia.org/wiki/Ssh-keygen>

Cygwin: <http://www.cygwin.com/>

OpenSSH: <http://www.openssh.com/>

# Chapter 7

Generating and Installing Junos Configuration Files

A common use for automation is configuring network devices, whether a new con- figuration on a new device or a change of configuration to an existing device. This chapter explores how Ansible can generate Junos configurations and apply those configurations to devices.

## Configuration Files

You can modify a Junos device’s configurations by loading a configuration file on the device. The configuration file can be a partial configuration containing only the settings to be changed, or it can be a complete device configuration that replac- es the entire existing configuration.

Configuration files are text files, typically using one of two formats. The “set” format contains a set of Junos configuration statements similar to what you would enter at the configuration-mode command line. For example:

set system name-server 1.2.3.4 set system name-server 1.2.3.5 delete system name-server 9.8.7.6

In other words, a set file’s contents are similar to what you would get if you showed (a portion of) a device’s configuration with the "| display set" modifier:

[edit]

sean@vsrx1# **show system name-server | display set**

set system name-server 1.2.3.4 set system name-server 1.2.3.5

The “text” or “config” format looks like the normal Junos configuration, or a portion thereof, complete with braces, semicolons, and indented lines:

system {

name-server {

1.2.3.4;

1.2.3.5;

}

}

Configuration files in either “set” or “text” format can be loaded either manually or with Ansible. We will first explore manually loading configuration files as this will help explain some of the available options.

## Manually Loading Configuration Files

The author’s test device has the following DNS servers and host name:

[edit]

sean@vsrx1# **show system name-server**

1.2.3.4;

1.2.3.5;

[edit]

sean@vsrx1# **show system host-name**

host-name vsrx1;

Create two configuration files in a convenient directory on your computer. File

dns1.set should contain the following:

set system name-server 2.3.4.9 set system name-server 2.3.4.8 delete system name-server 1.2.3.4 set system host-name vsrx-dns1

File dns2.conf should contain the following (indent using spaces not tabs; Junos uses four spaces for each level of indentation, though it is not imperative that your indentation matches):

system {

host-name vsrx-dns2; name-server {

2.3.4.5;

2.3.4.6;

}

}

Note that dns2.conf contains the top-level system hierarchy. Your “text” configura- tion files should always the complete structure. This is not strictly needed for man- ual configuration but is necessary for automated configuration.

Copy these files into your home directory on your test Junos device:

mbp15:~ sean$ **scp dns\* vsrx1:.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| dns1.set | 100% | 62 | 23.1KB/s | 00:00 |
| dns2.conf | 100% | 69 | 14.1KB/s | 00:00 |

Let’s load the set file first. This is accomplished in configuration mode using the

load set command:

[edit]

sean@vsrx1# **load set dns1.set**

load complete

[edit]

sean@vsrx1# **show system name-server**

1.2.3.5;

2.3.4.9;

2.3.4.8;

[edit]

sean@vsrx1# **show system host-name**

host-name vsrx-dns1;

Notice that the two new servers on the “set” lines of the file were added while the server on the “delete” line was removed, and that the host name has been changed.

Now let’s load the text file. This is accomplished in configuration mode using the

load merge command:

[edit]

sean@vsrx1# **load merge dns2.conf**

load complete

[edit]

sean@vsrx1# **show system name-server**

1.2.3.5;

2.3.4.9;

2.3.4.8;

2.3.4.5;

2.3.4.6;

[edit]

sean@vsrx1# **show system host-name**

host-name vsrx-dns2;

Note how the new servers were added without affecting the existing servers, and that the host name has been updated.

Loading a set configuration file causes the same configuration changes as you would expect if the same commands were issued manually. If a set command up- dates a setting with a single value, like the host-name, the prior value is replaced. If a set command updates a setting that takes a list of values, like the name-server list, the new entries are added to the list. Delete commands remove settings.

Loading a text configuration file with load merge incorporates the new settings into the existing configuration. Settings with a single value are updated to the new val- ue, while settings that take a list add the new values to the list.

A text configuration file can also delete or replace settings. This requires two ad- justments to the process above. First, in the text configuration file, you need to add

system {

delete: or replace: before the setting to be deleted or replaced. Second, you need to use the load replace command instead of load merge; load replace tells Junos to hon- or any delete: or replace: tags in the text file.

Let’s do an example with the delete: tag. Modify your configuration file dns2.conf

to include the delete: line shown in bold here:

host-name vsrx-dns2; name-server {

**delete: 1.2.3.5;**

2.3.4.5;

2.3.4.6;

}

}

Copy the file to your test device, then “load replace” it into the device’s configuration:

[edit]

sean@vsrx1# **load replace dns2.conf**

load complete

[edit]

sean@vsrx1# **show system name-server**

2.3.4.9;

2.3.4.8;

2.3.4.5;

2.3.4.6;

system {

replace:

name-server {

3.4.5.6;

3.4.5.7;

Observe that server 1.2.3.5 has been removed from the name-server list. Also note that re-applying the other name servers had no effect; Junos is very good about ignoring “*changes”* that do not actually change anything.

Now let’s do an example with the replace: tag. Assume you wish to replace the en- tire name-server list, regardless of what addresses are currently in the list, with new servers. Create a new file dns3.conf and enter the following text:

}

}

Copy the file to your test device and “load replace” it into the configuration:

[edit]

sean@vsrx1# **load replace dns3.conf**

load complete

[edit]

sean@vsrx1# **show system name-server**

3.4.5.6;

3.4.5.7;

system {

Observe that the entire name-server list was replaced with the new list specified in the file.

In a text configuration file, the delete: and replace: tags can be on the same line as the setting being altered, or on the line above. Both approaches were shown in the examples above. The author typically puts the tag on the same line as a single line setting, or on the line above the start of a multiple-line setting, but this is personal preference. Another example:

replace: authentication-order [ password radius ]; host-name vsrx1;

replace:

name-server {

3.4.5.6;

3.4.5.7;

}

}

Keep in mind that the replace: tag is not needed when replacing a setting that has only one value, like host-name, as the new value replaces the existing value anyway.

The load command also has an override variation (load override filename.conf) that replaces the entire configuration of the device with the configuration file. This may be useful for new-out-of-box setup when you have a complete configuration file for the new device. We will not do an example.

Everything above applies to loading configuration files through Ansible. There are two load command options available for loading configurations manually that do not apply to Ansible, which are addressed only briefly. Both options work with both set and text configuration formats, and these options can be combined. The first option is terminal, which allows you to copy-and-paste configuration from another source, such as another device, without needing to create and upload a text file to the device you are changing:

[edit]

sean@vsrx1# **load set terminal**

[Type ^D at a new line to end input] set system name-server 2.3.4.9

set system name-server 2.3.4.8 load complete

[edit]

sean@vsrx1# **show system name-server**

3.4.5.6;

3.4.5.7;

2.3.4.9;

2.3.4.8;

The second option is relative. In all of the examples above, the configuration files were loaded at the top level of the Junos hierarchy, and the files showed the con- figuration hierarchy being changed. The relative option allows you to load chang- es into the *current* level of the hierarchy, and the configuration being loaded should be written relative to the current hierarchy level instead of the top level:

[edit]

sean@vsrx1# **edit system name-server**

[edit system name-server]

sean@vsrx1# **load set terminal relative** [Type ^D at a new line to end input] delete 2.3.4.9

delete 2.3.4.8 load complete

[edit system name-server] sean@vsrx1# **show** 3.4.5.6;

3.4.5.7;

## Installing Configuration Files with Ansible

Juniper’s Galaxy module junos\_install\_config uploads a configuration file to a Ju- nos device, loads the file, and commits the change. The configuration file can be in set or text format. The junos\_install\_config module checks the file’s extension to determine the format -- set files should use the extension .set and text files should use .conf – and does the equivalent of a load set or load merge (by default).

The author prefers text format configuration files for automation, so most exam- ples in this book will use the text format. The text format makes it easier to include annotations (comments) in the configuration files (/\* this is an annotation \*/) that will become part of the Junos configuration, and he believes the hierarchical lay- out makes it easier to understand the configuration than a series of set commands.

The author suggests putting simple configuration files in a config subdirectory within the Ansible playbook directory. Create your ~/aja/config directory, then create in the config directory a file system.conf with the following contents:

system {

name-server {

4.5.6.7;

4.5.6.8;

}

}

---

Now create file install-config.yaml in your ~/aja playbook directory using the fol- lowing Ansible playbook. This playbook assumes that we are using SSH key au- thentication and the standard NETCONF port; add arguments user, passwd, or port if needed for your environment:

* name: Install Configuration File hosts:
* all roles:
* Juniper.junos connection: local

gather\_facts: no

tasks:

* name: install configuration file onto device junos\_install\_config:

host: "{{ ansible\_host }}" file: "config/system.conf" timeout: 120

The host argument, as we saw in Chapter 6, identifies the device to be configured. The file argument identifies the configuration file to upload. The timeout argu- ment tells Ansible the maximum time it should wait for the task to complete for each device; some devices need longer than the default 30 seconds to commit their configuration, so we specify 120 seconds.

Run the playbook (with --limit if you do not wish to update all your devices):

mbp15:aja sean$ **ansible-playbook install-con**fi**g.yaml --limit=vsrx1**

PLAY [Get device uptime] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=1 changed=1 unreachable=0 failed=0

Then check the device’s configuration:

sean@vsrx1> **show con**fi**guration system name-server**

3.4.5.6;

3.4.5.7;

4.5.6.7;

4.5.6.8;

---

Observe that the DNS servers in system.conf were added to the existing name serv- ers on the device. As noted above, the junos\_install\_config module defaults to a load merge. To do a load replace instead, and replace the name-server hierarchy with what is in our configuration file, add the argument replace: yes or replace: true to the junos\_install\_config task. Let’s also add a comment that will appear in the de- vice’s commit history:

* name: Install Configuration File hosts:
* all roles:
* Juniper.junos connection: local gather\_facts: no

tasks:

* name: install configuration file onto device junos\_install\_config:

host: "{{ ansible\_host }}" file: "config/system.conf"

timeout: 120

**replace: yes**

**comment: "playbook install-con**fi**g.yaml, con**fi**guration** fi**le system.conf"**

Also add a replace: tag to the system.conf file:

system {

**replace:**

name-server {

4.5.6.7;

4.5.6.8;

}

}

Now run the playbook again (not shown) and check the results on the device:

sean@vsrx1> **show con**fi**guration system name-server**

4.5.6.7;

4.5.6.8;

sean@vsrx1> **show system commit**

1. 2017-09-02 16:52:43 UTC by sean via netconf

playbook install-config.yaml, configuration file system.conf

1. 2017-09-02 16:08:37 UTC by sean via netconf 2 2017-09-02 16:04:42 UTC by sean via cli

...

Observe that the original name servers have been removed from the configuration, and that the commit history has a comment for the latest commit.

Should you have a playbook that needs to do a load override, use the argument overwrite: yes instead of replace: yes. Also keep in mind that replace: yes is not compatible with set files.

## Generating Configuration Files – Base Settings 1.0

Installing simple configuration files like we did in the last section is useful for changes that are the same for all devices. However, when we want to make a change that contains different settings for different devices, we need to use a tem- plate that will let Ansible generate a customized configuration file for each device, filling in host-specific settings using data stored in variables.

Ansible uses a templating language called Jinja2. We explore features of Jinja2 in several chapters through the remainder of the book; in this chapter, we start with a basic template example.

Assume we are creating a playbook to add some standard configuration settings to new devices. An initial deployment team connects the devices to the network, con- figures a management IP and management user account, and enables SSH. Our playbook should set the device’s hostname and DNS servers, enable NETCONF, and add our account with SSH public key. The hostname will obviously differ for

each device. DNS servers may also be different for devices in different locations. And if we are enabling NETCONF with this playbook then presumably it is not yet enabled, so our playbook will connect over port 22.

Start by adding variables for each device’s DNS servers to your inventory file. These values will fill in the template, customizing the configuration for each device:

vsrx1 ansible\_host=192.0.2.10 dns1=5.6.7.8 dns2=5.6.7.9 bilbo ansible\_host=198.51.100.5 dns1=5.7.9.11 dns2=5.7.9.12

(In Chapter 8, we will discuss a better way to set host-specific variables, including lists of data, and see how to process list data using Jinja2 templates.)

Next create a template directory within your Ansible playbook directory (~/aja/

template) to hold Jinja2 templates, then create file base-settings.j2 within the tem- plate directory. Enter the following Jinja2 template, substituting your user account name and public SSH key. (The template contents are shown with lines numbered for easy discussion, but you should NOT enter the line numbers or the | separator character in your file):

1|system {

2| host-name {{ inventory\_hostname }}; 3| login {

4| user sean {

5| uid 2000;

6| class super-user;

7| authentication {

8| ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local) 9| }

10| }

11| }

12| replace:

13| name-server { 14| {{ dns1 }};

15| {{ dns2 }};

16| }

17| services { 18| netconf {

19| ssh;

20| }

21| }

22|}

Most of the file is Junos configuration, as this template will create a Junos configu- ration file. However, we do see one feature of the Jinja2 templating language on lines 2, 14, and 15 – double curly braces ( {{ }} ) enclose a variable name or other expression to be evaluated. Line 2 of the template reference Ansible’s inventory\_ hostname variable to get the name of the device, and lines 14 and 15 reference the dns1 and dns2 variables we put in the inventory file above.

When the template is processed, each {{ variable }} will be replaced with the con- tents of the variable before the configuration file is created. For example, line 2 of

the template, "host-name {{ inventory-hostname }};" will become "host-name vsrx1;" in the configuration file for the *vsrx1* device.

Now let’s create playbook base-settings.yaml. We will start with the tasks neces- sary to generate the configuration file from our template, then add the task to in- stall the configuration file on the device a little later in this section:

1|---

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "tmp"

12| conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf" 13|

14| tasks:

15| - debug: var=tmp\_dir 16|

17| - debug: var=conf\_file 18|

19| - name: confirm or create configs directory 20| file:

21| path: "{{ tmp\_dir }}"

22| state: directory 23|

24| - name: save device information using template 25| template:

26| src: template/base-settings.j2 27| dest: "{{ conf\_file }}"

(The lines are numbered for easy reference, but you should *not* enter the line num- bers or the | characters.)

Lines 1 – 8 are basically the same as we have seen previously, identifying the play- book and the hosts to be processed and loading the Galaxy modules.

Lines 10 – 12 define two variables that will be used later in the playbook. Instances of these variables are created for each device that is processed, so these variables can be used to create or hold device-specific information. Line 11 creates variable tmp\_dir to hold the name of the directory, tmp, where the playbook will store the generated configuration files. Line 12 creates variable conf\_file to hold the path and filename for the configuration file generated for each device, using the direc- tory name from line 11 and Ansible’s inventory\_hostname for the device. For exam- ple, the path+filename for the *vsrx1* device will be tmp/vsrx1.conf.

The debug tasks on lines 15 and 17 display the contents of variables defined above, just a way to check that things are working as we expect.

Lines 19 – 22 use the Ansible core module file to check that our configuration di- rectory exists, and create it if it does not exist. The path argument (line 21) refer- ences our tmp\_dir variable to provide the name of the configuration directory, while the state argument (line 22) says the “file” in question should be a directory.

Lines 24 – 27 use the Ansible core module template to process our Jinja2 template. The src argument (line 26) tells the template module where to find our template file, while the dst argument (line 27) references our conf\_file variable to tell the template module where to store the result of processing our template.

The template module is an Ansible core module, not a Juniper module; it processes templates written in the Jinja2 template language, not Junos configuration files. Neither Ansible nor Jinja2 know, or need to know, anything about Junos in order to process the template we wrote and generate a Junos configuration file. We, the authors of the template, provided the Junos knowledge—processing the template just fills in the blanks where the template referenced device-specific variables.

Let’s run the playbook:

mbp15:aja sean$ **ansible-playbook base-settings.yaml**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"tmp\_dir": "tmp"

}

ok: [bilbo] => {

"tmp\_dir": "tmp"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"conf\_file": "tmp/vsrx1.conf"

}

ok: [bilbo] => {

"conf\_file": "tmp/bilbo.conf"

}

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

ok: [bilbo]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=4 changed=1 unreachable=0 failed=0

vsrx1 : ok=4 changed=2 unreachable=0 failed=0

Notice in the output of the second debug task that each device’s conf\_file variable contains a unique name. Notice the confirm or create configs directory task re- turned a changed state for the first device to be processed because it had to create the tmp directory, while the task returned ok (no change, the directory existed) when it processed the second device.

Now display the configuration files generated by the playbook, and note how the

host-name and name-server information is specific for each device:

mbp15:aja sean$ **cat tmp/bilbo.conf**

system {

host-name bilbo; login {

user sean {

uid 2000;

class super-user; authentication {

ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local)

}

}

}

replace:

name-server {

5.7.9.11;

5.7.9.12;

}

services {

netconf {

ssh;

}

}

}

mbp15:aja sean$ **cat tmp/vsrx1.conf**

system {

host-name vsrx1; login {

user sean {

uid 2000;

class super-user; authentication {

ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local)

}

}

}

replace:

name-server {

5.6.7.8;

5.6.7.9;

}

services {

netconf {

ssh;

}

}

}

This is a good opportunity to check the configuration files for problems, like miss- ing braces or semicolons, or misspelled words, that might cause Junos to reject the configurations. If you find any issues, check the base-settings.j2 template.

## Installing the Generated Configuration – Base Settings 1.1

Let’s update the playbook to push the configuration files to the devices (new lines are in boldface, line numbers added for discussion):

1|---

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "tmp"

12| conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf" 13|

**14| vars\_prompt:**

**15| - name: username**

**16| prompt: Junos Username 17| private: no**

**18|**

**19| - name: password**

**20| prompt: Junos Password 21| private: yes**

22|

23| tasks:

24| - name: confirm or create configs directory 25| file:

26| path: "{{ tmp\_dir }}"

27| state: directory 28|

29| - name: save device information using template 30| template:

31| src: template/base-settings.j2 32| dest: "{{ conf\_file }}"

33|

**34| - name: install generated con**fi**guration** fi**le onto device 35| junos\_install\_con**fi**g:**

**36| host: "{{ ansible\_host }}"**

**37| user: "{{ username }}"**

**38| passwd: "{{ password }}"**

**39| port: 22**

**40|** fi**le: "{{ conf\_**fi**le }}"**

**41| timeout: 120**

**42| replace: yes**

**43| comment: "playbook base-settings.yaml"**

Lines 14 – 21 add prompts for device username and password. Because this ex- ample assumes the devices do not yet have SSH public keys in their configurations, the playbook needs the username and password that it will use to authenticate to the devices.

The two debug tasks have been removed; having verified above that our variables were working, those tasks were no longer needed.

Lines 34 – 43 are the task to push the configuration files to the devices. Building on the example we saw earlier in this chapter, this task adds user and passwd argu- ments to pass the username and password to the junos\_install\_config module, and the port argument so the modules will use the standard SSH port.

Now run the updated playbook:

mbp15:aja sean$ **ansible-playbook base-settings.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password: **<enter password>**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=3 changed=1 unreachable=0 failed=0

Notice that the task save device information using template returned a status of ok, but it returned changed the last time we ran the playbook. Before the previous play- book run, the configuration files did not exist, so creating the file was a change.

This time the configuration file already existed and there was no change in the file generated during this playbook run relative to the last playbook run. (Because we had not changed the template or any of the variables, the resulting configuration file was unchanged.) If you wish, try making a change to one of the DNS variables in inventory and run the playbook again; you should see a changed status for the device(s) whose DNS data you changed.

## Displaying Changes

Most Ansible modules that make changes can show what they are changing, you simply need to run the playbook with the --diff argument.

In the last section of this chapter, the author ran the playbook only against his de- vice *vsrx1*. Let’s run the playbook again, this time for both *vsrx1* and *bilbo*, and use --diff to show what changes for *bilbo* (there should be no changes for *vsrx1*):

mbp15:aja sean$ **ansible-playbook base-settings.yaml --diff**

Junos Username: **sean**

Junos Password: **<enter password>**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

[edit system]

+ name-server {

+ 5.7.9.11;

+ 5.7.9.12;

+ }

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=3 changed=1 unreachable=0 failed=0

vsrx1 : ok=3 changed=0 unreachable=0 failed=0

Observe that Ansible shows not only that *bilbo’s* configuration changed, but it shows the exact change made:

[edit system]

+ name-server {

+ 5.7.9.11;

+ 5.7.9.12;

+ }

Of course, we did not change any settings in the template, or delete the tmp direc- tory, or alter anything else that would require the playbook make other changes. Let’s force more changes by deleting the tmp directory where the generated configu- ration files are stored:

mbp15:aja sean$ **rm -r tmp**

Now run the playbook with --diff again:

mbp15:aja sean$ **ansible-playbook base-settings.yaml --diff**

Junos Username: sean Junos Password:

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--- before

+++ after

@@ -1,4 +1,4 @@

{

"path": "tmp",

* "state": "absent"

+ "state": "directory"

}

changed: [vsrx1]

--- before

+++ after

@@ -1,4 +1,4 @@

{

"path": "tmp",

* "state": "absent"

+ "state": "directory"

}

changed: [bilbo]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--- before

+++ after: /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmphJ6eBw/base-settings.j2 @@ -0,0 +1,22 @@

+system {

+ host-name vsrx1;

+ login {

+ user sean {

+ uid 2000;

+ class super-user;

+ authentication {

+ ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local)

+ }

+ }

+ }

+ replace:

+ name-server {

+ 5.6.7.8;

+ 5.6.7.9;

+ }

+ services {

+ netconf {

+ ssh;

+ }

+ }

+}

changed: [vsrx1]

--- before

+++ after: /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmpWyXTlb/base-settings.j2 @@ -0,0 +1,22 @@

+system {

+ host-name bilbo;

+ login {

+ user sean {

+ uid 2000;

+ class super-user;

+ authentication {

+ ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local)

+ }

+ }

+ }

+ replace:

|  |  |
| --- | --- |
| + | name-server { |
| + | 5.7.9.11; |
| + | 5.7.9.12; |
| + | } |
| + | services { |
| + | netconf { |
| + | ssh; |
| + | } |
| + | } |
| +} |  |

changed: [bilbo]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=3 | changed=2 | unreachable=0 | failed=0 |
| vsrx1 | : ok=3 | changed=2 | unreachable=0 | failed=0 |

This time there were several changes – the playbook needed to create the tmp direc- tory, and both of the generated configuration files were new and thus a change.

## Cleaning Up Temporary Files – Base Settings 1.2

Let’s add a task to our playbook to have it delete the configuration files. In other words, we’ll have the playbook clean up its temporary files. Add the following to the end of the playbook:

44|

45| - name: delete generated configuration file 46| file:

47| path: "{{ conf\_file }}"

48| state: absent

This is similar to the task that created our configuration directory, except that

state: absent tells the file module to delete the file if it exists.

Run the playbook again and confirm that the configuration files created by the playbook are missing from the tmp directory after the playbook completes:

mbp15:aja sean$ **ansible-playbook base-settings.yaml**

Junos Username: **sean**

Junos Password: **<enter password>**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=4 | changed=1 | unreachable=0 | failed=0 |
| vsrx1 | : ok=4 | changed=1 | unreachable=0 | failed=0 |
| mbp15:aja sean$ **ls tmp/**  mbp15:aja sean$ |  |  |  |  |

## Deleting Settings That Might Not Be Present

Let’s update the base-settings.j2 template to delete the FTP and Telnet system ser- vices. These protocols are not encrypted and are thus insecure and their use should be avoided, but they might be enabled by default or may have been turned on dur- ing initial setup. Modify the services section of the template as follows (only the services section is shown, new lines are boldfaced):

17| services {

18| **delete: ftp;**

19| netconf {

20| ssh;

21| }

22| **delete: telnet;**

23| }

Run the playbook. You should get an error during the install generated configura- tion task as shown below. (If you did not get an error, run the playbook again, you will probably get the error the second time):

mbp15:aja sean$ **ansible-playbook base-settings.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password:

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "Unable to load config: ConfigLoadError(severity: warning, bad\_element: None, message: warning: statement not found\nwarning: statement not found)"}

to retry, use: --limit @/Users/sean/aja/base-settings.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=2 changed=2 unreachable=0 failed=1

“Unable to load config…*warning: statement not found*.” When you try to delete a configuration statement that does not exist, Junos issues a warning. The junos\_in- stall\_config module sees that warning and issues an error, causing the playbook to stop processing the device in question.

The author’s *vsrx1* device has the following system services:

sean@vsrx1> **show confifiguration system services**

ftp; ssh;

netconf {

ssh;

}

web-management { http {

interface fxp0.0;

}

}

...

Notice that Telnet is not enabled. Our modified template tries to delete the telnet service; this is the “statement [that was] not found” and caused our error. We dis- cuss two approaches for resolving this error.

Option A – ignore\_warning

The first, and preferred, option is to instruct the junos\_install\_config module to ignore warnings, using the ignore\_warning argument. This argument was added in version 1.4.3 of Juniper’s Galaxy modules; if you need to use an older version of the modules you should look at Option B below.

Add the ignore\_warning: yes option to the “install generated configuration file onto device” task as shown:

34| - name: install generated configuration file onto device 35| junos\_install\_config:

36| host: "{{ ansible\_host }}"

37| user: "{{ username }}"

38| passwd: "{{ password }}"

39| port: 22

40| file: "{{ conf\_file }}"

**41| ignore\_warning: yes**

42| timeout: 120

43| replace: yes

44| comment: "playbook base-settings.yaml"

...

Run the playbook...

mbp15:aja sean$ **ansible-playbook base-settings.yaml**

Junos Username: **sean**

Junos Password: **<enter password>**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=4 changed=2 unreachable=0 failed=0

vsrx1 : ok=4 changed=2 unreachable=0 failed=0

Much better!

This works for settings hierarchies as well as single-line settings. Update the bas- esettings.j2 template to delete the web-management service:

...

17| services {

18| delete: ftp;

19| netconf {

20| ssh;

21| }

22| delete: telnet;

**23| delete: web-management;**

24| }

25|}

Run the playbook again, and then check your test device(s) and confirm that the FTP, Telnet, and web management services have all been deleted:

sean@vsrx1> show configuration | compare rollback 2 [edit system services]

* ftp;
* web-management {
* http {
* interface fxp0.0;

- }

- }

Option B – Add and Delete

Another option for working around the problem of deleting settings that might not be present is to make (a minimalist version of) the setting before you delete it. Update the services portion of the template:

...

17| services {

18| **ftp;**

19| delete: ftp;

20| netconf {

21| ssh;

22| }

23| **telnet;**

24| delete: telnet;

25| }

26|}

The template now makes minimalist ftp and telnet service settings before the re- spective deletions of those settings.

Now run the playbook…

mbp15:aja sean$ **ansible-playbook base-settings.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password: **<enter password>**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=4 changed=3 unreachable=0 failed=0

Much better!

Sometimes the setting you wish to delete requires more than a one-line setting. The web-management service is an example; you need to tell it http and/or https or Junos will refuse to accept the setting. You can confirm this at the Junos CLI:

sean@vsrx1> configure Entering configuration mode

[edit]

sean@vsrx1# set system services web-management

^

missing argument.

[edit]

sean@vsrx1# set system services web-management http

[edit] sean@vsrx1#

1|system {

Update the services section of the template as follows (see lines 25 – 28; full tem- plate shown):

2| host-name {{ inventory\_hostname }}; 3| login {

4| user sean {

5| uid 2000;

6| class super-user;

7| authentication {

8| ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local) 9| }

10| }

11| }

12| replace:

13| name-server { 14| {{ dns1 }};

15| {{ dns2 }};

16| }

17| services {

18| ftp;

19| delete: ftp;

20| netconf {

21| ssh;

22| }

23| telnet;

24| delete: telnet;

**25| web-management { 26| http;**

**27| }**

**28| delete: web-management;**

29| }

30|}

Run the playbook…

mbp15:aja sean$ **ansible-playbook base-settings.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password:

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=4 changed=3 unreachable=0 failed=0

Check your test device(s) and confirm that the FTP, Telnet, and web management services have all been deleted:

sean@vsrx1> **show configuration | compare rollback 2**

[edit system services]

* ftp;
* web-management {
* http {
* interface fxp0.0;

- }

- }

## Commit Confirmed – Base Settings 1.3

One of the great features of Junos is *commit confirmed* – the ability to tentatively commit a configuration change, asking Junos to automatically roll back the change if the network engineer does not issue a second commit to confirm the change. Should the engineer lose contact with the device after the first commit – if, for example, the change being committed disabled a needed routing protocol – the device will automatically revert to its prior state and (hopefully) restore service.

Automation should mitigate the need for commit confirmed because automation should reduce human error. However, if the source data for the configuration tem- plates is created by humans, there is still a potential for human error.

Let’s add commit confirmed to our playbook.

We add the argument confirm: 10 to our install generated configuration task; this argument tells the junos\_install\_config module to use “commit confirmed 10” in- stead of just “commit” when committing the change.

And we need to add a new task that calls the junos\_commit module, which performs a commit without making any change to the configuration. Each time Ansible calls one of Juniper’s Galaxy modules, the module needs to establish a new NETCONF connection to the device. If the junos\_commit step completes—if the module is able to establish the connection to the device—it provides some assurance that the change did not break our ability to manage the device.

We add the new task as a *handler*, a task that is executed only when a previous task reports a change and *notifies* the handler to run. If the previous task did not report a change, it will not notify the handler to run.

We will add a notify term to our install generated configuration task, so it will no- tify the handler to run when the device’s configuration changed. However, when the device’s configuration did not change – when the configuration file installed on the device did not result in a configuration change – there is no point in confirming the “non-change” and thus the handler will not be notified.

Update the base-settings.yaml playbook as shown (new or edited lines in boldface, only the end of the file shown):

34| - name: install generated configuration file onto device 35| junos\_install\_config:

36| host: "{{ ansible\_host }}"

37| user: "{{ username }}"

38| passwd: "{{ password }}"

39| port: 22

40| file: "{{ conf\_file }}"

41| timeout: 120

42| replace: yes

**43| con**fi**rm: 10**

**44| comment: "playbook base-settings.yaml, commit con**fi**rmed" 45| notify: con**fi**rm commit**

46|

47| - name: delete generated configuration file 48| file:

49| path: "{{ conf\_file }}"

50| state: absent 51|

**52| handlers:**

**53| - name: con**fi**rm commit 54| junos\_commit:**

**55| host: "{{ ansible\_host }}"**

**56| user: "{{ username }}"**

**57| passwd: "{{ password }}"**

**58| port: 22**

**59| timeout: 120**

**60| comment: "playbook base-settings.yaml, con**fi**rming previous commit"**

Line 43 makes our existing install generated configuration task use “commit con- firmed 10.”

Line 45 tells our install generated configuration task to notify the handler whose

name is confirm commit to run when the device’s configuration changed.

Line 52 creates a new playbook section called handlers, which can contain one or more handler “tasks.”

Lines 53 – 60 define our new handler for calling the junos\_commit module. The ar- guments should be familiar from our earlier discussions of junos\_install\_config.

In order to see the handler work, we need our device’s configuration to be different from what our template will create. Let’s delete the name servers from the device:

sean@vsrx1> configure Entering configuration mode

[edit]

sean@vsrx1# delete system name-server

[edit]

sean@vsrx1# show | compare [edit system]

* name-server {

- 5.6.7.8;

- 5.6.7.9;

- }

[edit]

sean@vsrx1# commit and-quit commit complete

Exiting configuration mode

Now run the playbook…

mbp15:aja sean$ **ansible-playbook base-settings.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password:

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

RUNNING HANDLER [confirm commit] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=5 changed=4 unreachable=0 failed=0

Observe that the install generated configuration task reported a changed status, and that the handler ran.

Check the commit history on the device and observe the last two commits. You can see that commit #1 was “commit confirmed, rollback in 10 mins” – that was the install task:

sean@vsrx1> show system commit

1. 2017-09-03 05:03:55 UTC by sean via netconf

playbook base-settings.yaml, confirming previous commit

1. 2017-09-03 05:03:45 UTC by sean via netconf commit confirmed, rollback in 10mins playbook base-settings.yaml, commit confirmed

2 2017-09-03 04:51:42 UTC by sean via cli

...

Now run the playbook again:

mbp15:aja sean$ **ansible-playbook base-settings.yaml --limit=vsrx1**

Junos Username: **sean**

Junos Password:

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=4 changed=2 unreachable=0 failed=0

This time the install generated configuration task reported an ok status (because the device’s configuration was not modified) and thus did not notify the handler to run.

You may need to set a longer confirm time in your playbook than you would use manually on a single device. Ansible completes one task for all devices before mov- ing on to the next task, which means all devices must complete their configuration installation before Ansible will start the handlers that confirm the commits. The confirm time must be long enough for the configuration installation task to com- plete. If you are running the playbook against only a few devices this may require only a few minutes, but if you run the playbook against 100 devices it will need more time.

Of course, the problem with long confirm times is that, in the event there is a prob- lem, it takes longer for the devices to roll back their configurations. Consider using

--limit arguments that will cause the playbook to run against a modest number of devices at a time, allowing a shorter commit time in the playbook, even though you may need to re-run the playbook repeatedly with different --limit arguments to process all your devices.

## Loading Configuration Via Console

The initial configuration of Junos devices is normally done via the device’s serial port, or console port. The scenario described near the beginning of the “Generat- ing configuration files” section of this chapter, which informed our last example, assumed that someone already did enough initial setup via console that we could reach the device over the network, but what if that assumption is not valid? What if we want our automation to handle that initial setup via console?

Juniper’s Galaxy modules provide two approaches for accessing devices via con- sole. The older approach, available with only a few of the modules, uses the con- sole argument. The newer approach, available on all of Juniper’s Galaxy modules, uses the mode argument.

In this chapter, we discuss the newer approach – it is easier to use and is supported on all of the Juniper modules. The mode option currently supports direct serial con- nections, and terminal servers with no authentication via Telnet.

Assume we want to complete an initial configuration on an EX switch via a direct serial connection. The initial configuration needs to include the hostname, the root password, another admin account and password, and enable SSH and NET- CONF-over-SSH. The initial configuration should also configure a VLAN called *aja*, add a few ports to the VLAN using an interface-range called *aja*, and config- ure a Layer 3 interface on the VLAN. The following Jinja2 template, template/ini- tial-ex-vlan.j2, (line numbers added) fulfills these requirements:

1|#jinja2: lstrip\_blocks: True 2|system {

3| host-name {{ inventory\_hostname }}; 4| root-authentication {

5| encrypted-password "$5$AM...r1VfikrC"; 6| }

7| login {

8| user sean {

9| uid 2000;

10| class super-user;

11| authentication {

12| encrypted-password "$5$gGY...knZnU4X5"; 13| }

14| }

15| }

16| services {

17| ssh;

18| netconf {

19| ssh;

20| }

21| }

22|}

23|interfaces {

24| interface-range aja { 25| member ge-0/0/8;

26| member ge-0/0/9;

27| member ge-0/0/10;

28| member ge-0/0/11;

29| unit 0 {

30| family ethernet-switching { 31| port-mode access;

32| vlan {

33| members aja;

34| }

35| }

36| }

37| }

38| vlan {

39| unit 0 {

40| family inet {

41| address {{ ansible\_host }}/{{ netmask }}; 42| }

43| }

44| }

45|}

46|vlans { 47| aja {

48| l3-interface vlan.0; 49| }

50|}

1|---

Use the correct encrypted password for your root and user accounts on lines 5 and 12, and your username on line 8. (In a later chapter, we will discuss securely stor- ing credentials and other sensitive information, but for now you can just include the encrypted passwords directly in the template.)

The author’s switch is an ex2200c, which uses the legacy VLAN command set; if your test switch is a newer EX with the ELS command set, replace the keyword vlan with keyword irb on lines 38 and 48. You can also substitute different inter- faces on lines 25 – 28, if needed for your switch or environment.

There are three variable references in the template. Line 3 includes the switch’s hostname using Ansible’s inventory\_hostname variable. Line 41 includes the switch’s IP address using Ansible’s ansible\_host variable. (If you have avoided as- signing an IP address to this variable because name resolution on the inventory\_ hostname works for your environment, please take a moment and add this variable to your inventory file.)

Assigning an IP address to an interface or VLAN requires the subnet mask, so line 41 also references a variable netmask. This variable will be declared in the playbook for this example. (After we discuss host data files and group data files in Chapter 8, you may wish to relocate this variable’s declaration to a data file.)

We will discuss two variations of the playbook, one for direct serial connection from your computer to the switch, and one for console access through a terminal server using Telnet. Let’s start with the serial playbook, initial-setup-con.yaml:

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "tmp"

12| conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf" 13| netmask: "24"

14| username: root 15|

16| # vars\_prompt:

17| # - name: username

18| # prompt: Junos Username 19| # private: no

20| #

21| # - name: password

22| # prompt: Junos Password 23| # private: yes

24|

25| tasks:

26| - name: confirm or create configs directory 27| file:

28| path: "{{ tmp\_dir }}"

29| state: directory 30|

31| - name: save device information using template 32| template:

33| src: template/initial-ex-vlan.j2 34| dest: "{{ conf\_file }}"

35|

36| - name: install generated configuration file onto device 37| junos\_install\_config:

38| host: "{{ inventory\_hostname }}" 39| file: "{{ conf\_file }}"

40| user: "{{ username }}"

41| # passwd: "{{ password }}"

42| timeout: 120

43| overwrite: yes

44| mode: serial

45| port: "/dev/cu.usbserial-AH02PIG9"

46| comment: "playbook initial-setup-con.yaml"

Lines 1 – 8 are familiar from earlier playbooks.

Lines 10 – 14 declare some variables. We saw variables tmp\_dir and conf\_file in our last example. Variable netmask we mentioned above.

Variable username (line 14) is set to root, under the assumption that we are configuring a new-out-of-box device for which the first logon is done as root with no password. The author will use his *bilbo* switch, but will use the request system zeroize command to reset the device to factory default settings before running the playbook. If you do not wish to reset one of your test devices, and thus require a different username and password, comment out line 14 (username) and uncomment lines 16 – 23 (vars\_prompt) and 41 (passwd); this will cause the playbook to prompt for username and password and provide both when accessing the device.

Lines 26 – 34 are the same as the similar lines from the base-settings playbook, ensur- ing the tmp directory exists and generating the configuration file from the template; the only change is the template name on line 33.

Lines 36 – 46 install the generated configuration on the devices, similar to what we saw in the base-settings playbook, but there are several changes related to console access.

The host argument (line 38) is required for the junos\_install\_config module, but is not really relevant for serial console access (recall that host normally specifies the target for the NETCONF-over-SSH connection). The playbook sets it to inventory\_hostname so it has a useful value should there be a problem that generates an error message, as error messages often include the value of host.

The argument overwrite: yes (line 43) informs the junos\_install\_config module that it should use the equivalent of Junos’ load override command when loading the configuration file. In other words, completely replace the device’s existing configu- ration with what is being loaded. This is normally a good choice for an initial setup playbook and template as there are often default settings, such as VLAN settings or interface settings, that might interfere with the new settings we want to make.

The argument mode: serial (line 44) informs the junos\_install\_config module that it should connect to the device using a local serial port, not a normal NETCONF- over-SSH session.

The port argument (line 45) indicates the serial port that should be used for the connection. (Note the meaning of port changes in the context of the mode: serial argument; normally port specifies the TCP port for the NETCONF connection.)

*You need to adjust the value of* port *to correspond with your computer’s serial port or USB serial adapter*. As most computers no longer include traditional serial ports, you probably have a USB-to-serial adapter. On MacOS, try the command

ls /dev/cu\* and see if your serial adapter is listed:

mbp15:~ sean$ **ls /dev/cu\***

/dev/cu.Bluetooth-Incoming-Port

/dev/cu.usbserial-AH02PIG9

On Linux systems, try the command ls /dev/ttyUSB\*: sean@gandalf:~$ **ls /dev/ttyUSB\***

/dev/ttyUSB0

If your USB-to-serial adapter is not listed on MacOS or Linux, disconnect the adapter, run ls /dev/tty\*, then reconnect the adapter and re-run ls /dev/tty\*; see if any new listings appear after re-connecting the adapter. If no new TTY devices ap- pear, your adapter may not be supported by MacOS or Linux, or you may need to install drivers to add support.

Let’s run the playbook on the reset network device:

mbp15:aja sean$ **ansible-playbook initial-setup-con.yaml --limit=bilbo**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=3 changed=1 unreachable=0 failed=0

1|---

Keep in mind that 9600bps serial connections are a lot slower than typical net- work connections; expect the configuration installation step to take a lot longer than you saw with the base-settings playbook.

Now let’s discuss initial-setup-ts.yaml, a variation of the above playbook that will work with Telnet-based terminal server connections:

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "tmp"

12| conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf" 13| netmask: "24"

14| username: root

15| terminal\_server: 198.51.100.100

16| term\_serv\_port: 7001 17|

18| # vars\_prompt:

19| # - name: username

20| # prompt: Junos Username 21| # private: no

22| #

23| # - name: password

24| # prompt: Junos Password 25| # private: yes

26|

27| tasks:

28| - name: confirm or create configs directory 29| file:

30| path: "{{ tmp\_dir }}"

31| state: directory 32|

33| - name: save device information using template 34| template:

35| src: template/initial-ex-vlan.j2 36| dest: "{{ conf\_file }}"

37|

38| - name: install generated configuration file onto device 39| junos\_install\_config:

40| host: "{{ terminal\_server }}" 41| file: "{{ conf\_file }}"

42| user: "{{ username }}"

43| # passwd: "{{ password }}" 44| timeout: 120

45| overwrite: yes

46| mode: telnet

47| port: "{{ term\_serv\_port }}"

48| comment: "playbook initial-setup-ts.yaml"

Most of the initial-setup-ts.yaml playbook is the same as the initial-setup-con. yaml playbook, so we will discuss just the changes. There are two new variables defined, and some changes to the arguments to the junos\_install\_config module.

Lines 15 and 16 define the new variables. The terminal\_server variable is set to the hostname or IP address for the terminal server. The term\_serv\_port variable is set to the TCP port for the Telnet session. *Adjust these values as needed for your termi- nal server*. Terminal servers typically either assign a unique IP to each of their serial ports, or they use a single IP but assign a unique TCP port number to each serial port.

TIP The terminal\_server and/or term\_serv\_port values may be device-specific or site-specific, so after reading Chapter 8 you may wish to move these variable definitions to host or group data files.

The mode: telnet argument (line 46) tells junos\_install\_config to use Telnet to con- nect to the device, not the default SSH.

The host argument (line 40) provides the terminal server’s IP or hostname to the

junos\_install\_config module, so the playbook sets host to the value of the new ter- minal\_server variable.

The port argument (line 47) provides the correct Telnet port number to the junos\_ install\_config module, so the playbook sets port to the value of the new term\_serv\_ port variable.

Running this playbook looks very similar to the console version:

mbp15:aja sean$ **ansible-playbook initial-setup-ts.yaml --limit=bilbo**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*

ok: [bilbo]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*

ok: [bilbo]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*

bilbo : ok=3 changed=1 unreachable=0 failed=0

## Debugging Templates

As you develop more complex templates, debugging them can become challenging. This section presents a few tips for finding and fixing template problems.

In this chapter, we initially had our playbook process the template and stop, so that we could manually view the results; only after the results looked right did we add the tasks that tried to use the results from the template (install the generated configuration file). This is a process the author has used many times. In this chap- ter, we used a partially-completed playbook to make the manual check, but you can accomplish something similar using a completed playbook by temporarily commenting out all the tasks after the template generation and any other unneces- sary items, like prompts for device authentication credentials; for example:

---

* name: Generate and Install Configuration File hosts:
* all roles:
* Juniper.junos connection: local gather\_facts: no

vars:

tmp\_dir: "tmp"

conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf"

# vars\_prompt:

# - name: username

# prompt: Junos Username # private: no

#

# - name: password

# prompt: Junos Password # private: yes

tasks:

* name: confirm or create configs directory file:

path: "{{ tmp\_dir }}" state: directory

* name: save device information using template template:

src: template/base-settings.j2 dest: "{{ conf\_file }}"

# - name: install generated configuration file onto device # junos\_install\_config:

# host: "{{ ansible\_host }}" # user: "{{ username }}"

# passwd: "{{ password }}" # port: 22

# file: "{{ conf\_file }}" # timeout: 120

# replace: yes

# confirm: 10

# comment: "playbook base-settings.yaml, commit confirmed" # notify: confirm commit

#

# - name: delete generated configuration file # file:

# path: "{{ conf\_file }}" # state: absent

#

# handlers:

# - name: confirm commit # junos\_commit:

# host: "{{ ansible\_host }}" # user: "{{ username }}"

# passwd: "{{ password }}" # port: 22

# timeout: 120

# comment: "playbook base-settings.yaml, confirming previous commit"

One example of an error that you might identify during such a manual check is a variable reference that was not replaced correctly. Assume the generated configu- ration file contains the following:

...

replace:

name-server {

{ dns1 }};

5.6.7.9;

...

}

replace:

name-server {

**{ dns1 }};**

{{ dns2 }};

Clearly { dns1 }} is not the expected output. An inspection of the template shows that the problem was caused by a missing left brace ("{"):

}

...

If the playbook returns an error from the template processing task, there is likely a syntax error in the template. Read the error message carefully; they usually tell you what you should be looking for. For example:

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "AnsibleUndefinedVariable: 'inventoryhostname' is undefined"}

...

The error “AnsibleUndefinedVariable: ‘inventoryhostname’ is undefined” tells you that somewhere in the template (unfortunately, the error does not provide a line

...

number) there appears a variable reference {{ inventoryhostname }} – an attempt to read variable inventoryhostname – but that variable inventoryhostname was not previ- ously defined. This might be a typographical error in the variable name, or it might be that you need to define the variable in the inventory file or other host data file (discussed in a later chapter). In this case the problem is a typo: the variable name should be inventory\_hostname, with an underscore character in the name.

Another syntax message example:

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "AnsibleError: template error while templating string: unexpected '}'. String: system {\n host-name {{ inventory\_hostname }};\n login {\n user sean {\n uid 2000;\n class super-user;\n authentication

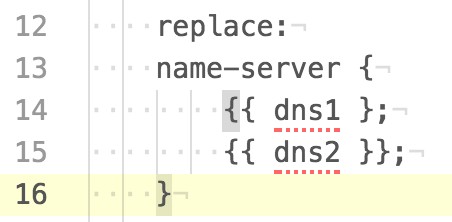
{\n ssh-rsa \"ssh-rsa AAAA...JzS8b sean@mbp15.local\";\n }\n }\n }\n replace:\n name-server {\n {{ dns1 };\n {{ dns2 }};\n }\n services {\n ftp;\n

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| delete: ftp;\n  web-management {\n | netconf {\n  http;\n | ssh;\n  }\n | }\n telnet;\n  delete: web-management;\n | delete: telnet;\n  }\n}\n"} |
| ... |  |  |  |  |

The significant part of the error message is “AnsibleError: template error while templating string: unexpected ‘}’.” The rest is the text of the template shown as a single string, which looks rather ugly but can be ignored. Unfortunately, while An- sible’s template module realized there was an “unexpected '}'” character, it does not know exactly where in the file the problem lies.

Most programmer’s text editors have a feature for identifying matching parenthe- ses, brackets, and braces. You will need to check your editor’s documentation to find out how to enable or access this feature. Some highlight matches any time your cursor is at a parenthesis/bracket/brace, some require you to press Ctrl+M or other shortcut to find the match, and some use both approaches. By looking for incorrect “matches” you can quickly zoom in on the extra or missing character.

The following screen capture from the author’s text editor shows a mismatch; ob- serve that the highlighted braces (left brace "{" on line 14 and right brace "}" on line 16) are clearly not a proper matched set, so the problem is between those two locations in the file:



In this example, the problem is a *missing* "}" at the end of line 14 of the template.

...

Mistakes in the template can also cause Junos to reject the configuration file. For example:

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "Unable to load config: ConfigLoadError(severity: error, bad\_element: name-servers, message: error: syntax error\nerror: error recovery ignores input until this point)"}

...

12| replace:

Here the error message tells us we have a "bad\_element" -- “Unable to load config: ConfigLoadError(severity: error, *bad\_element: name-servers*, message: error: syn- tax error...” Searching the template file for the incorrect element *name-servers* finds the following:

13| name-servers { 14| {{ dns1 }};

The correct name for Junos’ DNS server list is name-server, not name-servers.

Sometimes a “bad element” problem is easier to find in the configuration file than in the template. For example:

...

