JUNIPER NETWORKS | Engineering Simplicity

DAY ONE: AUTOMATING JUNOS WITH ANSIBLE, EDITION v2.1

Create Ansible playbooks to automate your network tasks. This new Second Edition is updated for Juniper's Galaxy modules version 2.x and the latest versions of Ansible and the Junos® OS.

By Sean Sawtell

DAY ONE: AUTOMATING JUNOS WITH ANSIBLE, EDITION v2.1

Network automation is expanding rapidly. Many network engineers are looking into automation 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, Second Edition*, is the newest book on network automation for network engineers. It includes a set-up guide, tutorials, and showcase scenarios whose Ansible scripts you can download from 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 experience. 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 screenshots, 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 configuration to devices
- Create custom Ansible modules for tasks not supported by existing modules

JUNIPER
NETWORKS

Day One: Automating Junos® with Ansible, Edition 2.1

by Sean Sawtell

Updated Edition 2.1, May 2020

JUNIPER
NETWORKS

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 certifications.

Feedback? Comments? Error reports? Email them to dayone@juniper.net.

This Book's GitHub Site

Go to: https://github.com/Juniper/junosautomation/tree/master/ansible/Automating_Junos_with_Ansible.

Notes for Edition 2.1

This "edition 2.1" of *Day One: Automating Junos with Ansible, 2nd Edition* is a modest update to the Second Edition. The author has three goals:

Python 3

When the author was writing the second edition of this book in the summer of 2018, he recommended using Python 2.7 instead of Python 3. At the time, he was seeing some problems when testing his playbooks with Python 3.

The author is happy to change that recommendation. As the author prepares this edition in April 2020, current versions of Ansible, PyEZ, and the other necessary modules are working beautifully with Python 3.7. Moreover, the Python Software Foundation ended support for Python 2 on January 1, 2020.

The author now recommends Python 3.7 or 3.8 and versions of Ansible and PyEZ that were current or released in late 2019 or after. Chapter 2 has been updated to reflect this recommendation.

NOTE   The author has tested selected playbooks to ensure they work with the newer software but has not updated all the example output in the book. There may be minor differences between what appears herein and what is displayed by newer versions of Ansible and the supporting modules.

Errata

The author has corrected a few errata that were reported by readers of the book. Thank you!

Limited permission accounts on Junos

A reader suggested including a discussion of using Junos accounts that provide limited management permissions on the device, rather than super-user access. (Thank you for the suggestion!)

A section has been added to the end of Chapter 9 to illustrate how we might use a read-only account on the Junos device to back up its configuration.

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 focused 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 managing Juniper's own internal network. It considers real-world concerns such as security, complying with corporate policy, and synchronizing the automation development work between team members.

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 editor[1], 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/). Whatever your chosen editor, see if it has (built-in or via an installable module) JSON and YAML syntax highlighting[2] .

- 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.

---

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 processor except that it handles only plain text – no boldface, no fonts, no tables, etc. – and saves files containing only ASCII or ANSI text with no formatting information.

2 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).

- 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 concepts as we work through the examples. However, if you have some programming 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.

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 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 automation.

- How to use SSH keys for making secure device access easier.

- How to create custom Ansible modules for things that Ansible does not already do, or does not do the way you want them done.

- Why network automation engineers should use source control and a short introduction to Git and GitHub.

Book Structure and Approach to Learning

This book takes a hands-on approach to learning. As you read most of the chapters, 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 immediately 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 assumes 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 programmers 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 resolve them. These examples provide an opportunity to show troubleshooting processes 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 automation. 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 Ansible support a network engineer's automation needs?

What is Automation?

According to the online Merriam-Webster dictionary (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 automatically

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 automation, 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 network engineer would otherwise need to accomplish.

The type of automation discussed throughout 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 network respond automatically to events, for example, taking some action to mitigate problems without human intervention. While getting to this level, event-driven automation, is outside the scope of this book, the topics discussed herein create a foundation upon which you can 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 connection, 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's 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 improve 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 hundreds 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. Automated solutions usually require an initial investment of time and effort to develop and debug the automation. The development investment is repaid by the time savings 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 status or error counts, routing tables, etc. This data may be used for planning upgrades, troubleshooting problems, or other needs.

Configuring devices: Automation can quickly push configuration changes to devices. 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 configuration, 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 devices 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 connection, 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 troubleshooting information in response to an event, such as uploading a "request 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 vs. 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 whether 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 implemented 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/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 operations that accomplish defined tasks. It is commonly used "to provision, deploy, and manage compute infrastructure across cloud, virtual, and physical environments." (https://www.ansible.com/webinars-training/introduction-to-ansible; accessed 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 knowledge – 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 discuss it in some detail.

Ansible includes a large selection of modules with the platform. While Ansible differentiates between three categories of included modules (core, curated, and community), this book refers to the included modules collectively as core modules. (http://docs.ansible.com/ansible/list_of_all_modules.html)

There are also modules developed by the Ansible user community and made available via Ansible Galaxy (https://galaxy.ansible.com/). This book refers to these as Galaxy modules. As you will see in Chapter 2, when installing Ansible 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://galaxy.ansible.com/Juniper/junos/). Supported operations include executing commands, 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 Administration*, O'Reilly Media, 2016.) Not surprisingly, these modules use Juniper's suggested techniques for how off-box automation communicates with Junos devices. This book discusses the recommended approach in Chapter 5, 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 Junos devices (http://docs.ansible.com/ansible/list_of_network_modules.html#junos). Supported operations are broadly similar to Juniper's Galaxy modules, 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. This book focuses on Juniper's Galaxy modules for working with Junos devices, rather than Ansible's core Junos modules, but the reader is encouraged to explore the latter as well.

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 this book.

Playbook: The file you create that defines your desired automation process by calling 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 arguments, 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 — the book discusses several approaches for this — 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 commands, but with some places where Ansible will "fill-in-the-blank" with data specific 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 often 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 collections 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` directories 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 (https://www.ansible.com/products/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 support, and other features intended to make Ansible more accessible to an IT team.

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 (https://github.com/ansible/awx) but Red Hat provides no paid support for it.

This book focuses on the command-line Ansible platform. However, many playbooks 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 automation scenarios mentioned in the "Business Scenarios for Network Automation" section of this chapter and see if, and how, Ansible 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 other 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 create 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 configuration, 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 compliance, but a Python programmer could write such a module, or you could have Ansible 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, possibly by using Juniper's JSNAPY tool. 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 configuration; this can include a 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 configuration via a serial connection. For those who use ZTP, 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; arranging 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 machine. The systems being managed by an Ansible control machine are called managed nodes.

An Ansible control machine that manages Junos devices requires:

- A non-Windows operating system. Linux, macOS, and other UNIX-type operating systems work well.
- Python 3.7 (or newer), including pip (package installer for Python).
- An SSH client, typically OpenSSH. This is usually installed by default on Linux/ UNIX systems and macOS.
- Juniper's Galaxy modules, version 2.3 or newer.
- Juniper's PyEZ Python library, version 2.4 or newer.

The control machine communicates with the Junos devices using the NETCONF 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 automation environment from your host OS, you may wish to use a VM or Docker container similar to the Windows users.

Software Versions Used While Writing This Book

The author used the following versions of Ansible and related modules while developing and testing the examples in this book (2nd and 2.1 editions):

- Ansible 2.4.3, 2.6.1, and 2.9.9
- Juniper.junos (Juniper's Galaxy modules) 2.0.2, 2.1.0, and 2.3.1
- Pip 9.0.1 and 20.0.2
- PyEZ (junos-eznc) 2.1.7, 2.1.8, and 2.4.1
- Python 2.7.14, 2.7.15, and 3.7.7

This second edition of *Day One: Automating Junos with Ansible* is written for version 2.x of the Juniper.junos Galaxy modules, which made significant changes from the previous 1.4.3 version. If you are using the older modules, please use the first edition of this book.

The author strongly recommends using Python 3.x, not the older Python 2.7 interpreter. The Python language was modified in version 3 in ways not backwards compatible with programs written for Python 2, but all the projects used here have made updates for Python 3.x compatibility. Also note that the maintainers of Python ended support for Python 2 on January 1, 2020, so please use Python 3 unless you have a compelling reason to stay with Python 2.

Generally, you do not need to use the exact versions listed above. All were current versions during the time the author was writing the second edition of this book, but most have probably been upgraded by the time you're reading this. Keep in mind that the maintainers of these open-source projects do occasionally change module names or arguments or the like. If a playbook is throwing 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.

Ansible's Installation Instructions

Ansible's website 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 it 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 will use 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 Apple's complete XCode development environment), open a Terminal window and enter the command xcode-select --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.
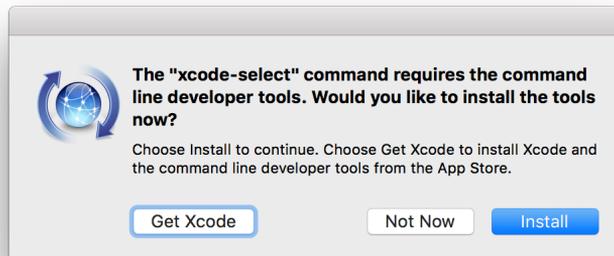
Click *Agree* in the *Command Line Tools License Agreement* dialog box that appears next. An installation status dialog box should appear as the software is downloaded and installed. Click *Done* when the installation is complete.

Downloading software

Time remaining: About a minute

Stop

The software was installed.

Done
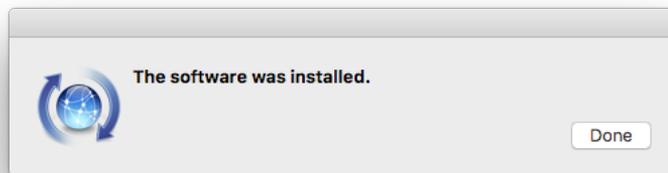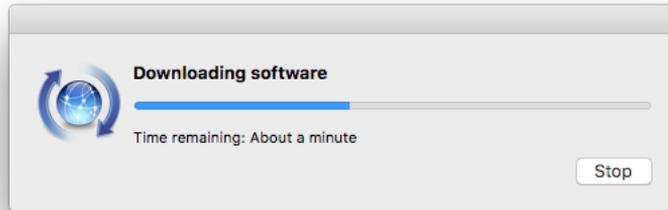
Homebrew and Python

Recent versions of macOS include a Python interpreter. While it is possible to install 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 library that is also used by PyEZ, but macOS locks the library so it cannot be altered. 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/) package manager[1] and use it to install a new Python environment[2]. 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.

---

1 There are other package managers for macOS, such as MacPorts (https://www.macports.org/), which may accomplish the same goal. The author has not worked with these other package managers, 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).

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 macOS Mojave system, the system-installed Python is version 2.7.16.

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

```
mbp15:~ sean$ python --version
Python 2.7.16
mbp15:~ sean$ which python
/usr/bin/python
```

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 Homebrew web page (https://brew.sh/):
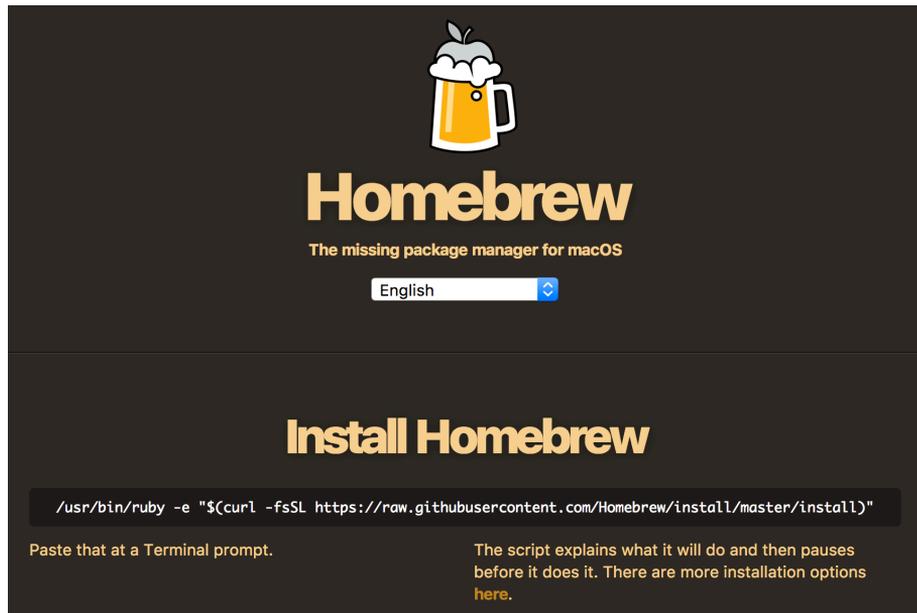
The following was the command when this chapter was written, but please visit the Homebrew website to ensure you are using the current installer:

```
/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
```

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 the current Python 3.x interpreter (emphasis added to output):

```
mbp15:~ sean$ brew search python
==> Formulae
app-engine-python   gst-python        python              python@3.8
boost-python        ipython           python-markdown     wxpython
boost-python3       micropython       python-yq
==> Casks
homebrew/cask/awips-python              homebrew/cask/mysql-connector-python
homebrew/cask/kk7ds-python-runtime
mbp15:~ sean$
```

Install the formula for the Python 3.7.x (shown) or 3.8 interpreter:

```
mbp15:~ sean$ brew install python
==> Downloading https://homebrew.bintray.com/bottles/python-3.7.7.catalina.bottle.tar.gz
==> Downloading from https://akamai.bintray.com/ac/
acd595852aecc2bfa46c57d86db716e64d57bb2753c45ff7f745b4
################################################################## 100.0%
==> Pouring python-3.7.7.catalina.bottle.tar.gz
==> /usr/local/Cellar/python/3.7.7/bin/python3 -s setup.py --no-user-cfg install --force --verbose
--inst
==> /usr/local/Cellar/python/3.7.7/bin/python3 -s setup.py --no-user-cfg install --force --verbose
--inst
==> /usr/local/Cellar/python/3.7.7/bin/python3 -s setup.py --no-user-cfg install --force --verbose
--inst
==> Caveats
Python has been installed as
  /usr/local/bin/python3

You can install Python packages with
  pip3 install <package>
They will install into the site-package directory
  /usr/local/lib/python3.7/site-packages

See: https://docs.brew.sh/Homebrew-and-Python
==> Summary
   /usr/local/Cellar/python/3.7.7: 4,006 files, 61.2MB
```

Now check to see if the newly installed interpreter is the default. Hopefully, you will see different results than before, like this:

```
mbp15:~ sean$ which python
/usr/local/bin/python
mbp15:~ sean$ python --version
Python 3.7.7
```

If the path and version are unchanged from what you saw prior to installing Homebrew, then Homebrew probably did not 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 checking to see if there is a symlink called python or whose name starts with python3:

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

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

mbp15:bin sean$ ls -l python3*
lrwxr-xr-x  1 ssawtell  admin  34 Apr  1 12:20 python3 -> ../Cellar/python/3.7.7/bin/python3
lrwxr-xr-x  1 ssawtell  admin  41 Apr  1 12:20 python3-config -> ../Cellar/python/3.7.7/bin/python3-
config
lrwxr-xr-x  1 ssawtell  admin  36 Apr  1 12:20 python3.7 -> ../Cellar/python/3.7.7/bin/python3.7
lrwxr-xr-x  1 ssawtell  admin  43 Apr  1 12:20 python3.7-config -> ../Cellar/python/3.7.7/bin/
python3.7-config
lrwxr-xr-x  1 ssawtell  admin  37 Apr  1 12:20 python3.7m -> ../Cellar/python/3.7.7/bin/python3.7m
lrwxr-xr-x  1 ssawtell  admin  44 Apr  1 12:20 python3.7m-config -> ../Cellar/python/3.7.7/bin/
python3.7m-config
```

Assuming you get results similar to those shown above, then create a new symlink called python to the same target as your python3 (or python3.7) symlink:

```
mbp15:bin sean$ ln -s ../Cellar/python/3.7.7/bin/python3 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 pip3 instead of python3.

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 (most output is omitted for brevity):

```
mbp15:~ sean$ pip install junos-eznc jxmlease
Collecting junos-eznc
  Downloading junos_eznc-2.4.0-py2.py3-none-any.whl (193 kB)
     |████████████████████████████████| 193 kB 1.3 MB/s
...
Installing collected packages: junos-eznc
Successfully installed junos-eznc-2.4.0
mbp15:~ sean$
```

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

```
mbp15:~ sean$ pip install ansible
Collecting ansible
  Downloading ansible-2.9.6.tar.gz (14.2 MB)
     |████████████████████████████████| 14.2 MB 9.1 MB/s
...
Successfully installed ansible-2.9.6
mbp15:~ sean$
```

TIP   By default, `pip` installs the currently 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. Because `ansible-galaxy` needs to modify some Ansible-related files in the system directory `/etc/`, you may need to `sudo` this command:

```
mbp15:~ sean$ ansible-galaxy install Juniper.junos
- downloading role 'junos', owned by Juniper
- downloading role from https://github.com/Juniper/ansible-junos-stdlib/archive/2.3.1.tar.gz
- extracting Juniper.junos to /Users/ssawtell/.ansible/roles/Juniper.junos
- Juniper.junos (2.3.1) was installed successfully
mbp15:~ sean$
```

TIP   By default, `ansible-galaxy` installs the currently 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: `ansible-galaxy install Juniper.junos,2.2.0`

Installing Ansible on Linux

Due to the variety of Linux distributions and their package managers, it is impossible 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 described below, but exact commands will change depending on the Linux distribution and possibly even the version of that distribution.

The commands that follow were tested with Ubuntu Linux Server versions 18.04 (Bionic Beaver) and 20.04 (Focal Fossa) and should work with minimal modification on other Debian-based distributions. Users of Red Hat or other non-Debian Linux flavors should alter these instructions to use the appropriate package manager 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 python3 python3-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 pip3 install ansible junos-eznc jxmlease
```

Finally, add Juniper's Galaxy modules to Ansible:

```
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 module called `setup`.

For this test, it is normal to get one or more `[WARNING]` lines at the beginning of the output because we have not yet set up an inventory file. The exact warnings differ between Ansible versions.

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

```
mbp15:~ sean$ ansible localhost -m setup
[WARNING]: No inventory was parsed, only implicit localhost is available

localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_XHC20": {
            "device": "XHC20",
            "flags": [],
            "ipv4": [],
            "ipv6": [],
            "macaddress": "unknown",
            "mtu": "0",
            "type": "unknown"
        },
        "ansible_all_ipv4_addresses": [
            "192.168.0.10"
        ],
...
```

```
    "ansible_date_time": {
        "date": "2020-04-03",
        "day": "03",
        "epoch": "1585937010",
        "hour": "14",
        "iso8601": "2020-04-03T18:03:30Z",
        "iso8601_basic": "20200403T140330795614",
        "iso8601_basic_short": "20200403T140330",
        "iso8601_micro": "2020-04-03T18:03:30.795763Z",
        "minute": "03",
        "month": "04",
        "second": "30",
        "time": "14:03:30",
        "tz": "EDT",
        "tz_offset": "-0400",
        "weekday": "Friday",
        "weekday_number": "5",
        "weeknumber": "13",
        "year": "2020"
    },
...
    "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 `-m` argument instructs `ansible` to run the specified module, so `-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:
https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html

Ansible's `setup` module:
http://docs.ansible.com/ansible/latest/setup_module.html

Juniper's Galaxy modules documentation, current stable version:
http://junos-ansible-modules.readthedocs.io/en/stable/

Juniper's resources page for Ansible:
https://www.juniper.net/documentation/en_US/release-independent/junos-ansible/information-products/pathway-pages/index.html

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

Juniper's PyEZ on GitHub:
https://github.com/Juniper/py-junos-eznc

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, it's short.

What are JSON and YAML?

JSON and YAML are both standards for storing, transmitting, or displaying computer 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 written. 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

Before we discuss collections of data (lists and dictionaries), we need to understand 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: 3.14, 21, -5.

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).

Lists

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 elements 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"`.

When a list is assigned to a variable, accessing a single element is done by placing the index in square brackets after the variable name, `variable[index]`. Assuming the list of names above was assigned to a variable called `names`, then `names[2]` would access element 2 in the list, `"Bridget"`.

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]
```

Dictionaries

A *dictionary*, also called an *associative array*, is a collection, like a list, but the elements 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"`. When a dictionary is assigned to a variable, accessing a single value is done by placing the key in square brackets after the variable name, `variable[key]`. Assuming the dictionary above was assigned to a variable called `country`, then `country["name"]` would access `"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 dictionaries.

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 dictionary, where the keys are the names of the weekdays 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 temperatures 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 programming 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 enforces unique keys, so reading JSON data that 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 is 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]},"test2":{"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 improve 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 to see 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

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-formatted 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 for this 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 representation 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 double-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. This example is just to provide an overview (we will discuss the details in a few paragraphs). The last example of JSON data from the previous section can be represented in YAML as follows:

```
test1:
  avg: 71
  sum: 215
  values:
  - 62
  - 74
  - 79
test2:
  avg: 86
  sum: 259
  values:
  - 94
  - 73
  - 92
test3:
  avg: 47
  sum: 142
  values:
  - 2
  - 46
  - 94
```

Notice how the YAML data has less punctuation (no quotes, no braces, no brackets, 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 primarily 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 provide enough visual distinction 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:

```
- element
```

The list `['hello', 'world', 10]` in YAML is:

```
- hello
- world
- 10
```

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:

```
- 0
- - 1
  - 2
- - 3
  - 4
  - - 5
    - 6
- 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 between 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 that is a dictionary, include the leading hyphen only on the first key:value pair that will be part of the nested dictionary. For example, 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
```

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…

```
– k1: v1
– k2: v2
```

…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 elements of the nested list 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:

```
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:

```
k1: v1
k2:
  – v2a
  – v2b
  – v2c
k3: v3
```

Text Editor Tips

As you can see, correct indentation is critical for YAML data. Your text editor can help with this! (If your text editor refuses to help, fire it and hire a new one.)

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 •
or similar 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 common as the others and may not be in your chosen editor.)

The screen capture below 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 write a short playbook that executes a command on Junos devices and displays the command's results. You learn about the structure and contents of a playbook, how to prompt for input, how to display output, one way to send a command to a Junos device, and a little about debugging playbooks.

The author is using a Virtual SRX named *aragorn* and an EX-2200-C named *bilbo* for most examples in this book. You can use any Junos devices you have available, preferably lab or test devices. The book uses only two Ansible-managed devices in order to keep the example output short, but if you have more devices you can run the playbook against all of them if you wish. The author routinely runs his production playbooks against hundreds of devices.

The (manual) Command

Assume you need to find out the date that each network device was booted or last configured, so you can confirm that devices have not been configured or rebooted since the last scheduled maintenance window.

The Junos CLI command for this is "show system uptime" and, when run manually, it will look something like this (the exact output differs based on device hardware, configuration, and uptime):

```
sean@aragorn> 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. We're going to discuss a lot of Ansible fundamentals along the way, so it will be a few pages before we actually start gathering uptime information. Don't worry, we'll get there.

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 book assumes you are using subdirectory `aja2` in your home directory:

```
$ pwd
/Users/sean
$ mkdir aja2
$ cd aja2
$ pwd
/Users/sean/aja2
```

In order to run a basic Ansible playbook you need three files:

- The Ansible configuration file, `ansible.cfg`.

- The inventory file, which we name `inventory`, that contains the list of devices that Ansible might access or manage.

- The playbook file containing 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

Let's 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 `ansible.cfg` in your `~/aja2` 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 current directory) for the list of devices that Ansible will manage.

The line `host_key_checking = False` tells Ansible that it should not use SSH host key checking[1]. Host key checking is desirable from a security perspective but can be a problem with automated connections. Disabling Ansible's host key checking

---

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 different from the ID provided by the server, the client displays an error and aborts the connection.

allows Ansible to connect even if the server's ID is not in the `known_hosts` file (for example, if you have not previously manually connected to that device and cached its 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 available 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 different playbooks (discussed in Chapter 8), and makes it easier to keep the inventory in source control with the playbooks (see the Appendix).

Create a file called `inventory` in your `~/aja2` 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`):

```
aragorn
bilbo
```

Inventory can also include variables, which define additional data about the device. For example, if your playbook needs to know the role of a device in the network (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 information, 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 recommended – 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 managed 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 `aragorn` (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 that does not yet have connectivity back to corporate, you need an alternative. That alternative is the `ansible_host` variable.

If we do not provide a value for `ansible_host`, Ansible automatically populates `ansible_host` with the inventory name, but instead of relying on that fact we populate `ansible_host` with the IP address of the target host. As we create our playbook, we use the `ansible_host` variable to specify the managed device (we will see this shortly).

NOTE   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:

```
aragorn    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 appropriate 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 display the device uptime from our network devices. We will build the playbook in several steps, explaining as we go.

Playbooks are Ansible's "scripts," describing a series of tasks that will be performed 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 *module* (the command) to execute. Tasks usually include *arguments* that provide additional details about how the module should run, such as the network device to control, or the username and password for connecting to the device.

Before we create and run the playbook, we need to discuss one other topic.

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 Python 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:aja2 sean$ which python
/usr/local/bin/python
```

On most UNIX-type systems, the default Python interpreter is `/usr/bin/python`. Ansible 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 the following boldfaced lines to your `inventory` file (use the correct path for your system, as it may be different than the author's system):

```
aragorn    ansible_host=192.0.2.10
bilbo

[all:vars]
ansible_python_interpreter=/usr/local/bin/python
```

The `[all:vars]` line introduces a new section in the inventory file containing variables that apply to all hosts. The next line sets the `ansible_python_interpreter` variable to the correct path for your system (copy whatever `which python` returned).

TIP   On most UNIX-type systems, including MacOS, you can use a trick to avoid worrying about a system-specific path to the Python interpreter. Instead of setting the `ansible_python_interpreter` variable to the actual path to the interpreter, set it to **/usr/bin/env python** (note the space before "python"). This essentially tells the operating system to use the **env** command to find the python interpreter based on the system's path.

Uptime Version 1.0

Create file `uptime.yaml` in your `~/aja2` 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 of the author's text editor 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).

```
1     ---¬
2     - name: Get device uptime¬
3     ··hosts:¬
4     ··· ·- all¬
5     ··connection: local¬
6     ··gather_facts: no¬
7     ¬
8     ··tasks:¬
9     ··· ·- debug: var=inventory_hostname¬
10    ¬
11    ··· ·- debug:¬
12    ··· ··· ·var: ansible_host¬
```

Let's 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 normally significant to Ansible, but it helps document what is happening both to the engineer editing the playbook itself, and during playbook 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: Declare 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:
  – aragorn
  – 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 (the *control machine*) 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 control machine). The line `connection: local` tells Ansible that it cannot upload a Python script to the managed device; instead, it needs to run the playbook locally on the Ansible control machine (even though modules called by the playbook may connect to a network device in order to control it in some fashion).

Line 6: When managing servers, Ansible normally gathers facts—such as operating 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 the default behavior; it tells Ansible to not spend time gathering facts we do not need. (Later in the book we provide 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. Despite 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 hyphen – 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 contents of the variable (`var`) called `inventory_hostname`.

Lines 11-12: Another task calling the debug module to print a variable's contents, but showing another way to provide arguments 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 remember that the ansible_python_interpreter variable is needed only if your system's Python interpreter is in a non-standard location:

```
aragorn    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 ~/aja2 directory (where your playbook and inventory files are located) then run the playbook:

```
mbp15:aja2 sean$ pwd
/Users/sean/aja2

mbp15:aja2 sean$ ansible-playbook uptime.yaml

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

TASK [debug] *******************************************************
ok: [aragorn] => {
    "inventory_hostname": "aragorn"
}
ok: [bilbo] => {
    "inventory_hostname": "bilbo"
}

TASK [debug] *******************************************************
ok: [aragorn] => {
    "ansible_host": "192.0.2.10"
}
ok: [bilbo] => {
    "ansible_host": "bilbo"
}

PLAY RECAP *********************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
bilbo                      : 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 device 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 provide names for the tasks, so Ansible displays the module name debug instead.

ok: [aragorn] 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 variable.

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 at all because the target was unreachable 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 display in green, and in the Play Recap section, devices for which all tasks returned ok will be in green:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml

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

TASK [debug] ************************************************************
ok: [aragorn] => {
    "inventory_hostname": "aragorn"
}
ok: [bilbo] => {
    "inventory_hostname": "bilbo"
}

TASK [debug] ************************************************************
ok: [aragorn] => {
    "ansible_host": "192.0.2.10"
}
ok: [bilbo] => {
    "ansible_host": "bilbo"
}

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

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 aragorn has an IP address, while bilbo has a hostname. This difference is because of the author's inventory file, which contains:

```
aragorn    ansible_host=192.0.2.10
bilbo
```

The inventory line for device `aragorn` 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:aja2 sean$ ansible-playbook uptime.yaml

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

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "inventory_hostname": "aragorn"
}
ok: [bilbo] => {
    "inventory_hostname": "bilbo"
}

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "ansible_hst": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "ansible_hst": "VARIABLE IS NOT DEFINED!"
}

PLAY RECAP ********************************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
bilbo                      : 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 use the Galaxy module `juniper_junos_command` to communicate with our devices and execute the "show system uptime" command. This module needs several arguments: the command to execute, the device to communicate with, and credentials for authenticating with the device.

Because we need to authenticate with the devices, our playbook must have a username 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
  roles:
    - Juniper.junos
  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 galaxy module
      juniper_junos_command:
        commands:
          - show system uptime
        provider:
          host: "{{ ansible_host }}"
          port: 22
          user: "{{ username }}"
          passwd: "{{ password }}"
```

A screen capture showing the same playbook with line numbers, etc:

```
 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    ······juniper_junos_command:¬
22    ········commands:¬
23    ··········- show system uptime¬
24    ········provider:¬
25    ··········host: "{{ ansible_host }}"¬
26    ··········port: 22¬
27    ··········user: "{{ username }}"¬
28    ··········passwd: "{{ password }}"¬
```

Again, let's discuss the playbook's contents by line number, focusing on the new or changed lines.

Lines 5-6: Include the `Juniper.junos` Galaxy modules, which enable Ansible to communicate with Junos devices. Before we can use Galaxy modules, we need to import their parent role into the play. A `roles:` list tells Ansible to include the functions in the specified roles. We discuss roles in detail in Chapter 12.

Line 10: The `vars_prompt:` line introduces a list, each element of which is a dictionary, 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 available 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 `2day` (starts with a numeral) and `task-results` (contains a hyphen). Variable names are case sensitive: `test1` and `Test1` are different variables.

Lines 11-13: The first dictionary in the `vars_prompt` list. Line 11 tells Ansible to put the user's input in a variable named `username`. Line 12 tells Ansible to display "Junos Username" as the prompt for input. Line 13 says the input is not private (the user will be able to see what they type).

Lines 15-17: The second dictionary in `vars_prompt` list. This time the input is stored in a variable called `password` and is private, meaning Ansible will not display what the user is typing.

Lines 20-28: Define a task named "get uptime using galaxy module" that calls the Ansible module `juniper_junos_command`. This task passes two arguments to `juniper_junos_command`. The first argument is `commands`, which is a list of Junos commands to execute (our playbook has only one element in the list); the second argument is `provider`, which is a dictionary that describes how to access the target device.

The `provider` dictionary (lines 24-28) has four entries (key:value pairs):

`host` (line 25) specifies the device on which Ansible should execute the commands; this is assigned the value of the device's `ansible_host` variable.

`port` (line 26) specifies the TCP port that Ansible should use for the connection; we specify the standard SSH port 22. We discuss the connection further in Chapter 5.

`user` and `passwd` (lines 27 and 28) are the credentials for accessing the device; these are assigned the values provided by the user in the `vars_prompt` portion of the playbook via the `username` and `password` variables.

As you can see in lines 25, 27, and 28, Ansible uses `{{ variable_name }}` to say "put the value of variable `variable_name` here." However, YAML considers { } to be a

dictionary, which would result in an error because {{ variable_name }} is not a valid dictionary. To make YAML happy, we need to include the variable reference in quotes – "{{ }}" – so YAML sees it as a string, leaving interpretation of the variable to Ansible.

Let's run the playbook!

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************
ok: [aragorn]
ok: [bilbo]

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

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

NOTE   Despite the fact that the juniper_junos_command module accepts a Junos CLI command, it does not connect to a Junos device's CLI and run the command the way you would. It actually calls the Junos API (application programming interface), specifically an RPC (remote procedure call) called <command> that executes the CLI command. Chapter 5 introduces RPCs in more detail.

Uptime Version 1.2

Previously we saw 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 currently constructed, the uptime values are lost. We need to assign the results of the juniper_junos_command module to a variable, which we can do by adding register: uptime to that task, where uptime is the name of the variable in which the task's output will be stored:

```
- name: get uptime using galaxy module
  juniper_junos_command:
    commands:
      - show system uptime
    provider:
      host: "{{ ansible_host }}"
      port: 22
      user: "{{ username }}"
      passwd: "{{ password }}"
  register: uptime
```

Note that the `register` argument is indented to the same level as the task's `name` and module (`juniper_junos_command`). `Register` is an argument for the task itself and needs to be at the same indentation level as other task entries. By contrast, `commands` and `provider` are arguments to the module `juniper_junos_command`, which is why they are indented further than the module's name.

We also need to add a task that calls the `debug` module to display the contents of the new `uptime` variable. This time let's give the `debug` task a name:

```
- name: display uptimes
  debug:
    var: uptime
```

The complete modified playbook (lines 29–33 were 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      ······juniper_junos_command:¬
22      ········commands:¬
23      ··········- show system uptime¬
24      ········provider:¬
25      ··········host: "{{ ansible_host }}"¬
26      ··········port: 22¬
27      ··········user: "{{ username }}"¬
28      ··········passwd: "{{ password }}"¬
29      ······register: uptime¬
30    ¬
31      ····- name: display uptimes¬
32      ······debug:¬
33      ········var: uptime¬
```

Let's run the playbook:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>
```

```
PLAY [Get device uptime] ********************************************

TASK [get uptime using galaxy module] ******************************
ok: [aragorn]
ok: [bilbo]

TASK [display uptimes] *********************************************
ok: [aragorn] => {
    "uptime": {
        "changed": false,
        "command": "show system uptime",
        "failed": false,
        "format": "text",
        "msg": "The command executed successfully.",
      "stdout": "\nCurrent time: 2018-02-26 18:34:07 UTC\nTime Source:  NTP CLOCK \nSystem booted:
2018-02-26 07:10:44 UTC (11:23:23 ago)\nProtocols started: 2018-02-26 07:10:45 UTC (11:23:22 ago)\
nLast configured: 2018-02-26 03:04:10 UTC (15:29:57 ago) by sean\n 6:34PM  up 11:23, 1 user, load
averages: 0.00, 0.00, 0.00\n",
        "stdout_lines": [
            "",
            "Current time: 2018-02-26 18:34:07 UTC",
            "Time Source:  NTP CLOCK ",
            "System booted: 2018-02-26 07:10:44 UTC (11:23:23 ago)",
            "Protocols started: 2018-02-26 07:10:45 UTC (11:23:22 ago)",
            "Last configured: 2018-02-26 03:04:10 UTC (15:29:57 ago) by sean",
            " 6:34PM  up 11:23, 1 user, load averages: 0.00, 0.00, 0.00"
        ]
    }
}
ok: [bilbo] => {
    "uptime": {
        "changed": false,
        "command": "show system uptime",
        "failed": false,
        "format": "text",
        "msg": "The command executed successfully.",
        "stdout": "\
nfpc0:\n--------------------------------------------------------------------------------\nCurrent time:
2018-02-26 18:34:12 UTC\nSystem booted: 2018-02-25 18:57:45 UTC (23:36:27 ago)\nProtocols started:
2018-02-25 19:00:54 UTC (23:33:18 ago)\nLast configured: 2018-02-26 16:17:54 UTC (02:16:18 ago) by
sean\n 6:34PM  up 23:36, 1 user, load averages: 0.08, 0.03, 0.01\n",
        "stdout_lines": [
            "",
            "fpc0:",
            "--------------------------------------------------------------------------------",
            "Current time: 2018-02-26 18:34:12 UTC",
            "System booted: 2018-02-25 18:57:45 UTC (23:36:27 ago)",
            "Protocols started: 2018-02-25 19:00:54 UTC (23:33:18 ago)",
            "Last configured: 2018-02-26 16:17:54 UTC (02:16:18 ago) by sean",
            " 6:34PM  up 23:36, 1 user, load averages: 0.08, 0.03, 0.01"
        ]
    }
}

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

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

Notice that the output of the `display uptimes` task is in JSON format, and that each device has its own `uptime` variable, registered (created) by the "get uptime using galaxy module" task. The `uptime` variables each contain a dictionary with a number of values, including the following:

`stdout` – the complete Junos command output as a single string.

`stdout_lines` – a list (array) where each element is one line of Junos command output.

`changed` – Boolean *true* if the module changed something (for example, the device's configuration), *false* otherwise.

`failed` – Boolean *true* if the module reported a failure, *false* otherwise.

`format` – the type of output, where "text" indicates human-readable text as seen at the Junos CLI. May also be "xml" or "json" – we discuss those options in Chapter 5.

MORE? Notice that the `commands` argument on line 22 seems to introduce a *list of commands* (note the leading hyphen on the command "`- show system uptime`"), though our example has only a single command in the list. The `juniper_junos_command` module supports executing multiple commands in one task (one call to the module). After you understand the use of this module with a single command, you may wish to read the online documentation to understand how the module handles a list of commands and their results.

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 just 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. Because the key is a string it needs to be quoted:

```
- name: display uptimes (Python style)
  debug:
    var: uptime['stdout_lines']
```

NOTE   You can use single-quotes (' ') or double-quotes (" ") around the key string.

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

```
– name: display uptimes (Ansible shortcut)
  debug:
    var: uptime.stdout_lines
```

The modified playbook is shown below. Both approaches discussed above are shown (see lines 33 and 37) 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. Comments are normally used to include notes about the playbook within the playbook itself, as documentation for anyone editing the playbook, but can also be used as shown here to disable (usually temporarily) specific lines of the playbook:

```
 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    ······juniper_junos_command:¬
22    ········commands:¬
23    ··········- show system uptime¬
24    ········provider:¬
25    ··········host: "{{ ansible_host }}"¬
26    ··········port: 22¬
27    ··········user: "{{ username }}"¬
28    ··········passwd: "{{ password }}"¬
29    ······register: uptime¬
30    ¬
31    ····# - name: display uptimes (Python style)¬
32    ····#   debug:¬
33    ····#     var: uptime['stdout_lines']¬
34    ¬
35    ····- name: display uptimes (Ansible shortcut)¬
36    ······debug:¬
37    ········var: uptime.stdout_lines¬
```

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

```
TASK [display uptimes (Ansible shortcut)] **************************************
ok: [aragorn] => {
    "uptime.stdout_lines": [
        "",
        "Current time: 2018-02-26 19:22:08 UTC",
        "Time Source:  NTP CLOCK ",
        "System booted: 2018-02-26 07:10:44 UTC (12:11:24 ago)",
        "Protocols started: 2018-02-26 07:10:45 UTC (12:11:23 ago)",
        "Last configured: 2018-02-26 03:04:10 UTC (16:17:58 ago) by sean",
        " 7:22PM  up 12:11, 1 user, load averages: 0.08, 0.02, 0.01"
    ]
}
ok: [bilbo] => {
    "uptime.stdout_lines": [
        "",
        "fpc0:",
        "--------------------------------------------------------------------------",
        "Current time: 2018-02-26 19:22:13 UTC",
        "System booted: 2018-02-25 18:57:45 UTC (1d 00:24 ago)",
        "Protocols started: 2018-02-25 19:00:54 UTC (1d 00:21 ago)",
        "Last configured: 2018-02-26 16:17:54 UTC (03:04:19 ago) by sean",
        " 7:22PM  up 1 day, 24 mins, 1 user, load averages: 0.24, 0.06, 0.02"
    ]
}
```

Uncomment lines 31–33 (delete the leading '#' and the space after it) and comment out lines 35–37 (add a leading '#'). Run the playbook again. The output should be essentially the same, demonstrating that the two approaches for referencing entries in a dictionary are equivalent.

You can take this a step further, displaying a specific element from the uptime.stdout_lines list, by appending the index of the element in square brackets. The first list element has an index of 0, the next an index of 1, etc. So, for example, uptime.stdout_lines[2] would reference element at index 2 (the third element) of the list. With some commands, which have very consistent output across different device types, this may give us exactly what we want.