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "Unable to load config: ConfigLoadError(severity: error, bad\_element: 5.6.7.9, message: error: syntax error\nerror: could not resolve name: services\nerror: error recovery ignores input until this point\nerror: syntax error)"}

...

Here the error says “bad\_element: 5.6.7.9” but the text “5.6.7.9” is nowhere in the template file. However, if we look at the generated configuration file we find:

name-server { 5.6.7.8

5.6.7.9;

}

...

Notice that the semicolon is missing from the first DNS server IP. Although we found the problem more easily in the configuration file, remember you must fix this in the template by adding the missing semicolon.

For our last example, let’s quickly revisit the error message associated with a prob- lem that we discussed earlier in this chapter, trying to delete a non-existent con- figuration element:

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "Unable to load config: ConfigLoadError(severity: warning, bad\_element: None, message: warning: statement not found)"}

...

Notice that the error message says there is no bad element (“bad\_element: *None*”) but that there was a “warning: statement not found.” Check the template for de- lete: tags and ensure that the corresponding configuration element will always be present before the attempt to delete it.

If the above troubleshooting steps for a Junos error fail to identify the problem, take the generated configuration file and manually load it on the Junos device. Use the correct load variation (load merge, load replace or load set) so it mimics the play- book’s action. Junos usually does a good job of identifying the problem, but some of the detail is lost when the Junos warnings or errors are passed back through the automation tools.

## References

Jinja2 information: <http://jinja.pocoo.org/docs/dev/>

<http://docs.ansible.com/ansible/latest/playbooks_templating.html> Junos\_install\_config module:

<http://junos-ansible-modules.readthedocs.io/en/1.4.2/junos_install_config.html>

Junos\_commit module:

<http://junos-ansible-modules.readthedocs.io/en/1.4.2/junos_commit.html>

Ansible handlers: [http://docs.ansible.com/ansible/latest/playbooks\_intro.](http://docs.ansible.com/ansible/latest/playbooks_intro) html#handlers-running-operations-on-change

# Chapter 8

Data Files and Inventory Groups

In previous chapters, we created a simple inventory file and added a few variables to that file. While functional for simple environments, this approach does not scale well to large numbers of devices or variables, or to complex variables such as lists. This chapter explores Ansible’s architecture for storing device inventory, including groups, and for storing information about the managed devices and groups.

## Variables

While executing a playbook, Ansible maintains a number of variables that can be referenced in the playbook or in Jinja templates. We have already seen and used a few variables, including Ansible’s pre-defined inventory\_hostname and ansible\_host, and the dns1 and dns2 variables we defined in our inventory file.

When defining a variable, keep in mind that a variable name should start with a letter and can contain letters, numerals, and the underscore ("\_") character. Valid variable names include my\_data and Results1; invalid variable names include 2days (starts with a numeral) and task-results (contains a hyphen). Variable names are case sensitive: test1 and Test1 are different variables.

There are several sources for variables, including (but not limited to) the following:

* Ansible’s pre-defined or "magic" variables, such as the inventory\_hostname and

ansible\_host variables we have already seen. Others include hostvars and group\_ names, which we will discuss later in this chapter.

* Facts discovered from the hosts being managed by the playbook.

1|---

* + Variables set in the inventory file, as we have shown in prior chapters, or set in the host and group variable files that are discussed later in this chapter.
  + Registered variables, set using the register option to capture the results of a play, as we saw in the uptime.yaml playbook.
  + Variables defined in a playbook, using either the vars: or vars\_prompt: sections of a play; we used both in the base-settings.yaml playbook. Playbooks can also use the set\_fact module to set variables; we use set\_fact in this chapter.
  + "Extra" variables provided at the command line when launching a playbook using the –e or --extra-vars arguments; we will see a brief example shortly and use them in a future chapter.

The *scope* of a variable is the region of the playbook during which a variable is valid, and the hosts for which the variable is valid. Variables defined by different sources have different scopes. This can be difficult to explain in the abstract, so let’s create a few small playbooks to illustrate variable scope.

Call this playbook show-vars-1.yaml:

2|- name: Show variables 1, first play 3| hosts:

4| - all

5| connection: local 6| gather\_facts: no 7|

8| vars:

9| test1: "test all lower-case" 10| Test1: "Test first capital"

11| name\_plus\_host: "{{ inventory\_hostname}} :: {{ ansible\_host }}" 12|

13| tasks:

14| - debug: var=test1 15| - debug: var=Test1

16| - debug: var=name\_plus\_host 17|

18|

19|- name: Show variables 1, second play 20| hosts:

21| - all

22| connection: local 23| gather\_facts: no 24|

25| tasks:

26| - debug: var=test1 27| - debug: var=Test1

28| - debug: var=name\_plus\_host

Run the playbook:

mbp15:aja sean$ **ansible-playbook show-vars-1.yaml**

PLAY [Show variables 1, first play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"test1": "test all lower-case"

}

ok: [bilbo] => {

"test1": "test all lower-case"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"Test1": "Test first capital"

}

ok: [bilbo] => {

"Test1": "Test first capital"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"name\_plus\_host": "vsrx1 :: 192.0.2.10"

}

ok: [bilbo] => {

"name\_plus\_host": "bilbo :: 198.51.100.5"

}

PLAY [Show variables 1, second play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"test1": "VARIABLE IS NOT DEFINED!"

}

ok: [bilbo] => {

"test1": "VARIABLE IS NOT DEFINED!"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"Test1": "VARIABLE IS NOT DEFINED!"

}

ok: [bilbo] => {

"Test1": "VARIABLE IS NOT DEFINED!"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"name\_plus\_host": "VARIABLE IS NOT DEFINED!"

}

ok: [bilbo] => {

"name\_plus\_host": "VARIABLE IS NOT DEFINED!"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=6 changed=0 unreachable=0 failed=0

vsrx1 : ok=6 changed=0 unreachable=0 failed=0

Observe that the variables test1 and Test1 are different and contain different data; variables names are case-sensitive. Also observe how the variables defined in the vars section of the first play are undefined in the second play. The scope of

variables defined in vars or vars\_prompt sections is the play in which they are de- fined. However, variables defined in the vars section are also specific to each host; notice how the value of name\_plus\_host contains host-specific data.

Run the playbook again, but provide an “extra” variable called test1 using the command-line option --extra-vars. The “extra” test1 variable clashes with the name of one variable in the vars section of the first play:

mbp15:aja sean$ **ansible-playbook show-vars-1.yaml --extra-vars 'test1="test one extra"'**

PLAY [Show variables 1, first play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"test1": "test one extra"

}

ok: [bilbo] => {

"test1": "test one extra"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"Test1": "Test first capital"

}

ok: [bilbo] => {

"Test1": "Test first capital"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"name\_plus\_host": "vsrx1 :: 192.0.2.10"

}

ok: [bilbo] => {

"name\_plus\_host": "bilbo :: 198.51.100.5"

}

PLAY [Show variables 1, second play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"test1": "test one extra"

}

ok: [bilbo] => {

"test1": "test one extra"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"Test1": "VARIABLE IS NOT DEFINED!"

}

ok: [bilbo] => {

"Test1": "VARIABLE IS NOT DEFINED!"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"name\_plus\_host": "VARIABLE IS NOT DEFINED!"

}

ok: [bilbo] => {

"name\_plus\_host": "VARIABLE IS NOT DEFINED!"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=6 changed=0 unreachable=0 failed=0

vsrx1 : ok=6 changed=0 unreachable=0 failed=0

Observe that the extra variable takes precedence over the variable of the same name defined in the playbook. Also observe that the extra variable is defined in *both* plays within the playbook (it has *global scope*), in contrast to the other vari- ables whose scope is the play in which they were defined.

Now create playbook show-vars-2.yaml:

1|---

2|- name: Show variables 2, first play 3| hosts:

4| - all

5| connection: local 6| gather\_facts: no 7| tasks:

8| - debug: var=inventory\_hostname 9| - debug: var=dns1

10|

11| - name: get uptime using ansible core module 12| junos\_rpc:

13| rpc: get-system-uptime-information 14| output: text

15| provider:

16| host: "{{ ansible\_host }}" 17| register: uptime

18|

19| - debug: var=uptime.output\_lines 20|

21| - set\_fact: device\_time={{ uptime['output\_lines'][0] }} 22| - debug: var=device\_time

23|

24|

25|- name: Show variables 2, second play 26| hosts:

27| - all

28| connection: local 29| gather\_facts: no 30| tasks:

31| - debug: var=inventory\_hostname 32| - debug: var=dns1

33| - debug: var=uptime.output\_lines 34| - debug: var=device\_time

Run the playbook:

mbp15:aja sean$ **ansible-playbook show-vars-2.yaml**

PLAY [Show variables 2, first play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"inventory\_hostname": "vsrx1"

}

ok: [bilbo] => { "inventory\_hostname": "bilbo"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"dns1": "5.6.7.8"

}

ok: [bilbo] => { "dns1": "5.7.9.11"

}

TASK [get uptime using ansible core module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"uptime.output\_lines": [

"Current time: 2017-09-17 09:42:45 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2017-09-16 22:40:52 UTC (11:01:53 ago)",

"Protocols started: 2017-09-16 22:40:53 UTC (11:01:52 ago)",

"Last configured: 2017-09-17 07:51:58 UTC (01:50:47 ago) by sean",

" 9:42AM up 11:02, 0 users, load averages: 0.10, 0.07, 0.02"

]

}

ok: [bilbo] => { "uptime.output\_lines": [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-06 02:36:47 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:26:34 UTC (5d 02:10 ago)",

"Protocols started: 2010-01-01 00:32:43 UTC (5d 02:04 ago)",

"Last configured: 2010-01-05 04:55:24 UTC (21:41:23 ago) by sean",

" 2:36AM up 5 days, 2:10, 0 users, load averages: 0.37, 0.22, 0.09"

]

}

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"device\_time": "Current time: 2017-09-17 09:42:45 UTC"

}

ok: [bilbo] => { "device\_time": "fpc0:"

}

PLAY [Show variables 2, second play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"inventory\_hostname": "vsrx1"

}

ok: [bilbo] => { "inventory\_hostname": "bilbo"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"dns1": "5.6.7.8"

}

ok: [bilbo] => { "dns1": "5.7.9.11"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"uptime.output\_lines": [

"Current time: 2017-09-17 09:42:45 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2017-09-16 22:40:52 UTC (11:01:53 ago)",

"Protocols started: 2017-09-16 22:40:53 UTC (11:01:52 ago)",

"Last configured: 2017-09-17 07:51:58 UTC (01:50:47 ago) by sean",

" 9:42AM up 11:02, 0 users, load averages: 0.10, 0.07, 0.02"

]

}

ok: [bilbo] => { "uptime.output\_lines": [

"fpc0:",

"--------------------------------------------------------------------------", "Current time: 2010-01-06 02:36:47 UTC",

"Time Source: LOCAL CLOCK ",

"System booted: 2010-01-01 00:26:34 UTC (5d 02:10 ago)",

"Protocols started: 2010-01-01 00:32:43 UTC (5d 02:04 ago)",

"Last configured: 2010-01-05 04:55:24 UTC (21:41:23 ago) by sean",

" 2:36AM up 5 days, 2:10, 0 users, load averages: 0.37, 0.22, 0.09"

]

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"device\_time": "Current time: 2017-09-17 09:42:45 UTC"

}

ok: [bilbo] => { "device\_time": "fpc0:"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=10 changed=0 unreachable=0 failed=0

vsrx1 : ok=10 changed=0 unreachable=0 failed=0

Observe that all the variables are valid in both plays. Ansible’s magic variables, like inventory\_hostname, and variables defined in inventory files, like dns1, have global scope and thus are valid for the entire playbook. This is also true for vari- ables defined in host and group variables files, discussed later in this chapter.

1|---

Registered variable uptime and the “set\_fact” variable device\_time, both defined in the first play, are valid after they are defined, including in subsequent plays.

Inventory, registered, host, group, and many “magic” variables are associated with a particular host; observe that each device displays different output for these variables.

To get a different look at Ansible’s magic variables, and some of the other variables Ansible maintains, create playbook show-vars-3.yaml:

2|- name: Show variables 3 3| hosts:

4| - all

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - name: ansible variables 10| debug:

11| var: vars

The following output, edited for length, shows the playbook limited to a single device. There is a lot of repetition in the output when the playbook is run for mul- tiple devices, but you should try it with two or three devices to get a feel for which variables contain host-specific data:

mbp15:aja sean$ **ansible-playbook show-vars-3.yaml --limit=bilbo**

PLAY [Show variables 3] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

"ansible\_check\_mode": false, "ansible\_host": "198.51.100.5", "ansible\_play\_batch": [

"bilbo"

],

"ansible\_play\_hosts": [ "bilbo"

],

...

"dns1": "5.7.9.11",

"dns2": "5.7.9.12",

"environment": [], "group\_names": [

"ungrouped"

],

"groups": {

"all": [

"vsrx1", "bilbo"

],

"ungrouped": [

"vsrx1", "bilbo"

]

},

"hostvars": {

"bilbo": {

"ansible\_check\_mode": false, "ansible\_host": "198.51.100.5",

...

"inventory\_hostname": "bilbo",

...

},

"vsrx1": {

"ansible\_check\_mode": false, "ansible\_host": "192.0.2.10",

...

"inventory\_hostname": "vsrx1",

...

}

},

"inventory\_dir": "/Users/sean/aja", "inventory\_file": "/Users/sean/aja/inventory",

...

"playbook\_dir": "/Users/sean/aja", "role\_names": []

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=0 unreachable=0 failed=0

Spend a little time looking at all the variables. We have discussed some of them already, and many of the others are self-explanatory. We’ll discuss group\_names and groups later in this chapter when we discuss inventory groups.

The hostvars variable deserves a little discussion here. The hostvars dictionary, keyed by inventory hostname, provides a way to gain access to variables for de- vices other than the current device. For example, a task in your playbook could reference hostvars['vsrx1']['ansible\_host'] to access the ansible\_host setting for the *vsrx1* device, even if the device being processed was *bilbo* or another inventory host.

The hostvars variable is useful in reading data known for *localhost* (the system ex- ecuting the Ansible playbook) and using that data in a task related to a network device.

For example, assume you wish to save a file and use the current date and time in the filename. Recall that Ansible normally gathers facts about the computers ex- ecuting a playbook, though we generally disable this as it does not work for net- work devices. The gathered facts include the system’s date and time. We can have a playbook gather facts from *localhost* and use those facts for tasks related to net- work devices.

You can preview the gathered facts with the following command, which you may recall from the end of Chapter 2 when we used it as a quick test to confirm Ansible was working:

mbp15:aja sean$ **ansible -m setup localhost**

localhost | SUCCESS => { "ansible\_facts": {

...

"ansible\_date\_time": { "date": "2017-09-20",

"day": "20",

"epoch": "1505954801",

"hour": "20",

"iso8601": "2017-09-21T00:46:41Z", "iso8601\_basic": "20170920T204641690552",

"iso8601\_basic\_short": "20170920T204641", "iso8601\_micro": "2017-09-21T00:46:41.690666Z",

"minute": "46",

"month": "09",

"second": "41",

"time": "20:46:41",

"tz": "EDT",

"tz\_offset": "-0400", "weekday": "Wednesday", "weekday\_number": "3",

"weeknumber": "38",

"year": "2017"

},

...

"ansible\_hostname": "mbp15",

...

"module\_setup": true

},

"changed": false

}

The following playbook, show-vars-4.yaml, illustrates one approach to gathering localhost data and using the date and time information from localhost later in the playbook during plays that act on network devices:

1|---

2|- name: Show variables 4, localhost play 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: yes 7| tasks:

8| - name: construct timestamp 9| set\_fact:

10| timestamp: "{{ ansible\_date\_time.date }}\_{{ ansible\_date\_time.hour }}-{{ ansible\_date\_ time.minute }}"

11|

12|- name: Show variables 4, devices play 13| hosts:

14| - all

15| connection: local 16| gather\_facts: no

17| tasks:

18| - name: display localhost timestamp 19| debug:

20| var: hostvars.localhost.timestamp

Observe that the first play executes on localhost and gathers facts (from local- host), while the second play executes for each device in inventory.

A sample playbook run:

mbp15:aja sean$ **ansible-playbook show-vars-4.yaml**

PLAY [Show variables 4, localhost play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [construct timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

PLAY [Show variables 4, devices play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [display localhost timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"hostvars.localhost.timestamp": "2017-09-20\_20-55"

}

ok: [bilbo] => {

"hostvars.localhost.timestamp": "2017-09-20\_20-55"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=1 | changed=0 | unreachable=0 | failed=0 |
| localhost | : ok=2 | changed=0 | unreachable=0 | failed=0 |
| vsrx1 | : ok=1 | changed=0 | unreachable=0 | failed=0 |

CAUTION When using --limit with playbooks that include tasks for localhost you need to include localhost with the --limit option. Observe how the following playbook run skips the localhost play and as a result never defines the timestamp variable:

mbp15:aja sean$ **ansible-playbook show-vars-4.yaml --limit=bilbo**

PLAY [Show variables 4, localhost play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: no hosts matched

PLAY [Show variables 4, devices play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [display localhost timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"hostvars.localhost.timestamp": "VARIABLE IS NOT DEFINED!"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=0 unreachable=0 failed=0

The following playbook run, with localhost in the --limit list, works correctly:

mbp15:aja sean$ **ansible-playbook show-vars-4.yaml --limit=bilbo,localhost**

PLAY [Show variables 4, localhost play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [construct timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

PLAY [Show variables 4, devices play] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [display localhost timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"hostvars.localhost.timestamp": "2017-09-20\_21-03"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=0 unreachable=0 failed=0

localhost : ok=2 changed=0 unreachable=0 failed=0

MORE? Using variables in an Ansible playbook is a large topic. For more information, see the *Playbook Variables* page in Ansible’s online documentation: <http://docs.ansible.com/ansible/latest/playbooks_variables.html>.

## Host Data Files

Our inventory file currently looks something like this:

vsrx1 ansible\_host=192.0.2.10 dns1=5.6.7.8 dns2=5.6.7.9 bilbo ansible\_host=198.51.100.5 dns1=5.7.9.11 dns2=5.7.9.12

[all:vars] ansible\_python\_interpreter=/usr/local/bin/python

We included in the inventory file some host-specific variables – ansible\_host, dns1, and dns2. We also included the ansible\_python\_intepreter variable for the all group, a default group that includes all devices in inventory. Putting these variables in the inventory file was convenient as we started exploring Ansible.

However, earlier chapters have mentioned that the inventory file is not the pre- ferred place for variables. This section of this chapter discusses a better approach for storing host-specific data; later in the chapter we will discuss group-specific data.

Ansible allows you to have a separate YAML data file for each host, in a directory called *host\_vars* within the playbook directory. Create directory ~/aja/host\_vars on your system to contain your host data files.

---

Now create data files in the host\_vars directory for our test hosts, starting with the variables we already have in inventory. The names of the data files should match the inventory hostnames for the devices, with a *.yaml* or *.yml* extension.

For the author’s device *bilbo*, the file bilbo.yaml contains the following:

ansible\_host: 198.51.100.5

dns1: 5.7.9.11

dns2: 5.7.9.12

---

For the device *vsrx1*, the file vsrx1.yaml contains the following:

ansible\_host: 192.0.2.10

dns1: 5.6.7.8

dns2: 5.6.7.9

Remove the host variables from inventory, leaving the following:

vsrx1 bilbo

[all:vars] ansible\_python\_interpreter=/usr/local/bin/python

If you wish, you can run the show-vars-3.yaml playbook and confirm that the hosts have the variables from the new files. You can also run the base-settings.yaml play- book from Chapter 7 and ensure the configuration files are created correctly from the new data files.

Among the benefits of using host\_vars files instead of putting variables in inven- tory is the ability to easily create dictionaries or lists in the host data, and to man- age larger data sets. DNS servers are naturally a list – a host can have an arbitrary number of DNS servers, not exactly two servers as allowed by our current vari- ables. Let’s change our files to use a list for DNS servers instead of the dns1 and dns2 variables, and add a third DNS server. The file for *bilbo* becomes:

---

ansible\_host: 198.51.100.5 dns\_servers:

- 5.7.9.11

- 5.7.9.12

- 5.7.9.13

...

You can use the show-vars-3.yaml playbook to confirm the changes (abbreviated output shown below). However, this change in our host data will require changes in the base-settings.j2 template before we can use the base-settings.yaml playbook; we will discuss the template changes shortly:

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

"ansible\_check\_mode": false,

"ansible\_host": "198.51.100.5",

...

"ansible\_version": { "full": "2.3.2.0",

"major": 2,

"minor": 3,

"revision": 2,

"string": "2.3.2.0"

},

"dns\_servers": [ "5.7.9.11",

"5.7.9.12",

"5.7.9.13"

],

...

"hostvars": {

"bilbo": {

...

"ansible\_version": { "full": "2.3.2.0",

"major": 2,

"minor": 3,

"revision": 2,

"string": "2.3.2.0"

},

"dns\_servers": [ "5.7.9.11",

"5.7.9.12",

"5.7.9.13"

},

...

},

...

],

...

"playbook\_dir": "/Users/sean/aja", "role\_names": []

}

}

Note that the dns1 and dns2 variables are gone, replaced by the dns\_servers list.

Take a look at the ansible\_version dictionary in the above output, and observe how that dictionary collects together a number of version-related variables. A number of Ansible’s variables are in dictionaries like this. We saw another example with playbook showvars4.yaml; localhost’s date and time data was in a dictionary called

ansible\_date\_time.

The author likes to organize host data in his host\_vars files into dictionaries, ex- cept for ansible\_host which, as one of Ansible’s “magic” variables, won’t work correctly if placed within a user-defined dictionary. Placing host data in dictionar- ies helps document the purpose of the data. Different dictionaries can group re- lated data together, separate from other types of host-related data. For example, a *host\_info* dictionary can contain general device settings like DNS servers, while a *host\_interface* dictionary can contain interface-related settings.

---

dns\_servers:

- 1.2.3.1

- 1.2.3.2

---

aja\_host: dns\_servers:

- 5.7.9.11

- 5.7.9.12

- 5.7.9.13

Later in this chapter we talk about creating groups and defining variables for groups. Consider what to do if we have a group for each office, and we want to have a dns\_servers list for the group (office) that will apply to all devices in the of- fice. If the group’s data file contains this:

...the group’s dns\_servers variable will clash with the dns\_servers variable already defined for the hosts. Using meaningfully-named dictionaries in each data file can avoid this type of name clash, while clarifying which name servers we are referenc- ing. Consider if the host data file contained this:

...and the group data file contained this:

---

aja\_office: dns\_servers:

- 1.2.3.1

- 1.2.3.2

---

Referencing the respective name server lists in a playbook or template would re- quire {{ aja\_host.dns\_servers }} and {{ aja\_office.dns\_servers }}. The different dic- tionary names document whether we are referencing host-specific or office-wide DNS server information, and the fact that both lists are called name\_servers does not create a name clash because the lists are in different dictionaries.

There are other ways of addressing this situation – for example, we could name the lists dns\_servers\_host and dns\_servers\_office without putting them in dictionar- ies – but consider that you may also have NTP servers, RADIUS servers, prefix lists for routing or firewall policies, and various other settings, for which you might have office-wide defaults with host-specific additions. Using host and group dictionaries is the approach the author preferred after trying a couple options.

Let’s update our host\_vars files to use a host dictionary. The file host\_vars/bilbo. yaml now contains:

ansible\_host: 198.51.100.5 aja\_host:

dns\_servers:

- 5.7.9.11

- 5.7.9.12

- 5.7.9.13

---

And the file host\_vars/vsrx1.yaml now contains:

ansible\_host: 192.0.2.10 aja\_host:

dns\_servers:

- 5.6.7.8

- 5.6.7.9

- 5.6.7.10

...

Run playbook show-vars-3.yaml and confirm the results of our changes:

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

"aja\_host": {

"dns\_servers": [ "5.7.9.11",

"5.7.9.12",

"5.7.9.13"

}

}

...

]

},

...

## Using List Data – Base Settings 2

Now let’s take a quick look at how to process the list of DNS servers in a template. Edit template/base-settings.j2 as shown (added or changed lines are in boldface, line numbers added for discussion):

1|**#jinja2: lstrip\_blocks: True**

2|system {

3| host-name {{ inventory\_hostname }}; 4| login {

5| user sean {

6| uid 2000;

7| class super-user;

8| authentication {

9| ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local) 10| }

11| }

12| }

13| replace:

14| name-server {

**15| {% for server in aja\_host.dns\_servers %} 16| {{ server }};**

**17| {% endfor %}**

18| }

19| services {

20| ftp;

21| delete: ftp;

22| netconf {

23| ssh;

24| }

25| telnet;

26| delete: telnet;

27| web-management { 28| http;

29| }

30| delete: web-management; 31| }

32|}

Lines 15 – 17 above replaced the two lines that said {{ dns1 }} and {{ dns2 }} in the prior version of the template. The new version creates a *for* loop that iterates over the list of DNS servers and adds a line of configuration for each server.

For readers who are not programmers that last sentence was probably gibberish, so let’s explain a bit. A *for* loop is a programming construct that visits each ele- ment of a list1 and performs some action. Line 15 starts the for loop. The {% %} braces-and-percent-signs tell Jinja2 that whatever is inside the {% %} is a command that Jinja2 needs to interpret, much the way {{ }} tells Jinja2 that the contents are the name of a variable to be referenced. The *for* keyword introduces the com- mand, telling Jinja2 this is a for loop. The next word, *server*, defines a variable that will allow us to reference each member of the list in turn. The keyword *in* in- troduces the name of the list whose elements we want to reference, in this case the aja\_host.dns\_servers list we created above.

Line 17 denotes the end of the for loop.

Anything between the beginning and end of the for loop, line 16 in our example, will be performed for each element in the list. In this case, line 16 simply references the temporary variable *server*, putting each DNS server IP into the configuration file.

Let’s walk through the operation of the loop. Assume we are running basesettings. yaml for *bilbo*. Jinja2 is processing the template, reaches line 15, and recognizes it as the start of a for loop. Jinja2 reads the contents of the variable aja\_host.dns\_ servers, a list containing three elements. Jinja2 puts the first element, 5.7.9.11, into variable server, then moves to line 16, the first (and only in our template) line within the for loop. Line 16 takes the value from variable server and puts it into the configuration file. Jinja2 then moves to line 17, which is the end of the loop, causing Jinja2 to return to line 15 and put the next value from the list, 5.7.9.12, into server, then move to line 16, update the configuration file, and reach the end of the loop at line 17. The process repeats again with the third element from the list. When Jinja2 returns to line 15 again, it finds it has read all the elements of the

1 Programmers with a background in C or C++ or Java may be thinking “No, for loops are counter- controlled loops!” For loops in Python and Jinja2 are more like C++11’s or Java’s *for-each* or *enhanced for* loops.

replace:

name-server {

5.6.7.8;

5.6.7.9;

5.6.7.10;

list. This causes Jinja2 to complete the for loop and move to the first line after the end of the loop, line 18.

The updated template base-settings.j2 also adds the following on line 1:

#jinja2: lstrip\_blocks: True

This is a Jinja2 directive, which causes Jinja2 to discard the leading spaces at the left of any line that is a Jinja2 command. (Technically, according to the Jinja2 API documentation page at [http://jinja.pocoo.org/docs/2.9/api,](http://jinja.pocoo.org/docs/2.9/api) "leading spaces and tabs are stripped from the start of a line to a block.") We did not need this direc- tive in the original version of the template because it contained only plain text and variable references, but the new version includes the for loop discussed above.

Without the directive on line 1, Jinja2 would include in the configuration file the leading spaces from line 15 each time Jinja2 visited the line, causing the indenta- tion of the DNS servers to be a bit unusual.

These are a few lines from the output configuration file generated with the direc- tive in the template; the indentation looks like a Junos configuration file:

}

services {

replace:

name-server {

These are the same configuration lines without the directive on line 1 in the tem- plate; note how much further indented the name server addresses and closing bracket are:

5.6.7.8;

5.6.7.9;

5.6.7.10;

}

services {

1|---

Speaking of running the playbook, we should update our base-settings.yaml play- book to use SSH key authentication (in other words, we will remove the username and password references): delete the vars\_prompt section, delete the user, passwd and port arguments from both the *install generated configuration file* task and the *con- firm commit* handler. (The following also shows the *delete generated configura- tion file* task on lines 35 – 38 commented out so that the generated configuration files will not be deleted. This is optional, but recommended, as it makes it easier to view the generated configuration files. Un-comment those lines once the template is debugged):

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "tmp"

12| conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf" 13|

14| tasks:

15| - name: confirm or create configs directory 16| file:

17| path: "{{ tmp\_dir }}"

18| state: directory 19|

20| - name: save device information using template 21| template:

22| src: template/base-settings.j2 23| dest: "{{ conf\_file }}"

24|

25| - name: install generated configuration file onto device 26| junos\_install\_config:

27| host: "{{ ansible\_host }}"

28| file: "{{ conf\_file }}"

29| timeout: 120

30| replace: yes

31| confirm: 10

32| comment: "playbook base-settings.yaml, commit confirmed" 33| notify: confirm commit

34|

**35| # - name: delete generated configuration le 36| # file:**

**37| # path: "{{ conf\_file }}" 38| # state: absent**

39|

40| handlers:

41| - name: confirm commit 42| junos\_commit:

43| host: "{{ ansible\_host }}"

44| timeout: 120

45| comment: "playbook base-settings.yaml, confirming previous commit"

Run the base-settings.yaml playbook. Confirm the generated configuration files, particularly the DNS servers, look right, and that they load onto your test devices.

Let’s add SNMP location and description information to our host variables. The author’s file host\_vars/bilbo.yaml now includes:

---

ansible\_host: 198.51.100.5 aja\_host:

dns\_servers:

- 5.7.9.11

- 5.7.9.12

- 5.7.9.13

snmp\_description: EX2200-C for testing snmp\_location: "Sean's home office"

---

And the author’s file host\_vars/vsrx1.yaml now includes:

ansible\_host: 192.0.2.10 aja\_host:

dns\_servers:

- 5.6.7.8

- 5.6.7.9

- 5.6.7.10

snmp\_description: virtual SRX for testing snmp\_location: Sean's Macbook Pro

Also append the following lines to the end of template/base-settings.j2 (shown with added line numbers):

33|snmp {

34| description "{{ aja\_host.snmp\_description }}"; 35| location "{{ aja\_host.snmp\_location }}";

36|}

[edit]

Observe how the template additions reference the snmp\_description and snmp\_loca- tion variables in the aja\_host dictionary. Also observe the quotes around the new variable references in the template (lines 34 and 35). These quotes are important because the description and location may contain spaces or other characters that Junos would regard as invalid if not quoted. Putting the quotes in the template means they will be put in the generated configuration file. You can confirm this necessity at the Junos command line:

sean@vsrx1# set snmp location Sean's office

^

syntax error.

[edit]

sean@vsrx1# set snmp location "Sean's office"

[edit] sean@vsrx1#

Run the base-settings.yaml playbook and confirm the generated configuration files are correct and that they load onto the devices.

## Inventory Options

The inventory of devices to be managed is a critical part of Ansible’s operation. As such, Ansible provides a number of options for handling inventory.

### Multiple inventory files

So far, in this book we have used a single inventory file (which we ingeniously called inventory) and notified Ansible of this fact using the ansible.cfg file:

mbp15:aja sean$ **cat ansible.cfg**

[defaults]

inventory = inventory

...

However, it is possible to have multiple, distinct inventory files, and let Ansible know which to use each time you run a playbook. This can be done using the i or inventoryfile command-line option to the ansible and ansible-playbook programs. If for example, you wished to create separate inventory files for test and production environments, you might run playbooks using commands similar to the following:

ansible-playbook uptime.yaml --inventory-file=test\_devices

…or…

ansible-playbook uptime.yaml -i production\_devices

The --inventory-file command-line option overrides the inventory setting in an- sible.cfg, so you can use ansible.cfg to set as default the inventory file you need most often, then tell Ansible to use an alternate inventory file when needed.

### Inventory directory

Another option is to place one or more inventory files in a directory, and tell An- sible to use the directory, whether via the --inventory-file command-line option or via ansible.cfg’s inventory setting. Ansible will combine the contents of all files in the inventory directory when running your playbooks. This can be useful, for ex- ample, when you want to maintain different inventory files for different physical locations (or another categorization that makes sense, such as lab vs. production), but run playbooks against all the devices as a single inventory.

[all:vars]

NOTE With only a handful of test devices it may not seem practical to maintain multiple files, but when you are maintaining an inventory of dozens or hundreds of devices it can be very helpful to have them categorized in some fashion. This will become even more clear when we add groups to these files later in this chapter.

Assume that our devices are in two different corporate offices, Boston and San Francisco, and we want to maintain separate inventory files for each office.

Create a new directory inventory2 in your ~/aja directory, and within ~/aja/inven- tory2 create files all\_vars, boston and san\_francisco. The all\_vars file will hold vari- ables applicable to all hosts, namely the ansible\_python\_interpreter setting (if your system did not require you to set this variable, you can skip this file):

ansible\_python\_interpreter=/usr/local/bin/python

The inventory file boston should contain the following (you can substitute one or more of your test devices for the author’s *bilbo*, but please ensure you have at least three names even if you do not have matching devices):

bilbo frodo sam

gimli gloin vsrx1

[defaults]

The inventory file san\_francisco should contain the following (again, substitute one of your devices for *vsrx1* and ensure you have at least three names even if you do not have matching devices):

Update your ansible.cfg file to use the new inventory2 directory:

**inventory = inventory2** host\_key\_checking = False log\_path = ~/aja/ansible.log

Let’s confirm that Ansible sees our updated inventory. We can use the --list-hosts option to the ansible or ansible-playbook commands to ask Ansible to tell us what hosts it would act upon:

mbp15:aja sean$ **ansible all --list-hosts**

hosts (6): bilbo frodo sam gimli gloin vsrx1

We can also use our show-vars-3.yaml playbook to check the membership of An- sible’s all group, which includes all devices in inventory, and to confirm that our ansible\_python\_interpreter variable is still set correctly:

mbp15:aja sean$ **ansible-playbook show-vars-3.yaml --limit=bilbo**

PLAY [Show variables 3] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

"aja\_host": {

"dns\_servers": [ "5.7.9.11",

"5.7.9.12",

"5.7.9.13"

],

"snmp\_description": "EX2200-C for testing", "snmp\_location": "Sean's home office"

},

...

**"ansible\_python\_interpreter": "/usr/local/bin/python",**

...

"groups": {

**"all": [**

**"bilbo",**

**"frodo",**

**"sam",**

**"gimli",**

**"gloin", "vsrx1"**

**],**

"ungrouped": [

"bilbo",

"frodo",

"sam",

"gimli",

"gloin", "vsrx1"

]

},

...

You can see the groups dictionary, including the automatically-created groups all and ungrouped and their members. Notice that separating the inventory into mul- tiple files does *not* automatically create a group for each file; we will create inven- tory groups in the next section of this chapter.

Do not worry about creating host data files in the host\_vars directory for the non- existent devices (i.e. frodo, gimli, gloin, sam); these devices, and a few more non- existent devices to be added in the next section, will help us discuss inventory groups and will be removed after we no longer need them.

### Inventory groups

Within your inventory file, or within the files in your inventory directory, you can define groups of devices. Group membership can be based on nearly any organiza- tional scheme that makes sense to you – location, device type, role in the network, test vs. production, etc. Groups can be nested (you can have groups whose mem- bers are other groups), and devices can be members of multiple groups. Groups can even be defined across multiple files in an inventory directory.

Once you have defined groups within your inventory, you can use set variables that will apply to all members of the group, and you can use --limit=groupname to restrict playbooks to operating on members of a group.

Each group in an inventory file begins with a heading, the group name in square brackets:

[groupname]

[groupname] device1 device2

After the group heading, list the inventory hostnames for the devices which are members of the group, one per line:

If a group’s members are other groups, the group’s heading must include the :chil- dren modifier after the group name:

[groupname:children] group1

group2

[boston] bilbo frodo sam

[sf] gimli gloin vsrx1

Let’s add location groups to our inventory files so we can easily run playbooks against devices in Boston or San Francisco. Inventory file boston becomes:

And inventory file san\_francisco becomes (abbreviating the city name to "sf"):

Use show-vars-3.yaml to confirm the groups were created:

mbp15:aja sean$ **ansible-playbook show-vars-3.yaml --limit=bilbo**

PLAY [Show variables 3] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

...

"groups": {

"all": [

"bilbo",

"frodo",

"sam",

"gimli",

"gloin", "vsrx1"

],

"boston": [

"bilbo",

"frodo", "sam"

],

"sf": [

"gimli",

"gloin", "vsrx1"

},

...

],

"ungrouped": []

You can also confirm group membership with --list-hosts; for example:

mbp15:aja sean$ **ansible boston --list-hosts**

hosts (3): bilbo frodo sam

mbp15:aja sean$ **ansible-playbook uptime.yaml --limit=sf --list-hosts**

playbook: uptime.yaml

play #1 (all): Get device uptime TAGS: [] pattern: [u'all']

hosts (3): gimli gloin vsrx1

[boston:children] bos\_ex

bos\_srx

[bos\_ex] bilbo frodo sam

[bos\_srx] Arwen

[sf:children] sf\_ex

sf\_srx

[sf\_ex] gimli gloin

[sf\_srx] galadriel vsrx1

Assume our Boston and San Francisco offices each have EX switches and SRX fire- walls. We want to create groups so that we can easily run playbooks against the switches or firewalls in each location, or the entire location. To do this, the boston and sf groups will become groups-of-groups and we will add new groups (and a couple new non-existent devices) for the switches and firewalls in each office.

The boston inventory file now contains:

The san\_francisco inventory file now contains:

Confirm the new groups using either or both of the approaches we have shown above, for example:

mbp15:aja sean$ **ansible-playbook uptime.yaml --limit=sf --list-hosts**

playbook: uptime.yaml

play #1 (all): Get device uptime TAGS: [] pattern: [u'all']

hosts (4): gimli

gloin vsrx1 galadriel

mbp15:aja sean$ **ansible bos\_ex --list-hosts**

hosts (3): bilbo frodo sam

mbp15:aja sean$ **ansible bos\_srx --list-hosts**

hosts (1): arwen

mbp15:aja sean$ **ansible-playbook show-vars-3.yaml --limit=bilbo**

PLAY [Show variables 3] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

...

"groups": {

"all": [

"bilbo",

"frodo",

"sam",

"arwen",

"gimli",

"gloin", "galadriel", "vsrx1"

],

"bos\_ex": [

"bilbo",

"frodo", "sam"

],

"bos\_srx": [

"arwen"

],

"boston": [

"bilbo",

"frodo",

"sam", "arwen"

],

"sf": [

"gimli",

"gloin", "galadriel", "vsrx1"

],

"sf\_ex": [

"gimli", "gloin"

],

"sf\_srx": [

"galadriel",

"vsrx1"

],

},

...

"ungrouped": []

Now we need to run a playbook against all our EX devices. We can run the play- book with --limit=bos\_ex,sf\_ex, but as the number of sites increases, remembering all of them could be challenging. Let’s create groups ex and srx to include all our switches and firewalls. These definitions of these new groups span both our inven- tory files, but Ansible puts them together for us.

The boston inventory file now contains:

[boston:children] bos\_ex

bos\_srx

**[ex:children] bos\_ex**

**[srx:children] bos\_srx**

[bos\_ex] bilbo frodo sam

[bos\_srx] Arwen

[sf:children] sf\_ex

sf\_srx

**[ex:children] sf\_ex**

**[srx:children] sf\_srx**

[sf\_ex] gimli gloin

[sf\_srx] galadriel vsrx1

The san\_francisco inventory file now contains:

Again, confirm the inventory updates using the approaches we have discussed:

mbp15:aja sean$ **ansible ex --list-hosts**

hosts (5): bilbo

frodo sam gimli gloin

mbp15:aja sean$ **ansible srx --list-hosts**

hosts (3): arwen galadriel vsrx1

mbp15:aja sean$ **ansible-playbook show-vars-3.yaml --limit=bilbo**

PLAY [Show variables 3] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

...

"groups": {

"all": [

"bilbo",

"frodo",

"sam",

"arwen",

"gimli",

"gloin", "galadriel", "vsrx1"

],

"bos\_ex": [

"bilbo",

"frodo", "sam"

],

"bos\_srx": [

"arwen"

],

"boston": [

"bilbo",

"frodo",

"sam", "arwen"

],

"ex": [

"bilbo",

"frodo",

"sam",

"gimli", "gloin"

],

"sf": [

"gimli",

"gloin", "galadriel", "vsrx1"

],

"sf\_ex": [

"gimli", "gloin"

],

"sf\_srx": [

"galadriel", "vsrx1"

],

"srx": [

"arwen", "galadriel", "vsrx1"

},

...

[phase1] bilbo gimli arwen

[phase2] bilbo frodo gloin

],

"ungrouped": []

Sometimes you want to create groups for special purposes, such as a list of devices to be updated during a scheduled maintenance. Assume our company is conduct- ing some maintenance that will affect a subset of the company’s devices, but not a subset identified by a current group. Assume further that the maintenance will be conducted in two stages, each stage affecting different devices, and there are play- books written to carry out each stage. Create a new inventory file maintenance in our ~/aja/inventory2 directory with the following:

Confirm Ansible’s understanding of the group memberships:

mbp15:aja sean$ **ansible --list-hosts phase1**

hosts (3): gimli bilbo arwen

mbp15:aja sean$ **ansible --list-hosts phase2**

hosts (3): gloin bilbo frodo

If playbooks phase1.yaml and phase2.yaml existed, we could then run those play-

books against the appropriate devices like this:

**ansible-playbook phase1.yaml --limit=phase1 ansible-playbook phase2.yaml --limit=phase2**

NOTE Ansible does not care that the devices listed in the phase1 and phase2 groups are also listed in other groups in other files in the inventory directory, or that bilbo appears in both phase1 and phase2. This flexibility can be very useful, as the above example illustrates. However, the author suggests that you use such duplication carefully: having a device appear in numerous groups in numerous inventory files means you must be careful to find all device instances when you need to alter the inventory, such as when you retire the device and need to remove it from inventory.

### Ansible’s group\_names variable

You have seen that Ansible’s groups variable contains a dictionary showing all in- ventory groups and their members. Ansible also maintains a group\_names variable for each host containing a list of groups of which the host is a member. In later chapters, you will see how this variable can be used to execute a task only when the current host is a member of a particular group; for example, you could execute a firewall-specific task only when a device is a member of the srx group.

Enter the following playbook, show-vars-5.yaml:

1|---

2|- name: Show variables 5 3| hosts:

4| - all

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - name: group names 10| debug:

11| var: group\_names

Then run the playbook (output edited for length) and observe that each device’s

group\_names variable lists the user-defined groups of which the device is a member:

mbp15:aja sean$ ansible-playbook show-vars-5.yaml

PLAY [Show variables 5] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [group names] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"group\_names": [ "bos\_ex",

"boston",

"ex",

"phase1", "phase2"

]

}

ok: [frodo] => { "group\_names": [

"bos\_ex",

"boston",

"ex", "phase2"

]

}

ok: [arwen] => { "group\_names": [

"bos\_srx", "boston",

"phase1", "srx"

]

}

...

ok: [vsrx1] => { "group\_names": [

"sf",

"sf\_srx", "srx"

]

}

...

### Single inventory file with groups

You can define groups even when using a single inventory file. Indeed, for small environments, a single inventory file is likely to be easiest to maintain. If your test environment is like the author’s, it probably consists of just a few devices, so let’s return to using a single inventory file.

However, let’s assume that the test environment is meant to be a microcosm of the production network, so we want to replicate groups that would be helpful in a much larger environment, even if the groups contain only one device. With this in mind, let’s continue the assumption that our devices represent two offices, Boston and San Francisco. Let’s also continue the assumption that we want groups for dif- ferent device types (our lab has EX and SRX test devices, but if you have other de- vice types, feel free to create appropriately named groups).

Create a new inventory file, ~/aja/inventory3, with the following contents (adjust as needed for your device types and hostnames, but if possible have at least one de- vice in each city):

[boston:children] bos\_ex

bos\_srx

[sf:children] sf\_ex

sf\_srx

[ex:children] bos\_ex

sf\_ex [srx:children]

bos\_srx sf\_srx

[bos\_ex] bilbo

[bos\_srx] [sf\_ex]

[sf\_srx] vsrx1

[defaults]

Notice that it is possible to define empty groups, like the bos\_srx group above. The author has found this to be useful; his team has written scripts to help create and maintain the inventory files for the various corporate offices, and the inventory file for every site (office) contains the same set of “*site*\_*type*” groups (like bos\_ex) whether or not the site actually has devices of each type. This consistency between different inventory files makes manual maintenance easier, when needed, and makes the inventory scripts easier to write and maintain as we do not need to test if a given site has, for example, an SRX device before creating the *site*\_srx group.

Also notice we did not include the [all:vars] section and its ansible\_python\_inter- preter variable; we will handle this variable a little differently in the next section of this chapter.

Update your ansible.cfg file to use the new inventory inventory3 file:

**inventory = inventory3** host\_key\_checking = False log\_path = ~/aja/ansible.log

Run the show-vars-5.yaml and show-vars-3.yaml playbooks to confirm the inventory and groups are what you expect:

mbp15:aja sean$ **ansible-playbook show-vars-5.yaml**

PLAY [Show variables 5] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [group names] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"group\_names": [ "bos\_ex",

"boston", "ex"

]

}

ok: [vsrx1] => { "group\_names": [

"sf",

"sf\_srx", "srx"

]

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=0 unreachable=0 failed=0

vsrx1 : ok=1 changed=0 unreachable=0 failed=0 mbp15:aja sean$ **ansible-playbook show-vars-3.yaml --limit=bilbo**

PLAY [Show variables 3] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

...

"groups": {

"all": [

"bilbo", "vsrx1"

],

"bos\_ex": [

"bilbo"

],

"bos\_srx": [], "boston": [

"bilbo"

],

"ex": [

"bilbo"

],

"sf": [

"vsrx1"

],

"sf\_ex": [], "sf\_srx": [

"vsrx1"

],

"srx": [

"vsrx1"

],

"ungrouped": []

},

...

## Group Data Files

One benefit of defining inventory groups is that we can associate variables with groups. This can be useful if, for example, you want each device in an office to use par- ticular NTP servers, but devices in different offices use different NTP servers.

Ansible looks in a directory called group\_vars for a YAML file with the same name as a group, but with a .yaml or .yml extension. Variables in that file are made available to any hosts that are members of the group.

(Notice the similarity with the host\_vars directory and host data files within.)

Create directory ~/aja/group\_vars. Within that directory, create files boston.yaml, sf.yaml, and all.yaml.

---

The all.yaml file contains variables that are available to all hosts. This is a good place to put the ansible\_python\_interpreter variable, if it is needed for your system:

ansible\_python\_interpreter: /usr/local/bin/python

For Boston and San Francisco, we will define a list of NTP servers to be used at each site. As discussed earlier for host variables, the author likes to create a dic- tionary to help make variable names self-documenting and help avoid name collisions.

The file group\_vars/boston.yaml contains:

---

aja\_site: ntp\_servers:

- 5.7.9.101

- 5.7.9.102

The file group\_vars/sf.yaml contains:

---

aja\_site: ntp\_servers:

- 5.6.7.201

- 5.6.7.202

...

You can confirm that the correct NTP server settings are seen by the correct hosts using the show-vars-3.yaml playbook:

TASK [ansible variables] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"vars": {

"aja\_host": {

"dns\_servers": [ "5.7.9.11",

"5.7.9.12",

"5.7.9.13"

],

"snmp\_description": "EX2200-C for testing", "snmp\_location": "Sean's home office"

},

"aja\_site": {

"ntp\_servers": [ "5.7.9.101",

"5.7.9.102"

]

},

...

}

}

ok: [vsrx1] => {

"vars": {

"aja\_host": {

"dns\_servers": [

"5.6.7.8",

"5.6.7.9",

"5.6.7.10"

],

"snmp\_description": "virtual SRX for testing", "snmp\_location": "Sean's Macbook Pro"

},

"aja\_site": {

"ntp\_servers": [ "5.6.7.201",

"5.6.7.202"

}

...

]

},

...

}

Observe that each host has the aja\_site.ntp\_servers list appropriate for its location.