Unfortunately, the output of this command on different devices puts similar information in different indexes of the list. For example, if we specify we want only element 5 by modifying the playbook as follows:

```
- name: display uptimes (Ansible shortcut)
  debug:
    var: uptime.stdout_lines[5]
```

We get output similar to the following:

```
...
TASK [display uptimes (Ansible shortcut)] **************************************
ok: [aragorn] => {
    "uptime.stdout_lines[5]": "Last configured: 2018-02-27 19:38:47 UTC (2d 08:55 ago) by sean"
}
```

```
ok: [bilbo] => {
    "uptime.stdout_lines[5]": "Protocols started: 2018-02-25 19:00:54 UTC (4d 23:49 ago)"
}
...
```

Observe that we get the "Last configured" information for *aragorn*, but the "Protocols started" information for *bilbo*. We discuss two different approaches in Chapter 5 that will let us get the data we want despite output differences.

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 unreachable devices or authentication errors, not syntax or other errors within the playbook.

Ansible tracks errors separately for each device. When an error related to a particular device occurs, Ansible stops processing that device; subsequent tasks will not execute for it. However, if other devices have *not* had errors, tasks for those devices may be executed.

TIP   It is possible, and occasionally useful, to have Ansible ignore errors in a particular task and continue processing a device despite errors, by adding the argument `ignore_errors: yes` to the task where errors are expected.

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 image:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password:

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

TASK [get uptime using galaxy module] *******************************
ok: [aragorn]
fatal: [bilbo]: FAILED! => {"changed": false, "msg": "Unable to make a PyEZ conn
ection: ConnectTimeoutError(bilbo)"}

TASK [display uptimes (Ansible shortcut)] ***************************
ok: [aragorn] => {
    "uptime.stdout_lines": [
        "",
        "Current time: 2018-02-26 19:40:11 UTC",
        "Time Source:  NTP CLOCK ",
        "System booted: 2018-02-26 07:10:44 UTC (12:29:27 ago)",
        "Protocols started: 2018-02-26 07:10:45 UTC (12:29:26 ago)",
        "Last configured: 2018-02-26 03:04:10 UTC (16:36:01 ago) by sean",
        " 7:40PM  up 12:29, 1 user, load averages: 0.00, 0.00, 0.00"
    ]
}
        to retry, use: --limit @/Users/sean/aja2/uptime.retry

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

The results for TASK [get uptime using galaxy module] show that *aragorn* succeeded – ok: [aragorn] – 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 juniper_junos_command module returned the error message "Unable to make a PyEZ connection: ConnectTimeoutError(bilbo)" seen above for *bilbo*. The "ConnectTimeoutError" part makes sense – because *bilbo* was unreachable (disconnected) any attempt to connect to *bilbo* would have timed out. The reference to "a PyEZ connection" illustrates that Juniper's Galaxy modules rely on Juniper's PyEZ connection framework.

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

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: tracy
Junos Password:

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

TASK [get uptime using galaxy module] ********************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Unable to make a PyEZ co
nnection: ConnectAuthError(192.0.2.10)"}
fatal: [bilbo]: FAILED! => {"changed": false, "msg": "Unable to make a PyEZ conn
ection: ConnectAuthError(bilbo)"}
        to retry, use: --limit @/Users/sean/aja2/uptime.retry

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

Note that the results for TASK [get uptime using galaxy module] show both devices failed: fatal: [aragorn] and fatal: [bilbo], each followed by the error message "Unable to make a PyEZ connection: ConnectAuthError." The "ConnectAuthError" part of the message shows we had an authentication failure.

Notice that TASK [display uptimes] never executed (it does not appear in the output). 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:
    - aragorn
    - newdevice
...
```

The problem with this approach is that it requires updating the playbook to change the devices being managed, and doing so in a way that may not be obvious to someone else who uses the playbook and expects it to work on all, or a different subset of, your devices. 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:aja2 sean$ ansible-playbook uptime.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************************
ok: [aragorn]

TASK [display uptimes (Ansible shortcut)] **************************************
ok: [aragorn] => {
    "uptime.stdout_lines": [
        "",
        "Current time: 2018-02-26 21:12:37 UTC",
        "Time Source:  NTP CLOCK ",
        "System booted: 2018-02-26 07:10:44 UTC (14:01:53 ago)",
        "Protocols started: 2018-02-26 07:10:45 UTC (14:01:52 ago)",
        "Last configured: 2018-02-26 03:04:10 UTC (18:08:27 ago) by sean",
        " 9:12PM  up 14:02, 1 user, load averages: 0.00, 0.00, 0.00"
    ]
}

PLAY RECAP *********************************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
```

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

The --limit argument can accept multiple inventory names separated with commas, and can accept the wildcards '*' (match zero or more characters) and '?' (match any single character). In Chapter 8, we discuss inventory groups; --limit can accept group names also. Two command-line examples:

```
ansible-playbook uptime.yaml --limit=aragorn,newdevice

ansible-playbook uptime.yaml --limit='b*'
```

Note that when using a wildcard, you should enclose the limit value in quotes to prevent the shell from attempting to interpret the wildcard.

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 argument causes Ansible to display which devices it will manage, but not actually run the playbook. For example:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml --limit='b*' --list-hosts

playbook: uptime.yaml

  play #1 (all): Get device uptimeTAGS: []
    pattern: [u'all']
    hosts (1):
      bilbo
```

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 extension `.retry` instead of `.yaml`. By default, "retry" files are stored in the playbook directory.

TIP   You can change the directory where Ansible saves "retry" files by adding the option `retry_files_save_path` to the `[defaults]` section of the `ansible.cfg` file.

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

```
mbp15:aja2 sean$ ls *.retry
uptime.retry

mbp15:aja2 sean$ cat uptime.retry
aragorn
bilbo
```

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:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************
ok: [aragorn]
fatal: [bilbo]: FAILED! => {"changed": false, "msg": "Unable to make a PyEZ connection:
ConnectTimeoutError(bilbo)"}

TASK [display uptimes (Ansible shortcut)] **************************
ok: [aragorn] => {
    "uptime.stdout_lines": [
        "",
        "Current time: 2018-02-26 21:30:05 UTC",
        "Time Source:  NTP CLOCK ",
        "System booted: 2018-02-26 07:10:44 UTC (14:19:21 ago)",
        "Protocols started: 2018-02-26 07:10:45 UTC (14:19:20 ago)",
        "Last configured: 2018-02-26 03:04:10 UTC (18:25:55 ago) by sean",
        " 9:30PM  up 14:19, 1 user, load averages: 0.05, 0.02, 0.00"
    ]
}
        to retry, use: --limit @/Users/sean/aja2/uptime.retry

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

mbp15:aja2 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 any 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, prefixing the filename with an "at sign" ("@"), like this:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml --limit=@uptime.retry
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************
ok: [bilbo]

TASK [display uptimes (Ansible shortcut)] **************************
ok: [bilbo] => {
    "uptime.stdout_lines": [
        "",
        "fpc0:",
        "----------------------------------------------------------------",
        "Current time: 2018-02-26 21:32:57 UTC",
        "System booted: 2018-02-25 18:57:45 UTC (1d 02:35 ago)",
        "Protocols started: 2018-02-25 19:00:54 UTC (1d 02:32 ago)",
        "Last configured: 2018-02-26 21:01:01 UTC (00:31:56 ago) by sean",
        " 9:32PM  up 1 day,  2:35, 0 users, load averages: 0.08, 0.02, 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/aja2/uptime.retry`."

With only one failed device out of only two test devices, it would be easy to just specify `--limit=bilbo` for the repeat run. Consider, however, what it would be like when running a playbook against 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 abruptly 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 following errors one at a time, and reverse each before proceeding to the next one. The line numbers refer to the screen capture of uptime.yaml 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:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>
ERROR! the field 'hosts' is required but was not set
```

The error message "the field 'hosts' is required but was not set" exactly describes the problem we introduced.

Incorrect Indentation

Remove two spaces from the beginning of lines 15-17, 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:aja2 sean$ ansible-playbook uptime.yaml
ERROR! Syntax Error while loading YAML.


The error appears to have been in '/Users/sean/aja2/uptime.yaml': line 15, column 5, but may
be elsewhere in the file depending on the exact syntax problem.
```

```
The offending line appears to be:
```

```
    – name: password
    ^ here
```

Recall that playbooks are YAML files, and YAML is sensitive to correct indentation. Because the password prompt lines were not correctly indented, Ansible detected a problem trying to load the playbook.

Unmatched (missing) Quotation Mark

Delete the quotation mark from the end of line 25; the revised line should be:

```
host: "{{ ansible_host }}
```

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

```
mbp15:aja2 sean$ ansible–playbook uptime.yaml
ERROR! Syntax Error while loading YAML.


The error appears to have been in '/Users/sean/aja2/uptime.yaml': line 27, column 20, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

          port: 22
          user: "{{ 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 27 as the likely problem, not line 25. 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. Please keep in mind that 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 command-line argument –v to the `ansible-playbook` command, like this:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml --limit=bilbo -v
Using /Users/sean/aja2/ansible.cfg as config file
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************************
ok: [bilbo] => {"changed": false, "command": "show system uptime", "format": "text", "msg": "The
command executed successfully.", "stdout": "\
nfpc0:\n--------------------------------------------------------------------------\nCurrent time:
2018-02-27 17:00:18 UTC\nSystem booted: 2018-02-25 18:57:45 UTC (1d 22:02 ago)\nProtocols started:
2018-02-25 19:00:54 UTC (1d 21:59 ago)\nLast configured: 2018-02-26 21:01:01 UTC (19:59:17 ago) by
sean\n 5:00PM  up 1 day, 22:03, 0 users, load averages: 0.08, 0.02, 0.01\n", "stdout_lines": ["",
"fpc0:", "--------------------------------------------------------------------------", "Current time:
2018-02-27 17:00:18 UTC", "System booted: 2018-02-25 18:57:45 UTC (1d 22:02 ago)", "Protocols started:
2018-02-25 19:00:54 UTC (1d 21:59 ago)", "Last configured: 2018-02-26 21:01:01 UTC (19:59:17 ago) by
sean", " 5:00PM  up 1 day, 22:03, 0 users, load averages: 0.08, 0.02, 0.01"]}

TASK [display uptimes (Ansible shortcut)] **************************************
ok: [bilbo] => {
    "uptime.stdout_lines": [
        "",
        "fpc0:",
        "--------------------------------------------------------------------------",
        "Current time: 2018-02-27 17:00:18 UTC",
        "System booted: 2018-02-25 18:57:45 UTC (1d 22:02 ago)",
        "Protocols started: 2018-02-25 19:00:54 UTC (1d 21:59 ago)",
        "Last configured: 2018-02-26 21:01:01 UTC (19:59:17 ago) by sean",
        " 5:00PM  up 1 day, 22:03, 0 users, load averages: 0.08, 0.02, 0.01"
    ]
}

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

Observe the additional details provided by verbose mode (–v), including the name of the config file and the data returned by the `juniper_junos_command` module.

You can get still more detail by using –vv or –vvv; each "v" adds a little more "verbosity" 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 removed 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 `juniper_junos_command` module. However, if we add the `de‐bug` calls back in the way we had them originally, the playbook would *always* display the variable's contents, even when we were *not* troubleshooting, which means most of the time we would be getting information we do not need.

We can ask `debug` to display the variable's data only when we have enabled verbose mode as described above. Add lines 20-24 shown in the screen capture below into your `uptime.yaml` playbook:

```
19    ··tasks:¬
20    ··· ·─ name: show ansible_host in verbose mode¬
21    ··· ··· debug:¬
22    ··· ··· ·var: ansible_host¬
23    ··· ··· ·verbosity: 1¬
24    ¬
25    ··· ·─ name: get uptime using galaxy module¬
26    ··· ··· juniper junos command:¬
```

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 module 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.

When you run the playbook with (–v) Ansible will run the task and display the `ansible_host` variables:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml -v
...
TASK [show ansible_host in verbose mode] ****************************************
ok: [aragorn] => {
    "ansible_host": "192.0.2.10"
}
ok: [bilbo] => {
    "ansible_host": "bilbo"
}
...
```

However, when you run the playbook *without* enabling verbose mode (no –v argument), Ansible will skip that task for each host:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
...
TASK [show ansible_host in verbose mode] ****************************************
skipping: [aragorn]
skipping: [bilbo]
...
```

Logging

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):

```
[defaults]
inventory = inventory
host_key_checking = False
log_path = ~/aja2/ansible.log
```

Now when you run the playbook (not shown), Ansible will log the playbook's output and some additional details:

```
mbp15:aja2 sean$ cat ansible.log
2018-02-27 12:19:39,833 p=54882 u=sean | PLAY [Get device uptime] ***********************************
*******************
2018-02-27 12:19:39,849 p=54882 u=sean | TASK [show ansible_host in verbose mode] *******************
*******************
2018-02-27 12:19:39,881 p=54882 u=sean | skipping: [aragorn]
2018-02-27 12:19:39,889 p=54882 u=sean | skipping: [bilbo]
2018-02-27 12:19:39,999 p=54882 u=sean | TASK [get uptime using galaxy module] *********************
*******************
2018-02-27 12:19:42,047 p=54882 u=sean | ok: [aragorn]
2018-02-27 12:19:47,706 p=54882 u=sean | ok: [bilbo]
2018-02-27 12:19:47,712 p=54882 u=sean | TASK [display uptimes (Ansible shortcut)] *****************
*******************
2018-02-27 12:19:47,751 p=54882 u=sean |  ok: [aragorn] => {
    "uptime.stdout_lines": [
        "",
        "Current time: 2018-02-27 17:19:41 UTC",
        "Time Source:  NTP CLOCK ",
        "System booted: 2018-02-26 10:22:47 UTC (1d 06:56 ago)",
        "Protocols started: 2018-02-26 10:22:48 UTC (1d 06:56 ago)",
        "Last configured: 2018-02-26 03:04:10 UTC (1d 14:15 ago) by sean",
        " 5:19PM  up 1 day,  6:57, 1 user, load averages: 0.00, 0.00, 0.00"
    ]
}
2018-02-27 12:19:47,757 p=54882 u=sean |  ok: [bilbo] => {
    "uptime.stdout_lines": [
        "",
        "fpc0:",
        "--------------------------------------------------------------------------",
        "Current time: 2018-02-27 17:19:46 UTC",
        "System booted: 2018-02-25 18:57:45 UTC (1d 22:22 ago)",
        "Protocols started: 2018-02-25 19:00:54 UTC (1d 22:18 ago)",
        "Last configured: 2018-02-26 21:01:01 UTC (20:18:45 ago) by sean",
        " 5:19PM  up 1 day, 22:22, 0 users, load averages: 0.08, 0.04, 0.01"
    ]
}
2018-02-27 12:19:47,760 p=54882 u=sean | PLAY RECAP ************************************************
*******************
2018-02-27 12:19:47,761 p=54882 u=sean | aragorn                    : ok=2    changed=0    unreachable=0
failed=0
2018-02-27 12:19:47,761 p=54882 u=sean | bilbo                      : ok=2    changed=0    unreachable=0
failed=0
```

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

Example Ansible playbooks by another "Juniperite," Khelil Sator:
https://github.com/ksator/junos-automation-with-ansible

Juniper's documentation for their Galaxy modules:
http://junos-ansible-modules.readthedocs.io/en/stable/index.html

https://www.juniper.net/documentation/en_US/release-independent/junos-ansible/
information-products/pathway-pages/index.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

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, including some options available when running playbooks and some troubleshooting tips.

This chapter starts with a little theory about the Junos management architecture, briefly introducing RPC, NETCONF, and XML. We then revise the playbook from Chapter 4 to explore more options for running commands on Junos devices using Ansible.

Junos Management Architecture

Junos includes a management daemon (process), MGD, that is responsible for executing 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 computer-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 at the end of the chapter contains links for those who wish to investigate 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@aragorn> show system alarms
1 alarms currently active
Alarm time                 Class  Description
2018-02-25 15:35:46 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@aragorn> show system alarms | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">
    <alarm-information xmlns="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="1519572946">
                2018-02-25 15:35:46
            </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 *elements*, 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>.

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 remote procedure call (RPC) from the CLI to the MGD process. We will revisit this idea in a few paragraphs.

NETCONF

The Network Configuration Protocol (NETCONF) is a standard protocol for remote 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 management 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 NETCONF connections even without the settings above configured. 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 5
set system services netconf ssh rate-limit 5
```

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 service 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 appropriate changes on your test systems:

```
[edit]
sean@aragorn# 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. Doing so requires knowing the RPCs (or using an automation platform that knows the RPCs). Fortunately, Junos makes it easy to find the RPC equivalent for most CLI commands: append " | display xml rpc" to the command at the Junos CLI. For example:

```
sean@aragorn> show system alarms | display xml rpc
<rpc-reply xmlns: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@aragorn> show interfaces terse lo0 | display xml rpc
<rpc-reply xmlns: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 expressed 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 devices and determine the RPC for the *show system uptime* command as discussed above:

```
sean@aragorn> show system uptime | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">
    <rpc>
        <get-system-uptime-information>
        </get-system-uptime-information>
    </rpc>
    <cli>
        <banner></banner>
    </cli>
</rpc-reply>
```

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

Juniper's Galaxy module for calling Junos RPCs is `juniper_junos_rpc`. The arguments are similar to the `juniper_junos_command` module we used in Chapter 4, but instead of a `commands` argument with a list of CLI commands, it uses a `rpcs` argument with a list of RPC names. Also, the `juniper_junos_rpc` module will (by default) return results as XML data, not text.

Modify the `uptime.yaml` playbook so it looks like the following (line numbers have been added for discussion, but do not include the line numbers or 'l' separators in your playbook; lines added or changed are **boldfaced**):

```
 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|      juniper_junos_rpc:
22|        rpcs:
23|          - get-system-uptime-information
24|        provider:
25|          host: "{{ ansible_host }}"
26|          port: 22
27|          user: "{{ username }}"
28|          passwd: "{{ password }}"
29|      register: uptime
30|
31|    - name: display uptimes
32|      debug:
33|        var: uptime
```

The changes made were, by line number:

After line 19: Remove the debug task with the verbosity option.

Line 21: Change juniper_junos_command module to juniper_junos_rpc module.

Lines 22 and 23: Change commands list to rpcs list.

After line 29: Remove the commented-out debug task.

Line 33: Adjust the debug task to display the full uptime variable, not just the uptime.stdout_lines element.

Also note that line 26, port: 22, was left in place so this playbook will use the standard SSH port for NETCONF, not the preferred port 830. This was done to illustrate that it works.

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

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] *******************************
ok: [aragorn]

TASK [display uptimes] **********************************************
ok: [aragorn] => {
    "uptime": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "xml",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "parsed_output": {
            "system-uptime-information": {
                "current-time": {
```

```
                    "date-time": "2018-02-27 23:17:20 UTC"
                },
                "last-configured-time": {
                    "date-time": "2018-02-27 19:38:47 UTC",
                    "time-length": "03:38:33",
                    "user": "sean"
                },
                "protocols-started-time": {
                    "date-time": "2018-02-26 11:31:15 UTC",
                    "time-length": "1d 11:46"
                },
                "system-booted-time": {
                    "date-time": "2018-02-26 11:31:15 UTC",
                    "time-length": "1d 11:46"
                },
                "time-source": "LOCAL CLOCK",
                "uptime-information": {
                    "active-user-count": "2",
                    "date-time": "11:17PM",
                    "load-average-1": "0.05",
                    "load-average-15": "0.00",
                    "load-average-5": "0.01",
                    "up-time": "1 day, 11:46"
                }
            }
        },
        "rpc": "get-system-uptime-information",
        "stdout": "<system-uptime-information>\n  <current-time>\n    <date-
time seconds=\"1519773440\">2018-02-27 23:17:20 UTC</date-time>\n  </
current-time>\n  <time-source>LOCAL CLOCK</
time-source>\n  <system-booted-time>\n    <date-
time seconds=\"1519644675\">2018-02-26 11:31:15 UTC</
date-time>\n    <time-length seconds=\"128765\">1d 11:46</time-length>\n  </
system-booted-time>\n  <protocols-started-time>\n    <date-
time seconds=\"1519644675\">2018-02-26 11:31:15 UTC</
date-time>\n    <time-length seconds=\"128765\">1d 11:46</time-length>\n  </protocols-started-
time>\n  <last-configured-time>\n    <date-time seconds=\"1519760327\">2018-02-27 19:38:47 UTC</
date-time>\n    <time-length seconds=\"13113\">03:38:33</time-length>\n    <user>sean</user>\n  </
last-configured-time>\n  <uptime-information>\n    <date-time seconds=\"1519773440\">11:17PM</
date-time>\n    <up-time seconds=\"128795\">1 day, 11:46</
up-time>\n    <active-user-count format=\"2 users\">2</
active-user-count>\n    <load-average-1>0.05</load-average-1>\n    <load-average-5>0.01</load-
average-5>\n    <load-average-15>0.00</load-average-15>\n  </uptime-information>\n</system-uptime-
information>\n",
        "stdout_lines": [
            "<system-uptime-information>",
            "  <current-time>",
            "    <date-time seconds=\"1519773440\">2018-02-27 23:17:20 UTC</date-time>",
            "  </current-time>",
            "  <time-source>LOCAL CLOCK</time-source>",
            "  <system-booted-time>",
            "    <date-time seconds=\"1519644675\">2018-02-26 11:31:15 UTC</date-time>",
            "    <time-length seconds=\"128765\">1d 11:46</time-length>",
            "  </system-booted-time>",
            "  <protocols-started-time>",
            "    <date-time seconds=\"1519644675\">2018-02-26 11:31:15 UTC</date-time>",
            "    <time-length seconds=\"128765\">1d 11:46</time-length>",
            "  </protocols-started-time>",
```

```
            "   <last-configured-time>",
            "     <date-time seconds=\"1519760327\">2018-02-27 19:38:47 UTC</date-time>",
            "     <time-length seconds=\"13113\">03:38:33</time-length>",
            "     <user>sean</user>",
            "   </last-configured-time>",
            "   <uptime-information>",
            "     <date-time seconds=\"1519773440\">11:17PM</date-time>",
            "     <up-time seconds=\"128795\">1 day, 11:46</up-time>",
            "     <active-user-count format=\"2 users\">2</active-user-count>",
            "     <load-average-1>0.05</load-average-1>",
            "     <load-average-5>0.01</load-average-5>",
            "     <load-average-15>0.00</load-average-15>",
            "   </uptime-information>",
            "</system-uptime-information>"
        ]
    }
}

PLAY RECAP ***********************************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
```

At first glance, the output looks similar to what we saw from version 1.2 of our uptime.yaml playbook. However, notice that much of the output is now in XML, not easily readable text. Also notice that there are some new or different keys in the uptime dictionary, including one called parsed_output that we will discuss in the next section of this chapter, and the format key now has the value xml confirming the observation that we have XML data.

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 the section *Path to the Python Interpreter* in Chapter 4.

Parsed Output – Uptime Version 2.1

Processing XML data can be tricky (though later in this chapter we discuss XPath, a tool that helps). The parsed_output data returned by juniper_junos_rpc is a JSON representation of the raw XML data, intended to make it easier to access the data in an Ansible playbook.

Let's modify the playbook to show just the parsed_output element of the uptime registered variable:

```
...
31|    - name: display uptimes
32|      debug:
33|        var: uptime.parsed_output
```

Run the playbook again and take look at the results:

```
...
TASK [display uptimes] **********************************************************
ok: [aragorn] => {
```

```
        "uptime.parsed_output": {
            "system-uptime-information": {
                "current-time": {
                    "date-time": "2018-03-02 20:28:29 UTC"
                },
                "last-configured-time": {
                    "date-time": "2018-02-27 19:38:47 UTC",
                    "time-length": "3d 00:49",
                    "user": "sean"
                },
                "protocols-started-time": {
                    "date-time": "2018-02-27 16:33:29 UTC",
                    "time-length": "3d 03:55"
                },
                "system-booted-time": {
                    "date-time": "2018-02-27 16:33:28 UTC",
                    "time-length": "3d 03:55"
                },
                "time-source": "NTP CLOCK",
                "uptime-information": {
                    "active-user-count": "1",
                    "date-time": "8:28PM",
                    "load-average-1": "0.00",
                    "load-average-15": "0.00",
                    "load-average-5": "0.00",
                    "up-time": "3 days, 3:55"
                }
            }
        }
    }
}
ok: [bilbo] => {
    "uptime.parsed_output": {
        "multi-routing-engine-results": {
            "multi-routing-engine-item": {
                "re-name": "fpc0",
                "system-uptime-information": {
                    "current-time": {
                        "date-time": "2018-03-02 20:28:33 UTC"
                    },
                    "last-configured-time": {
                        "date-time": "2018-02-28 15:58:56 UTC",
                        "time-length": "2d 04:29",
                        "user": "sean"
                    },
                    "protocols-started-time": {
                        "date-time": "2018-02-25 19:00:54 UTC",
                        "time-length": "5d 01:27"
                    },
                    "system-booted-time": {
                        "date-time": "2018-02-25 18:57:45 UTC",
                        "time-length": "5d 01:30"
                    },
                    "uptime-information": {
                        "active-user-count": "0",
                        "date-time": "8:28PM",
                        "load-average-1": "0.12",
                        "load-average-15": "0.01",
                        "load-average-5": "0.03",
```

```
                    "up-time": "5 days, 1:31"
                }
            }
        }
    }
}
...
```

Notice that the output of both devices is a set of nested JSON dictionaries, and among the inner dictionaries we see keys `"system-booted-time"` and `"last-configured-time"` with the respective date and time data. This should let us get exactly what we want.

However, the outer dictionary levels are different – because *bilbo* is an EX it could (theoretically) be part of a virtual chassis, so the original XML data and the parsed JSON data contains keys related to multiple routing engine systems; we would see the same keys if we ran "show system uptime invoke-on all-routing-engines" on an MX-class device. On the other hand, *aragorn*, a virtual SRX, does not have keys related to multiple routing engines; it has just a `"system-uptime-information"` key.

How are we going to handle the different outer dictionary keys? Let's start with just the data from *aragorn*, then work our way to handling the different data for *bilbo*.

Modify the last line of the playbook as follows:

```
...
31|    - name: display uptimes
32|      debug:
33|        var: uptime.parsed_output.system-uptime-information
```

Run the playbook. You should get the following error (boldface added):

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] *******************************
ok: [aragorn]

TASK [display uptimes] **********************************************
ok: [aragorn] => {
    "uptime.parsed_output.system-uptime-information": "VARIABLE IS NOT DEFINED!"
}

PLAY RECAP *********************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
```

There is a limitation to the "Ansible shortcut" for referencing dictionary entries – the key needs to be a valid Ansible variable name. Recall from Chapter 4 that variable names cannot include hyphens (-). Because Junos uses hyphens in many of its hierarchy names and other identifiers, we need to use the Python style of dic-

tionary reference for processing the Junos results.

Change the last line of the playbook as shown:

```
...
31|   – name: display uptimes
32|     debug:
33|       uptime.parsed_output['system–uptime–information']['system–booted–time']
```

Run the playbook again.

```
mbp15:aja2 sean$ ansible–playbook uptime.yaml ––limit=aragorn
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************************
ok: [aragorn]

TASK [display uptimes] *********************************************************
ok: [aragorn] => {
    "uptime.parsed_output['system–uptime–information']['system–booted–time']": {
        "date–time": "2018–02–28 19:14:42 UTC",
        "time–length": "3d 20:21"
    }
}

PLAY RECAP ********************************************************************
aragorn                     : ok=2    changed=0    unreachable=0    failed=0
```

Much better! To display the "last–configured–time" key, we could simply add another debug task (this will be left as an exercise for the reader).

But what happens if we run this for *bilbo*?

```
mbp15:aja2 sean$ ansible–playbook uptime.yaml ––limit=bilbo
...
TASK [display uptimes] *********************************************************
ok: [bilbo] => {
    "uptime.parsed_output['system–uptime–information']['system–booted–
time']": "VARIABLE IS NOT DEFINED!"
}
...
```

We need to update the playbook to reference the correct outer keys in bilbo's parsed_output dictionary. Modify the playbook as follows (line 37 may wrap below but should be a single line in your playbook):

```
...
31|   – name: display uptimes (single–RE)
32|     debug:
33|       var: uptime.parsed_output['system–uptime–information']['system–booted–time']
34|
35|   – name: display uptimes (multi–RE)
36|     debug:
```

```
37|        var: uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item']
['system-uptime-information']['system-booted-time']
```

Now run the playbook on both devices:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] *******************************
ok: [aragorn]
ok: [bilbo]

TASK [display uptimes (single-RE)] **********************************
ok: [aragorn] => {
    "uptime.parsed_output['system-uptime-information']['system-booted-time']": {
        "date-time": "2018-02-28 19:14:42 UTC",
        "time-length": "3d 20:38"
    }
}
ok: [bilbo] => {
    "uptime.parsed_output['system-uptime-information']['system-booted-
time']": "VARIABLE IS NOT DEFINED!"
}

TASK [display uptimes (multi-RE)] ***********************************
ok: [aragorn] => {
    "uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item']['system-
uptime-information']['system-booted-time']": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item']['system-
uptime-information']['system-booted-time']": {
        "date-time": "2018-02-25 18:57:45 UTC",
        "time-length": "6d 20:55"
    }
}

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

Each debug task succeeds for one device but fails (VARIABLE IS NOT DEFINED!) for the other. We get the data we want, but the variable errors are a bit ugly. Can we clean this up, perhaps by executing each debug task only for the device where it should should display output?

NOTE   If you execute this playbook against an EX virtual chassis (VC), rather than a single EX, the JSON data gets more complicated because uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item'] becomes a *list* of dictionaries, not a single dictionary. As a result, the playbook above still

generates a "variable is not defined" message for an EX VC. You can display the uptime for the first VC member by inserting `[0]` to reference the first element of the list, like this:

```
- name: display uptimes (multi-RE)
  debug:
    var: uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item'][0]
['system-uptime-information']['system-booted-time']
```

Introducing "When" – Uptime Version 2.2

Ansible offers several *conditionals*, statements that allow the playbook to make a yes-or-no decision, and alter what the playbook does, based on a *condition* such as the value of a variable. The `when` conditional is the most fundamental conditional; it allows Ansible to determine whether or not it should run the task with which the `when` conditional is associated.

We will add `when` conditionals to the `debug` tasks, so the debug tasks execute only when the variable we wish to display is defined. This will allow the playbook to avoid the VARIABLE IS NOT DEFINED errors. The basic format for our test is:

```
when: variable_name is defined
```

In our playbook, it looks like this (note that `when` is indented at the level of the task):

```
- name: display uptimes (single-RE)
  debug:
    var: uptime.parsed_output['system-uptime-information']['system-booted-time']
  when: uptime.parsed_output['system-uptime-information'] is defined
```

More generally, the *condition* for `when` (or any other conditional) is an expression that must evaluate to the Boolean values *true* or *false*. We will see more examples of `when` later in the book.

For readers with a programming background, think of `when` as Ansible's equivalent to the `if` or `if-then` statement in most programming languages. If this were a Python program, an equivalent expression might look something like this:

```
if uptime['parsed_output']['system-uptime-information'] is defined:
    print uptime['parsed_output']['system-uptime-information'] ['system-booted-time']
```

Modify the playbook as follows (add the boldfaced lines):

```
...
31|    - name: display uptimes (single-RE)
32|      debug:
33|        var: uptime.parsed_output['system-uptime-information']['system-booted-time']
34|      when: uptime.parsed_output['system-uptime-information'] is defined
35|
36|    - name: display uptimes (multi-RE)
37|      debug:
38|        var: uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item']
```

```
['system-uptime-information']['system-booted-time']
39|      when: uptime.parsed_output['multi-routing-engine-results'] is defined
```

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>

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

TASK [get uptime using galaxy module] ******************************
ok: [aragorn]
ok: [bilbo]

TASK [display uptimes (single-RE)] *********************************
skipping: [bilbo]
ok: [aragorn] => {
    "uptime.parsed_output['system-uptime-information']['system-booted-time']": {
        "date-time": "2018-02-28 19:14:42 UTC",
        "time-length": "3d 21:34"
    }
}

TASK [display uptimes (multi-RE)] **********************************
skipping: [aragorn]
ok: [bilbo] => {
    "uptime.parsed_output['multi-routing-engine-results']['multi-routing-engine-item']['system-
uptime-information']['system-booted-time']": {
        "date-time": "2018-02-25 18:57:45 UTC",
        "time-length": "6d 21:51"
    }
}

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

Observe that the task "display uptimes (single-RE)" skips *bilbo*, while the task "display uptimes (multi-RE)" skips *aragorn*. Excellent!

juniper_junos_rpc Options – Show Interfaces

Let's explore some of the arguments available for the `juniper_junos_rpc` module. In the next few pages we look at three features: providing arguments to the RPC being called by `juniper_junos_rpc`, changing the format of the data returned by `juniper_junos_rpc`, and saving the results to a file.

Let's start by creating a new playbook that displays interface information for a Junos device. The RPC for this, as discussed earlier in this chapter, is `get-interface-information`. Create playbook `interfaces.yaml` as shown (line numbers added for discussion, do not include them in your playbook). You can copy and modify the `uptime.yaml` playbook if you like, as most of the new playbook is similar.

```
 1|---
 2|- name: Get device interfaces
 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 interface information
21|      juniper_junos_rpc:
22|        rpcs:
23|          - get-interface-information
24|        provider:
25|          host: "{{ ansible_host }}"
26|          user: "{{ username }}"
27|          passwd: "{{ password }}"
28|      register: interfaces
29|
30|    - name: display interfaces
31|      debug:
32|        var: interfaces
```

Lines 1–19 are identical to the `uptime.yaml` playbook.

Lines 20–28 call the RPC and register the results in variable `interfaces`. Note the RPC `get-interface-information` on line 23 and the `register` argument on line 28. Note that we removed the `port` argument from the `provider` dictionary, which means the connection will default to NETCONF port 830. Be sure you have enabled NETCONF on (some of) your test devices as discussed earlier in this chapter.

Lines 30-32 display the contents of variable `interfaces`, the interface data.

As currently written, this playbook will display data for all interfaces, as if you used the command "show interfaces" at the Junos CLI. Because the output is so long, we will show only a portion of the output here. Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook interfaces.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>

PLAY [Get device interfaces] *********************************************

TASK [get uptime using galaxy module] ***********************************
ok: [aragorn]
```

```
TASK [display interfaces] *******************************************************
ok: [aragorn] => {
    "interfaces": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "xml",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "parsed_output": {
...
        },
        "rpc": "get-interface-information",
        "stdout_lines": [
            "<interface-information style=\"normal\">",
            "  <physical-interface>",
            "    <name>ge-0/0/0</name>",
            "    <admin-status format=\"Enabled\">up</admin-status>",
            "    <oper-status>down</oper-status>",
            "    <local-index>135</local-index>",
            "    <snmp-index>510</snmp-index>",
            "    <link-level-type>Ethernet</link-level-type>",
            "    <mtu>1514</mtu>",
            "    <sonet-mode>LAN-PHY</sonet-mode>",
            "    <source-filtering>disabled</source-filtering>",
            "    <link-mode>Half-duplex</link-mode>",
            "    <speed>1000mbps</speed>",
...
            "    <logical-interface>",
            "      <name>ge-0/0/0.0</name>",
            "      <local-index>72</local-index>",
            "      <snmp-index>520</snmp-index>",
...
            "      <address-family>",
            "        <address-family-name>inet</address-family-name>",
            "        <mtu>1500</mtu>",
            "        <address-family-flags>",
            "          <ifff-sendbcast-pkt-to-re/>",
            "        </address-family-flags>",
            "        <interface-address>",
            "          <ifa-flags>",
            "            <ifaf-down/>",
            "            <ifaf-current-preferred/>",
            "            <ifaf-current-primary/>",
            "          </ifa-flags>",
            "          <ifa-destination>198.51.100.0/26</ifa-destination>",
            "          <ifa-local>198.51.100.1</ifa-local>",
            "          <ifa-broadcast>198.51.100.63</ifa-broadcast>",
            "        </interface-address>",
            "      </address-family>",
            "    </logical-interface>",
            "  </physical-interface>",
            "  <physical-interface>",
            "    <name>gr-0/0/0</name>",
...
            "  </physical-interface>",
            "</interface-information>"
        ]
```

```
    }
}

PLAY RECAP ***********************************************************************
aragorn                     : ok=2    changed=0    unreachable=0    failed=0
```

RPC Arguments – Interfaces Version 1.1

What if we want to display only specific interfaces, or obtain terse (or detailed) information about the interfaces? To do to this at the Junos CLI we add arguments to the "show interfaces" command. We can do the same thing with the RPC call in our playbook using the `kwargs` argument to the `juniper_junos_rpc` module. Kwargs (short for "key word arguments") accepts a dictionary of one or more key:value pairs, where the key corresponds with an RPC argument.

Recall from earlier in this chapter:

```
sean@aragorn> show interfaces terse lo0 | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/15.1X49/junos">
    <rpc>
        <get-interface-information>
                <terse/>
                <interface-name>lo0</interface-name>
        </get-interface-information>
    </rpc>
...
</rpc-reply>
```

Translating this to arguments to the `juniper_junos_rpc` module would look like this:

```
juniper_junos_rpc:
  rpcs:
    - get-interface-information
  kwargs:
    terse: True
    interface_name: lo0
```

The XML RPC element `<interface-name>lo0</interface-name>` becomes `kwargs` argument `interface_name: lo0`. Notice that the hyphens (-) in the RPC tag are replaced by underscores (_) in the `kwargs` key.

The XML RPC element `<terse/>` is an empty tag, meaning there is no explicit "value" for the `kwargs` key:value pair. The tag name becomes the key with the Boolean value *true* added to complete the key:value pair `terse: True`.

Let's update our `interfaces.yaml` playbook to give us detailed information for only interface ge-0/0/0:

```
...
19|  tasks:
20|    - name: get interface information
21|      juniper_junos_rpc:
22|        rpcs:
23|          - get-interface-information
24|        kwargs:
25|          detail: True
26|          interface_name: ge-0/0/0
27|        provider:
```

```
28|          host: "{{ ansible_host }}"
29|          user: "{{ username }}"
30|          passwd: "{{ password }}"
31|      register: interfaces
...
```

TIP   You can use wildcards in the interface name; for example, you can specify
`interface_name: ge-0/*` to get *all* Gigabit Ethernet interfaces on FPC0.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook interfaces.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>


PLAY [Get device interfaces] *********************************************

TASK [get interface information] *****************************************
ok: [aragorn]

TASK [display interfaces] ************************************************
ok: [aragorn] => {
    "interfaces": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "xml",
        "kwargs": {
            "detail": true,
            "interface_name": "ge-0/0/0"
        },
        "msg": "The RPC executed successfully.",
...
        "stdout_lines": [
            "<interface-information style=\"normal\">",
            "  <physical-interface>",
            "    <name>ge-0/0/0</name>",
            "    <admin-status format=\"Enabled\">up</admin-status>",
...
            "    <traffic-statistics style=\"verbose\">",
            "      <input-bytes>6480104</input-bytes>",
            "      <input-bps>0</input-bps>",
            "      <output-bytes>1329149</output-bytes>",
            "      <output-bps>0</output-bps>",
            "      <input-packets>79611</input-packets>",
            "      <input-pps>0</input-pps>",
            "      <output-packets>5370</output-packets>",
            "      <output-pps>0</output-pps>",
            "    </traffic-statistics>",
            "    <queue-counters style=\"brief\">",
            "      <interface-cos-short-summary>",
            "        <intf-cos-queue-type>Egress queues</intf-cos-queue-type>",
            "        <intf-cos-num-queues-supported>8</intf-cos-num-queues-supported>",
            "        <intf-cos-num-queues-in-use>4</intf-cos-num-queues-in-use>",
            "      </interface-cos-short-summary>",
            "      <queue>",
            "        <queue-number>0</queue-number>",
```

```
           "        <forwarding-class-name>best-effort</forwarding-class-name>",
           "        <queue-counters-queued-packets>732</queue-counters-queued-packets>",
           "        <queue-counters-trans-packets>732</queue-counters-trans-packets>",
           "        <queue-counters-total-drop-packets>0</queue-counters-total-drop-packets>",
           "      </queue>",
...
           "    </queue-counters>",
           "    <queue-num-forwarding-class-name-map>",
           "      <queue-number>0</queue-number>",
           "      <forwarding-class-name>best-effort</forwarding-class-name>",
           "    </queue-num-forwarding-class-name-map>",
...
           "  </physical-interface>",
           "</interface-information>"
       ]
    }
}

PLAY RECAP **********************************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
```

Notice that the results include traffic statistics, class-of-service counters, and other "detail" level information for the interface.

Output Format – Interfaces Version 1.2

As we have seen, the juniper_junos_rpc module returns results as XML data by default. However, we can request that juniper_junos_rpc return results as CLI-style text, as the juniper_junos_command module does by default, or as JSON data. This is accomplished with the formats argument, which accepts the values json, text, or xml.

TIP   The formats argument also works with the juniper_junos_command module.

Modify the interfaces.yaml playbook as follows (new or changed lines shown in bold):

```
 1|---
 2|- name: Get device interfaces
 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 interface information
21|      juniper_junos_rpc:
22|        rpcs:
23|          - get-interface-information
24|        formats: text
25|        kwargs:
26|          terse: True
27|          interface_name: ge-0/0/0
28|        provider:
29|          host: "{{ ansible_host }}"
30|          user: "{{ username }}"
31|          passwd: "{{ password }}"
32|      register: interfaces
33|
34|    - name: display interfaces
35|      debug:
36|        var: interfaces
```

Line 24 is the `formats` argument, in this case requesting that the results be in `text` format.

Line 26 changed to request `terse` interface information instead of `detail` information, just to keep the output short.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook interfaces.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>

PLAY [Get device interfaces] ********************************************

TASK [get interface information] ***************************************
ok: [aragorn]

TASK [display interfaces] **********************************************
ok: [aragorn] => {
    "interfaces": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "text",
        "kwargs": {
            "interface_name": "ge-0/0/0",
            "terse": true
        },
        "msg": "The RPC executed successfully.",
        "rpc": "get-interface-information",
        "stdout": "\nInterface              Admin Link Proto    Local                 Remote\
nge-0/0/0               up    down\nge-0/0/0.0              up    down inet    198.51.100.1/26 \n",
        "stdout_lines": [
            "",
            "Interface              Admin Link Proto    Local                 Remote",
            "ge-0/0/0               up    down",
            "ge-0/0/0.0             up    down inet    198.51.100.1/26 "
```

```
        ]
    }
}
```

```
PLAY RECAP **********************************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
```

Observe that the results are text format, similar to what we saw from the `juniper_junos_command` module in Chapter 4.