Update the base-settings.j2 template to include the NTP servers. Lines 32 – 37 be- low are very similar to the DNS server update we made earlier in this chapter, but reference the site-specific NTP server information:

1|#jinja2: lstrip\_blocks: True 2|system {

3| host-name {{ inventory\_hostname }}; 4| login {

5| user sean {

6| uid 2000;

7| class super-user;

8| authentication {

9| ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local) 10| }

11| }

12| }

13| replace:

14| name-server {

15| {% for server in aja\_host.dns\_servers %} 16| {{ server }};

17| {% endfor %}

18| }

19| services {

20| ftp;

21| delete: ftp;

22| netconf {

23| ssh;

24| }

25| telnet;

26| delete: telnet;

27| web-management { 28| http;

29| }

30| delete: web-management; 31| }

**32| replace:**

**33| ntp {**

**34| {% for ntp in aja\_site.ntp\_servers %} 35| server {{ ntp }};**

**36| {% endfor %}**

**37| }**

38|}

39|snmp {

40| description "{{ aja\_host.snmp\_description }}"; 41| location "{{ aja\_host.snmp\_location }}";

42|}

...

Run the base-settings.yaml playbook. Confirm the generated configurations look right and that they install on your test devices. The following are the new NTP server settings (and a few lines before and after) from the configuration file for *bilbo*:

web-management { http;

}

delete: web-management;

}

replace:

ntp {

server 5.7.9.101;

server 5.7.9.102;

}

}

snmp {

...

## References

Ansible Inventory: <http://docs.ansible.com/ansible/latest/intro_inventory.html>

Ansible Variables: <http://docs.ansible.com/ansible/latest/playbooks_variables.html>

# Chapter 9

Backing Up Device Configuration

It is a good idea to keep backups of your devices’ configurations. This is particu- larly true when developing automation that changes device configurations, as a mistake in automation has the ability to break dozens of devices very quickly.

This chapter introduces a playbook for archiving complete device configurations, and a playbook for getting a subset of a device’s configuration.

## Juniper’s junos\_get\_config Module

One of Juniper’s Galaxy modules is junos\_get\_config, which saves the configuration of a Junos device to a file on your computer. In addition to the by now familiar arguments host, user, passwd, port, and mode, junos\_get\_config accepts several argu- ments we will use in this chapter:

* + - dest The path and filename of the configuration backup file.
    - format The output format for the configuration file: “text” for the Junos text “braces-and-semicolons” format, or “xml” for XML format.
    - filter The hierarchy of the configuration to be retrieved. The module defaults to retrieving the entire configuration; filter lets us specify a subset of the con- figuration.
    - options Additional options that can alter how the configuration is retrieved or represented, such as showing inherited settings.

## Playbook for Backing Up Device Configurations – Get Config 1

The following playbook, get-config.yaml, will create a directory called *backups* in the Ansible playbook directory and back up device configurations into that direc- tory. Configurations will be saved in files named <inventory\_hostname>.conf using the text format (braces and semicolons):

1|---

2|- name: Backup Device Configuration 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| backup\_dir: "backups"

12| conf\_file: "{{ backup\_dir}}/{{ inventory\_hostname }}.conf" 13|

14| tasks:

15| - name: create backup directory if needed 16| file:

17| path: "{{ backup\_dir }}"

18| state: directory 19|

20| - name: get device configuration 21| junos\_get\_config:

22| host: "{{ ansible\_host }}"

23| dest: "{{ conf\_file }}"

24| format: text

There is not much new here – other than the junos\_get\_config module, we have seen most of the playbook’s contents in earlier examples.

Lines 1 – 8 are the typical start of an Ansible playbook for Junos automation.

Lines 10 – 12 define a variable for the backup directory name and a variable for the name of the configuration file within the backup directory.

Lines 15 – 18 ensure the backup directory exists.

Lines 20 – 24 call the junos\_get\_config module to back up the device’s configuration.

Run the playbook against your test devices:

mbp15:aja sean$ **ansible-playbook get-config.yaml**

PLAY [Backup Device Configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

ok: [bilbo]

TASK [get device configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

vsrx1 : ok=2 changed=2 unreachable=0 failed=0

If you check your Ansible playbook (~/aja) directory there should now be a subdi- rectory called backups, within which there should be a backup of each of your de- vices’ configurations:

mbp15:aja sean$ **ls -ld backups**

drwxr-xr-x 4 sean staff 136 Oct 5 11:22 backups

mbp15:aja sean$ **ls backups/**

bilbo.conf vsrx1.conf

mbp15:aja sean$ **cat backups/bilbo.conf**

## Last changed: 2010-01-01 00:42:21 UTC

version 15.1R6.7; system {

host-name bilbo;

...

This is a good start, but there are a few things that could be improved.

## Using a User-Specific Backup Path – Get Config 2

The playbook currently places the configuration backup files within the Ansible playbook directory. This may not be desirable. Consider what will happen when you want to share your Ansible playbooks and supporting files with someone else, but not the configurations of all your network devices: you will need to exclude the backups directory from what you share with the other person. (In this book’s Appendix we discuss source control, and this problem will become even more ap- parent in that context.)

What if we put the *backups* directory in our home directory instead? Alter line 11 of the playbook, the backup\_dir variable assignment, as shown:

backup\_dir: "~/backups"

Now run the playbook again:

mbp15:aja sean$ **ansible-playbook get-config.yaml**

PLAY [Backup Device Configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

ok: [vsrx1]

TASK [get device configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"changed": false, "failed": true, "msg": "Uncaught exception - please

report: [Errno 2] No such file or directory: '~/backups/vsrx1.conf'"}

fatal: [bilbo]: FAILED! => {"changed": false, "failed": true, "msg": "Uncaught exception - please report: [Errno 2] No such file or directory: '~/backups/bilbo.conf'"}

to retry, use: --limit @/Users/sean/aja/get-config.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=1 changed=1 unreachable=0 failed=1

vsrx1 : ok=1 changed=0 unreachable=0 failed=1

As you can see from the errors during the get device configuration task, not every Ansible module expands the UNIX shortcut ~ for "my home directory."

What if we changed line 11 like this (substitute your home directory path)?

backup\_dir: "/Users/sean/backups"

Now the playbook will work fine for you, in your home directory, but it will be useless to anyone else who wants to use the playbook to back up device configura- tions in their home directory.

Instead, let’s add a user-specific “ansible data path” variable to the data file group\_ vars/all.yaml. This file already contains (if needed for your environment) the path to the Python interpreter on your system, which may be system-specific, so adding a user-specific variable to this file would be fairly consistent, collecting variables that need to be customized per-system or per-user in a single file. (Again, this will become even more clear when we discuss source control.) Add the user\_data\_path variable shown below, adjusting as needed for your home directory:

---

ansible\_python\_interpreter: /usr/local/bin/python user\_data\_path: /Users/sean/Ansible

Update line 11 of the get-config.yaml playbook as follows:

backup\_dir: "{{ user\_data\_path }}/config\_backups"

Run the playbook again:

mbp15:aja sean$ **ansible-playbook get-config.yaml**

PLAY [Backup Device Configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

ok: [bilbo]

TASK [get device configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=2 unreachable=0 failed=0

vsrx1 : ok=2 changed=2 unreachable=0 failed=0

Confirm that the new ~/ansible/config\_backups directory exists and that the device backups are within it:

mbp15:aja sean$ **ls -d ~/an\***

/Users/sean/ansible

mbp15:aja sean$ **ls ~/ansible/**

config\_backups

mbp15:aja sean$ **ls ~/ansible/config\_backups/**

bilbo.conf vsrx1.conf

Did you notice that the above playbooks try to create the backup directory once for each device?

...

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

ok: [bilbo]

...

The playbook will be altered shortly to address this duplicate effort.

Take a moment to delete the ~/backups and ~/aja/backups directories that were cre- ated earlier in the chapter; you won’t need them again.

## Keeping a Configuration History – Get Config 3

The current get-config.yaml playbook will keep only the most recent configuration backup for each device, because it replaces any prior backup file each time it is run. What if we wanted to keep all configuration backups, allowing us to have a configuration history?

There are two aspects to making this happen. The first is simply ensuring we cre- ate a new file each time we run the playbook, which can easily be accomplished by including a serial number or a date-and-time string in the filename. The latter is easier because, as we have seen in previous chapters, we can get the local system date and time from Ansible.

The second issue is what to do when a new configuration backup is a duplicate of the previous one; in other words, the device’s configuration did not change be- tween the two backups. The simple approach is to ignore this consideration, but that would result in wasting disk space with numerous files that are identical but for their filename, and could result in making it difficult to find configuration changes in our history due to all the duplicates. It is preferable to save only con- figuration files that represent a change of configuration from the previous backup.

If you run get-config.yaml again without making any changes to the device’s set- tings, you should note something interesting:

...

TASK [get device configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

...

1|---

Contrast the “ok” status with the “changed” status seen in the last playbook run. If the junos\_get\_config module realizes it is overwriting an existing file, it checks to see if what it is writing is different from what is already in the file and reports the status as “changed” or “ok” (unchanged) appropriately. We can use this fact to determine if our new configuration backup represents a change of configuration from the previous backup. We will save the configuration to a temporary file and, if that temporary file is changed, we will copy it to a permanent, timestamped file- name. By copying—not moving or renaming—the temporary file, the next backup to the same temporary file will indicate whether or not there was a change of de- vice configuration.

Change get-config.yaml so it looks like the following (new or changed lines are boldfaced, line numbers added for discussion):

**2|- name: Prepare timestamp 3| hosts:**

**4| - localhost**

**5| connection: local 6| gather\_facts: yes 7|**

**8| vars:**

**9| systime: "{{ ansible\_date\_time.time | replace(':', '-') }}" 10|**

**11| tasks:**

**12| - debug: var=ansible\_date\_time.time 13| - debug: var=systime**

**14|**

**15| - name: get system date and time 16| set\_fact:**

**17| timestamp: "{{ ansible\_date\_time.date }}\_{{ systime }}"**

18|

19|- name: Backup Device Configuration 20| hosts:

21| - all 22| roles:

23| - Juniper.junos 24| connection: local 25| gather\_facts: no 26|

27| vars:

**28| backup\_dir: "{{ user\_data\_path }}/config\_backups/{{ inventory\_hostname }}" 29| temp\_conf\_file: "{{ backup\_dir}}/{{ inventory\_hostname }}"**

**30| conf\_file: "{{ temp\_conf\_file }}\_{{ hostvars.localhost.timestamp }}.conf"**

31|

32| tasks:

33| - name: create backup directory if needed 34| file:

35| path: "{{ backup\_dir }}"

36| state: directory

37|

**38| - name: save device configuration in temporary file**

39| junos\_get\_config:

40| host: "{{ ansible\_host }}"

**41| dest: "{{ temp\_conf\_file }}"**

42| format: text

**43| register: config\_results**

44|

**45| - debug: var=config\_results**

46|

**47| - name: copy temporary file to timestamped config file if different 48| copy:**

**49| src: "{{ temp\_conf\_file }}"**

**50| dest: "{{ conf\_file }}" 51| when: config\_results.changed**

Lines 2 – 17 introduce a new play that runs on localhost and creates a variable timestamp with the date and time. The timestamp variable will be used later in the playbook to add a timestamp to a configuration file’s name.

Line 9 uses a feature of Ansible we have not previously discussed, *filters*. A filter acts on a variable in some way. There are a number of filters available; the Refer- ences section at the end of the chapter has links if you wish to investigate further. We will see more filters later in this book.

The basic syntax for applying a filter to a variable is:

{{ variable | filter }}

In this playbook, we want to change the format of the system time. Ansible’s vari- able ansible\_date\_time.time uses colons as the separators between hour, minute, and second, such as 14:47:36. The problem is that colons can have special mean- ing to some UNIX command-line tools, so we should avoid them in filenames. The filter replace(':', '-') switches the colons with hyphens.

Note that the replace() filter does *not* modify the variable on which it acts; it reads the data from the variable, modifies that data, and returns the modified data so that it can be assigned to a new variable. So line 9 reads the time from ansible\_ date\_time.time, replaces colons with dashes, and assigns the modified time to vari- able systime without changing variable ansible\_date\_time.time.

The debug commands on lines 12 and 13 are to help see the change made by the filter; they can be removed later.

Lines 15 – 17 create the timestamp variable containing the date and time (with hy- phens) from the system clock. The playbook uses set\_fact for this because the vari- able needs to survive into the next play. (Recall the discussion about variable scope at the beginning of Chapter 8.)

Line 28 redefines the backup\_dir variable so each device will have its own directory for configuration backups. As the number of archived configurations for each de-

vice grows, organizing them becomes important.

Line 29 introduces the new temp\_conf\_file variable that will be used to save the temporary configuration backup.

Line 30 redefines the conf\_file variable to include the timestamp variable created above. Remember that timestamp was generated by *localhost*, not by the current device, so we need to reference it via Ansible’s hostvars variable.

Line 38 renames the configuration backup task to be more descriptive.

Line 41 alters the destination of the configuration backup to the temporary file.

Line 43 records the results of the configuration backup process in a variable con- fig\_results. The results include a variable, changed, a Boolean (*true* or *false*) value indicating whether the (temporary) configuration file just saved differed from what was there before.

The debug command on line 45 is to help see the backup results; the line can be re- moved later.

Lines 47 – 51 copy the temporary backup file to a timestamped filename using the Ansible core module copy. The src and dest arguments are the source and destina- tion filenames. The when conditional on line 51 is new; let’s discuss this in some detail.

Ansible offers several *conditionals*, statements which allow the playbook to make a yes-or-no decision, and alter what the playbook does, based on some *condition* like the value of a variable. The when conditional is the most fundamental condi- tional; it allows Ansible to determine whether or not it should run the task with which the when conditional is associated. In this playbook, the when conditional is associated with the “copy temporary file...” task, which lets the playbook decide whether or not to execute the file copy.

The *condition* for when (or any other conditional) is a valid Jinja2 expression, with- out double braces ( {{ }} ), that must evaluate to the Boolean values *true* or *false*. Our playbook simply reads the value of the config\_results.changed variable, which already contains a Boolean value (see the discussion for line 43).

In short, the when conditional on line 51 assures that the temporary file will be cop- ied to a “permanent” file *only* when the temporary file was changed by the most recent device backup, as indicated by the config\_results.changed variable.

For readers with a programming background, think of when as Ansible’s equivalent to an if or if-then statement in most programming languages. If this were a Python program, an equivalent expression might look something like this:

if config\_results['changed']: copy(src=temp\_conf\_file, dest=conf\_file)

Run the playbook:

mbp15:aja sean$ **ansible-playbook get-config.yaml**

PLAY [Prepare timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"ansible\_date\_time.time": "15:35:05"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"systime": "15-35-05"

}

TASK [get system date and time] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

PLAY [Backup Device Configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [save device configuration in temporary file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"config\_results": { "changed": true

}

}

ok: [vsrx1] => { "config\_results": {

"changed": true

}

}

TASK [copy temporary file to timestamped config file if different] \*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=4 | changed=3 | unreachable=0 | failed=0 |
| localhost | : ok=4 | changed=0 | unreachable=0 | failed=0 |
| vsrx1 | : ok=4 | changed=3 | unreachable=0 | failed=0 |

...

Look at the first and second debug tasks and observe how the replace filter (play- book line 9) changed the format of the system time:

"ansible\_date\_time.time": "15:35:05"

"systime": "15-35-05"

This is the first backup to the new device-specific configuration directories, mean- ing there is no prior temporary backup file, and thus each device shows the backup file has changed:

TASK [copy temporary file to timestamped config file if different] \*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

And the third debug task confirms this, showing that the variable changed is set to

true for each device:

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"config\_results": { "changed": true

}

}

ok: [vsrx1] => { "config\_results": {

"changed": true

}

}

TIP Notice that the debug output shows "changed": true, not "changed": "true". The absence of quotes around true is subtle but important: the value is the Boolean value *true*, not the string of characters "true".

The fact that both backups reported a change means that both temporary files get copied to the “permanent” files:

TASK [copy temporary file to timestamped config file if different] \*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

You can confirm this by checking each device’s directory:

mbp15:aja sean$ **ls -l ~/ansible/config\_backups/bilbo/**

total 16

-rw-r--r-- 1 sean staff 3347 Oct 5 15:35 bilbo

-rw-r--r-- 1 sean staff 3347 Oct 5 15:35 bilbo\_2017-10-05\_15-35-05.conf

Run the playbook again. Because the devices’ configurations have not changed, this time the backup step should report “ok” for each device and the changed vari- ables should be false. As a result, the copy step should show “skipping” for each device because the when conditional determines that the copy task should not execute:

...

TASK [save device configuration in temporary file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"config\_results": {

"changed": false

}

}

ok: [vsrx1] => { "config\_results": {

"changed": false

}

}

TASK [copy temporary file to timestamped config file if different] \*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

skipping: [vsrx1]

...

Check the devices’ backup directories and notice the system modification times on the files (your dates and times will be different from what is shown).

mbp15:aja sean$ **ls -l ~/ansible/config\_backups/bilbo/**

total 16

-rw-r--r-- 1 sean staff 3347 Oct 5 15:54 bilbo

-rw-r--r-- 1 sean staff 3347 Oct 5 15:35 bilbo\_2017-10-05\_15-35-05.conf

Notice that the temporary file’s modification time has changed, but there is no new “permanent” backup file and the modification time on the existing file is unchanged.

Now make a change to the configuration of one of your test devices. The author added a new VLAN to his switch *bilbo*, but you can make any configuration change you like:

sean@bilbo# **show | compare**

[edit interfaces interface-range unused]

* member ge-0/0/0;
* member ge-0/0/1; [edit interfaces]

interface-range unused { ... }

+ interface-range widget {

+ member ge-0/0/0;

+ member ge-0/0/1;

+ unit 0 {

+ family ethernet-switching {

+ port-mode access;

+ vlan {

+ members widget;

+ }

+ }

+ }

+ }

[edit vlans]

+ widget;

...

Now run the playbook a third time:

TASK [save device configuration in temporary file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

changed: [bilbo]

...

TASK [copy temporary file to timestamped config file if different] \*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [vsrx1]

changed: [bilbo]

...

The playbook knows that *bilbo*’s configuration changed while *vsrx1*’s configura- tion has not, and copied only *bilbo*’s temporary configuration file to the “perma- nent” file.

Look in *bilbo*’s backup directory and note the new backup file:

mbp15:aja sean$ **ls -l ~/ansible/config\_backups/bilbo/**

total 24

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -rw-r--r-- | 1 sean | staff | 3586 Oct | 5 16:09 bilbo |
| -rw-r--r-- | 1 sean | staff | 3347 Oct | 5 15:35 bilbo\_2017-10-05\_15-35-05.conf |
| -rw-r--r-- | 1 sean | staff | 3586 Oct | 5 16:09 bilbo\_2017-10-05\_16-09-05.conf |

The nice thing about having a configuration history is that you can see how the device’s configuration changed between configuration backups:

mbp15:aja sean$ **cd ~/ansible/config\_backups/bilbo/**

mbp15:bilbo sean$ **diff bilbo\_2017-10-05\_15-35-05.conf bilbo\_2017-10-05\_16-09-05.conf**

2c2

< ## Last changed: 2017-10-05 14:53:08 UTC

---

> ## Last changed: 2017-10-05 16:07:54 UTC 75,76d74

< member ge-0/0/0;

< member ge-0/0/1;

87a86,97

* interface-range widget {
* member ge-0/0/0;
* member ge-0/0/1;
* unit 0 {
* family ethernet-switching {
* port-mode access;
* vlan {
* members widget;
* }
* }
* }
* }

127a138

* widget;

TIP Should you run this playbook with --limit, remember to include localhost with the list of devices, otherwise the first play will not run and the playbook will fail for any device whose configuration has changed. Because the timestamp variable will be undefined, the dest argument for the copy task will be invalid. For example, assuming that the configuration for *vsrx1* has changed:

mbp15:aja sean$ **ansible-playbook get-config.yaml --limit=vsrx1**

PLAY [Prepare timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: no hosts matched

PLAY [Backup Device Configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device configuration in temporary file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"config\_results": { "changed": true

}

}

TASK [copy temporary file to timestamped config file if different] \*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"failed": true, "msg": "the field 'args' has an invalid value, which appears to include a variable that is undefined. The error was: {{ temp\_conf\_file }}\_{{ hostvars. localhost.timestamp }}.conf: 'dict object' has no attribute 'timestamp'\n\nThe error appears to have been in '/Users/sean/aja/get-config.yaml': line 47, column 7, but may\nbe elsewhere in the file depending on the exact syntax problem.\n\nThe offending line appears to be:\n\n\n - name: copy temporary file to timestamped config file if different\n ^ here\n"}

to retry, use: --limit @/Users/sean/aja/get-config.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=3 changed=1 unreachable=0 failed=1

## Partial Configuration Backups – Get Partial Config 1

Sometimes you do not need the entire configuration; a single Junos hierarchy is sufficient. For the author, this has most often occurred when trying to either con- firm a setting exists, or identify devices with an old setting that needs to be updat- ed. For these situations, the saved configurations are needed only temporarily, so the playbook in this section will store the configuration files in a temporary direc- tory and the filenames will not need a timestamp.

Assume that your company has recently introduced new NTP servers. You need to confirm that all network devices are using only the new NTP servers before the server team retires the old servers.

If you are regularly backing up all device configurations, perhaps using the play- book in the previous section of this chapter, you can search those backups. There are a couple of reasons why this may be more challenging than taking and search- ing a fresh backup of just the NTP server hierarchy.

1|---

* + Similar information may appear in other parts of the configuration. For exam- ple, the old NTP servers may have also been, and may still be, DNS or RADIUS servers. As a result, searching for the old NTP server’s IP may generate numer- ous false matches from other configuration hierarchies.
  + If you have a history of archived configurations, the old NTP servers are likely to appear in a number of older configurations simply because they were the current NTP servers at the time the configuration backups were taken. This makes it likely that a search will turn up numerous meaningless matches from old configuration backups, and numerous duplicate matches from repeated backups of each device. Restricting a search to configuration files created in the last X days should help, but this approach also has limitations when work- ing with backups saved only when a device’s configuration has changed, be- cause the age of the last saved configuration varies by device.

These concerns can be mitigated by making a new backup, containing only the relevant configuration hierarchy, for all devices, and putting the new backup in a directory free of old backups. Searches in that directory should be much more fo- cused because the files contain only relevant settings and the files represent the current state of the devices.

Create the following playbook get-partial-config.yaml:

2|- name: Prepare temp backup directory 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - set\_fact: tmp\_dir=tmp 10|

11| - name: erase (old) backup directory (if it exists) 12| file:

13| path: "{{ tmp\_dir }}"

14| state: absent 15|

16| - name: create backup directory 17| file:

18| path: "{{ tmp\_dir }}"

19| state: directory 20|

21|- name: Get partial device configurations 22| hosts:

23| - all 24| roles:

25| - Juniper.junos 26| connection: local 27| gather\_facts: no 28|

29| tasks:

30| - name: retrieve configuration and save to file

31| junos\_get\_config:

32| host: "{{ ansible\_host }}"

33| dest: "{{ hostvars.localhost.tmp\_dir }}/{{ inventory\_hostname }}.conf" 34| format: "text"

35| filter: "system/ntp"

Lines 2 – 19 define the first play in the playbook, running on *localhost*, which de- letes and re-creates the temporary backup directory. This ensures the partial device configuration backups will be stored in an empty directory.

Line 9 puts the backup directory path (in this example the tmp directory within the playbook directory) in a variable tmp\_dir that will survive into the next play.

Lines 21 – 35 define the second play in the playbook, running against each device, which creates the filtered configuration backups. This is done with the same ju- nos\_get\_config module that we used for complete configuration backups; the differ- ence is in the filter argument.

Line 33 sets the name of the backup file using the tmp\_dir variable from the first play. As the first play ran only on *localhost*, the second play must read the variable

hostvars.localhost.tmp\_dir.

Line 35 sets the filter argument that specifies the configuration hierarchy to be saved. Note that the filter argument is formatted similar to a UNIX directory path, using slashes ( / ) between descending levels of the hierarchy. The filter paths always start from the top of the Junos configuration hierarchy and must specify each intermediate hierarchy to the desired hierarchy.

Run the playbook:

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml**

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=1 | changed=1 | unreachable=0 | failed=0 |
| localhost | : ok=3 | changed=2 | unreachable=0 | failed=0 |
| vsrx1 | : ok=1 | changed=1 | unreachable=0 | failed=0 |

Check the results in the ~/aja/tmp directory:

mbp15:aja sean$ **ls tmp/**

bilbo.conf vsrx1.conf mbp15:aja sean$ **cat tmp/bilbo.conf**

## Last changed: 2017-10-05 16:07:54 UTC

system {

ntp {

server 5.7.9.101;

server 5.7.9.102;

}

}

mbp15:aja sean$ **cat tmp/vsrx1.conf**

## Last changed: 2017-09-29 14:23:09 UTC

system {

ntp {

server 5.6.7.201;

server 5.6.7.202;

server 1.2.3.4;

}

}

You can see in the output above that the author already added an “old” NTP serv- er to the configuration on *vsrx1*(1.2.3.4). Take a moment to add an “old” NTP server to one or more of your test systems, then re-run the get-partial-config.yaml playbook.

To search the partial backups for the old NTP server, change to the backup direc- tory and use grep:

mbp15:aja sean$ **cd tmp**

mbp15:tmp sean$ **grep "1.2.3.4" \***

vsrx1.conf: server 1.2.3.4;

## Other junos\_get\_config Options – Get Partial Config 2

The junos\_get\_config module accepts an options argument that can influence what is included in the configuration backup. The options argument accepts a dictionary containing one or more key-value pairs; we will examine three such settings and what they do.

### Candidate Versus Committed Configuration

The module normally reads the candidate configuration from the Junos device, not the last committed configuration. This means that if someone is configuring the device at the time the configuration backup is taken, the uncommitted change may

be captured by the configuration backup.

Delete the “old” NTP server setting you added to your network device in the pre- vious section of this chapter, but do not commit the change:

sean@vsrx1# **delete system ntp server 1.2.3.4**

sean@vsrx1# **show system ntp**

server 5.6.7.201;

server 5.6.7.202;

[edit]

sean@vsrx1# **show | compare**

[edit system ntp]

- server 1.2.3.4;

Run the get-partial-config.yaml playbook and check the results for that device:

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml --limit=localhost,vsrx1**

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=3 changed=2 unreachable=0 failed=0

vsrx1 : ok=1 changed=1 unreachable=0 failed=0 mbp15:aja sean$ **cat tmp/vsrx1.conf**

## Last changed: 2017-09-29 15:39:01 UTC

system {

ntp {

server 5.6.7.201;

server 5.6.7.202;

}

}

Notice that the “old” NTP server entry is gone. But that change has not been com- mitted! You can confirm this on your device by viewing the *running* configuration instead of the *candidate* configuration:

sean@vsrx1# **run show configuration system ntp**

server 5.6.7.201;

server 5.6.7.202;

server 1.2.3.4;

You can use the junos\_get\_config module’s options argument to specify that you want the committed configuration, not a candidate configuration. Add line 36 to the end of the get-partial-config.yaml playbook:

30| - name: retrieve configuration and save to file 31| junos\_get\_config:

32| host: "{{ ansible\_host }}"

33| dest: "{{ hostvars.localhost.tmp\_dir }}/{{ inventory\_hostname }}.conf" 34| format: "text"

35| filter: "system/ntp"

**36| options: {'database':'committed'}**

Now re-run the playbook and check the results:

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml --limit=localhost,vsrx1**

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=3 changed=2 unreachable=0 failed=0

vsrx1 : ok=1 changed=1 unreachable=0 failed=0 mbp15:aja **sean$ cat tmp/vsrx1.conf**

## Last commit: 2017-09-29 14:23:05 UTC by sean system {

ntp {

server 5.6.7.201;

server 5.6.7.202;

server 1.2.3.4;

}

}

Observe that this time the backed-up configuration matches the device’s commit- ted configuration with the *old* NTP server, not the uncommitted candidate con- figuration where the NTP server has been deleted.

While demonstrated here with a filtered configuration backup, you may wish to add this option to your get-config.yaml playbook to ensure your full configuration backups do not include uncommitted changes.You can now commit the configura- tion on the test device.

### Inherited Settings

Two other options relate to displaying inherited configuration settings, such as from a group or an interface-range. The two options are 'groups': 'groups' and

'inherit': 'inherit'.

These two options can be used separately, but the author finds that it is most useful to combine them. The combination provides results similar to using the "| dis- play inheritance" modifier at the Junos command line. For example:

sean@bilbo> **show configuration interfaces | display inheritance**

...

interface-range widget { member ge-0/0/0; member ge-0/0/1; unit 0 {

family ethernet-switching { port-mode access;

vlan {

members widget;

}

}

}

} ##

## 'ge-0/0/0' was expanded from interface-range 'widget' ##

ge-0/0/0 { ##

## '0' was expanded from interface-range 'widget' ##

unit 0 { ##

## 'ethernet-switching' was expanded from interface-range 'widget' ##

family ethernet-switching { ##

## 'access' was expanded from interface-range 'widget' ##

port-mode access; ##

## 'vlan' was expanded from interface-range 'widget' ##

vlan {

##

## 'widget' was expanded from interface-range 'widget' ##

members widget;

}

}

}

}

...

Be sure one or more of your test devices includes a group and/or an interface- range. The author’s switch *bilbo* contains the interface-range shown previously.

The author’s firewall *vsrx1* contains a group that helps establish VRRP settings, which are common across numerous interfaces on the device:

sean@vsrx1> **show configuration groups**

vrrp-priority { interfaces {

<\*> {

unit <\*> {

family inet {

address <\*> {

vrrp-group <\*> { priority 110;

advertise-interval 1; accept-data;

}

}

}

}

}

}

}

sean@vsrx1> **show configuration interfaces ge-0/0/0**

apply-groups vrrp-priority; unit 0 {

family inet {

address 203.0.113.2/24 { vrrp-group 113 {

virtual-address 203.0.113.1;

}

}

}

}

...

Modify the last two lines of the get-partial-config.yaml playbook as follows:

29| tasks:

30| - name: retrieve configuration and save to file 31| junos\_get\_config:

32| host: "{{ ansible\_host }}"

33| dest: "{{ hostvars.localhost.tmp\_dir }}/{{ inventory\_hostname }}.conf" 34| format: "text"

**35| filter: "interfaces"**

**36| options: {'groups': 'groups', 'inherit': 'inherit'}**

Run the playbook and examine the configuration file for each device. The file for

*bilbo* contains, in part:

interface-range widget { member ge-0/0/0; member ge-0/0/1; unit 0 {

family ethernet-switching { port-mode access;

vlan {

members widget;

}

}

}

} ##

## 'ge-0/0/0' was expanded from interface-range 'widget' ##

ge-0/0/0 { ##

## '0' was expanded from interface-range 'widget' ##

unit 0 { ##

## 'ethernet-switching' was expanded from interface-range 'widget' ##

family ethernet-switching { ##

## 'access' was expanded from interface-range 'widget' ##

port-mode access; ##

## 'vlan' was expanded from interface-range 'widget' ##

vlan {

##

## 'widget' was expanded from interface-range 'widget' ##

members widget;

}

}

}

} ##

## 'ge-0/0/1' was expanded from interface-range 'widget' ##

ge-0/0/1 {

...

}

ge-0/0/0 {

unit 0 {

The file for *vsrx1* contains, in part:

family inet {

address 203.0.113.2/24 { vrrp-group 113 {

virtual-address 203.0.113.1;

priority 110; ##

## '1' was inherited from group 'vrrp-priority' ##

advertise-interval 1; ##

## 'accept-data' was inherited from group 'vrrp-priority' ##

accept-data;

}

}

}

}

}

One thing to keep in mind about the 'inherit': 'inherit' option: the groups por- tion of the configuration will not appear in the configuration file, even if the filter argument were not present. This makes the 'inherit': 'inherit' option of limited value for full configuration backups as you would not be able to restore a device to its original configuration with such a backup file. If you want to see this, comment out line 35 of the get-partial-config.yaml playbook and run it again; without the filter argument you would normally get a complete configuration, but thanks to inherit the “complete” configuration will be missing the groups hierarchy.

However, if you are auditing a configuration to confirm all settings have been ap- plied correctly, seeing the configuration with groups and interface-ranges “ex- panded” might be exactly what you want.

## Extra and Required Variables – Get Partial Config 3

It is likely that you will need to modify the filter argument in the get-partial-con- fig.yaml playbook each time you use the playbook, based on the Junos hierarchy you need to check. However, modifying a playbook each time you need to use it is generally not good practice. Among other reasons, it makes using the playbook more difficult, particularly for users who may not be comfortable editing a “pro- gram” file each time they wish to run the program.

An alternative approach is to assign the filter value using a variable provided on the command-line, what Ansible calls an “extra” variable. When running a play- book, you provide an extra variable using the --extra-vars or -e command-line options.

Make the following (boldfaced) changes to the get-partial-config.yaml playbook:

1|---

2|- name: Prepare temp backup directory 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - set\_fact: tmp\_dir=tmp 10|

11| - name: erase (old) backup directory (if it exists) 12| file:

13| path: "{{ tmp\_dir }}"

14| state: absent 15|

16| - name: create backup directory 17| file:

18| path: "{{ tmp\_dir }}"

19| state: directory 20|

**21| - name: show filter setting from command line**

**22| debug:**

**23| var: filter**

**24| verbosity: 1**

25|

26|- name: Get partial device configurations 27| hosts:

28| - all 29| roles:

30| - Juniper.junos 31| connection: local 32| gather\_facts: no 33|

34| tasks:

35| - name: retrieve configuration and save to file 36| junos\_get\_config:

37| host: "{{ ansible\_host }}"

38| dest: "{{ hostvars.localhost.tmp\_dir }}/{{ inventory\_hostname }}.conf" 39| format: "text"

**40| filter: "{{ filter }}"**

**41| options: {'database':'committed','groups':'groups','inherit':'inherit'}**

Run the playbook with the arguments -v --extra-vars "filter=system/host-name": mbp15:aja sean$ **ansible-playbook get-partial-config.yaml -v --extra-vars "filter=system/host-name"**

Using /Users/sean/aja/ansible.cfg as config file

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {"ansible\_facts": {"tmp\_dir": "tmp"}, "changed": false}

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost] => {"changed": true, "path": "tmp", "state": "absent"}

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => {"changed": true, "gid": 20, "group": "staff", "mode": "0755", "owner":

"sean", "path": "tmp", "size": 68, "state": "directory", "uid": 502}

TASK [show filter setting from command line] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"filter": "system/host-name"

}

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1] => {"changed": true}

changed: [bilbo] => {"changed": true}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=1 | changed=1 | unreachable=0 | failed=0 |
| localhost | : ok=4 | changed=2 | unreachable=0 | failed=0 |
| vsrx1 | : ok=1 | changed=1 | unreachable=0 | failed=0 |

Observe that the output from the debug step shows the value provided at the com- mand-line for the filter variable.

Display one of the partial configuration files to confirm it worked, capturing the system’s host name:

mbp15:aja sean$ **cat tmp/bilbo.conf**

## Last commit: 2017-10-06 14:54:34 UTC by sean system {

host-name bilbo;

}

Great! But what happens if you forget to provide the value for filter?

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml -v**

Using /Users/sean/aja/ansible.cfg as config file

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {"ansible\_facts": {"tmp\_dir": "tmp"}, "changed": false}

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost] => {"changed": true, "path": "tmp", "state": "absent"}

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => {"changed": true, "gid": 20, "group": "staff", "mode": "0755", "owner":

"sean", "path": "tmp", "size": 68, "state": "directory", "uid": 502}

TASK [show filter setting from command line] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"filter": "VARIABLE IS NOT DEFINED!"

}

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [bilbo]: FAILED! => {"failed": true, "msg": "the field 'args' has an invalid value, which appears to include a variable that is undefined. The error was: 'filter' is undefined\n\nThe error appears to have been in '/Users/sean/aja/get-partial-config.yaml': line 35, column 7, but may\nbe elsewhere in the file depending on the exact syntax problem.\n\nThe offending line appears to be:\n\n tasks:\n - name: retrieve configuration and save to file\n ^ here\n"}

fatal: [vsrx1]: FAILED! => {"failed": true, "msg": "the field 'args' has an invalid value, which appears to include a variable that is undefined. The error was: 'filter' is undefined\n\nThe error appears to have been in '/Users/sean/aja/get-partial-config.yaml': line 35, column 7, but may\nbe elsewhere in the file depending on the exact syntax problem.\n\nThe offending line appears to be:\n\n tasks:\n - name: retrieve configuration and save to file\n ^ here\n"}

to retry, use: --limit @/Users/sean/aja/get-partial-config.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=0 | changed=0 | unreachable=0 | failed=1 |
| localhost | : ok=4 | changed=2 | unreachable=0 | failed=0 |
| vsrx1 | : ok=0 | changed=0 | unreachable=0 | failed=1 |

Clearly a failure. The error message makes it fairly clear that “'filter' is unde- fined,” though the user does need to look carefully to see that part of the message.

We can tell Ansible that a variable is mandatory, which allows Ansible to provide a somewhat clearer error message. This is done with a filter. Modify lines 23 and 40 to include the "| mandatory" filter as shown:

23| var: filter | mandatory

40| filter: "{{ filter | mandatory }}"

Run the playbook again. Depending on whether or not you include the –v option, and thus whether or not the debug module on lines 21-24 is processed, you will get somewhat different results, but in either case the error message “Mandatory vari- able not defined” is easier to understand, if less specific:

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml**

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [show filter setting from command line] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [localhost]

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [bilbo]: FAILED! => {"failed": true, "msg": "Mandatory variable not defined."} fatal: [vsrx1]: FAILED! => {"failed": true, "msg": "Mandatory variable not defined."} to retry, use: --limit @/Users/sean/aja/get-partial-config.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=0 | changed=0 | unreachable=0 | failed=1 |
| localhost | : ok=3 | changed=2 | unreachable=0 | failed=0 |
| vsrx1 | : ok=0 | changed=0 | unreachable=0 | failed=1 |

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml -v**

Using /Users/sean/aja/ansible.cfg as config file

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {"ansible\_facts": {"tmp\_dir": "tmp"}, "changed": false}

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost] => {"changed": true, "path": "tmp", "state": "absent"}

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => {"changed": true, "gid": 20, "group": "staff", "mode": "0755", "owner":

"sean", "path": "tmp", "size": 68, "state": "directory", "uid": 502}

TASK [show filter setting from command line] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [localhost]: FAILED! => {"failed": true, "msg": "Mandatory variable not defined."} to retry, use: --limit @/Users/sean/aja/get-partial-config.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=3 changed=2 unreachable=0 failed=1

One problem with these error messages, with or without the mandatory filter, is that they do not tell the user how to define the required variable. Can we tell the play- book to fail with a meaningful error message if the filter variable is not defined?

Ansible includes a core module called fail which tells the playbook to stop and, optionally, provide an error message. The fail module will normally have a when statement or other conditional (rarely do we want a playbook to *always* fail at the same place!). We can use the fail module with a when condition that determines if the filter variable is undefined.

Modify the playbook, removing the mandatory filters and adding the fail task:

1|---

2|- name: Prepare temp backup directory 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| tasks:

**9| - name: fail if filter not defined 10| fail:**

**11| msg: >**

**12| Specify the Junos configuration hierarchy you want to back up by 13| providing the extra variable 'filter' on the command line.**

**14| For example, --extra-vars 'filter=system/ntp' 15| when: filter is not defined**

16|

17| - set\_fact: tmp\_dir=tmp 18|

19| - name: erase (old) backup directory (if it exists) 20| file:

21| path: "{{ tmp\_dir }}"

22| state: absent 23|

24| - name: create backup directory 25| file:

26| path: "{{ tmp\_dir }}"

27| state: directory 28|

29| - name: show filter setting from command line 30| debug:

**31| var: filter**

32| verbosity: 1 33|

34|- name: Get partial device configurations 35| hosts:

36| - all 37| roles:

38| - Juniper.junos 39| connection: local 40| gather\_facts: no 41|

42| tasks:

43| - name: retrieve configuration and save to file 44| junos\_get\_config:

45| host: "{{ ansible\_host }}"

46| dest: "{{ hostvars.localhost.tmp\_dir }}/{{ inventory\_hostname }}.conf" 47| format: "text"

**48| filter: "{{ filter }}"**

49| options: {'database':'committed','groups':'groups','inherit':'inherit'}

Lines 9 – 15 are the new task that fails (exits) the playbook when the variable fil- ter is not defined.

The msg argument on lines 11 – 14 dictates the error message that fail will display. This playbook uses a feature of YAML, the greater-than sign (">"), to spread the rather long error message across three lines of the playbook. Ansible assembles the lines, stripping the leading spaces (indentation), and inserting a single space be- tween each assembled line.

The when condition on line 15 returns Boolean true when the variable filter is not defined, which causes the fail module to execute. If the user provided filter on the command line, the when condition returns false (because filter is defined) and fail does not execute.

Run the playbook without the required argument and note the error message, par- ticularly how the three lines in the playbook were assembled into a single message:

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml**

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TASK [fail if filter not defined] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [localhost]: FAILED! => {"changed": false, "failed": true, "msg": "Specify the Junos configuration hierarchy you want to back up by providing the value of extra variable 'filter' on the command line. For example, --extra-vars 'filter=system/ntp'\n"}

to retry, use: --limit @/Users/sean/aja/get-partial-config.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=0 changed=0 unreachable=0 failed=1

Run the playbook with the required argument:

mbp15:aja sean$ **ansible-playbook get-partial-config.yaml --extra-vars 'filter=system/name- server' --limit=localhost,vsrx1**

PLAY [Prepare temp backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [fail if filter not defined] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [localhost]

TASK [set\_fact] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [erase (old) backup directory (if it exists)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [create backup directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [show filter setting from command line] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [localhost]

PLAY [Get partial device configurations] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [retrieve configuration and save to file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=3 changed=2 unreachable=0 failed=0

vsrx1 : ok=1 changed=1 unreachable=0 failed=0 mbp15:aja sean$ **cat tmp/vsrx1.conf**

## Last commit: 2017-09-29 16:38:50 UTC by sean system {

name-server {

5.6.7.8;

5.6.7.9;

5.6.7.10;

}

}

## References

Ansible and Jinja2 filters: <http://docs.ansible.com/ansible/latest/playbooks_filters.html>

<http://jinja.pocoo.org/docs/2.9/templates/#list-of-builtin-filters>

Ansible conditionals: <http://docs.ansible.com/ansible/latest/playbooks_conditionals.html>

Junos get-configuration RPC: [https://www.juniper.net/documentation/en\_US/junos/topics/reference/tag-summa-](https://www.juniper.net/documentation/en_US/junos/topics/reference/tag-summary/junos-xml-protocol-get-configuration.html) [ry/junos-xml-protocol-get-configuration.html](https://www.juniper.net/documentation/en_US/junos/topics/reference/tag-summary/junos-xml-protocol-get-configuration.html)

# Chapter 10

Gathering and Using Device Facts

In Chapter 4, we used the junos\_command and junos\_rpc (junos\_run\_rpc) modules to run a command or RPC to get specific information, the system’s uptime, from a Junos device. This process could be repeated with other commands or RPCs to gather different information. However, if you need a variety of facts about a device

– perhaps for a device inventory report that includes serial number, model number, Junos version, and the like – it may become tedious to run numerous commands or RPCs to gather and assemble the disparate facts required.

Juniper’s Galaxy modules include junos\_get\_facts, a module that gathers a number of frequently needed facts about a Junos device and presents those facts in a single dictionary.

In this chapter we create two playbooks, each of which illustrates how junos\_get\_ facts may be helpful. We create a playbook that generates a device inventory re- port, and we create a playbook that generates different configuration settings based on the model of the device being configured. These playbooks also allow us to further explore Jinja2 templates, including how to use a template to save data to a file.

## Device Inventory Report

We want to generate a report containing basic information about our Ansible- managed network devices, including serial number and Junos version. The report should be in CSV (comma-separated value) format so it is plain text and thus easy to create and troubleshoot, yet it can easily be read and analyzed using Microsoft Excel or another spreadsheet application.

1|---

Because we may wish to keep the inventory reports over time, the playbook cre- ates a report directory within the directory we set up for our device configuration backups.

Let’s start with seeing which facts are returned by the junos\_get\_facts module. Cre- ate the playbook get-device-facts.yaml containing the following:

2|- name: Get facts from Junos device 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| tasks:

11| - name: get device facts 12| junos\_get\_facts:

13| host: "{{ ansible\_host }}" 14| register: junos\_facts

15|

16| - name: show device facts 17| debug:

18| var: junos\_facts

Everything here should be familiar from playbooks in earlier chapters. Let’s run the playbook and see which facts are gathered:

mbp15:aja sean$ **ansible-playbook get-device-facts.yaml**

PLAY [Get facts from Junos device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [show device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"junos\_facts": { "changed": false, "facts": {

"HOME": "/var/home/sean", "RE0": {

"last\_reboot\_reason": "Router rebooted after a normal shutdown.", "mastership\_state": "master",

"model": "EX2200-C-12T-2G",

"status": "Absent",

"up\_time": "14 days, 20 hours, 1 minute, 35 seconds"

...

},

"RE1": null,

"has\_2RE": false, "hostname": "bilbo", "hostname\_info": {

"fpc0": "bilbo"

},

...

...

...

}

}

}

"ifd\_style": "SWITCH",

"master": "RE0",

"model": "EX2200-C-12T-2G",

"model\_info": {

"fpc0": "EX2200-C-12T-2G"

},

"personality": "SWITCH",

"serialnumber": "GP0211463844", "srx\_cluster": null, "srx\_cluster\_id": null, "srx\_cluster\_redundancy\_group": null, "switch\_style": "VLAN",

"vc\_capable": true, "vc\_fabric": false, "vc\_master": "0", "vc\_mode": "Enabled", "version": "15.1R6.7",

ok: [vsrx1] => { "junos\_facts": {

"changed": false, "facts": {

"HOME": "/var/home/sean", "RE0": {

"last\_reboot\_reason": "0x4000:VJUNOS reboot", "mastership\_state": "master",

"model": "VSRX-S",

"status": "OK",

"up\_time": "31 minutes, 36 seconds"

...

},

"RE1": null,

"has\_2RE": false, "hostname": "vsrx1", "hostname\_info": {

"re0": "vsrx1"

...

},

"ifd\_style": "CLASSIC",

"master": "RE0",

"model": "VSRX", "model\_info": {

"re0": "VSRX"

...

},

"personality": null,

"serialnumber": "18E60B68CF6D", "srx\_cluster": false, "srx\_cluster\_id": null, "srx\_cluster\_redundancy\_group": null, "switch\_style": "VLAN\_L2NG", "vc\_capable": false,

"vc\_fabric": null,

...

}

}

"vc\_master": null, "vc\_mode": null,

"version": "15.1X49-D90.7",

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

vsrx1 : ok=2 changed=0 unreachable=0 failed=0

The junos\_get\_facts module returns a dictionary, which we assigned to playbook variable junos\_facts, containing two keys. The key "changed" is a Boolean indicat- ing if the module changed the device; this value should always be false for this module. The other key, "facts", is a dictionary of device facts.

There are too many facts to discuss them in detail in this book. Many facts are self- explanatory. The Ansible module basically reads the PyEZ facts dictionary jnpr. junos.facts; see the PyEZ documentation for a description of the facts (there is a link in the References section at the end of this chapter).

Note that some facts are set to null; this usually means the device has no meaning- ful answer to the “question” posed by that fact. For example, in the author’s out- put above, both test devices have a single routing engine, so the references to RE1 are null because the devices have no RE1. Another example, the switch *bilbo* sets the fields for SRX cluster details to null because an EX2200 cannot be part of an SRX cluster, while the virtual SRX firewall *vsrx1* sets the fields for virtual chassis details to null because an SRX cannot be a member of a switch virtual chassis.

If your test environment includes any EX virtual chassis, or SRX clusters, or MX routers with dual routing engines, run the playbook against those devices and ex- plore the facts gathered. Many of the facts include more information when work- ing with multi-RE or multi-chassis logical devices.

How will we save these results to a file? Ansible does not have a “save variable” module, but we can use Ansible’s core module template to accomplish this task. We previously used the template module in the base-settings.yaml playbook to generate Junos configuration files from data in host and group variable files. For this play- book we use template to take facts from the junos\_facts variable and write them to a file on disk.

Let’s start with a very basic template. Create file ~/aja/template/device-facts.j2

with the following content:

{{ junos\_facts.facts }}

This template inserts the entire junos\_facts.facts dictionary into the result file. (We will clean this up shortly.)