JSON output requires Junos support. If one or more of your devices is running Junos 14.2 or newer, change line 24 to request JSON results and run the playbook again:

24| **formats: json**

Trying this with older versions of Junos will result in an error. The author's vSRX *aragorn* is running Junos 15.1X49, but his switch *bilbo* is running Junos 12.3R12.4. Observe that *bilbo* fails the "get interface information" task (the significant part of the error message is boldfaced below), while *aragorn* returns JSON data:

```
mbp15:aja2 sean$ ansible–playbook interfaces.yaml
Junos Username: sean
Junos Password: <enter password>


PLAY [Get device interfaces] *********************************************

TASK [get interface information] ***************************************
ok: [aragorn]
fatal: [bilbo]: FAILED! => {"changed": false, "module_stderr": "/usr/local/Cellar/python/2.7.14_2/
Frameworks/Python.framework/Versions/2.7/lib/python2.7/site–packages/jnpr/junos/device.
py:818: RuntimeWarning: Native JSON support is only from 14.2 onwards\n  RuntimeWarning)\
nTraceback (most recent call last):\n  File \"/var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/
ansible_15vMtO/ansible_module_juniper_junos_rpc.py\", line 662, in <module>\n    main()\n  File \"/
var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/ansible_15vMtO/ansible_module_juniper_junos_rpc.
py\", line 648, in main\n    junos_module.exit_json(**results[0])\n  File \"/private/etc/ansible/
roles/Juniper.junos/module_utils/juniper_junos_common.py\", line 758, in exit_
json\n    super(JuniperJunosModule, self).exit_json(**kwargs)\n  File \"/var/folders/y1/nqmc7hf13kz5
rckn40p5jfbh0000gp/T/ansible_15vMtO/ansible_modlib.zip/ansible/module_utils/basic.
py\", line 2304, in exit_json\n  File \"/var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/
ansible_15vMtO/ansible_modlib.zip/ansible/module_utils/basic.py\", line 2297, in _return_
formatted\n  File \"/var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/ansible_15vMtO/ansible_modlib.
zip/ansible/module_utils/basic.py\", line 514, in remove_values\n  File \"/var/folders/y1/nqmc7hf13k
z5rckn40p5jfbh0000gp/T/ansible_15vMtO/ansible_modlib.zip/ansible/module_utils/basic.
py\", line 497, in _remove_values_conditions\nTypeError: Value of unknown type: <type 'lxml.
etree._Element'>, <Element interface–information at 0x10307d488>\n", "module_
stdout": "", "msg": "MODULE FAILURE", "rc": 0}

TASK [display interfaces] ***********************************************
ok: [aragorn] => {
    "interfaces": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "json",
```

```
"kwargs": {
    "interface_name": "ge-0/0/0",
    "terse": true
},
"msg": "The RPC executed successfully.",
"parsed_output": {
    "interface-information": [
        {
            "attributes": {
                "junos:style": "terse",
                "xmlns": "http://xml.juniper.net/junos/15.1X49/junos-interface"
            },
            "physical-interface": [
                {
                    "admin-status": [
                        {
                            "data": "up"
                        }
                    ],
                    "logical-interface": [
                        {
                            "address-family": [
                                {
                                    "address-family-name": [
                                        {
                                            "data": "inet"
                                        }
                                    ],
                                    "interface-address": [
                                        {
                                            "ifa-local": [
                                                {
                                                    "attributes": {
                                                        "junos:emit": "emit"
                                                    },
                                                    "data": "198.51.100.1/26"
                                                }
                                            ]
                                        }
                                    ]
                                }
                            ],
                            "admin-status": [
                                {
                                    "data": "up"
                                }
                            ],
                            "filter-information": [
                                {}
                            ],
                            "name": [
                                {
                                    "data": "ge-0/0/0.0"
                                }
                            ],
                            "oper-status": [
                                {
                                    "data": "up"
```

```
                                                }
                                            ]
                                        }
                                    ],
                                    "name": [
                                        {
                                            "data": "ge-0/0/0"
                                        }
                                    ],
                                    "oper-status": [
                                        {
                                            "data": "up"
                                        }
                                    ]
                                }
                            ]
                        }
                    ]
                },
                "rpc": "get-interface-information",
                "stdout": "{u'interface-
information': [{u'attributes': {u'junos:style': u'terse', u'xmlns': u'http://xml.juniper.net/
junos/15.1X49/junos-interface'}, u'physical-interface': [{u'oper-
status': [{u'data': u'up'}], u'logical-interface': [{u'oper-status': [{u'data': u'up'}], u'address-
family': [{u'address-family-name': [{u'data': u'inet'}], u'interface-address': [{u'ifa-
local': [{u'attributes': {u'junos:emit': u'emit'}, u'data': u'198.51.100.1/26'}]}]}], u'admin-
status': [{u'data': u'up'}], u'name': [{u'data': u'ge-0/0/0.0'}], u'filter-
information': [{}]}], u'admin-status': [{u'data': u'up'}], u'name': [{u'data': u'ge-0/0/0'}]}]}]}",
                "stdout_lines": [
                    "{u'interface-
information': [{u'attributes': {u'junos:style': u'terse', u'xmlns': u'http://xml.juniper.net/
junos/15.1X49/junos-interface'}, u'physical-interface': [{u'oper-
status': [{u'data': u'up'}], u'logical-interface': [{u'oper-status': [{u'data': u'up'}], u'address-
family': [{u'address-family-name': [{u'data': u'inet'}], u'interface-address': [{u'ifa-
local': [{u'attributes': {u'junos:emit': u'emit'}, u'data': u'198.51.100.1/26'}]}]}], u'admin-
status': [{u'data': u'up'}], u'name': [{u'data': u'ge-0/0/0.0'}], u'filter-
information': [{}]}], u'admin-status': [{u'data': u'up'}], u'name': [{u'data': u'ge-0/0/0'}]}]}]}"
                ]
            }
        }
            to retry, use: --limit @/Users/sean/aja2/interfaces.retry

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

Saving Results to a File – Interfaces Version 1.3

The juniper_junos_command and juniper_junos_rpc tasks we have seen so far have all returned their results to the playbook. These results could be used in various ways by subsequent tasks, and we will shortly see examples doing more than printing those results with debug.

What if you want to save the results to a file? Perhaps you want to send the results to someone (email, FTP, etc.) or perhaps the results are large enough that keeping

them in memory across numerous devices seems ill advised (for example, "show interfaces extensive" on dozens of 10-member EX virtual chassis).

The `juniper_junos_command` and `juniper_junos_rpc` modules have arguments for saving results to a file, instead of or in addition to returning the results to the playbook.

Update the `interfaces.yaml` playbook as follows (new/changed lines in boldface):

```
1|---
2|- name: Get device interfaces
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 interface information
21|      juniper_junos_rpc:
22|        rpcs:
23|          - get-interface-information
24|        dest_dir: '.'
25|        return_output: no
26|        formats: xml
27|        provider:
28|          host: "{{ ansible_host }}"
29|          user: "{{ username }}"
30|          passwd: "{{ password }}"
31|      register: interfaces
32|
33|    - name: display interfaces
34|      debug:
35|        var: interfaces
```

Line 24, the `dest_dir` argument, provides the name of the directory where the `juniper_junos_rpc` module should save the files containing the RPC results. When this argument is absent or set to `null` the module does not save results to a file; including this argument with a valid directory path tells the module to save results to a file. The module will automatically generate a filename using the value provided with the `host` argument (or `inventory_hostname` if `host` is not provided) and the RPC to be executed, with an extension from the `formats` argument. In this example, setting `dest_dir` to `'.'` tells the `juniper_junos_rpc` module to save the results file in the current directory, the directory where the playbook is located.

Line 25, the `return_output` argument, accepts a Boolean value (True/False or yes/no). This argument tells the module whether or not it should return results to the playbook. When this argument is absent, the module defaults to True (return results).

NOTE   The `dest_dir` and `return_output` arguments may be used independently from each other. This example uses both to save results to a file and not return them to the playbook, but you can use one or the other to achieve different outcomes.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook interfaces.yaml
Junos Username: sean
Junos Password: <enter password>

PLAY [Get device interfaces] *********************************************

TASK [get interface information] *****************************************
ok: [aragorn]
ok: [bilbo]

TASK [display interfaces] ************************************************
ok: [aragorn] => {
    "interfaces": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "xml",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "rpc": "get-interface-information"
    }
}
ok: [bilbo] => {
    "interfaces": {
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "xml",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "rpc": "get-interface-information"
    }
}

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

Observe that the registered variable `interfaces` does *not* include RPC results.

Check that `*.xml` files (because `formats` was set to `xml`) were created with the results of the RPC:

```
mbp15:aja2 sean$ ls -l *.xml
```

```
192.0.2.10_get-interface-information.xml
bilbo_get-interface-information.xml

mbp15:aja2 sean$ more 192.0.2.10_get-interface-information.xml
<interface-information style="normal">
  <physical-interface>
    <name>ge-0/0/0</name>
    <admin-status format="Enabled">up</admin-status>
    <oper-status>up</oper-status>
    <local-index>135</local-index>
    <snmp-index>510</snmp-index>
    <link-level-type>Ethernet</link-level-type>
    <mtu>1514</mtu>
    <sonet-mode>LAN-PHY</sonet-mode>
    <source-filtering>disabled</source-filtering>
    <link-mode>Full-duplex</link-mode>
    <speed>1000mbps</speed>
...
    <current-physical-address>00:0c:29:49:0c:e8</current-physical-address>
    <hardware-physical-address>00:0c:29:49:0c:e8</hardware-physical-address>
    <interface-flapped seconds="68469">2018-03-05 04:17:49 UTC (19:01:09 ago)</interface-flapped>
...
    <logical-interface>
      <name>ge-0/0/0.0</name>
...
      <address-family>
        <address-family-name>inet</address-family-name>
        <mtu>1500</mtu>
        <address-family-flags>
          <ifff-sendbcast-pkt-to-re/>
        </address-family-flags>
        <interface-address>
          <ifa-flags>
            <ifaf-current-preferred/>
            <ifaf-current-primary/>
          </ifa-flags>
          <ifa-destination>198.51.100.0/26</ifa-destination>
          <ifa-local>198.51.100.1</ifa-local>
          <ifa-broadcast>198.51.100.63</ifa-broadcast>
        </interface-address>
      </address-family>
    </logical-interface>
  </physical-interface>
...
```

Thought exercise for the reader: why was *aragorn's* output file named by IP address while *bilbo's* output file was named by hostname? Hint: check the `inventory` file.

XML and XPath

Our `interfaces.yaml` playbook is gathering interface data as XML for each interface. How can we extract certain fields that we want from that XML data?

Ansible, starting in version 2.4, includes a module called *xml* that can perform

various tasks on XML data. With the xml module we can extract specific elements of interest 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.

Use the following playbook, test-xml.yaml, to experiment with XPath expressions:

```
 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: xpath
10|      xml:
11|        path: bilbo_get-interface-information.xml
12|        xpath: /interface-information/physical-interface/name
13|        content: text
14|      register: interface_info
15|
16|    - name: show xpath results
17|      debug:
18|        var: interface_info
```

Line 11 is the path argument, which identifies an XML data file to read. You can use one of the .xml files created in the last section of this chapter, just substitute the correct filename in the path argument.

Line 12 is the xpath argument, the XPath path expression to search. Substitute the different XPath path expressions discussed below, or any other you would like to try, and run the playbook to observe the results. The path shown here will list the names of all the physical interfaces in the XML data. XPath path expressions are explained momentarily.

Line 13, the content argument, tells the xml module what data from the matching element to return; text means the text contents of the element (the CDATA). The other supported setting is attribute.

Running the playbook looks something like this (edited for length):

```
mbp15:aja2 sean$ ansible-playbook test-xml.yaml

PLAY [Experiment with Ansible's xml module] **********************************

TASK [xpath] ****************************************************************
ok: [localhost]

TASK [show xpath results] ***************************************************
ok: [localhost] => {
    "interface_info": {
        "actions": {
            "namespaces": {},
```

```
            "state": "present",
            "xpath": "/interface-information/physical-interface/name"
        },
        "changed": false,
        "count": 27,
        "failed": false,
        "matches": [
            {
                "name": "ge-0/0/0"
            },
            {
                "name": "ge-0/0/1"
            },
...
            {
                "name": "ge-0/1/1"
            },
            {
                "name": "bme0"
            },
            {
                "name": "dsc"
            },
...
            {
                "name": "vlan"
            },
            {
                "name": "vme"
            }
        ],
        "msg": 27
    }
}

PLAY RECAP *********************************************************************
localhost                   : ok=2    changed=0    unreachable=0    failed=0
```

Notice that the interface_info variable contains a matches list which includes a dictionary for each match. There is also an interface_info.count entry that indicates the number of matches found.

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 interface data, the path to an interface's MAC address is:

```
/interface-information/physical-interface/hardware-physical-address
```

The leading slash indicates we are specifying the path from the root of the XML hierarchy in question.

As you saw above, XPath can return multiple matches. Take a look at the XML output file for one of your test devices and you will observe that each physical interface is described inside of a <physical-interface> element, meaning that element

name exists one for each interface on the device. Many of the elements within the `<physical-interface>` element repeat for all interfaces, and some, like `<logical-interface>`, may even repeat within a given physical interface (a single physical interface may have multiple logical sub-interfaces).

XPath offers a shortcut to specifying a full path: a leading double-slash ('//') searches down the XML hierarchy for matches, regardless of how many levels deep they may be. For example, we could use the XPath path `//hardware-physical-address` instead of the full path above.

In addition, XPath will ignore data that does not match. For example, the path expression `//logical-interface/name` will ignore physical interfaces that do not have logical sub-interfaces, returning only the names of logical interfaces.

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 element for that element to be included in the results.

For example, if we wanted to find information for the `physical-interface` whose name is `ge-0/1/0`, we could use this XPath path with predicate:

```
//physical-interface[name='ge-0/1/0']
```

To return all child elements of the matching `physical-interface` element, add the asterisk wildcard (*) to the end of the path:

```
//physical-interface[name='ge-0/1/0']/*
```

To return a single child element of the matching `physical-interface` element, add the child element's name to the end of the path:

```
//physical-interface[name='ge-0/1/0']/hardware-physical-address
```

There are a number of functions that can be used in predicates when you might not know or have an exact match. For example, if we want the `oper-status` (link-up or link-down) for all gigabit Ethernet interfaces on FPC 0, PIC 1, we can look for physical interface names that *start with* the known portion of the interface name 'ge-0/1' by using the `starts-with()` function:

```
//physical-interface[starts-with(name,'ge-0/1')]/oper-status
```

XPath predicates can include multiple tests joined by the Boolean operators **and** and **or**. For example, to find the name of each GE interface that is link-up:

```
//physical-interface[oper-status='up' and starts-with(name, 'ge-')]/name
```

You can combine two XPath paths and receive results that match either of the paths by joining the paths with the pipe character ('|'). This acts as a Boolean *or* between the two paths. For example, to return both the name and operational status of all physical interfaces:

```
//physical-interface/name | //physical-interface/oper-status
```

> 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.

Querying Interface Data with XPath– Interfaces Version 1.4

Now let's use XPath to query our interface information. Assume we want to know the IPv4 address(es) for the device's interfaces.

Modify the `interfaces.yaml` playbook as follows:

```
 1|---
 2|- name: Get device interfaces
 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 interface information
21|      juniper_junos_rpc:
22|        rpcs:
23|          - get-interface-information
24|        formats: xml
25|        provider:
26|          host: "{{ ansible_host }}"
27|          user: "{{ username }}"
28|          passwd: "{{ password }}"
29|      register: interfaces
30|
31|    - name: query interface information
32|      xml:
33|        xmlstring: "{{ interfaces.stdout }}"
34|        xpath: //logical-interface/address-family[address-family-name='inet']/interface-address/ifa-local
35|        content: text
36|      register: ip_addr
37|
38|    - name: show query results
39|      debug:
40|        var: ip_addr.matches
```

Lines 20-29 gather the interface data and store it in registered variable `interfaces`. (Note that we remove the `dest_dir` and `return_output` arguments so the data is returnd to the playbook not stored in a file.)

Lines 31-36 use the `xml` module to search the interface data using our XPath path expression.

Line 33, the `xmlstring` argument, tells the `xml` module to search a string, in this case the `stdout` string in the `interfaces` variable. (This is a little different from the `test-xml.yaml` playbook, which searched a file.)

Line 34 is our XPath path expression. This finds all `logical-interface` elements that have an `ifa-local` value (an address), filtering them with a predicate to match only `inet` family addresses (IPv4).

Line 36 stores the results in registered variable `ip_addr`.

Lines 38-40 display the `matches` list within the `ip_addr` variable, which are the values that matched the XPath path expression.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook interfaces.yaml
Junos Username: sean
Junos Password: <enter password>

PLAY [Get device interfaces] ********************************************

TASK [get interface information] **************************************
ok: [aragorn]
ok: [bilbo]

TASK [query interface information] ***********************************
ok: [aragorn]
ok: [bilbo]

TASK [show query results] ********************************************
ok: [aragorn] => {
    "ip_addr.matches": [
        {
            "ifa-local": "198.51.100.1"
        },
        {
            "ifa-local": "128.0.0.1"
        },
        {
            "ifa-local": "192.0.2.10"
        },
        {
            "ifa-local": "127.0.0.1"
        },
        {
            "ifa-local": "128.0.0.1"
        },
        {
            "ifa-local": "128.0.0.4"
        },
        {
            "ifa-local": "128.0.1.16"
        }
```

```
    ]
}
ok: [bilbo] => {
    "ip_addr.matches": [
        {
            "ifa-local": "128.0.0.1"
        },
        {
            "ifa-local": "128.0.0.16"
        },
        {
            "ifa-local": "128.0.0.32"
        },
        {
            "ifa-local": "127.0.0.1"
        },
        {
            "ifa-local": "198.51.100.5"
        },
        {
            "ifa-local": "198.51.100.66"
        }
    ]
}

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

Very nice!

Exercise for the reader: modify the XPath expression to include a second path, so the results include the logical interface name.

Querying Uptime Data with XPath – Uptime Version 2.3

This XPath stuff is pretty cool. Can we use XPath expressions to improve the output of our `uptime.yaml` playbook? Recall that the playbook had two debug tasks to display results, but each task would be skipped for some of our devices due to the different data structures from different devices. Can we reduce this to a single debug task? Even better, can we display both "last booted" and "last configured" times?

Modify `uptime.yaml` as follows:

```
 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|      juniper_junos_rpc:
22|        rpcs:
23|          – get–system–uptime–information
24|        provider:
25|          host: "{{ ansible_host }}"
26|          port: 22
27|          user: "{{ username }}"
28|          passwd: "{{ password }}"
29|      register: uptime
30|
31|    – name: query uptime information
32|      xml:
33|        xmlstring: "{{ uptime.stdout }}"
34|        xpath: //system–booted–time/date–time | //last–configured–time/date–time
35|        content: text
36|      register: last_boot
37|
38|    – name: show query results
39|      debug:
40|        var: last_boot.matches
```

The first 30 lines of the playbook are unchanged. However, instead of using `debug` tasks with `when` conditions based on the parsed_output from the `juniper_junos_rpc` module, we use an XPath expression.

Lines 31-36 use the `xml` module to query the uptime data.

Line 34 is the XPath path expression that matches both `//system–booted–time/date–time` and `//last–configured–time/date–time` so we get both values of interest.

Let's run the playbook:

```
mbp15:aja2 sean$ ansible–playbook uptime.yaml
Junos Username: sean
Junos Password: <enter password>


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


TASK [get uptime using galaxy module] ******************************
ok: [aragorn]
ok: [bilbo]


TASK [query uptime information] ************************************
ok: [aragorn]
ok: [bilbo]


TASK [show query results] *****************************************
```

```
ok: [aragorn] => {
    "last_boot.matches": [
        {
            "date-time": "2018-03-02 00:59:16 UTC"
        },
        {
            "date-time": "2018-02-27 19:38:47 UTC"
        }
    ]
}
ok: [bilbo] => {
    "last_boot.matches": [
        {
            "date-time": "2018-02-25 18:57:45 UTC"
        },
        {
            "date-time": "2018-02-28 15:58:56 UTC"
        }
    ]
}

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

Not bad, but which date-time value is which? Can we identify them somehow?

In the author's experience, matches to a multiple-path XPath expression are in the order of the XPath paths, so the first date-time should be the system-booted-time and the second should be the last-configured-time. If we change the debug task to display a message, not just a variable, we can identify the two dates.

Change line 40 of the playbook as follows (line may wrap in this book but it should be a single line in your playbook):

```
38|    – name: show query results
39|      debug:
40|        msg: "Last booted {{ last_boot.matches[0]['date-time'] }}; last configured {{ last_boot.
matches[1]['date-time'] }}"
```

Using the debug module's msg argument instead of the var argument lets us specify a complete message to display. Be sure the quotes around the entire message are different from the quotes around the dictionary keys (the ['date-time']) – here the author used double-quotes around the message and single-quotes for the keys.

Run the playbook again and observe the output:

```
...
TASK [show query results] *******************************************************
ok: [aragorn] => {
    "msg": "Last booted 2018-03-02 00:59:16 UTC; last configured 2018-02-27 19:38:47 UTC"
}
ok: [bilbo] => {
```

```
    "msg": "Last booted 2018-02-25 18:57:45 UTC; last configured 2018-02-28 15:58:56 UTC"
}
...
```

Very nice!

Exercise for the reader: The author's experience regarding the order of the XPath results may not be guaranteed. Modify the playbook to use two `xml` tasks, each using a single XPath path and saving their results into different variables. This will ensure the playbook does not mix up the `system-booted-time` and the `last-configured-time`. Update the `debug` message to read both of the variables.

References

Juniper's documentation for their Galaxy modules:
http://junos-ansible-modules.readthedocs.io/en/stable/index.html

https://www.juniper.net/documentation/en_US/release-independent/junos-ansible/information-products/pathway-pages/index.html

NETCONF background:
https://en.wikipedia.org/wiki/NETCONF

Ansible Conditionals:
http://docs.ansible.com/ansible/latest/playbooks_conditionals.html

Ansible XML module:
http://docs.ansible.com/ansible/latest/xml_module.html

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 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. Basic authentication uses a username and password; the relevant configuration on a Junos device would look something like this:

```
sean@aragorn> show configuration system login
...
user sean {
    uid 2000;
    class super-user;
    authentication {
        encrypted-password "$5$kqsva9Mc$T...j3939giNv/"; ## SECRET-DATA
    }
}
...
```

SSH offers an alternative authentication method based on asymmetric cryptography, 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 respective keys to authenticate the connection. No device 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 passphrase 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 computer 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 client 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 configuration for multiple key pairs, focus 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 multiple 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.

CAUTION   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 the relevant files or settings.

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 questions, 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 "Enter 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:HI0QSmeVsa2fsDjw8w/Y1BhsDaWI9azBRHQB2iYiCrU sean@mbp15.local
The key's randomart image is:
+---[RSA 2048]----+
|  . .o@+*=.      |
| . o @ B.O       |
|o E = * @ +      |
|o. . o = *       |
|.   . . S .      |
|     o = + .     |
|      * + o      |
|       + .       |
|        ...      |
+----[SHA256]-----+
```

The single filename prompt is a little misleading because `ssh-keygen` actually generates two files, the public and private key files. The filename 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. Note the lines in the output starting "Your identification has been saved…" and "Your public key has been saved…." You should be able to see the files in your file system:

```
mbp15:~ sean$ ls -1 ~/.ssh/id*
/Users/sean/.ssh/id_rsa
/Users/sean/.ssh/id_rsa.pub
```

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 terminal or open it in your text editor, whichever you prefer:

```
mbp15:~ sean$ cat ~/.ssh/id_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCxgT8ga1uYbS3bxXPPv7aEiTvSwXnK/7xu3NB0+t1njMBuUcgwn7zwtnayQyLS
+ef3rNP7WZXwFYxUeFbVwdkLUn9/xvDM5Qi2m/6WRP/yrTRtEvNP4lUsZRH+IXQc59JOKfYqGkvbgfshnmtHJHYVOn/1E/
w0cNDYg4oH6KbcqYb+isbKhdiqpDBvLsF9h0GwhaiLk2BpVutw2BZoeKN9vrF+OmcaB0WVzGvwblSHDpXdLfMJuHAyEhZImNSv4b
XNAYFGht9zpdTwudP5qfwJo5304Sn62UaOzVN2zGogXKzxgxgAjeJ87io0Graiwo5q9kZYksjXvPzOaX3gt8Uv sean@mbp15.
local
```

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@aragorn> configure
Entering configuration mode

sean@aragorn# edit system login user sean

[edit system login user sean]
sean@aragorn#
```

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@aragorn# set authentication ssh-rsa "ssh-rsa AAAAB3NzaC1y ... vPzOaX3gt8Uv sean@mbp15.local"

[edit system login user sean]
sean@aragorn#
```

Commit that change. Now, from your computer's command line or your SSH client, SSH to the device:

```
mbp15:~ sean$ ssh aragorn
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@aragorn>
```

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 authentication 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 authentication 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 headache 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` commands, which work together to cache SSH key passphrases. The `ssh-agent` command 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 authentication 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 access 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 aragorn`) to a device that uses key-based authentication, macOS will display a dialog 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.
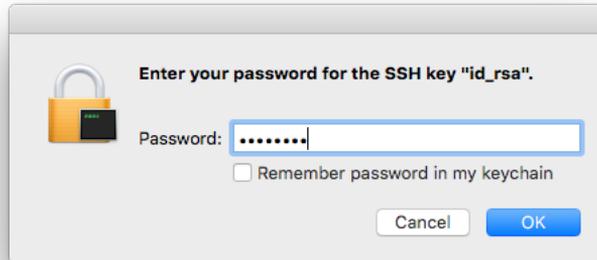
With macOS Sierra (10.12) and High Sierra (10.13), passphrase caching is not enabled by default. You can use the `ssh-add` command discussed above for Linux systems to cache passphrases. Curiously, macOS does not seem to require that you first run `ssh-agent`:

```
mbp15:~ sean$ ssh aragorn
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@aragorn> exit

Connection to aragorn 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 aragorn
--- JUNOS 15.1X49-D90.7 built 2017-04-29 06:51:16 UTC
sean@aragorn> exit

Connection to aragorn closed.
```

Multiple Key Pairs

When establishing a connection to a server that indicates it accepts key-based authentication, the `ssh` client program will look in file ~/.ssh/id_rsa by default for a private key. 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 connection because *gandalf* is outside your organization (perhaps it is owned by a customer). 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 -m PEM -t rsa -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:qUPb3n1K1aqynoS7BEWGopFQOk7Ped8RypnAp9nwTMM Seans key for gandalf
The key's randomart image is:
+---[RSA 2048]----+
|.o..   .o        |
| .o o oo         |
|o. o = E..       |
|o.+ . %.=..    . |
| . + +oOS.    ..|
|    ...=...  .. |
|      +.+..  .  |
|       + +.+ .. |
|        +o=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 passphrase 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 different servers or devices. (There are a number of other arguments, including selecting 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 -1 ~/.ssh/g*
/Users/sean/.ssh/gandalf
/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 AAAAB3NzaC1yc2EAAAADAQABAAAABAQDHd05KBPYi/Q/gPflEIYXum6v6lksE49s4VBV6b+fduysYdCDQkpz8r6/3/KkJd-
0JyLkNvaui4GjiY61Mgtp7nCSCO6g7jfaPMABdtNsGmYBYyJj4TUlxKLaLi9jY/BmXjDSPL7gTLpNS0b2adTDy9DdpLMcAEPGS/+7w
lJLHD8j7RscYyBvb1Gt2bEqNjB839JZYV9uj2OubGZoKfgPBeAHCbfIBqA67V+b0tfQtEU7aHmOIpcKeKZoJnk/HSKTW1YYm4k-
m7uLFJhmIY9HQzRl0E3T6IKHu3Bkia5YOrEWuBGyHjT0HtZyNEspEt8HEMJLCdzuiMDo7v0fk/TEddV Seans key for gandalf
```

Give the public key file (gandalf.pub) to the administrator of *gandalf* and ask them to install the public key in your user account on the server.

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 existing file:

```
Host gandalf
  User sean
  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. The IdentityFile 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 *gandalf* using the alternate key pair:

```
mbp15:~ sean$ ssh gandalf
Enter passphrase for key '/Users/sean/.ssh/gandalf': <enter passphrase>
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-83-generic x86_64)

Last login: Tue Mar  6 13:52:11 2018 from 192.0.2.1

sean@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: <enter passphrase>
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 protocols 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 automation tasks, consider making the corresponding account on the Junos devices a read-only account. This mitigates the damage that could be caused should the private 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 with administrative privileges is a major security concern.

Playbook Using Key-based Authentication – Uptime Version 3

Let's modify the `uptime.yaml` playbook from Chapter 5 to use SSH key-based authentication. This mostly means deleting lines that are no longer needed. Juniper's Galaxy 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 described 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`, `passwd` and `port` arguments from the `juniper_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 galaxy module
```

```
12|      juniper_junos_rpc:
13|        rpcs:
14|          – get-system-uptime-information
15|        provider:
16|          host: "{{ ansible_host }}"
17|      register: uptime
18|
19|    – name: query uptime information
20|      xml:
21|        xmlstring: "{{ uptime.stdout }}"
22|        xpath: //system-booted-time/date-time | //last-configured-time/date-time
23|        content: text
24|      register: last_boot
25|
26|    – name: show query results
27|      debug:
28|        msg: "Last booted {{ last_boot.matches[0]['date-time'] }}; last configured {{ last_boot.
matches[1]['date-time'] }}"
```

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:

```
...
10|  tasks:
11|    – name: get uptime using galaxy module
12|      juniper_junos_rpc:
13|        rpcs:
14|          – get-system-uptime-information
15|        provider:
16|          host: "{{ ansible_host }}"
17|          user: sean
18|      register: uptime
...
```

If you have not already done so, use `ssh-add` (and `ssh-agent` if needed) to cache your private key passphrase:

```
mbp15:aja2 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:aja2 sean$ ansible-playbook uptime.yaml --limit=aragorn

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

TASK [get uptime using galaxy module] ******************************************
ok: [aragorn]

TASK [query uptime information] ************************************************
ok: [aragorn]

TASK [show query results] *****************************************************
```

```
ok: [aragorn] => {
    "msg": "Last booted 2018-03-02 00:59:16 UTC; last configured 2018-03-06 21:17:26 UTC"
}

PLAY RECAP *********************************************************************
aragorn                    : ok=3    changed=0    unreachable=0    failed=0
```

Observe that you did not need to enter the username and password for the playbook, but the playbook was still able to connect to the network device. Nice!

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 installing a new configuration on a new device or changing the configuration of 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 replaces 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 services ssh connection-limit 5
set system services ssh rate-limit 5
delete system services telnet
```

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:

```
sean@aragorn> show configuration system services ssh | display set
set system services ssh connection-limit 5
set system services ssh rate-limit 5
```

The "text" or "config" format looks like the normal Junos configuration, or a portion thereof, complete with braces, semicolons, and indented lines:

```
system {
    services {
        ssh {
            connection-limit 5;
            rate-limit 5;
        }
    }
}
```

NOTE   Junos also supports configuration files in XML and JSON formats. Because these formats are less familiar to many Junos users than the "set" and "text" formats, we will not do any examples using configuration files in these formats.

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

One of the author's test devices has the following DNS servers and host name:

```
sean@aragorn> show configuration system name-server
8.8.8.8;
8.8.4.4;
198.51.100.10;
198.51.100.11;

sean@aragorn> show configuration system host-name
host-name aragorn;
```

Create two configuration files in a convenient directory on your computer, such as in your ~/aja2 directory. File dns1.set should contain the following:

```
set system name-server 198.51.100.29
set system name-server 198.51.100.28
delete system name-server 198.51.100.10
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 {
        198.51.100.25;
        198.51.100.26;
    }
}
```

Note that both files contain the top-level system hierarchy. Your configuration files should always contain the complete structure. This is not strictly needed for manual configuration but is necessary for automated configuration.

NOTE   Many of the DNS server IP addresses used in this section and the next are fictitious; they are used simply to illustrate the concepts of loading configuration files. Do not worry, we will clean up these fictitious name servers later.

Copy these files into your home directory on your test Junos device:

```
mbp15:~ sean$ scp dns* aragorn:.
dns1.set                                           100%  127   133.5KB/s   00:00
dns2.conf                                          100%   94   120.8KB/s   00:00
```

Let's load the set file first. This is accomplished in configuration mode using the `load set` command:

```
sean@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# load set dns1.set
load complete

[edit]
sean@aragorn# show | compare
[edit system]
-  host-name aragorn;
+  host-name vsrx-dns1;
[edit system name-server]
    198.51.100.11 { ... }
+   198.51.100.28;
+   198.51.100.29;
-   198.51.100.10;

[edit]
sean@aragorn# show system host-name
host-name vsrx-dns1;

[edit]
sean@aragorn# show system name-server
8.8.8.8;
8.8.4.4;
198.51.100.11;
198.51.100.28;
198.51.100.29;
```

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@aragorn# load merge dns2.conf
load complete

[edit]
sean@aragorn# show | compare
[edit system]
-  host-name aragorn;
```

```
+  host-name vsrx-dns2;
[edit system name-server]
     198.51.100.11 { ... }
+   198.51.100.28;
+   198.51.100.29;
+   198.51.100.25;
+   198.51.100.26;
-   198.51.100.10;

[edit]
sean@aragorn# show system host-name
host-name vsrx-dns2;

[edit]
sean@aragorn# show system name-server
8.8.8.8;
8.8.4.4;
198.51.100.11;
198.51.100.28;
198.51.100.29;
198.51.100.25;
198.51.100.26;
```

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 updates 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 value, 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 adjustments to the process above. First, in the text configuration file, you need to add `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 honor any `delete:` or `replace:` tags in the text file.

Before we do examples with the `delete:` and `replace:` tags, roll back the uncommitted changes we made above:

```
[edit]
sean@aragorn# rollback
load complete

[edit]
sean@aragorn# show system host-name
host-name aragorn;

[edit]
sean@aragorn# show system name-server
8.8.8.8;
```

```
8.8.4.4;
198.51.100.10;
198.51.100.11;
```

Now let's try the `delete:` tag. Create configuration file `dns3.conf` as follows:

```
system {
    host-name vsrx-dns3;
    name-server {
        198.51.100.10;
        delete: 198.51.100.11;
    }
}
```

Copy the configuration file `dns3.conf` to your test device and "load replace" it:

```
[edit]
sean@aragorn# load replace dns3.conf
load complete

[edit]
sean@aragorn# show | compare
[edit system]
-   host-name aragorn;
+   host-name vsrx-dns3;
[edit system name-server]
-    198.51.100.11;

[edit]
sean@aragorn# show system host-name
host-name vsrx-dns3;

[edit]
sean@aragorn# show system name-server
8.8.8.8;
8.8.4.4;
198.51.100.10;
```

Observe that the `host-name` changed and that server 198.51.100.11 has been removed from the `name-server` list. Also note that re-applying name server 198.51.100.10 had no effect; Junos is very good about ignoring "changes" that do not change anything. Finally, note that the other name servers in the configuration were unaffected.

Roll back that uncommitted change.

Now let's do an example with the `replace:` tag. Assume you wish to replace the entire `name-server` list, regardless of what addresses are currently in the list, with new servers. Create a new file `dns4.conf` and enter the following text:

```
system {
    replace:
    name-server {
        198.51.100.30;
        8.8.8.8;
    }
}
```

Copy the file to your test device and "load replace" it into the configuration:

```
[edit]
sean@aragorn# load replace dns4.conf
load complete

[edit]
sean@aragorn# show | compare
[edit system name-server]
+   198.51.100.30;
    8.8.8.8 { ... }
[edit system name-server]
-   8.8.4.4;
-   198.51.100.10;
-   198.51.100.11;

[edit]
sean@aragorn# show system name-server
198.51.100.30;
8.8.8.8;
```

Observe that the entire `name-server` list was replaced with the new list specified in the file (not withstanding the entry 8.8.8.8 that was in the original and replacement).

Roll back the uncommitted change and exit configuration mode.

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:

```
system {
    replace: authentication-order [ password radius ];
    host-name aragorn;
    replace:
    ntp {
        boot-server 9.8.7.6;
        server 9.8.7.6;
    }
}
```

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 can be useful when you have a complete configuration file for the new device, perhaps for NOOB (new-out-of-box) setup. We will not perform an example here.

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 here. 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. Use Ctrl+D to complete the load:

```
[edit]
sean@aragorn# load set terminal
[Type ^D at a new line to end input]
set system name-server 198.51.100.29
set system name-server 198.51.100.28
load complete
```

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 configuration hierarchy being changed. The `relative` option allows you to load changes 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@aragorn# edit system name-server

[edit system name-server]
sean@aragorn# load set terminal relative
[Type ^D at a new line to end input]
delete 198.51.100.29
delete 198.51.100.28
load complete
```

Now, how do we load configuration files using Ansible?

Installing Text Configuration Files with Ansible

Juniper's Galaxy module `juniper_junos_config` lets us configure a Junos device. There are several ways to use this module: it can accept a configuration file, a list of configuration statements, or a template and a dictionary of values (we discuss templates later in this chapter, though we use them in a somewhat different way). Let's start with a configuration file, basically automating the process we discussed in the previous section of this chapter.

Given a configuration file, in either set or text format, `juniper_junos_config` can load the configuration onto the device and commit the change. By default, the `juniper_junos_config` module checks the file's extension to determine the format: set files should use the extension `.set` and text files should use `.conf`.

The author prefers text format configuration files for automation, so most examples in this book 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 the author believes the hierarchical layout 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 `~/aja2/config` directory, then create in the `config` directory a file `nameserver.conf` with the following contents:

```
system {
    name-server {
        8.8.4.4;
        8.8.8.8;
        198.51.100.100;
    }
}
```

In your ~/aja2 playbook directory, create the playbook install-config.yaml as shown below (line numbers added for discussion, do not enter them into your playbook):

```
 1|---
 2|- name: Install Configuration File
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  tasks:
11|    - name: install configuration file onto device
12|      juniper_junos_config:
13|        provider:
14|          host: "{{ ansible_host }}"
15|          timeout: 120
16|        load: merge
17|        src: "config/nameserver.conf"
```

This playbook assumes that we are using SSH key authentication and the standard NETCONF port. If needed for your environment, add arguments user, passwd, or port to the provider dictionary.

Lines 1-8 are the now-familiar introduction to an Ansible playbook using Juniper's Galaxy modules.

Lines 11-17 are the task that uploads configuration file to the Junos devices.

Line 12 calls the juniper_junos_config module with the arguments on the following lines.

Line 13-15, the provider dictionary, contains the connection information. We have not previously used the timeout value on line 15. By default, juniper_junos_config uses a 30 second timeout for its RPC connections. Some devices take longer than that to commit a configuration change, particularly if the configuration is complex or if the device includes on-box validation scripts. Setting a longer timeout, 120 seconds in this example, instructs juniper_junos_config to wait a little longer before it declares an RPC timeout error.

Line 16, the load argument, tells juniper_junos_config that we want to load a configuration, and which type of load operation we wish to perform. There are several possible values for this setting; merge requests a "load merge." We will see some of the other possible values later in this chapter.

Line 17, the `src` argument, specifies the configuration file to load.

Run the playbook (with `--limit` if you do not wish to update all your devices):

```
mbp15:aja2 sean$ ansible-playbook install-config.yaml --limit=aragorn

PLAY [Install Configuration File] ************************************************

TASK [install configuration file onto device] ***********************************
changed: [aragorn]

PLAY RECAP **********************************************************************
aragorn                    : ok=1    changed=1    unreachable=0    failed=0
```

Notice that the "install configuration…" task recorded a `changed` status for *aragorn*, indicating the device's configuration was changed.

Now check the device's configuration:

```
sean@aragorn> show configuration | compare rollback 1
[edit system name-server]
    198.51.100.11 { ... }
+   198.51.100.100;

sean@aragorn> show configuration system name-server
8.8.8.8;
8.8.4.4;
198.51.100.10;
198.51.100.11;
198.51.100.100;
```

Observe that the new DNS server in `nameserver.conf` was added to the existing name servers on the device without changing existing servers, exactly as we would expect from a "load merge" operation.

To do a "load replace" instead, and replace the `name-server` hierarchy with what is in our configuration file, we need to make two changes. First, add the `replace:` tag to the `nameserver.conf` configuration file:

```
system {
    replace:
    name-server {
        8.8.4.4;
        8.8.8.8;
        198.51.100.100;
    }
}
```

Second, change the `load` argument in our playbook to request a replace operation. While we are editing the playbook, let's also add a commit comment so that anyone reviewing the commit history on the device will be able to see that the change was made with automation (new line 18):

```
...
10|  tasks:
11|    - name: install configuration file onto device
12|      juniper_junos_config:
13|        provider:
14|          host: "{{ ansible_host }}"
```

```
15|          timeout: 120
16|        load: replace
17|        src: "config/nameserver.conf"
18|        comment: install-config.yaml playbook with nameserver.conf file
```

Now run the playbook again (not shown) and check the results on the device:

```
sean@aragorn> show configuration | compare rollback 1
[edit system name-server]
!   8.8.4.4 { ... }
[edit system name-server]
-   198.51.100.10;
-   198.51.100.11;

sean@aragorn> show configuration system name-server
8.8.4.4;
8.8.8.8;
198.51.100.100;

sean@aragorn> show system commit
0   2018-03-09 05:04:58 UTC by sean via netconf
    install-config.yaml playbook with nameserver.conf file
1   2018-03-09 04:52:50 UTC by sean via netconf
2   2018-03-09 04:52:26 UTC by sean via cli
...
```

Observe that the device's previous name-server list has been replaced with the new list from nameserver.conf (at first glance the 8.8.8.x addresses do not appear to have changed because they were in both the device configuration and in the config file, but they have actually been replaced – look closely and notice that their order reversed!) Also notice the comment in the commit history.

Run the playbook again. This time notice that the "install configuration…" task returns an OK status for *aragorn* because no change was made to the device (the device's configuration matched what was in the config file):

```
...
TASK [install configuration file onto device] *********************************
ok: [aragorn]
...
```

Check the commit history again and notice that the configuration was not committed this time, because there was effectively no change to commit.

TIP   Should you have a playbook that needs to do a "load override" operation, use the argument load: override or load: overwrite.

Installing Set Commands with Ansible

Let's take a quick look at how you can install "set" style configurations using the juniper_junos_config module. We look at two approaches:

- Using a "set" configuration file, which is similar to what we did in the last section with a "text" configuration file.

- Including the set commands in the playbook, which is sometimes convenient for quick changes to one or two settings.

For the "set" configuration file, let's assume we want to configure the NTP server settings for our devices. Create file `config/ntp.set` as follows:

```
set system ntp boot-server 17.253.20.253
set system ntp server 17.253.20.253
set system ntp server 129.6.15.30
```

For the in-playbook commands, let's assume the DNS server 198.51.100.100 set in the previous section was supposed to be 198.51.100.101 and we need to fix it; this requires a set statement and a delete statement.

Create playbook `install-set.yaml` as follows (line numbers added for discussion):

```
 1|---
 2|- name: Install Configuration File
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|      timeout: 120
14|
15|  tasks:
16|    - name: install set file onto device
17|      juniper_junos_config:
18|        provider: "{{ connection_settings }}"
19|        load: set
20|        src: "config/ntp.set"
21|        comment: install-set.yaml playbook -- load ntp.set file
22|
23|    - name: install set commands onto device
24|      juniper_junos_config:
25|        provider: "{{ connection_settings }}"
26|        load: set
27|        lines:
28|          - delete system name-server 198.51.100.100
29|          - set system name-server 198.51.100.101
30|        comment: install-set.yaml playbook -- fix name server address
```

Line 10 starts a `vars` section of the play. We have not used a `vars` section previously; it allows us to define one or more variables that may be used in tasks in the play. A separate instance of each variable is created for each host, so these variables may contain host-specific information by referencing another (host-specific) variable.

Lines 11–13 define the variable `connection_settings` containing a dictionary. Line 12, the `host` key, is set to the value of each host's `ansible_host` variable, meaning the `host` key's value is specific to each host for which the playbook runs.

Why define the `connection_settings` variable? We have two `juniper_junos_config` tasks in this play, each of which needs a `provider` dictionary with connection information. Rather than repeating the exact same settings in two tasks, we put those settings in the `connection_settings` variable and reference that variable in each task (lines 18 and 25). Should we need to change the connection settings in the future, perhaps adjusting the timeout value or adding a port number, we can do so in one place, the `connection_settings` variable in the `vars` section, instead of in multiple tasks.

Lines 16–21 are the task that loads the `ntp.set` file onto the device. Note that line 19 sets the `load` argument to correspond with the file type.

Lines 23–30 are the task that loads the in-playbook commands onto the device. Note there is no `src` argument; instead, the `lines` argument starting on line 27 contains the list of commands (lines 28 and 29) to execute.

Let's run the playbook:

```
mbp15:aja2 sean$ ansible-playbook install-set.yaml --limit=aragorn

PLAY [Install Configuration File] ************************************************

TASK [install set file onto device] ************************************************
changed: [aragorn]

TASK [install set commands onto device] ************************************************
changed: [aragorn]

PLAY RECAP ************************************************************************
aragorn                    : ok=2    changed=2    unreachable=0    failed=0
```

Observe that both tasks returned a `changed` status.

Check on the device that the changes are what we expected and that the commit history reflects our changes:

```
sean@aragorn> show configuration | compare rollback 2
[edit system name-server]
    8.8.8.8 { ... }
+   198.51.100.101;
-   198.51.100.100;
[edit system ntp]
+   boot-server 17.253.20.253;
[edit system ntp]
    server 17.253.20.253 { ... }
+   server 129.6.15.30;

sean@aragorn> show system commit
0   2018-03-10 18:58:35 UTC by sean via netconf
    install-set.yaml playbook -- fix name server address
1   2018-03-10 18:58:32 UTC by sean via netconf
    install-set.yaml playbook -- load ntp.set file
...
```

Run the playbook again. Does it work a second time?

```
mbp15:aja2 sean$ ansible-playbook install-set.yaml --limit=aragorn

PLAY [Install Configuration File] ************************************************

TASK [install set file onto device] ********************************************
ok: [aragorn]

TASK [install set commands onto device] ****************************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: warning, bad_element: None, message: warning: statement not found)"}
        to retry, use: --limit @/Users/sean/aja2/install-set.retry

PLAY RECAP **********************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=1
```

The "install set file…" task worked and returned an OK status because the device's configuration did not change. However, the "install set command…" task failed. Note the error message says, in part, "warning: statement not found." Because this task tries to delete a setting that does not exist – we already deleted `name-server` `198.51.100.100` during the previous playbook run – Junos issues a warning. The `juniper_junos_config` module interprets that warning as a failure. Later in this chapter we discuss how to handle this situation by ignoring the warning.

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 *template* 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, but 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, configures a management IP and initial management user account, and enables SSH. Our playbook should set the device's hostname and DNS servers, enable NETCONF, and add our account with its 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 SSH 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:

```
aragorn    ansible_host=192.0.2.10      dns1=8.8.8.8      dns2=198.51.100.100
bilbo      ansible_host=198.51.100.5    dns1=8.8.4.4      dns2=198.51.100.101
```

NOTE   In Chapter 8 we 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 (`~/aja2/template`) to hold Jinja2 templates, then create file `base-settings.j2` within the `template` directory. Enter the following Jinja2 template, substituting your user account name and public SSH key for the author's (lines 4-10). (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...vPzOaX3gt8Uv 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|}
```

NOTE   This template creates a user account using Junos' default super-user class. This keeps our example template simple and allows us to use this account to modify various parts of the Junos configuration during examples in this book. However, if a user account does not need super-user permissions, the principle of least privilege suggests we should create a class with limited permissions and assign the user to that class. In Chapter 9, we do an example with a limited-permission account.

Most of the file is straightforward Junos configuration, as this template will create a Junos configuration file. However, we see one feature of the Jinja2 templating language on lines 2, 14, and 15: double curly braces (`{{ }}`) enclose a variable name. Line 2 of the template references 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 }}` reference is replaced with the contents of the appropriate variable before the configuration file is created. For example, line 2 of the template, "`host-name {{ inventory-hostname }};`" becomes "`host-name aragorn;`" in the generated configuration file for the *aragorn* device.

Now let's create the playbook. We will start with the tasks necessary to generate the configuration file from our template, then add the task to install the configuration file on the device a little later in this section. Create playbook `base-settings.yaml` as shown (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|  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 }}"
```

Lines 1 – 8 are basically the same as we have seen previously, identifying the playbook and the hosts to be processed and loading the Galaxy modules.

Lines 10 – 12 define two variables that are used later in the playbook. Instances of these variables are created for each device that is processed, so these variables can be used for 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 directory name from line 11 and Ansible's `inventory_hostname` for the device. For example, the path+filename for the *aragorn* device will be `tmp/aragorn.conf`.

The `debug` tasks on lines 15 and 17 display the contents of variables defined above; this is 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 directory exists or create it if it does not exist. The `path` argument (line 21) references 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, provide the Junos knowledge—processing the template just "fills in the blanks" where the template references device-specific variables.

Let's run the playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml


PLAY [Generate and Install Configuration File] ********************************

TASK [debug] *****************************************************************
ok: [aragorn] => {
    "tmp_dir": "tmp"
}
ok: [bilbo] => {
    "tmp_dir": "tmp"
}

TASK [debug] *****************************************************************
ok: [aragorn] => {
    "conf_file": "tmp/aragorn.conf"
}
ok: [bilbo] => {
    "conf_file": "tmp/bilbo.conf"
}

TASK [confirm or create configs directory] ************************************
changed: [aragorn]
ok: [bilbo]

TASK [save device information using template] *********************************
changed: [aragorn]
changed: [bilbo]

PLAY RECAP *******************************************************************
aragorn                    : ok=4    changed=2    unreachable=0    failed=0
bilbo                      : ok=4    changed=1    unreachable=0    failed=0
```

Notice in the output of the second debug task that each device's `conf_file` variable contains a unique name, which is appropriate as we are using the template to generate unique configuration files for each device.

Notice the task "confirm or create configs directory" returned 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 already existed) when it processed the second device. We really only need to execute this task once each time the

playbook is run, not once per device. We will make an alteration momentarily to address this.

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:aja2 sean$ cat tmp/aragorn.conf
system {
    host-name aragorn;
    login {
        user sean {
            uid 2000;
            class super-user;
            authentication {
                ssh-rsa "ssh-rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
            }
        }
    }
    replace:
    name-server {
        8.8.8.8;
        198.51.100.100;
    }
    services {
        netconf {
            ssh;
        }
    }
}

mbp15:aja2 sean$ cat tmp/bilbo.conf
system {
    host-name bilbo;
    login {
        user sean {
            uid 2000;
            class super-user;
            authentication {
                ssh-rsa "ssh-rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
            }
        }
    }
    replace:
    name-server {
        8.8.4.4;
        198.51.100.101;
    }
    services {
        netconf {
            ssh;
        }
    }
}
```

This is a good opportunity to check the configuration files for problems like missing braces or semicolons, or misspelled words, that might cause Junos to reject the configurations. If you find any issues, adjust 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|      run_once: yes
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|      juniper_junos_config:
37|        provider:
38|          host: "{{ ansible_host }}"
39|          user: "{{ username }}"
40|          passwd: "{{ password }}"
41|          port: 22
42|          timeout: 120
43|        src: "{{ conf_file }}"
44|        load: replace
45|        comment: "playbook base-settings.yaml"
```

Lines 14–21 add prompts for device username and password, as we saw in Chapters 4 and 5. Because this example 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.

Line 28, the `run_once: yes` option added to the task "confirm or create configs directory" ensures that task runs only once each time the playbook is executed, not once per device. The `run_once` option accepts a Boolean *true* or *false* value. However, Ansible accepts `yes` and `no` as synonyms for `True` and `False`, because `yes` and `no` are often easier to understand when reading a playbook.

Lines 35–45 are the task to push the configuration files to the devices. Building on the examples we saw earlier in this chapter, this task adds `user` and `passwd` arguments to pass the username and password to the `juniper_junos_config` module, and the `port` argument so the modules will use the standard SSH port.

Now run the updated playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml
Junos Username: sean
Junos Password:

PLAY [Generate and Install Configuration File] **********************************

TASK [confirm or create configs directory] *************************************
ok: [aragorn]

TASK [save device information using template] **********************************
ok: [aragorn]
ok: [bilbo]

TASK [install generated configuration file onto device] ************************
changed: [aragorn]
changed: [bilbo]

PLAY RECAP *********************************************************************
aragorn                    : ok=3    changed=1    unreachable=0    failed=0
bilbo                      : ok=2    changed=1    unreachable=0    failed=0
```

Notice that the task "confirm or create configs directory" ran only once, for only one device. That is thanks to the `run_once` setting we added to this task. Ansible selected the device under whose name the task would run, *aragorn* in this example. Which device Ansible selects is not important, only that the task runs once.

Notice that the task "save device information using template" returned a status of `ok`, even though it returned `changed` the last time we ran the playbook. Before the previous playbook run, the configuration files did not exist, so creating the file was a change. This time the configuration files already existed and, as we did not change the template, or any variables used by the template, there was no change in the file generated during this playbook run relative to the last playbook run.

Displaying Changes – Base Settings 1.2

Many Ansible core modules that make changes, such as the `template` module, can show what they are changing, you simply need to run the playbook with the `--diff` command-line argument.

The `juniper_junos_config` module can show what changed in a device's configuration. In fact, it records the change by default, but we need to capture and display the information, or ask the module to save the information to a file.

Modify the playbook as follows:

```
...
35|    - name: install generated configuration file onto device
36|      juniper_junos_config:
37|        provider:
38|          host: "{{ ansible_host }}"
39|          user: "{{ username }}"
40|          passwd: "{{ password }}"
41|          port: 22
42|          timeout: 120
43|        src: "{{ conf_file }}"
44|        load: replace
45|        comment: "playbook base-settings.yaml"
46|        diff: yes
47|        dest_dir: "{{ tmp_dir }}"
48|      register: config_results
49|
50|    - debug:
51|        var: config_results
```

Line 46 instructs the `juniper_junos_config` module to record the configuration change that was made (the difference in the configuration before and after loading the config file). This Boolean value defaults to *true* when loading configuration files; this line simply makes the default explicit for illustration. If you set `diff` to `no` or `False`, the module will not record any configuration change that was made.

Line 47 instructs the `juniper_junos_config` module to save configuration results to a file in the specified directory, here set to our temporary directory. The module will use a filename of `{{ ansible_host }}.diff`.

Line 48 records the module's results in variable `config_results`.

Lines 50-51 display the `config_results` variable, the results from the `juniper_junos_config` module.

In order to see changes when we run the playbook, roll back the last commit (from the previous run of the playbook) on one of your test devices:

```
sean@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# rollback 1
```

```
load complete

[edit]
sean@aragorn# commit and-quit
commit complete
Exiting configuration mode
```

Also delete the generated configuration file for that same device:

```
mbp15:aja2 sean$ rm tmp/aragorn.conf
```

Now run the playbook again, adding the --diff command-line agument:

```
mbp15:aja2 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: [aragorn]

TASK [save device information using template] *********************************
ok: [bilbo]
--- before
+++ after: /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmpEJzouu/base-settings.j2
@@ -0,0 +1,22 @@
+system {
+    host-name aragorn;
+    login {
+        user sean {
+            uid 2000;
+            class super-user;
+            authentication {
+                ssh-rsa "ssh-rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
+            }
+        }
+    }
+    replace:
+    name-server {
+        8.8.8.8;
+        198.51.100.100;
+    }
+    services {
+        netconf {
+            ssh;
+        }
+    }
+}

changed: [aragorn]

TASK [install generated configuration file onto device] ************************
changed: [aragorn]
ok: [bilbo]

TASK [debug] *******************************************************************
ok: [aragorn] => {
```

```
    "config_results": {
        "changed": true,
        "diff": "\n[edit system name-server]\n    8.8.8.8 { ... }\
n+  198.51.100.100;\n-   8.8.4.4;\n-   198.51.100.101;\n",
        "diff_lines": [
            "",
            "[edit system name-server]",
            "    8.8.8.8 { ... }",
            "+   198.51.100.100;",
            "-   8.8.4.4;",
            "-   198.51.100.101;"
        ],
        "failed": false,
        "file": "tmp/aragorn.conf",
        "msg": "Configuration has been: opened, loaded, checked, diffed, committed, closed."
    }
}
ok: [bilbo] => {
    "config_results": {
        "changed": false,
        "failed": false,
        "file": "tmp/bilbo.conf",
        "msg": "Configuration has been: opened, loaded, checked, diffed, closed."
    }
}

PLAY RECAP **********************************************************************
aragorn                    : ok=4    changed=2    unreachable=0    failed=0
bilbo                      : ok=3    changed=0    unreachable=0    failed=0
```

Observe the output of task "save device information using template." *Bilbo's* configuration file did not change but *aragorn's* file, which we deleted, needed to be recreated. The --diff argument caused Ansible's template module to display changes, so it showed a series of lines prefixed with a '+' character to indicate a line added to the file. Had we left the configuration file, but changed one of the DNS server IP addresses in the inventory file, we would have seen something like this, where the leading '-' indicates a line removed from the prior configuration file:

```
...
--- before: tmp/bilbo.conf
+++ after: /var/folders/y1/nqmc7hf13kz5rckn40p5jfbh0000gp/T/tmpHN85WZ/base-settings.j2
@@ -12,7 +12,7 @@
     replace:
     name-server {
         8.8.4.4;
-        198.51.100.101;
+        198.51.100.123;
     }
     services {
         netconf {

changed: [bilbo]
...
```

Observe the output of the debug task. *Aragorn's* configuration changed, and its config_results variable contains keys diff and diff_lines showing the changes made.

Bilbo's configuration did not change, and its `config_results` variable does not contain those keys.

Did the `juniper_junos_config` module save the changes in files?

```
mbp15:aja2 sean$ ls tmp/*.diff
tmp/192.0.2.10.diff

mbp15:aja2 sean$ cat tmp/192.0.2.10.diff

[edit system name-server]
    8.8.8.8 { ... }
+   198.51.100.100;
-   8.8.4.4;
-   198.51.100.101;
```

Aragorn's configuration change was saved. Because *bilbo's* configuration did not change, no `.diff` file was created.

Take a close look at the changes in the configuration files shown with the `--diff` argument, and contrast that with the differences returned by the `juniper_junos_config` module. Ansible's `template` module does a simple file compare, similar to what you would get from the UNIX/Linux `diff` command. It can tell which lines changed in the configuration file, but not whether those lines will alter the devices's configuration. With a change affecting only a line or two it may not be clear from the difference output which Junos hierarchy was modified, though `template` includes some unmodified lines before and after the changed lines to try to provide that context. This is not surprising; `template` knows nothing about the files it is generating.

By contrast, `juniper_junos_config` does the equivalent of a Junos `show | compare` after installing the configuration file but before committing the change. Because Junos understands its configuration hierarchy and what changed, it can provide more context about the change.

Cleaning Up Temporary Files – Base Settings 1.3

Let's add a task to our playbook to have it delete the generated configuration files – in other words, we'll have the playbook clean up its temporary files. Let's also stop saving the `.diff` files that we added in the last section.

Modify the playbook as follows:

```
...
35|    - name: install generated configuration file onto device
36|      juniper_junos_config:
37|        provider:
38|          host: "{{ ansible_host }}"
39|          user: "{{ username }}"
40|          passwd: "{{ password }}"
41|          port: 22
```

```
42|          timeout: 120
43|        src: "{{ conf_file }}"
44|        load: replace
45|        comment: "playbook base-settings.yaml"
46|        diff: yes
47|      register: config_results
48|
49|    - name: show configuration change
50|      debug:
51|        var: config_results.diff_lines
52|      when: config_results.diff_lines is defined
53|
54|    - name: delete generated configuration file
55|      file:
56|        path: "{{ conf_file }}"
57|        state: absent
```

The argument dest_dir: "{{ tmp_dir }}", formerly line 47, has been removed.

Lines 49–52 now print just the diff_lines element of the config_results variable (line 51), and only when diff_lines is defined (line 52). This avoids getting a "variable is not defined" error from a device that did not have a configuration change and thus whose config_results variable has no diff_lines element.

Lines 54–57 are a new task that deletes the generated configuration files. The task is similar to the task that creates our temporary directory, except that state: absent tells the file module to delete the file if it exists.

So that we can see a change in one device's configuration, manually change a name server setting on a test device. The author altered one DNS server IP on *bilbo*.

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:aja2 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: [aragorn]

TASK [save device information using template] ********************************
ok: [aragorn]
ok: [bilbo]

TASK [install generated configuration file onto device] *********************
ok: [aragorn]
changed: [bilbo]

TASK [show configuration change] *********************************************
skipping: [aragorn]
ok: [bilbo] => {
    "config_results.diff_lines": [
        "",
```

```
        "[edit system name-server]",
        "    8.8.4.4 { ... }",
        "+   198.51.100.101;",
        "-   198.51.100.123;"
    ]
}

TASK [delete generated configuration file] **************************************
changed: [aragorn]
changed: [bilbo]

PLAY RECAP **********************************************************************
aragorn                    : ok=4    changed=1    unreachable=0    failed=0
bilbo                      : ok=4    changed=2    unreachable=0    failed=0

mbp15:aja2 sean$ ls tmp/*.conf
ls: tmp/*.conf: No such file or directory
```

Notice that the task "show configuration change" shows ok: [bilbo] followed by some change details, because *bilbo's* configuration was changed by the "install generated configuration file onto device" task.

However, the task "show configuration change" shows skipping: [aragorn] because *aragorn's* configuration did not change. This means the config_results.diff_lines element was not defined, so the when conditional caused the debug task to be skipped.

Deleting Settings That Might Not Be Present

Let's update the template/base-settings.j2 template to delete the FTP and Telnet system services. These protocols are not encrypted and thus insecure, and their use should be avoided, but they might be enabled by default or may have been turned on during initial device setup; we want to ensure they are turned off.

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 configuration task as shown below. (If you did not get an error, run the playbook again, you will get the error the second time):

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>
```

```
PLAY [Generate and Install Configuration File] **********************************

TASK [confirm or create configs directory] **********************************
ok: [aragorn]

TASK [save device information using template] **********************************
changed: [aragorn]

TASK [install generated configuration file onto device] ************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: warning, bad_element: None, message: warning: statement not found)"}
        to retry, use: --limit @/Users/sean/aja2/base-settings.retry

PLAY RECAP *********************************************************************
aragorn                    : ok=2    changed=1    unreachable=0    failed=1
```

"Failure loading the configuraton…*warning: statement not found*." When you try to delete a configuration statement that does not exist, Junos issues a warning. The `juniper_junos_config` module sees that warning and issues an error, causing the playbook to stop processing the device in question.

The author's *aragorn* device has the following system services:

```
sean@aragorn> show configuration system services
ftp;
ssh {
    connection-limit 5;
    rate-limit 5;
}
netconf {
    ssh {
        connection-limit 5;
        rate-limit 5;
    }
}
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 can instruct the `juniper_junos_config` module to ignore warnings by using the `ignore_warning` argument. Add the `ignore_warning: yes` option to the "install generated configuration file onto device" task as shown:

```
...
35|   - name: install generated configuration file onto device
36|     juniper_junos_config:
37|       provider:
38|         host: "{{ ansible_host }}"
39|         user: "{{ username }}"
40|         passwd: "{{ password }}"
41|         port: 22
42|         timeout: 120
43|       src: "{{ conf_file }}"
```

```
44|      load: replace
45|      comment: "playbook base-settings.yaml"
46|      diff: yes
47|      ignore_warning: yes
48|    register: config_results
...
```

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>

PLAY [Generate and Install Configuration File] **********************************

TASK [confirm or create configs directory] ************************************
ok: [aragorn]

TASK [save device information using template] **********************************
ok: [aragorn]

TASK [install generated configuration file onto device] ************************
changed: [aragorn]

TASK [show configuration change] **********************************************
ok: [aragorn] => {
    "config_results.diff_lines": [
        "",
        "[edit system services]",
        "-    ftp;"
    ]
}

TASK [delete generated configuration file] ************************************
changed: [aragorn]

PLAY RECAP *********************************************************************
aragorn                    : ok=5    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:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml --limit=aragorn
Junos Username: sean
Junos Password: <enter password>
```

```
PLAY [Generate and Install Configuration File] ********************************

TASK [confirm or create configs directory] **********************************
ok: [aragorn]

TASK [save device information using template] ********************************
ok: [aragorn]

TASK [install generated configuration file onto device] *********************
changed: [aragorn]

TASK [show configuration change] ********************************************
ok: [aragorn] => {
    "config_results.diff_lines": [
        "",
        "[edit system services]",
        "-    web-management {",
        "-        http {",
        "-            interface fxp0.0;",
        "-        }",
        "-    }"
    ]
}

TASK [delete generated configuration file] **********************************
changed: [aragorn]

PLAY RECAP ******************************************************************
aragorn                    : ok=5    changed=2    unreachable=0    failed=0
```

Check your test device(s) and confirm that the FTP, Telnet, and web management services have all been deleted:

```
sean@aragorn> show configuration system services
ssh {
    connection-limit 5;
    rate-limit 5;
}
netconf {
    ssh {
        connection-limit 5;
        rate-limit 5;
    }
}

sean@aragorn> show configuration | compare rollback 2
[edit system services]
-    ftp;
-    web-management {
-        http {
-            interface fxp0.0;
-        }
-    }
```

NOTE   Using the argument `ignore_warning: yes`, as shown above, causes `juniper_ju-nos_config` to ignore *all* warnings from the device. To avoid ignoring warnings other than "statement not found" you can specify the exact text of the warning you wish to ignore: `ignore_warning: "statement not found"`.

Commit Confirmed – Base Settings 1.4

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 or changed the IP address of the interface being used to manage the device – the device will automatically revert to its prior state and (hopefully) restore service.

Automation should reduce the need for "commit confirmed" because automation should reduce human error. However, if the source data for the configuration templates is created by humans, there is still a potential for human error.

Let's add "commit confirmed" to our playbook. This requires that the playbook calls the `juniper_junos_config` module twice:

- Load the configuration change and perform a "commit confirmed."
- Perform a simple "commit" and thereby confirm the previous commit.

We use a *handler* for the second call to the `juniper_junos_config` module. A handler is a special task, one triggered (notified in Ansible's lingo) only when another task causes a change. This approach works well for our "commit confirmed" operation. If the first call to `juniper_junos_config` changes the device's configuration, the task will notify the handler to confirm the commit. However, if the first call to `juniper_junos_config` does not change the device's configuration, we have no change to commit and thus no need of making the second call to `juniper_junos_config`.

Modify the `base-settings.yaml` playbook as follows:

```
 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|    connection_settings:
14|      host: "{{ ansible_host }}"
15|      user: "{{ username }}"
```

```
16|      passwd: "{{ password }}"
17|      port: 22
18|      timeout: 120
19|
20|  vars_prompt:
21|    - name: username
22|      prompt: Junos Username
23|      private: no
24|
25|    - name: password
26|      prompt: Junos Password
27|      private: yes
28|
29|  tasks:
30|    - name: confirm or create configs directory
31|      file:
32|        path: "{{ tmp_dir }}"
33|        state: directory
34|      run_once: yes
35|
36|    - name: save device information using template
37|      template:
38|        src: template/base-settings.j2
39|        dest: "{{ conf_file }}"
40|
41|    - name: install generated configuration file onto device
42|      juniper_junos_config:
43|        provider: "{{ connection_settings }}"
44|        src: "{{ conf_file }}"
45|        load: replace
46|        comment: "playbook base-settings.yaml, commit confirmed"
47|        confirmed: 5
48|        diff: yes
49|        ignore_warning: yes
50|      register: config_results
51|      notify: confirm previous commit
52|
53|    - name: show configuration change
54|      debug:
55|        var: config_results.diff_lines
56|      when: config_results.diff_lines is defined
57|
58|    - name: delete generated configuration file
59|      file:
60|        path: "{{ conf_file }}"
61|        state: absent
62|
63|  handlers:
64|    - name: confirm previous commit
65|      juniper_junos_config:
66|        provider: "{{ connection_settings }}"
67|        comment: "playbook base-settings.yaml, confirming previous commit"
68|        commit: yes
69|        diff: no
```

Lines 13–18 and line 43 move the `juniper_junos_config` module's `provider` settings into their own variable so we do not need to repeat the settings in both calls to the

juniper_junos_config module. This is similar to the approach we used in the play-book install-set.yaml earlier in this chapter.

Line 47 instructs the juniper_junos_config module to do a "commit confirmed 5" instead of a simple "commit" operation. The confirm time – 5 minutes in this ex-ample – can be adjusted as needed.

Line 51 causes the "install generated configuration file onto device" task to notify the handler when there is a configuration change.

Line 63 introduces a new handlers section to our playbook. Like the tasks section, the handlers section contains a list of one or more tasks, but these special tasks, called *handlers*, are executed only when they are *notified* (called) by another task.

Lines 64–69 are the new handler that confirms our earler commit. Line 68 tells the juniper_junos_config module to perform a "commit" operation. Notice there are no src or lines or load arguments; we are not loading any configuration settings.

In order to see the handler work, we need our device's configuration to be different from what our template will create. Delete the name-server hierarchy from one of your test devices:

```
sean@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# delete system name-server

[edit]
sean@aragorn# commit and-quit
commit complete
Exiting configuration mode
```

Now run the playbook, preferably on several devices:

```
mbp15:aja2 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: [aragorn]

TASK [save device information using template] ***********************************
changed: [bilbo]
changed: [aragorn]

TASK [install generated configuration file onto device] *************************
changed: [aragorn]
ok: [bilbo]

TASK [show configuration change] ***********************************************
skipping: [bilbo]
ok: [aragorn] => {
```

```
    "config_results.diff_lines": [
        "",
        "[edit system]",
        "+  name-server {",
        "+      8.8.8.8;",
        "+      198.51.100.100;",
        "+  }"
    ]
}

TASK [delete generated configuration file] **************************************
changed: [aragorn]
changed: [bilbo]

RUNNING HANDLER [confirm previous commit] ***************************************
ok: [aragorn]

PLAY RECAP *********************************************************************
aragorn                    : ok=6    changed=3    unreachable=0    failed=0
bilbo                      : ok=3    changed=2    unreachable=0    failed=0
```

Observe that the task "install generated configuration file onto device" reported that *aragorn* changed but *bilbo* did not. As a result, the task notified the handler "confirm previous commit" only for *aragorn*; the handler did not run for *bilbo*.

Check the commit history on the device which changed. Look closely at the last two commits:

```
sean@aragorn> show system commit
0    2018-03-15 17:21:04 UTC by sean via netconf
     playbook base-settings.yaml, confirming previous commit
1    2018-03-15 17:20:53 UTC by sean via netconf commit confirmed, rollback in 5mins
     playbook base-settings.yaml, commit confirmed
...
```

You can see that commit #1 was "commit confirmed, rollback in 5 mins" – that was the install task – and commit #0 is the second, confirming commit.

When running a playbook that uses this "commit confirmed" approach, 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 moving on to the next task, which means all devices must complete the task that loads the configuration before Ansible will start the handlers that confirm the commits. The confirm time must be long enough for the configuration load task to complete. 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 problem, 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.

TIP   Readers considering the "commit confirmed" approach shown in this section for their playbooks should look at the `serial` option for Ansible plays. `Serial` allows you to specify a "batch size" so that Ansible limits the number of hosts being processed during one execution of the play, repeating the play as many times as needed with the next "batch" of hosts until all hosts have been completed. Using `serial` can let you specify a reasonably short confirm time without needing to manually `--limit` the hosts being processed into small sets.

Loading Configuration Via Console

The initial configuration of new-out-of-box (NOOB) Junos devices is normally done via the device's serial console port. The scenario described near the beginning of the "Generating 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. What if that assumption is not valid? What if we want our automation to handle the NOOB setup via console?

Juniper's Galaxy modules provide a `mode` argument that can instruct the module to operate over a transport other than SSH, which typically means a console connection. The `mode` option currently supports direct serial connections and Telnet, which typically means Telnet to a terminal server.

Assume we want to complete an initial configuration on an EX switch via a serial console 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 *aja2*, add a few ports to the VLAN using an interface-range called *aja2*, and configure a layer-3 interface on the VLAN. The following Jinja2 template, `template/initial-ex-vlan.j2`, (line numbers added) fulfills these requirements:

```
1|system {
2|    host-name {{ inventory_hostname }};
3|    root-authentication {
4|        encrypted-password "$1$7as5CZnA$Xc1QTe5dW2ph8Y59l8.0j1";
5|    }
6|    login {
7|        user sean {
8|            uid 2000;
9|            class super-user;
10|            authentication {
11|                encrypted-password "$1$U25qzyIz$AxMzsxhD/lj1wlDpd1f96.";
12|            }
13|        }
14|    }
15|    services {
16|        ssh;
17|        netconf {
```

```
18|            ssh;
19|        }
20|    }
21|}
22|interfaces {
23|    interface-range aja2 {
24|        member ge-0/1/0;
25|        member ge-0/1/1;
26|        unit 0 {
27|            family ethernet-switching {
28|                port-mode access;
29|                vlan {
30|                    members aja2;
31|                }
32|            }
33|        }
34|    }
35|    vlan {
36|        unit 0 {
37|            family inet {
38|                address {{ ansible_host }}/{{ netmask }};
39|            }
40|        }
41|    }
42|}
43|vlans {
44|    aja2 {
45|        l3-interface vlan.0;
46|    }
47|}
```

Insert your username in line 7 in place of the author's. Use the correct encrypted password for your root and user accounts on lines 4 and 11; you can copy the encrypted strings from the existing configuration on your test switch. (In Chapter 11 we discuss securely storing credentials and other sensitive information, but for now just include the encrypted passwords directly in the template.)

The author's switch is an EX-2200-C that uses the legacy VLAN command set; if your test switch is a newer device with the ELS command set, replace the keyword vlan with keyword irb on line 35, and replace vlan.0 with irb.0 on line 45. Substitute different interfaces on lines 24 and 25 if needed for your switch or environment.

There are three variable references in the template. Line 2 sets the switch's hostname using Ansible's inventory_hostname variable. Line 38 sets the switch's IP address using Ansible's ansible_host variable. (If you have avoided assigning 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 38 also references a variable netmask. This variable is defined in the playbook for this example. (After we discuss host and group data files in Chapter 8, you may

wish to relocate this variable's declaration to a data file.)

NOTE   This is a fairly minimal configuration, intended for our discussion. You could easily extend this template to include all the settings from the "base settings" playbooks and templates and any other settings you need for a NOOB device.

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-serial.yaml`:

```
 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|    netmask: "26"
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|      run_once: yes
31|
32|    - name: save device information using template
33|      template:
34|        src: template/initial-ex-vlan.j2
35|        dest: "{{ conf_file }}"
36|
37|    - name: install generated configuration file onto device
38|      juniper_junos_config:
39|        provider:
40|          host: "{{ inventory_hostname }}"
41|          mode: serial
42|          port: "/dev/cu.usbserial-AH02PIG9"
```

```
43|          user: "{{ username }}"
44|          # passwd: "{{ password }}"
45|          timeout: 120
46|       src: "{{ conf_file }}"
47|       load: override
48|       comment: "playbook initial-setup-serial.yaml"
49|
50|    # - name: delete generated configuration file
51|    #   file:
52|    #     path: "{{ conf_file }}"
53|    #     state: absent
```

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 in the discussion about setting the switch's IP address.

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 after running the "request system zeroize" command to reset the device to factory default settings. 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 44 (`passwd`); this will cause the playbook to prompt for username and password and provide both when accessing the device.

Lines 26–35 are the same as the corresponding lines from the `base-settings.yaml` playbook, ensuring the `tmp` directory exists and generating the configuration file from the template; the only change is the template name on line 34.

Lines 37–48 install the generated configuration on the devices, similar to what we saw in the `base-settings.yaml` playbook, but there are several changes related to console access. Let's discuss those changes.

Line 40, the `host` argument, is required for the `juniper_junos_config` module but is not really relevant for direct serial console access (recall that `host` normally specifies the target for the NETCONF-over-SSH connection). The playbook sets it to `inventory_hostname` just so it has a value. We could also omit the setting because the `juniper_junos_config` module will use `inventory_hostname` as a default value.

Line 41, the argument `mode: serial`, informs the `juniper_junos_config` module that it should connect to the device using a local serial port, not a normal NETCONF-over-SSH session.

Line 42, the `port` argument, takes on a new meaning in the context of `mode: serial`. Normally port specifies the TCP port for the NETCONF-over-SSH connection, but that is not useful for serial connections. Instead, `port` in the context of serial connections specified the serial port over which the `juniper_junos_config` module should talk to the device. *You must set the value of* `port` *to correspond with your computer's serial port or USB serial adapter*, not the author's USB 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 -l /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 appear, your adapter may not be supported by macOS or Linux, or you may need to install drivers to add support.

Line 47, the argument `load: override`, informs the `juniper_junos_config` module that it should use the equivalent of the Junos `load override` command when loading the configuration file. In other words, completely replace the device's existing configuration with what is being loaded. This is normally a good choice for an initial setup playbook and template as there are often default settings, including VLAN and interface settings, that might interfere with the new settings we want to make.

Lines 50-53 delete the generated configuration file. They are shown commented out to give you the opportunity to inspect the file if you wish to do so.

The author just ran "request system zeroize" on *bilbo* so it looks like a NOOB device. Let's run the playbook:

```
mbp15:aja2 sean$ ansible-playbook initial-setup-serial.yaml --limit=bilbo

PLAY [Generate and Install Configuration File] *********************************

TASK [confirm or create configs directory] ************************************
ok: [bilbo]

TASK [save device information using template] *********************************
changed: [bilbo]

TASK [install generated configuration file onto device] ***********************
changed: [bilbo]

PLAY RECAP ********************************************************************
bilbo                      : ok=3    changed=2    unreachable=0    failed=0
```

Keep in mind that 9600bps serial connections are a lot slower than typical network connections; expect the "install generated configuration file onto device" task to take a lot longer than you saw with the `base-settings.yaml` playbook.

After the playbook ends, check the config on your switch. Also look at the commit

history, which should be only a few entries long if you reset your device:

```
sean@bilbo> show system commit
0   2016-01-20 07:29:54 UTC by root via netconf
    playbook initial-setup-serial.yaml
1   2016-01-20 07:26:44 UTC by root via button
2   2016-01-20 07:24:52 UTC by root via other
```

Now let's discuss `initial-setup-ts.yaml`, a variation of the above playbook that will work with Telnet-based terminal server (console server) connections:

```
 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|    netmask: "26"
14|    username: root
15|    terminal_server: 198.51.100.50
16|    term_srv_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|      run_once: yes
33|
34|    - name: save device information using template
35|      template:
36|        src: template/initial-ex-vlan.j2
37|        dest: "{{ conf_file }}"
38|
39|    - name: install generated configuration file onto device
40|      juniper_junos_config:
41|        provider:
42|          host: "{{ terminal_server }}"
43|          mode: telnet
44|          port: "{{ term_srv_port }}"
45|          user: "{{ username }}"
46|          # passwd: "{{ password }}"
47|          timeout: 120
48|        src: "{{ conf_file }}"
```

```
49|        load: override
50|        comment: "playbook initial-setup-ts.yaml"
51|
52|    # - name: delete generated configuration file
53|    #    file:
54|    #      path: "{{ conf_file }}"
55|    #      state: absent
```

Most of the `initial-setup-ts.yaml` playbook is the same as the `initial-setup-con.yaml` playbook, so we will discuss only the changes.

Lines 15 and 16 define two 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 terminal server*. Typically, terminal servers either assign a unique IP to each of their serial ports and use the standard Telnet port TCP:23, or they use a single IP but assign a unique TCP port number to each serial port.