1|---

Now update the get-device-facts.yaml playbook as follows:

**2|- name: Set up report directory 3| hosts:**

**4| - localhost**

**5| connection: local 6| gather\_facts: no 7|**

**8| tasks:**

**9| - name: generate report directory name 10| set\_fact:**

**11| report\_dir: "{{ user\_data\_path }}/reports" 12|**

**13| - name: confirm/create report directory 14| file:**

**15| path: "{{ report\_dir }}"**

**16| state: directory**

17|

18|- name: Get facts from Junos device 19| hosts:

20| - all 21| roles:

22| - Juniper.junos 23| connection: local 24| gather\_facts: no 25|

26| tasks:

27| - name: get device facts 28| junos\_get\_facts:

29| host: "{{ ansible\_host }}" 30| register: junos\_facts

31|

32| - name: show device facts 33| debug:

34| var: junos\_facts

**35| verbosity: 1 36|**

**37| - name: save device information using template 38| template:**

**39| src: template/device-facts.j2**

**40| dest: "{{ hostvars.localhost.report\_dir }}/{{ inventory\_hostname }}.txt"**

The first play, on lines 2 – 16, runs as localhost to create the reports directory, which is ~/ansible/reports if your variable user\_data\_path (in group\_vars/all.yaml) is set to your user account’s version of /Users/sean/ansible. The playbook executes these tasks as localhost because we need to create—or confirm the existence of— the reports directory only once, not once per device. (If you use --limit when run- ning this playbook, remember to include localhost in the --limit list.)

The second play, on lines 18 – 40, gets the device facts and registers them in vari- able junos\_facts, then saves each device’s facts using the device-facts.j2 template. The facts for each device are saved in a file <inventory\_hostname>.txt in the reports directory.

Run the playbook (remember to include localhost if using --limit): mbp15:aja sean$ **ansible-playbook get-device-facts.yaml**

PLAY [Set up report directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [generate report directory name] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [confirm/create report directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY [Get facts from Junos device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [show device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

skipping: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=2 | changed=1 | unreachable=0 | failed=0 |
| localhost | : ok=2 | changed=1 | unreachable=0 | failed=0 |
| vsrx1 | : ok=2 | changed=1 | unreachable=0 | failed=0 |

Confirm that the playbook created the ~/ansible/reports directory and wrote an output file for each device, then look at the contents of a file:

mbp15:aja sean$ **ls ~/ansible/**

config\_backups reports

mbp15:aja sean$ **ls ~/ansible/reports/**

bilbo.txt vsrx1.txt

mbp15:aja sean$ **cat ~/ansible/reports/bilbo.txt**

{u'domain': None, u'hostname\_info': {u'fpc0': u'bilbo'}, u'version\_RE1': None, u'version\_RE0': None, u're\_master': {u'default': u'0'}, u'serialnumber': u'GP0211463844', u'vc\_master': u'0', u'RE\_hw\_mi': False, u're\_info': {u'default': {u'default': {u'status': u'Absent', u'last\_reboot\_reason': u'Router rebooted after a normal shutdown.', u'model': u'EX2200-C-12T-2G', u'mastership\_state': u'master'}, u'0': {u'status': u'Absent', u'last\_reboot\_reason': u'Router rebooted after a normal shutdown.', u'model': u'EX2200-C-12T-2G', u'mastership\_state': u'master'}}}, u'HOME': u'/var/home/sean', u'srx\_ cluster\_id': None, u'hostname': u'bilbo', u'virtual': False, u'version': u'15.1R6.7', u'master': u'RE0', u'vc\_fabric': False, u'personality': u'SWITCH', u'srx\_cluster\_redundancy\_group': None, u'version\_info': {u'major': [15, 1], u'type': u'R', u'build': 7, u'minor': u'6'}, u'srx\_cluster': None, u'vc\_mode': u'Enabled', u'vc\_capable': True, u'ifd\_style': u'SWITCH', u'model\_info': {u'fpc0': u'EX2200-C-12T-2G'}, u'RE0': {u'status': u'Absent', u'last\_reboot\_reason': u'Router rebooted after a normal shutdown.', u'model': u'EX2200-C-12T-2G', u'up\_time': u'14 days, 23 hours, 11 minutes, 20 seconds', u'mastership\_state': u'master'}, u'RE1': None, u'fqdn': None, u'junos\_info': {u'fpc0':

{u'text': u'15.1R6.7', u'object': {u'major': [15, 1], u'type': u'R', u'build': 7, u'minor': u'6'}}}, u'has\_2RE': False, u'switch\_style': u'VLAN', u'model': u'EX2200-C-12T-2G', u'current\_re': [u'master', u'node', u'fwdd', u'member', u'pfem', u'fpc0', u'feb0', u'fpc16']}

The file’s contents are pretty ugly, not formatted for human consumption, but it is recognizably the same data we saw previously. We can make this data more ap- pealing by modifying the template to print each key:value pair on its own line.

Modify template/device-facts.j2 as shown (line numbers added for discussion):

1|- - - lightly formatted facts for {{ inventory\_hostname }} - - - 2|{% for fact\_name,fact\_data in junos\_facts.facts.iteritems() %} 3| {{ fact\_name }}: {{ fact\_data }}

4|{% endfor %}

Line 1 labels the output, including the inventory\_hostname for the device.

Lines 2 – 4 define a *for* loop, which we have seen before, but this one is a little different.

The *for* loops we used previously, when creating configuration files, were iterating over a list (array) of data -- each element in a list is a single value, such as an NTP server IP. In this example, the *for* loop is iterating over the junos\_facts.facts dic- tionary, meaning each entry consists of both a *key* and a *value* (remember that dic- tionaries consist of key:value pairs).

Because we want to display both the key (name) and value (data) for each diction- ary entry, the *for* loop declaration needs to follow this pattern:

**{% for , in .iteritems() %}**

The new iteritems() word (technically, a function or method call on the underlying Python dictionary structure) in the for loop causes the loop to return both the key and value for each element in the variable dictionary, which are assigned to the respective *key* and *value* variables.

Run the playbook again (not shown) and view the results:

mbp15:aja sean$ **cat ~/ansible/reports/bilbo.txt**

- - - lightly formatted facts for bilbo - - - domain:

hostname\_info: {u'fpc0': u'bilbo'} version\_RE1:

version\_RE0:

re\_master: {u'default': u'0'} serialnumber: GP0211463844 vc\_master: 0

RE\_hw\_mi: False

re\_info: {u'default': {u'default': {u'status': u'Absent', u'last\_reboot\_reason': u'Router rebooted after a normal shutdown.', u'model': u'EX2200-C-12T-2G', u'mastership\_state': u'master'}, u'0':

{u'status': u'Absent', u'last\_reboot\_reason': u'Router rebooted after a normal shutdown.', u'model': u'EX2200-C-12T-2G', u'mastership\_state': u'master'}}}

HOME: /var/home/sean srx\_cluster\_id: hostname: bilbo virtual: False version: 15.1R6.7 master: RE0

vc\_fabric: False personality: SWITCH

srx\_cluster\_redundancy\_group:

version\_info: {u'major': [15, 1], u'type': u'R', u'build': 7, u'minor': u'6'} srx\_cluster:

vc\_mode: Enabled vc\_capable: True ifd\_style: SWITCH

model\_info: {u'fpc0': u'EX2200-C-12T-2G'}

RE0: {u'status': u'Absent', u'last\_reboot\_reason': u'Router rebooted after a normal shutdown.', u'model': u'EX2200-C-12T-2G', u'up\_time': u'14 days, 23 hours, 19 minutes, 35 seconds', u'mastership\_ state': u'master'}

RE1:

fqdn:

junos\_info: {u'fpc0': {u'text': u'15.1R6.7', u'object': {u'major': [15, 1], u'type': u'R', u'build': 7, u'minor': u'6'}}}

has\_2RE: False switch\_style: VLAN model: EX2200-C-12T-2G

current\_re: [u'master', u'node', u'fwdd', u'member', u'pfem', u'fpc0', u'feb0', u'fpc16']

Notice how each element of the facts dictionary starts on a new line, and each line has the format *key: value*. Because many of the values are themselves dictionaries with further key:value pairs, many of the lines wrap and are poorly formatted. The order of the dictionary entries may vary, because Python dictionaries do not guar- antee the sequence of elements in a dictionary. Still, this represents a significant improvement over the first template.

Should you iterate over a dictionary without using iteritems() – in other words, as if the dictionary were a list – you will get only the keys from the dictionary. If you wish to see this, modify the template as follows:

- - - lightly formatted facts for {{ inventory\_hostname }} - - -

**{% for fact in junos\_facts.facts %}**

**{{ fact }}**

{% endfor %}

...and re-run the playbook. Go ahead, give it a try, we’ll wait for you.

### Changing the output to CSV format

Our original intent, described at the start of the section, was to create a single re- port in CSV format. Right now, we have a separate file for each device, not a single report, and the devices’ files are not comma separated.

In order to generate a single CSV file, with one line for each device, we need to make a few changes. First, our template needs to generate a single line for a device, with different fields separated by commas. Second, while not strictly necessary, it would be nice if each “column” in the CSV file had a label. Third, we need to as- semble the individual device files and column labels into a single report.

Adjusting the template to put all desired values on a single line, separated by com- mas, could be done as follows. (Do not type this, we will alter it momentarily to make it shorter and easier to type). This is a single line in the template, though it wraps to several lines here:

"{{ inventory\_hostname }}","{{ junos\_facts.facts.version }}","{{ junos\_facts.facts.model }}","{{ junos\_facts.facts.switch\_style }}","{{ junos\_facts.facts.serialnumber }}","{{ junos\_facts.facts. has\_2RE }}","{{ junos\_facts.facts.master }}","{{ junos\_facts.facts.vc\_capable }}","{{ junos\_facts. facts.vc\_fabric }}","{{ junos\_facts.facts.vc\_master }}","{{ junos\_facts.facts.vc\_mode }}","{{ junos\_ facts.facts.srx\_cluster }}","{{ junos\_facts.facts.srx\_cluster\_id }}"

Because most of these facts are from the junos\_facts.facts dictionary, nearly every key name is preceded with "junos\_facts.facts." It quickly gets annoying to repeat- edly type the same long variable name. Fortunately, Jinja2 offers a way to set vari- ables within the template, so we can set a very short variable name, like f, to hold the contents of junos\_facts.facts, and then reference f instead of junos\_facts.facts.

Enter the following *two lines* in the device-facts.j2 template; the second line, start- ing with "{{ inventory\_hostname }}," wraps below:

{% set f=junos\_facts.facts %}

"{{ inventory\_hostname }}","{{ f.version }}","{{ f.model }}","{{ f.switch\_style }}","{{ f.serialnumber

}}","{{ f.has\_2RE }}","{{ f.master }}","{{ f.vc\_capable }}","{{ f.vc\_fabric }}","{{ f.vc\_master

}}","{{ f.vc\_mode }}","{{ f.srx\_cluster }}","{{ f.srx\_cluster\_id }}"

(This report uses a subset of the available device facts; feel free to add or alter facts to suit your reporting needs.)

Run the playbook (not shown) and show the resulting files:

mbp15:aja sean$ **cat ~/ansible/reports/vsrx1.txt**

"vsrx1","15.1X49-D90.7","VSRX","VLAN\_L2NG","18E60B68CF6D","False","RE0","False","","","","False",""

Pretty good! It looks like a line from a CSV file. But the empty strings – "" – may not be ideal; when viewed in a spreadsheet program, these will appear as empty cells. These are the result of facts with null values. It would be nice if null values showed as a hyphen ("-") instead of nothing; it helps assure anyone looking at the results that the entry was not simply missed.

Jinja2 can test if a value is null, though Jinja2 calls the same state none. We can use an *if-else* control structure to either return a hyphen if a variable is null, or return the value if the variable is not null. The basic format is:

1| {% if f.srx\_cluster\_id is none %} 2| "-"

3| {% else %}

4| "{{ f.srx\_cluster\_id }}" 5| {% endif %}

Line 1 starts the *if-else* control structure and contains the *condition*, the test that evaluates to a Boolean true-or-false value. Here the condition f.srx\_cluster\_id is none tests if the variable f.srx\_cluster\_id contains a null (none) value.

Line 2 is what the template will put in the output file if the condition is true. This can be multiple lines, though here only one line is needed.

Line 3 starts the *else* portion of the control structure, separating the “true result” portion of the *if-else* structure from the “false result” portion.

Line 4 is what the template will put in the output file if the condition is false. This can be multiple lines, though here only one line is needed.

Line 5 ends the *if-else* control structure.

Note that, for Jinja2, the *else* section is optional with an *if* control structure. In a template that should include something when a condition is true, but should do nothing when the condition is false, you might have something like this:

{% if <condition> %}

include this line in the output when <condition> is true

{% endif %}

While it is generally preferable for an *if-else* control structure to be formatted as shown above, because seeing it across multiple lines and with indentation makes it easier to understand, Jinja2 does not care about the format. This following one line is logically equivalent to the five lines above, except for the leading spaces:

{% if f.srx\_cluster\_id is none %}"-"{% else %}"{{ f.srx\_cluster\_id }}"{% endif %}

This equivalence is sometimes important to getting correctly formatted output from the template, because sometimes we want the template’s output to be differ- ent from the preferred code layout. This template is one of those times; we want the entire output to be on a single line.

Change the device-facts.j2 template as follows (line numbers added for discussion, and line 2 may wrap in the book):

1|{% set f=junos\_facts.facts %}

2|"{{ inventory\_hostname }}","{{ f.version }}","{{ f.model }}","{{ f.switch\_style }}",

"{{ f.serialnumber }}","{{ f.has\_2RE }}","{{ f.master }}","{{ f.vc\_capable }}", 3|{% if f.vc\_fabric is none %}

4| "-",

5|{% else %}

6| "{{ f.vc\_fabric }}", 7|{% endif %}

8|{% if f.vc\_master is none %} 9| "-",

10|{% else %}

11| "{{ f.vc\_master }}", 12|{% endif %}

13|{% if f.vc\_mode is none %} 14| "-",

15|{% else %}

16| "{{ f.vc\_mode }}", 17|{% endif %}

18|{% if f.srx\_cluster is none %} 19| "-",

20|{% else %}

21| "{{ f.srx\_cluster }}", 22|{% endif %}

23|{% if f.srx\_cluster\_id is none %} 24| "-"

25|{% else %}

26| "{{ f.srx\_cluster\_id }}" 27|{% endif %}

Run the playbook (not shown) and examine the results:

mbp15:aja sean$ **cat ~/ansible/reports/bilbo.txt**

"bilbo","15.1R6.7","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True",

"False",

"0",

"Enabled", "-",

"-"

mbp15:aja sean$ **cat ~/ansible/reports/vsrx1.txt**

"vsrx1","15.1X49-D90.7","VSRX","VLAN\_L2NG","18E60B68CF6D","False","RE0","False", "-",

"-",

"-",

"False",

"-"

Not quite what we want. We got the hyphens we wanted, but we do not want the output spread across multiple lines like that, and the leading spaces before the hy- phens or data are not really desirable either. The problem here is that Jinja2 in- cludes the leading spaces and the trailing newline characters from each line of plain text, or a line with a variable reference that is not part of a Jinja2 control structure statement ({%...%}).

What if we reformat each of the *if-else* control structures onto a single line, like this?

1|{% set f=junos\_facts.facts %}

2|"{{ inventory\_hostname }}","{{ f.version }}","{{ f.model }}","{{ f.switch\_style

}}","{{ f.serialnumber }}","{{ f.has\_2RE }}","{{ f.master }}","{{ f.vc\_capable }}", 3|{% if f.vc\_fabric is none %}"-",{% else %}"{{ f.vc\_fabric }}",{% endif %}

4|{% if f.vc\_master is none %}"-",{% else %}"{{ f.vc\_master }}",{% endif %} 5|{% if f.vc\_mode is none %}"-",{% else %}"{{ f.vc\_mode }}",{% endif %}

6|{% if f.srx\_cluster is none %}"-",{% else %}"{{ f.srx\_cluster }}",{% endif %} 7|{% if f.srx\_cluster\_id is none %}"-"{% else %}"{{ f.srx\_cluster\_id }}"{% endif %}

Now the template’s output looks something like this:

mbp15:aja sean$ **cat ~/ansible/reports/bilbo.txt**

"bilbo","15.1R6.7","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True",

"False","0","Enabled","-","-"

mbp15:aja sean$ **cat ~/ansible/reports/vsrx1.txt**

"vsrx1","15.1X49-D90.7","VSRX","VLAN\_L2NG","18E60B68CF6D","False","RE0","False", "-","-","-","False","-"

We are getting closer. Jinja2 automatically suppresses the trailing newline from lines that end with a control structure, so the newlines from lines 1 and 3 – 7 are not included in the template’s output. But the newline from line 2 is still present, as this line contains only variable references and text (the commas and quotation marks), causing our results to be spread across two lines.

One approach to fixing this is to add a hyphen to the opening of the control struc- ture on the lines following the newline we wish to suppress – in other words, con- trol structure lines start with {%- instead of just {%. The added hyphen tells Jinja2 to suppress the newline from the previous line of the template. Update your template to look like this:

1|{% set f=junos\_facts.facts %}

2|"{{ inventory\_hostname }}","{{ f.version }}","{{ f.model }}","{{ f.switch\_style

}}","{{ f.serialnumber }}","{{ f.has\_2RE }}","{{ f.master }}","{{ f.vc\_capable }}", 3|{%- if f.vc\_fabric is none %}"-",{% else %}"{{ f.vc\_fabric }}",{% endif %}

4|{%- if f.vc\_master is none %}"-",{% else %}"{{ f.vc\_master }}",{% endif %}

5|{%- if f.vc\_mode is none %}"-",{% else %}"{{ f.vc\_mode }}",{% endif %}

6|{%- if f.srx\_cluster is none %}"-",{% else %}"{{ f.srx\_cluster }}",{% endif %} 7|{%- if f.srx\_cluster\_id is none %}"-"{% else %}"{{ f.srx\_cluster\_id }}"{% endif %}

While we needed to modify only line 3 to fix the current problem, the author chose to update lines 3 – 7. Adding the hyphen to lines 4 – 7 has no effect because Jinja2 is already suppressing the newlines from the preceding lines, but if we ever insert a new variable reference, say {{ f.ifd\_style }}, between the current lines 5 and 6, we will not need to remember to add the hyphen to the next line.

The template’s output now looks something like this (one line of output, though it may wrap as shown here):

mbp15:aja sean$ **cat ~/ansible/reports/bilbo.txt**

"bilbo","15.1R6.7","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl ed","-","-"

mbp15:aja sean$ **cat ~/ansible/reports/vsrx1.txt**

"vsrx1","15.1X49-D90.7","VSRX","VLAN\_L2NG","18E60B68CF6D","False","RE0","False","-","-","-

","False","-"

We now have CSV-formatted output for each device...but we’re not quite done yet.

### Building a single CSV file

How do we assemble the data files for different devices into a single CSV file? An- sible has a core module called assemble that concatenates a group of files into a new file. However, if we want our final CSV file in our report directory, we should probably move the device output files into a temporary directory. We will have our playbook create a reports/build directory to store the device data, and the CSV file will be stored in the reports directory. We will put a date stamp in the report file- name so the reports from differnt runs of the playbook will have unique names.

1|---

Modify the get-device-facts.yaml playbook as follows:

2|- name: Set up report directory 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - name: generate report directory names 10| set\_fact:

11| report\_dir: "{{ user\_data\_path }}/reports" 12|

**13| - name: generate report build directory name**

**14| set\_fact:**

**15| build\_dir: "{{ report\_dir }}/build"**

16|

17| - name: confirm/create report directory 18| file:

19| path: "{{ report\_dir }}"

20| state: directory 21|

**22| - name: delete old report build directory**

**23| file:**

**24| path: "{{ build\_dir }}"**

**25| state: absent 26|**

**27| - name: confirm/create new report build directory 28| file:**

**29| path: "{{ build\_dir }}"**

**30| state: directory**

31|

32|- name: Get facts from Junos device 33| hosts:

34| - all 35| roles:

36| - Juniper.junos 37| connection: local 38| gather\_facts: no 39|

40| tasks:

41| - name: get device facts 42| junos\_get\_facts:

43| host: "{{ ansible\_host }}" 44| register: junos\_facts

45|

46| - name: show device facts 47| debug:

48| var: junos\_facts

49| verbosity: 1 50|

51| - name: save device information using template 52| template:

53| src: template/device-facts.j2

**54| dest: "{{ hostvars.localhost.build\_dir }}/{{ inventory\_hostname }}.txt"**

55|

**56|- name: Assemble device report**

**57| hosts: localhost 58| connection: local 59| gather\_facts: yes 60|**

**61| vars:**

**62| systime: "{{ ansible\_date\_time.time | replace(':', '-') }}" 63| timestamp: "{{ ansible\_date\_time.date }}\_{{ systime }}"**

**64| report\_file: "{{ report\_dir }}/device-facts\_{{ timestamp }}.csv" 65|**

**66| tasks:**

**67| - name: assemble device files into new report 68| assemble:**

**69| src: "{{ build\_dir }}"**

**70| dest: "{{ report\_file }}"**

Lines 13 – 15 add a new fact, build\_dir, to hold the path to the report build directory.

Lines 22 – 30 delete and re-create the report build directory, using the pattern we have used in previous playbooks for other directories for temporary data.

Line 54 changes the template output directory to build\_dir from report\_dir.

Lines 56 – 70 add a third play to the playbook, run as localhost, to get the date and time from the localhost, create a timestamp, and assemble the device output into a single CSV file.

Lines 62 – 64 are similar to things we’ve seen in previous playbooks, setting vari- ables that will be used for filenames with timestamps.

Lines 67 – 70 are the play with the assemble module, concatenating the files in the report build directory (src argument) and creating a single CSV file (dest argument).

Run the playbook and review the CSV file:

mbp15:aja sean$ **ansible-playbook get-device-facts.yaml**

PLAY [Set up report directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [generate report directory names] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [generate report build directory name] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [confirm/create report directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [delete old report build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [confirm/create new report build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY [Get facts from Junos device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [show device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

skipping: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

PLAY [Assemble device report] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [assemble device files into new report] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=2 | changed=1 | unreachable=0 | failed=0 |
| localhost | : ok=7 | changed=3 | unreachable=0 | failed=0 |
| vsrx1 | : ok=2 | changed=1 | unreachable=0 | failed=0 |

mbp15:aja sean$ **ls -l ~/ansible/reports/**

total 40

drwxr-xr-x 4 sean staff 136 Oct 14 19:11 build

-rw-r--r-- 1 sean staff 213 Oct 14 19:11 device-facts\_2017-10-14\_19-11-11.csv

mbp15:aja sean$ **ls -l ~/ansible/reports/build/**

total 16

-rw-r--r-- 1 sean staff 109 Oct 14 19:11 bilbo.txt

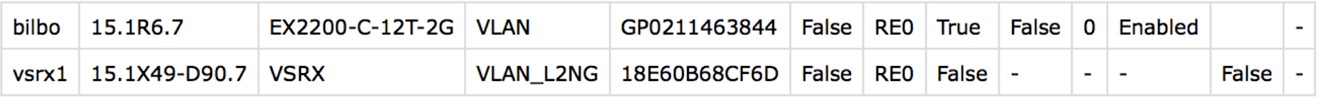
-rw-r--r-- 1 sean staff 104 Oct 14 19:11 vsrx1.txt

mbp15:aja sean$ **cat ~/ansible/reports/device-facts\_2017-10-14\_19-11-11.csv** "bilbo","15.1R6.7","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl ed","","-"

"vsrx1","15.1X49-D90.7","VSRX","VLAN\_L2NG","18E60B68CF6D","False","RE0","False","-","-","-

","False","-"

Great! Now take a look at the CSV file using a spreadsheet or using the preview feature of MacOS:



The only thing missing is the column headers. We can include those in another file, which we can copy into the report build directory before the assemble step. How- ever, the order of assembly becomes a consideration; we want the column headers to be the first of the files to be assembled, so the headers are the first row of the CSV file.

The UNIX-type systems that the author has worked with seem to present file lists in alphabetical order by filename, including the list provided to the assemble mod- ule. As long as the filename for the column headers will sort, alphabetically, before the first device’s filename, the column headers will come first. One approach is to name the column headers file something like AAA-column-headers.txt because a file- name starting with “AAA” should sort to the top of the list (remember to use capi- tal “A” not lower-case "a” because UNIX sorts are case sensitive and “A” precedes “a”).

The author uses a slightly different approach, but his approach may not work for everyone. The underscore ( \_ ) is easily typed, commonly used in filenames, and has no other meaning to UNIX (many other symbols which can be used in file- names also have meaning to the shell, and thus need to be escaped when entered as part of a command). The underscore character ( \_ ) also precedes the lower-case letter “a” when sorted, at least in the standard US sort order. The author prefixes the filename for his column names with an underscore.

Unfortunately, the underscore comes *after* a capital “A” when sorted, so this nam- ing convention will work correctly only if all the devices’ output filenames – which really means all the inventory hostnames – start with lower-case letters. If you are using capital first letters for your devices, you may wish to use the “AAA” prefix instead.

Let’s create the column headers file in the template directory; the playbook will copy it to the report build directory (keep in mind the build directory is a tempo- rary directory, which will be erased and recreated, so we cannot leave our column headers file there). Create file template/\_device-facts-columns.txt with the following single line of column names (may wrap to multiple lines in the book):

"Hostname","Junos version","Model","Switch Style","Serial Number","Dual RE","Master","VC Capable","VC Fabric","VC Master","VC Mode","SRX Cluster","SRX Cluster ID"

Add the “copy column headers file” task to the last play of the playbook:

...

56|- name: Assemble device report 57| hosts: localhost

58| connection: local 59| gather\_facts: yes 60|

61| vars:

62| systime: "{{ ansible\_date\_time.time | replace(':', '-') }}" 63| timestamp: "{{ ansible\_date\_time.date }}\_{{ systime }}"

64| report\_file: "{{ report\_dir }}/device-facts\_{{ timestamp }}.csv" 65|

66| tasks:

**67| - name: copy column headers file 68| copy:**

**69| src: template/\_device-facts-columns.txt 70| dest: "{{ build\_dir }}/"**

71|

72| - name: assemble device files into new report 73| assemble:

74| src: "{{ build\_dir }}"

75| dest: "{{ report\_file }}"

Run the playbook and examine the results:

mbp15:aja sean$ **ansible-playbook get-device-facts.yaml**

PLAY [Set up report directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [generate report directory name] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [generate report build directory name] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [confirm/create report directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [delete old report build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [confirm/create new report build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY [Get facts from Junos device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [show device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

skipping: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

PLAY [Assemble device report] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [copy column headers file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

TASK [assemble device files into new report] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [localhost]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=2 | changed=1 | unreachable=0 | failed=0 |
| localhost | : ok=8 | changed=4 | unreachable=0 | failed=0 |
| vsrx1 | : ok=2 | changed=1 | unreachable=0 | failed=0 |

mbp15:aja sean$ **tree ~/ansible/reports/**

/Users/sean/ansible/reports/

├── build

│ ├── \_device-facts-columns.txt

│ ├── bilbo.txt

│ └── vsrx1.txt

└── device-facts\_2017-10-16\_11-25-54.csv

1 directory, 4 files

mbp15:aja sean$ **cat ~/ansible/reports/device-facts\_2017-10-16\_11-25-54.csv**

"Hostname","Junos version","Model","Switch Style","Serial Number","Dual RE","Master","VC Capable","VC Fabric","VC Master","VC Mode","SRX Cluster","SRX Cluster ID"

"bilbo","15.1R6.7","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl ed","-","-"

"vsrx1","15.1X49-D90.7","VSRX","VLAN\_L2NG","18E60B68CF6D","False","RE0","False","-","-","-

","False","-"



Perfect!

## Device Configuration Based on Device Type – Base Settings 3

Junos configuration statements and options are remarkably consistent across dif- ferent types of devices, but there are some places where configuration require- ments diverge. One example is with configuring VLANs: MX and high-end SRX devices use a bridge domain command set, legacy EX and branch SRX devices use a VLAN command set, and newer EX and SRX devices use an ELS (Enhanced Layer-2 Software) command set. If you write a playbook to configure VLAN set- tings, the playbook will need to accommodate these differences.

For this chapter, we will use a simpler example: configuring the maximum number of SSH or NETCONF sessions allowed by a device. Most Junos devices allow large numbers of simultaneous management connections, and you can limit in the hundreds of simultaneous connections. For example, the author’s EX2200 will allow a limit as high as 250:

sean@bilbo> **configure**

Entering configuration mode

{master:0}[edit]

sean@bilbo# **set system services ssh connection-limit ?**

Possible completions:

<connection-limit> Maximum number of allowed connections (1..250)

{master:0}[edit]

sean@bilbo# **set system services ssh rate-limit ?**

Possible completions:

<rate-limit> Maximum number of connections per minute (1..250)

{master:0}[edit] sean@bilbo#

However, some Junos devices accept far smaller values for the maximum number of simultaneous connections. For example, the author’s VSRX will allow a limit only as high as 5:

sean@vsrx1> **configure**

Entering configuration mode

[edit]

sean@vsrx1# **set system services ssh connection-limit ?**

Possible completions:

<connection-limit> Maximum number of allowed connections (1..5) [edit]

sean@vsrx1# **set system services ssh rate-limit ?**

Possible completions:

<rate-limit> Maximum number of connections per minute (1..5) [edit]

sean@vsrx1#

1|---

Similarly, branch SRX devices, like SRX210 or SRX300, allow only 3 or 5 depend- ing on model (and possibly Junos version).

That’s a big difference. Limiting simultaneous connections, preferably to far less than 250, is a good way to mitigate the impact of some brute force or denial-of- service attacks, but limiting all devices to 3 is probably excessive.

Let’s update our base-settings.yaml playbook and base-settings.j2 template to in- clude connection limits for SSH and NETCONF. The basesettings.yaml playbook will gather facts from the devices and register them; these facts will be used by the template to set the appropriate values.

Add the boldfaced lines to the base-settings.yaml playbook as shown:

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "tmp"

12| conf\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}.conf" 13|

14| tasks:

**15| - name: get device facts 16| junos\_get\_facts:**

**17| host: "{{ ansible\_host }}"**

**18| register: jfact 19|**

**20| - name: show device facts 21| debug:**

**22| var: jfact**

**23| verbosity: 1**

24|

25| - name: confirm or create configs directory 26| file:

27| path: "{{ tmp\_dir }}"

28| state: directory 29|

30| - name: save device information using template 31| template:

32| src: template/base-settings.j2 33| dest: "{{ conf\_file }}"

34|

35| - name: install generated configuration file onto device 36| junos\_install\_config:

37| host: "{{ ansible\_host }}"

38| file: "{{ conf\_file }}"

39| timeout: 120

40| replace: yes

41| confirm: 10

42| comment: "playbook base-settings.yaml, commit confirmed" 43| notify: confirm commit

44|

45| # - name: delete generated configuration file 46| # file:

47| # path: "{{ conf\_file }}" 48| # state: absent

49|

50| handlers:

51| - name: confirm commit 52| junos\_commit:

53| host: "{{ ansible\_host }}"

54| timeout: 120

55| comment: "playbook base-settings.yaml, confirming previous commit"

The new tasks are copied with little modification from the get-device-facts.yaml playbook. Lines 15 – 18 gather device facts and assign them to the registered vari- able jfact; lines 20 – 23 display the device facts when the playbook is run in ver- bose mode for debugging.

To keep the logic for determining the connection limit fairly simple, we’ll use 3 for all branch SRX devices, even though some will support 5. We will use a limit of 5 for VSRX, and 10 for all other devices. Add or update the boldfaced lines in the template base-settings.j2:

1|#jinja2: lstrip\_blocks: True

**2|{# copy device facts into shorter variable names and lower-case the data #} 3|{% set model = jfact.facts.model.lower() %}**

**4|{% set personality = jfact.facts.personality|lower %} 5|**

**6|{#- Determine SSH connection-limit and rate-limit based on device facts #} 7|{% if model == 'vsrx' %}**

**8| {% set max\_ssh = 5 %}**

**9|{% elif personality == 'srx\_branch' %} 10| {% set max\_ssh = 3 %}**

**11|{% else %}**

**12| {% set max\_ssh = 10 %} 13|{% endif %}**

**14|**

**15|{#- Generate basic settings for the device #}**

16|system {

17| host-name {{ inventory\_hostname }}; 18| login {

19| user sean {

20| uid 2000;

21| class super-user;

22| authentication {

23| ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local) 24| }

25| }

26| }

27| replace:

28| name-server {

29| {% for server in aja\_host.dns\_servers %} 30| {{ server }};

31| {% endfor %}

32| }

33| services {

34| ftp;

35| delete: ftp;

36| netconf {

**37| ssh {**

**38| connection-limit {{ max\_ssh }};**

**39| rate-limit {{ max\_ssh }}; 40| }**

41| }

**42| ssh {**

**43| connection-limit {{ max\_ssh }};**

**44| rate-limit {{ max\_ssh }}; 45| }**

46| telnet;

47| delete: telnet;

48| web-management { 49| http;

50| }

51| delete: web-management; 52| }

53| replace:

54| ntp {

55| {% for ntp in aja\_site.ntp\_servers %} 56| server {{ ntp }};

57| {% endfor %}

58| }

59|}

60|snmp {

61| description "{{ aja\_host.snmp\_description }}"; 62| location "{{ aja\_host.snmp\_location }}";

63|}

Lines 2, 6, and 15 are comments – note the {#...#} delimiters. These lines are es- sentially ignored by Jinja2, but they help anyone reading the template understand what is happening.

Note the extra hyphen after the opening of the comment on lines 6 and 15 – the comments start with {#- instead of just {#. As we discussed earlier in this chapter,

the added hyphen suppresses the newline from the previous line. In this template, that serves to suppress the blank lines on lines 5 and 14 – the vertical white space helps us understand the template by visually separating three “sections” of the template, but we do not need the blank lines in the output.

Lines 3 and 4 copy variables from the device facts into template variables whose names are shorter and thus easier to work with if used repeatedly in the template. At the same time, these lines convert to lower-case the text from the device facts before assigning it to the template variables. Two different approaches are shown

– line 3 calls the lower() method of Python’s string class, while line 4 uses a Jinja2 filter.

String comparisons are case-sensitive – “Hello” is different from “hello” – and the author prefers to have strings in a known case before making comparisons, unless preserving the original case is important. For this template, keeping the original case is not important. Ensuring everything is lower-case means we need not be concerned that some devices might return their model as all capitals while others might use mixed case, or that a change in PyEZ might alter the case of the personal- ity fact from “SRX\_BRANCH” to “SRX\_branch.”

Lines 7 – 13 are the logic for determining the maximum sessions. We discussed *if- else* control structures earlier in this chapter, but here we added an *elif* (“else if”) element. The *elif* statement adds another comparison to an *if-else* structure. When the condition of *if* is false, the condition of *elif* will be evaluated – if true, the subse- quent lines will be in the template output; if false, move on to the next *elif* state- ment (you can have more than one) or the *else* statement.

Note the double equal sign in the conditions on lines 7 and 9. A single equal sign

( = ) is an *assignment*, copying the value on the right into the variable on the left. A double equal sign ( == ) is a *comparison* that will return Boolean *true* if the values on either side of the == are equal, *false* if the values are different.

Lines 37 – 45 add the additional connection-limit and rate-limit settings we want, using the max\_ssh value determined earlier in the template.

Run the playbook (not shown) and take a look at the resulting configuration files:

mbp15:aja sean$ **grep limit tmp/bilbo.conf**

connection-limit 10;

rate-limit 10;

connection-limit 10;

rate-limit 10;

mbp15:aja sean$ **grep limit tmp/vsrx1.conf**

connection-limit 5;

rate-limit 5;

connection-limit 5;

rate-limit 5;

Note that the configuration of each device has the correct values.

## References

PyEZ jnpr.junos.facts module:

<https://junos-pyez.readthedocs.io/en/2.1.7/jnpr.junos.facts.html>

(Note that the link is for the current version, 2.1.7, at the time this book was written.)

Jinja2 filters:

<http://jinja.pocoo.org/docs/2.9/templates/#builtin-filters>

Jinja2 whitespace control:

<http://jinja.pocoo.org/docs/2.9/templates/#whitespace-control>

# Chapter 11

Storing Private Variables – Ansible Vault

You probably have some confidential data that you wish to use in your playbooks, such as passwords, that should be kept encrypted for security. However, all the Ansible data sources we have discussed so far are plain-text. How can we create an encrypted data store that Ansible can read?

Ansible provides a solution with *Ansible vault*. In this chapter, we will discuss the ansible-vault command for creating vault files and editing vault data, and we will create and run playbooks that read vault files.

The early part of this chapter shows fundamental ideas. Later in this chapter we perform a practical example, building on the “base settings” playbook from earlier chapters.

## Creating a Vault File

To create a new vault file (create a new file encrypted with vault), use the com- mand ansible-vault create followed by the name of the new file to create. This will prompt you for a password, then open your default text editor (probably *vi* or *vim* unless you have altered the default on your system) so you can add data to the file. When you save the file and exit the editor, ansible-vault encrypts the data and saves the vault file.

Let’s create a new vault file called *vault1.yml*, which will contain variables for a playbook:

mbp15:aja sean$ **ansible-vault create vault1.yml**

New Vault password: **<enter password>**

Confirm New Vault password: **<re-enter password>**

You should now be in your text editor. Enter the following, then save the file and exit the editor:

---

vault\_test\_1: hello world

You should now be back at the command prompt.

The ansible-vault command does not care about contents of the files it encrypts, but if the file will contain variables for Ansible playbooks then you should use YAML format for the data in vault files.

If you view the vault file as text, you will see it contains mostly gibberish:

mbp15b:aja sean$ **cat vault1.yml**

$ANSIBLE\_VAULT;1.1;AES256 61633036343864616135356636323963613563363935323564636262373635336231616564623031

6234326361663036623831393432313033333236653231640a383830653263343335316332376164

36386632313332393130343733356435646536663837306337383236383133356165626433363439

6636306565356161360a653138613832643732653936303533343761316232343932353262623630

33303364653061313862613233623033313239653334656461383165633338306264

If you already have a file you wish to encrypt, you could use ansible-vault’s encrypt option instead of create. Create a text file called *vault2.yml* with the following data:

---

vault\_test\_2: goodbye cruel world

Now encrypt that file:

mbp15:aja sean$ **ansible-vault encrypt vault2.yml**

New Vault password: **<enter password>**

Confirm New Vault password: **<re-enter password>**

Encryption successful

mbp15:aja sean$ **cat vault2.yml**

$ANSIBLE\_VAULT;1.1;AES256 36656432653166666139626261373939313766376238373532386534333933306635343837383065

3038333233333230363635643232366563653662646232340a333664323136393738643535353533

33653236376435363838633762383065616232323831623937383731373230666664663262373731

6435336235383166320a363437306166393930643662376463303366666539633935646139303239

38386164613765316363336532333662396438303630316264303737383934316264343932653739

6539326639666633303439626561363531323139636264383031

## Viewing or Editing the Contents of a Vault File

The ansible-vault command has a view option that displays the (decrypted) con- tents of a vault file:

mbp15:aja sean$ **ansible-vault view vault1.yml**

Vault password: **<enter password>**

---

vault\_test\_1: hello world

There is also an edit option that will open the vault file’s (decrypted) contents in your system text editor (usually vim) so you can modify the file:

mbp15:aja sean$ **ansible-vault edit vault2.yml**

Vault password: **<enter password>**

*< File opens in text editor >*

NOTE Some versions of ansible-vault may require that you use the --ask-vault- pass option in order to prompt you for the vault password, for example:

ansible-vault view --ask-vault-pass vault1.yml

## Playbook That Reads a Vault File

Now let’s create a simple playbook that displays a variable from a vault file. Cre- ate file test-vault.yaml with the following content (line numbers added for discussion):

1|---

2|- name: Display variable from a vault 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - name: import vault data 10| include\_vars: vault1.yml 11|

12| - name: display variable 13| debug:

14| var: vault\_test\_1

There are two tasks in this playbook. Lines 12-14 are a typical debug command to display a variable.

Lines 9-10 import the data from the (decrypted) vault file. This is needed because our test vault file is not in a *standard* location for a data file, so Ansible will not import the data by default.

Now let’s run the playbook. We need to include the option --ask-vault-pass to tell the ansible-playbook command that it needs to prompt for a vault password:

mbp15:aja sean$ **ansible-playbook test-vault.yaml --ask-vault-pass**

Vault password: **<enter password>**

PLAY [Display variable from a vault] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [import vault data] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [display variable] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"vault\_test\_1": "hello world"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=2 changed=0 unreachable=0 failed=0

Observe the output from the debug task, showing the contents of the variable in the vault file.

The ansible-playbook command will not automatically prompt for a password if you forget the --ask-vault-pass option, resulting in a playbook failure when it can- not decrypt the vault file:

mbp15:aja sean$ **ansible-playbook test-vault.yaml**

PLAY [Display variable from a vault] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [import vault data] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* fatal: [localhost]: FAILED! => {"ansible\_

facts": {}, "changed": false, "failed": true, "message": "Decryption failed on /Users/sean/aja/ vault1.yml"}

to retry, use: --limit @/Users/sean/aja/test-vault.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=0 changed=0 unreachable=0 failed=1

## Considerations for Vault Passwords

With Ansible versions up to and including 2.3, you can specify only one vault password when running a playbook. If a playbook needs to read multiple vault files, all the vault files must use the same password.

Ansible version 2.4, released during the writing of this book, permits the user to specify multiple passwords, allowing a playbook to use several vault files with dif- ferent passwords. This is accomplished with the new-in-2.4 option --vault-id.

This book discusses and uses the options that work with Ansible 2.3 and earlier, in part to avoid problems for readers who cannot yet upgrade to 2.4. Readers who are using 2.4, and who do not require backwards compatibility, are encouraged to investigate the new --vault-id option.

The security of a vault may be increased by using longer passwords. Of course, longer passwords are more difficult to enter (correctly) when prompted. In addi- tion, the need to enter a password at a prompt means that a playbook that requires a vault file cannot be run on a scheduled or unattended basis.

Ansible supports the ability to put your vault password in a text file, and have the ansible-vault and ansible-playbook commands read that text file to get the pass- word. This permits the use of longer passwords and unattended execution of a playbook. However, the password file itself is a potential security risk. Should you use this approach, take steps to ensure the password file is readable only to autho- rized users, and consider making the password file hidden and putting it in a direc- tory separate from the Ansible playbooks and related files.

Use the --vault-password-file option with ansible-vault and ansible-playbook to provide the (location and) name of the file containing the vault password.

Create file ~/vault-pass.txt and put your password for the vault1.yml file into the new file. View the contents of the vault1.yml file using the password file:

mbp15:aja sean$ **ansible-vault view --vault-password-file=~/vault-pass.txt vault1.yml**

---

vault\_test\_1: hello world

Now run the playbook test-vault.yaml using the password file:

mbp15:aja sean$ **ansible-playbook test-vault.yaml --vault-password-file=~/vault-pass.txt**

PLAY [Display variable from a vault] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [import vault data] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [display variable] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"vault\_test\_1": "hello world"

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=2 changed=0 unreachable=0 failed=0

## Adding Passwords to Base Settings – Base Settings 4

Let’s add to our “base settings” playbook the password for the root account, and a second, read-only account with a static password to be used by network monitor- ing tools.

Junos stores most sensitive data, including passwords, in an encrypted form in the device’s configuration file. This example will store the encrypted passwords in the vault, and enter the encrypted passwords into the device configuration.

Storing already-encrypted passwords in a vault may seem to be redundant. How- ever, Junos prior to version 15.1 used MD5 for password hashing. MD5 is no lon- ger considered to be cryptographically secure. Putting MD5-hashed passwords in a vault adds a layer of security (ansible-vault uses SHA256 by default).

Even if you choose not to store your encrypted passwords in a vault, the pattern shown here will work for other sensitive data.

Earlier in this chapter we saw how to import vault data using an include\_vars task. There is a problem with this approach: the playbook references a variable that you cannot find unless you already know to decrypt the vault file. Pretend that you are updating a playbook written by someone else and encounter the variable reference "{{ root\_password }}" in the playbook or in a Jinja2 template referenced by the playbook. Where will you look for the definition of root\_password? If the root\_pass- word variable is defined in a vault file, you will not be able to locate it using the search feature of your text editor or the UNIX grep command. What then? With our first example, where the playbook is small and the variable is referenced in the playbook immediately after including the vault file, this may not seem to be a problem. However, as playbooks get larger and more complex, it gets more dif- ficult to see the connection.

For this second example, we will use a different approach, one proposed by An- sible as a best practice (see the link in the References section at the end of the chap- ter). We will modify one of our group\_vars entries, creating a vault file that Ansible will open automatically, thus eliminating the need for an include\_vars task. In addi- tion, our Jinja2 template will reference a variable that will be in a plain-text data file, but that variable will in turn reference a vault-encrypted variable in a file that will be easy to locate, because it will be in the same directory as the plain-text vari- ables file.

One downside to this approach: we need to provide the vault password even for playbooks that do not use the vault data, because the playbook needs to read the group\_vars data whether or not the playbook references vault-encrypted variables.

### Variable and Vault Files

Let’s start with the changes to our variables and adding the vault file. We currently have a single file in group\_vars for each group’s variables. However, Ansible will let you create a subdirectory in group\_vars for each group, and place multiple files con- taining data in the group’s directory. When running a playbook, Ansible will load the variables from all the files in the group’s subdirectory. This is the approach we will use for this example.

(Ansible supports a similar option for individual device variables – you can create a subdirectory within host\_vars for a device, and Ansible will read the data from all files in the device’s subdirectory. We will not do an example with a host data direc- tory, but you may find it useful if you need to save a lot of host-specific data.)

This example assumes all devices across the environment will share the same root and management account passwords, and thus we will modify the *all* group’s

variables. If your environment uses different credentials for different sites, you could modify the respective site-specific groups instead of the *all* group.

We currently have variables for all managed devices in file group\_vars/all.yaml. Create a directory group\_vars/all (which will correspond to the *all* group). Move the all.yaml variables file into the new directory, renaming it to vars.yaml:

mbp15:group\_vars sean$ **ls -l**

total 24

|  |  |  |
| --- | --- | --- |
| -rw-r--r--@ 1 sean | staff | 90 Nov 24 15:19 all.yaml |
| -rw-r--r--@ 1 sean | staff | 61 Sep 22 15:18 boston.yaml |
| -rw-r--r--@ 1 sean | staff | 61 Sep 22 15:20 sf.yaml |

mbp15:group\_vars sean$ **mkdir all** mbp15:group\_vars sean$ **mv all.yaml all/vars.yaml** mbp15:group\_vars sean$ **tree .**

.

├── all

│ └── vars.yaml

├── boston.yaml

└── sf.yaml

1. directory, 3 files

Now modify group\_vars/all/vars.yaml to include the new (boldface) lines:

---

ansible\_python\_interpreter: /usr/local/bin/python user\_data\_path: /Users/sean/ansible

**root\_hash: "{{ vault\_root\_hash }}"**

**monitor\_hash: "{{ vault\_monitor\_hash }}"**

Observe that these variables, in a plain-text variables file, reference other vari- ables. These two "vault\_" variables will be in a new vault file that we will create momentarily.

Because we want to store the hashed (encrypted) password, log into one of your test devices and copy the *root* password hash. (Change the *root* password first if you wish to do so.) If you have a mix of Junos versions, particularly 12.3 or earlier, copy the root password hash from a device running the oldest version to ensure you are getting a hash that will be compatible with all the Junos versions in use.

See the link in the references section at the end of the chapter for more details:

sean@vsrx1> **show configuration system root-authentication**

encrypted-password "$5$XbH2.66u$oYt2w3qsLqbxMYI.5.DUQFd88MYdamGG1VaKanPd6YB"; ## SECRET-DATA

Create file group\_vars/all/vault.yaml with the following variable definition, but use the appropriate password hash for your devices:

---

vault\_root\_hash: "$5$XbH2.66u$oYt2w3qsLqbxMYI.5.DUQFd88MYdamGG1VaKanPd6YB"

On one of your test devices, create a new account monitor and set its password. Show the change. You can roll back the change, we just needed the new hash:

sean@vsrx1> **configure**

Entering configuration mode

[edit]

sean@vsrx1# **set system login user monitor authentication plain-text-password**

New password:

Retype new password:

[edit]

sean@vsrx1# show | compare [edit system login]

+ user monitor {

+ authentication {

+ encrypted-password "$5$KgfRZFNQ$UrQvpUdXFJVwDrOzojJtIfNtJC1b.6g4jkMuVoFp tQ4"; ## SECRET-DATA

+ }

+ ## Warning: missing mandatory statement(s): 'class'

+ }

[edit]

sean@vsrx1# **rollback**

load complete

[edit] sean@vsrx1# **exit**

Exiting configuration mode

Copy that password hash and add it to group\_vars/all/vault.yaml:

---

vault\_root\_hash: "$5$XbH2.66u$oYt2w3qsLqbxMYI.5.DUQFd88MYdamGG1VaKanPd6YB" vault\_monitor\_hash: "$5$KgfRZFNQ$UrQvpUdXFJVwDrOzojJtIfNtJC1b.6g4jkMuVoFptQ4"

Save the vault.yaml file, then use ansible-vault encrypt to encrypt the file:

mbp15:aja sean$ **pwd**

/Users/sean/aja

mbp15:aja sean$ **ansible-vault encrypt --vault-password-file=~/vault-pass.txt group\_vars/all/vault. yaml**

Encryption successful

### Add Accounts to Base Settings Template

Now let’s update the template. Modify the template/base-settings.j2 template to include the following boldfaced lines. (Line numbers added. Only the relevant por- tion of the file is shown; do *not* delete lines not shown below):

16|system {

17| host-name {{ inventory\_hostname }};

**18| root-authentication {**

**19| encrypted-password "{{ root\_hash }}"; 20| }**

21| login {

**22| user monitor {**

**23| uid 2005;**

**24| class read-only;**

**25| authentication {**

**26| encrypted-password "{{ monitor\_hash }}"; 27| }**

**28| }**

29| user sean {

30| uid 2000;

31| class super-user;

32| authentication {

33| ssh-rsa "ssh-rsa AAAAB3NzaC1y...iIJeUsUJzS8b [sean@mbp15.local";](mailto:sean@mbp15.local) 34| }

35| }

36| }

Run the base-settings.yaml playbook:

mbp15:aja sean$ **ansible-playbook base-settings.yaml --ask-vault-pass --limit=vsrx1**

Vault password: **<enter password>**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [show device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [vsrx1]

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

TASK [save device information using template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [install generated configuration file onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

TASK [delete generated configuration file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

RUNNING HANDLER [confirm commit] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vsrx1 : ok=6 changed=4 unreachable=0 failed=0

Observe that no change to the playbook itself was needed. The additional vari- ables, the new vault file, and the additions to the template completed the changes.

## Decrypting the Vault

If your test devices are using non-production passwords, and thus security is not of

paramount importance, the author suggests decrypting the vault file with the root and monitor password hashes. This removes the need to enter the vault password when running playbooks in the remaining chapters of the book. If you wish to do this, follow these steps:

mbp15:aja sean$ **cd group\_vars/all/**

mbp15:all sean$ **ansible-vault decrypt vault.yaml**

Vault password: **<enter vault password>**

Decryption successful

mbp15:all sean$ **cat vault.yaml**

---

vault\_root\_hash: "$5$XbH2.66u$oYt2w3qsLqbxMYI.5.DUQFd88MYdamGG1VaKanPd6YB" vault\_monitor\_hash: "$5$KgfRZFNQ$UrQvpUdXFJVwDrOzojJtIfNtJC1b.6g4jkMuVoFptQ4"

If you do not wish to decrypt the vault, remember to use the --ask-vault-pass op- tion when running the playbooks in the remaining chapters.

## References

Vault instructions: <http://docs.ansible.com/ansible/latest/vault.html>

Vault variables best practices: [http://docs.ansible.com/ansible/latest/](http://docs.ansible.com/ansible/latest/playbooks_best_practices#variables-and-vaults) [playbooks\_best\_practices#variables-and-vaults](http://docs.ansible.com/ansible/latest/playbooks_best_practices#variables-and-vaults)

Junos hashing algorithms by version: [https://kb.juniper.net/InfoCenter/index?page=content&id=KB31903](https://kb.juniper.net/InfoCenter/index?page=content&amp;id=KB31903)

# Chapter 12 Roles

As your playbooks become more complicated, and as you create more templates to manage more aspects of a device’s configuration, you will want to organize your templates and tasks into logical units.

Ansible’s *roles* provide such a mechanism. Roles group tasks, templates, variables, and other files into a directory structure. When a role is included in a playbook, the tasks within the role execute as part of the playbook.

A given role may be included in more than one playbook. For example, you might want a playbook that updates only SNMP settings on your devices, but might also want to include those same SNMP settings in your “all settings” playbook. If you put the SNMP settings into a role, you can easily include that role in both your “all settings” playbook and your “update SNMP” playbook.

## Roles Directory and Files

Ansible will automatically look for roles in a *roles* subdirectory within the play- book’s directory. Each role is in an eponymously named subdirectory within the *roles* subdirectory; for example, the files related to a role named *snmp* would be in the subdirectory *roles/snmp*.

The directory for a role must contain one or more subdirectories, as needed for the role, with specific names understood by Ansible. In this chapter, we will discuss the following subdirectories, but Ansible supports a few others.

* + *tasks*: Tasks that will execute as part of the role.
  + *handlers*: Handlers that may be notified by tasks in the role or in the play (in the playbook).
  + *vars*: Variables that may be referenced by the playbook or the role.
  + *templates*: Jinja2 templates that may be used by the role.

The *tasks*, *handlers*, and *vars* subdirectories should each contain a *main.yml* file with the appropriate contents (the expected contents will become clear in our examples).

The *templates* subdirectory should contain one or more Jinja2 templates. The tem- plates’ filenames are not dictated by Ansible; the author suggests using descriptive names for the template files.

Create directory ~/aja/roles to contain the roles we create in this chapter:

mbp15:aja sean$ **mkdir roles**

## A Role for SNMP Settings

Let’s start by creating a role to generate configuration files for SNMP settings. In the next section of this chapter, we will create the playbook that will use the role.

It is the author’s experience that, at least initially, a role and playbook are devel- oped in parallel, with a lot of switching back-and-forth between role and play- book as the various files are developed. The linear format of a book makes it challenging to clearly represent the back-and-forth, so this chapter presents the role and playbook separately. However, in the following discussion about the role there are some “forward-looking” statements about the playbook, when the infor- mation is needed to understand the contents of the role and its files.

For purposes of this example role, assume that we need some common SNMP set- tings across all our devices, but that we have different SNMP community names for managing firewalls versus switches. The role will use different Jinja2 templates for the common settings and the different communities.

To start a role for SNMP settings, create directory ~/aja/roles/snmp: mbp15:aja sean$ **mkdir roles/snmp**

Our SNMP role will need tasks and templates; create the respective subdirectories:

mbp15:aja sean$ **mkdir roles/snmp/tasks**

mbp15:aja sean$ **mkdir roles/snmp/templates**

Create the following three Jinja2 templates in the roles/snmp/templates directory:

*File* snmp.j2 *for the common SNMP settings*:

#jinja2: lstrip\_blocks: True snmp {

description "{{ aja\_host.snmp\_description }}"; location "{{ aja\_host.snmp\_location }}"; contact ["netadmin@aja.com";](mailto:netadmin@aja.com)

{# the following lines ensure there will be no community public on a device #} community public;

delete: community public;

}

*File* community\_fw.j2 *for the firewall management community*:

#jinja2: lstrip\_blocks: True snmp {

replace: community aja\_fw { authorization read-only; clients {

192.168.1.100/32;

0.0.0.0/0 restrict;

}

}

}

*File* community\_sw.j2 *for the switch management community*:

#jinja2: lstrip\_blocks: True snmp {

replace: community aja\_sw { authorization read-only; clients {

192.168.2.200/32;

0.0.0.0/0 restrict;

}

}

}

Next, we must write the tasks that render the templates, thereby creating configu- ration files. To do this, we need to determine where the generated configuration files will be saved.

Because we are generating multiple configuration files (fragments) from our sev- eral template files, the fragments need to be assembled into a single configuration file before installing the assembled file on the device. The assembly step will be part of the playbook that uses the role. For this to work, we need a directory for each device to contain its assembled configuration file, with a “build” subdirec- tory to contain the configuration fragments to be assembled.

Assume the playbook defines a config\_assemble\_build variable containing the path to the “build” subdirectory. The tasks in the role that render the templates refer- ence the config\_assemble\_build variable, so the generated configuration fragments are placed in a location known to the playbook.

Create file roles/snmp/tasks/main.yml (line numbers added for discussion):

1|---

2|- name: common snmp settings 3| template:

4| src: snmp.j2

5| dest: "{{ config\_assemble\_build }}/snmp.conf"

6|

7|- name: firewall community 8| template:

9| src: community\_fw.j2

10| dest: "{{ config\_assemble\_build }}/snmp\_community\_fw.conf" 11| when: ('srx' in group\_names)

12|

13|- name: switch community 14| template:

15| src: community\_sw.j2

16| dest: "{{ config\_assemble\_build }}/snmp\_community\_sw.conf" 17| when: ('ex' in group\_names)

Lines 2 – 5 use Ansible’s template module, which we have seen previously, to render a configuration file from the snmp.j2 template created above. Notice we need to provide only the filename for the template (src) file, not the full path to the file; because the template and task are part of the same role, Ansible knows the correct path. (If a task references a template outside the role, a full path would be needed.) The destination for the completed configuration file needs both path and filename; the path is read from the config\_assemble\_build variable defined in the playbook.

Lines 7 – 11 build a configuration file from the community\_fw.j2 template. Because we want this configuration only when the device is a firewall, the when condition on line 11 tests to see if the current device is in group srx (more specifically, it tests to see if the string 'srx' is in the list of group\_names associated with the device).

Lines 13 – 17 are similar to lines 7 – 11, but for the community\_sw.j2 template need- ed only for switches.

The role should be complete. Do a quick check that you have the following files and directories for the role:

mbp15:aja sean$ tree roles/snmp/ roles/snmp/

├── tasks

│ ├── main.yml

└── templates

├── community\_fw.j2

├── community\_sw.j2

└── snmp.j2

1. directories, 4 files

## A Playbook for the SNMP Role

Now create playbook snmp-settings.yaml in your ~/aja directory (line numbers add- ed for discussion):

1|---

2|- name: Generate and Install Configuration File 3| hosts:

4| - all

5| roles:

6| - Juniper.junos 7| - snmp

8| connection: local 9| gather\_facts: no

10|

11| vars:

12| config\_assemble: "{{ user\_data\_path }}/config/{{ inventory\_hostname }}" 13| config\_assemble\_build: "{{ config\_assemble }}/build"

14| config\_file: "{{ config\_assemble }}/snmp.conf" 15| playbook\_name: snmp\_settings.yaml

16|

17| pre\_tasks:

18| - name: confirm or create device config directory 19| file:

20| path: "{{ config\_assemble }}" 21| state: directory

22|

23| - name: delete previous build directory 24| file:

25| path: "{{ config\_assemble\_build }}" 26| state: absent

27|

28| - name: create build directory 29| file:

30| path: "{{ config\_assemble\_build }}" 31| state: directory

32|

33| tasks:

34| - name: assemble config fragments 35| assemble:

36| src: "{{ config\_assemble\_build }}" 37| dest: "{{ config\_file }}"

38| notify: install config onto device 39|

40| handlers:

41| - name: install config onto device 42| junos\_install\_config:

43| host: "{{ ansible\_host }}"

44| file: "{{ config\_file }}"

45| timeout: 120

46| replace: yes

47| confirm: 10

48| comment: "playbook {{ playbook\_name }}, commit confirmed" 49| notify: confirm commit

50|

51| - name: confirm commit 52| junos\_commit:

53| host: "{{ ansible\_host }}"

54| timeout: 120

55| comment: "playbook {{ playbook\_name }}, confirming previous commit"

Lines 2 – 9 are the familiar start of a playbook that works with Junos devices, but with a notable addition. In the roles list, observe that we added the snmp role (line 7). This addition causes the play to, essentially, import and execute the tasks de- fined in the role.

NOTE Yes, the Galaxy modules we have been using, imported on line 6, are actually a role! The details of that role and its directories are outside the scope of this book. If you want to explore a little, follow the instructions in the "Dueling junos\_rpc Modules" section of Chapter 5 to locate the directory where ansible- galaxy installed the role, and look in the Juniper.junos/library subdirectory.

Lines 11 – 15 define some variables for the play, which are also available to roles imported by the play. Note in particular line 13, the config\_assemble\_build variable that we referenced in the role’s tasks (in file roles/snmp/tasks/main.yml).

Line 17 introduces something new, a pre\_tasks section of the play. Ansible imports and executes the tasks defined in a role *before* it executes the tasks defined in the tasks section of the play that imports the role. However, we need to create the di- rectories that will receive the generated configuration files before we ask the role to generate those files. The pre\_tasks section of the play defines tasks that should be executed first, before the role’s tasks are executed.

Lines 18 – 31 define three pre\_tasks that ensure we have a directory in which to assemble our SNMP configuration, and a clean “build” subdirectory to place the configuration fragments created by the role. This is a pattern we have used in other examples in this book.

Lines 33 – 38 define the play task that assembles the configuration fragments into a single configuration file, then notifies the handler that will install that configura- tion file onto a Junos device. Note the references to variables defined on lines 13 and 14 of the playbook.

Finally, lines 40 – 55 define two handlers. The first installs a configuration file on a Junos device using a “commit confirmed” and the second confirms the commit.

Notice that a handler can notify another handler to run (line 49).

Now let’s run the playbook. As the playbook runs, observe the order of the differ- ent tasks; you can see how the play’s pre\_tasks run first, followed by the snmp role’s tasks, and finally the play’s task and handlers:

mbp15:aja sean$ **ansible-playbook snmp-settings.yaml**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [confirm or create device config directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [delete previous build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [create build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [snmp : common snmp settings] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [snmp : firewall community] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

changed: [vsrx1]

TASK [snmp : switch community] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [vsrx1]

changed: [bilbo]

TASK [assemble config fragments] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

RUNNING HANDLER [install config onto device] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

RUNNING HANDLER [confirm commit] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=8 changed=7 unreachable=0 failed=0

vsrx1 : ok=8 changed=7 unreachable=0 failed=0

Look at the files generated by the playbook. The hierarchy should be similar to the following, adjusted for your home directory and device names:

mbp15:aja sean$ **tree ~/ansible/config**

/Users/sean/ansible/config

├── bilbo

│ ├── build

│ │ ├── snmp.conf

│ │ └── snmp\_community\_sw.conf

│ └── snmp.conf

└── vsrx1

├── build

│ ├── snmp.conf

│ └── snmp\_community\_fw.conf

└── snmp.conf

4 directories, 6 files

View the contents of the snmp.conf file, and the files in the build directory, for one or more of the devices and notice how the fragments are assembled.

TIP As you run the playbook repeatedly, you will find it stops processing devices after the "assemble config fragments" task. Delete the assembled snmp.conf files in the devices’ directories created by the previous run of the playbook before running the playbook again (for example, rm ~/ansible/config/bilbo/snmp.conf).

If you are currently thinking “we could have written this playbook without creat- ing the SNMP role and all its files,” you are correct, but this was a fairly simple example. As we increase the complexity of the examples through the rest of the chapter, it should become clearer how roles become building blocks that help us build complicated playbooks and provide flexibility while minimizing duplicate code.

## Moving Setup Tasks and Handlers into a Role

Take another look at the snmp-settings.yaml playbook. Notice that there is very lit- tle in this playbook that is specific to SNMP information. That playbook could be copied to, for example, login-settings.yaml, and, by replacing the snmp role with a login role (assuming such a role has been created), and changing the values as- signed to the playbook\_name and config\_file variables, the new file would become a playbook for updating login settings.

However, doing so would require duplicate code between the snmp-settings.yaml and login-settings.yaml playbooks, particularly the pre\_tasks and handlers sections of the files. Code sometimes needs to be changed. Why maintain the same code in two different files? Why not move that code into a role?

Create a new role config\_setup\_commit with the following files and directories:

roles/config\_setup\_commit/

├── handlers

│ └── main.yml

├── tasks

│ └── main.yml

└── vars

└── main.yml

---

File roles/config\_setup\_commit/handlers/main.yml contains the following (copied from the handlers section of snmp-settings.yaml, extra indentation removed, and variables used for confirm and timeout values):

* name: install config onto device junos\_install\_config:

host: "{{ ansible\_host }}"

file: "{{ config\_file }}" timeout: "{{ commit\_timeout }}" replace: yes

confirm: "{{ confirm\_time }}"

comment: "playbook {{ playbook\_name }}, commit confirmed {{ confirm\_time }}" notify: confirm commit

* name: confirm commit junos\_commit:

host: "{{ ansible\_host }}" timeout: "{{ commit\_timeout }}"

comment: "playbook {{ playbook\_name }}, confirming previous commit"

---

File roles/config\_setup\_commit/tasks/main.yml contains the following (copied from the pre\_tasks section of snmp-settings.yaml and extra indentation removed):

* name: confirm or create device config directory file:

path: "{{ config\_assemble }}" state: directory

* name: delete previous build directory file:

path: "{{ config\_assemble\_build }}" state: absent

* name: create build directory file:

path: "{{ config\_assemble\_build }}" state: directory

File roles/config\_setup\_commit/vars/main.yml contains the following (the first two variables were copied from the vars section of the playbook; the last two variables correspond with the new variable references in the handlers):

---

config\_assemble: "{{ user\_data\_path }}/config/{{ inventory\_hostname }}" config\_assemble\_build: "{{ config\_assemble }}/build"

commit\_timeout: 120

confirm\_time: 10

1|---

Now delete the pre\_tasks and handlers sections of the snmp-settings.yaml playbook, and delete the two variables that were moved into the role. Also add the new role to the roles list (the order of the list is important, for reasons we will discuss momentarily):

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos

**7| - config\_setup\_commit**

8| - snmp

9| connection: local 10| gather\_facts: no 11|

12| vars:

13| config\_file: "{{ config\_assemble }}/snmp.conf" 14| playbook\_name: snmp\_settings.yaml

15|

16| tasks:

17| - name: assemble config fragments 18| assemble:

19| src: "{{ config\_assemble\_build }}" 20| dest: "{{ config\_file }}"

21| notify: install config onto device

Because we have not changed the templates, in order to see the full effects of the playbook, roll back the previous change on your test devices and delete the assem- bled snmp.conf files from the last playbook run.

Run the playbook again:

mbp15:aja sean$ **ansible-playbook snmp-settings.yaml**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [config\_setup\_commit : confirm or create device config directory] \*\*\*\*\*\*\*\*\* ok: [bilbo]

ok: [vsrx1]

TASK [config\_setup\_commit : delete previous build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [config\_setup\_commit : create build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [snmp : common snmp settings] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [snmp : firewall community] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

changed: [vsrx1]

TASK [snmp : switch community] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [vsrx1]

changed: [bilbo]

TASK [assemble config fragments] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

RUNNING HANDLER [config\_setup\_commit : install config onto device] \*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

RUNNING HANDLER [config\_setup\_commit : confirm commit] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=8 changed=7 unreachable=0 failed=0

vsrx1 : ok=8 changed=7 unreachable=0 failed=0

Observe again the order of the tasks. You can see the tasks from the config\_setup\_ commit role run first, followed by the tasks from the snmp role, then the remaining task from the playbook itself, and finally the handlers from the config\_setup\_commit role.

How did the playbook know to run the tasks from the config\_setup\_commit role be- fore the tasks from the snmp role? Because that was the order of the roles list in the playbook. Roles are evaluated in order.

The Juniper.junos role needs to be imported before the config\_setup\_commit role, because config\_setup\_commit relies on modules that are part of Juniper.junos. If you reverse their order, you’ll get an error similar to this:

ERROR! no action detected in task. This often indicates a misspelled module name, or incorrect module path.

The error appears to have been in '/Users/sean/aja/roles/config\_setup\_commit/handlers/main.yml': line 2, column 3, but may

be elsewhere in the file depending on the exact syntax problem. The offending line appears to be:

---

* name: install config onto device

^ here

How did the handlers from config\_setup\_commit run separately, later than, the tasks from the same role? Handlers run *when notified*, not when loaded. The handlers were imported into the play when the tasks were run, but they were not executed until the first handler was notified by the “assemble config fragments” task in the playbook.

## Adding a System Role and Playbook

Let’s define a second role, one that creates settings for the Junos system hierarchy (except the login hierarchy), and a new playbook to apply that role.

Create the following directories and files:

roles/system/

├── tasks

│ └── main.yml

└── templates

└── system.j2

---

File roles/system/tasks/main.yml contains:

* name: system settings template:

src: system.j2

dest: "{{ config\_assemble\_build }}/system.conf"

File roles/system/templates/system.j2 contains the following, mostly copied from our earlier base-settings.j2 template:

#jinja2: lstrip\_blocks: True

{# copy device facts into shorter variable names and lower-case the data #}

{% set model = jfact.facts.model.lower() %}

{% set personality = jfact.facts.personality|lower %}

{#- Determine SSH connection-limit and rate-limit based on device facts #}

{% if model == 'vsrx' %}

{% set max\_ssh = 5 %}

{% elif personality == 'srx\_branch' %}

{% set max\_ssh = 3 %}

{% else %}

{% set max\_ssh = 10 %}

{% endif %}

{#- Generate basic settings for the device #} system {

host-name {{ inventory\_hostname }};

**domain-name aja.com;**

**domain-search [ aja.com aja.net ];**

root-authentication {

encrypted-password "{{ root\_hash }}";

}

replace:

name-server {

{% for server in aja\_host.dns\_servers %}

{{ server }};

{% endfor %}

}

services { ftp;

delete: ftp; netconf {

ssh {

connection-limit {{ max\_ssh }}; rate-limit {{ max\_ssh }};

}

}

ssh {

connection-limit {{ max\_ssh }}; rate-limit {{ max\_ssh }};

}

telnet;

delete: telnet; web-management {

http;

}

delete: web-management;

}

replace:

ntp {

{% for ntp in aja\_site.ntp\_servers %} server {{ ntp }};

{% endfor %}

}

}

Now copy the snmp-settings.yaml playbook to system-settings.yaml and change or add the boldfaced lines in the new playbook (line numbers added for discussion):

1|---

2|- name: Generate and Install Configuration File 3| hosts:

4| - all 5| roles:

6| - Juniper.junos

7| - config\_setup\_commit

**8| - system**

9| connection: local 10| gather\_facts: no 11|

12| vars:

**13| config\_file: "{{ config\_assemble }}/system.conf" 14| playbook\_name: system\_settings.yaml**

15|

**16| pre\_tasks:**

**17| - name: get device facts 18| junos\_get\_facts:**

**19| host: "{{ ansible\_host }}"**

**20| register: jfact**

21|

22| tasks:

23| - name: assemble config fragments 24| assemble:

25| src: "{{ config\_assemble\_build }}" 26| dest: "{{ config\_file }}"

27| notify: install config onto device

Line 8 imports the new system role, replacing the snmp role.

Line 13 generates the name of the configuration file to be applied to the devices, which should be changed from snmp.conf to system.conf.

Line 14 documents the name of the playbook during commits.

Lines 16 – 20 add a pre\_task to gather system facts, copied from our playbook bas- esettings.yaml. The template for the system hierarchy needs certain facts about the device in order to set correct values. This was not needed for the SNMP settings playbook because the SNMP templates did not rely on gathering information from the device.

(The fact gathering could have been added to the system role itself as a task, and would have worked fine in this example. However, what would happen if we later create another role that needs the same data? Do we run the fact gathering in both roles, duplicating code, and duplicating effort if we use both roles in the same playbook? Putting the fact gathering in the playbook avoids this concern.)

Run the new playbook:

mbp15:aja sean$ **ansible-playbook system-settings.yaml**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [config\_setup\_commit : confirm or create device config directory] \*\*\*\*\*\*\*\*\* ok: [bilbo]

ok: [vsrx1]

TASK [config\_setup\_commit : delete previous build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [config\_setup\_commit : create build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [system : system settings] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [assemble config fragments] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

RUNNING HANDLER [config\_setup\_commit : install config onto device] \*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

RUNNING HANDLER [config\_setup\_commit : confirm commit] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=8 changed=6 unreachable=0 failed=0

vsrx1 : ok=8 changed=6 unreachable=0 failed=0

Observe the order of the tasks. Take a look at the generated files. Note how little we needed to change in the playbook itself to obtain very different results.

## Building an “all settings” Playbook

Now let’s build a playbook to configure all settings for which we have roles. At present, that means only the snmp and system roles, but this can easily be expanded to include additional roles.

Copy the system-settings.yaml file to all-settings.yaml and add the snmp role. (The order, relative to each other, of the snmp and system roles in the roles list is not im- portant, as they do not depend on each other.) Also update the two playbook vari- ables as shown:

1|---

2|- name: Generate and Install Configuration File 3| hosts:

4| - all

5| roles:

6| - Juniper.junos

7| - config\_setup\_commit

**8| - snmp**

9| - system

10| connection: local 11| gather\_facts: no 12|

13| vars:

**14| config\_file: "{{ config\_assemble }}/all.conf" 15| playbook\_name: all\_settings.yaml**

16|

17| pre\_tasks:

18| - name: get device facts 19| junos\_get\_facts:

20| host: "{{ ansible\_host }}"

21| register: jfact 22|

23| tasks:

24| - name: assemble config fragments 25| assemble:

26| src: "{{ config\_assemble\_build }}" 27| dest: "{{ config\_file }}"

28| notify: install config onto device

Run the playbook. (In order for the playbook to confirm the commit you will need to roll back the last change or two on your test devices. Otherwise, the applied configuration will be unchanged from what is already on the device. The author rolled back changes on *bilbo*, but not on *vsrx1*; observe the differences between the devices in the output for the two handlers):

mbp15:aja sean$ **ansible-playbook all-settings.yaml**

PLAY [Generate and Install Configuration File] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get device facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [config\_setup\_commit : confirm or create device config directory] \*\*\*\*\*\*\*\*\* ok: [bilbo]

ok: [vsrx1]

TASK [config\_setup\_commit : delete previous build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [config\_setup\_commit : create build directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

changed: [vsrx1]

TASK [snmp : common snmp settings] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [snmp : firewall community] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [bilbo]

changed: [vsrx1]

TASK [snmp : switch community] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* skipping: [vsrx1]

changed: [bilbo]

TASK [system : system settings] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

TASK [assemble config fragments] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

changed: [bilbo]

RUNNING HANDLER [config\_setup\_commit : install config onto device] \*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

changed: [bilbo]

RUNNING HANDLER [config\_setup\_commit : confirm commit] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=10 changed=8 unreachable=0 failed=0

vsrx1 : ok=9 changed=6 unreachable=0 failed=0

Observe the order of the tasks. Take a look at the generated configuration files.

Consider how little we needed to change the playbook itself to create a new play- book that sets both SNMP and System settings (and any other roles we may add in the future). This is the power of using roles!

## References

Ansible’s Roles reference:

<http://docs.ansible.com/ansible/latest/playbooks_reuse_roles.html>

# Chapter 13 Repeating Tasks

Sometimes a playbook needs to be able to repeat a task several times. This chapter discusses two situations where repeating a task is useful and how to accomplish it.

The first is automatically retrying a task that failed. This is most likely to be useful for tasks that might fail because a device was temporarily unreachable, or its con- figuration was temporarily locked thereby blocking the playbook from changing the configuration.

The second is repeating a task for each element in a list. The list could come from a number of sources – a data file, results from querying a device, etc. – but the gen- eral idea is you want to repeat some task for each element.

## Re-trying a Failed Task

Ansible has the ability to retry a task that failed. You can specify how many times to retry the task, how long to wait between attempts, and even a condition that can limit the types of failures that will cause the task to be retried.

Create playbook get-version-galaxy.yaml with the following (line numbers added for discussion):

1|---

2|- name: Get Junos version 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| tasks:

11| - name: get junos version using galaxy module 12| junos\_rpc:

13| rpc: get-software-information 14| format: text

15| dest: "{{ inventory\_hostname }}-version.txt" 16| host: "{{ ansible\_host }}"

17| register: jversion

18| retries: 2

19| delay: 15

20| until: jversion | success 21|

22| - name: display junos\_rpc result 23| debug: var=jversion

We used Juniper’s junos\_rpc module in Chapter 5, so most of the playbook’s con- tents are already familiar. The RPC get-software-information is the equivalent of the Junos CLI command "show version."

Lines 18 – 20 are new. These lines together tell Ansible how it should retry the task should it fail.

Line 18, the retries argument, specifies the number of additional times the task may be retried after the initial failure. In this example, we are asking for up to two retries, or three attempts total.

Line 19, the delay argument, specifies how many seconds to wait between a failure and the (next) retry.

Line 20, the until option, is the looping construct. In programming, a *do-until* or *until* loop is a loop that repeats until some condition is true (while the condition is false). The condition shown here, jversion | success, uses the success filter on the jversion registered variable to return *true* if the task succeeded, or *false* if the task failed. So, line 20 tells Ansible to repeat the junos\_rpc task until either it succeeds (the jversion | success condition returns *true*) or the maximum number of retries have been attempted.

Disconnect one of your test devices – the author disconnected *bilbo* – and run the playbook:

mbp15:aja sean$ **ansible-playbook get-version-galaxy.yaml**

PLAY [Get Junos version] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get junos version using galaxy module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FAILED - RETRYING: get junos version using galaxy module (2 retries left). changed: [vsrx1]

FAILED - RETRYING: get junos version using galaxy module (1 retries left).

fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "unable to connect to 198.51.100.5: ConnectRefusedError(198.51.100.5)"}

TASK [display junos\_rpc result] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"jversion": {

"attempts": 1,

"changed": true, "check\_mode": false, "failed": false, "kwargs": {},

"rpc": "get-software-information"

}

}

to retry, use: --limit @/Users/sean/aja/get-version-galaxy.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=0 changed=0 unreachable=0 failed=1

vsrx1 : ok=2 changed=1 unreachable=0 failed=0

Observe that *vsrx1* succeeded, but that *bilbo* failed. During TASK [get junos version using galaxy module] you can see two lines starting with "FAILED – RETRYING" – these lines show that Ansible is retrying the task for a device (unfortunately, the message does not say which). Only after the retries do we see the failure message for *bilbo*:

fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "unable to connect to 198.51.100.5: ConnectRefusedError(198.51.100.5)"}

Notice that the failure message includes "attempts": 2 showing that the task was

attempted several times.

Run the playbook again, but this time reconnect your test device right after the first "FAILED – RETRYING" message appears:

mbp15:aja sean$ **ansible-playbook get-version-galaxy.yaml**

PLAY [Get Junos version] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get junos version using galaxy module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FAILED - RETRYING: get junos version using galaxy module (2 retries left). changed: [vsrx1]

changed: [bilbo]

TASK [display junos\_rpc result] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"jversion": {

"attempts": 2, "changed": true, "check\_mode": false, "failed": false, "kwargs": {},

"rpc": "get-software-information"

}

}

ok: [vsrx1] => {

"jversion": {

"attempts": 1, "changed": true, "check\_mode": false, "failed": false, "kwargs": {},

"rpc": "get-software-information"

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

vsrx1 : ok=2 changed=1 unreachable=0 failed=0

Because connectivity to the device was restored after the first failure, this time we see a single "FAILED – RETRYING" line followed by a successful result for *bilbo*.

Retrying a failed task is an Ansible feature, not specific to Juniper’s Galaxy mod- ules. The following playbook, get-version-core.yaml, shows the same retry options used with Ansible’s core junos\_rpc module:

1|---

2|- name: Get Junos version 3| hosts:

4| - all

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - name: get junos version using ansible core module 10| junos\_rpc:

11| rpc: get-software-information 12| output: text

13| provider:

14| host: "{{ ansible\_host }}" 15| register: jversion

16| retries: 2

17| delay: 15

18| until: jversion | success 19|

20| - name: display junos version output 21| debug: var=jversion

### Refining the Until Condition

Our get-version-galaxy.yaml playbook will retry the junos\_rpc task on any failure of that task. That’s fine if the failure is something that might correct itself in a short time, but if the failure is unlikely to be fixed, retrying the task several times does not make much sense.

Fortunately, Juniper’s junos\_rpc module returns error messages that indicate the type of failure it is experiencing, and we can create an until condition that checks the error message. Note again the failure message from bilbo when we left it dis- connected for the entire playbook run:

fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "unable to con- nect to 198.51.100.5: ConnectRefusedError(198.51.100.5)"}

The phrase ConnectRefusedError suggests the nature of the problem, which in our forced test was a disconnected network cable. Under more realistic conditions, the device might have reached the maximum number of NETCONF sessions or there may have been some other, possibly temporary, problem.

...

But what if the error message had contained “ConnectAuthError” indicating an authentication failure? If the login credentials on the device are different from what we are using, that problem is not likely to fix itself in the next few minutes, so trying to connect again is unlikely to prove useful.

Let’s modify the until condition in the get-version-galaxy.yaml playbook to avoid retrying the task if we get an authentication failure, while still retrying the task for any other failure:

20| until: (jversion | success) or (jversion.msg.find("ConnectAuthError") >= 0)

...

There’s a lot going on in this condition, so let’s break it down.

The expression (jversion | success) is the same condition we had before, applying the success filter to the registered variable jversion and returning a Boolean value indicating whether or not the task succeeded. The parentheses ensure this expres- sion is evaluated as a unit, before being considered by the **or** operator that follows.

The **or** operator takes two Boolean values, from the expressions on either side of the **or**, and returns a single Boolean value. If either of the two expressions are *true*, then **or** determines that the entire condition is *true*. If both of the expressions are *false*, then the entire condition is *false*.

The expression (jversion.msg.find("ConnectAuthError") >= 0) checks the message for the phrase ConnectAuthError and returns *true* if found, *false* otherwise. Again, the surrounding parentheses ensure this expression is evaluated as a unit, before being considered by the **or** operator. Let’s break this expression down even further:

find("string") is a function that searches for *string* in the variable on which find is called. Thus, jversion.msg.find("ConnectAuthError") searches for the string Connect- AuthError in the variable jversion.msg.

If find() locates the string, it returns the location in the variable where the string started, where the first character in the variable is location 0, the next character is location 1, etc. However, if find() cannot locate the string in the variable, it returns

-1 to indicate that the string was not found.

The final part of the expression, >= 0, tests the number returned by find() to see if it is zero or greater. If this test returns *true*, then the string ConnectAuthError was found in the variable jversion.msg. However, if the >= 0 test returns *false*, it means that find() returned -1 because the string was not found in the variable.

Putting it all together, if the task succeeds, or if the task fails with a message con- taining “ConnectAuthError,” the until condition is *true* and Ansible moves on to the next task in the playbook. On the other hand, if the task fails with some other message, the until condition is *false* and Ansible repeats the task, if there are re- tries left.

TIP To help understand the results of find(), run the following simple play- book called show-find.yaml and observe the results. Try changing the variable definition on line 9 and the find() tests on lines 12 and 14.

1|---

2|- name: Illustrating the find function 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| vars:

9| my\_string: Hello World 10|

11| tasks:

12| - debug: msg={{ my\_string.find("World") }} 13|

14| - debug: msg={{ my\_string.find("Apple") }}

MORE? Ansible also has Boolean operators **and** and **not**. Ansible’s Boolean operators **or** and **and** use short-circuit evaluation. See the References at the end of the chapter for more information about these topics. Be grateful for the short- circuit evaluation of Boolean expressions, because without it our until condition would need to be even longer, something like this:

(jversion | success) or ((jversion | failed) and (jversion.msg.find("ConnectAuthError") >= 0))

Let’s run our updated get-version-galaxy.yaml playbook with one test device dis- connected; the author unplugged the network cable for *bilbo*. This run should look similar to what we saw previously:

mbp15:aja sean$ **ansible-playbook get-version-galaxy.yaml**

PLAY [Get Junos version] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get junos version using galaxy module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* FAILED - RETRYING: get junos version using galaxy module (2 retries left). changed: [vsrx1]

FAILED - RETRYING: get junos version using galaxy module (1 retries left).

fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "unable to connect to 198.51.100.5: ConnectRefusedError(198.51.100.5)"}

TASK [display junos\_rpc result] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1] => {

"jversion": {

"attempts": 1, "changed": true, "check\_mode": false, "failed": false, "kwargs": {},

"rpc": "get-software-information"

}

}

to retry, use: --limit @/Users/sean/aja/get-version-galaxy.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=0 changed=0 unreachable=0 failed=1

vsrx1 : ok=2 changed=1 unreachable=0 failed=0

Because the failure for *bilbo* is not an authentication problem, Ansible retries the task twice.

Re-connect your test device.

Now let’s force an authentication failure so we can see how the playbook’s behav- ior changes. The author disabled his account on his *vsrx1* firewall, as follows:

root@vsrx1> **configure**

Entering configuration mode

[edit]

root@vsrx1# **deactivate system login user sean**

[edit]

root@vsrx1# **show | compare**

[edit system login]

! inactive: user sean { ... }

[edit]

root@vsrx1# **commit confirmed and-quit**

commit confirmed will be automatically rolled back in 10 minutes unless confirmed commit complete

Exiting configuration mode

# commit confirmed will be rolled back in 10 minutes root@vsrx1>

Run the playbook again:

mbp15:aja sean$ **ansible-playbook get-version-galaxy.yaml**

PLAY [Get Junos version] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TASK [get junos version using galaxy module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

fatal: [vsrx1]: FAILED! => {"attempts": 1, "changed": false, "msg": "unable to connect to 192.0.2.10: ConnectAuthError(192.0.2.10)"}

changed: [bilbo]

TASK [display junos\_rpc result] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"jversion": {

"attempts": 1, "changed": true, "check\_mode": false, "failed": false, "kwargs": {},

"rpc": "get-software-information"

}

}

to retry, use: --limit @/Users/sean/aja/get-version-galaxy.retry

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

vsrx1 : ok=0 changed=0 unreachable=0 failed=1

The attempt to communicate with *vsrx1* failed, as expected, but notice there were no retries – the test for “ConnectAuthError” worked!

TIP Ansible normally stops processing a device after a failure. That remains true after retrying a device as shown above. However, depending on the task, if it is important enough to retry, you may want the playbook to continue processing a device after a failure. You can tell Ansible to ignore errors in a particular task, and thus continue processing a failed device, using the ignore\_errors: yes option on the task.

## Repeating a Task Based on a List

In Chapter 8 we talked about a *for* loop in a Jinja2 template, and showed an ex- ample using a list of DNS server IP addresses. This section shows how to do some- thing similar in a playbook using Ansible’s with\_items option.

The examples in this section use XML data from Junos devices. We discussed XML briefly in Chapter 5. Previously when we used an RPC to query a device for data we requested the response in text format, because that is easier for humans to understand. This time, we request data in XML format because we are going to have Ansible process it for us, and it is easier to select a specific datum from a data set when the data set is in XML format instead of text format.

interfaces { ge-0/1/0 {

TIP If you ever need to process data in text format, the author strongly encourages you to become familiar with *regular expressions*.

Assume we want to query the LLDP neighbor data from our switches, with the in- tention of using the name of the neighbor to create a description of the local inter- face on our target device. However, assume that our switches may have connected to them a number of IP phones or other devices that provide LLDP data, but that we do not care about documenting. Thus, we wish to limit our interface descrip- tions to known “uplink” interfaces which connect to other network devices.

In other words, we want to use a device’s LLDP neighbor data to create a configu- ration for the device’s uplink interfaces similar to this:

description "to device strider port ge-0/0/5";

}

ge-0/1/1 {

description "to device frodo port ge-0/1/1.0";

}

}

In order to keep our discussion in this chapter focused on the looping construct and XML query, we will stop short of uploading the interface descriptions to the device; the reader can do this as an exercise, following the examples from earlier chapters.

This is the LLDP neighbor data from the author’s switch *bilbo*:

sean@bilbo> show lldp neighbors

|  |  |  |  |
| --- | --- | --- | --- |
| Local Interface | Parent Interface Chassis Id | Port info | System Name |
| ge-0/1/1.0 | - 78:fe:3d:3d:f6:40 | ge-0/1/1.0 | frodo |
| ge-0/0/9.0 | - 88:a2:5e:69:ef:14 | ge-0/0/0.0 | elrond |
| ge-0/1/0.0 | - f4:cc:55:24:84:00 | ge-0/0/5 | strider |

---

The uplink interfaces are ge-0/1/0 and ge-0/1/1, so the names *frodo* and *strider* are of interest. The device *elrond* connected to ge-0/0/9 is not of interest.

Add a list called uplinks containing the interface names for uplink interfaces to one or more of your test devices’ host\_vars files, within the aja\_host dictionary. The author is updating his host\_vars/bilbo.yaml file as follows (boldfaced lines):

ansible\_host: 198.51.100.5 aja\_host:

dns\_servers:

- 5.7.9.11

- 5.7.9.12

- 5.7.9.13

snmp\_description: EX2200-C for testing snmp\_location: "Sean's home office" **uplinks:**

**- ge-0/1/0**

**- ge-0/1/1**

### LLDP as XML

The two LLDP playbooks we create in the following pages repeat certain tasks for each element (interface) in the aja\_host.uplinks list.

Let’s start by getting a feel for what the LLDP data looks like in XML format. Log into one of your test devices and run the following command:

sean@bilbo> **show lldp neighbors | display xml**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/12.3R12/junos">](http://xml.juniper.net/junos/12.3R12/junos)

<lldp-neighbors-information junos:style="brief">

<lldp-neighbor-information>

<lldp-local-interface>ge-0/1/1.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/1/1.0</lldp-remote-port-description>

<lldp-remote-system-name>frodo</lldp-remote-system-name>

</lldp-neighbor-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/0/9.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>88:a2:5e:69:ef:14</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/0/0.0</lldp-remote-port-description>

<lldp-remote-system-name>elrond</lldp-remote-system-name>

</lldp-neighbor-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/1/0.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>f4:cc:55:24:84:00</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/0/5</lldp-remote-port-description>

<lldp-remote-system-name>strider</lldp-remote-system-name>

</lldp-neighbor-information>

</lldp-neighbors-information>

<cli>

<banner>{master:0}</banner>

</cli>

</rpc-reply>

Notice all the LLDP data is contained within element lldp-neighbors-information, within which is an element lldp-neighbor-information for each neighbor. Within each lldp-neighbor-information element is a series of elements describing the neigh- bor or the local interface to which the neighbor connects.

We want the lldp-remote-system-name elements, but only for the desired local inter- faces. We can see the lldp-local-interface element identifies the local interface to which the neighbor is connected.

The RPC we need Ansible to call is get-lldp-neighbors-information, which you can confirm as follows:

sean@bilbo> **show lldp neighbors | display xml rpc**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/12.3R12/junos">](http://xml.juniper.net/junos/12.3R12/junos)

<rpc>

<get-lldp-neighbors-information>

</get-lldp-neighbors-information>

</rpc>

<cli>

<banner>{master:0}</banner>

</cli>

</rpc-reply>

We can also specify an interface, whether on the CLI or as part of the RPC request. We get a lot more detail about the neighbor, but the XML hierarchy is the same:

sean@bilbo> **show lldp neighbors interface ge-0/1/0 | display xml**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/12.3R12/junos">](http://xml.juniper.net/junos/12.3R12/junos)

<lldp-neighbors-information junos:style="detail">

<lldp-neighbor-information>

<lldp-index>3</lldp-index>

<lldp-ttl>120</lldp-ttl>

<lldp-timemark>Sat Jan 23 08:05:04 2016</lldp-timemark>

<lldp-age>21</lldp-age>

<lldp-local-interface>ge-0/1/0.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-local-port-id>526</lldp-local-port-id>

...

<lldp-local-port-ageout-count>0</lldp-local-port-ageout-count>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>f4:cc:55:24:84:00</lldp-remote-chassis-id>

<lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>

<lldp-remote-port-id>515</lldp-remote-port-id>

<lldp-remote-port-description>ge-0/0/5</lldp-remote-port-description>

<lldp-remote-system-name>strider</lldp-remote-system-name>

</lldp-neighbor-information>

</lldp-neighbors-information>

<cli>

<banner>{master:0}</banner>

</cli>

</rpc-reply>

### LLDP by Interface

Let’s start our first LLDP playbook by just getting the LLDP data in XML format. Create the following playbook get-lldp-interface.yaml:

1|---

2|- name: Get LLDP neighbor and save for configuring interface descriptions 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "{{ user\_data\_path }}/tmp"

12| lldp\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}-lldp.xml" 13|

14| tasks:

15| - name: confirm or create configs directory 16| file:

17| path: "{{ tmp\_dir }}"

18| state: directory 19|

20| - name: get lldp neighbor table 21| junos\_rpc:

22| rpc: get-lldp-neighbors-information 23| format: xml

24| host: "{{ ansible\_host }}" 25| dest: "{{ lldp\_file }}"

The variable on line 11 and the task on lines 15 – 18 ensure we have a temporary directory in which to store the results from the junos\_rpc module.

The variable on line 12 creates a unique filename for each device’s results.

Lines 20 – 25 call the junos\_rpc module. The format argument on line 23 requests that the module return the results as XML.

Run the playbook on your test device(s) which have LLDP data:

mbp15:aja sean$ **ansible-playbook get-lldp-interface.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

Display the results file. The data may not be indented nicely as it was on screen, but it should be recognizably the same data:

mbp15:aja sean$ **cat ~/ansible/tmp/bilbo-lldp.xml**

<lldp-neighbors-information style="brief">

<lldp-neighbor-information>

<lldp-local-interface>ge-0/1/1.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/1/1.0</lldp-remote-port-description>

<lldp-remote-system-name>frodo</lldp-remote-system-name>

</lldp-neighbor-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/0/9.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>88:a2:5e:69:ef:14</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/0/0.0</lldp-remote-port-description>

<lldp-remote-system-name>elrond</lldp-remote-system-name>

</lldp-neighbor-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/1/0.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>f4:cc:55:24:84:00</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/0/5</lldp-remote-port-description>

<lldp-remote-system-name>strider</lldp-remote-system-name>

</lldp-neighbor-information>

</lldp-neighbors-information>

Now let’s update the playbook to gather LLDP data for a single interface. Initially we will specify a single interface.

The RPC call requesting LLDP data for a specific interface changes name to get- lldp-interface-neighbors-information and adds an interface-name element (shown here in bold) as an argument:

sean@bilbo> **show lldp neighbors interface ge-0/1/0 | display xml rpc**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/12.3R12/junos">](http://xml.juniper.net/junos/12.3R12/junos)

<rpc>

<get-lldp-interface-neighbors-information>

**<interface-name>ge-0/1/0.0</interface-name>**

</get-lldp-interface-neighbors-information>

</rpc>

<cli>

<banner>{master:0}</banner>

</cli>

</rpc-reply>

...

Modify the get-lldp-interface.yaml playbook by changing or adding the boldfaced lines:

20| - name: get lldp neighbor table 21| junos\_rpc:

**22| rpc: get-lldp-interface-neighbors-information**

23| format: xml

**24| kwargs:**

**25| interface\_name: ge-0/1/0**

26| host: "{{ ansible\_host }}"

27| dest: "{{ lldp\_file }}"

Line 22 changes the name of the RPC call.

Lines 24 and 25 provide an argument to the RPC that specifies the interface name. The name kwargs is short for “key-word arguments” – this argument to junos\_rpc passes a dictionary of key:value arguments to the RPC. Notice that the hyphen in the RPC argument interface-name shown above is replaced with an underscore in the Ansible playbook. This does not seem to be required in newer versions of the junos\_rpc module but was necessary in earlier versions.

Run the playbook again and display the results. There should be data for a single interface, ge-0/1/0, in the output file.

mbp15:aja sean$ **grep -E "local|name" ~/ansible/tmp/bilbo-lldp.xml**

<lldp-local-interface>ge-0/1/0.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-local-port-id>526</lldp-local-port-id>

<lldp-local-port-ageout-count>0</lldp-local-port-ageout-count>

<lldp-remote-system-name>strider</lldp-remote-system-name>

### Looping Through Interfaces

Now let’s have the playbook loop through the aja\_host.uplinks list we created in our host\_vars file(s), executing the junos\_rpc task for each interface on the list. Modify the get-lldp-interface.yaml playbook by changing or adding the boldfaced lines:

...

20| - name: get lldp neighbor table 21| junos\_rpc:

22| rpc: get-lldp-interface-neighbors-information 23| format: xml

24| kwargs:

**25| interface\_name: "{{ item }}"**

26| host: "{{ ansible\_host }}"

27| dest: "{{ lldp\_file }}"

**28| with\_items: "{{ aja\_host.uplinks }}"**

Line 28, the with\_items option, tells Ansible that the task is a loop and should be repeated once for each element of the list in the aja\_host.uplinks variable. Ansible takes the first element from aja\_host.uplinks, puts it in a variable called item, and runs the task with that variable. When the task completes, Ansible takes the next element from the list, puts it in item, and runs the task again.

Line 25 provides the value of variable item, defined by the with\_items loop con- struct, to the keyword argument interface\_name. This causes the task to get LLDP data for a different interface on each iteration of the loop.