TIP   The terminal server IP address or port number, or both, are likely to be specific to each Junos device, based on which serial port on the terminal server is connected to the device's console port. After reading Chapter 8 you may wish to move these variables to host or group data files.

Line 42, the `host` argument, provides the terminal server's address to the `juniper_junos_config` module, so the playbook sets `host` to the value of the new `terminal_server` variable.

Line 43, the `mode: telnet` argument, tells `juniper_junos_config` to use Telnet to connect to the device, not the default SSH. However, `juniper_junos_config` accepts a single username and password and expects a single authentication step. As Junos requires authentication, even if only the NOOB authentication of `root` with no password, the terminal server should be configured to require no authentication.

Line 44, the `port` argument, provides the correct Telnet port number to the `juniper_junos_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:aja2 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
```

If you reset your test switch, run `base-settings.yaml` against it to reinstall your SSH public key and other settings.

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 installed the generated configuration file. This is a process the author has used many times when developing new playbooks and templates.

But what if you are updating a template used by an existing playbook that already includes the tasks to install the configuration, and perhaps even cleans up the generated configuration file as our `base-settings.yaml` playbook does? It becomes hard to debug a template if the playbook deletes the file generated from the template before you can look at it!

In this situation, the author uses a combination of two approaches: comment out anything not needed for the playbook to process the template that precedes the `template` task, and terminate the playbook after the `template` task.

Assume we are updating our `template/base-settings.j2` template and want to inspect the generated configuration file without trying to install it. We could modify the `base-settings.yaml` playbook as follows (only the first 45 lines shown):

```
 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|    connection_settings:
14|      host: "{{ ansible_host }}"
15|      user: "{{ username }}"
16|      passwd: "{{ password }}"
17|      port: 22
18|      timeout: 120
19|
20|  # vars_prompt:
21|  #   - name: username
22|  #     prompt: Junos Username
23|  #     private: no
24|  #
```

```
25|  #    - name: password
26|  #      prompt: Junos Password
27|  #      private: yes
28|
29|  tasks:
30|    - name: confirm or create configs directory
31|      file:
32|        path: "{{ tmp_dir }}"
33|        state: directory
34|      run_once: yes
35|
36|    - name: save device information using template
37|      template:
38|        src: template/base-settings.j2
39|        dest: "{{ conf_file }}"
40|
41|    - fail:
42|        msg: "early exit for template troubleshooting"
43|
44|    - name: install generated configuration file onto device
45|      juniper_junos_config:
...
```

Lines 20-27 are commented out because we do not need to provide device credentials if we are not going to install the generated configuration on a device.

Lines 41-42 call Ansible's `fail` method to end the playbook at that point. Nothing past here will execute. This causes the playbook to end immediately after processing the template (lines 36-39), which will let us inspect the generated file.

NOTE   The `fail` method is normally used with a `when` condition to let us end processing for a specific device when we detect a problem, but it works well for our purpose also.

Running the playbook would look like this:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml --limit=aragorn

PLAY [Generate and Install Configuration File] ***********************************

TASK [confirm or create configs directory] *************************************
ok: [aragorn]

TASK [save device information using template] ***********************************
ok: [aragorn]

TASK [fail] ********************************************************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "early exit for template troubleshooting"}
        to retry, use: --limit @/Users/sean/aja2/base-settings.retry

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

mbp15:aja2 sean$ ls tmp/*.conf
tmp/aragorn.conf
```

As its name implies, the `fail` module records a failure. This is not a problem for our purposes. Just remember to delete that task after the template is debugged!

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 configuration file contains the following:

```
...
    replace:
    name-server {
        { dns1 }};
        5.6.7.9;
    }
...
```

Clearly { dns1 }} is not the expected output. An inspection of the template shows that the problem was caused by a missing left brace ("{"):

```
    replace:
    name-server {
        { dns1 }};
        {{ dns2 }};
    }
```

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: [aragorn]: 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 previously 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: [aragorn]: 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        netconf {\n          ssh;\n        }\n        telnet;\n        delete: telnet;\n
web-management {\n          http;\n        }\n        delete: web-management;\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 Ansible'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 parentheses, brackets, and braces – 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; observe 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:

```
12        replace:¬
13        name-server {¬
14            {{ dns1 };¬
15            {{ dns2 }};¬
16        }¬
```

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: [aragorn]: 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)"}
...
```

Here the error message tells us we have a "bad_element" -- "Unable to load config: ConfigLoadError(severity: error, *bad_element: name-servers*, message: error: syntax error..." Searching the template file for the incorrect element *name-servers* finds the following:

```
12|   replace:
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: [aragorn]: 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 problem that we discussed earlier in this chapter, trying to delete a non-existent configuration element:

```
...
TASK [install generated configuration file onto device] *************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
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 attempting to delete it, or add the `ignore_warning: yes` option to the task.

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 playbook's action. Junos usually does a good job of identifying the problem, but some of the detail is lost when Junos' warnings or errors are passed back through the automation tools.

References

Jinja2 information:
http://jinja.pocoo.org
http://docs.ansible.com/ansible/latest/playbooks_templating.html

Juniper_junos_config module:
http://junos-ansible-modules.readthedocs.io/en/stable/juniper_junos_config.html

Ansible handlers:
http://docs.ansible.com/ansible/latest/playbooks_intro.
html#handlers-running-operations-on-change

Ansible *serial* option:
http://docs.ansible.com/ansible/latest/playbooks_delegation.
html#rolling-update-batch-size

Ansible *fail* module:
http://docs.ansible.com/ansible/latest/fail_module.html

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.

In addition, Ansible can use a *dynamic inventory*, meaning Ansible queries an external system, such as a configuration management database, for information about the managed devices. Setting up a dynamic inventory for Ansible is outside the scope of this book, but there is a link in the references section at the end of this chapter for readers who wish to explore this idea further.

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 discuss later in this chapter.

- Facts discovered from the hosts being managed by the playbook.

- Variables set in the inventory file, as we have shown in prior chapters, or set in the host and group variable files that we discuss later in this chapter.

- Registered variables set using the `register` option to capture the results of a task, as we have seen in several playbooks including `uptime.yaml` and `interfaces.yaml`.

- 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, which we see 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 more in later chapters.

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`:

```
 1|---
 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:
15|        var: test1
16|
17|    - debug:
18|        var: Test1
19|
20|    - debug:
21|        var: name_plus_host
22|
23|- name: Show variables 1, second play
24|  hosts:
25|    - all
26|  connection: local
27|  gather_facts: no
```

```
28|
29|  tasks:
30|    - debug:
31|        var: test1
32|
33|    - debug:
34|        var: Test1
35|
36|    - debug:
37|        var: name_plus_host
```

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook show-vars-1.yaml

PLAY [Show variables 1, first play] ******************************************

TASK [debug] ****************************************************************
ok: [aragorn] => {
    "test1": "test all lower-case"
}
ok: [bilbo] => {
    "test1": "test all lower-case"
}

TASK [debug] ****************************************************************
ok: [aragorn] => {
    "Test1": "Test first capital"
}
ok: [bilbo] => {
    "Test1": "Test first capital"
}

TASK [debug] ****************************************************************
ok: [aragorn] => {
    "name_plus_host": "aragorn :: 192.0.2.10"
}
ok: [bilbo] => {
    "name_plus_host": "bilbo :: 198.51.100.5"
}

PLAY [Show variables 1, second play] *****************************************

TASK [debug] ****************************************************************
ok: [aragorn] => {
    "test1": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "test1": "VARIABLE IS NOT DEFINED!"
}

TASK [debug] ****************************************************************
ok: [aragorn] => {
    "Test1": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "Test1": "VARIABLE IS NOT DEFINED!"
}
```

```
TASK [debug] ******************************************************************
ok: [aragorn] => {
    "name_plus_host": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "name_plus_host": "VARIABLE IS NOT DEFINED!"
}

PLAY RECAP ********************************************************************
aragorn                    : ok=6    changed=0    unreachable=0    failed=0
bilbo                      : 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 defined. 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's name clashes with the name of one variable in the vars section of the first play:

```
mbp15:aja2 sean$ ansible-playbook show-vars-1.yaml --extra-vars 'test1="test one extra"'

PLAY [Show variables 1, first play] ******************************************

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "test1": "test one extra"
}
ok: [bilbo] => {
    "test1": "test one extra"
}

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "Test1": "Test first capital"
}
ok: [bilbo] => {
    "Test1": "Test first capital"
}

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "name_plus_host": "aragorn :: 192.0.2.10"
}
ok: [bilbo] => {
    "name_plus_host": "bilbo :: 198.51.100.5"
}

PLAY [Show variables 1, second play] *****************************************

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "test1": "test one extra"
}
```

```
}
ok: [bilbo] => {
    "test1": "test one extra"
}

TASK [debug] ********************************************************************
ok: [aragorn] => {
    "Test1": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "Test1": "VARIABLE IS NOT DEFINED!"
}

TASK [debug] ********************************************************************
ok: [aragorn] => {
    "name_plus_host": "VARIABLE IS NOT DEFINED!"
}
ok: [bilbo] => {
    "name_plus_host": "VARIABLE IS NOT DEFINED!"
}

PLAY RECAP ********************************************************************
aragorn                    : ok=6    changed=0    unreachable=0    failed=0
bilbo                      : 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*), and for both devices, in contrast to the other variables whose scope is the play and device for which they were defined.

Now create playbook show-vars-2.yaml:

```
 1|---
 2|- name: Show variables 2, first play
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|  tasks:
10|    - debug:
11|        var: inventory_hostname
12|
13|    - debug:
14|        var: dns1
15|
16|    - name: get device uptime
17|      juniper_junos_command:
18|        commands:
19|          - show system uptime
20|        provider:
21|          host: "{{ ansible_host }}"
22|        formats: xml
23|      register: uptime
24|
```

```
25|     – name: query uptime information
26|       xml:
27|         xmlstring: "{{ uptime.stdout }}"
28|         xpath: //current–time/date–time
29|         content: text
30|       register: current_time
31|
32|     – debug:
33|         var: current_time.matches
34|
35|     – set_fact:
36|         device_time: "{{ current_time.matches[0] }}"
37|
38|     – debug:
39|         var: device_time
40|
41|
42|– name: Show variables 2, second play
43|  hosts:
44|     – all
45|  connection: local
46|  gather_facts: no
47|  tasks:
48|     – debug:
49|         var: inventory_hostname
50|
51|     – debug:
52|         var: dns1
53|
54|     – debug:
55|         var: current_time.matches
56|
57|     – debug:
58|         var: device_time
```

Most of this playbook should be familiar, but let's quickly discuss it. The first play of this playbook displays the "magic" variable `inventory_hostname` and the variable `dns1` set in the inventory file (lines 10-14). The play then queries each device for its uptime information in XML format, from which it extracts the device's current time using XPath (lines 16-30) and displays the XPath matches (lines 32-33).

The `set_fact` module (lines 35-36) is new. Set_fact defines one or more *facts*, a type of variable that is device-specific and whose scope is the rest of the playbook. Here the playbook assigns the device's current time to a new fact `device_time`.

The second play of the playbook just displays four variables to show whether the variable's scope extend outside of the first play.

Run the playbook:

```
mbp15:aja2 sean$ ansible–playbook show–vars–2.yaml

PLAY [Show variables 2, first play] ********************************************

TASK [debug] ******************************************************************
ok: [aragorn] => {
```

```
    "inventory_hostname": "aragorn"
}
ok: [bilbo] => {
    "inventory_hostname": "bilbo"
}

TASK [debug] ***********************************************************
ok: [aragorn] => {
    "dns1": "8.8.8.8"
}
ok: [bilbo] => {
    "dns1": "8.8.4.4"
}

TASK [get device uptime] **********************************************
ok: [aragorn]
ok: [bilbo]

TASK [query uptime information] ***************************************
ok: [aragorn]
ok: [bilbo]

TASK [debug] ***********************************************************
ok: [aragorn] => {
    "current_time.matches": [
        {
            "date-time": "2018-03-17 16:42:16 UTC"
        }
    ]
}
ok: [bilbo] => {
    "current_time.matches": [
        {
            "date-time": "2016-01-21 02:30:13 UTC"
        }
    ]
}

TASK [set_fact] *******************************************************
ok: [aragorn]
ok: [bilbo]

TASK [debug] ***********************************************************
ok: [aragorn] => {
    "device_time": {
        "date-time": "2018-03-17 16:42:16 UTC"
    }
}
ok: [bilbo] => {
    "device_time": {
        "date-time": "2016-01-21 02:30:13 UTC"
    }
}

PLAY [Show variables 2, second play] *********************************

TASK [debug] ***********************************************************
ok: [aragorn] => {
```

```
    "inventory_hostname": "aragorn"
}
ok: [bilbo] => {
    "inventory_hostname": "bilbo"
}

TASK [debug] *********************************************************
ok: [aragorn] => {
    "dns1": "8.8.8.8"
}
ok: [bilbo] => {
    "dns1": "8.8.4.4"
}

TASK [debug] *********************************************************
ok: [aragorn] => {
    "current_time.matches": [
        {
            "date-time": "2018-03-17 16:42:16 UTC"
        }
    ]
}
ok: [bilbo] => {
    "current_time.matches": [
        {
            "date-time": "2016-01-21 02:30:13 UTC"
        }
    ]
}

TASK [debug] *********************************************************
ok: [aragorn] => {
    "device_time": {
        "date-time": "2018-03-17 16:42:16 UTC"
    }
}
ok: [bilbo] => {
    "device_time": {
        "date-time": "2016-01-21 02:30:13 UTC"
    }
}

PLAY RECAP **********************************************************
aragorn                    : ok=11   changed=0    unreachable=0    failed=0
bilbo                      : ok=11   changed=0    unreachable=0    failed=0
```

Hmmm...it looks like the author needs to configure NTP on *bilbo*! We'll take care of that with a playbook a little later.

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 variables defined in host and group variables files, discussed later in this chapter. 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:

```
 1|---
 2|- name: Show variables 3
 3|  hosts:
 4|    - all
 5|  connection: local
 6|  gather_facts: yes
 7|
 8|  tasks:
 9|    - name: ansible variables
10|      debug:
11|        var: vars
```

Notice that line 6 sets gather_facts to yes; this instructs Ansible to gather additional data and set additional facts that we did not care about in earlier playbooks. Because of the connection: local setting the gathered facts will be for the control machine, the local host on which you are running Ansible, not the managed devices, but it is still interesting to see all the data gathered. For less data (to avoid gathering data from the local host) just change gather_facts to no.

The following output, edited for length, shows the playbook limited to a single device. There is a lot of repetition in the output, and there is even more repetition when the playbook is run for multiple devices, but you should try it with two or three devices to get a feel for which variables are host-specific:

```
mbp15:aja2 sean$ ansible-playbook show-vars-3.yaml --limit=aragorn

PLAY [Show variables 3] ********************************************************

TASK [Gathering Facts] ********************************************************
ok: [aragorn]

TASK [ansible variables] ********************************************************
ok: [aragorn] => {
    "vars": {
...
        "ansible_check_mode": false,
        "ansible_date_time": {
            "date": "2018-03-17",
            "day": "17",
            "epoch": "1521302344",
            "hour": "11",
            "iso8601": "2018-03-17T15:59:04Z",
            "iso8601_basic": "20180317T115904532376",
            "iso8601_basic_short": "20180317T115904",
            "iso8601_micro": "2018-03-17T15:59:04.532476Z",
            "minute": "59",
            "month": "03",
```

```
            "second": "04",
            "time": "11:59:04",
            "tz": "EDT",
            "tz_offset": "-0400",
            "weekday": "Saturday",
            "weekday_number": "6",
            "weeknumber": "11",
            "year": "2018"
        },
...
        "ansible_host": "192.0.2.10",
        "ansible_hostname": "mbp15",
...
        "ansible_machine": "x86_64",
        "ansible_memfree_mb": 80,
        "ansible_memtotal_mb": 16384,
        "ansible_model": "MacBookPro11,3",
        "ansible_nodename": "mbp15.local",
        "ansible_os_family": "Darwin",
        "ansible_osrevision": "199506",
        "ansible_osversion": "17D102",
...
        "ansible_play_batch": [
            "aragorn"
        ],
        "ansible_play_hosts": [
            "aragorn"
        ],
        "ansible_play_hosts_all": [
            "aragorn"
        ],
        "ansible_playbook_python": "/usr/local/opt/python/bin/python2.7",
        "ansible_processor": "Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz",
        "ansible_processor_cores": "4",
...
        "ansible_python_interpreter": "/usr/local/bin/python",
        "ansible_python_version": "2.7.14",
...
        "ansible_version": {
            "full": "2.4.3.0",
            "major": 2,
            "minor": 4,
            "revision": 3,
            "string": "2.4.3.0"
        },
...
        "dns1": "8.8.8.8",
        "dns2": "198.51.100.100",
        "environment": [],
        "gather_subset": [
            "all"
        ],
        "group_names": [
            "ungrouped"
        ],
        "groups": {
            "all": [
                "aragorn",
```

```
                    "bilbo"
                ],
            "ungrouped": [
                "aragorn",
                "bilbo"
            ]
        },
        "hostvars": {
            "aragorn": {
...
                "ansible_check_mode": false,
                "ansible_date_time": {
                    "date": "2018-03-17",
                    "day": "17",
                    "epoch": "1521302344",
                    "hour": "11",
                    "iso8601": "2018-03-17T15:59:04Z",
                    "iso8601_basic": "20180317T115904532376",
                    "iso8601_basic_short": "20180317T115904",
                    "iso8601_micro": "2018-03-17T15:59:04.532476Z",
                    "minute": "59",
                    "month": "03",
                    "second": "04",
                    "time": "11:59:04",
                    "tz": "EDT",
                    "tz_offset": "-0400",
                    "weekday": "Saturday",
                    "weekday_number": "6",
                    "weeknumber": "11",
                    "year": "2018"
                },
...
                "ansible_host": "192.0.2.10",
                "ansible_hostname": "mbp15",
...
                "ansible_python_interpreter": "/usr/local/bin/python",
                "ansible_python_version": "2.7.14",
...
                "ansible_version": {
                    "full": "2.4.3.0",
                    "major": 2,
                    "minor": 4,
                    "revision": 3,
                    "string": "2.4.3.0"
                },
...
                "dns1": "8.8.8.8",
                "dns2": "198.51.100.100",
                "gather_subset": [
                    "all"
                ],
                "group_names": [
                    "ungrouped"
                ],
                "groups": {
                    "all": [
                        "aragorn",
                        "bilbo"
```

```
                        ],
                        "ungrouped": [
                            "aragorn",
                            "bilbo"
                        ]
                    },
                    "inventory_dir": "/Users/sean/aja2",
                    "inventory_file": "/Users/sean/aja2/inventory",
                    "inventory_hostname": "aragorn",
                    "inventory_hostname_short": "aragorn",
...
                    "playbook_dir": "/Users/sean/aja2"
                },
                "bilbo": {
                    "ansible_check_mode": false,
                    "ansible_host": "198.51.100.5",
                    "ansible_playbook_python": "/usr/local/opt/python/bin/python2.7",
                    "ansible_python_interpreter": "/usr/local/bin/python",
                    "ansible_version": {
                        "full": "2.4.3.0",
                        "major": 2,
                        "minor": 4,
                        "revision": 3,
                        "string": "2.4.3.0"
                    },
                    "dns1": "8.8.4.4",
                    "dns2": "198.51.100.101",
                    "group_names": [
                        "ungrouped"
                    ],
                    "groups": {
                        "all": [
                            "aragorn",
                            "bilbo"
                        ],
                        "ungrouped": [
                            "aragorn",
                            "bilbo"
                        ]
                    },
                    "inventory_dir": "/Users/sean/aja2",
                    "inventory_file": "/Users/sean/aja2/inventory",
                    "inventory_hostname": "bilbo",
                    "inventory_hostname_short": "bilbo",
...
                    "playbook_dir": "/Users/sean/aja2"
                }
            },
            "inventory_dir": "/Users/sean/aja2",
            "inventory_file": "/Users/sean/aja2/inventory",
            "inventory_hostname": "aragorn",
            "inventory_hostname_short": "aragorn",
...
            "play_hosts": [
                "aragorn"
            ],
            "playbook_dir": "/Users/sean/aja2",
            "role_names": []
    }
```

```
}

PLAY RECAP **********************************************************************
aragorn                    : ok=2    changed=0    unreachable=0    failed=0
```

Spend a little time looking at all the variables. We have discussed some of them already, and some of the others are self-explanatory. Many of the more obscure variables deal with the environment in which the playbook is executing and are of little interest to most users. 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 devices other than the current device. For example, a task in your playbook could reference hostvars['aragorn']['ansible_host'] to access the ansible_host setting for the *aragorn* device, even if the device being processed was *bilbo* or another inventory host.

The author has found hostvars particularly useful in reading data known for *localhost* (the system executing the Ansible playbook) and using that data in a task related to a network device. For example, notice in the output above the ansible_date_time dictionary containing various current date and time data for the local host. This could be used to put a "date stamp" as part of a filename, so files saved by different runs of the same playbook could all be retained.

Another way of viewing the facts gathered from the local host is 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:aja2 sean$ ansible -m setup localhost
localhost | SUCCESS => {
    "ansible_facts": {
...
        "ansible_date_time": {
            "date": "2018-03-17",
            "day": "17",
            "epoch": "1521304127",
            "hour": "12",
            "iso8601": "2018-03-17T16:28:47Z",
            "iso8601_basic": "20180317T122847204752",
            "iso8601_basic_short": "20180317T122847",
            "iso8601_micro": "2018-03-17T16:28:47.204869Z",
            "minute": "28",
            "month": "03",
            "second": "47",
            "time": "12:28:47",
            "tz": "EDT",
            "tz_offset": "-0400",
            "weekday": "Saturday",
            "weekday_number": "6",
            "weeknumber": "11",
            "year": "2018"
        },
...
```

```
        "ansible_hostname": "mbp15",
...
        "ansible_pkg_mgr": "homebrew",
        "ansible_processor": "Intel(R) Core(TM) i7-4870HQ CPU @ 2.50GHz",
        "ansible_processor_cores": "4",
        "ansible_python": {
            "executable": "/usr/local/Cellar/python/2.7.14_2/Frameworks/Python.framework/
Versions/2.7/Resources/Python.app/Contents/MacOS/Python",
            "has_sslcontext": true,
            "type": "CPython",
            "version": {
                "major": 2,
                "micro": 14,
                "minor": 7,
                "releaselevel": "final",
                "serial": 0
            },
            "version_info": [
                2,
                7,
                14,
                "final",
                0
            ]
        },
        "ansible_python_version": "2.7.14",
...
        "gather_subset": [
            "all"
        ],
        "module_setup": true
    },
    "changed": false
}
```

The following playbook, `show-vars-4.yaml`, illustrates one approach to gathering localhost date and time information and using it later in the playbook:

```
 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.weekday}} {{ ansible_date_time.date }} at {{ ansible_
date_time.time }}"
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
```

The first play, lines 2–10, executes on localhost and saves the local date and time into a variable called `timestamp`. Take note of line 6, which causes this play to gather facts from the localhost. It is by gathering facts that Ansible learns the date and time; without gathering facts, the `ansible_date_time` variable would be undefined.

The second play, lines 12–20, executes for each device in inventory. Note on line 20 how this play references localhost's `timestamp` variable. The first play defines `timestamp` only for `localhost`, not for all devices, so other devices need to access the variable through the `hostvars` dictionary.

A sample playbook run:

```
mbp15:aja2 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: [aragorn] => {
    "hostvars.localhost.timestamp": "Saturday 2018-03-17 at 12:36:15"
}
ok: [bilbo] => {
    "hostvars.localhost.timestamp": "Saturday 2018-03-17 at 12:36:15"
}

PLAY RECAP *******************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=0
bilbo                      : ok=1    changed=0    unreachable=0    failed=0
localhost                  : ok=2    changed=0    unreachable=0    failed=0
```

CAUTION   When using `--limit` with playbooks that include tasks for `localhost` you must 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:aja2 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:aja2 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": "Saturday 2018-03-17 at 12:39:21"
}

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

Using variables in an Ansible playbook is a large topic. See the References section at the end of this chapter for links to additional information.

Host Data Files

Our `inventory` file currently looks something like this:

```
mbp15:aja2 sean$ cat inventory
aragorn    ansible_host=192.0.2.10     dns1=8.8.8.8    dns2=198.51.100.100
bilbo      ansible_host=198.51.100.5   dns1=8.8.4.4    dns2=198.51.100.101

[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, as earlier chapters have mentioned, the inventory file is not the preferred place for variables. Here, we discuss a better approach for storing host-specific data; later in this chapter we 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 `~/aja2/host_vars` on your system to contain your host data files.

Now create data files in the `host_vars` directory for your 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 device *aragorn*, the file `host_vars/aragorn.yaml` contains the following:

```
---
ansible_host: 192.0.2.10
dns1: 8.8.8.8
dns2: 198.51.100.100
```

For device *bilbo*, the file `host_vars/bilbo.yaml` contains the following:

```
---
ansible_host: 198.51.100.5
dns1: 8.8.4.4
dns2: 198.51.100.101
```

Remove the host variables from the file `inventory`, leaving the following:

```
aragorn
bilbo

[all:vars]
ansible_python_interpreter=/usr/local/bin/python
```

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` playbook 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 inventory is the ability to easily create dictionaries or lists in the host data, and thereby manage 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 variables. Let's change our files to use a list called `dns_servers` to store for DNS server IP addresses, instead of the `dns1` and `dns2` variables. We can also add a third DNS server address. The author's files become:

```
mbp15:aja2 sean$ cat host_vars/aragorn.yaml
---
ansible_host: 192.0.2.10
dns_servers:
  - 8.8.4.4
  - 8.8.8.8
  - 198.51.100.100

mbp15:aja2 sean$ cat host_vars/bilbo.yaml
---
ansible_host: 198.51.100.5
dns_servers:
  - 8.8.4.4
  - 8.8.8.8
  - 198.51.100.10
```

You can use the `show-vars-3.yaml` playbook to confirm the changes are seen by Ansible playbooks (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:

```
mbp15:aja2 sean$ ansible-playbook show-vars-3.yaml

PLAY [Show variables 3] ********************************************************

TASK [ansible variables] ******************************************************
ok: [aragorn] => {
    "vars": {
...
        "ansible_version": {
            "full": "2.4.3.0",
            "major": 2,
            "minor": 4,
            "revision": 3,
            "string": "2.4.3.0"
        },
        "dns_servers": [
            "8.8.4.4",
            "8.8.8.8",
            "198.51.100.100"
        ],
...
    }
}
ok: [bilbo] => {
    "vars": {
...
        "dns_servers": [
            "8.8.4.4",
            "8.8.8.8",
            "198.51.100.101"
        ],
...
    }
}

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

TIP   In the `show-vars-3.yaml` playbook, set `gather_facts: no`. This will dramatically reduce the amount of output relative to what we saw earlier in the chapter.

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 a number of version-related variables. A number of Ansible's variables are in dictionaries like this. We saw another example with playbook `show-vars-4.yaml`; the date and time data for localhost was in a dictionary called `ansible_date_time`.

The author likes to organize host data in his host_vars files into dictionaries, except 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 dictionaries helps document the purpose of the data. Different dictionaries can group related 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.

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 office. If the group's data file contains this:

```
---
dns_servers:
  - 1.2.3.1
  - 1.2.3.2
```

...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 referencing. Consider if the host data file contained this:

```
---
aja2_host:
  dns_servers:
    - 5.7.9.11
    - 5.7.9.12
    - 5.7.9.13
```

...and the group data file contained this:

```
---
aja2_office:
  dns_servers:
    - 1.2.3.1
    - 1.2.3.2
```

Referencing the respective name server lists in a playbook or template would require {{ aja2_host.dns_servers }} and {{ aja2_office.dns_servers }}. The different dictionary names document whether we are referencing host-specific or office-wide DNS server information, and the fact that both lists are called `dns_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 dictionaries – 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 settings" dictionary called `aja2_host`. The author's files now contain:

```
mbp15:aja2 sean$ cat host_vars/aragorn.yaml
---
ansible_host: 192.0.2.10
aja2_host:
  dns_servers:
    - 8.8.4.4
    - 8.8.8.8
```

```
    – 198.51.100.100

mbp15:aja2 sean$ cat host_vars/bilbo.yaml
---
ansible_host: 198.51.100.5
aja2_host:
  dns_servers:
    – 8.8.4.4
    – 8.8.8.8
    – 198.51.100.101
```

Later we will add more settings to the `aja2_host` dictionary.

Run playbook `show-vars-3.yaml` and confirm the results of our changes:

```
...
TASK [ansible variables] ********************************************
ok: [aragorn] => {
    "vars": {
        "aja2_host": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8",
                "198.51.100.100"
            ]
        },
...
    }
}
ok: [bilbo] => {
    "vars": {
        "aja2_host": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8",
                "198.51.100.101"
            ]
        },
...
    }
}
...
```

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 comment #}
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...vPzOaX3gt8Uv sean@mbp15.local";
10|            }
11|        }
12|    }
13|    replace:
14|    name-server {
15|      {% for server in aja2_host.dns_servers %}
16|        {{ server }};
17|      {% endfor %}
18|    }
19|    services {
20|        delete: ftp;
21|        netconf {
22|            ssh;
23|        }
24|        delete: telnet;
25|        delete: web-management;
26|    }
27|}
```

Line 1 shows a Jinja2 comment – the {# and #} are *delimiters* that identify the start and end of the comment text. Jinja2 ignores the {# comment #} when processing the template; the comment will not appear in the generated output. Comments can be used to describe or document the template, or to temporarily remove a portion of the template from processing during troubleshooting.

Lines 15–17 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 element of a list[1] and performs some action. Line 15 starts the for loop. The {% %} braces-and-percent-signs delimiters identify a command that Jinja2 needs to interpret, much the same way that {{ }} tells Jinja2 that the contents are the name of a variable to be referenced. The *for* keyword introduces the command, 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* introduces the name of the list whose elements we want to reference, in this case the aja2_host.dns_servers list we created above in the host_vars files.

Line 17 denotes the end of the for loop; note the use of the {% and %} delimiters around the endfor command.

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.

---

1 Programmers whose background is 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.

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 `aja2_host.dns_servers`, a list containing three elements. Jinja2 puts the first element, `8.8.4.4`, 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, `8.8.8.8`, 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, `198.51.100.101`. When Jinja2 returns to line 15 again, it finds it has read all the elements of the list. This causes Jinja2 to complete the for loop and move to the line after the end of the loop, line 18, and continue processing the template from there.

Let's use the `base-settings.yaml` playbook to generate the configuration files, thereby testing our updated template. The author modified his playbook as described in the *Debugging Templates* section of Chapter 7 so the playbook generates the configuration files and stops. The results for *aragorn* are:

```
mbp15:aja2 sean$ cat tmp/aragorn.conf
system {
    host-name aragorn;
    login {
        user sean {
            uid 2000;
            class super-user;
            authentication {
                ssh-rsa "ssh-rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
            }
        }
    }
    replace:
    name-server {
            8.8.4.4;
            8.8.8.8;
            198.51.100.100;
        }
    services {
        delete: ftp;
        netconf {
            ssh;
        }
        delete: telnet;
        delete: web-management;
    }
}
```

Looks good overall. The comment from line 1 of the template is not in the generated file, and we have our three DNS server IPs in the name-server hierarchy. However, the indentation of the name-server list is a bit off – specifically, the three IP addresses and the closing brace are all indented too far.

The extra indentation in the output is from the indentation of lines 15 and 17, the beginning and end of the for loop. Each cycle through the for loop includes the six spaces in front of one of these lines with commands on them. Jinja2 automatically suppresses the newline at the end of the commands, so we do not see "blank" lines in our output; instead, we see extra spaces in front of the output from line 16 and 18.

Junos won't care about the unusual indentation; we could upload the configuration files as they are and Junos would happily accept them. However, consistent indentation helps we humans understand the configuration files.

We can instruct Jinja2 to suppress the leading whitespace (space and tab characters) before a command by placing this directive at the top of the template file:
`#jinja2: lstrip_blocks: True`

Technically, this directive sets Jinja2's `lstrip_blocks` option to `True`, with the result that "leading spaces and tabs are stripped from the start of a line to a block," according to the Jinja2 documentation [http://jinja.pocoo.org/docs/2.10/api/]. (The commands in `{% %}` delimiters are also called *blocks* in Jinja2.) We did not need this directive in the original version of the template because it contained only plain text and variable references, but the new version includes the `{% for ... %}` loop.

Let's replace the comment on line 1 with that directive:

```
1|#jinja2: lstrip_blocks: True
2|system {
3|    host-name {{ inventory_hostname }};
...
```

Run the `base-settings.yaml` playbook again and check the generated configuration. This time the indentation should look more like Junos:

```
...
    }
    replace:
    name-server {
        8.8.4.4;
        8.8.8.8;
        198.51.100.100;
    }
    services {
...
```

Speaking of running the playbook, let's update our `base-settings.yaml` playbook to use SSH key authentication (in other words, remove the username and password references) and undo any changes made for template debugging. Delete the `vars_prompt` section and delete the `user`, `passwd` and `port` arguments from the `connection_settings` dictionary. Undo any other changes related to template debugging, such as deleting the fail task if you added one.

The result should look like this:

```
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|    connection_settings:
14|      host: "{{ ansible_host }}"
15|      timeout: 120
16|
17|  tasks:
18|    – name: confirm or create configs directory
19|      file:
20|        path: "{{ tmp_dir }}"
21|        state: directory
22|      run_once: yes
23|
24|    – name: save device information using template
25|      template:
26|        src: template/base–settings.j2
27|        dest: "{{ conf_file }}"
28|
29|    – name: install generated configuration file onto device
30|      juniper_junos_config:
31|        provider: "{{ connection_settings }}"
32|        src: "{{ conf_file }}"
33|        load: replace
34|        comment: "playbook base–settings.yaml, commit confirmed"
35|        confirmed: 5
36|        diff: yes
37|        ignore_warning: yes
38|      register: config_results
39|      notify: confirm previous commit
40|
41|    – name: show configuration change
42|      debug:
43|        var: config_results.diff_lines
44|      when: config_results.diff_lines is defined
45|
46|    – name: delete generated configuration file
47|      file:
48|        path: "{{ conf_file }}"
49|        state: absent
50|
51|  handlers:
52|    – name: confirm previous commit
53|      juniper_junos_config:
54|        provider: "{{ connection_settings }}"
55|        comment: "playbook base–settings.yaml, confirming previous commit"
56|        commit: yes
57|        diff: no
```

Run the `base-settings.yaml` playbook; it should update the name servers on your test devices. We did not change anything else in the template, so no other changes should be needed on the devices:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml

PLAY [Generate and Install Configuration File] *********************************

TASK [confirm or create configs directory] ************************************
ok: [aragorn]

TASK [save device information using template] *********************************
ok: [bilbo]
ok: [aragorn]

TASK [install generated configuration file onto device] ***********************
changed: [aragorn]
changed: [bilbo]

TASK [show configuration change] **********************************************
ok: [aragorn] => {
    "config_results.diff_lines": [
        "",
        "[edit system name-server]",
        "+    8.8.4.4;",
        "    8.8.8.8 { ... }"
    ]
}
ok: [bilbo] => {
    "config_results.diff_lines": [
        "",
        "[edit system name-server]",
        "    8.8.4.4 { ... }",
        "+    8.8.8.8;",
        "    198.51.100.101 { ... }"
    ]
}

TASK [delete generated configuration file] ************************************
changed: [aragorn]
changed: [bilbo]

RUNNING HANDLER [confirm previous commit] *************************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP ********************************************************************
aragorn                    : ok=6    changed=2    unreachable=0    failed=0
bilbo                      : ok=5    changed=2    unreachable=0    failed=0
```

Notice that our device configuration changes were implemented through a template change, not a playbook change. While we modified the playbook for our convenience, or for template debugging, the modifications did not include changes to the tasks "save device information using template" or "install generated configuration file onto device" or to the "confirm previous commit" handler.

More Device-Specific Data and Escape Characters

Let's add SNMP location and description information to our host variables. These settings are specific to each device.

The author's file `host_vars/aragorn.yaml` now includes:

```
---
ansible_host: 192.0.2.10
aja2_host:
  dns_servers:
    - 8.8.4.4
    - 8.8.8.8
    - 198.51.100.100
  snmp:
    description: virtual SRX for testing
    location: Sean's Macbook Pro
```

The author's file `host_vars/bilbo.yaml` now includes:

```
---
ansible_host: 198.51.100.5
aja2_host:
  dns_servers:
    - 8.8.4.4
    - 8.8.8.8
    - 198.51.100.101
  snmp:
    description: EX2200-C for testing
    location: Sean's home office
```

Notice how the update to each file creates a new dictionary `snmp` within the `aja2_host` dictionary, containing `description` and `location` keys with appropriate values. We can easily add additional SNMP-related data to the `snmp` dictionary, such as a contact or a client-list, and the fact that the new data is part of the `snmp` dictionary provides context to that data.

TIP   Using dictionaries like this can help to make your data self-documenting. This will benefit you and your team when trying to understand each other's work, or when revisiting your own work months or years later.

We need to update `template/base-settings.j2` to include the new SNMP settings in the generated configuration file. Append the following lines to the end of the template (only new lines shown, with line numbers added):

```
...
28|snmp {
29|    description "{{ aja2_host.snmp.description}}"
30|    location "{{ aja2_host.snmp.location}}"
31|}
```

Observe how the template additions reference the `snmp.description` and `snmp.location` variables in the `aja2_host` dictionary. Also observe the quotes around the new variable references in the template (lines 29 and 30). 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:

```
{master:0}
sean@bilbo> configure
Entering configuration mode

{master:0}[edit]
sean@bilbo# set snmp location Sean's office
                                ^
syntax error.

{master:0}[edit]
sean@bilbo# set snmp location "Sean's office"

{master:0}[edit]
sean@bilbo# show snmp
location "Sean's office";
```

Run the base-settings.yaml playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml

PLAY [Generate and Install Configuration File] **********************************

TASK [confirm or create configs directory] **********************************
ok: [aragorn]

TASK [save device information using template] **********************************
changed: [aragorn]
changed: [bilbo]

TASK [install generated configuration file onto device] **********************************
changed: [aragorn]
changed: [bilbo]

TASK [show configuration change] **********************************
ok: [aragorn] => {
    "config_results.diff_lines": [
        "",
        "[edit]",
        "+  snmp {",
        "+      description \"virtual SRX for testing\";",
        "+      location \"Sean's Macbook Pro\";",
        "+  }"
    ]
}
ok: [bilbo] => {
    "config_results.diff_lines": [
        "",
        "[edit]",
        "+  snmp {",
        "+      description \"EX2200-C for testing\";",
        "+      location \"Sean's home office\";",
        "+  }"
```

```
    ]
}

TASK [delete generated configuration file] **************************************
changed: [aragorn]
changed: [bilbo]

RUNNING HANDLER [confirm previous commit] **************************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *********************************************************************
aragorn                    : ok=6    changed=3    unreachable=0    failed=0
bilbo                      : ok=5    changed=3    unreachable=0    failed=0
```

Look closely at the output of the "show configuration change" task. Notice that several of the changed lines seem to contain backslashes (\ ) that we did not put in our template, for example:

```
"+       location \"Sean's home office\";"
```

Log onto one of your test devices and check the change that was made:

```
sean@aragorn> show system commit
0    2018-03-19 19:16:35 UTC by sean via netconf
     playbook base-settings.yaml, confirming previous commit
1    2018-03-19 19:16:02 UTC by sean via netconf commit confirmed, rollback in 5mins
     playbook base-settings.yaml, commit confirmed
...

sean@aragorn> show configuration | compare rollback 2
[edit]
+ snmp {
+     description "virtual SRX for testing";
+     location "Sean's Macbook Pro";
+ }
```

No extraneous backslashes here. So why are they in the playbook results?

The change made by our playbook includes double-quotes (" "). Ansible, when displaying the list of changed lines, puts each line in double-quotes. Because the change itself already has double-quotes, Ansible sees a problem – it needs to somehow differentiate between the opening and closing quotes that surround the entire line, and the similar quotes that are in the middle of the changed line. If Ansible cannot distinguish the two sets of quotes, it may misinterpret the results, as might another another program that gets its data from Ansible.

For example, the first double-quote starts a string, the next double-quote ends the string, there is some other text that is not quoted and thus should have meaning but does not, followed by another string between double-quotes. In other words, the following interpretation, where ^ marks the strings delimited by the double-quotes and ! marks the "unknown stuff" between the two double-quoted strings:

```
"+      location "Sean's home office";"
 ^^^^^^^^^^^^^^^ !!!!!!!!!!!!!!!!!! ^
```

To avoid this problem, Ansible *escapes* the double-quotes in the middle of the string by inserting a backslash before those double-quotes. *Escaping* is a programming concept that essentially means adding to a string a special character, called an *escape character*, that modifies the meaning of the next character in the string. Ansible (and Python and many other programming languages) use the backslash character as an escape character.