NOTE Querying the device several times for LLDP data for each interface may be inefficient. However, we start with this approach because it illustrates concepts that can be used elsewhere and gives us the opportunity to explore the tools needed to solve our problem. The second LLDP playbook, later in this chapter, illustrates an alternative approach that is probably more efficient for most devices.

MORE? Ansible offers several different "with\_*something*" loop variants, where *something* is a different type of data (technically, a different lookup func- tion). The References section contains a link to Ansible’s loop documentation.

Run the playbook:

mbp15:aja sean$ **ansible-playbook get-lldp-interface.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo] => (item=ge-0/1/0)

changed: [bilbo] => (item=ge-0/1/1)

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

Notice that the section TASK [get lldp neighbor table] shows it ran twice for *bilbo*, using two different values for item (two interface names). Perfect!

So why does the output file contain data only for the second interface?

mbp15:aja sean$ **grep -E "local|name" ~/ansible/tmp/bilbo-lldp.xml**

<lldp-local-interface>ge-0/1/1.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-local-port-id>528</lldp-local-port-id>

<lldp-local-port-ageout-count>0</lldp-local-port-ageout-count>

<lldp-remote-system-name>frodo</lldp-remote-system-name>

...

10| vars:

This is because our filename is unique for the *host*, but not for the *interface*. The second time Ansible runs the task it overwrites the file created the first time it ran the task. Let’s add the interface name to the filename, replacing the slash charac- ters with hyphens to keep the file system happy.

Modify the boldfaced lines of the playbook:

11| tmp\_dir: "{{ user\_data\_path }}/tmp"

**12| lldp\_file\_prefix: "{{ tmp\_dir}}/{{ inventory\_hostname }}-lldp-"**

13|

14| tasks:

15| - name: confirm or create configs directory 16| file:

17| path: "{{ tmp\_dir }}"

18| state: directory 19|

20| - name: get lldp neighbor table 21| junos\_rpc:

22| rpc: get-lldp-interface-neighbors-information 23| format: xml

24| kwargs:

25| interface\_name: "{{ item }}"

26| host: "{{ ansible\_host }}"

**27| dest: "{{ lldp\_file\_prefix }}{{ item | replace('/', '-') }}.xml"**

28| with\_items: "{{ aja\_host.uplinks }}"

Line 12 replaces the previous lldp\_file variable containing the full filename, with the variable lldp\_file\_prefix containing the start of the filename, to which we will append the interface name.

Line 27 completes the filename, using the item variable from the with\_items loop and the replace filter to replace slash characters with hyphens.

Run the playbook again and confirm that we get two output files:

mbp15:aja sean$ **rm ~/ansible/tmp/bilbo-lldp.xml**

mbp15:aja sean$ **ansible-playbook get-lldp-interface.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo] => (item=ge-0/1/0)

changed: [bilbo] => (item=ge-0/1/1)

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=1 unreachable=0 failed=0

mbp15:aja sean$ **ls -1 ~/ansible/tmp/**

bilbo-lldp-ge-0-1-0.xml bilbo-lldp-ge-0-1-1.xml

...

Confirm that each file contains the correct data for one interface:

mbp15:aja sean$ **grep -E "local|name" ~/ansible/tmp/bilbo-lldp-ge-0-1-0.xml**

<lldp-local-interface>ge-0/1/0.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-local-port-id>526</lldp-local-port-id>

<lldp-local-port-ageout-count>0</lldp-local-port-ageout-count>

<lldp-remote-system-name>strider</lldp-remote-system-name>

mbp15:aja sean$ **grep -E "local|name" ~/ansible/tmp/bilbo-lldp-ge-0-1-1.xml**

<lldp-local-interface>ge-0/1/1.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-local-port-id>528</lldp-local-port-id>

<lldp-local-port-ageout-count>0</lldp-local-port-ageout-count>

<lldp-remote-system-name>frodo</lldp-remote-system-name>

### XML and XPath

Now our playbook is saving LLDP data as XML for each interface that we care about. How do we extract from that XML data just the fields we want?

Ansible, starting in version 2.4, includes a module called *xml* that can perform var- ious tasks on XML data. With the xml module we can extract just the elements that we wish to use for our interface description by using *XPath*, a method of quickly navigating an XML hierarchy for specific data. The xml module can read the XML data from a file or can accept a string in a playbook argument; we use a file.

MORE?XML and XPath are big topics and we discuss them only superficially. The References section at the end of the chapter includes links for further exploration.

With XPath, you can specify the path to an element by listing all the different opening tags, starting from the root, connecting them with slash (‘/’) characters. For example, in our LLDP data, the path to a neighbor’s system name is:

/lldp-neighbors-information/lldp-neighbor-information/lldp-remote-system-name

The leading slash indicates we are specifying the path from the root of the XML hierarchy in question.

XPath offers a shortcut: a leading double-slash (‘//’) searches down the XML hier- archy for matches, regardless of how many levels deep they may be. For example, we could use the XPath path //lldp-remote-system-name instead of the full path above.

XPath can return multiple matches. Consider the following XML LLDP data:

<lldp-neighbors-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/1/1.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/1/1.0</lldp-remote-port-description>

<lldp-remote-system-name>frodo</lldp-remote-system-name>

</lldp-neighbor-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/0/9.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>88:a2:5e:69:ef:14</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/0/0.0</lldp-remote-port-description>

<lldp-remote-system-name>elrond</lldp-remote-system-name>

</lldp-neighbor-information>

<lldp-neighbor-information>

<lldp-local-interface>ge-0/1/0.0</lldp-local-interface>

<lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>

<lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>

<lldp-remote-chassis-id>f4:cc:55:24:84:00</lldp-remote-chassis-id>

<lldp-remote-port-description>ge-0/0/5</lldp-remote-port-description>

<lldp-remote-system-name>strider</lldp-remote-system-name>

</lldp-neighbor-information>

</lldp-neighbors-information>

There are three lldp-neighbor-information elements, each containing an lldp-remote- system-name element. Either of the XPath paths above would match all three of the lldp-remote-system-name elements. The path //lldp-remote-system-name could poten- tially find even more matches if there were other elements (not lldp-neighbor-infor- mation) which also contained lldp-remote-system-name child elements.

You can limit the matches to specific instances by using *predicates*, which specify a condition to match an element. Predicates are enclosed in square brackets – [ ] – and contain a match condition, an expression which must be *true* for a given ele- ment for that element to be included in the results.

For example, if we wanted to find the lldp-neighbor-information element which has an lldp-local-interface value of *ge-0/0/9.0*, we could use this XPath path with predicate:

//lldp-neighbor-information[lldp-local-interface='ge-0/0/9.0']

To return all child elements of the matching lldp-neighbor-information element, add the asterisk wildcard (\*) to the end of the path:

//lldp-neighbor-information[lldp-local-interface='ge-0/0/9.0']/\*

To return a single child element, add the element’s name to the end of the path:

//lldp-neighbor-information[lldp-local-interface='ge-0/0/9.0']/lldp-remote-system-name

There are a number of functions that can be used in predicates when you might not know an exact match. For example, if we do not know the unit number (the digit after the period) of the interface we are looking for, we can look for lldp-lo- cal-interface values which *start with* the known portion of the interface name by using the starts-with() function:

//lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-0/0/9')]/\*

The same approach can be used if you want to find all interfaces on the same PIC by simply leaving off the port number:

//lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-0/1/')]/\*

Use the following test-xml.yaml playbook to experiment with XPath and the xml

module:

1|---

2|- name: Experiment with Ansible's xml module 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: no 7|

8| tasks:

9| - name: get neighbor details 10| xml:

11| path: bilbo-lldp.xml

12| # xpath: //lldp-neighbor-information[lldp-local-interface='ge-0/0/9.0'] 13| # xpath: //lldp-neighbor-information[lldp-local-interface='ge-0/0/9.0']/\*

14| # xpath: //lldp-neighbor-information[lldp-local-interface='ge-0/0/9.0']/lldp-remote- system-name

15| # xpath: //lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-0/0/9')]/\* 16| xpath: //lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-0/1/')]/ lldp-remote-system-name

17| content: text

18| register: neighbors 19|

20| - debug: var=neighbors

The path argument (line 11) identifies the XML data file to read. You can use any of the LLDP .xml files created earlier in this chapter, just substitute the correct path and filename in the path argument.

The xpath argument (line 16 is the active example, lines 12 – 15 are commented out but provided for your experimentation) is the XPath path to search.

The content argument tells xml what data from the matching element to return; text

means the text contents of the element (the CDATA). One example run:

mbp15:aja sean$ **ansible-playbook test-xml.yaml**

PLAY [Experiment with Ansible's xml module] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get neighbor details] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"neighbors": {

"actions": {

"namespaces": {},

"state": "present",

"xpath": "//lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-0/1/')]/ lldp-remote-system-name"

},

"changed": false, "count": 2, "failed": false, "matches": [

{

"lldp-remote-system-name": "frodo"

},

{

"lldp-remote-system-name": "strider"

}

],

"msg": 2

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* localhost : ok=2 changed=0 unreachable=0 failed=0

Notice that the neighbors registered variable contains an "actions" dictionary that confirms the requested action, a "count" key showing the number of matches, and a "matches" list showing each match returned by the XPath path.

### Querying our LLDP Data

Now let’s add the XML XPath query to our get-lldp-interface.yaml playbook. Ap- pend the boldfaced lines:

...

20| - name: get lldp neighbor table 21| junos\_rpc:

22| rpc: get-lldp-interface-neighbors-information 23| format: xml

24| kwargs:

25| interface\_name: "{{ item }}"

26| host: "{{ ansible\_host }}"

27| dest: "{{ lldp\_file\_prefix }}{{ item | replace('/', '-') }}.xml" 28| with\_items: "{{ aja\_host.uplinks }}"

29|

**30| - name: get neighbor details 31| xml:**

**32| path: "{{ lldp\_file\_prefix }}{{ item | replace('/', '-') }}.xml" 33| xpath: //lldp-remote-system-name**

**34| content: text**

**35| with\_items: "{{ aja\_host.uplinks }}" 36| register: neighbors**

**37|**

**38| - debug: var=neighbors**

Lines 30 – 36 call the xml module and register the results in variable neighbors.

Line 32, the path argument, specifies the XML data file to query; notice the pro- vided path is the same as the dest argument for the junos\_rpc module (line 27). Be- cause it uses the item variable created by the with\_items loop (line 35), it will reference a different XML file (different interface) on each iteration of the task.

Line 33 is the XPath path expression. We do not need to worry about multiple matches here, as the XML file contains data from a single LLDP neighbor, so we do not need a predicate.

Run the playbook. The contents of the neighbors registered variable can be rather long and has been edited here:

mbp15:aja sean$ **ansible-playbook get-lldp-interface.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo] => (item=ge-0/1/0)

changed: [bilbo] => (item=ge-0/1/1)

TASK [get neighbor details] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => (item=ge-0/1/0)

ok: [bilbo] => (item=ge-0/1/1)

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"neighbors": { "changed": false,

"msg": "All items completed", "results": [

{

...

"actions": {

"namespaces": {},

"state": "present",

"xpath": "//lldp-remote-system-name"

...

},

"changed": false, "count": 1,

"item": "ge-0/1/0", "matches": [

{

"lldp-remote-system-name": "strider"

}

],

...

"msg": 1

},

{

"actions": {

"namespaces": {},

"state": "present",

"xpath": "//lldp-remote-system-name"

},

...

"changed": false, "count": 1,

"item": "ge-0/1/1", "matches": [

{

"lldp-remote-system-name": "frodo"

}

],

"msg": 1

}

]

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=4 changed=1 unreachable=0 failed=0

Notice that both TASK [get lldp neighbor table] and TASK [get neighbor details] re- peat once for each interface, as each task has a with\_items loop.

Note that the neighbors registered variable changed format somewhat from what we saw with the test-xml.yaml playbook. Most significantly, there is now a results key containing a list of results, each of which is a dictionary. This is thanks to the with\_items loop; the list allows the results of each iteration of the task to be record- ed. Within each dictionary in the results list, note there is an item key showing the value of the item variable for that iteration of the task. Each results dictionary also has a matches list containing each match found by XPath.

What if we wanted to return two child elements of each lldp-neighbor-information element? Say we want both the lldp-remote-system-name and the lldp-remote-port- description elements? We could use the asterisk wildcard to get all child elements, but there is also a way to specify multiple specific elements to match.

XPath can accept multiple paths separated by a pipe or vertical bar character (‘|’), something like:

*path* | *path* | *path*

The pipe character is essentially a logical *or* condition; if an element matches any of the paths it will be included in the results.

Modify line 33 of the get-lldp-interface.yaml playbook as shown:

...

30| - name: get neighbor details 31| xml:

32| path: "{{ lldp\_file\_prefix }}{{ item | replace('/', '-') }}.xml" **33| xpath: //lldp-remote-system-name | //lldp-remote-port-description 34| content: text**

**35| with\_items: "{{ aja\_host.uplinks }}" 36| register: neighbors**

...

...

Run the playbook again. The results should now include both elements for each LLDP neighbor:

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"neighbors": { "changed": false,

"msg": "All items completed", "results": [

{

...

"item": "ge-0/1/0", "matches": [

{

"lldp-remote-port-description": "ge-0/0/5"

},

{

"lldp-remote-system-name": "strider"

}

],

...

"msg": 2

},

{

"item": "ge-0/1/1", "matches": [

{

"lldp-remote-port-description": "ge-0/1/1.0"

},

{

"lldp-remote-system-name": "frodo"

}

],

}

}

...

"msg": 2

}

]

Nice!

### Using a Single RPC Call

One concern with the get-lldp-interface.yaml playbook is that it calls the junos\_rpc module once for every interface for which we want LLDP information. This could be rather inefficient if we wish to know about a large number of interfaces. We can change the playbook to make a single call to junos\_rpc to get the entire LLDP neighbor table. Can we use XPath expressions with predicates (see the “XML and XPath” section of this chapter) to extract only the interfaces of interest?

Enter the following playbook, get-lldp-list.yaml:

1|---

2|- name: Get LLDP neighbor and save for configuring interface descriptions 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "{{ user\_data\_path }}/tmp"

12| lldp\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}-lldp.xml" 13|

14| tasks:

15| - name: confirm or create configs directory 16| file:

17| path: "{{ tmp\_dir }}"

18| state: directory 19|

20| - name: get lldp neighbor table 21| junos\_rpc:

22| rpc: get-lldp-neighbors-information 23| format: xml

24| host: "{{ ansible\_host }}"

25| dest: "{{ lldp\_file }}" 26|

27| - name: get neighbor details 28| xml:

29| path: "{{ lldp\_file }}"

30| xpath: >

31| //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp- remote-system-name |

32| //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp- remote-port-description

33| content: text

34| with\_items: "{{ aja\_host.uplinks }}" 35| register: neighbors

36|

37| - debug: var=neighbors

Lines 12, 25, and 29 define or use a single filename for XML data for each device, as there is no need to have a separate file for each interface.

Lines 20 – 25 get the device’s entire LLDP neighbor table and store it in the XML file. Notice we no longer need a with\_items loop on this task.

Lines 27 – 35 query the saved XML data, but with XPath path expressions that include predicates to filter for a specific interface. Note the with\_items loop (line

34) and the reference to the item variable in the XPath predicates, [starts- with(lldp-local-interface, '{{ item }}')], on lines 31 and 32.

Because the xpath argument (lines 30 – 32) is rather long – it contains two XPath path expressions with predicates – it is spread across multiple lines using the “>” trick we saw in the get-partial-config.yaml playbook near the end of Chapter 9. Be sure to include the pipe character (‘|’) at the end of line 31.

Run the playbook:

mbp15:aja sean$ **ansible-playbook get-lldp-list.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

TASK [get neighbor details] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => (item=ge-0/1/0)

ok: [bilbo] => (item=ge-0/1/1)

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"neighbors": { "changed": false,

"msg": "All items completed", "results": [

{

...

"actions": {

"namespaces": {},

"state": "present",

"xpath": "//lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-

0/1/0')]/lldp-remote-system-name | //lldp-neighbor-information[starts-with(lldp-local- interface, 'ge-0/1/0')]/lldp-remote-port-description\n"

},

"changed": false, "count": 2,

...

"item": "ge-0/1/0", "matches": [

{

"lldp-remote-port-description": "ge-0/0/5"

},

{

"lldp-remote-system-name": "strider"

}

],

...

"msg": 2

},

{

"actions": {

"namespaces": {},

"state": "present",

"xpath": "//lldp-neighbor-information[starts-with(lldp-local-interface, 'ge-

0/1/1')]/lldp-remote-system-name | //lldp-neighbor-information[starts-with(lldp-local- interface, 'ge-0/1/1')]/lldp-remote-port-description\n"

},

"changed": false, "count": 2,

...

"item": "ge-0/1/1", "matches": [

{

"lldp-remote-port-description": "ge-0/1/1.0"

},

{

"lldp-remote-system-name": "frodo"

}

],

"msg": 2

}

]

}

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=4 changed=1 unreachable=0 failed=0

The output is similar to the previous playbook. Note that TASK [get lldp neighbor table] does not repeat (no loop) but TASK [get neighbor details] does.

Note that the xpath variables (neighbors.results[].actions.xpath) show the complete XPath path joined together from the two lines in the playbook.

### Two Templates for Interface Descriptions

Let’s create two templates that take the results of our XML query from the regis- tered variable neighbors and create Junos configuration snippets that assign de- scriptions to device interfaces. Why two templates? To illustrate two approaches, each of which has some benefits over the other.

First, modify the get-lldp-list.yaml playbook as shown (boldfaced lines):

1|---

2|- name: Get LLDP neighbor and save for configuring interface descriptions 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| vars:

11| tmp\_dir: "{{ user\_data\_path }}/tmp"

12| lldp\_file: "{{ tmp\_dir}}/{{ inventory\_hostname }}-lldp.xml"

**13| template\_dir: "template"**

14|

15| tasks:

16| - name: confirm or create configs directory 17| file:

18| path: "{{ tmp\_dir }}"

19| state: directory 20|

21| - name: get lldp neighbor table 22| junos\_rpc:

23| rpc: get-lldp-neighbors-information 24| format: xml

25| host: "{{ ansible\_host }}"

26| dest: "{{ lldp\_file }}" 27|

28| - name: get neighbor details 29| xml:

30| path: "{{ lldp\_file }}"

31| xpath: >

32| //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp- remote-system-name |

33| //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp- remote-port-description

34| content: text

35| with\_items: "{{ aja\_host.uplinks }}" 36| register: neighbors

37|

**38| - name: save interface descriptions, template 1 39| template:**

**40| src: "{{ template\_dir }}/int-desc-1.j2"**

**41| dest: "{{ tmp\_dir }}/{{ inventory\_hostname }}-{{ item.item | replace('/', '-') }}.conf" 42| with\_items: "{{ neighbors.results }}"**

Line 13 defines a new template\_dir variable specifying the directory where the tem- plates will be located.

Lines 38 – 42 use the template module to generate a configuration file based on a template file. Note the use of the template\_dir variable in the src argument.

The with\_items loop (line 42) references the registered variable neighbors, which contains the results of the xml module’s queries. More specifically, it references neighbors.results, the list of query results within the xml module’s results.

Our previous examples of with\_items all referenced a list defined in a host\_vars file, but that is not a requirement; any list can be used. Keep in mind also that the neigh- bors.results list is a list of dictionaries, which means the item variable for each it- eration of this loop will contain not a single value, but a dictionary of values.

The dest argument (line 41) saves the results in our temporary directory. (This is a convenience for the example, change this to a config build directory if you choose to extend the example to installing the configuration onto your devices.) Our first template saves a different config file for each interface, so the filename includes the interface name, which we find in the item.item variable.

That item.item reference may be a bit confusing. Look back at the results from the last time we ran the playbook. Notice that each dictionary in the neighbors.results list includes an item key containing the “current” interface name. Because the item variable in this task contains the current entry from the neighbors.results list, item. item in this task refers to the current neighbors.results.item value.

Now, let’s create our first template. Create, if needed, a template directory within your playbook directory:

mbp15:aja sean$ **mkdir template**

Create file int-desc-1.j2 in the template directory:

1|#jinja2: lstrip\_blocks: True 2|{% set neighbor\_name = '-' %} 3|{% set neighbor\_desc = '-' %} 4|{% for match in item.matches %}

5| {% if match.has\_key('lldp-remote-system-name') %}

6| {% set neighbor\_name = match['lldp-remote-system-name'] %} 7| {% endif %}

8| {% if match.has\_key('lldp-remote-port-description') %}

9| {% set neighbor\_desc = match['lldp-remote-port-description'] %} 10| {% endif %}

11|{% endfor %} 12|interfaces {

13| {{ item.item }} {

14| description "to device {{ neighbor\_name }} port {{ neighbor\_desc }}"; 15| }

16|}

This template will not work correctly, for reasons we will discuss shortly, but it would be a logical starting point.

Lines 2 and 3 declare a pair of variables to hold the neighbor’s hostname and interface description after we extract them from the item variable. They are declared at the top of the file to ensure they are valid; they should be updated with correct information in the loop on lines 4 – 11.

Lines 4 – 11 are a *for* loop that iterates over the item.matches list, the list of XPath matches from the XML data. Recall that each entry in the list is a single-item diction- ary (if needed, look back a few pages to the results from the last time we ran get-lldp- list.yaml). The two *if* statements (lines 5 – 7 and 8 – 10) test each entry to see whether it contains the key ‘lldp-remote-system-name’ or ‘lldp-remote-port-description’ and, when the appropriate key is found, assign the value to the appropriate variable.

Lines 12 – 16 are the Junos config snippet. Line 13 is the interface name from item. item, while line 14 creates the interface description from the variables declared on lines 2 and 3.

Run the updated get-lldp-list.yaml and check the resulting .conf files:

mbp15:aja sean$ **ansible-playbook get-lldp-list.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

TASK [get neighbor details] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => (item=ge-0/1/0)

ok: [bilbo] => (item=ge-0/1/1)

TASK [save interface descriptions, template 1] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=5 changed=3 unreachable=0 failed=0

mbp15:aja sean$ **cat ~/ansible/tmp/bilbo-ge-0-1-0.conf**

interfaces {

ge-0/1/0 {

description "to device - port -";

}

}

Where are the device name and interface? Why does the output still have the hy- phens from lines 2 and 3 of the template?

With Jinja2 templates, when you update a simple variable within a loop, the up- date exists only within the loop. The change to the variable does not survive after the loop ends. As a result, the assignments made on lines 6 and 9 of the template do not survive past the end of the loop on line 11. The original values from lines 2 and 3 are still valid, however, and are used on line 14.

Let’s revise our template to work around this situation. Modify template/int-desc-1.j2 as follows:

1|#jinja2: lstrip\_blocks: True

**2|{% set neighbor = {'name':'', 'desc':''} %}**

3|{% for match in item.matches %}

4| {% if match.has\_key('lldp-remote-system-name') %}

**5| {% if neighbor.update({'name': match['lldp-remote-system-name']}) %}{% endif %}**

6| {% endif %}

7| {% if match.has\_key('lldp-remote-port-description') %}

**8| {% if neighbor.update({'desc': match['lldp-remote-port-description']}) %}{% endif %}**

9| {% endif %} 10|{% endfor %} 11|interfaces {

12| {{ item.item }} {

**13| description "to device {{ neighbor.name }} port {{ neighbor.desc }}";**

14| }

15|}

Line 2 declares a variable neighbor that contains a dictionary, with key:value pairs to hold the LLDP neighbor’s name and description.

Lines 5 and 8 now update the neighbor dictionary when the appropriate key is found. This change will survive the *for* loop, but it needs to be done in a rather in- teresting way. The code neighbor.update({'name': match['lldp-remote-system-name']}) calls the update() function on the neighbor dictionary. This update() function is the Python function from the underlying Python dictionary implementation, not a Jin- ja2 function. Because this is not a Jinja2 function, we need to “trick” Jinja2 into running it, so we used the function call as the condition of an *if* statement. Jinja2

thinks it is evaluating an *if* condition, so it calls neighbor.update() for us, passing the updated dictionary entry {'name': match['lldp-remote-system-name']} as an argu- ment. Because the *if* statement is otherwise empty it does nothing else.

Now the updated description on line 13 should get the correct data. Note the change from underscore (‘\_’) to period in the variable references, because they are now referencing the name and desc keys of the neighbor dictionary.

Run the playbook again and confirm we get the desired results in the conf files:

mbp15:aja sean$ **cat ~/ansible/tmp/bilbo-ge-0-1-0.conf**

interfaces {

ge-0/1/0 {

description "to device strider port ge-0/0/5";

}

}

mbp15:aja sean$ **cat ~/ansible/tmp/bilbo-ge-0-1-1.conf**

interfaces {

ge-0/1/1 {

description "to device frodo port ge-0/1/1.0";

}

}

...

One nice thing about this first template is that the logic (the *set*, *for* and *if* state- ments) is all together, and the Junos configuration lines are all together, so it is easy to follow the logic and to envision the Junos configuration emerging from the template.

One downside to the approach used with this first template is that it generates a separate config file for each interface, and these files need to be assembled before being applied to the device’s configuration.

Can we create a single config file with all desired interfaces’ descriptions? Yes, we can, if we move the loop over the neighbors.results list from the playbook into the template.

Add a new task (boldfaced lines) to the end of the get-lldp-list.yaml playbook:

38| - name: save interface descriptions, template 1 39| template:

40| src: "{{ template\_dir }}/int-desc-1.j2"

41| dest: "{{ tmp\_dir }}/{{ inventory\_hostname }}-{{ item.item | replace('/', '-') }}.conf" 42| with\_items: "{{ neighbors.results }}"

43|

**44| - name: save interface descriptions, template 2 45| template:**

**46| src: "{{ template\_dir }}/int-desc-2.j2"**

**47| dest: "{{ tmp\_dir }}/{{ inventory\_hostname }}.conf"**

Notice that the new task does *not* include a loop.

Create the new template, template/int-desc-2.j2: 1|#jinja2: lstrip\_blocks: True

2|interfaces {

3|{% for result in neighbors.results %} 4| {{ result.item }} {

5| {% set neighbor = {'name':'', 'desc':''} %} 6| {% for match in result.matches %}

7| {% if match.has\_key('lldp-remote-system-name') %}

8| {% if neighbor.update({'name': match['lldp-remote-system-name']}) %}{% endif %} 9| {% endif %}

10| {% if match.has\_key('lldp-remote-port-description') %}

11| {% if neighbor.update({'desc': match['lldp-remote-port-description']}) %}{% endif %} 12| {% endif %}

13| {% endfor %}

14| description "to device {{ neighbor.name }} port {{ neighbor.desc }}"; 15| }

16|{% endfor %} 17|}

The new template uses a *for* loop (lines 3 – 16) to iterate over neighbors.results, re-

placing the with\_items: neighbors.results loop used with the earlier task. Because the loop is within the template, the template creates a single file with the descriptions for all interfaces.

The logic within the outer *for* loop is similar to the previous template; in fact, many lines are exactly the same. The most notable change is the result variable from the *for* loop (line 3) replaces the item variable from the playbook’s with\_items loop.

However, the Junos configuration lines get scattered through the template (lines 2, 4, 14, 15, and 17), making it a bit harder to follow the logic and to envision the config file that will emerge from the template. Let’s run the playbook and check the resulting config file:

mbp15:aja sean$ **ansible-playbook get-lldp-list.yaml --limit=bilbo**

PLAY [Get LLDP neighbor and save for configuring interface descriptions] \*\*\*\*\*\*\*

TASK [confirm or create configs directory] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

TASK [get lldp neighbor table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [bilbo]

TASK [get neighbor details] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => (item=ge-0/1/0)

ok: [bilbo] => (item=ge-0/1/1)

TASK [save interface descriptions, template 1] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

...

TASK [save interface descriptions, template 2] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=5 changed=1 unreachable=0 failed=0

mbp15:aja sean$ **cat ~/ansible/tmp/bilbo.conf**

interfaces {

ge-0/1/0 {

description "to device strider port ge-0/0/5";

}

ge-0/1/1 {

description "to device frodo port ge-0/1/1.0";

}

}

Nice! Just the way a Junos configuration should look. We could bring together the Junos lines of the template just a little bit – swap lines 2 and 3, and swap lines 16 and 17 – but the resulting configuration would not be quite as nicely formatted:

interfaces {

ge-0/1/0 {

description "to device strider port ge-0/0/5";

}

}

interfaces {

ge-0/1/1 {

description "to device frodo port ge-0/1/1.0";

}

}

Each of these templates is viable; select the approach you like best.

## References

Ansible playbook loops: <http://docs.ansible.com/ansible/latest/playbooks_loops.html>

Ansible error handling: <http://docs.ansible.com/ansible/latest/playbooks_error_handling.html>

Boolean expressions: <https://en.wikipedia.org/wiki/Boolean_expression>

Regular Expressions: <https://en.wikipedia.org/wiki/Regular_expression> <https://www.regular-expressions.info/>

Short-circuit evaluation: <https://en.wikipedia.org/wiki/Short-circuit_evaluation>

XML tutorial: <https://www.w3schools.com/xml/default.asp>

XPath tutorial: <https://www.w3schools.com/xml/xpath_intro.asp>

XPath functions: <https://www.w3schools.com/xml/xsl_functions.asp>

# Chapter 14

Custom Ansible Modules

There will come a time when you will want a playbook to do something for which an Ansible module does not yet exist. At that time, you may need to create a new, custom Ansible module to accomplish the task.

This chapter introduces the ideas behind writing custom modules for Ansible. See Ansible’s documentation link in the References section at the end of the chapter for more complete information about this topic.

Writing custom modules means programming, usually in Python. Ansible does not really care what programming language you use to write your module, but Ansible is written mostly in Python and includes a Python library to help with the interface between your module and the Ansible playbook that called it. As a result, writing modules in Python is usually the easiest option.

The discussion in this chapter assumes the reader is already familiar with Python

2.7 and PyEZ. The code is presented complete, not built in stages, and discussed at a fairly high level.

Note that the author uses a somewhat different development process than what Ansible proposes in their developer documentation, using their “hacking/test- module script.” However, the reader is encouraged to explore Ansible’s process as well, and use whichever approach you find works best.

## Functional Description of Our Custom Module

We will create a custom module to get the commit history from a Junos device, similar to the Junos CLI command show system commit, and both save the results to a file and return the results to the calling playbook. The RPC for getting this data is

get-commit-information.

The results of the custom module will differ from the results of calling the RPC using either Ansible’s core junos\_rpc module, or Juniper’s Galaxy junos\_rpc module, in two important ways:

* + The core module only returns data to the playbook, while the Galaxy module only saves data to a file. Our custom module will be able to do both.
  + Our module will return the data as a JSON dictionary for easy use by subse- quent tasks in the playbook, not as XML or text.
  + Our module will, optionally, return only the first *n* commits, where *n* is pro- vided by the playbook. Custom modules are good places for additional pro- cessing or filtering of data before returning the data to the playbook. With the junos\_rpc modules we get the complete output of the RPC; any additional pro- cessing would need to be done after the junos\_rpc task completed.

Our module should accept the following arguments from the playbook:

* + host (required). Normally set to ansible\_host by the playbook.
  + user (optional; default to username of current user).
  + passwd (optional; default to None to indicate public key authentication).
  + filename (optional). This is the (path and) filename of the output file. If omitted, no file will be saved.
  + max\_commits (optional; default to None to indicate all results). This integer repre- sents the maximum number of commits to return.

The functionality of the module will divide into two major areas:

* + A new class that handles the communication with the Junos device and subse- quent processing of the data retrieved from the device. The *Developing the Class* section of this chapter focuses on our new class.
  + The code that interfaces with Ansible and the playbook, and which instantiates an object of our new class. The *Creating the Ansible Module* section of this chapter discusses the interface code.

Ansible provides a Python library, containing the AnsibleModule class, that han- dles most of the interface work. This means the interface code looks very similar across different custom modules. Part of the functionality provided by the Ansible- Module class includes assigning default values to arguments that may not have been passed by the playbook, which means our class need not worry about missing arguments or assigning default values. We further discuss AnsibleModule in the *Creating the AnsibleModule* section of this chapter.

## Developing the Class

Let’s start by creating the JunosCommits class that will talk with the Junos device and process the results. Recall that Junos can tell us the RPC equivalent for a com- mand, and display the XML output for a command:

sean@vsrx1> **show system commit | display xml rpc**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">](http://xml.juniper.net/junos/15.1X49/junos)

<rpc>

<get-commit-information>

</get-commit-information>

</rpc>

<cli>

<banner></banner>

</cli>

</rpc-reply>

sean@vsrx1> **show system commit | display xml**

<rpc-reply [xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">](http://xml.juniper.net/junos/15.1X49/junos)

<commit-information>

<commit-history>

<sequence-number>0</sequence-number>

<user>sean</user>

<client>netconf</client>

<date-time junos:seconds="1510389588">2017-11-11 08:39:48 UTC</date-time>

<log>playbook base-settings.yaml, confirming previous commit</log>

</commit-history>

<commit-history>

<sequence-number>1</sequence-number>

<user>sean</user>

<client>netconf</client>

<date-time junos:seconds="1510389578">2017-11-11 08:39:38 UTC</date-time>

<comment>commit confirmed, rollback in 10mins</comment>

<log>playbook base-settings.yaml, commit confirmed</log>

</commit-history>

...

</commit-information>

<cli>

<banner></banner>

</cli>

</rpc-reply>

Our class needs to call the get-commit-information RPC on the Junos device, and ex- tract the desired information -- all the commit-history elements -- from the returned XML.

While developing a class that will be part of a custom Ansible module, the author has found it helpful to create a stand-alone program that tests the class. Executing the class from a stand-alone program, outside of Ansible, usually makes it easier to debug the class. There are a few reasons for this:

* The feedback that Python provides at the command-line is often better than what you get when the feedback is filtered through Ansible and a running play- book.
  + You can use a Python debugger with a stand-alone program.
  + Ansible modules should not print to the screen (we discuss why shortly), but printing variables and other data is often useful during module development and debugging.
  + A test program often makes it easier to change the test data being supplied to the class than would be the case with a complete playbook.

Our class, and the test program, will do everything we discussed in the *Functional Description of Our Custom Module* section of this chapter, but there is a small wrinkle with regard to saving the file containing the device’s commit history. An- sible’s developer documentation states:

“Don’t write to files directly; use a temporary file and then use the *atomic\_move* function from *ansible.module\_utils.basic* to move the updated temporary file into place. This prevents data corruption and ensures that the correct context for the file is kept.”[December 11, 2017: [http://docs.ansible.com/ansible/latest/dev\_guide/](http://docs.ansible.com/ansible/latest/dev_guide/developing_modules_best_practices.html#common-pitfalls) [developing\_modules\_best\_practices.html#common-pitfalls](http://docs.ansible.com/ansible/latest/dev_guide/developing_modules_best_practices.html#common-pitfalls)]

Because we will not use the AnsibleModule class in our test program, and the *atomic\_move* function is a method of the AnsibleModule class, our class will only create the temporary file. We will wait until the next section of this chapter, when we complete the custom module, to move the temporary file into its final location.

Create file test\_commit\_history.py with the following code (line numbers added for discussion):

1|#!/usr/bin/env python 2|

3|import sys 4|import tempfile

5|from jnpr.junos import Device 6|from pprint import pprint

7|

8|

9|######################################################################

10|

11|class JunosCommits(object):

12| """Get commit history from a Junos device.""" 13|

14| def init (self, host, gen\_file, username, password, max\_commits): 15| self.host = host

16| self.generate\_file = gen\_file 17| self.username = username

18| self.password = password

19| self.max\_commits = max\_commits 20|

21| # instantiate a PyEZ Device to communicate with the Junos device 22| self.dev = Device(host=self.host,

23| user=self.username,

24| passwd=self.password,

25| normalize=True)

26|

27| # list to store commit history 28| self.commits = []

29|

30| # number of commits returned by device 31| self.num\_commits = 0

32|

33| # path+filename and file descriptor for output tempfile 34| self.filespec = ''

35| self.file\_descriptor = None 36|

37| # ------------------------- #

38|

39| def get\_commit\_history\_from\_device(self):

40| """Get commit history from Junos device and store in list of dicts.""" 41| try:

42| self.dev.open()

43| except Exception as err:

44| msg = 'Error opening connection to Junos device: %s' % str(err) 45| raise Exception(msg)

46|

47| # get from device the equivalent of "show sytem commit" 48| try:

49| commit\_info = self.dev.rpc.get\_commit\_information() 50| except Exception as err:

51| msg = 'Error getting commit history from device: %s' % str(err) 52| raise Exception(msg)

53|

54| # extract all 'commit-history' elements from XML 55| # put data in a list of dictionaries

56| try:

57| commits\_xml = commit\_info.findall('commit-history') 58| self.num\_commits = len(commits\_xml)

59| for commit in commits\_xml:

60| commit\_dict = {

61| 'num': commit.findtext('sequence-number'),

62| 'user': commit.findtext('user'),

63| 'client': commit.findtext('client'),

64| 'date\_time': commit.findtext('date-time'),

65| 'comment': commit.findtext('log') 66| }

67| self.commits.append(commit\_dict) 68|

69| # truncate list if a max\_commits value was specified 70| if (self.max\_commits is not None) and \

71| (self.max\_commits < self.num\_commits):

72| del self.commits[self.max\_commits:] 73|

74| except Exception as err:

75| msg = 'Error processing commit history: %s' % str(err) 76| raise Exception(msg)

77|

78| # ------------------------- #

79|

80| def temp\_commit\_history\_file(self):

81| """Save commit history to temporary file.""" 82| try:

83| self.file\_descriptor, self.filespec = tempfile.mkstemp() 84| outfile = open(self.filespec, 'w')

85|

86| outfile.write('Device returned %s commits.\n' % self.num\_commits) 87| if self.max\_commits is not None:

88| outfile.write('Saving latest %s commits.\n' 89| % self.max\_commits)

90| outfile.write('\n- - - Commit History - - -\n') 91|

92| for c in self.commits:

93| line = '%2s: %s by %s via %s (%s)\n' % \

94| (c['num'], c['date\_time'], c['user'],

95| c['client'], c['comment']) 96| outfile.write(line)

97|

98| outfile.close() 99|

100| except Exception as err:

101| msg = 'Error writing to file %s: %s' % (self.filespec, str(err)) 102| raise Exception(msg)

103|

104| # ------------------------- #

105|

106| def run(self):

107| """Process Junos device."""

108| self.get\_commit\_history\_from\_device() 109| if self.generate\_file:

110| self.temp\_commit\_history\_file() 111|

112|

113|######################################################################

114|

115|def main():

116| """Test class JunosCommits.""" 117| host = 'vsrx1'

118| filename = 'commit-history.txt' 119| max\_commits = 4

120| user = 'sean' 121| password = None 122|

123| gen\_file = False if filename is None else True 124|

125| jc = JunosCommits(host, gen\_file, user, password, max\_commits) 126| try:

127| jc.run()

128| except Exception as err:

129| print str(err)

130| sys.exit(1) 131|

132| pprint(jc.commits)

133| print "Temporary file (if created) is %s" % jc.filespec 134|

135|

136|######################################################################

137|

138|if name == ' main ': 139| main()

Lines 11 – 110 define class JunosCommits, the class that gets the commit history from a Junos device. The class has four methods.

Lines 14 – 35 comprise the class’s init () method, called when we instantiate an object from the class. This method initializes our instance variables, including self.dev for the PyEZ Device object, and self.commits to contain the commit history.

Lines 39 – 76 comprise the class’s get\_commit\_history\_from\_device() method. As the name suggests, this method connects to the device (line 42), calls the get-commit- information RPC (line 49), and extracts the data into the self.commits list variable (lines 56 – 67). Each commit is put into a dictionary, and each dictionary is ap- pended to the list. If a max\_commits argument was provided, the list will be truncated so it contains only the desired number of commit entries (lines 70 – 72).

Lines 80 – 102 comprise the class’s temp\_commit\_history\_file() method, which saves the (truncated) commit history from the self.commits variable to a temporary file. The path and filename of the temporary file is stored in self.filespec.

Lines 106 – 110 are the class’s run() method, which provides a single method that the calling program can use to do everything needed by the object. This method just calls get\_commit\_history\_from\_device() and, when needed, temp\_commit\_history\_ file(), as those methods perform the functions needed for this class.

Lines 115 – 133 comprise the main() method that drives the testing of the class. This method declares variables equivalent to those that will be passed from the Ansible playbook (remember to change the username and, if needed, password, for your testing), instantiates an object of the JunosCommits class, and then calls the run() method on the object. This method also prints the object’s commits variable to display the commit history, and prints the filespec variable to display the path and filename of the temporary file (if any).

There are a few things to observe in this program:

* + Exception handling is used within the class to trap exceptions raised by PyEZ or file access. The class then adds a meaningful error message to the exception and raises a new exception to be caught by the calling routine, main().
  + Results that should be passed back to the Ansible playbook are collected in Python data structure(s), such as this example’s commits list. Because playbook results are converted to JSON format, try to use simple variables, lists and dic- tionaries.
  + The JunosCommits class does not print anything. During testing as part of a stand-alone program, like this one, you can add print statements to the class as needed for debugging, but remove all of them before creating the Ansible mod- ule. The Ansible module returns JSON data to the playbook on STDOUT, and print statements in the module are likely to interfere with the playbook’s inter- pretation of the results because the results are not valid JSON. (See also "Py- thon logging module" in the References at the end of the chapter.)

Now run the test program. Display the temporary file. Delete the temporary file when you are done:

mbp15:aja sean$ **python test\_commit\_history.py**

[{'client': 'netconf',

'comment': 'playbook base-settings.yaml, confirming previous commit', 'date\_time': '2017-12-11 15:15:17 UTC',

'num': '0',

'user': 'sean'},

{'client': 'netconf',

'comment': 'playbook base-settings.yaml, commit confirmed', 'date\_time': '2017-12-11 15:15:02 UTC',

'num': '1',

'user': 'sean'},

{'client': 'cli', 'comment': None,

'date\_time': '2017-12-11 15:14:33 UTC',

'num': '2',

'user': 'sean'},

{'client': 'netconf',

'comment': 'playbook base-settings.yaml, confirming previous commit', 'date\_time': '2017-12-11 15:09:26 UTC',

'num': '3',

'user': 'sean'}]

Temporary file (if created) is /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmp1iPZJg

mbp15:aja sean$ **cat /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmp1iPZJg**

Device returned 39 commits. Saving latest 4 commits.

- - - Commit History - - -

0: 2017-12-11 15:15:17 UTC by sean via netconf (playbook base-settings. yaml, confirming previous commit)

1: 2017-12-11 15:15:02 UTC by sean via netconf (playbook base-settings.yaml, commit confirmed) 2: 2017-12-11 15:14:33 UTC by sean via cli (None)

3: 2017-12-11 15:09:26 UTC by sean via netconf (playbook base-settings. yaml, confirming previous commit)

mbp15:aja sean$ **rm /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmp1iPZJg**

Change the variables defined in the main() method to get different results. For ex- ample, set filename to None to avoid generating a temporary file, and max\_commits to None to get a complete commit history.

## Creating the Ansible Module

Now let’s convert our test program into a custom Ansible module. This should re- quire no changes to the JunosCommits class, but will require some adjustments to the import statements and the main() method.

Ansible looks for modules in a few locations. Probably the easiest is the *library* subdirectory within the directory where the playbooks live. If you are using GitHub or other source control (see the Appendix), this location also keeps the new module within the project’s directory for inclusion in the source code repository.

Within your ~/aja directory, create a new library subdirectory, then copy the test\_ commit\_history.py program into the library subdirectory as commit\_history.py.

mbp15:aja sean$ **mkdir library**

mbp15:aja sean$ **cp test\_commit\_history.py library/commit\_history.py**

Open the library/commit\_history.py program in your text editor. Modify the import statements at the top of the program as shown (line numbers added for discussion):

1|#!/usr/bin/env python 2|

3|import os 4|import tempfile

5|from ansible.module\_utils.basic import AnsibleModule 6|from jnpr.junos import Device

7|

...

...

115|def main():

Line 5 imports the AnsibleModule class, which provides a lot of the interface be- tween the playbook and our new module.

When a playbook calls a module, Ansible assembles the arguments from the play- book into a JSON data set and passes the JSON data to the called module. The AnsibleModule class makes it easy for the module to parse the argument data.

Some of what AnsibleModule can do:

* Throw an error if required arguments are missing.
* Throw an error if an unexpected argument is provided.
* Assign default values for optional arguments that are not provided by the play- book.
* Confirm the data type of arguments (e.g., integer versus string). If no type is specified, arguments are assumed to be strings.

Let’s put AnsibleModule to work. Delete the current main() method and replace it with the following (line numbers added for discussion):

116| """Query Junos device and interface with Ansible playbook.""" 117| # define arguments from Ansible

118| module = AnsibleModule( 119| argument\_spec=dict(

120| host=dict(required=True),

121| filename=dict(required=False, default=None),

122| user=dict(required=False, default=os.getenv('USER')), 123| passwd=dict(required=False, default=None, no\_log=True),

124| max\_commits=dict(required=False, type='int', default=None) 125| )

126| )

127|

128| # copy playbook arguments into local variables 129| host = module.params['host']

130| filename = module.params['filename'] 131| username = module.params['user'] 132| password = module.params['passwd']

133| max\_commits = module.params['max\_commits'] 134|

135| # determine if module should generate output file 136| gen\_file = False if filename is None else True 137|

138| # instantiate JunosCommits and run

139| jc = JunosCommits(host, gen\_file, username, password, max\_commits) 140| try:

141| jc.run()

142| if gen\_file:

143| module.atomic\_move(jc.filespec, filename) 144| except Exception as err:

145| module.fail\_json(msg=str(err)) 146|

147| module.exit\_json(changed=False, commits=jc.commits)

...

Lines 118 – 126 instantiate an object module from the AnsibleModule class. The argument\_spec dictionary tells AnsibleModule about the arguments it should expect from the playbook.

Line 120 says we expect an argument called host. Because required=True, the host

argument must be provided; the module will throw an error if it is missing.

Line 121 says we might get an argument called filename. Because required=False, this argument is optional. Should filename not be provided, default=None automati- cally sets the value of filename to None.

Line 122 says we might (required=False) get an argument called user. This time the default value is determined by the function os.getenv('USER'), which reads the user- name from the operating system’s environment variables.

Line 123 says we might get an argument called password. Passwords are confiden- tial, and we do not want our password to appear in any Ansible logs. The option no\_log=True tells the module not to record this value in any log files.

Line 124 says we might get an argument called max\_commits. Should this argument be provided, type='int' says its value must be an integer. Other supported data types include 'str' (string, the default), 'bool' (Boolean), 'list' (list or array), and 'dict' (dictionary).

Lines 129 – 133 copy each of the arguments from the module.params dictionary into local variables. The AnsibleModule object module keeps all the arguments in a dic- tionary called params. It is often easier to work with the argument’s values if they are copied into local variables.

Line 136 sets Boolean gen\_file to False if no filename was provided (meaning the module should not create an output file), or True otherwise.

Line 139 instantiates object jc from our class JunosCommits, providing all the ar- guments needed for the class to do its work.

Line 141 calls jc.run() to process the device and generate the required output.

Lines 142 – 143 copy the temporary file, if one was created, to the final location specified by the filename argument using the AnsibleModule class’s atomic\_move() method. This is the second and final part of the approach recommended by An- sible for modules that must create files.

Lines 145 and 147 both exit the module and return results to the calling playbook. The difference is that line 145 uses AnsibleModule’s fail\_json() method to indi- cate the module encountered an error, while line 147 uses AnsibleModule’s exit\_ json() method to indicate a “normal” exit.

When a module exits, it returns results to the calling playbook as a JSON diction- ary (key:value data). The dictionary should include at least the key "changed" with a Boolean value indicating whether the module changed the target host in some way. This value lets the calling playbook know whether to display the green “ok” or yellow “changed” status for the task (module) results.

If the module encountered an error it should include "failed": True in the results dictionary. This value (when present and True) tells the playbook to display the red “fatal” status for the task results.

The module can include in the results a "msg" entry with a status or error message. A module should always include an error message when it encounters a failure.

The module can also include any other key:value data it needs to return to the call- ing playbook, where the key should be a meaningful string and the value can be anything (simple value, list, dictionary) in JSON format.

Line 145 shows how the module should exit when there is a failure. The Ansible- Module method fail\_json() automatically adds the entries "failed": true and "changed": false to the JSON dictionary returned to the playbook (the module can include the argument changed=True if appropriate despite the failure). The module adds to the results a dictionary entry with key “msg” and its value set to an error message describing the problem ( msg=str(err) ). Observe that the names of the keyword arguments to the fail\_json() method are the *unquoted* keys desired in the JSON results.

Line 147 shows a normal exit (no error) using AnsibleModule’s exit\_json() meth- od. The module needs to set the changed argument to a Boolean value indicating whether or not the host was modified; as this particular module never changes the target host, the only normal exit here sets changed=False, which becomes "changed": false in the resulting JSON dictionary. The module also passes to exit\_json the

argument commits=jc.commits, which becomes the "commits" key in the results dic- tionary with a list value containing the commit history.

If a custom module may or may not change the target host, it could have two exit\_ json() calls, one that sets changed=False and another that sets changed=True. Alter- nately, the module can set a changed variable to True or False and return that variable’s value in the exit\_json() call.

The structure of our example module allows all errors to be handled in a single try except stanza (lines 140 – 145), so a single fail\_json() call is sufficient. However, a module that handles failures at different places in the code might have multiple fail\_json() calls.

TIP In order to keep this example fairly straightforward, the author suggested copying the test\_commit\_history.py test program to create the library/commit\_his- tory.py custom module. This creates a problem should the JunosCommits class need to be modified in the future because you have two copies of the class defini- tion, one in the test program and one in the module. If you update the class in test\_commit\_history.py, you need to manually copy the changes to the module, an error-prone process. It would be better to put the class definition in a separate file and import that file into both the test program and the module; changes to the single copy of the class definition are incorporated into both the test program and the module. The GitHub repository for the book has a second version of selected files for this chapter showing one approach for importing a class definition.

### Testing the Custom Module

Let’s create a simple playbook to test the completed module. In your ~/aja direc- tory, create playbook get-commit-history.yaml:

1|---

2|- name: Get configuration history 3| hosts:

4| - all 5| roles:

6| - Juniper.junos 7| connection: local 8| gather\_facts: no 9|

10| tasks:

11| - name: get commit history 12| commit\_history:

13| host: "{{ ansible\_host }}"

14| max\_commits: 3

15| filename: "{{ inventory\_hostname}}-commit-history.txt" 16| register: history

17|

18| - debug: var=history.commits

Run the playbook and observe the results:

mbp15:aja sean$ **ansible-playbook get-commit-history.yaml**

PLAY [Get configuration history] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [get commit history] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

ok: [bilbo]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo] => {

"history.commits": [

{

"client": "cli", "comment": null,

"date\_time": "2017-10-06 14:54:34 UTC",

"num": "0",

"user": "sean"

},

{

"client": "cli", "comment": null,

"date\_time": "2017-10-06 14:24:08 UTC",

"num": "1",

"user": "sean"

},

{

"client": "cli", "comment": null,

"date\_time": "2017-10-06 11:33:16 UTC",

"num": "2",

"user": "sean"

}

]

}

ok: [vsrx1] => { "history.commits": [

{

"client": "netconf",

"comment": "playbook base-settings.yaml, confirming previous commit", "date\_time": "2017-12-11 15:15:17 UTC",

"num": "0",

"user": "sean"

},

{

"client": "netconf",

"comment": "playbook base-settings.yaml, commit confirmed", "date\_time": "2017-12-11 15:15:02 UTC",

"num": "1",

"user": "sean"

},

{

"client": "cli", "comment": null,

"date\_time": "2017-12-11 15:14:33 UTC",

"num": "2",

"user": "sean"

}

]

}

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* bilbo : ok=2 changed=0 unreachable=0 failed=0

vsrx1 : ok=2 changed=0 unreachable=0 failed=0

And check the results files:

mbp15:aja sean$ **cat bilbo-commit-history.txt**

Device returned 14 commits. Saving latest 3 commits.

- - - Commit History - - -

0: 2017-10-06 14:54:34 UTC by sean via cli (None)

1: 2017-10-06 14:24:08 UTC by sean via cli (None)

2: 2017-10-06 11:33:16 UTC by sean via cli (None)

mbp15:aja sean$ **cat vsrx1-commit-history.txt**

Device returned 39 commits. Saving latest 3 commits.

- - - Commit History - - -

0: 2017-12-11 15:15:17 UTC by sean via netconf (playbook base-settings. yaml, confirming previous commit)

1: 2017-12-11 15:15:02 UTC by sean via netconf (playbook base-settings.yaml, commit confirmed) 2: 2017-12-11 15:14:33 UTC by sean via cli (None)

Change some of the arguments in the playbook – for example, remove or com- ment out the filename and max\_commits arguments – and run the playbook again. Be sure the results are what you expect.

## Adding Commit History to Configuration Backups

A stand-alone playbook to get commit history may be useful, but it might be even more useful if we capture the commit history as part of our configuration backup process. When a device’s configuration has changed, it could be useful to record who made the most recent change(s) to the device.

Copy the get-config.yaml playbook to get-config-with-commits.yaml: mbp15:aja sean$ **cp get-config.yaml get-config-with-commits.yaml**

Open the new get-config-with-commits.yaml playbook in your editor and update it as shown (new lines shown boldfaced, line numbers added for discussion):

1|---

2|- name: Prepare timestamp 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: yes 7|

8| vars:

9| systime: "{{ ansible\_date\_time.time | replace(':', '-') }}" 10|

11| tasks:

12| - debug: var=ansible\_date\_time.time 13| - debug: var=systime

14|

15| - name: get system date and time 16| set\_fact:

17| timestamp: "{{ ansible\_date\_time.date }}\_{{ systime }}" 18|

19|- name: Backup Device Configuration 20| hosts:

21| - all 22| roles:

23| - Juniper.junos 24| connection: local 25| gather\_facts: no 26|

27| vars:

28| backup\_dir: "{{ user\_data\_path }}/config\_backups/{{ inventory\_hostname }}" 29| temp\_conf\_file: "{{ backup\_dir}}/{{ inventory\_hostname }}"

30| conf\_file: "{{ temp\_conf\_file }}\_{{ hostvars.localhost.timestamp }}.conf"

**31| commit\_file: "{{ backup\_dir }}/{{ inventory\_hostname }}\_{{ hostvars.localhost.timestamp }}. commit"**

32|

33| tasks:

34| - name: create backup directory if needed 35| file:

36| path: "{{ backup\_dir }}"

37| state: directory 38|

39| - name: save device configuration in temporary file 40| junos\_get\_config:

41| host: "{{ ansible\_host }}"

42| dest: "{{ temp\_conf\_file }}"

43| format: text

**44| notify:**

**45| - copy temporary file to timestamped config file if different 46| - get commit history**

**47|**

**48| handlers:**

**49| - name: copy temporary file to timestamped config file if different 50| copy:**

**51| src: "{{ temp\_conf\_file }}"**

**52| dest: "{{ conf\_file }}" 53|**

**54| - name: get commit history 55| commit\_history:**

**56| host: "{{ ansible\_host }}" 57| filename: "{{ commit\_file }}"**

When we wrote the get-config.yaml playbook in Chapter 9, we used a registered variable and when condition to determine if the temporary file should be copied. Ansible’s handlers provide an alternative approach, which this new playbook uses.

Line 31 adds a new variable to hold the path and filename for the commit history data.

Lines 44 – 46 cause the “save device configuration...” task, when the temporary configuration file has changed, to notify (trigger) the handlers that copy the tem- porary file to a permanent file, and that get the commit history.

Lines 48 – 57 are the new handlers. Lines 49 – 52 are the same as the task of the same name from the get-config.yaml playbook, while lines 54 – 57 call the new com- mit\_history.py custom module we wrote in this chapter. Observe that we do not register the JSON data returned by the module, because we do not need it for this playbook.

Run the playbook:

mbp15:aja sean$ **ansible-playbook get-config-with-commits.yaml**

PLAY [Prepare timestamp] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"ansible\_date\_time.time": "14:03:11"

}

TASK [debug] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost] => {

"systime": "14-03-11"

}

TASK [get system date and time] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [localhost]

PLAY [Backup Device Configuration] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [create backup directory if needed] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [bilbo]

ok: [vsrx1]

TASK [save device configuration in temporary file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* changed: [vsrx1]

ok: [bilbo]

RUNNING HANDLER [copy temporary file to timestamped config file if different] \*\*\* changed: [vsrx1]

RUNNING HANDLER [get commit history] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ok: [vsrx1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bilbo | : ok=2 | changed=0 | unreachable=0 | failed=0 |
| localhost | : ok=4 | changed=0 | unreachable=0 | failed=0 |
| vsrx1 | : ok=4 | changed=2 | unreachable=0 | failed=0 |

In this example run, there is no change to *bilbo*’s configuration, so the handlers are not notified. However, the configuration for firewall *vsrx1* has changed, so the handlers were notified to execute, the temporary config file is copied and the com- mit history is gathered and saved.

Review the files in a device’s backup directory:

mbp15:aja sean$ **ls -l ~/ansible/config\_backups/vsrx1/**

total 104

|  |  |  |  |
| --- | --- | --- | --- |
| -rw-r--r--  -rw-r--r--  ... | 1 sean  1 sean | staff staff | 4385 Dec 13 14:03 vsrx1  3315 Oct 5 15:35 vsrx1\_2017-10-05\_15-35-05.conf |
| -rw-r--r-- | 1 sean | staff | 3695 Oct 6 14:33 vsrx1\_2017-10-06\_14-33-13.conf |
| -rw-r--r-- | 1 sean | staff | 2914 Dec 13 14:03 vsrx1\_2017-12-13\_14-03-11.commit |
| -rw-r--r-- | 1 sean | staff | 4385 Dec 13 14:03 vsrx1\_2017-12-13\_14-03-11.conf |

mbp15:aja sean$ **cat ~/ansible/config\_backups/vsrx1/vsrx1\_2017-12-13\_14-03-11.commit**

Device returned 39 commits.

- - - Commit History - - -

0: 2017-12-11 15:15:17 UTC by sean via netconf (playbook base-settings. yaml, confirming previous commit)

1: 2017-12-11 15:15:02 UTC by sean via netconf (playbook base-settings.yaml, commit confirmed) 2: 2017-12-11 15:14:33 UTC by sean via cli (None)

...

## References

Ansible’s documentation about developing modules: <http://docs.ansible.com/ansible/latest/dev_guide/developing_modules.html>

Python tempfile module: <https://docs.python.org/2/library/tempfile.html>

Python logging module (may be used within a custom module to gather informa- tion to aid in debugging):

<https://docs.python.org/2/library/logging.html>

This book’s GitHub site: [https://github.com/Juniper/junosautomation/tree/master/ansible/](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible) [Automating\_Junos\_with\_Ansible](https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible)

# Appendix

## Using Source Control

Most professional programmers use a source control or version control system to track changes to their source code. Network engineers typically do not think of themselves as programmers, but anyone developing automation, including Ansible playbooks and associated files, is doing work similar to programming. Readers should consider treating their automation work with the same care that a tradi- tional programmer treats their source code.

This Appendix is a very brief introduction to source control, just enough to get you and your team started. Entire books have been written about using various source control systems, so if your team embraces this technology you should be able to find additional resources to help.

### What is Source Control and Why Use It?

*Version control*, or *revision control*, is a system or process for managing changes to documents, computer programs, web sites, etc. Version control can be a manual process, such as appending a “-2” to a filename when saving an updated version of a document, or it can be implemented using (features of a) computer program.

A *version control system* is software intended to manage changes to documents or source code. Version control systems intended to manage source code typically in- clude a *source code repository*, a way of storing source code and related files, and making those files available to the users (programmers) as needed. The author re- fers to these as *source control systems*.

For our purposes, *source code* includes our Ansible playbooks, Jinja2 templates, inventory files, and host and group data files.

Source control systems usually offer a number of features of interest to automa- tion engineers, programmers, web site developers, and similar information work- ers. The following descriptions are intentionally generic; exact terms, features, and operational details vary between different source control systems.

* + Shared repository for the "master" copy of the source files. Team members share source code or other files via the repository, not by emailing the files to each other or handing around flash drives. (Raise your hand if you ever partici- pated in a "sneaker net" using floppy disks. Yes, the author has earned his gray hair.)
  + Controlled access to the source files – authorized users can read or download the files, and a possibly smaller group of authorized users can upload or change files.
  + Some form of *branching*, the ability for a developer to work on one or more files without changing the "master" copy being used by others. For example, a developer may create a branch when they are making significant changes to an existing program (or playbook or template), changes that may temporarily break the program until the update is complete and tested. Other users can continue to work with the unmodified "master" copy while the developer completes his or her work in their private branch.
  + Some form of *merging*, bringing the changes made in a branch into the "mas- ter" version of the project so they are available to all users.
  + The ability to roll back changes to a previous state. For example, a developer realizes that the changes he has been making are going in the wrong direction and wants to return to a "known good" version of the project.

In short, source control makes it easy to share automation work between team members, and provides a backup and restore facility.

## Check Company Standards

This Appendix uses Git and GitHub to illustrate the use of source control. While both are popular choices, particularly for open-source software projects, there are other version control systems and source code repositories. If your company em- ploys engineers, programmers, web developers, or similar information workers, they may already have a source control system in place. It might even be Git!

Even if there is no corporate standard source control system, check with the infor- mation security team or other appropriate approvers before putting corporate data in an Internet-based system like GitHub, which could put your company’s intellectual property outside of your company’s exclusive control.

The examples in this Appendix will avoid using the Ansible playbooks and related files we have been developing. This was done so that the reader can work through the Git and GitHub examples without using files in the ~/aja directory that might contain corporate hostnames, IP addresses, or credentials.

## Brief Introduction to Git

Git was originally developed by Linus Torvalds in 2005 when he and the other de- velopers working on the Linux kernel needed a new version control system. Git is a distributed version control system, designed to support developers working in different locations, connected to each other via the Internet. Each developer’s sys- tem has a full copy of the repository for each project they are working on, so they can work offline.

This Appendix introduces Git using the git command-line program. We will create a new repository (project), create a branch, and merge the branch.

### Global settings

Start by telling Git your name and email address. Git associates this information with your changes so team members will know who made what change. These set- tings are *global*, meaning they apply to all repositories created or copied to your computer. You should make these settings on each computer where you use Git:

mbp15:aja sean$ **git config --global user.name "Sean Sawtell"**

mbp15:aja sean$ **git config --global user.email** [**"my\_email@juniper.net"**](mailto:my_email@juniper.net)

mbp15:aja sean$ **git config --global --list** user.name=Sean Sawtell [user.email=my\_email@juniper.net](mailto:user.email%3Dmy_email@juniper.net)

### Starting a repository

Now let’s create a new repository to hold the files for a new project called *widget*. In your home directory, create a new subdirectory widget, then change into that directory:

mbp15:aja sean$ **cd ~** mbp15:~ sean$ **mkdir widget** mbp15:~ sean$ **cd widget/** mbp15:widget sean$

Now tell Git that this directory is a new repository by using the git init command. Git creates a hidden subdirectory, .git, that Git uses to keep track of changes made to the files created in the repository:

mbp15:repo1 sean$ **git init**

Initialized empty Git repository in /Users/sean/widget/.git/

mbp15:widget sean$ **ls -a**

. .. .git

mbp15:widget sean$ **ls -a .git/**

. HEAD config hooks objects

.. branches description info refs

The git status command tells you the current state of your repository:

mbp15:widget sean$ **git status**

On branch master Initial commit

nothing to commit (create/copy files and use "git add" to track)

“On branch master” tells us we are currently working on the original branch of the project’s files, called *master* by default. When we create and use a different branch, the status will reflect the alternate branch name.

“Initial commit” tells us we have not yet committed a change. A *commit* is when you tell Git to take a “snapshot” of the current state of the repository, keeping track of changes to existing files or files that have been added or deleted.

“nothing to commit” confirms that there are no files yet in the repository. Let’s add a couple files -- for now, they can be empty – and then use git status

again to see the status change:

mbp15:widget sean$ **touch ansible.cfg**

mbp15:widget sean$ **touch play-widget.yaml**

mbp15:widget sean$ **git status**

On branch master Initial commit

Untracked files:

(use "git add <file>..." to include in what will be committed)

ansible.cfg

play-widget.yaml

nothing added to commit but untracked files present (use "git add" to track)

Notice that Git sees the new files but does not yet consider them part of the reposi- tory – they are “untracked files” at this point. To include the files in the repository, use the git add command:

mbp15:widget sean$ **git add ansible.cfg**

mbp15:widget sean$ **git add play-widget.yaml**

mbp15:widget sean$ **git status**

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: ansible.cfg

new file: play-widget.yaml

(Instead of adding the files individually, we could have used "git add ." to add all files in the current directory, or "git add –-all" to add all untracked files.)

Now commit the change (the addition of the two new files) using the git commit command. The –m option includes a message with the commit. If you forget to pro- vide a message with –m then git will open your system’s default text editor (which is probably vi or vim, unless you have changed the default) and ask you to enter a message there. Messages should briefly describe the change:

mbp15:widget sean$ **git commit -m "first widget playbook"**

[master (root-commit) 6b5b8e6] first widget playbook

2 files changed, 0 insertions(+), 0 deletions(-) create mode 100644 ansible.cfg

create mode 100644 play-widget.yaml

mbp15:widget sean$ **git status**

On branch master

nothing to commit, working tree clean

### Making and committing changes

Let’s modify a file and add another file. Add the following to file ansible.cfg:

[defaults]

inventory = inventory

Create file inventory containing the following:

localhost

Now check the repository’s status:

mbp15:widget sean$ **git status**

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory) modified: ansible.cfg

Untracked files:

(use "git add <file>..." to include in what will be committed) inventory

no changes added to commit (use "git add" and/or "git commit -a")

Git knows ansible.cfg has been modified, though the change is “not staged for commit.” Git sees the new file inventory as an untracked file. We need to add inven- tory as we did above:

mbp15:widget sean$ **git add inventory**

mbp15:widget sean$ **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage) new file: inventory

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory) modified: ansible.cfg

There are two approaches to including a changed file in a commit. One is to git add

the file, which will “stage” it for the next commit. The easier approach, assuming you wish to include all changed files in the commit, is to add the –a flag to the

git commit command:

mbp15:widget sean$ **git commit -a -m "add Ansible defaults and inventory file"**

[master 057299a] add Ansible defaults and inventory file

2 files changed, 4 insertions(+) create mode 100644 inventory

mbp15:widget sean$ **git status**

On branch master

nothing to commit, working tree clean

Now that we have two commits, let’s use the git log command to review the com- mit log (history):

mbp15:widget sean$ **git log**

commit 057299ab9b60810ca3b06ec4da0458462b649a84 (HEAD -> master) Author: Sean Sawtell [<my\_email@juniper.net>](mailto:my_email@juniper.net)

Date: Sun Oct 1 12:34:42 2017 -0400

add Ansible defaults and inventory file

commit 6b5b8e6d67ce44d52d8c744282232ee301cce72c Author: Sean Sawtell [<my\_email@juniper.net>](mailto:my_email@juniper.net)

Date: Sun Oct 1 11:59:28 2017 -0400

first widget playbook

Each commit has a unique ID number, generated by your system. *The ID numbers you will see on your system will be different from those shown above*. The log also shows the name and email address of the “author” who committed the change, the date and time of the commit, and the commit message.

You can also view the log in an abbreviated format using the --oneline option:

mbp15:widget sean$ **git log --oneline**

057299a (HEAD -> master) add Ansible defaults and inventory file 6b5b8e6 first widget playbook

Note the short ID number at the left of each line, followed by the commit message.

### Branching and Merging

List the existing branches using the git branch command. At the moment, the only branch is master:

mbp15:widget sean$ **git branch**

\* master

Let’s create a new branch called play1 where we can work on our first playbook. The command to switch to a different, existing branch is git checkout <branch>. To create a new branch and immediately switch to it, use git checkout –b <newbranch>:

mbp15:widget sean$ **git checkout -b play1**

Switched to a new branch 'play1'

mbp15:widget sean$ **git branch**

master

\* play1

The asterisk in the output of git branch indicates that play1 is the active branch. Update play-widget.yaml to contain the following:

---

* name: Show system date hosts:
* localhost connection: local gather\_facts: yes

tasks:

* debug: var=ansible\_date\_time.date

Then commit the change:

mbp15:widget sean$ **git commit -am "added task to playbook"**

[play1 7e60c92] added task to playbook

1 file changed, 9 insertions(+)

And confirm it appears in the commit log:

mbp15:widget sean$ **git log --oneline**

7e60c92 (HEAD -> master) added task to playbook 057299a add Ansible defaults and inventory file 6b5b8e6 first widget playbook

Run the playbook, if you like, to ensure it works.At the moment, the updates to play-widget.yaml exist only in the play1 branch, not in the master branch. Confirm this by checking out master and displaying the file:

mbp15:widget sean$ **git checkout master**

Switched to branch 'master' mbp15:widget sean$ **cat play-widget.yaml**

mbp15:widget sean$ **ls -l play-widget.yaml**

-rw-r--r-- 1 sean staff 0 Oct 3 10:21 play-widget.yaml

Observe that the play-widget.yaml file in the master branch is empty. Viewing the commit log shows why; the master branch does not have the commit associated with the updated playbook:

mbp15:widget sean$ **git log --oneline**

057299a (HEAD -> master) add Ansible defaults and inventory file 6b5b8e6 first widget playbook

Let’s merge the changes into the master branch using the git merge command and the name of the branch to be merged into the current branch:

mbp15:widget sean$ **git merge play1**

Updating 057299a..7e60c92 Fast-forward

play-widget.yaml | 9 +++++++++

1 file changed, 9 insertions(+)

mbp15:widget sean$ **git log --oneline**

7e60c92 (HEAD -> master, play1) added task to playbook 057299a add Ansible defaults and inventory file 6b5b8e6 first widget playbook

The output from git merge shows it updated the play-widget.yaml file, adding nine lines (note the “ + ” symbols for added lines; deleted lines display " - " and modi- fied lines display both). The output from git log shows the commit made after changing the playbook file.

### Merges with Conflicts

It is best if changes are made in only one of the branches prior to the merge, but git merge is pretty good about figuring out how to blend changes made in both branches. This is true even with changes made to the same file in both branches (for example, play-widget.yaml was altered in both master and play1 branches). There are limits, however, and one example is conflicting changes to the same line of the file. Let’s make conflicting changes to the playbook and see how to resolve them.

You should be on the master branch. Change the task in play-widget.yaml as shown:

tasks:

- name: show date

debug: var=ansible\_date\_time.date

Then commit that change:

mbp15:widget sean$ **git commit -am "add label to debug task"**

[master aa136d9] add label to debug task

1 file changed, 2 insertions(+), 1 deletion(-)

Now switch to the play1 branch:

mbp15:widget sean$ **git checkout play1**

Switched to branch 'play1'

Edit task in play-widget.yaml as shown:

tasks:

- debug: var=ansible\_date\_time.date

Commit the change, then switch back to the master branch and merge the change:

mbp15:widget sean$ **git commit -am "change debug task to key-value format"**

[play1 5317c89] change debug task to key-value format

1 file changed, 2 insertions(+), 1 deletion(-)

mbp15:widget sean$ **git checkout master**

Switched to branch 'master'

mbp15:widget sean$ **git merge play1**

Auto-merging play-widget.yaml

CONFLICT (content): Merge conflict in play-widget.yaml

Automatic merge failed; fix conflicts and then commit the result.

Observe that the merge cannot complete due to a “Merge conflict” in the play- book. Human intervention is needed to resolve the problem.

Open the playbook in your text editor. You will find some new lines in the file; these are to help you identify the conflict (line numbers added for discussion):

1|---

2|- name: Show system date 3| hosts:

4| - localhost

5| connection: local 6| gather\_facts: yes 7|

8| tasks:

9|<<<<<<< HEAD

10| - name: show date

11| debug: var=ansible\_date\_time.date 12|=======

13| - debug:

14| var: ansible\_date\_time.date 15|>>>>>>> play1

Line 9 and line 15 bracket the conflicting lines, while line 12 separates the changes between the two branches. In this example, the change to the master branch is shown first, identified by line 9 (think of HEAD as "the last commit on the current branch"), while the changes to the play1 branch are second, identified by line 15.

tasks:

Remove line 11 and delete the hyphen ("-") from line 13 to resolve the conflict, in this case keeping aspects of both changes. Remove lines 9, 12, and 15 (as labeled above) to delete Git’s markers from the file. The task should now look like this:

- name: show date debug:

var: ansible\_date\_time.date

Save the file, then commit the change:

mbp15:widget sean$ **git commit -am "fix merge conflict on playbook"**

[master 702e7f7] fix merge conflict on playbook

Take a look at the commit log; you can see the commits from each branch, fol- lowed by the commit resolving the conflict:

mbp15:widget sean$ **git log**

commit 702e7f74e02882a77c35039ffd2bb077beffc780 (HEAD -> master) Merge: aa136d9 5317c89

Author: Sean Sawtell [<my\_email@juniper.net>](mailto:my_email@juniper.net)

Date: Tue Oct 3 11:17:36 2017 -0400

fix merge conflict on playbook

commit 5317c89976db45009327fe234ae61604a20ddc2e (play1) Author: Sean Sawtell [<my\_email@juniper.net>](mailto:my_email@juniper.net)

Date: Tue Oct 3 10:56:23 2017 -0400

change debug task to key-value format

commit aa136d945c415b6df353a9adcf8589a106e7c00d Author: Sean Sawtell [<my\_email@juniper.net>](mailto:my_email@juniper.net) Date: Tue Oct 3 10:54:24 2017 -0400

...

add label to debug task

The master branch is now up-to-date, but you should also update the play1 branch:

mbp15:widget sean$ **git checkout play1**

Switched to branch 'play1'

mbp15:widget sean$ **git merge master**

Updating 5317c89..702e7f7 Fast-forward

play-widget.yaml | 3 ++-

1 file changed, 2 insertions(+), 1 deletion(-)

### Deleting Branches

You may wish to delete a temporary or working branch, generally after all changes have been committed and merged to master. Let’s delete the play1 branch. Start by making master the current branch, then use the git branch --delete command to de- lete play1:

mbp15:widget sean$ **git checkout master**

Switched to branch 'master'

mbp15:widget sean$ **git branch --delete play1**

Deleted branch play1 (was 702e7f7).

mbp15:widget sean$ **git branch**

\* master

Note that master is now the only branch.

You can force the deletion of a branch that contains committed changes that have not been merged to master using git branch –D. Assume our repository has a branch test1 with committed changes that have not been merged to master, and we want

to delete test1 without merging the changes (we wish to discard the changes):

mbp15:widget sean$ **git branch --delete test1**

error: The branch 'test1' is not fully merged.

If you are sure you want to delete it, run 'git branch -D test1'.

mbp15:widget sean$ **git branch -D test1**

Deleted branch test1 (was 1fc496f).

### Showing differences

The command git status shows you which files have changed since the last com- mit, but it does not show what changed in those files. We can use git diff to see the changes made within the files.

Edit the task in the play-widget.yaml file as follows:

tasks:

- name: show date and time debug:

var: ansible\_date\_time.iso8601

Use git diff to see the change:

mbp15:widget sean$ **git diff**

diff --git a/play-widget.yaml b/play-widget.yaml index d324fae..0e6ce1e 100644

--- a/play-widget.yaml

+++ b/play-widget.yaml @@ -6,6 +6,6 @@

gather\_facts: yes

tasks:

* - name: show date

+ - name: show date and time debug:

* var: ansible\_date\_time.date

+ var: ansible\_date\_time.iso8601

You can also see what changed relative to an earlier commit by specifying the com- mit ID of the commit (from git log):

mbp15:widget sean$ **git diff 7e60c9220e8f7a310cbed6cb6fcc7180dbbaadaa**

diff --git a/play-widget.yaml b/play-widget.yaml index 4576fb7..0e6ce1e 100644

--- a/play-widget.yaml

+++ b/play-widget.yaml @@ -6,4 +6,6 @@

gather\_facts: yes

tasks:

* - debug: var=ansible\_date\_time.date

+ - name: show date and time

+ debug:

+ var: ansible\_date\_time.iso8601

We can also show the differences between any two commits. This gets more conve- nient if we use the short IDs from git log --oneline. Let’s show the change from commit “add label to debug task” and the commit “add Ansible defaults and in- ventory file”:

mbp15:widget sean$ **git log --oneline**

702e7f7 (HEAD -> master) fix merge conflict on playbook 5317c89 change debug task to key-value format

aa136d9 add label to debug task 7e60c92 added task to playbook

057299a add Ansible defaults and inventory file 6b5b8e6 first widget playbook

mbp15:widget sean$ **git diff aa136d9 057299a** diff --git a/play-widget.yaml b/play-widget.yaml index 43040bd..e69de29 100644

--- a/play-widget.yaml

+++ b/play-widget.yaml @@ -1,10 +0,0 @@

----

-- name: Show system date

* hosts:
* - localhost
* connection: local
* gather\_facts: yes

-

* tasks:
* - name: show date
* debug: var=ansible\_date\_time.date

Commit the outstanding changes to the playbook:

mbp15:widget sean$ **git commit -am "update date output to include time"**

[master 65e9037] update date output to include time

1 file changed, 2 insertions(+), 2 deletions(-)

mbp15:widget sean$ **git status**

On branch master

nothing to commit, working tree clean

mbp15:widget sean$ **git log --oneline**

65e9037 (HEAD -> master) update date output to include time 702e7f7 fix merge conflict on playbook

5317c89 change debug task to key-value format aa136d9 add label to debug task

7e60c92 added task to playbook

057299a add Ansible defaults and inventory file 6b5b8e6 first widget playbook

We will soon discuss using Git with a shared repository, which will enable a team to share source files and synchronize changes between each team member’s system. First, however, we need to talk about GitHub, which will be our shared repository.

## Brief Introduction to GitHub

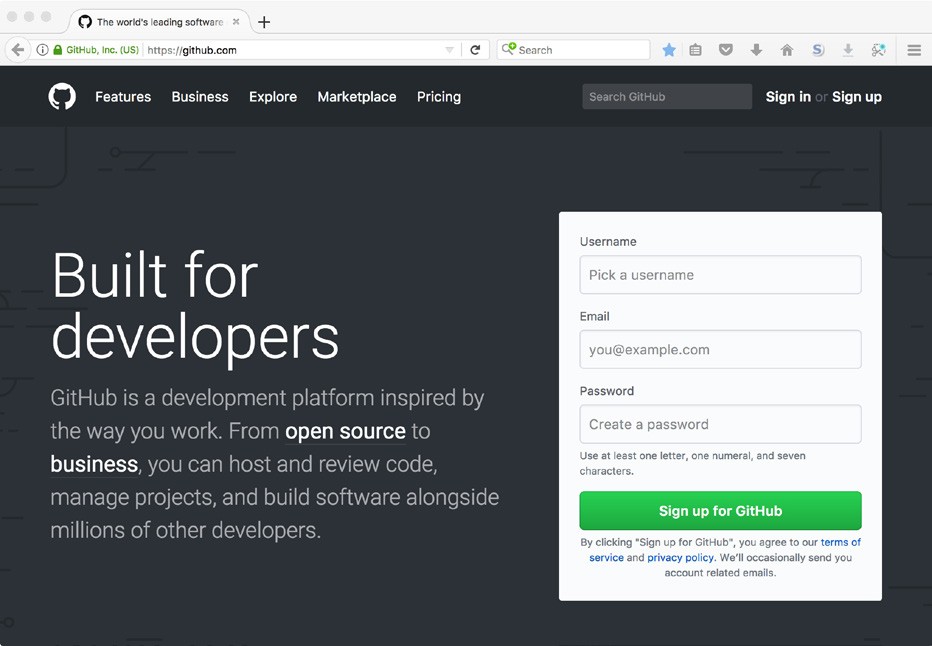
GitHub is a web-based Git repository. Launched in 2008, today it is “the largest host of source code in the world” [Wikipedia, Sept. 30 2017]. Developers can cre- ate repositories, branch and merge, and even edit files, using the GitHub WebUI. Developers can also work in their local development environment and use the git program to push their changes to GitHub over the Internet.

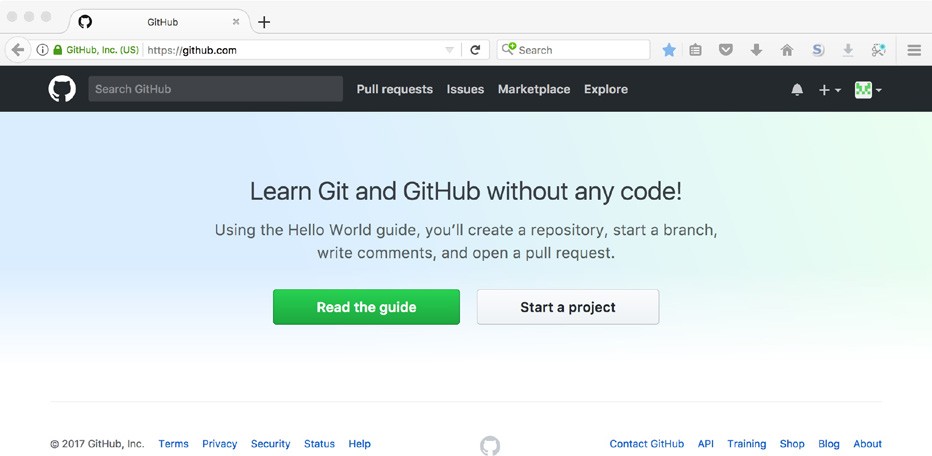
The popularity of Git and GitHub for open source projects, combined with the desire of many enterprises to keep their source code on servers they control rather than “in the cloud,” has led to the creation of enterprise GitHub-like products.

Examples include GitHub Enterprise and GitLab. Many of these products work similarly to GitHub; if your company uses one of these products, you may wish to follow the examples using your corporate solution, altering instructions as needed to accommodate WebUI differences.

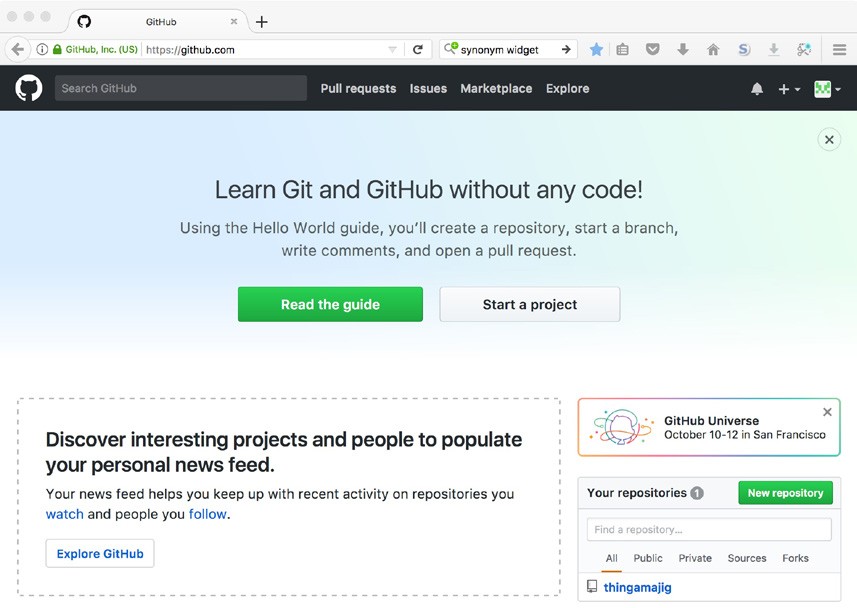
NOTE The GitHub screen captures in this section and the next were taken between October 7, 2017 and February 2, 2018. What you see may differ should GitHub update their WebUI.

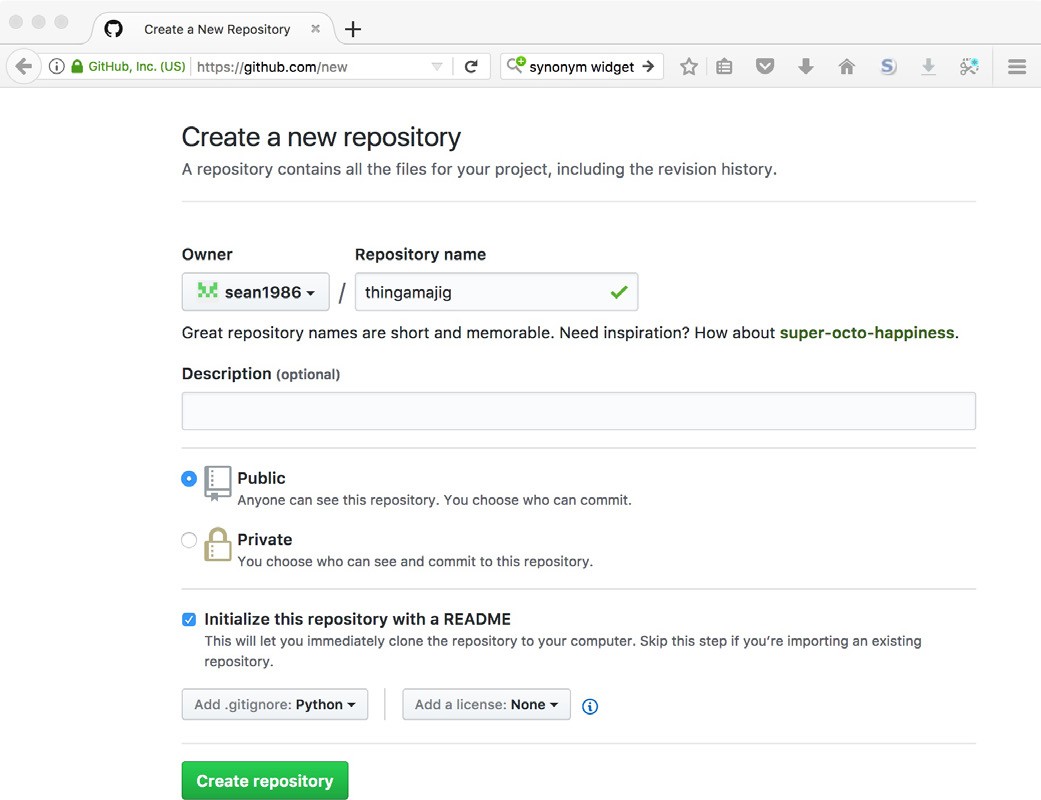
Open you web browser and navigate to <https://github.com/>. If you do not already have a GitHub account (that you can use for these examples), sign up for one using the box on the main page or the *Sign up* link in the upper right corner of the page. If you already have an account, click the *Sign in* link and log in.



If this is a new account, or if your existing account has no associated repositories, the screen after logging in will offer only a couple of choices. Click the *Start a proj- ect* button to begin a new project in a new repository.

If your account already has one or more associated repositories, the screen after logging in will have more options, including a list of your repositories. Click either the *Start a project* button or the *New repository* button to start a new project.



Fill in the fields for creating a new repository as shown in the following screen capture, except that the *Owner* should be your account.

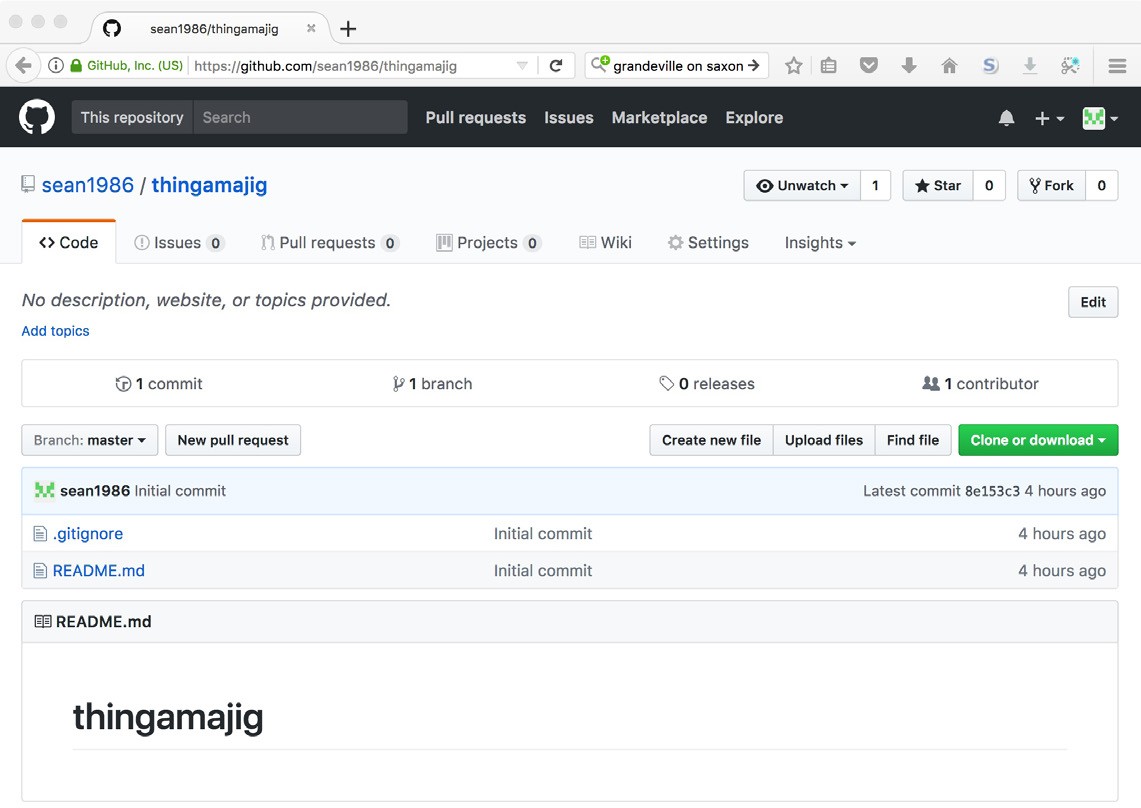
The fields on this page are:

* Owner: The GitHub account that has administrative authority for the reposi- tory. Normally this will be the creator of the repository.
* Repository name: A name for the repository, ideally a descriptive name for the project. The name must be unique within the list of repositories associated with the owner.
* Public or Private: GitHub allows public repositories that can be seen by any- one, and private repositories that are visible only to accounts selected by the

repository owner. GitHub charges for private repositories, so we will use pub- lic repositories for our examples.

* Intitialize...README: GitHub can include in the new repository a “read me” file with which you can describe the contents or purpose of the repository. The filename will be README.md. The extension .md indicates the file uses GitHub’s Markdown format.
* Add .gitignore: GitHub can include in the new repository a .gitignore file, which includes appropriate settings for the programming language selected here. (We discuss .gitignore later in this chapter; briefly, it tells Git to ignore certain files.)
* Add a license: Open source projects are typically made available under one of several common open-source licenses. GitHub can include a file containing the license agreement if you select the appropriate license type.

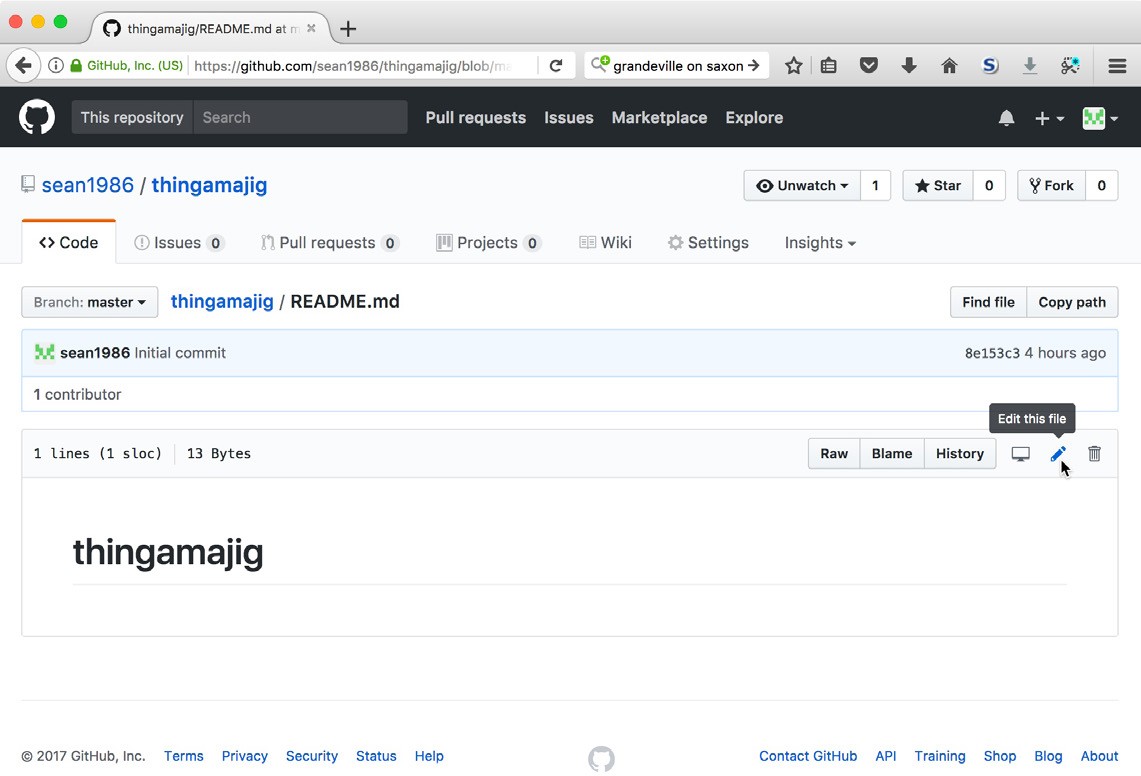
Click the *Create repository* button to have GitHub initialize the new repository. This should take you to the main screen for the repository, displaying the list of files or directories in the repository and the contents of the README file.



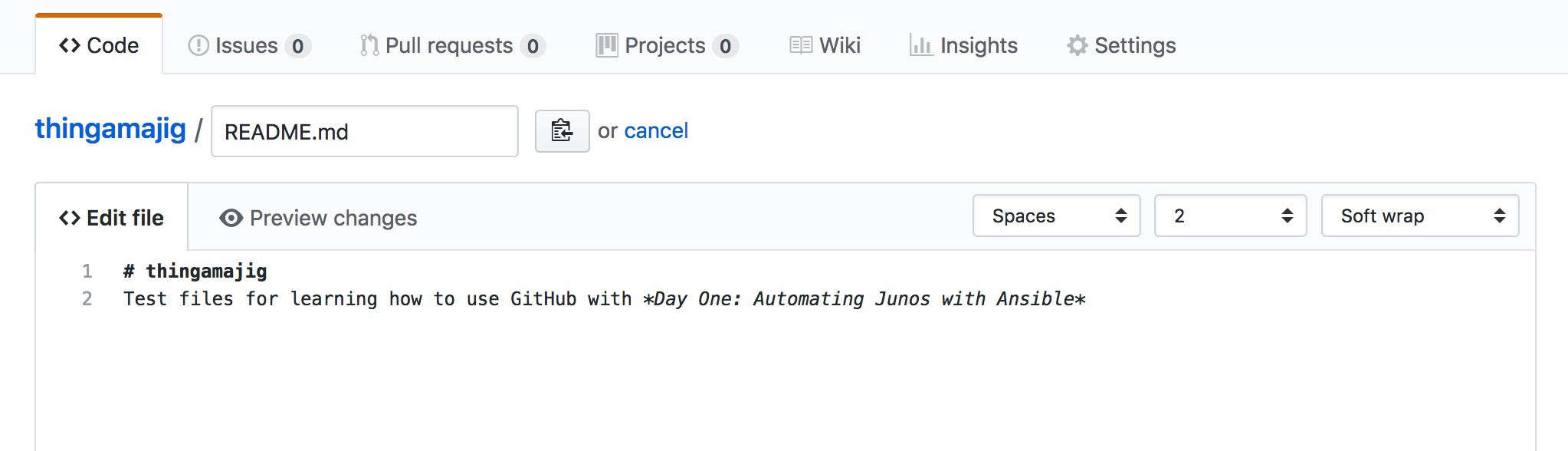
You can see the new repository, including the file list showing the .gitignore and

README.md files.