With the escape character (backslash) altering the meaning of the double-quotes in the middle of the string – removing their meaning as string delimiters and treating them as regular characters – Ansible can correctly interpret the string:

```
"+      location \"Sean's home office\";"
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Unless you are a programmer you do not need to fully understand the concept of escaping in strings. Just keep the general idea in mind when you see strange backslashes appearing in Ansible's output.

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:aja2 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 by using the `ansible-playbook` command-line options `-i` or `--inventory-file`. For example, you may create separate inventory files for test and production environments, and then 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` or `-i` command-line options overrides the `inventory` setting in `ansible.cfg`, so you can use `ansible.cfg` to set as a default the inventory file you need most often, then tell Ansible to use an alternate inventory when needed.

Inventory Directory

Another option is to place one or more inventory files in a directory, and tell Ansible to use the directory, whether via the `-i/--inventory-file` command-line options 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 example, 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.

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, San Francisco and Boston, and we want to maintain separate inventory files for each office.

Create a new directory `inventory2` in your `~/aja2` directory. Within `~/aja2/inventory2` create files `all_vars`, `boston` and `san_francisco`.

The `all_vars` file will hold variables 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):

```
[all:vars]
ansible_python_interpreter=/usr/local/bin/python
```

The file `inventory2/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
```

The file `inventory2/san_francisco` should contain the following (again, substitute one of your devices for *aragorn* and ensure you have at least three names even if you do not have matching devices):

```
aragorn
eowyn
faramir
```

NOTE   Do not worry that these new devices, and a few more we will add in the next section of this chapter, do not exist. We will use these inventory entries only to illustrate some inventory concepts. You need not create host data files in the `host_vars` directory for the nonexistent devices.

Update your `ansible.cfg` file to use the new `inventory2` directory:

```
[defaults]
inventory = inventory2
host_key_checking = False
```

Let's confirm that Ansible sees our updated inventory. We can ask Ansible to tell us what hosts it would act upon, without having it do anything, using --list-hosts option to the ansible or ansible-playbook commands:

```
mbp15:aja2 sean$ ansible all --list-hosts
  hosts (6):
    bilbo
    frodo
    sam
    aragorn
    eowyn
    faramir
```

We can also use our show-vars-3.yaml playbook to check the membership of Ansible's all group, which includes all devices in inventory, and to confirm that our ansible_python_interpreter variable is still set correctly:

```
mbp15:aja2 sean$ ansible-playbook show-vars-3.yaml --limit=aragorn

PLAY [Show variables 3] *********************************************************

TASK [ansible variables] ********************************************************
ok: [aragorn] => {
    "vars": {
        "aja2_host": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8",
                "198.51.100.100"
            ],
            "snmp": {
                "description": "virtual SRX for testing",
                "location": "Sean's Macbook Pro"
            }
        },
...
        "ansible_python_interpreter": "/usr/local/bin/python",
...
        "groups": {
            "all": [
                "bilbo",
                "frodo",
                "sam",
                "aragorn",
                "eowyn",
                "faramir"
            ],
            "ungrouped": [
                "bilbo",
                "frodo",
                "sam",
                "aragorn",
                "eowyn",
                "faramir"
```

```
        ]
    },
...
    }
}

PLAY RECAP *****************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=0
```

You can see the groups dictionary, including the automatically-created groups `all` and `ungrouped` and their members. Notice that separating the inventory into multiple files does *not* automatically create a group for each file; we will create inventory groups in the next section of this chapter.

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 organizational 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 members 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]
```

After the group heading, list the inventory hostnames for the devices which are members of the group, one per line:

```
[groupname]
device1
device2
```

If a group's members are other groups, the group's heading must include the modifier `:children` after the group name:

```
[groupname:children]
group1
group2
```

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:

```
[boston]
bilbo
frodo
sam
```

And inventory file `san_francisco` becomes (abbreviating the city name to "sf" for the group name):

```
[sf]
gimli
gloin
aragorn
```

We can use `show-vars-3.yaml` to confirm the groups were created, but let's create a variation of that playbook that shows just the groups, not all the other variables. Create playbook `show-groups.yaml`:

```
 1|---
 2|- name: Show groups
 3|  hosts:
 4|    - localhost
 5|  connection: local
 6|  gather_facts: no
 7|
 8|  tasks:
 9|    - name: ansible variables
10|      debug:
11|        var: vars.groups
```

Run the show-groups.yaml playbook to confirm our new groups exist:

```
mbp15:aja2 sean$ ansible-playbook show-groups.yaml

PLAY [Show groups] ************************************************

TASK [ansible variables] ******************************************
ok: [localhost] => {
    "vars.groups": {
        "all": [
            "bilbo",
            "frodo",
            "sam",
            "gimli",
            "gloin",
            "aragorn"
        ],
        "boston": [
            "bilbo",
            "frodo",
            "sam"
        ],
        "sf": [
            "gimli",
            "gloin",
            "aragorn"
        ],
        "ungrouped": []
    }
}

PLAY RECAP ********************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0
```

You can also confirm group membership with `--list-hosts`; for example:

```
mbp15:aja2 sean$ ansible boston --list-hosts
  hosts (3):
```

```
    bilbo
    frodo
    sam

mbp15:aja2 sean$ ansible sf --list-hosts
  hosts (3):
    aragorn
    eowyn
    faramir
```

Assume our Boston and San Francisco offices each have EX switches and SRX firewalls. 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 some more non-existent devices) for the switches and firewalls in each office.

The boston inventory file now contains:

```
[boston:children]
bos_ex
bos_srx

[bos_ex]
bilbo
frodo
sam

[bos_srx]
arwen
```

The san_francisco inventory file now contains:

```
[sf:children]
sf_ex
sf_srx

[sf_ex]
gimli
gloin

[sf_srx]
galadriel
aragorn
```

Confirm the new groups using either or both of the approaches shown above:

```
mbp15:aja2 sean$ ansible-playbook show-groups.yaml

PLAY [Show groups] ***********************************************

TASK [ansible variables] ****************************************
ok: [localhost] => {
    "vars.groups": {
        "all": [
            "bilbo",
            "frodo",
            "peregrin",
            "sam",
            "eowyn",
            "faramir",
```

```
            "aragorn",
            "arwen"
        ],
        "bos_ex": [
            "bilbo",
            "frodo"
        ],
        "bos_srx": [
            "peregrin",
            "sam"
        ],
        "boston": [
            "bilbo",
            "frodo",
            "peregrin",
            "sam"
        ],
        "sf": [
            "eowyn",
            "faramir",
            "aragorn",
            "arwen"
        ],
        "sf_ex": [
            "eowyn",
            "faramir"
        ],
        "sf_srx": [
            "aragorn",
            "arwen"
        ],
        "ungrouped": []
    }
}

PLAY RECAP ********************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0

mbp15:aja2 sean$ ansible sf_srx --list-hosts
  hosts (2):
    aragorn
    arwen
```

Now assume we need to run a playbook against all our EX devices. We can run the playbook with `--limit=bos_ex,sf_ex`, but as the number of sites increases, ensuring we list all the correct groups 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 inventory files, but as we will see momentarily, Ansible assembles the group definitions across both files 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

[bos_srx]
peregrin
sam
```

The san_francisco inventory file now contains:

```
[sf:children]
sf_ex
sf_srx

[ex:children]
sf_ex

[srx:children]
sf_srx

[sf_ex]
eowyn
faramir

[sf_srx]
aragorn
arwen
```

Again, confirm the inventory updates using the approaches we have discussed. Notice that the ex and srx groups contain devices from both the boston and san_ francisco inventory files.

```
mbp15:aja2 sean$ ansible srx --list-hosts
  hosts (4):
    peregrin
    sam
    aragorn
    Arwen

mbp15:aja2 sean$ ansible ex --list-hosts
  hosts (4):
    bilbo
    frodo
    eowyn
    faramir

mbp15:aja2 sean$ ansible-playbook show-groups.yaml

PLAY [Show groups] ***********************************************

TASK [ansible variables] ****************************************
ok: [localhost] => {
    "vars.groups": {
        "all": [
            "bilbo",
```

```
            "frodo",
            "peregrin",
            "sam",
            "eowyn",
            "faramir",
            "aragorn",
            "arwen"
        ],
        "bos_ex": [
            "bilbo",
            "frodo"
        ],
        "bos_srx": [
            "peregrin",
            "sam"
        ],
        "boston": [
            "bilbo",
            "frodo",
            "peregrin",
            "sam"
        ],
        "ex": [
            "bilbo",
            "frodo",
            "eowyn",
            "faramir"
        ],
        "sf": [
            "eowyn",
            "faramir",
            "aragorn",
            "arwen"
        ],
        "sf_ex": [
            "eowyn",
            "faramir"
        ],
        "sf_srx": [
            "aragorn",
            "arwen"
        ],
        "srx": [
            "peregrin",
            "sam",
            "aragorn",
            "arwen"
        ],
        "ungrouped": []
    }
}

PLAY RECAP *********************************************************************
localhost                   : ok=1    changed=0    unreachable=0    failed=0
```

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

conducting 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 phases, each phase affecting different devices, and there are play-books written to carry out each stage. Create a new inventory file `maintenance` in our `~/aja2/inventory2` directory with the following:

```
[phase1]
arwen
bilbo
sam

[phase2]
bilbo
eowyn
faramir
```

Confirm Ansible's understanding of the group memberships:

```
mbp15:aja2 sean$ ansible phase1 --list-hosts
  hosts (3):
    arwen
    bilbo
    sam

mbp15:aja2 sean$ ansible phase2 --list-hosts
  hosts (3):
    bilbo
    eowyn
    faramir
```

If playbooks `phase1.yaml` and `phase2.yaml` existed, we could then run those playbooks 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 and remove all device instances when you need to remove the device from inventory, such as when you retire it.

Ansible's group_names Variable

We have seen that Ansible's `groups` variable contains a dictionary showing all inventory 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, we 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, we could execute a firewall-specific task only when a device is a member of the `srx` group.

Enter the following playbook, `show-group-names.yaml`:

```
1|---
2|- name: Show group names
3|  hosts:
4|    - all
5|  connection: local
6|  gather_facts: no
7|
8|  tasks:
9|    - name: group names
10|      debug:
11|        var: group_names
```

Run the playbook and observe that each device's `group_names` variable lists the user-defined groups of which the device is a member:

```
mbp15:aja2 sean$ ansible-playbook show-group-names.yaml

PLAY [Show group names] *********************************************

TASK [group names] **************************************************
ok: [arwen] => {
    "group_names": [
        "phase1",
        "sf",
        "sf_srx",
        "srx"
    ]
}
ok: [bilbo] => {
    "group_names": [
        "bos_ex",
        "boston",
        "ex",
        "phase1",
        "phase2"
    ]
}
ok: [sam] => {
    "group_names": [
        "bos_srx",
        "boston",
        "phase1",
        "srx"
    ]
}
ok: [eowyn] => {
    "group_names": [
        "ex",
        "phase2",
        "sf",
        "sf_ex"
    ]
}
ok: [faramir] => {
    "group_names": [
        "ex",
```

```
        "phase2",
        "sf",
        "sf_ex"
    ]
}
ok: [frodo] => {
    "group_names": [
        "bos_ex",
        "boston",
        "ex"
    ]
}
ok: [peregrin] => {
    "group_names": [
        "bos_srx",
        "boston",
        "srx"
    ]
}
ok: [aragorn] => {
    "group_names": [
        "sf",
        "sf_srx",
        "srx"
    ]
}

PLAY RECAP *********************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=0
arwen                      : ok=1    changed=0    unreachable=0    failed=0
bilbo                      : ok=1    changed=0    unreachable=0    failed=0
eowyn                      : ok=1    changed=0    unreachable=0    failed=0
faramir                    : ok=1    changed=0    unreachable=0    failed=0
frodo                      : ok=1    changed=0    unreachable=0    failed=0
peregrin                   : ok=1    changed=0    unreachable=0    failed=0
sam                        : ok=1    changed=0    unreachable=0    failed=0
```

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, we will continue the assumption that our devices represent two offices, Boston and San Francisco. We will also continue the assumption that we want groups for different device types; the author has EX and SRX test devices, but if you have other device types, feel free to create appropriately named groups.

Create a new inventory file, `~/aja2/inventory3`, with the following contents (adjust as needed for your device types and hostnames, but ensure you have at least one

device 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]
aragorn
```

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 a similar set of "*site_type*" groups (like bos_ex) whether or not each 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_interpreter 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:

```
[defaults]
inventory = inventory3
host_key_checking = False
log_path = ~/aja2/ansible.log
```

Run the show-groups.yaml and show-group-names.yaml playbooks to confirm the inventory and groups are what you expect:

```
mbp15:aja2 sean$ ansible-playbook show-groups.yaml

PLAY [Show groups] ************************************************************
```

```
TASK [ansible va\riables] ********************************************
ok: [localhost] => {
    "vars.groups": {
        "all": [
            "bilbo",
            "aragorn"
        ],
        "bos_ex": [
            "bilbo"
        ],
        "bos_srx": [],
        "boston": [
            "bilbo"
        ],
        "ex": [
            "bilbo"
        ],
        "sf": [
            "aragorn"
        ],
        "sf_ex": [],
        "sf_srx": [
            "aragorn"
        ],
        "srx": [
            "aragorn"
        ],
        "ungrouped": []
    }
}

PLAY RECAP **********************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0

mbp15:aja2 sean$ ansible-playbook show-group-names.yaml

PLAY [Show group names] *********************************************

TASK [group names] *************************************************
ok: [bilbo] => {
    "group_names": [
        "bos_ex",
        "boston",
        "ex"
    ]
}
ok: [aragorn] => {
    "group_names": [
        "sf",
        "sf_srx",
        "srx"
    ]
}

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

Group Data Files

One benefit of defining inventory groups is that we can associate variables with groups. This can be useful if, for example, all devices in a given office use particular NTP servers, but devices in another office 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 `~/aja2/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 devices, because all devices are members of the `all` group. This is a good place to put the `ansible_python_interpreter` variable, if it is needed for your system, keeping in mind this is a YAML file and so we use key:value variable definitions:

```
---
ansible_python_interpreter: /usr/local/bin/python
```

For Boston and San Francisco, we define a list of NTP servers to be used at each site. As discussed earlier for host variables, the author likes to create a dictionary to help make variable names self-documenting and help avoid name collisions.

The file `group_vars/boston.yaml` contains NTP server addresses from time.nist.gov (feel free to substitute different servers if appropriate in your environment):

```
---
aja2_site:
  ntp_servers:
    - 132.163.97.4
    - 129.6.15.27
```

The file `group_vars/sf.yaml` contains NTP server addresses from time.apple.com (feel free to substitute different servers if appropriate in your environment):

```
---
aja2_site:
  ntp_servers:
    - 17.253.6.125
    - 17.253.20.125
```

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": {
        "aja2_host": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8",
```

```
                    "198.51.100.101"
                ],
                "snmp": {
                    "description": "EX2200-C for testing",
                    "location": "Sean's home office"
                }
            },
            "aja2_site": {
                "ntp_servers": [
                    "132.163.97.4",
                    "129.6.15.27"
                ]
            },
...
    }
}
ok: [aragorn] => {
    "vars": {
        "aja2_host": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8",
                "198.51.100.100"
            ],
            "snmp": {
                "description": "virtual SRX for testing",
                "location": "Sean's Macbook Pro"
            }
        },
        "aja2_site": {
            "ntp_servers": [
                "17.253.6.125",
                "17.253.20.125"
            ]
        },
...
```

Observe that each host has the `aja2_site.ntp_servers` list appropriate for its location.

Update the `base-settings.j2` template to include the NTP servers as shown below. Lines 27–32 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
AAAAB3NzaC1yc2EAAAADAQABAAABAQCxgT8ga1uYbS3bxXPPv7aEiTvSwXnK/7xu3NB0+t1njMBuUcgwn7zwtnayQyLS+ef3rNP7
WZXwFYxUeFbVwdkLUn9/xvDM5Qi2m/6WRP/yrTRtEvNP4lUsZRH+IXQc59JOKfYqGkvbgfshnmtHJHYVOn/1E/w0cNDYg4oH6K-
bcqYb+isbKhdiqpDBvLsF9h0GwhaiLk2BpVutw2BZoeKN9vrF+OmcaB0WVzGvwblSHDpXdLfMJuHAyEhZImNSv4bXNAYFGht9zpd
TwudP5qfwJo5304Sn62UaOzVN2zGogXKzxgxgAjeJ87io0Graiwo5q9kZYksjXvPzOaX3gt8Uv sean@mbp15.local";
```

```
10|              }
11|          }
12|      }
13|      replace:
14|      name-server {
15|        {% for server in aja2_host.dns_servers %}
16|          {{ server }};
17|        {% endfor %}
18|      }
19|      services {
20|          delete: ftp;
21|          netconf {
22|              ssh;
23|          }
24|          delete: telnet;
25|          delete: web-management;
26|      }
27|      replace:
28|      ntp {
29|        {% for ntp in aja2_site.ntp_servers %}
30|          server {{ ntp }};
31|        {% endfor %}
32|      }
33|}
34|snmp {
35|    description "{{ aja2_host.snmp.description}}"
36|    location "{{ aja2_host.snmp.location}}"
37|}
```

Run the `base-settings.yaml` playbook to update your devices' configurations:

```
mbp15:aja2 sean$ ansible-playbook base-settings-2.yaml

PLAY [Generate and Install Configuration File] **********************************

TASK [confirm or create configs directory] *************************************
ok: [bilbo]

TASK [save device information using template] **********************************
changed: [bilbo]
changed: [aragorn]

TASK [install generated configuration file onto device] ************************
changed: [aragorn]
changed: [bilbo]

TASK [show configuration change] ***********************************************
ok: [bilbo] => {
    "config_results.diff_lines": [
        "",
        "[edit system]",
        "+    ntp {",
        "+        server 132.163.97.4;",
        "+        server 129.6.15.27;",
        "+    }"
    ]
}
ok: [aragorn] => {
```

```
    "config_results.diff_lines": [
        "",
        "[edit system ntp]",
        "-   boot-server 17.253.20.253;",
        "[edit system ntp]",
        "+    server 17.253.6.125;",
        "+    server 17.253.20.125;",
        "-    server 17.253.20.253;",
        "-    server 129.6.15.30;"
    ]
}

TASK [delete generated configuration file] ***************************************
changed: [bilbo]
changed: [aragorn]

RUNNING HANDLER [confirm previous commit] ***************************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *********************************************************************
aragorn                    : ok=5    changed=3    unreachable=0    failed=0
bilbo                      : ok=6    changed=3    unreachable=0    failed=0
```

Observe that each device gets the appropriate site-specific NTP servers.

Alternate Inventory Directory Layout

The layout of files and directories described so far in this chapter is but one way of organizing your inventory, host data, and group data files. This layout is the first one presented on Ansible's best practices page (link in References section) and is perhaps the easiest layout to use when learning Ansible. To summarize the inventory and data file and directory layout we have discussed so far:

```
playbook_directory
├── group_vars
│       ├── group1.yaml
│       └── group2.yaml
├── host_vars
│       ├── host1.yaml
│       └── host2.yaml
├── inventory1_directory
│       ├── inventory1a
│       └── inventory1b
└── inventory2_file
```

This section of this chapter presents an alternative layout, also shown on Ansible's best practices page. After we illustrate the alternate layout, we suggest when each option might be the most appropriate.

The major change in the alternative layout is that you create a directory for each inventory, and the host_vars and group_vars directories for each inventory are placed *within the inventory directory*. The inventory itself can be a single file or

several files in another subdirectory of the inventory directory. Ansible's best practices page suggests calling the inventory file or directory *hosts*, but in the author's tests this name does not seem to be necessary for Ansible to find the inventory.

Ansible also suggests creating a directory called *inventories* within the playbook directory and placing your inventory directories within the `inventories` directory. This is not a requirement, but it provides a single, meaningfully named location for all inventory data, regardless of how many inventories you may need.

Please create your `inventories` directory now:

```
mbp15:aja2 sean$ mkdir inventories
```

Assume we need to manage some servers in addition to our Junos devices. The servers should have their own *servers* inventory. Create the following directory hierarchy and files within the new `inventories` directory:

```
~/aja2/inventories
└── servers
    ├── group_vars
    │   ├── all.yaml
    │   ├── database.yaml
    │   └── web.yaml
    ├── host_vars
    │   ├── gandalf.yaml
    │   └── saruman.yaml
    └── hosts
```

Populate the files as follows:

```
inventories/servers/hosts:
[database]
gandalf

[web]
Saruman


inventories/servers/host_vars/gandalf.yaml:
---
aja2_host:
  oob_ip: 192.0.2.20


inventories/servers/host_vars/saruman.yaml:
---
aja2_host:
  oob_ip: 192.0.2.21


inventories/servers/group_vars/all.yaml:
---
aja2_servers:
  dns_servers:
```

```
  - 8.8.4.4
  - 8.8.8.8


inventories/servers/group_vars/database.yaml:
---
db_servers:
  db_port: 3306


inventories/servers/group_vars/web.yaml:
---
web_servers:
  http_dir: /srv/http/
```

Now use the show-groups.yaml and show-vars-3.yaml playbooks to confirm the new inventory is doing what we expect. Remember to specify the inventory on the command line as we have not changed the default inventory in ansible.cfg.

```
mbp15:aja2 sean$ ansible-playbook show-groups.yaml -i inventories/servers/

PLAY [Show groups] ****************************************************************************
****

TASK [ansible variables] *********************************************************************
****
ok: [localhost] => {
    "vars.groups": {
        "all": [
            "saruman",
            "gandalf"
        ],
        "database": [
            "gandalf"
        ],
        "ungrouped": [],
        "web": [
            "saruman"
        ]
    }
}

PLAY RECAP ************************************************************************************
*****
localhost                  : ok=1    changed=0    unreachable=0    failed=0

mbp15:aja2 sean$ ansible-playbook show-vars-3.yaml -i inventories/servers/ --limit=gandalf

PLAY [Show variables 3] **********************************************************************
****

TASK [ansible variables] *********************************************************************
****
ok: [gandalf] => {
    "vars": {
        "aja2_host": {
```

```
            "oob_ip": "192.0.2.20"
        },
        "aja2_servers": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8"
            ]
        },
...
        "db_servers": {
            "db_port": 3306
        },
...
        "hostvars": {
...
            "saruman": {
                "aja2_host": {
                    "oob_ip": "192.0.2.21"
                },
                "aja2_servers": {
                    "dns_servers": [
                        "8.8.4.4",
                        "8.8.8.8"
                    ]
                },
...
                "web_servers": {
                    "http_dir": "/srv/http/"
                }
            }
        },
...
    }
}

PLAY RECAP ***********************************************************************************************
*****
gandalf                    : ok=1    changed=0    unreachable=0    failed=0
```

Let's add an inventory of Junos devices to our inventories directory, using this alternative directory layout. Let's populate the new *junos* inventory by copying the inventory files from the inventory2 directory we created earlier in this chapter, and copying our host_vars and group_vars files from earlier in this chapter:

```
mbp15:aja2 sean$ mkdir inventories/junos
mbp15:aja2 sean$ mkdir inventories/junos/hosts
mbp15:aja2 sean$ mkdir inventories/junos/host_vars
mbp15:aja2 sean$ mkdir inventories/junos/group_vars

mbp15:aja2 sean$ cp inventory2/* inventories/junos/hosts/
mbp15:aja2 sean$ cp group_vars/* inventories/junos/group_vars/
mbp15:aja2 sean$ cp host_vars/* inventories/junos/host_vars/
```

That should yield the following hierarchy:

```
mbp15:aja2 sean$ tree inventories/
inventories/
```

```
├── junos
│   ├── group_vars
│   │   ├── all.yaml
│   │   ├── boston.yaml
│   │   └── sf.yaml
│   ├── host_vars
│   │   ├── aragorn.yaml
│   │   └── bilbo.yaml
│   └── hosts
│       ├── all_vars
│       ├── boston
│       ├── maintenance
│       └── san_francisco
└── servers
    ├── group_vars
    │   ├── all.yaml
    │   ├── database.yaml
    │   └── web.yaml
    ├── host_vars
    │   ├── gandalf.yaml
    │   └── saruman.yaml
    └── hosts
```
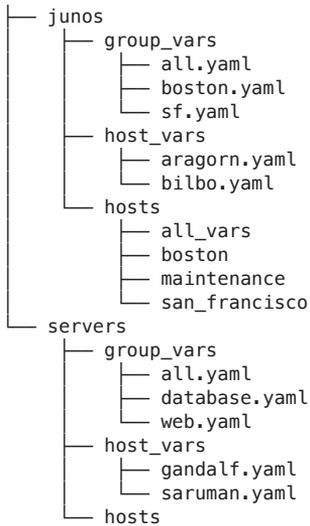
Run the show-vars-3.yaml playbooks, specifying the new *junos* inventory on the
command line, to confirm the new inventory is doing what we expect:

```
mbp15:aja2 sean$ ansible-playbook show-groups.yaml -i inventories/junos/

PLAY [Show groups] *************************************************************************
****

TASK [ansible variables] ******************************************************************
****
ok: [localhost] => {
    "vars.groups": {
        "all": [
            "arwen",
            "bilbo",
            "sam",
            "eowyn",
            "faramir",
            "frodo",
            "peregrin",
            "aragorn"
        ],
        "bos_ex": [
            "bilbo",
            "frodo"
        ],
        "bos_srx": [
            "peregrin",
            "sam"
        ],
        "boston": [
            "bilbo",
            "frodo",
            "peregrin",
            "sam"
```

```
            ],
            "ex": [
                "bilbo",
                "frodo",
                "eowyn",
                "faramir"
            ],
            "phase1": [
                "arwen",
                "bilbo",
                "sam"
            ],
            "phase2": [
                "bilbo",
                "eowyn",
                "faramir"
            ],
            "sf": [
                "eowyn",
                "faramir",
                "aragorn",
                "arwen"
            ],
            "sf_ex": [
                "eowyn",
                "faramir"
            ],
            "sf_srx": [
                "aragorn",
                "arwen"
            ],
            "srx": [
                "peregrin",
                "sam",
                "aragorn",
                "arwen"
            ],
            "ungrouped": []
        }
}

PLAY RECAP ***********************************************************************************
*****
localhost                  : ok=1    changed=0    unreachable=0    failed=0

mbp15:aja2 sean$ ansible-playbook show-vars-3.yaml -i inventories/junos/ --limit=aragorn

PLAY [Show variables 3] *********************************************************************
****

TASK [ansible variables] ********************************************************************
*****
ok: [aragorn] => {
    "vars": {
        "aja2_host": {
            "dns_servers": [
                "8.8.4.4",
                "8.8.8.8",
```

```
                "198.51.100.100"
            ],
            "snmp": {
                "description": "virtual SRX for testing",
                "location": "Sean's Macbook Pro"
            }
        },
        "aja2_site": {
            "ntp_servers": [
                "17.253.6.125",
                "17.253.20.125"
            ]
        },
...
    }
}


PLAY RECAP *****************************************************************************************************
*****
aragorn                          : ok=1    changed=0    unreachable=0    failed=0
```

When should you use this alternative directory layout? One benefit of this alternative arrangement is that it can be used with AWX or Ansible Tower. If you are using, or expect to use, AWX or Tower, and would like to share inventory data between the command-line Ansible environment and the AWX/Tower environment, seriously consider the alternative directory arrangement described in this section.

Also consider how much overlap exists between your different inventories. The original layout collects all host data files into a single host_vars directory and all group data files into a single group_vars directory. If there is no overlap between different inventories (for example, between the *junos* and *servers* inventories created in this section of this chapter) then a clean separation of host_vars and group_vars files may help with organization and will not result in duplicate files.

If, on the other hand, there is significant overlap between inventories, the original layout can avoid duplicate files that would be required by the alternative layout. Assume you have the inventories *firewalls*, *switches*, and *routers*. Further assume that a single device may appear in two of those inventories; for example, an EX switch used primarily as a Layer 3 device would appear in both the *switches* and *routers* inventories. The original layout would require a single host_vars file for that EX switch, while the alternative layout would require two host_vars files for that one device, one file in each of the relevant inventories. Depending on how many devices appear in multiple inventories, the duplication could become a problem.

References

Ansible Inventory:
http://docs.ansible.com/ansible/latest/intro_inventory.html

Ansible Best Practices Directory Layouts:
http://docs.ansible.com/ansible/latest/playbooks_best_practices.html#directory-layout
http://docs.ansible.com/ansible/latest/playbooks_best_practices.html#alternative-directory-layout

Ansible Variables:
http://docs.ansible.com/ansible/latest/playbooks_variables.html

Ansible set_fact module:
http://docs.ansible.com/ansible/latest/set_fact_module.html

Ansible Dynamic Inventory:
http://docs.ansible.com/ansible/latest/intro_dynamic_inventory.html

Chapter 9

Backing Up Device Configuration

It is a good idea to keep backups of your devices' configurations. This is particularly 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 playbooks for archiving complete device configurations, and a playbook for getting a subset of a device's configuration.

Revisiting the juniper_junos_config Module

Chapters 7 and 8 used the `juniper_junos_config` module to configure Junos devices, either executing configuration commands or uploading configuration files. This same module can retrieve the configuration from a Junos devices when called with the `retrieve` argument.

The `retrieve` argument accepts one of three options: `none` (the default) indicating the module should not save the device's configuration, or `candidate` or `committed` indicating which device configuration database should be saved.

You may wish to provide the `format` option to specify the format of the saved configuration file. The `juniper_junos_config` module defaults to `text` format (braces and semicolons) but you can also specify `json`, `set`, or `xml`.

Keep in mind that `json` and `set` formats require Junos support which was added in recent years; for example, if the author tries to get `json` output from his EX2200-C running Junos 12.3, he gets a lengthy error message containing the following text: "RuntimeWarning: Native JSON support is only from 14.2 onwards."

When using `retrieve` with `juniper_junos_config`, you must also specify either the `dest` or `dest_dir` options to indicate the file or directory to store the device's

configuration. The `dest` option expects a (path and) filename for the output file. By contrast the `dest_dir` option expects a path; it automatically assigns a filename in the form ansible_host.format as seen in Chapter 5 with the `juniper_junos_rpc` module.

Playbook for Backing Up Device Configurations – Get Config 1

The following playbook, `get-config.yaml`, creates a directory called *backups* in the Ansible playbook directory and backs up device configurations into that directory. Configurations are saved in files named `<inventory_hostname>.conf` (text format):

```
1|---
2|- name: Save configurations from Junos devices to files
3|  hosts:
4|    - all
5|  roles:
6|    - Juniper.junos
7|  connection: local
8|  gather_facts: no
9|
10|  vars:
11|    config_dir: "backups"
12|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}.conf"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|
16|  tasks:
17|    - name: confirm/create device configuration directory
18|      file:
19|        path: "{{ config_dir }}"
20|        state: directory
21|      run_once: yes
22|      delegate_to: localhost
23|
24|    - name: save device configuration
25|      juniper_junos_config:
26|        provider: "{{ connection_settings }}"
27|        dest: "{{ config_filename }}"
28|        format: text
29|        retrieve: committed
```

Lines 1–8 are the typical start of an Ansible playbook for Junos automation.

Lines 10–14 define variables for the configuration backup directory, the name of the configuration file within the backup directory, and the connection settings.

Lines 17–22 ensure the backup directory exists. We saw this pattern, including the `run_once` argument, with the `base-settings.yaml` playbook in Chapter 7. But what is the new argument `delegate_to`? That tells the task to execute on a host other than the one currently being processed. In this case, setting `delegate_to: localhost` makes it explicit that this task (checking or creating a directory on the local file system) is to be executed on the *localhost*, not on the network device being processed (*aragorn* or *bilbo*).

This use of delegate_to: localhost is actually redundant – because the play has connection: local set, all tasks are executed on localhost, even if the task itself reaches across the network to communicate with a network device. However, for traditional server-centric administration, which may not have connection: local set, it may be important to force a task to execute locally rather than on a remote host. In addition, we will see momentarily that this setting makes a nice change in the playbook's output for this task.

Lines 24–29 call the juniper_junos_config module to back up the device's configuration. The dest argument tells the module where to save the configuration, the format argument specifies the module should return the configuration in text format (this is the default, so this line could be omitted), and the retrieve argument tells the module to return the last committed configuration (in contrast with the candidate configuration, meaning that if a configuration change is being made, return the configuration with the in-process change).

Run the playbook against your test devices:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml

PLAY [Save configurations from Junos devices to files] *************************

TASK [confirm/create device configuration directory] **************************
changed: [bilbo -> localhost]

TASK [save device configuration] **********************************************
ok: [aragorn]
ok: [bilbo]

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

Notice the output for the "confirm/create device configuration directory" task, specifically how it says [bilbo -> localhost] instead of just [bilbo]. This is the result of the delegate_to option discussed above.

If you check your Ansible playbook (~/aja2) directory there should now be a subdirectory called backups, within which there should be a backup of each of your devices' configurations:

```
mbp15:aja2 sean$ ls -ld backups
drwxr-xr-x@ 5 sean  staff  160 Apr 28 20:20 backups

mbp15:aja2 sean$ ls -1 backups/
aragorn.conf
bilbo.conf

mbp15:aja2 sean$ cat backups/aragorn.conf
```

```
## Last commit: 2018—04—20 03:33:20 UTC by sean
version 15.1X49—D90.7;
system {
    host—name aragorn;
...
```

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. The Appendix discusses source control and the problem of sharing too much becomes even more apparent in that context.

So, what if we put the *backups* directory in our home directory instead? We could change line 11 of the playbook, the `backup_dir` variable assignment, as shown:

```
11|    config_dir: "/Users/sean/backups"
```

That would work for the author, but it makes the playbook useless for anyone else. What if we used the tilde (~) shortcut for "my home directory?"

```
11|    config_dir: "~/backups"
```

That will work for everyone, as it lets the system figure out the user's home directory. However, it assumes that every module to which we would pass the `config_dir` variable, or any other variables built from the `config_dir` variable, like `config_filename`, understands the tilde (~) as the Unix/Linux shortcut for "my home directory." Is this a good assumption?

As it turns out, both the `file` and `juniper_junos_config` modules understand the tilde and "expand" it to your home directory path (`/Users/sean` for the author's system), so the change mentioned above would work in this playbook. However, not every module handles this "tilde expansion." The author thinks it is poor practice, therefore, to pass the tilde to a module as part of a pathname.

Fortunately, Ansible includes a *filter* that can help us. Ansible (and Jinja2) uses filters to modify data in some way. The basic format for using a filter is:

{{ *data | filter* }}

The data can be simple text or a variable. The filter does not modify the data directly; instead, it reads the data and returns a modified version of the data. For this reason, you will usually see filters used in assignments (setting a new variable to the modified version of the original data) or in `when:` conditions (testing the "truthiness" of the modified version of the original data).

Jinja2 includes a number of filters that Ansible can use, and Ansible includes additional filters of its own (see the links in the References section).

The filter we need is `expanduser`, which expands a path containing a tilde (~). Change line 11 of the playbook as follows:

```
11|    config_dir: "{{ '~/backups' | expanduser }}"
```

Note the single quotes around the path `'~/backups'` – quotes make it clear that this is a single string being passed to the `expanduser` filter (if it were a variable name, you would not need the quotes), and the single quotes are to avoid confusion with the double quotes surrounding the entire expression.

Also add a `debug` task before the current "confirm/create device configuration directory" task to display the `config_dir` variable, just so we can confirm that the filter works:

```
17|    - debug:
18|        var: config_dir
```

Now run the playbook again and check the results:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml --limit=aragorn

PLAY [Save configurations from Junos devices to files] *************************

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "config_dir": "/Users/sean/backups"
}

TASK [confirm/create device configuration directory] **************************
changed: [aragorn -> localhost]

TASK [save device configuration] **********************************************
ok: [aragorn]

PLAY RECAP ********************************************************************
aragorn                    : ok=3    changed=1    unreachable=0    failed=0

mbp15:aja2 sean$ ls -d ~/back*
/Users/sean/backups

mbp15:aja2 sean$ ls -1 ~/backups/
aragorn.conf
```

Observe the output from the `debug` task; the path *~/backups* has been expanded to */Users/sean/backups* (your home directory path will be different).

This is pretty nice. In fact, let's expand on this a little bit – instead of creating just a directory for configuration backups, what if we used this approach to create a directory hierarchy for all Ansible temporary and output files? We can move the *tmp* directory used in Chapter 7 for template output into our new directory, use the new directory to store reports we generate (see Chapter 10 for an example), etc.

Moving all temporary and output files out of the playbook directory hierarchy makes it easier to share the playbooks when needed, because you do not need to figure out what files from the playbook directory need to be excluded (not copied).

Modify the playbook as follows:

```
...
10|  vars:
11|    user_data_path: "{{ '~/ansible' | expanduser }}"
12|    config_dir: "{{ user_data_path }}/config_backups"
13|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}.conf"
14|    connection_settings:
15|      host: "{{ ansible_host }}"
16|
17|  tasks:
18|    - debug:
19|        var: user_data_path
20|
21|    - debug:
22|        var: config_dir
23|
24|    - name: confirm/create device configuration directory
25|      file:
26|        path: "{{ config_dir }}"
27|        state: directory
28|      run_once: yes
29|      delegate_to: localhost
...
```

Line 11 defines the user_data_path variable to store the path to new parent directory *~/ansible* for our temporary and output files.

Line 12 builds on the user_data_path variable to store the path to the directory for configuration backups.

Lines 18–22 display the two variables so we can confirm everything is working.

Run the playbook and check the results:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml --limit=aragorn

PLAY [Save configurations from Junos devices to files] *************************

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "user_data_path": "/Users/sean/ansible"
}

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "config_dir": "/Users/sean/ansible/config_backups"
}

TASK [confirm/create device configuration directory] ******************************************
ok: [aragorn -> localhost]

TASK [save device configuration] *************************************************
ok: [aragorn]
```

```
PLAY RECAP ********************************************************************
aragorn                    : ok=4    changed=0    unreachable=0    failed=0

mbp15:aja2 sean$ ls -d ~/ans*
/Users/sean/ansible

mbp15:aja2 sean$ ls -1 ~/ansible/config_backups/
aragorn.conf
```

Nice!

Because we plan to use the new `user_data_path` variable in other playbooks, it may be useful to put it in our *group_vars/all.yaml* file so it will be automatically available for all our other playbooks. In addition, should we need to change the location of this parent directory for our output, we can change the variable in just one location instead of across a number of playbooks.

Modify *group_vars/all.yaml* as follows (remember your `ansible_python_interpreter` path may be different than the author's):

```
---
ansible_python_interpreter: /usr/local/bin/python
user_data_path: "{{ '~/ansible' | expanduser }}"
```

Delete the `user_data_path` definition from the *get-config.yaml* playbook:

```
 1|---
 2|- name: Save configurations from Junos devices to files
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    config_dir: "{{ user_data_path }}/config_backups"
12|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}.conf"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|
16|  tasks:
17|    - debug:
18|        var: user_data_path
19|
20|    - debug:
21|        var: config_dir
22|
23|    - name: confirm/create device configuration directory
24|      file:
25|        path: "{{ config_dir }}"
26|        state: directory
27|      run_once: yes
28|      delegate_to: localhost
29|
30|    - name: save device configuration
31|      juniper_junos_config:
32|        provider: "{{ connection_settings }}"
```

```
33|        dest: "{{ config_filename }}"
34|        format: text
35|        retrieve: committed
```

Run the playbook again (not shown). The results should be exactly the same as last time – moving the definition of the user_data_path variable from the playbook to a *group_vars* file should not change the value assigned to the variable.

NOTE   Keep in mind that the value of the user_data_path variable is the same for all devices. The value of a playbook variable that contains different values for different devices could be affected if moved to a *group_vars* file.

NOTE   In his production playbooks, the author has used several path variables predefined in his equivalent of our *group_vars/all.yaml* variable file – not only an equivalent to the user_data_path variable shown in this example, but also equivalents to config_dir and several other output and temporary paths. However, the author has found that when he showed a new team member a playbook which used any of these predefined paths, the new team member was often confused by the use of variables not defined in the playbook. For this reason, the author has started moving many of his path variables out of his *all.yaml* file and back into the playbooks where the paths are used. The author wanted to present the technique of defining path variables in the *all.yaml* file, particularly for paths used in several playbooks, but the reader should consider what balance of variables defined in playbook vs. *all.yaml* will be best for his or her environment and team.

Before continuing into the next section, please delete the configuration backups created so far:

```
mbp15:aja2 sean$ rm -r ~/ansible/
mbp15:aja2 sean$ rm -r ~/backups/
```

That gives us a "clean slate" as we improve our backup strategy.

Keeping a Configuration History – Get Config 3

The current get-config.yaml playbook will keep only the most recent configuration backup for each device—it replaces any prior backup file each time it is run because, for a given device, the filename of the configuration backup is always the same. What if we want to keep all unique configuration backups, allowing us to create a configuration history?