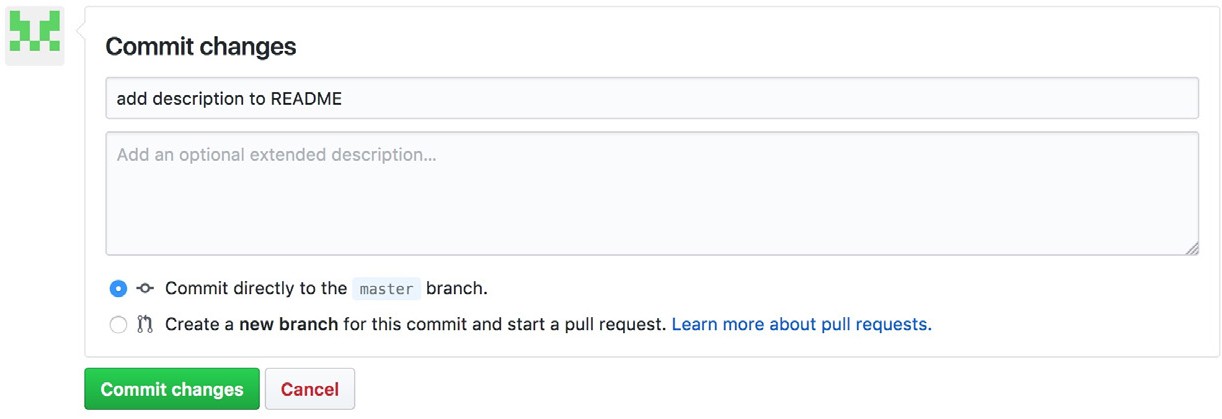
Let’s edit one of the files. Click the blue README.md in the file list. The next screen will show the file’s contents.



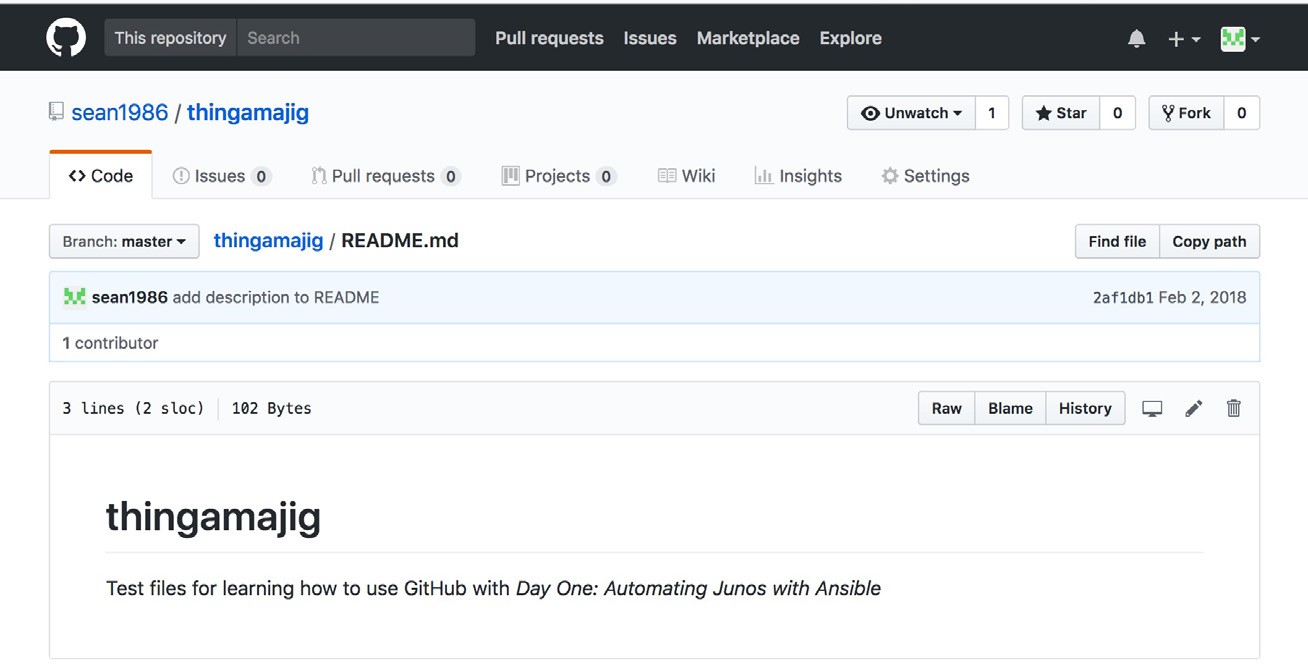
Click the pencil icon to the upper right of the file’s contents to edit the file. Modify the file as shown here.



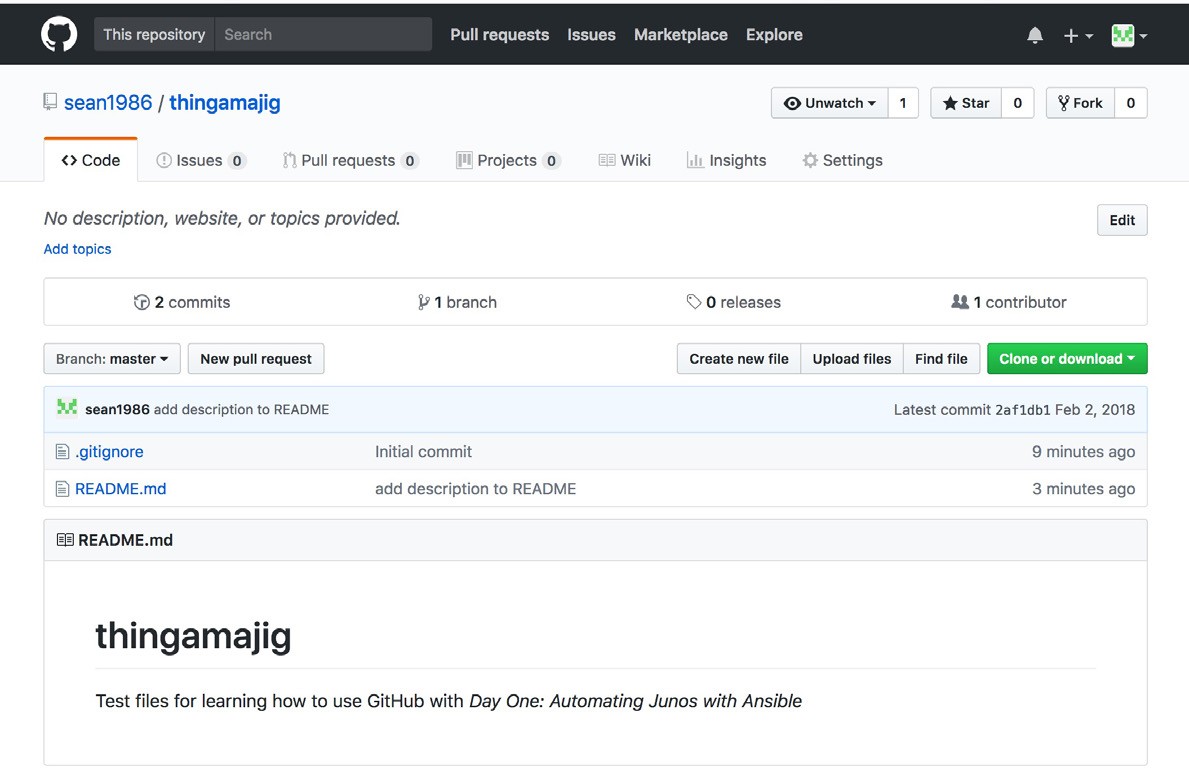
To commit (save) the change, scroll to the bottom of the page, add a commit mes- sage in the small text box (where the picture shows “add description to READ- ME”) and click the *Commit changes* button.

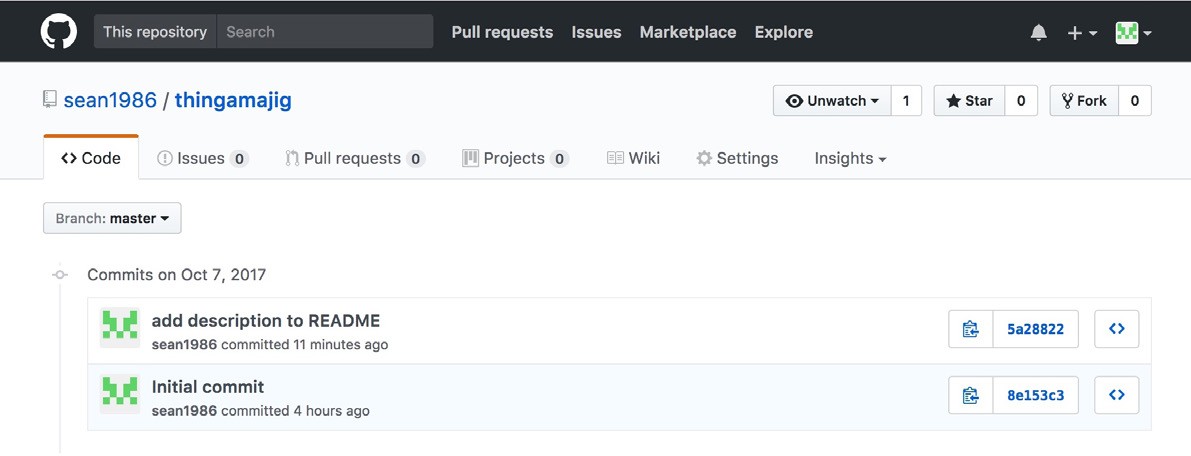


The screen should now look like this.



Click the *thingamajig* link to return to the project’s file list.



You can see the commit history by clicking the *2 commits* link (of course, the num- ber will change as more changes are committed).

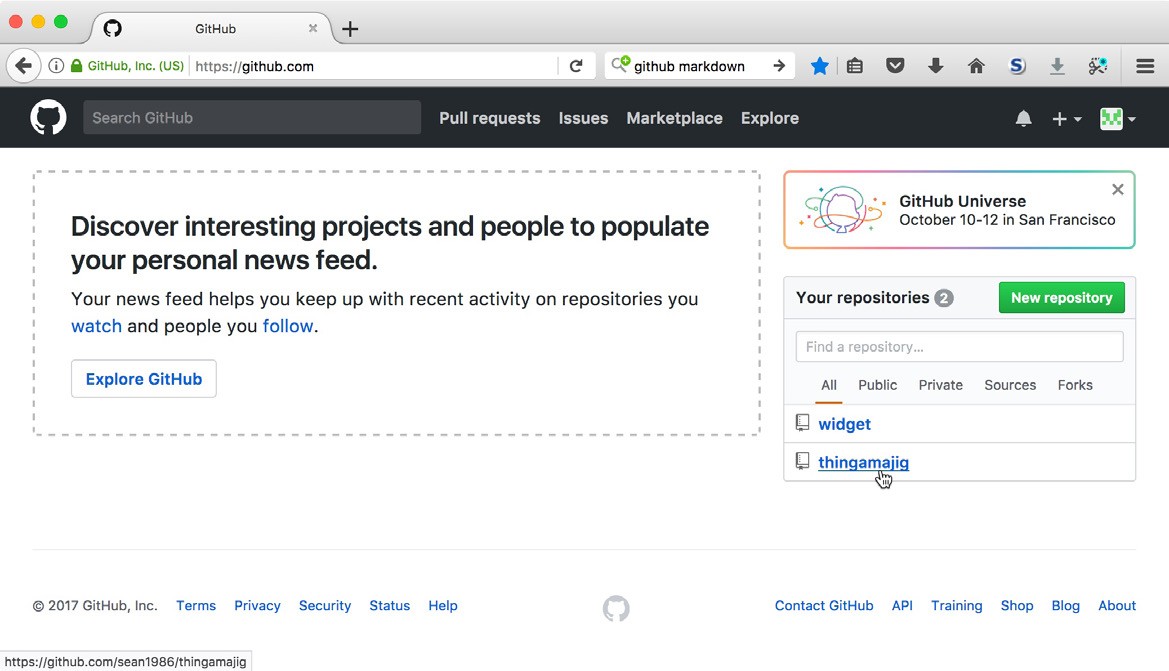
## Using Git with GitHub

While it is possible to edit files and commit changes using the GitHub WebUI, a more common approach is for developers to work with a local copy of a GitHub repository. Developers use their preferred text editor to edit local files, and syn- chronize changes between their local repository and GitHub.

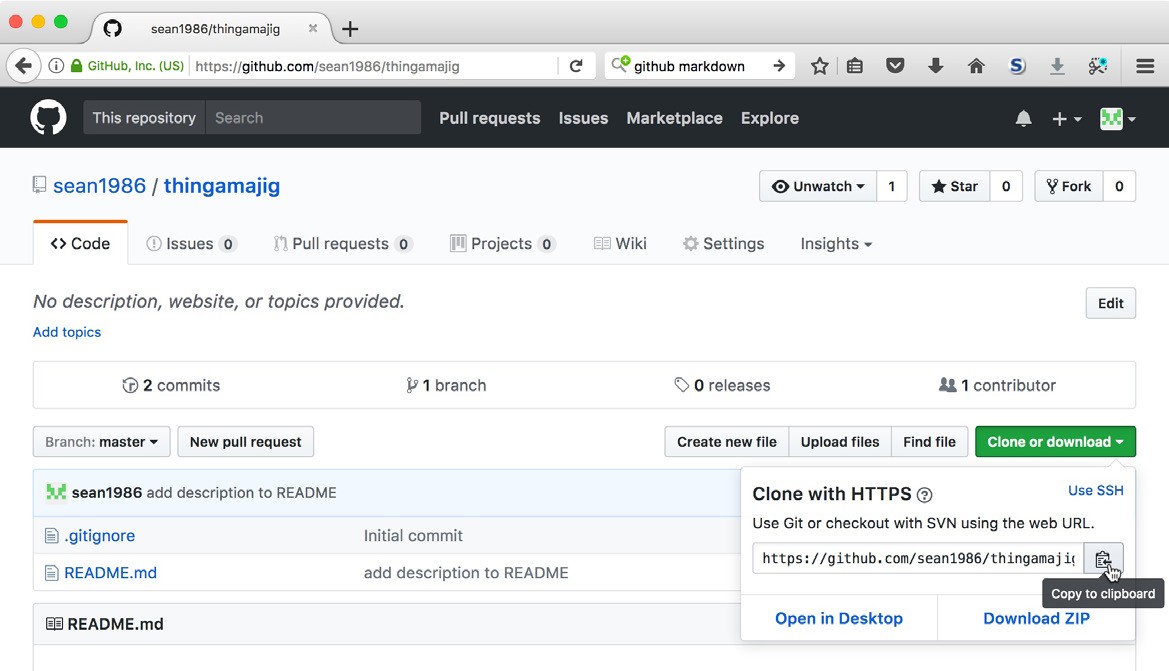
GitHub allows developers to use either HTTPS or SSH when synchronizing reposi- tories. Using SSH requires creating and configuring an SSH key pair, while HTTPS uses your GitHub username and password. The following examples show HTTPS, but the author encourages you to explore the SSH option if you will be using GitHub for production use. The References section at the end of this Appendix includes a link to GitHub’s SSH instructions.

### Cloning an existing repository

Let’s start with showing how to clone an existing repository. This is a common sit- uation: there is an existing project on GitHub that you wish to copy to a local re- pository and work with. We use our *thingamajig* project for this example.

In your web browser, navigate to your *thingamajig* project. One approach is to click the GitHub icon in the upper left corner of the GitHub page, then click the *thingamajig* link in the list of your repositories.

Click the *Clone or download* button to expose the options for cloning the reposi- tory. Click the *Copy to clipboard* button to the right of the URL to copy the URL to your system’s clipboard.



In your command shell, change to your home directory (or whatever directory you want to be the parent of the repository directory). Then run the command git

clone with the copied URL to create a new repository in a new directory containing a copy of the *thingamajig* project. Your URL will differ from what is shown in the examples because your GitHub username is different from the author’s username:

mbp15:widget sean$ **cd ~**

mbp15:~ sean$ **git clone https://github.com/sean1986/thingamajig.git**

Cloning into 'thingamajig'... remote: Counting objects: 7, done.

remote: Compressing objects: 100% (6/6), done.

remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0 Unpacking objects: 100% (7/7), done.

You should now have a subdirectory ~/thingamajig. Change into that subdirectory and note the files within that match what we created on GitHub, plus the .git di- rectory where Git tracks changes:

mbp15:~ sean$ **cd thingamajig/**

mbp15:thingamajig sean$ **ls -al**

total 16

drwxr-xr-x 5 sean staff 170 Oct 9 13:11 .

drwxr-xr-x+ 38 sean staff 1292 Oct 9 13:11 ..

drwxr-xr-x 12 sean staff 408 Oct 9 13:11 .git

-rw-r--r-- 1 sean staff 1157 Oct 9 13:11 .gitignore

-rw-r--r-- 1 sean staff 103 Oct 9 13:11 README.md

mbp15:thingamajig sean$ **cat README.md**

# thingamajig

Test files for learning how to use GitHub with \*Day One: Automating Junos with Ansible\*

In order to simulate a second team member who is also working with the *thing- amajig* repository, create a directory ~/repos and clone *thingamajig* into that direc- tory also (because we will switch back and forth between these two copies of the repository, it may be convenient to use a second command shell window for this):

mbp15:~ sean$ mkdir repos mbp15:~ sean$ cd repos/

mbp15:repos sean$ **git clone https://github.com/sean1986/thingamajig.git**

Cloning into 'thingamajig'... remote: Counting objects: 7, done.

remote: Compressing objects: 100% (6/6), done.

remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0 Unpacking objects: 100% (7/7), done.

mbp15:repos sean$ **cd thingamajig/**

mbp15:thingamajig sean$ **pwd**

/Users/sean/repos/thingamajig

Let’s make a change and push that change back to GitHub. In your first command shell window, or in ~/thingamajig directory, open README.md in your text editor and add a copyright notice as follows:

# thingamajig

Test files for learning how to use GitHub with \*Day One: Automating Junos with Ansible\*

**(c)2017 Juniper Networks, Inc.**

Commit the change and check the status:

mbp15:thingamajig sean$ **git commit -am "add copyright to README"**

[master 681856e] add copyright to README

1 file changed, 1 insertion(+)

mbp15:thingamajig sean$ **git status**

On branch master

Your branch is ahead of 'origin/master' by 1 commit. (use "git push" to publish your local commits)

nothing to commit, working tree clean

Observe that Git knows your local repository is “ahead of ‘origin/master’ by 1 commit,” meaning that the local repository has a newer commit than the last com- mit on GitHub. To push the changes – the most recent commit – to GitHub, use the git push command:

mbp15:thingamajig sean$ **git push**

Counting objects: 3, done.

Delta compression using up to 8 threads. Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 349 bytes | 349.00 KiB/s, done. Total 3 (delta 1), reused 0 (delta 0)

remote: Resolving deltas: 100% (1/1), completed with 1 local object. To https://github.com/sean1986/thingamajig.git

5a28822..681856e master -> master

Refresh the GitHub page in your web browser. You should see that GitHub has an updated copy of README.md containing the copyright notice.

Switch to your second command shell window or to ~/repos/thingamajig and look at the README file:

mbp15:thingamajig sean$ **pwd**

/Users/sean/repos/thingamajig

mbp15:thingamajig sean$ **cat README.md**

# thingamajig

Test files for learning how to use GitHub with \*Day One: Automating Junos with Ansible\*

Observe that this “second team member” has an old copy of the repository. GitHub has been updated, based on the changes pushed by the “first team mem- ber,” but all other team members need to pull an update to the repository using the git pull command:

mbp15:thingamajig sean$ **git pull**

remote: Counting objects: 3, done.

remote: Compressing objects: 100% (2/2), done.

remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0 Unpacking objects: 100% (3/3), done.

From https://github.com/sean1986/thingamajig e023752..4cbdf38 master -> origin/master

Updating e023752..4cbdf38 Fast-forward

README.md | 2 ++

1 file changed, 2 insertions(+)

mbp15:thingamajig sean$ **cat README.md**

# thingamajig

Test files for learning how to use GitHub with \*Day One: Automating Junos with Ansible\* (c)2017 Juniper Networks, Inc.

Run git log to see the change log. Note that the log shows the commit made by the

other team member:

mbp15:thingamajig sean$ **git log --oneline**

4cbdf38 (HEAD -> master, origin/master, origin/HEAD) add copyright to README 5a28822 add description to README

8e153c3 Initial commit

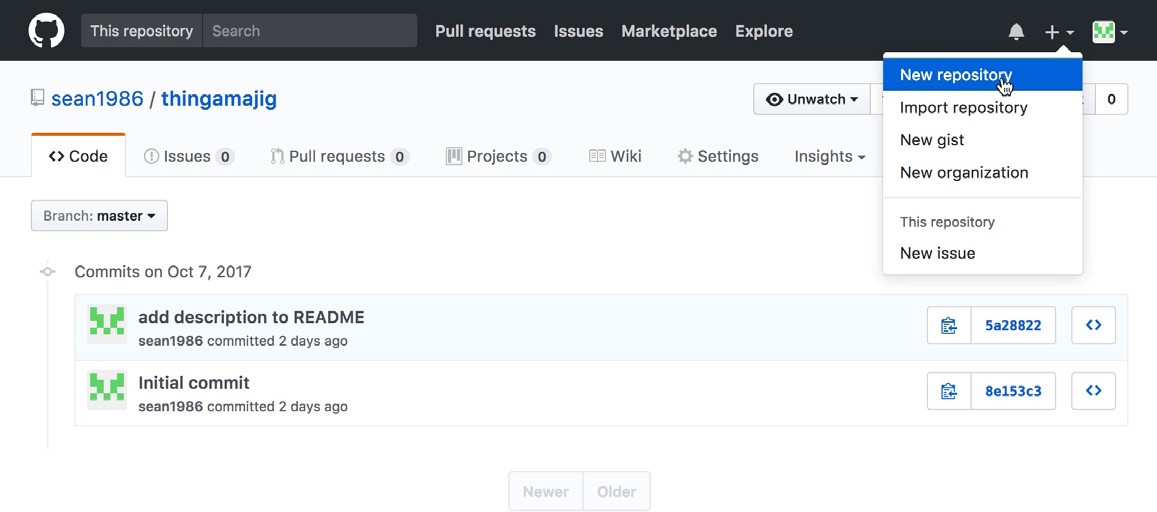
Any team member (with appropriate GitHub permissions) can make changes, commit them, and push those changes to GitHub. All other team members can pull those updates.

### Pushing a Local Repository

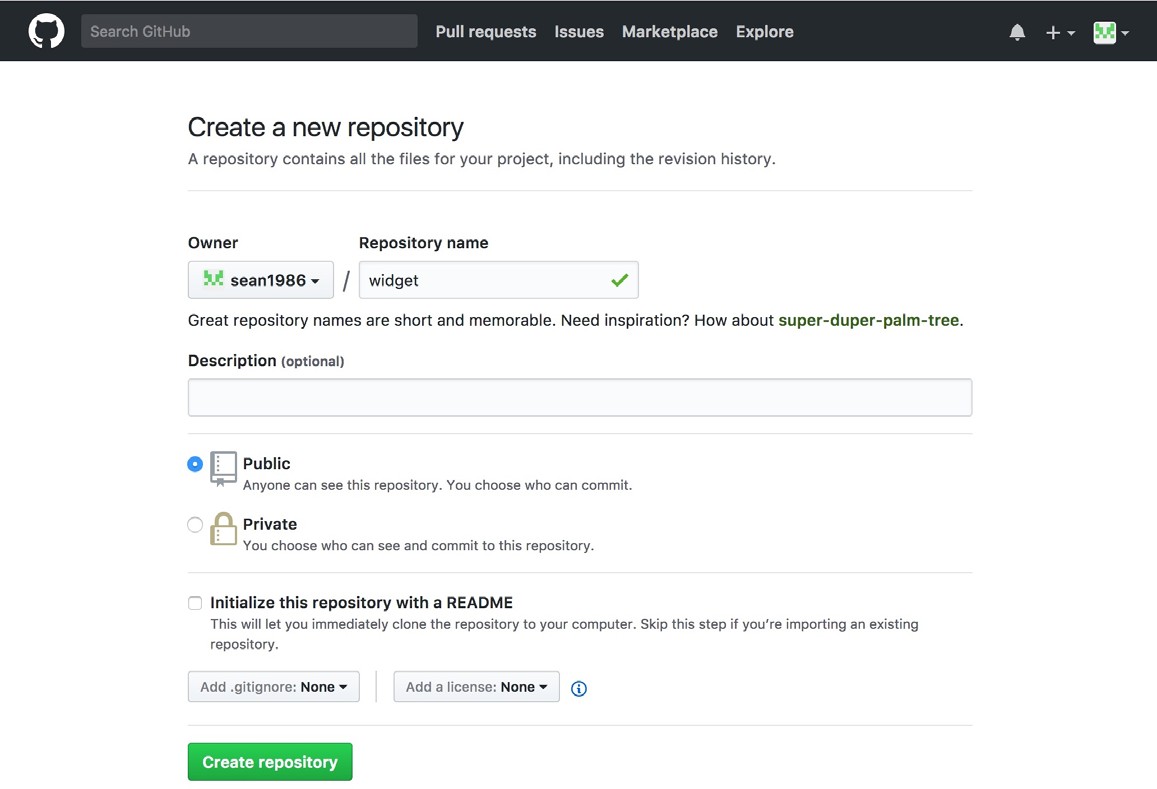
The git clone command works well for repositories that were originally created on GitHub, or at least are up-to-date on GitHub, but what if we started a project lo- cally and now want to upload it to GitHub?

Our *widget* project is such a project – we initially created it on our local system, not on GitHub. Assume that we now want to share the *widget* project with other team members using GitHub.

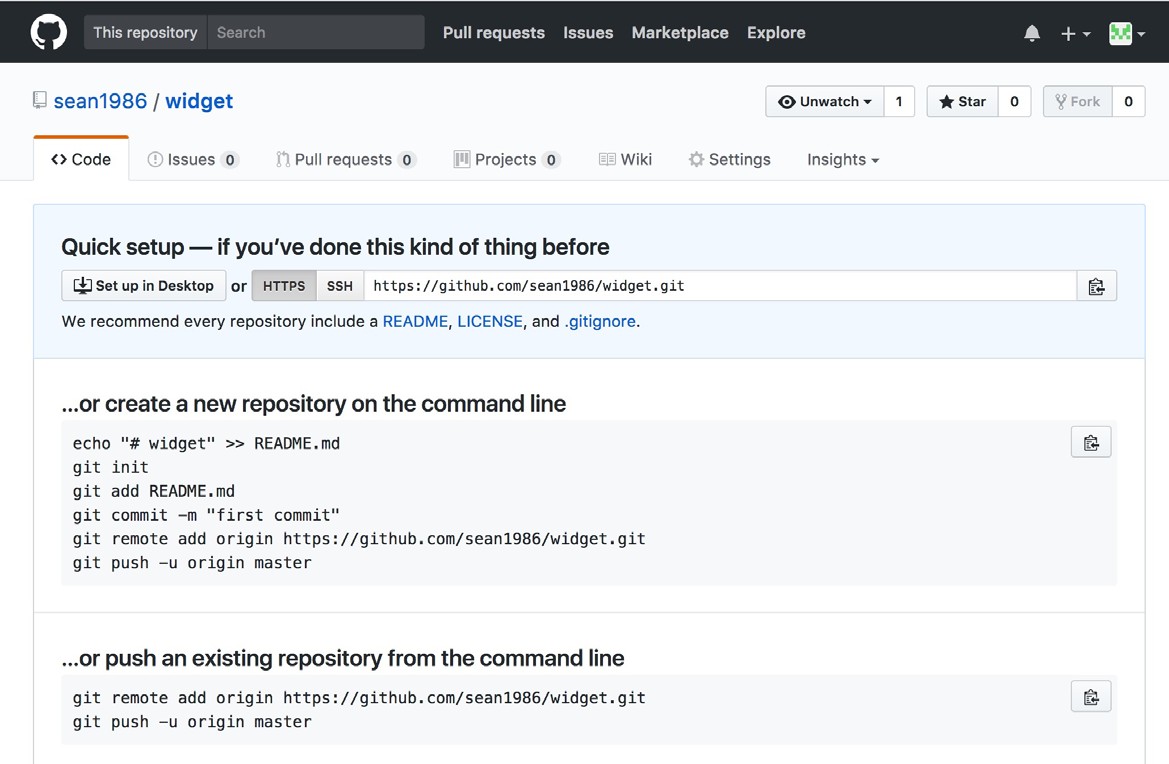
The first step is to create an empty project on GitHub that will become the shared repository for the project. In your web browser, on the GitHub page, click the + button in the upper right corner and choose *New repository* from the menu:



We want a completely empty GitHub repository, because we will upload the files from our local Git repository. Enter the repository name *widget* and, optionally, a description, as shown, then click the *Create repository* button. (Do *not* include a README.md, .gitignore, or license file):



This time GitHub, seeing the repository is empty, should display some instructions for uploading repository files from other locations:



Note the instructions to “push an existing repository from the command line.” This is what we want to do! Keep in mind that your URL will be different from what is shown, as your URL will include your GitHub username, not that author’s username.

In your command shell, change to the widget project directory and ensure there are no uncommitted changes:

mbp15:thingamajig sean$ **cd ~/widget/**

mbp15:widget sean$ **git status**

On branch master

nothing to commit, working tree clean mbp15:widget sean$

Now run the two commands from the GitHub instructions:

mbp15:widget sean$ **git remote add origin https://github.com/sean1986/widget.git**

mbp15:widget sean$ **git push -u origin master**

Counting objects: 22, done.

Delta compression using up to 8 threads. Compressing objects: 100% (20/20), done.

Writing objects: 100% (22/22), 1.85 KiB | 316.00 KiB/s, done. Total 22 (delta 10), reused 0 (delta 0)

remote: Resolving deltas: 100% (10/10), done. To https://github.com/sean1986/widget.git

\* [new branch] master -> master

Branch master set up to track remote branch master from origin.

The git remote add command adds to your local repository a link to a remote reposi- tory. The expected name for a remote or shared repository is origin. The URL specifies the location for the remote repository.

The git push command, as we have already seen, pushes the current state of the local repository (more specifically, the current branch) to the remote repository. However, when doing the initial push of a branch created locally, we need to in- clude the -u (or --set-upstream) argument followed by the remote repository name origin. The argument master indicates we are pushing the master branch.

In your web browser, refresh the GitHub page. You should now see the files that are part of the *widget* project.

Because the command git push -u origin master pushes the master branch to the ori- gin server, you should repeat this command for any other branches you wish to up- load to the remote repository. For example, assuming the repository has a branch called sean, the following command would push branch sean to GitHub:

mbp15:widget sean$ **git push -u origin sean**

Counting objects: 3, done.

Delta compression using up to 8 threads. Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done. Total 3 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), completed with 2 local objects. To https://github.com/sean1986/widget.git

\* [new branch] sean -> sean

Branch sean set up to track remote branch sean from origin.

Once the initial push is completed, subsequent pushes will not require the addi- tional command line options. To demonstrate this, add a new play to play-widget. yaml:

---

* name: Show system date hosts:
* localhost connection: local gather\_facts: yes

tasks:

* name: show date and time debug:

var: ansible\_date\_time.iso8601

**- name: show hostname debug:**

**var: ansible\_hostname**

Commit and push this change:

mbp15:widget sean$ **git commit -am "add hostname to playbook"**

[master 7f85d8c] add hostname to playbook

1 file changed, 5 insertions(+), 1 deletion(-)

mbp15:widget sean$ **git push**

Counting objects: 3, done.

Delta compression using up to 8 threads. Compressing objects: 100% (3/3), done.

Writing objects: 100% (3/3), 341 bytes | 341.00 KiB/s, done.

Total 3 (delta 2), reused 0 (delta 0)

remote: Resolving deltas: 100% (2/2), completed with 2 local objects. To https://github.com/sean1986/widget.git

65e9037..7f85d8c master -> master

Other team members will clone, pull, and push normally. For example, become your “second user” by changing to your ~/repos directory and clone the *widget* repository using its URL (obtained from GitHub):

mbp15:widget sean$ **cd ~/repos/**

mbp15:repos sean$ **git clone https://github.com/sean1986/widget.git**

Cloning into 'widget'...

remote: Counting objects: 28, done.

remote: Compressing objects: 100% (12/12), done.

remote: Total 28 (delta 15), reused 27 (delta 14), pack-reused 0 Unpacking objects: 100% (28/28), done.

mbp15:repos sean$ **cat widget/play-widget.yaml**

---

* name: Show system date hosts:
* localhost connection: local gather\_facts: yes

tasks:

* name: show date and time debug:

var: ansible\_date\_time.iso8601

* name: show hostname debug:

var: ansible\_hostname

## The .gitignore File

There will be some files that need to exist within the repository directory, but that you will not want included in the repository or pushed to the remote repository. The .gitignore file provides a way to tell Git to ignore such files.

Consider, for example, the ~/aja/group\_vars/all.yaml file that we created and up- dated in the previous few chapters:

mbp15:~ sean$ **cat ~/aja/group\_vars/all.yaml**

---

ansible\_python\_interpreter: /usr/local/bin/python user\_data\_path: /Users/sean/Ansible

These paths, particularly the user\_data\_path, are specific to your system or user- name. Should you choose to place the *aja* project in source control, other team members will need to have their own, unique versions of this file.

Other examples include temporary files or directories, such as the ~/aja/tmp/ direc- tory where some of our playbooks from earlier chapters have placed result files that we did not need to keep long-term.

Let’s simulate these files in our widget project, along with a group variables file that will be the same for all users and should be included in the repository:

mbp15:~ sean$ **cd ~/widget/** mbp15:widget sean$ **mkdir tmp** mbp15:widget sean$ **mkdir group\_vars**

mbp15:widget sean$ **touch tmp/result.json** mbp15:widget sean$ **touch group\_vars/all.yaml** mbp15:widget sean$ **touch group\_vars/boston.yaml** mbp15:widget sean$ **tree**

.

├── ansible.cfg

├── group\_vars

│ ├── all.yaml

│ └── boston.yaml

├── inventory

├── play-widget.yaml

└── tmp

└── result.json

2 directories, 6 files

Add the boston.yaml file to the repository, then check the status of the repository:

mbp15:widget sean$ **git add group\_vars/boston.yaml**

mbp15:widget sean$ **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage) new file: group\_vars/boston.yaml

Untracked files:

(use "git add <file>..." to include in what will be committed)

group\_vars/all.yaml tmp/

Git identifies the contents of the tmp directory and the group\_vars/all.yaml files as untracked. These files will not be pushed to GitHib. However, if we leave things as they are, we will keep seeing these untracked files in the output every time we run git status; this will quickly get annoying.

Git looks for a file called .gitignore in the top directory of a repository. The files and directories listed in .gitignore will be ignored by Git. Create ~/widget/.gi- tignore with the following contents:

# user-specific data for all hosts all.yaml

# temp files or contents of a temp directory

\*.tmp tmp/ temp/

# Ansible retry (after failure) files

\*.retry

# Mac OS X Desktop Services Store

.DS\_Store

\*.tmp tmp/ temp/

Lines starting with hash marks (#) are comments. The line group\_vars/all.yaml will cause Git to ignore that file.

The following lines will cause Git to ignore the contents of our tmp directory and other common temporary files and directories:

The line \*.retry tells Git to ignore Ansible’s “retry” files – if a playbook encounters an error for one or more hosts, Ansible creates a file *<playbook>.retry* with the name of each host that had an error.

Finally, for MacOS users, the line .DS\_Store tells Git to ignore any .DS\_Store files that MacOS might create in your project directories.

Check the status of the repository again:

mbp15:widget sean$ **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage) new file: group\_vars/boston.yaml

Untracked files:

(use "git add <file>..." to include in what will be committed)

.gitignore

Notice that the ignored files no longer appear in the output. However, .gitignore itself now appears as an untracked file. We should add .gitignore to our repository, because it is unlikely this file will contain any user-specific settings:

mbp15:widget sean$ **git add .gitignore**

mbp15:widget sean$ **git status**

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

new file: .gitignore

new file: group\_vars/boston.yaml

Commit the changes and push the repository to GitHub:

mbp15:widget sean$ **git commit -am "new files including .gitignore"**

[master e519e97] new files including .gitignore

2 files changed, 13 insertions(+) create mode 100644 .gitignore

create mode 100644 group\_vars/boston.yaml

mbp15:widget sean$ **git push**

Counting objects: 4, done.

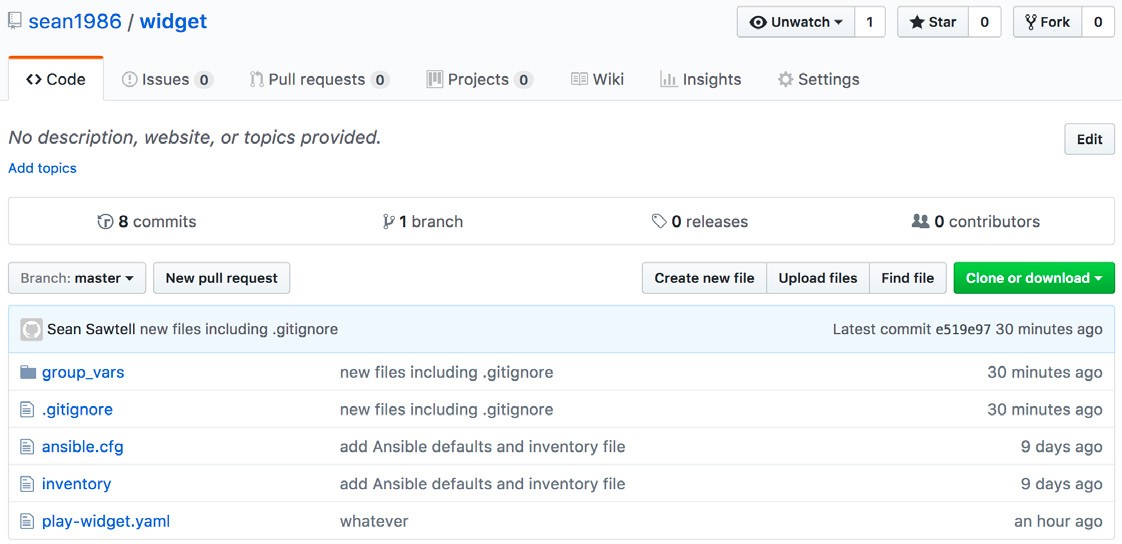
Delta compression using up to 8 threads. Compressing objects: 100% (2/2), done.

Writing objects: 100% (4/4), 313 bytes | 313.00 KiB/s, done. Total 4 (delta 1), reused 0 (delta 0)

remote: Resolving deltas: 100% (1/1), completed with 1 local object.

To https://github.com/sean1986/widget.git e519e97..5e4d58e master -> master

Switch to your web browser and look at the GitHub *widget* repository file list. Notice that the tmp directory does not appear, and within the group\_vars directory only the boston.yaml file appears:



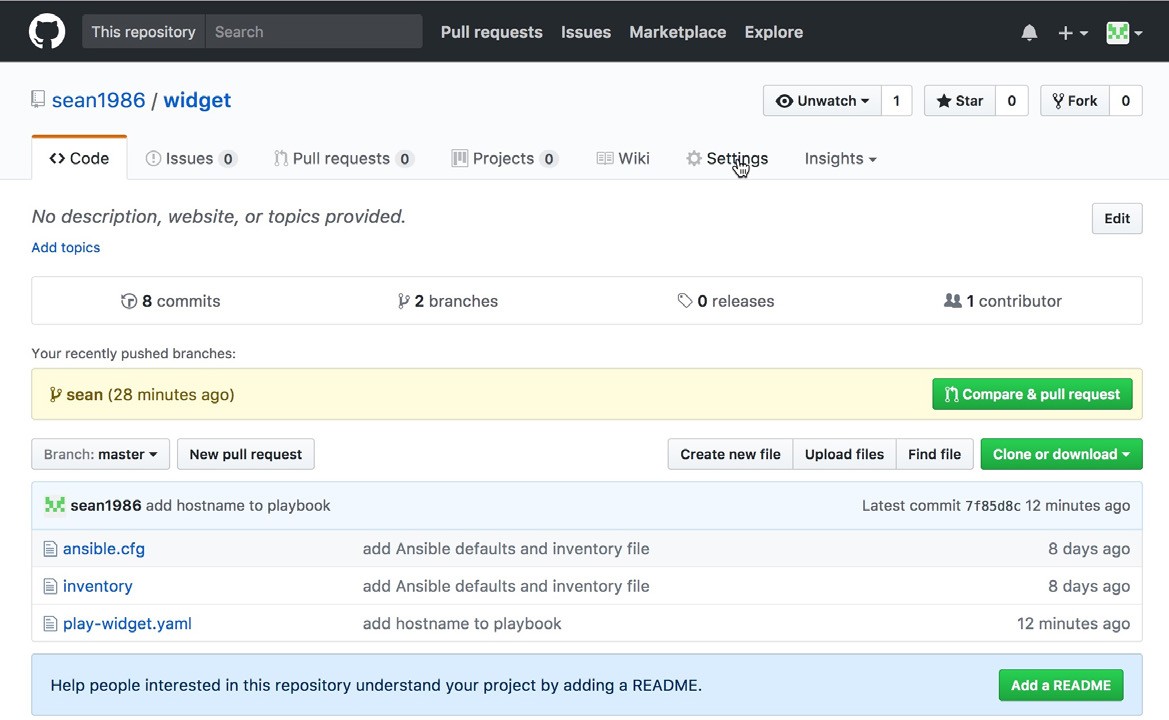
## Cleaning Up Example Repositories

Feel free to delete the local repositories created during this Appendix by simply deleting the directories:

mbp15:repos sean$ **cd ~** mbp15:~ sean$ **rm -rf repos/** mbp15:~ sean$ **rm -rf widget/**

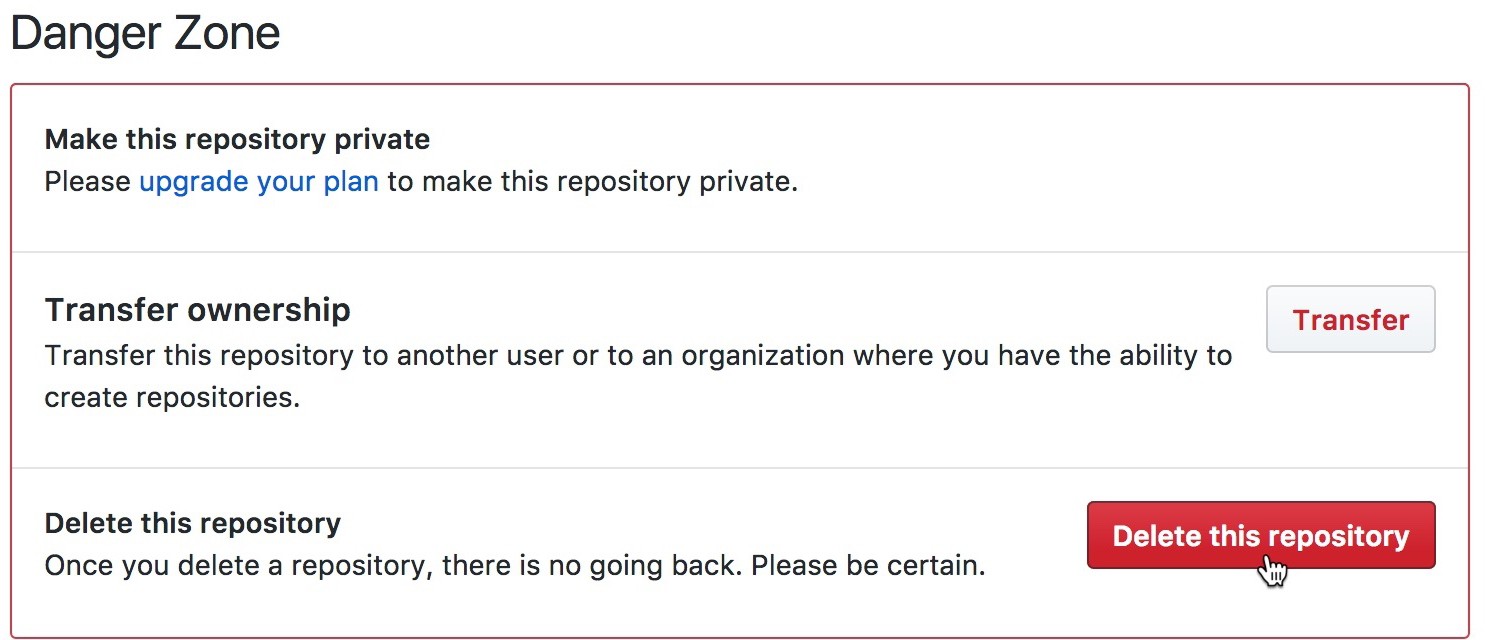
mbp15:~ sean$ **rm -rf thingamajig/**

To delete a repository from GitHub, view the repository and click the *Settings* link:



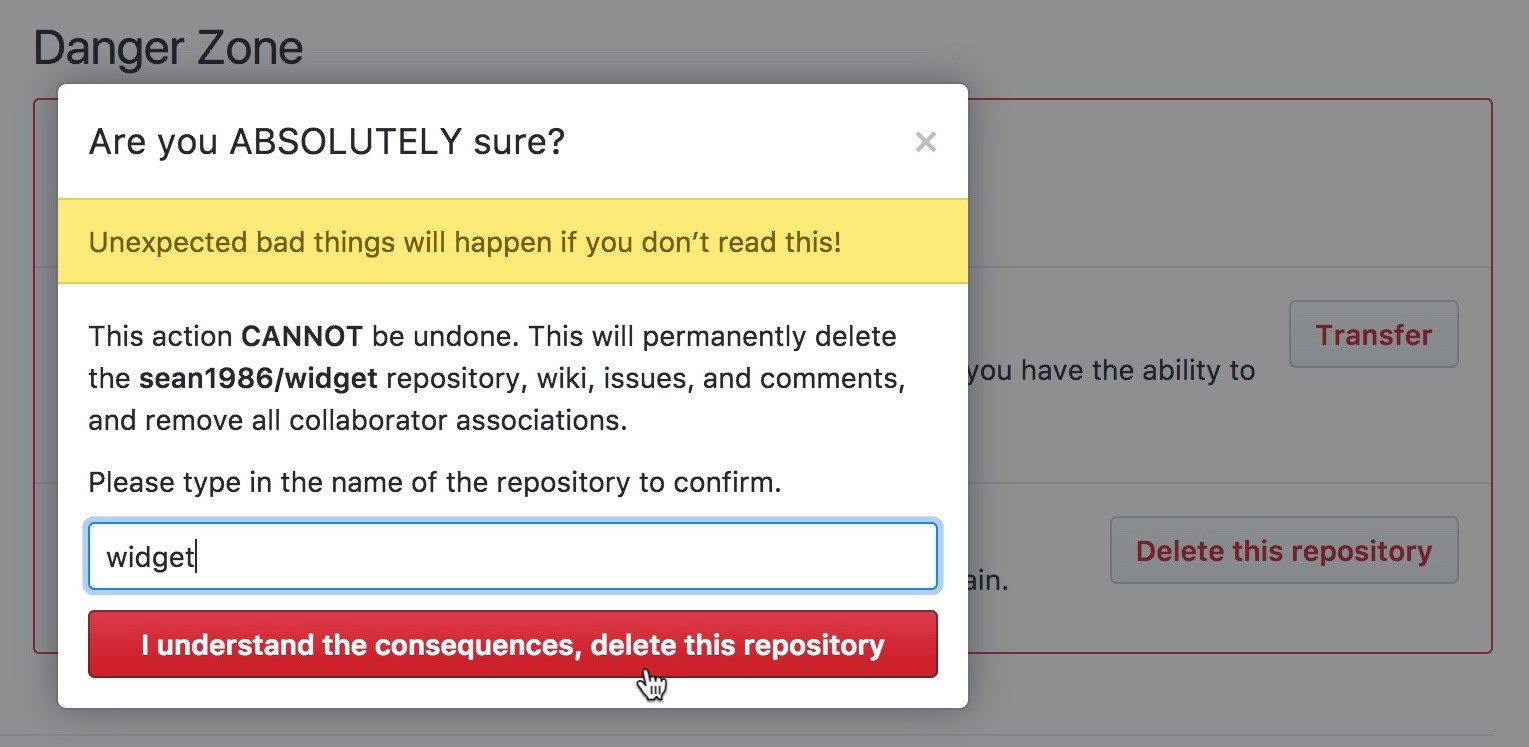
Scroll to the bottom of the Settings page and click the *Delete this repository*

button:



In the confirmation window, enter the name of the repository and click the button

*I understand the consequences, delete this repository* to delete the repository:



## Appendix References

Version control and distributed version control: <https://en.wikipedia.org/wiki/Version_control> <https://en.wikipedia.org/wiki/Distributed_revision_control>

Git:

<https://git-scm.com/> <https://en.wikipedia.org/wiki/Git>

GitHub:

<https://github.com/> <https://en.wikipedia.org/wiki/GitHub>

GitHub’s Markdown reference: <https://guides.github.com/features/mastering-markdown/>

GitHub using SSH keys: <https://help.github.com/articles/connecting-to-github-with-ssh/>