There are two considerations we must address to make this happen. The first is simply ensuring we create a new (uniquely named) 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. Our examples use the date and time because, as we have seen in previous chapters, we can get the local system date and time from Ansible, and it lets us create filenames that will sort in chronological order.

The second 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 between the two backups. The simple approach is to ignore this consideration, but over time that would result in wasting disk space with numerous files that are identical but for their filename. The numerous duplicate configuration files would also make it more difficult to locate configuration changes in our history. It is preferable to save only configuration files that represent a change of configuration from the previous backup.

Let's start with adding a date-and-time stamp to the filename. At the same time, if we are going to keep a configuration history, we should probably save the configuration files for each device in a device-specific subdirectory, so backups for different devices are logically separated.

Change the get-config.yaml playbook as follows. In addition to adding the bold-faced lines, delete the run_as and delegate_to arguments on the "confirm/create device configuration directory" task because each device now has a unique directory, and delete the debug tasks that displayed the user_data_path and config_dir variables:

```
 1|---
 2|- name: Create timestamp for filenames
 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:
13|        var: ansible_date_time.time
14|
15|    - debug:
16|        var: systime
17|
18|    - name: save timestamp in a variable for later use
19|      set_fact:
20|        timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
21|
22|- name: Save configurations from Junos devices to files
23|  hosts:
24|    - all
25|  roles:
26|    - Juniper.junos
27|  connection: local
28|  gather_facts: no
29|
30|  vars:
31|    config_dir: "{{ user_data_path }}/config_backups/{{ inventory_hostname }}"
32|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ hostvars.localhost.
timestamp }}.conf"
```

```
33|    connection_settings:
34|      host: "{{ ansible_host }}"
35|
36|  tasks:
37|    – name: confirm/create device configuration directory
38|      file:
39|        path: "{{ config_dir }}"
40|        state: directory
41|
42|    – name: save device configuration
43|      juniper_junos_config:
44|        provider: "{{ connection_settings }}"
45|        dest: "{{ config_filename }}"
46|        format: text
47|        retrieve: committed
48|
49|    – name: display path to latest backup file
50|      debug:
51|        msg: "The configuration backup is in {{ config_filename }}"
```

Lines 2–20 introduce a new play that runs on *localhost* and creates a variable `time-stamp` with the date and time. The `timestamp` variable will be used later in the play-book to add a timestamp to a configuration file's name. This is modeled on the `showvars4.yaml` playbook from Chapter 8.

Line 9 uses an Ansible filter, `replace()`, to modify how the system time is formatted and assign the altered time to variable `systime`. Ansible's variable `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 meaning to some UNIX command-line tools, so we should avoid them in filenames. The filter `replace(':', '-')` switches the colons with hyphens.

Remember from our earlier discussion of filters that `replace()` does *not* modify the `ansible_date_time.time` variable; instead, 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 variable `systime` without changing variable `ansible_date_time.time`.

The `debug` tasks on lines 12–16 are to help us see the change made by the filter; they will be removed later.

Lines 18–20 create the `timestamp` variable containing the date and time (with hyphens) from the system clock. The playbook uses `set_fact` for this because the variable needs to survive into the next play. (Recall from the discussion about variable scope at the beginning of Chapter 8 that variables defined in the `vars:` section of a play are in scope only for that play, not for subsequent plays.)

Line 31 redefines the `config_dir` variable so each device will have its own directory for configuration backups.

Line 32 modifies the `config_filename` variable to include the `timestamp` defined in the

first play. Remember that `timestamp` was generated by *localhost*, not by the current device, so we need to reference it via Ansible's `hostvars` variable.

Lines 49-51 display the path and filename of the configuration backup just saved. This is a convenience for anyone using the playbook.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml

PLAY [Create timestamp for filenames] *******************************************

TASK [Gathering Facts] **********************************************************
ok: [localhost]

TASK [debug] ********************************************************************
ok: [localhost] => {
    "ansible_date_time.time": "15:38:54"
}

TASK [debug] ********************************************************************
ok: [localhost] => {
    "systime": "15-38-54"
}

TASK [save timestamp in a variable for later use] *******************************
ok: [localhost]

PLAY [Save configurations from Junos devices to files] **************************

TASK [confirm/create device configuration directory] ***************************
changed: [bilbo]
changed: [aragorn]

TASK [save device configuration] ************************************************
ok: [aragorn]
ok: [bilbo]

TASK [display path to latest backup file] **************************************
ok: [bilbo] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/bilbo/
bilbo_2018-05-01_15-38-54.conf"
}
ok: [aragorn] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-01_15-38-54.conf"
}

PLAY RECAP **********************************************************************
aragorn                    : ok=3    changed=1    unreachable=0    failed=0
bilbo                      : ok=3    changed=1    unreachable=0    failed=0
localhost                  : ok=4    changed=0    unreachable=0    failed=0
```

Look at the output for the first task that runs in the first play:

```
TASK [Gathering Facts] **********************************************************
ok: [localhost]
```

We do not have a "Gathering Facts" task in the playbook! Where did this come

from? Because the first play has `gather_facts: yes` set, Ansible implicitly adds a task to gather facts from each host in the play, in this case only *localhost*.

Now look at the output from the first and second `debug` tasks and observe how the `replace` filter (playbook line 9) changed the format of the system time:

```
...
    "ansible_date_time.time": "15:38:54"
...
    "systime": "15-38-54"
...
```

Confirm that the configuration backups exist:

```
mbp15:aja2 sean$ ls -1 ~/ansible/config_backups/
aragorn
bilbo

mbp15:aja2 sean$ ls -1 ~/ansible/config_backups/aragorn/
aragorn_2018-05-01_15-38-54.conf
```

Nice!

Removing the Localhost Play – Get Config 4

What if we run the playbook for only one device?

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml --limit=aragorn

PLAY [Create timestamp for filenames] ******************************************
skipping: no hosts matched

PLAY [Save configurations from Junos devices to files] *************************

TASK [confirm/create device configuration directory] **************************
ok: [aragorn]

TASK [save device configuration] **********************************************
fatal: [aragorn]: FAILED! => {"msg": "The task includes an option with an undefined variable. The error
was: {{ config_dir }}/{{ inventory_hostname }}_{{ hostvars.localhost.timestamp }}.conf: 'dict object'
has no attribute 'timestamp'\n\nThe error appears to have been in '/Users/sean/aja2/get-config.yaml':
line 42, 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: save device configuration\n      ^ here\n\nexception
type: <class 'ansible.errors.AnsibleUndefinedVariable'>\nexception: {{ config_dir }}/{{ inventory_
hostname }}_{{ hostvars.localhost.timestamp }}.conf: 'dict object' has no attribute 'timestamp'"}
        to retry, use: --limit @/Users/sean/aja2/get-config.retry

PLAY RECAP ********************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=1
```

What happened here? Notice the first play was skipped:

```
PLAY [Create timestamp for filenames] ******************************************
skipping: no hosts matched
```

The first play runs only on *localhost*, but *localhost* was excluded by `--limit=aragorn`.

Because the first play did not run, the `timestamp` variable was never defined, which means `timestamp` could not be referenced in the second play, resulting in the following error (emphasis added):

"The task includes an option with an undefined variable. The error was: {{ config_dir }}/{{ inventory_hostname }}_{{ *hostvars.localhost.timestamp* }}.conf: 'dict object' *has no attribute 'timestamp'*."

To run the playbook with `--limit` you must include localhost in the limit:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml --limit=localhost,aragorn

PLAY [Create timestamp for filenames] ******************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
    "ansible_date_time.time": "17:32:20"
}

TASK [debug] *******************************************************************
ok: [localhost] => {
    "systime": "17-32-20"
}

TASK [save timestamp in a variable for later use] ******************************
ok: [localhost]

PLAY [Save configurations from Junos devices to files] *************************

TASK [confirm/create device configuration directory] **************************
ok: [aragorn]

TASK [save device configuration] **********************************************
ok: [aragorn]

TASK [display path to latest backup file] *************************************
ok: [aragorn] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-01_17-32-20.conf"
}

PLAY RECAP ********************************************************************
aragorn                    : ok=3    changed=0    unreachable=0    failed=0
localhost                  : ok=4    changed=0    unreachable=0    failed=0
```

That's better!

However, as you create more playbooks, some of which require `localhost` in the `limit` list and some of which do not, it becomes difficult to remember which does and which does not. And as you can see, the error message that results from forgetting to include `localhost` when using `--limit` is unlikely to be much help in remind-

ing you that is the problem, because it says nothing about `--limit`.

Can we modify this playbook to not require `localhost` with `--limit`? Yes, we can!

Delete the entire first play of the playbook, and modify or add the boldfaced lines:

```
 1|---
 2|- name: Save configurations from Junos devices to files
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    systime: "{{ ansible_date_time.time | replace(':', '-') }}"
12|    timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
13|    config_dir: "{{ user_data_path }}/config_backups/{{ inventory_hostname }}"
14|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ timestamp }}.conf"
15|    connection_settings:
16|      host: "{{ ansible_host }}"
17|
18|  tasks:
19|    - name: get localhost environment info (for date and time)
20|      setup:
21|      run_once: yes
22|      delegate_to: localhost
23|
24|    - name: confirm/create device configuration directory
25|      file:
26|        path: "{{ config_dir }}"
27|        state: directory
28|
29|    - name: save device configuration
30|      juniper_junos_config:
31|        provider: "{{ connection_settings }}"
32|        dest: "{{ config_filename }}"
33|        format: text
34|        retrieve: committed
35|
36|    - name: display path to latest backup file
37|      debug:
38|        msg: "The configuration backup is in {{ config_filename }}"
```

The entire first play, the one that ran on *localhost*, is gone.

Line 11 defines the `systime` variable from `ansible_date_time.time`, using the `replace()` filter to change colons into hyphens.

Line 12 defines the timestamp variable from `ansible_date_time.date` and `systime`.

These two variables are defined for all hosts, not just localhost as was the case with the previous version of the playbook.

Line 14 replaces the `hostvars.localhost.timestamp` reference with a simple `timestamp` reference, because `timestamp` is now defined for each device.

Recall from our discussion of the previous version of the playbook that we needed the play to gather facts from localhost in order to define the `ansible_date_time` variable. But line 8 of our playbook has `gather_facts: no`. How are we going to get the local date and time information?

When Ansible gathers facts, it basically runs the `setup` module. We have manually run the `setup` module in previous chapters (`ansible -m setup localhost`) to view some of the system-defined variables.

Lines 19–22 call the `setup` module from our playbook. There are no arguments passed to the `setup` module itself (no lines indented after line 20), but we use the `run_once` and `delegate_to` arguments on the task to ensure we run `setup` only one time and that it gathers its facts from *localhost*.

One interesting thing about using `run_once`: even though the task is run under the name of a single host, any resulting variables are automatically added to the host variables for *all* hosts. This means that the `ansible_date_time` variable is defined for all hosts against which we run the playbook.

Run the playbook on one of your test devices:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml --limit=bilbo

PLAY [Save configurations from Junos devices to files] *************************

TASK [get localhost environment info (for date and time)] *********************
ok: [bilbo -> localhost]

TASK [confirm/create device configuration directory] *************************************
ok: [bilbo]

TASK [save device configuration] ************************************************
ok: [bilbo]

TASK [display path to latest backup file] **************************************
ok: [bilbo] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/bilbo/
bilbo_2018-05-01_17-41-22.conf"
}

PLAY RECAP ********************************************************************
bilbo                      : ok=4    changed=0    unreachable=0    failed=0
```

Run the playbook again without `--limit` so it runs against all your devices. Notice how all devices' filenames are defined and include the timestamp, even though only one device ran the `setup` module:

```
...
TASK [save device configuration] **********************************************
ok: [aragorn]
ok: [bilbo]

TASK [display path to latest backup file] *************************************
ok: [bilbo] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/bilbo/
```

```
bilbo_2018-05-01_18-13-09.conf"
}
ok: [aragorn] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-01_18-13-09.conf"
}
...
```

Nice!

Avoiding Duplicate Configuration Backups – Get Config 5

Now let's address the problem of duplicate configuration backups. At this point, our get-config.yaml playbook saves every configuration backup under a unique, time-stamped filename, with no attempt to determine if the newly backed up configuration has the same contents (the same configuration) as the previous backup, or to delete the new file when it is a duplicate. Let's add those features.

You can see the problem if you simply diff the last two backups made by our playbook for one of your test devices, because we have not changed the device's configuration while developing the playbook:

```
mbp15:aja2 sean$ cd ~/ansible/config_backups/aragorn/

mbp15:aragorn sean$ ls -1
aragorn_2018-05-01_15-38-54.conf
aragorn_2018-05-01_17-32-20.conf
aragorn_2018-05-01_17-42-42.conf
aragorn_2018-05-01_18-13-09.conf
aragorn_2018-05-01_18-24-56.conf

mbp15:aragorn sean$ diff aragorn_2018-05-01_18-13-09.conf aragorn_2018-05-01_18-24-56.conf

mbp15:aragorn sean$ cd ~/aja2/
```

Here the diff command returned no results, meaning the contents of the files were the same. If there had been differences between the files, we would have gotten some output from the diff command showing the differences. There is no need to keep one of these files. The author's preference is to delete the newer of the duplicates; keeping the older one, with its time-stamped filename, helps to document when we first backed up the changed configuration.

In the next few pages we update our playbook to do the following:

- Get a list of existing configuration backups, if any.

- If previous backups were found, save the name of the most recent backup.

- Save the new configuration backup.

- If there was a previous backup (step 2), diff the new backup with the previous backup. If there was no previous backup, skip this step.

■ If the diff shows no change, delete the newer backup file. If no diff was done (because there was no previous backup), skip this step.

There are several decisions in these steps. How does an Ansible playbook make decisions?

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 conditional; it allows Ansible to determine whether or not it should run the task with which the `when` conditional is associated. Our updated playbook uses `when` on several tasks to avoid executing the tasks for steps 4 and 5 when there was no previous backup to diff, and to decide if the diff found a change and thus if the newest backup should be deleted.

The *condition* for `when` (or any other conditional) is a valid Jinja2 expression, without double braces (`{{ }}`), that must evaluate to the Boolean values *true* or *false*.

An example of a task with a `when` condition:

```
- name: save file when data defined
  template:
    src: templates/save-info.j2
    dest: "{{ output_file }}"
  when: some_data is defined
```

This task would use the `template` module to process `template/save-info.j2` only when variable `some_data` exists. This would be useful if `some_data` might not exist but contains data needed by the template.

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 the example task were in a Python program, an equivalent expression might look something like this:

```
if some_data is defined:
    template(src=templates/save-info.j2, dest=output_file)
```

Now let's modify the playbook to include the additional processing discussed above. Add the boldfaced lines shown:

```
 1|---
 2|- name: Save configurations from Junos devices to files
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    systime: "{{ ansible_date_time.time | replace(':', '-') }}"
12|    timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
```

```
13|     config_dir: "{{ user_data_path }}/config_backups/{{ inventory_hostname }}"
14|     config_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ timestamp }}.conf"
15|     connection_settings:
16|       host: "{{ ansible_host }}"
17|
18|   tasks:
19|     - name: get localhost environment info (for date and time)
20|       setup:
21|       run_once: yes
22|       delegate_to: localhost
23|
24|     - name: confirm/create device configuration directory
25|       file:
26|         path: "{{ config_dir }}"
27|         state: directory
28|
29|     - name: get list of previous backups (if any)
30|       find:
31|         file_type: file
32|         path: "{{ config_dir }}"
33|         patterns: "{{ inventory_hostname }}*.conf"
34|       register: old_configs_details
35|
36|     - debug:
37|         var: old_configs_details
38|
39|     - name: save name of most recent previous backup
40|       set_fact:
41|         prev_config: "{{ old_configs_details.
files | sort(attribute='path') | map(attribute='path') | list | last }}"
42|       when: old_configs_details.matched > 0
43|
44|     - debug:
45|         var: prev_config
46|
47|     - name: save device configuration
48|       juniper_junos_config:
49|         provider: "{{ connection_settings }}"
50|         dest: "{{ config_filename }}"
51|         format: text
52|         retrieve: committed
53|
54|     - name: display path to latest backup file
55|       debug:
56|         msg: "The configuration backup is in {{ config_filename }}"
57|
58|     - name: get difference between backups
59|       shell: diff -I '^## Last [change|commit]' {{ prev_config }} {{ config_filename }}
60|       when: prev_config is defined
61|       register: diff_result
62|       failed_when: diff_result.rc > 1
63|
64|     - debug:
65|         var: diff_result
66|
67|     - name: delete new backup if same as previous
68|       file:
69|         path: "{{ config_filename }}"
70|         state: absent
```

```
71|      when: (diff_result.changed) and (diff_result.rc == 0)
```

Lines 29–34 use Ansible's `find` module, which returns a list of files matching specified criteria, to get a list of earlier configuration backup files. Lines 31–33 provide the matching criteria: only files (directory entries whose `file_type` is `file`), not directories or links; in the `path` (directory) specified by the `config_dir` variable; and whose names match the specified `patterns`. Line 34 registers the results in variable `old_configs_details` for use later in the playbook.

Lines 36–37 display the file information from the previous task. This task can be removed, or its output suppressed with a `verbosity` argument, once you understand how the playbook works. When we run the playbook shortly, observe two key fields in the `old_configs_details` dictionary: `matched`, an integer count of the number of items that matched the criteria, and `files`, a list of dictionaries in which each dictionary contains the filename and other information about a matching file.

Lines 39–42 get the last file from the list of old backup files, sorted alphabetically by name. Because the filenames contain a timestamp that will sort alphabetically from oldest to newest, this effectively gives us the name of the most recent previous configuration backup.

There is a lot happening in line 41, so let's break it down. The variable reference `old_configs_details.files` returns the list of dictionaries of information about each file. This list is run through a series of filters (notice the "vertical bar" or pipe characters, |). The first filter is `sort(attribute='path')`, which sorts the file list by the `path` attribute of each dictionary in the list; `path` is the dictionary's key for the fully qualified filename. The next filter, `map(attribute='path')`, extracts just the `path` attribute from the dictionaries and creates a new list containing those paths. However, the list created by `map` is in a special format, so the next filter, `list`, reformats it as a standard list. Finally, the `last` filter removes and returns the last entry in the list. Because the list is, effectively, sorted by timestamp, this gives us the path (and filename) of the most recent backup file. That path is then assigned to the `prev_config` variable.

Note the `when` condition on line 42. If there were no old backup files (if this is the first backup for a device), the `old_configs_details.files` reference would return an empty list, which will cause errors in the subsequent filters. The `when` condition tests if there was at least one file found by the "get list of previous backups" task by ensuring the `old_configs_details.matched` value is greater than zero; if `old_configs_details.matched` is zero, the condition is *false* and the task is skipped, in which case the `prev_config` variable is undefined.

The `debug` tasks on lines 44–45 and 64–65 can be removed, or their output suppressed with `verbosity` arguments, when convenient. For now, they show intermediate results to help us understand the playbook's operation.

Lines 58–62 find the difference between the new configuration backup (taken by

lines 47–52) and the previous backup. The Ansible `shell` module calls the Unix/Linux command shell and executes the specified command. In our task, the base command is `diff` with the filenames of the previous and new config backups. The additional argument to *diff*, `-I '^## Last [change|commit]'`, tells *diff* to ignore the header line that Junos adds to the top of a configuration that includes the date and time the configuration was committed. Junos updates the timestamp on this line even if there was no configuration change in the commit. Ignoring this line lets *diff* regard as the same two configuration backups which differ only by the commit timestamp.

Line 62 uses a feature of Ansible we have not previously discussed. The `failed_when` argument on a task lets us alter what Ansible module regards as failure for a given task. Normally, the `shell` module looks at the return code from the called command; a non-zero return code is a failure, while a zero is success. However, *diff*, at least on MacOS, uses the return code to indicate if the files were the same (return code 0) or different (return code 1). We use `failed_when` to check the return code (`diff_result.rc`) and consider the task a failure only if the code is 2 or greater.

NOTE   The manpages for *diff* checked by the author do not document the return codes, so the author is not sure if the behavior documented here is consistent for all implentations. Should your version of *diff* use different return codes you may be able to remove the `failed_when` argument, or change the test used by `failed_when`. Another possible test is `diff_result.stderr != ""` to check if the *diff* command send output (anything other than the empty string) to STDERR.

Lines 67–71 delete the new configuration backup when it matches the previous backup. The `when` condition has two tests joined with `and`, so both tests must be true. The first test, `diff_result.changed`, basically confirms that task "get difference between backups" ran. The second test, `diff_result.rc == 0`, checks the return code from the *diff* command called by task "get difference between backups" -- a return code of 0 means no difference was found between the files.

NOTE   Should your system's implementation of *diff* use different return codes, an alternate test is `diff_result.stdout == ""`. If *diff* returned no output (the empty string) then there was no difference found between the files.

Okay, its time to run the playbook and see how it works! So that we can see how the playbook works for new devices and for unchanged configurations, let's delete all existing configuration backups for one of our test devices:

```
mbp15:aja2 sean$ rm -r ~/ansible/config_backups/aragorn/
```

Now run the playbook for that device:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml -l aragorn

PLAY [Save configurations from Junos devices to files] **************************
```

```
TASK [get localhost environment info (for date and time)] *********************
ok: [aragorn -> localhost]

TASK [confirm/create device configuration directory] *************************
changed: [aragorn]

TASK [get list of previous backups (if any)] ********************************
ok: [aragorn]

TASK [debug] ***************************************************************
ok: [aragorn] => {
    "old_configs_details": {
        "changed": false,
        "examined": 0,
        "failed": false,
        "files": [],
        "matched": 0,
        "msg": ""
    }
}

TASK [save name of most recent previous backup] *****************************
skipping: [aragorn]

TASK [debug] ***************************************************************
ok: [aragorn] => {
    "prev_config": "VARIABLE IS NOT DEFINED!"
}

TASK [save device configuration] *********************************************
ok: [aragorn]

TASK [display path to latest backup file] ************************************
ok: [aragorn] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-15-49.conf"
}

TASK [get difference between backups] ****************************************
skipping: [aragorn]

TASK [debug] ***************************************************************
ok: [aragorn] => {
    "diff_result": {
        "changed": false,
        "skip_reason": "Conditional result was False",
        "skipped": true
    }
}

TASK [delete new backup if same as previous] *********************************
skipping: [aragorn]

PLAY RECAP ****************************************************************
aragorn                    : ok=8    changed=1    unreachable=0    failed=0
```

Observe that the "get list of previous backups (if any)" task found no existing files:

```
ok: [aragorn] => {
```

```
    "old_configs_details": {
        "changed": false,
        "examined": 0,
        "failed": false,
        "files": [],
        "matched": 0,
        "msg": ""
    }
}
```

As a result, the variable `prev_config` does not get defined:

```
TASK [save name of most recent previous backup] ********************************
skipping: [aragorn]

TASK [debug] *******************************************************************
ok: [aragorn] => {
    "prev_config": "VARIABLE IS NOT DEFINED!"
}
```

This means we skip testing for differences between the new and previous backup (because there is no previous backup):

```
TASK [get difference between backups] ******************************************
skipping: [aragorn]
```

And, finally, we skip deleting the new configuration backup:

```
TASK [delete new backup if same as previous] ***********************************
skipping: [aragorn]
```

Check the directory where the backups for this device are stored to confirm the new configuration file is still there:

```
mbp15:aja2 sean$ ls -l ~/ansible/config_backups/aragorn/
total 16
-rw-r--r--  1 sean  staff  4527 May 20 11:15 aragorn_2018-05-20_11-15-49.conf
```

Okay, the playbook handles the first backup for a device. Now let's test if it handles a backup with a change. Make a minor change to your test device:

```
sean@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# set snmp description "vSRX for writing AJA2"

[edit]
sean@aragorn# commit and-quit
commit complete
Exiting configuration mode
```

Run the playbook again:

```
mbp15:aja2 sean$ ansible-playbook get-config.yaml -l aragorn

PLAY [Save configurations from Junos devices to files] *************************

TASK [get localhost environment info (for date and time)] *********************
```

```
ok: [aragorn -> localhost]

TASK [confirm/create device configuration directory] ****************************
ok: [aragorn]

TASK [get list of previous backups (if any)] ***********************************
ok: [aragorn]

TASK [debug] *******************************************************************
ok: [aragorn] => {
    "old_configs_details": {
        "changed": false,
        "examined": 1,
        "failed": false,
        "files": [
            {
                "atime": 1526829355.509161,
                "ctime": 1526829353.8808928,
                "dev": 16777221,
                "gid": 20,
                "inode": 8604987922,
                "isblk": false,
                "ischr": false,
                "isdir": false,
                "isfifo": false,
                "isgid": false,
                "islnk": false,
                "isreg": true,
                "issock": false,
                "isuid": false,
                "mode": "0644",
                "mtime": 1526829353.8808928,
                "nlink": 1,
                "path": "/Users/sean/ansible/config_backups/aragorn/aragorn_2018-05-20_11-15-49.
conf",
                "rgrp": true,
                "roth": true,
                "rusr": true,
                "size": 4527,
                "uid": 502,
                "wgrp": false,
                "woth": false,
                "wusr": true,
                "xgrp": false,
                "xoth": false,
                "xusr": false
            }
        ],
        "matched": 1,
        "msg": ""
    }
}

TASK [save name of most recent previous backup] ********************************
ok: [aragorn]

TASK [debug] *******************************************************************
ok: [aragorn] => {
    "prev_config": "/Users/sean/ansible/config_backups/aragorn/aragorn_2018-05-20_11-15-49.conf"
}
```

```
TASK [save device configuration] ********************************************
ok: [aragorn]

TASK [display path to latest backup file] ***********************************
ok: [aragorn] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-38-35.conf"
}

TASK [get difference between backups] ***************************************
changed: [aragorn]

TASK [debug] ****************************************************************
ok: [aragorn] => {
    "diff_result": {
        "changed": true,
        "cmd": "diff -d -I '^## Last [change|commit]' /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-15-49.conf /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-38-35.conf",
        "delta": "0:00:00.014214",
        "end": "2018-05-20 11:38:39.072608",
        "failed": false,
        "failed_when_result": false,
        "msg": "non-zero return code",
        "rc": 1,
        "start": "2018-05-20 11:38:39.058394",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "173c173\n<     description \"virtual SRX for testing\";\n---\
n>     description \"vSRX for writing AJA2\";",
        "stdout_lines": [
            "173c173",
            "<     description \"virtual SRX for testing\";",
            "---",
            ">     description \"vSRX for writing AJA2\";"
        ]
    }
}

TASK [delete new backup if same as previous] *******************************
skipping: [aragorn]

PLAY RECAP ****************************************************************
aragorn                    : ok=10    changed=1    unreachable=0    failed=0
```

This time, there is a previous backup file:

```
ok: [aragorn] => {
    "old_configs_details": {
        "changed": false,
        "examined": 1,
        "failed": false,
        "files": [
            {
...
                "path": "/Users/sean/ansible/config_backups/aragorn/aragorn_2018-05-20_11-15-49.
conf",
```

```
...
            }
        ],
        "matched": 1,
        "msg": ""
    }
}
```

This means the prev_config variable gets defined:

```
ok: [aragorn] => {
    "prev_config": "/Users/sean/ansible/config_backups/aragorn/aragorn_2018-05-20_11-15-49.conf"
}
```

Thus the playbook runs the *diff* command, which finds a difference between the new and previous configuration files:

```
ok: [aragorn] => {
    "diff_result": {
        "changed": true,
        "cmd": "diff -d -I '^## Last [change|commit]' /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-15-49.conf /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-38-35.conf",
        "delta": "0:00:00.014214",
        "end": "2018-05-20 11:38:39.072608",
        "failed": false,
        "failed_when_result": false,
        "msg": "non-zero return code",
        "rc": 1,
        "start": "2018-05-20 11:38:39.058394",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "173c173\n<      description \"virtual SRX for testing\";\n---\
n>      description \"vSRX for writing AJA2\";",
        "stdout_lines": [
            "173c173",
            "<      description \"virtual SRX for testing\";",
            "---",
            ">      description \"vSRX for writing AJA2\";"
        ]
    }
}
```

Because the files differ, the new backup file is *not* deleted:

```
TASK [delete new backup if same as previous] ************************************
skipping: [aragorn]
```

Confirm we now have two backup files:

```
mbp15:aja2 sean$ ls -l ~/ansible/config_backups/aragorn/
total 32
-rw-r--r--  1 sean  staff  4527 May 20 11:15 aragorn_2018-05-20_11-15-49.conf
-rw-r--r--  1 sean  staff  4525 May 20 11:38 aragorn_2018-05-20_11-38-35.conf
```

Manually *diff* the files. Note how the output from the manual *diff* command includes the changed timestamp which is excluded from consideration by the playbook and which does not appear in the playbook's output:

```
mbp15:aragorn sean$ pwd
/Users/sean/ansible/config_backups/aragorn
```

```
mbp15:aragorn sean$ diff aragorn_2018-05-20_11-15-49.conf aragorn_2018-05-20_11-38-35.conf
2c2
< ## Last commit: 2018-05-13 12:59:23 UTC by sean
---
> ## Last commit: 2018-05-13 18:59:10 UTC by sean
173c173
<     description "virtual SRX for testing";
---
>     description "vSRX for writing AJA2";
```

Run the playbook again. Because the device's configuration has not changed, this time the *diff* command finds no difference between the new and previous configuration backup, and the new file is deleted:

```
...
TASK [debug] ***********************************************************
ok: [aragorn] => {
    "diff_result": {
        "changed": true,
        "cmd": "diff -d -I '^## Last [change|commit]' /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-38-35.conf /Users/sean/ansible/config_backups/aragorn/
aragorn_2018-05-20_11-53-35.conf",
        "delta": "0:00:00.011416",
        "end": "2018-05-20 11:53:39.342873",
        "failed": false,
        "failed_when_result": false,
        "rc": 0,
        "start": "2018-05-20 11:53:39.331457",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "",
        "stdout_lines": []
    }
}

TASK [delete new backup if same as previous] ************************************
changed: [aragorn]
...
```

Finally, run a commit on your test device without making a configuration change. This will update the timestamp in the configuration.

```
sean@aragorn> show configuration | match Last
## Last commit: 2018-05-13 18:59:10 UTC by sean

sean@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# commit and-quit
commit complete
Exiting configuration mode

sean@aragorn> show configuration | match Last
## Last commit: 2018-05-13 19:19:59 UTC by sean
```

Run the playbook again (not shown). The results should be essentially the same as

the previous run when there was no configuration change. This is thanks to playbook ignoring the timestamp lines in the configurations.

Exercise for the Reader

Add to the playbook the ability to save to a file the difference found between new and previous configurations. Only save the difference when there is a change. The filename for the difference file should be the same as the new configuration backup file but with a `.diff` extension.

Hint: use a template to save the difference file.

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 confirm a setting exists or identify devices with an old setting that needs to be updated or removed. For these situations, the saved configurations are needed only temporarily, so the playbook in this section will store the configuration files in a temporary directory and the filenames will not need a timestamp.

Assume that your company has recently deployed 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 playbook 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 searching a fresh backup of just the NTP server hierarchy.

■ Similar information may appear in other parts of the configuration. For example, 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 numerous 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 working with backups saved only when a device's configuration has changed, because the age of the last saved configuration varies by device.

These concerns can be mitigated by making a new backup for all devices, containing only the relevant configuration hierarchy, and putting the new backup in a directory free of old backups. Searches in that directory should be much more

focused 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`:

```
 1|---
 2|- name: Save partial configurations from Junos devices
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    config_dir: "{{ user_data_path }}/tmp"
12|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}.conf"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|
16|  tasks:
17|    - name: erase old backup directory if it exists
18|      file:
19|        path: "{{ config_dir }}"
20|        state: absent
21|      run_once: yes
22|      delegate_to: localhost
23|
24|    - name: create backup directory
25|      file:
26|        path: "{{ config_dir }}"
27|        state: directory
28|      run_once: yes
29|      delegate_to: localhost
30|
31|    - name: save device configuration
32|      juniper_junos_config:
33|        provider: "{{ connection_settings }}"
34|        dest: "{{ config_filename }}"
35|        format: text
36|        retrieve: committed
37|        filter: "system/ntp"
```

Most of this playbook follows familiar patterns. Lines 2–8 are the typical start to a playbook. Lines 10–14 define some variables for the play, including the directory and filename for the partial config backups.

Lines 17–22 delete the partial configuration backup directory, if it exists; this ensures we do not have old configuration backups from a prior run of the playbook before we create new configuration backups. Lines 24–29 create the partial configuration backup directory. Both tasks use the `run_once` and `delegate_to` options to ensure the tasks are executed only once each during the playbook run, not once per device, and the processing is done by the local system.

Lines 31–37 back up the device's configuration as seen earlier in this chapter, with one new addition: line 37 adds a new argument, `filter`, which limits the portion of

the device's configuration that gets archived. The `filter` argument is formatted similar to a UNIX directory path, using slashes (/) between descending levels of the hierarchy (but never a leading slash). The `filter` paths always start from the top of the Junos configuration hierarchy and must include each intermediate hierarchy down to the desired hierarchy of the configuration.

NOTE   The `filter` argument only works with predefined Junos configuration hierarchy elements, not with names or user-defined elements. For example, you can filter on `interfaces` to get the Junos *interfaces* configuration hierarchy, but not on `interfaces/lo0` because *lo0* is a name.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-partial-config.yaml

PLAY [Save partial configurations from Junos devices] **************************

TASK [erase old backup directory if it exists] ********************************
ok: [bilbo -> localhost]

TASK [create backup directory] ************************************************
changed: [bilbo -> localhost]

TASK [save device configuration] **********************************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP ********************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=0
bilbo                      : ok=3    changed=1    unreachable=0    failed=0
```

Check the results in the temporary directory:

```
mbp15:aja2 sean$ ls -1 ~/ansible/tmp/
aragorn.conf
bilbo.conf

mbp15:aja2 sean$ cat ~/ansible/tmp/aragorn.conf

## Last commit: 2018-05-13 21:36:10 UTC by sean
system {
    ntp {
        server 17.253.6.125;
        server 17.253.20.125;
    }
}

mbp15:aja2 sean$ cat ~/ansible/tmp/bilbo.conf

## Last changed: 2016-02-22 02:15:39 UTC
system {
    ntp {
        server 132.163.97.4;
        server 129.6.15.27;
    }
}
```

Pretend that 129.6.15.27 is one of the old NTP servers that we intend to retire. To search the partial backups for the old NTP server, use `grep`:

```
mbp15:aja2 sean$ grep '129.6.15.27' ~/ansible/tmp/*.conf
/Users/sean/ansible/tmp/bilbo.conf:        server 129.6.15.27;
```

It looks like we need to update *bilbo's* NTP settings!

Other Options – Get Partial Config 2

The `juniper_junos_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 specifying options for how the configuration will be displayed; we'll discuss two of the options and what they do.

The first option is `inherit` and it can be set to either `inherit` or `default`. This option instructs Junos to show certain inherited or default settings when returning the configuration. The author has found `inherit: inherit` to be the more useful setting.

The second option is `groups` and it can be set only to the value `groups`. This option instructs Junos to label (add comments to help identify) inherited settings.

Because these options relate to inherited settings within a Junos configuration, such as groups and interface-ranges, please ensure (some of) your test devices' configurations contain inheritable settings.

The author's firewall *aragorn* includes a group to help set VRRP settings, which is applied to one interface:

```
sean@aragorn> show configuration groups
vrrp-priority {
    interfaces {
        <*> {
            unit <*> {
                family inet {
                    address <*> {
                        vrrp-group <*> {
                            priority 110;
                            advertise-interval 1;
                            accept-data;
                        }
                    }
                }
            }
        }
    }
}

sean@aragorn> show configuration interfaces ge-0/0/0
apply-groups vrrp-priority;
unit 0 {
    family inet {
        address 198.51.100.2/26 {
```

```
            vrrp-group 100 {
                virtual-address 198.51.100.1;
            }
        }
    }
}
```

The author's switch *bilbo* includes an interface-range:

```
sean@bilbo> show configuration interfaces
interface-range aja2 {
    member ge-0/1/0;
    member ge-0/1/1;
    unit 0 {
        family ethernet-switching {
            port-mode access;
            vlan {
                members aja2;
            }
        }
    }
}
vlan {
    unit 0 {
        family inet {
            address 198.51.100.5/26;
        }
    }
}
```

The groups and inherit 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 "| display inheritance" modifier at the Junos command line. For example:

```
sean@bilbo> show configuration interfaces | display inheritance
##
## 'ge-0/1/0' was expanded from interface-range 'aja2'
##
ge-0/1/0 {
    ##
    ## '0' was expanded from interface-range 'aja2'
    ##
    unit 0 {
        ##
        ## 'ethernet-switching' was expanded from interface-range 'aja2'
        ##
        family ethernet-switching {
            ##
            ## 'access' was expanded from interface-range 'aja2'
            ##
            port-mode access;
            ##
            ## 'vlan' was expanded from interface-range 'aja2'
            ##
            vlan {
                ##
                ## 'aja2' was expanded from interface-range 'aja2'
                ##
                members aja2;
```

```
            }
        }
    }
}
##
## 'ge-0/1/1' was expanded from interface-range 'aja2'
##
ge-0/1/1 {
    ##
    ## '0' was expanded from interface-range 'aja2'
    ##
    unit 0 {
        ##
        ## 'ethernet-switching' was expanded from interface-range 'aja2'
        ##
        family ethernet-switching {
            ##
            ## 'access' was expanded from interface-range 'aja2'
            ##
            port-mode access;
            ##
            ## 'vlan' was expanded from interface-range 'aja2'
            ##
            vlan {
                ##
                ## 'aja2' was expanded from interface-range 'aja2'
                ##
                members aja2;
            }
        }
    }
}
vlan {
    unit 0 {
        family inet {
            address 198.51.100.5/26;
        }
    }
}
```

Modify the end of the `get-partial-config.yaml` playbook as follows:

```
...
31|    - name: save device configuration
32|      juniper_junos_config:
33|        provider: "{{ connection_settings }}"
34|        dest: "{{ config_filename }}"
35|        format: text
36|        retrieve: committed
37|        filter: "interfaces"
38|        options:
39|          groups: groups
40|          inherit: inherit
```

Run the playbook and examine the configuration file for each device. The file for *aragorn* contains, in part:

```
...
interfaces {
```

```
    ge-0/0/0 {
        unit 0 {
            family inet {
                address 198.51.100.2/26 {
                    vrrp-group 100 {
                        virtual-address 198.51.100.1;
                        ##
                        ## '110' was inherited from group 'vrrp-priority'
                        ##
                        priority 110;
                        ##
                        ## '1' was inherited from group 'vrrp-priority'
                        ##
                        advertise-interval 1;
                        ##
                        ## 'accept-data' was inherited from group 'vrrp-priority'
                        ##
                        accept-data;
                    }
                }
            }
        }
    }
...
```

Observe how the settings from the applied group are shown and identified. However, notice that the apply-groups vrrp-priority statement that is part of the configuration for interface ge-0/0/0 is *not* visible in the modified configuration output.

The file for *bilbo* contains, in part:

```
...
interfaces {
    ##
    ## 'ge-0/1/0' was expanded from interface-range 'aja2'
    ##
    ge-0/1/0 {
        ##
        ## '0' was expanded from interface-range 'aja2'
        ##
        unit 0 {
            ##
            ## 'ethernet-switching' was expanded from interface-range 'aja2'
            ##
            family ethernet-switching {
                ##
                ## 'access' was expanded from interface-range 'aja2'
                ##
                port-mode access;
                ##
                ## 'vlan' was expanded from interface-range 'aja2'
                ##
                vlan {
                    ##
                    ## 'aja2' was expanded from interface-range 'aja2'
                    ##
                    members aja2;
                }
            }
        }
```

```
    }
...
```

This output shows how Junos built the configuration for ge-0/1/0 (and, not shown, for ge-0/1/1) from the `interface-range` shown previously. However, note that the `interface-range` definition itself is missing.

As you can see above, these options – specifically, the `inherit` option – supress the display of certain configuration elements when showing the results of inheriting those elements. The suppressed portions of the hierarchy include the `groups` hierarchy, `apply-groups` settings, and any `interface-range` definition.

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. However, if you are auditing a configuration to confirm all settings have been applied correctly, seeing the configuration with groups and interface-ranges "expanded" might be exactly what you want.

Take a few minutes to experiment with removing or commenting out the `groups:` `groups` option and/or changing the `inherit` option to `inherit: default`. It may help to comment out the `filter` option as well, so you see the entire configuration.

Extra and Required Variables – Get Partial Config 3

It is likely that you will need to modify the `filter` argument in the `get-partial-config.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 to use, particularly for users who may not be comfortable editing the playbook file each time they wish to run the playbook.

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 playbook, you provide an extra variable using the `--extra-vars` or `-e` command-line options.

Modify the `get-partial-config.yaml` playbook as follows:

```
 1|---
 2|- name: Save partial configurations from Junos devices
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    config_dir: "{{ user_data_path }}/tmp"
12|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}.conf"
13|    connection_settings:
```

```
14|     host: "{{ ansible_host }}"
15|
16|  tasks:
17|    - name: erase old backup directory if it exists
18|      file:
19|        path: "{{ config_dir }}"
20|        state: absent
21|      run_once: yes
22|      delegate_to: localhost
23|
24|    - name: create backup directory
25|      file:
26|        path: "{{ config_dir }}"
27|        state: directory
28|      run_once: yes
29|      delegate_to: localhost
30|
31|    - name: show filter setting from extra-vars command-line argument
32|      debug:
33|        var: filter
34|      run_once: yes
35|
36|    - name: save device configuration
37|      juniper_junos_config:
38|        provider: "{{ connection_settings }}"
39|        dest: "{{ config_filename }}"
40|        format: text
41|        retrieve: committed
42|        filter: "{{ filter }}"
```

Run the playbook with the argument --extra-vars="filter=system/host-name":

```
mbp15:aja2 sean$ ansible-playbook get-partial-config.yaml --extra-vars="filter=system/host-name"

PLAY [Save partial configurations from Junos devices] ***************************

TASK [erase old backup directory if it exists] *********************************
changed: [bilbo -> localhost]

TASK [create backup directory] *************************************************
changed: [bilbo -> localhost]

TASK [show filter setting from extra-vars command-line argument] ***************
ok: [bilbo] => {
    "filter": "system/host-name"
}

TASK [save device configuration] **********************************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *********************************************************************
aragorn                    : ok=1    changed=0    unreachable=0    failed=0
bilbo                      : ok=4    changed=2    unreachable=0    failed=0
```

Observe that the output from the "show filter setting..." task shows the value provided at the command-line for the `filter` variable.

Display one of the partial configuration files to confirm it worked:

```
mbp15:aja2 sean$ cat ~/ansible/tmp/aragorn.conf

## Last commit: 2018-05-13 22:23:04 UTC by sean
system {
    host-name aragorn;
}
```

Great! But what happens if we forget to provide the value for `filter`?

```
mbp15:aja2 sean$ ansible-playbook get-partial-config.yaml -l aragorn

PLAY [Save partial configurations from Junos devices] ****************************

TASK [erase old backup directory if it exists] *********************************
changed: [aragorn -> localhost]

TASK [create backup directory] *************************************************
changed: [aragorn -> localhost]

TASK [show filter setting from extra-vars command-line argument] ***************
ok: [aragorn] => {
    "filter": "VARIABLE IS NOT DEFINED!"
}

TASK [save device configuration] ***********************************************
fatal: [aragorn]: FAILED! => {"msg": "The task includes an option with an undefined variable. The error
was: 'filter' is undefined\n\nThe error appears to have been in '/Users/sean/aja2/get-partial-config.
yaml': line 36, 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: save device configuration\n     ^ here\n\nexception
type: <class 'ansible.errors.AnsibleUndefinedVariable'>\nexception: 'filter' is undefined"}
        to retry, use: --limit @/Users/sean/aja2/get-partial-config.retry

PLAY RECAP *********************************************************************
aragorn                    : ok=3    changed=2    unreachable=0    failed=1
```

Forgetting to provide the variable causes a failure. Digging into the somewhat lengthy error message we see "The task includes an option with an undefined variable. The error was: 'filter' is undefined." However, if we do not know that the `filter` variable is supposed to be provided by the user at the command-line, we could easily assume this was a bug in the playbook, not a user omission.

We can tell Ansible that a variable is mandatory, which allows Ansible to provide a somewhat clearer error message. This is done by adding the `mandatory` filter to the variable references. Modify lines 33 and 42 as shown:

```
31|   - name: show filter setting from extra-vars command-line argument
32|     debug:
33|       var: filter | mandatory
34|     run_once: yes
35|
36|   - name: save device configuration
37|     juniper_junos_config:
38|       provider: "{{ connection_settings }}"
39|       dest: "{{ config_filename }}"
40|       format: text
```

```
41|        retrieve: committed
42|        filter: "{{ filter | mandatory }}"
```

NOTE   The variable reference on line 42 is the important one because this is the reference that is critical to the operation of the playbook. The `debug` task containing line 33 already gracefully handles undefined variables, and this task would likely to be removed from a production version of this playbook. However, it is normally best to include "`| mandatory`" on the first reference for a mandatory variable, which in this example is line 33. If you wish to experiment, comment out lines 31–34 and run the playbook to see how the results change from what is shown below.

Run the playbook again without provided the `filter` variable:

```
mbp15:aja2 sean$ ansible-playbook get-partial-config.yaml -l aragorn

PLAY [Save partial configurations from Junos devices] ***************************

TASK [erase old backup directory if it exists] *********************************
changed: [aragorn -> localhost]

TASK [create backup directory] ************************************************
changed: [aragorn -> localhost]

TASK [show filter setting from extra-vars command-line argument] ***************
fatal: [aragorn]: FAILED! => {"msg": "Mandatory variable not defined."}

NO MORE HOSTS LEFT ************************************************************
        to retry, use: --limit @/Users/sean/aja2/get-partial-config.retry

PLAY RECAP ******************************************************************
aragorn                    : ok=2    changed=2    unreachable=0    failed=1
```

Observe that we now get the error message "Mandatory variable not defined" when `filter` is referenced. This is probably an improvement, if only because it is short and easy to understand, but it is actually less specific than the previous error.

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 playbook 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 processing a device and, optionally, provide an error message. (We used `fail` during our debugging examples near the end of Chapter 7.) 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 as follows, removing the `mandatory` filters and adding the `fail` task:

```
1|---
```

```
 2|- name: Save partial configurations from Junos devices
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    config_dir: "{{ user_data_path }}/tmp"
12|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}.conf"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|
16|  tasks:
17|    - name: fail if variable 'filter' is not defined
18|      fail:
19|        msg: >
20|          Specify the Junos configuration hierarchy you want to back up by
21|          providing the extra variable 'filter' on the command line.
22|          For example, --extra-vars 'filter=system/ntp'
23|      when: filter is not defined
24|
25|    - name: erase old backup directory if it exists
26|      file:
27|        path: "{{ config_dir }}"
28|        state: absent
29|      run_once: yes
30|      delegate_to: localhost
31|
32|    - name: create backup directory
33|      file:
34|        path: "{{ config_dir }}"
35|        state: directory
36|      run_once: yes
37|      delegate_to: localhost
38|
39|    - name: show filter setting from extra-vars command-line argument
40|      debug:
41|        var: filter
42|      run_once: yes
43|
44|    - name: save device configuration
45|      juniper_junos_config:
46|        provider: "{{ connection_settings }}"
47|        dest: "{{ config_filename }}"
48|        format: text
49|        retrieve: committed
50|        filter: "{{ filter }}"
```

Lines 17–23 are the new task which calls `fail` module. Note the `when` condition on line 23, `filter is not defined` – this will return Boolean *true* when variable `filter` is not defined, *false* when `filter` is defined (regardless of its value).

The `msg` argument on lines 19–22 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 between each assembled line.

Run the playbook without the required argument and note the error message, particularly how the three lines in the playbook were assembled into a single message:

```
mbp15:aja2 sean$ ansible-playbook get-partial-config.yaml -l aragorn

PLAY [Save partial configurations from Junos devices] **************************

TASK [fail if variable 'filter' is not defined] *******************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Specify the Junos configuration hierarchy you
want to back up by providing the extra variable 'filter' on the command line. For example, --extra-vars
'filter=system/ntp'\n"}
         to retry, use: --limit @/Users/sean/aja2/get-partial-config.retry

PLAY RECAP ********************************************************************
aragorn                  : ok=0    changed=0    unreachable=0    failed=1
```

The playbook fails, as expected, but with a more meaningful error message.

Run the playbook with the required argument:

```
mbp15:aja2 sean$ ansible-playbook get-partial-config.yaml -l aragorn -e "filter=system/host-name"

PLAY [Save partial configurations from Junos devices] **************************

TASK [fail if variable 'filter' is not defined] *******************************
skipping: [aragorn]

TASK [erase old backup directory if it exists] ********************************
changed: [aragorn -> localhost]

TASK [create backup directory] ************************************************
changed: [aragorn -> localhost]

TASK [show filter setting from extra-vars command-line argument] **************
ok: [aragorn] => {
    "filter": "system/host-name"
}

TASK [save device configuration] **********************************************
ok: [aragorn]

PLAY RECAP ********************************************************************
aragorn                  : ok=4    changed=2    unreachable=0    failed=0
```

Observe that Ansible skipped the fail task – because filter was defined, the when condition was false.

```
mbp15:aja2 sean$ cat ~/ansible/tmp/aragorn.conf

## Last commit: 2018-05-13 22:23:04 UTC by sean
system {
    host-name aragorn;
}
```

Nice!

Backups with Junos Read-Only Account

The principle of least privilege suggests we should use an account with the minimum permissions required for a given task. We have been taking our configuration backups with the same super-user Junos account we've been using to change device configuration. Can we instead use an account with read-only permissions on our Junos devices? A configuration backup, after all, requires that we read the configuration, but should not require that we enter configuration mode or alter the configuration.

We can do it, but there are a couple of challenges. Before we discuss the challenges, however, let's create a new, read-only account on our Junos devices.

NOTE   We manually create – and will soon modify – our read-only account for this example. The author leaves it as an exercise for the reader to integrate the account into their base settings template.

Let's start by creating a new SSH key pair, in RSA PEM format, for the read-only account we intend to create. We then display the public key so we can copy it. At your system command prompt:

```
$ ssh-keygen -m PEM -t rsa -f ~/.ssh/backup -C "backup@junos"
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase): <enter passphrase>
Enter same passphrase again: <re-enter passphrase>
Your identification has been saved in /Users/sean/.ssh/backup.
Your public key has been saved in /Users/sean/.ssh/backup.pub.
The key fingerprint is:
SHA256:6YX4HsdSgT6XzcO6jsaRRzKGRrHNE1DI45fkbt4HMN0 backup@junos
The key's randomart image is:
+---[RSA 2048]----+
|    .o=o         |
|     =+.o        |
|    o.==o..      |
|     +oX+o*E     |
|    ..+SB= =     |
|     o=*o. .     |
|     +=++.       |
|     .+=...      |
|     .o.o.       |
+----[SHA256]-----+
```

```
$ cat ~/.ssh/backup.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAAABAQC/WXGkDIYTG1sE8KVrlDfW+ynDYj0LQ8IBrSUA6m7FpXW07srnk45o1W9Z39Gn
46u/s1ti6Kripyqru4lpfMPgJhYh48XzcsvDoa1kzyE6dgZudy+2/YXLGWyoexXPho3TR2BYYCPoAk+l5nXS4VVEE34Dpvz+aQoO
0K/UwsZGvKdGL1SPMmJrY6ub6cYq1yA2wjqeXMpoxTZ0+yylQZt7BkPtd0OROQNkZ81U02AYIp2ktsqdyFq/
t399DH1Dweew81cEsNLPnd1YkAhB7686e32lchX9/rI+0MoMCn2vorqO5X5S2MOhDAbSNG0LfjnL6wIJSWuDu0l9tA5bydbJ
backup@junos
```

Now, let's create an account called `backup` on one of our Junos devices using the new public key and the built-in `read-only` class:

```
sean@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# set system login user backup class read-only

[edit]
sean@aragorn# set system login user backup authentication ssh-rsa "ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC/WXGkDIYTG1sE8KVrlDfW+ynDYj0LQ8IBrSUA6m7FpXW07srnk45o1W9Z
39Gn46u/s1ti6Kripyqru4lpfMPgJhYh48XzcsvDoa1kzyE6dgZudy+2/YXLGWyoexXPho3TR2BYYCPoAk+l5nXS
4VVEE34Dpvz+aQoO0K/
UwsZGvKdGL1SPMmJrY6ub6cYq1yA2wjqeXMpoxTZ0+yylQZt7BkPtd0OROQNkZ81U02AYIp2ktsqdyFq/
t399DH1Dweew81cEsNLPnd1YkAhB7686e32lchX9/rI+0MoMCn2vorq05X5S2MOhDAbSNG0LfjnL6wIJSWuDu0l9
tA5bydbJ backup@junos"

[edit]
sean@aragorn# commit and-quit
commit complete
Exiting configuration mode

sean@aragorn>
```

Now, back at our system command prompt, let's SSH to the device using our new private key and confirm everything works. Use the -i ("identity") option with the ssh command to use the new private key file, and include the backup username, as shown:

```
$ ssh -i ~/.ssh/backup backup@aragorn
Enter passphrase for key '/Users/sean/.ssh/backup': <enter passphrase>
--- JUNOS 15.1X49-D90.7 built 2017-04-29 06:51:16 UTC

backup@aragorn> show configuration
## Last commit: 2020-04-17 17:29:21 UTC by sean
version /* ACCESS-DENIED */;
system { /* ACCESS-DENIED */ };
security { /* ACCESS-DENIED */ };
interfaces { /* ACCESS-DENIED */ };
snmp { /* ACCESS-DENIED */ };
routing-options { /* ACCESS-DENIED */ };
protocols { /* ACCESS-DENIED */ };

backup@aragorn> exit
```

Hmm…the account works for login, but we cannot view the configuration. The default read-only login class does not provide the necessary permissions.

Permissions for the Backup Account

Our first challenge is creating a login class with sufficient permissions to view the configuration. Let's create a new login class called read-config and update the backup account to use that class:

```
sean@aragorn> configure
Entering configuration mode

[edit]
```

```
sean@aragorn# edit system login

[edit system login]
sean@aragorn# set class read-config permissions [ secret view view-configuration ]

[edit system login]
sean@aragorn# set user backup class read-config

[edit system login]
sean@aragorn# commit and-quit
commit complete
Exiting configuration mode

sean@aragorn>
```

Log back into the device as backup and view the configuration again. This time, we should be able to see everything, including the encrypted passwords and other "secrets" in the configuration:

```
$ ssh -i ~/.ssh/backup backup@aragorn
Enter passphrase for key '/Users/sean/.ssh/backup': <enter passphrase>
--- JUNOS 15.1X49-D90.7 built 2017-04-29 06:51:16 UTC

backup@aragorn> show configuration
## Last commit: 2020-04-17 19:09:18 UTC by sean
version 15.1X49-D90.7;
system {
    host-name aragorn;
    root-authentication {
        encrypted-password "$5$2VhUeUC5$ba4WLLZc8SoifKhetMBN5M16BnIs2KCKLZ90MV5L6i."; ## SECRET-DATA
    }
...
}

backup@aragorn> exit

Connection to aragorn closed.
```

In theory, the view-configuration permission on the login class should allow our backup user to view the entire configuration except secrets, and the secret permission adds that ability. However, the author has found there are a few areas of the Junos configuration hierarchy that might still be unreadable.

For example, the author added the following to his firewall's configuration using his full-permission account:

```
system {
    scripts {
        op {
            file test.slax;
        }
    }
}
event-options {
    policy isis-adjacency {
        events rpd_isis_adjdown;
```

```
      then {
          raise-trap;
      }
   }
}
```

If we view the configuration as the `backup` user, we see the following:

```
system {
...
    scripts { /* ACCESS-DENIED */ };
...
}
...
event-options { /* ACCESS-DENIED */ };
...
```

With our full-permission account, we can alter the `read-config` login class as follows (boldfaced line):

```
sean@aragorn> show configuration system login class read-config
idle-timeout 5;
permissions [ secret view view-configuration ];
allow-configuration "event-options|system script";
```

This change allows our `backup` account to view the relevant sections of the device's configuration. View your device's configuration to confirm the change worked.

TIP   Before finalizing your backup strategy, contrast configuration backups taken using your full-permission account with those taken using your read-only account, across a representative sampling of your devices and configurations. Be sure the read-only account's backups are not missing anything (or anything important), whether marked ACCESS DENIED, as seen here, or simply missing from the displayed configuration. Adjust the permissions on your read-only login class as needed.

Running the Playbook with the Backup Account

Our second challenge is running the configuration backup playbook so that it uses the new `backup` account when connecting to the Junos device. There are several ways we can accomplish this.

One approach would be to create a login account on our system called `backup`, copy the private key into that account's `~/.ssh/` directory, use `ssh-add` to cache the passphrase for the private key, and run the backup playbook as that user. This approach mimics the way we've been running our playbooks as our normal user, so we will not do an example. This approach also assumes that we have the ability to create user accounts as needed, which may not be the case in production environments.

Instead, let's see how we can run the playbook as a different user while we are logged in as ourselves. We discuss three approaches. In your production environment, choose the approach that works best based on your requirements.

Ansible-playbook command-line arguments

One option is to use command-line arguments to `ansible-playbook`. We need three arguments:

`-u` to specify the alternate username for the SSH connection

`-k` to tell Ansible to prompt for the SSH password, which will be used by the `juniper_junos_config` module as the passphrase for the private key

`--private-key` to specify the private key file to use for the SSH connection

Let's run the playbook with the additional arguments:

```
$ ansible-playbook get-config.yaml -l aragorn -u backup --private-key ~/.ssh/backup -k
SSH password: <enter passphrase>

PLAY [Save configurations from Junos devices to files] *************************************************
***

TASK [get localhost environment info (for date and time)] ***********************
ok: [aragorn -> localhost]

TASK [confirm/create device configuration directory] ***************************
ok: [aragorn]

TASK [get list of previous backups (if any)] ***********************************
ok: [aragorn]

TASK [debug] *******************************************************************
...

TASK [save name of most recent previous backup] *******************************
ok: [aragorn]

TASK [debug] *******************************************************************
ok: [aragorn] => {
    "prev_config": "/Users/sean/ansible/config_backups/aragorn/aragorn_2020-04-26_10-26-15.conf"
}

TASK [save device configuration] **********************************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive
mode: LockError(severity: error, bad_element: lock-configuration, message: permission denied)"}

PLAY RECAP ********************************************************************
aragorn                    : ok=7    changed=0    unreachable=0    failed=1    skipped=0    rescued=0
ignored=0
```

We got a "permission denied" error in the "save device configuration" task – that is our third challenge, which we return to shortly. Note, however, that the error is *not* "Unable to make a PyEZ connection: ConnectAuthError(192.0.2.10)", which is what we would have gotten if the problem were an authentication failure.

Confirm that the authentication to the device worked by checking the messages log. You should see a series of messages similar to this, showing successful authentication and attempting to execute commands via NETCONF:

```
Apr 26 20:06:25  aragorn sshd[8422]: (pam_sm_acct_mgmt): DEBUG: PAM_USER: backup
Apr 26 20:06:25  aragorn sshd[8422]: Accepted publickey for backup from 192.0.2.1 port 61471 ssh2: RSA
db:cc:c3:af:24:e2:c4:60:d0:95:24:7a:21:47:36:69
Apr 26 20:06:26  aragorn mgd[8426]: UI_AUTH_EVENT: Authenticated user 'backup' at permission level
'j-read-config'
Apr 26 20:06:26  aragorn mgd[8426]: UI_LOGIN_EVENT: User 'backup' login, class 'j-read-config' [8426],
ssh-connection '192.0.2.1 61471 192.0.2.10 830', client-mode 'cli'
Apr 26 20:06:26  aragorn mgd[8426]: UI_CMDLINE_READ_LINE: User 'backup', command 'xml-mode netconf
need-trailer '
Apr 26 20:06:26  aragorn mgd[8426]: UI_LOGOUT_EVENT: User 'backup' logout
Apr 26 20:06:26  aragorn mgd[8425]: UI_AUTH_EVENT: Authenticated user 'backup' at permission level
'j-read-config'
Apr 26 20:06:26  aragorn mgd[8425]: UI_LOGIN_EVENT: User 'backup' login, class 'j-read-config' [8425],
ssh-connection '192.0.2.1 61471 192.0.2.10 830', client-mode 'netconf'
Apr 26 20:06:26  aragorn mgd[8425]: UI_NETCONF_CMD: User 'backup' used NETCONF client to run command
'close-session'
Apr 26 20:06:26  aragorn mgd[8425]: UI_LOGOUT_EVENT: User 'backup' logout
```

Using SSH-authentication agent

A second option is to use the `ssh-add` command to add the private key credentials to the SSH authentication agent. (Remember to use `ssh-agent` first on Linux/Unix systems.)

```
$ ssh-add ~/.ssh/backup
Enter passphrase for /Users/sean/.ssh/backup:
Identity added: /Users/sean/.ssh/backup (/Users/sean/.ssh/backup)
```

We can then run the playbook, adding the `-u` argument to connect as the `backup` user. The `-k` and `--private-key` arguments we added in the previous approach are not needed because the authentication agent provided the necessary identity.

```
$ ansible-playbook get-config.yaml -l aragorn -u backup

PLAY [Save configurations from Junos devices to files] **************************

TASK [get localhost environment info (for date and time)] **********************
ok: [aragorn -> localhost]

TASK [confirm/create device configuration directory] ***************************
ok: [aragorn]

TASK [get list of previous backups (if any)] ***********************************
ok: [aragorn]

TASK [debug] *******************************************************************
...

TASK [save name of most recent previous backup] ********************************
ok: [aragorn]

TASK [debug] *******************************************************************
ok: [aragorn] => {
    "prev_config": "/Users/sean/ansible/config_backups/aragorn/aragorn_2020-04-26_10-26-15.conf"
}
```

```
TASK [save device configuration] *************************************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive
mode: LockError(severity: error, bad_element: lock-configuration, message: permission denied)"}

PLAY RECAP ***********************************************************************
aragorn                  : ok=6    changed=0    unreachable=0    failed=1    skipped=0    rescued=0
ignored=0
```

Again, we get a "permission denied" error in the "save device configuration" task, which we fix shortly, but we can see that the playbook is accessing our Junos device with the alternate credentials.

This approach works because the SSH client can attempt to connect to the server with each identity (private key) cached by the authentication agent, stopping when it finds a combination of username and cached identity that works.

Arguments to juniper_junos_config module

A third option is to use arguments to the `juniper_junos_config` module that provide the same data as we provided above at the command line – alternate username, private key file, and passphrase.

Copy the `get-config.yaml` playbook to a new file `get-config-readonly.yaml`. Modify or add the boldfaced lines:

```
 1|---
 2|- name: Save configurations from Junos devices to files using read-only account
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    systime: "{{ ansible_date_time.time | replace(':', '-') }}"
12|    timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
13|    config_dir: "{{ user_data_path }}/config_backups/{{ inventory_hostname }}"
14|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ timestamp }}.conf"
15|    diff_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ timestamp }}.diff"
16|    connection_settings:
17|      host: "{{ ansible_host }}"
18|      user: backup
19|      passwd: Passw0rd
20|      ssh_private_key_file: '~/.ssh/backup'
21|
22|  tasks:
23|    - name: get localhost environment info (for date and time)
24|      setup:
25|      run_once: yes
26|      delegate_to: localhost
...
51|    - name: save device configuration
52|      juniper_junos_config:
53|        provider: "{{ connection_settings }}"
54|        dest: "{{ config_filename }}"
```

```
55|      format: text
56|      retrieve: committed
...
```

Lines 18, 19, and 20 provide the necessary arguments to the `juniper_junos_config` module, by way of the `connection_settings` dictionary. Substitute your private key's passphrase for the author's on line 19.

Run the playbook and check the output.

```
$ ansible-playbook get-config-readonly.yaml -l aragorn

PLAY [Save configurations from Junos devices to files using read-only account] *

TASK [get localhost environment info (for date and time)] **********************
ok: [aragorn -> localhost]

TASK [confirm/create device configuration directory] **************************
ok: [aragorn]

TASK [get list of previous backups (if any)] **********************************
ok: [aragorn]

TASK [debug] ******************************************************************
...

TASK [save name of most recent previous backup] *******************************
ok: [aragorn]

TASK [debug] ******************************************************************
ok: [aragorn] => {
    "prev_config": "/Users/sean/ansible/config_backups/aragorn/aragorn_2020-04-26_10-26-15.conf"
}

TASK [save device configuration] **********************************************
fatal: [aragorn]: FAILED! => {"changed": false, "msg": "Unable to open the configuration in exclusive
mode: LockError(severity: error, bad_element: lock-configuration, message: permission denied)"}

PLAY RECAP ********************************************************************
aragorn                : ok=6    changed=0    unreachable=0    failed=1    skipped=0    rescued=0
ignored=0
```

Obviously, having our private key's passphrase in our playbook in plain text is a serious security concern. If you choose this approach for your production environment, consider putting the passphrase in the Ansible vault, as discussed in Chapter 11.

With that security concern in mind, delete lines 19 and 20 (the `passwd` and `ssh_private_key_file` lines) from `get-config-readonly.yaml`. We will rely on the authentication agent, combined with the `user` setting (line 18), as we further update the playbook in the next section.

Changing Configuration Backup Method

Now let's discuss that "permission denied" error, the third and final challenge we must overcome to run our configuration backup as a read-only user.

We have seen that the `juniper_junos_config` module can both retrieve a device's configuration, and change the configuration. The module attempts to enter configuration mode even if it does not need to change the configuration. The "permission error" we're getting is because our backup account does not have permission to enter configuration mode.

We could add the `configure` permission to the `read-config` role, which would allow our `backup` account to enter configuration mode, like this:

```
sean@aragorn# show | compare
[edit system login class read-config]
-    permissions [ secret view view-configuration ];
+    permissions [ configure secret view view-configuration ];
```

The problem is, entering configuration mode is the first step to altering the configuration, exactly what our read-only account is not supposed to be able to do.

Instead, let's change the playbook to use a different module, one that does not need to enter configuration mode. We can use either `juniper_junos_command` or `juniper_junos_rpc`, similar to either of the following:

```
- name: save device configuration
  juniper_junos_command:
    provider: "{{ connection_settings }}"
    commands: show configuration
    format: text


- name: save device configuration
  juniper_junos_rpc:
    provider: "{{ connection_settings }}"
    rpcs: get-configuration
    format: text
    attrs: { 'database': 'committed' }
```

Let's use the first option for our playbook. Modify the "save device configuration" task in our `get-config-readonly.yaml` playbook as follows:

```
1|---
2|- name: Save configurations from Junos devices to files using read-only account
3|  hosts:
4|    - all
5|  roles:
6|    - Juniper.junos
7|  connection: local
8|  gather_facts: no
9|
10|  vars:
11|    systime: "{{ ansible_date_time.time | replace(':', '-') }}"
12|    timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
13|    config_dir: "{{ user_data_path }}/config_backups/{{ inventory_hostname }}"
14|    config_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ timestamp }}.conf"
```

```
15|    diff_filename: "{{ config_dir }}/{{ inventory_hostname }}_{{ timestamp }}.diff"
16|    connection_settings:
17|      host: "{{ ansible_host }}"
18|      user: backup
19|
20|  tasks:
...
49|    - name: save device configuration
50|      juniper_junos_command:
51|        provider: "{{ connection_settings }}"
52|        commands: show configuration
53|        dest: "{{ config_filename }}"
54|        format: text
55|        return_output: no
...
```

Now run the playbook:

```
$ ansible-playbook get-config-readonly.yaml -l aragorn

PLAY [Save configurations from Junos devices to files using read-only account] ***

TASK [get localhost environment info (for date and time)] *********************
ok: [aragorn -> localhost]

TASK [confirm/create device configuration directory] *************************
ok: [aragorn]

TASK [get list of previous backups (if any)] *********************************
ok: [aragorn]

TASK [debug] *****************************************************************
...

TASK [save name of most recent previous backup] ******************************
ok: [aragorn]

TASK [debug] *****************************************************************
ok: [aragorn] => {
    "prev_config": "/Users/sean/ansible/config_backups/aragorn/aragorn_2020-04-28_10-57-06.conf"
}

TASK [save device configuration] *********************************************
ok: [aragorn]

TASK [display path to latest backup file] ************************************
ok: [aragorn] => {
    "msg": "The configuration backup is in /Users/sean/ansible/config_backups/aragorn/
aragorn_2020-04-28_11-16-31.conf"
}

TASK [get difference between backups] ****************************************
changed: [aragorn]

TASK [debug] *****************************************************************
...

TASK [save diff file when change found] **************************************
changed: [aragorn]
```

```
TASK [delete new backup if same as previous] ************************************
skipping: [aragorn]

PLAY RECAP *********************************************************************
aragorn                : ok=11   changed=2   unreachable=0   failed=0   skipped=1   rescued=0
ignored=0
```

With the permissions problem solved, the playbook runs as expected. Excellent!

References:

Ansible and Jinja2 filters:
http://docs.ansible.com/ansible/latest/user_guide/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

Ansible's fail module:
http://docs.ansible.com/ansible/latest/modules/fail_module.html

Ansible's find module:
http://docs.ansible.com/ansible/latest/modules/find_module.html

Junos get-configuration RPC and options:
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

Chapters 4 and 5 used the `juniper_junos_command` and `juniper_junos_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 `juniper_junos_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 a way that `juniper_junos_facts` may be helpful: we create a playbook that generates a device inventory report, which also allow us to further explore Jinja2 templates, including how to use a template to save data to a file; and we create a playbook that generates different configuration settings based on the model of the device being configured.

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.

Because we may wish to keep the inventory reports over time, the playbook creates a report directory to hold the ouptut.

Exploring juniper_junos_facts – Get Device Facts Version 1

Let's start with seeing which facts are returned by the juniper_junos_facts module. Create the playbook get-device-facts.yaml containing the following:

```
1|---
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|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|
14|  tasks:
15|    - name: get device facts
16|      juniper_junos_facts:
17|        provider: "{{ connection_settings }}"
18|      register: junos_facts
19|
20|    - name: show device facts
21|      debug:
22|        var: junos_facts
```

Everything here should be familiar from playbooks in earlier chapters, other than the results of the juniper_junos_facts module itself, which we will see below. Let's run the playbook and see what facts are gathered (output edited for length):

```
mbp15:aja2 sean$ ansible-playbook get-device-facts.yaml

PLAY [Get facts from Junos device] **************************************************************
****

TASK [get device facts] ************************************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [show device facts] ************************************************************************
****
ok: [bilbo] => {
    "junos_facts": {
        "ansible_facts": {
            "junos": {
                "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": "4 days, 19 hours, 57 minutes, 49 seconds"
                },
...
```

```
                    "has_2RE": false,
                    "hostname": "bilbo",
                    "hostname_info": {
                        "fpc0": "bilbo"
                    },
                    "ifd_style": "SWITCH",
...
                    "master": "RE0",
                    "master_state": true,
                    "model": "EX2200-C-12T-2G",
                    "model_info": {
                        "fpc0": "EX2200-C-12T-2G"
                    },
                    "personality": "SWITCH",
...
                    "re_name": "fpc0",
                    "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": "12.3R12.4",
...
                    "virtual": false
                }
            },
            "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": "4 days, 19 hours, 57 minutes, 49 seconds"
                },
...
                "has_2RE": false,
                "hostname": "bilbo",
                "hostname_info": {
                    "fpc0": "bilbo"
                },
                "ifd_style": "SWITCH",
...
                "master": "RE0",
                "master_state": true,
                "model": "EX2200-C-12T-2G",
                "model_info": {
                    "fpc0": "EX2200-C-12T-2G"
                },
                "personality": "SWITCH",
...
                "re_name": "fpc0",
                "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": "12.3R12.4",
...
                "virtual": false
            },
            "failed": false
        }
}
ok: [aragorn] => {
    "junos_facts": {
        "ansible_facts": {
            "junos": {
                "HOME": "/var/home/sean",
                "RE0": {
                    "last_reboot_reason": "0x4000:VJUNOS reboot",
                    "mastership_state": "master",
                    "model": "VSRX-S",
                    "status": "OK",
                    "up_time": "27 minutes"
                },
...
                "has_2RE": false,
                "hostname": "aragorn",
                "hostname_info": {
                    "re0": "aragorn"
                },
                "ifd_style": "CLASSIC",
...
                "master": "RE0",
                "master_state": true,
                "model": "VSRX",
                "model_info": {
                    "re0": "VSRX"
                },
                "personality": null,
...
                "re_name": "re0",
                "serialnumber": "3EE63E490CDE",
                "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",
...
                "virtual": null
            }
        },
```

```
        "changed": false,
        "facts": {
            "HOME": "/var/home/sean",
            "RE0": {
                "last_reboot_reason": "0x4000:VJUNOS reboot",
                "mastership_state": "master",
                "model": "VSRX-S",
                "status": "OK",
                "up_time": "27 minutes"
            },
...
            "has_2RE": false,
            "hostname": "aragorn",
            "hostname_info": {
                "re0": "aragorn"
            },
            "ifd_style": "CLASSIC",
...
            "master": "RE0",
            "master_state": true,
            "model": "VSRX",
            "model_info": {
                "re0": "VSRX"
            },
            "personality": null,
...
            "re_name": "re0",
            "serialnumber": "3EE63E490CDE",
            "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",
...
            "virtual": null
        },
        "failed": false
    }
}

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

For each device, the juniper_junos_facts module returns a dictionary, which we as‐
signed to playbook variable junos_facts, containing several keys. The keys changed
and failed are Boolean values indicating if the module changed the device (should
always be false for this module) or encountered an error.

The ansible_facts key holds the junos key, which contains the facts returned by the
juniper_junos_facts module. There is no key other than junos visible within the

`ansible_facts` key, which raises the question of why the module uses the nested key? There is an interesting reason, which we discuss in the next section of this chapter.

Finally, the `facts` key holds a duplicate of the data in `ansible_facts.junos`. This is primarily for backwards compatibility; this was the key used to return facts by the older `junos_get_facts` module from Juniper's `Juniper.junos` Galaxy modules, versions 1.4.3 and older.

There are too many facts to discuss them in detail in this book. Many facts are self-explanatory. The Ansible module reads the PyEZ facts dictionary `jnpr.junos.facts`, then adds or renames a few facts. See the PyEZ documentation for descriptions of most facts, and the `juniper_junos_facts` module documentation for the few changes. Links are 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 meaningful answer to the "question" posed by that fact. For example, in the author's output 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 *aragorn* 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 an EX virtual chassis, or an SRX cluster, or an MX router with dual routing engines, run the playbook against those devices and explore the facts gathered. Many of the facts include more information when working with multi-RE or multi-chassis devices.

Displaying Specific Facts – Get Device Facts Version 2

We can use keys from within either the `facts` or the `ansible_facts.junos` dictionaries to display specific data about our devices. For example, if we wanted only the Junos version, we could alter the final task of the playbook, lines 20 – 22, to read either...

```
20|    – name: show junos version
21|      debug:
22|        var: junos_facts.facts.version
```

...or...

```
20|    – name: show junos version
21|      debug:
22|        var: junos_facts.ansible_facts.junos.version
```

Take a moment and try both variations to confirm they work.

However, the `ansible_facts` key provides us an alternative approach that should be easier to use. Ansible stores facts it discovers about hosts in its `ansible_facts` dictionary. To see this, run the `setup` module against `localhost`, as we have done previously to view discovered facts such as the `ansible_date_time` dictionary:

```
mbp15:aja2 sean$ ansible –m setup localhost
localhost | SUCCESS => {
    "ansible_facts": {
...
        "ansible_date_time": {
            "date": "2018–04–17",
            "day": "17",
            "epoch": "1523986296",
            "hour": "13",
            "iso8601": "2018–04–17T17:31:36Z",
            "iso8601_basic": "20180417T133136413732",
            "iso8601_basic_short": "20180417T133136",
            "iso8601_micro": "2018–04–17T17:31:36.413828Z",
            "minute": "31",
            "month": "04",
            "second": "36",
            "time": "13:31:36",
            "tz": "EDT",
            "tz_offset": "–0400",
            "weekday": "Tuesday",
            "weekday_number": "2",
            "weeknumber": "16",
            "year": "2018"
        },
...
```

Notice that `ansible_date_time` is contained within an `ansible_facts` dictionary!

But when we referenced `ansible_date_time`, such as in the `show-vars-4.yaml` playbook in Chapter 8, we did not need to reference `ansible_facts`; we were able to directly reference `ansible_date_time`. Ansible provides a little magic here, letting us directly access discovered facts stored in `ansible_facts`.

The `juniper_junos_facts` module adds facts it discovers about devices to the `ansible_facts` dictionary, using the `junos` key and its dictionary. Even better, `juniper_junos_facts` does so whether or not we register the results from the module. What does this mean for our playbook? It means we can skip the `register` argument on the `juniper_junos_facts` task and access the keys within the `junos` dictionary without additional qualification.

Modify the playbook as follows (delete the old lines 18 – 22 and add the boldfaced lines):

```
 1|---
 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|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|
```

```
14|  tasks:
15|    - name: get device facts
16|      juniper_junos_facts:
17|        provider: "{{ connection_settings }}"
18|
19|    - name: show Junos version
20|      debug:
21|        var: junos.version
22|
23|    - name: show device uptime
24|      debug:
25|        var: junos.RE0.up_time
```

Now run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-device-facts.yaml --limit=aragorn

PLAY [Get facts from Junos device] ******************************************************
****

TASK [get device facts] *****************************************************************
****
ok: [aragorn]

TASK [show Junos version] ***************************************************************
****
ok: [aragorn] => {
    "junos.version": "15.1X49-D90.7"
}

TASK [show device uptime] ***************************************************************
****
ok: [aragorn] => {
    "junos.RE0.up_time": "5 hours, 45 minutes, 11 seconds"
}

PLAY RECAP ******************************************************************************
*****
aragorn                 : ok=3    changed=0    unreachable=0    failed=0
```

Nice!

Saving Facts to a File – Get Device Facts Version 3

What if we want to save these facts to a file? The juniper_junos_facts module can do this for us, provided we want all the facts and are not picky about the filename.

The savedir argument to the juniper_junos_facts module lets us specify a directory in which we want to save a JSON file with the device's facts. The file will be named *hostname-facts.json*, where *hostname* is the value of the junos.hostname fact as discovered by the module when it queries the device.

Modify the playbook as shown:

```
 1|---
 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|  vars:
11|    facts_dir: "{{ user_data_path }}/facts"
12|    connection_settings:
13|      host: "{{ ansible_host }}"
14|
15|  tasks:
16|    – name: confirm/create report directory
17|      file:
18|        path: "{{ facts_dir }}"
19|        state: directory
20|      run_once: yes
21|      delegate_to: localhost
22|
23|    – name: get device facts
24|      juniper_junos_facts:
25|        provider: "{{ connection_settings }}"
26|        savedir: "{{ facts_dir }}"
```

The facts_dir variable on line 11 holds the name of the directory in which we will store the facts file, based on the user_data_path value we defined in Chapter 9 for our device configuration backups and temporary files.

The task on lines 16–21 ensures the facts directory exists.

Line 25 is the savedir option, instructing the juniper_junos_facts module to save the device's facts in our new facts directory.

Now run the playbook:

```
mbp15:aja2 sean$ ansible–playbook get–device-facts.yaml --limit=aragorn

PLAY [Get facts from Junos device] ***************************************************
****

TASK [confirm/create report directory] **********************************************
****
changed: [aragorn -> localhost]

TASK [get device facts] *************************************************************
****
ok: [aragorn]

PLAY RECAP *************************************************************************
*****
aragorn                 : ok=2    changed=1    unreachable=0    failed=0
```

Observe that a JSON file was created in the indicated directory:

```
mbp15:aja2 sean$ ls -l ~/ansible/facts/
total 16
-rw-r--r-- 1 sean staff 1406 Apr 17 22:05 aragorn-facts.json
-rw-r--r-- 1 sean staff  784 Apr 17 22:05 aragorn-inventory.xml
```

The `juniper_junos_facts` module also creates a *hostname-inventory.xml* file which contains the XML results of the `chassis-inventory` RPC ("show chassis hardware" command).

Display the JSON data:

```
mbp15:aja2 sean$ cat ~/ansible/facts/aragorn-facts.json
```
```
{"domain": null, "hostname_info": {"re0": "aragorn"}, "version_RE1": null, "version_RE0":
"15.1X49-D90.7", "re_master": {"default": "0"}, "serialnumber": "3EE63E490CDE", "vc_master": null,
"RE_hw_mi": false, "HOME": "/var/home/sean", "master_state": true, "re_info": {"default": {"default":
{"status": "OK", "last_reboot_reason": "0x4000:VJUNOS reboot", "model": "VSRX-S", "mastership_state":
"master"}, "0": {"status": "OK", "last_reboot_reason": "0x4000:VJUNOS reboot", "model": "VSRX-S",
"mastership_state": "master"}}}, "srx_cluster_id": null, "hostname": "aragorn", "virtual": null,
"version": "15.1X49-D90.7", "master": "RE0", "vc_fabric": null, "personality": null, "srx_cluster_
redundancy_group": null, "version_info": {"major": [15, 1], "type": "X", "build": 7, "minor": [49,
"D", 90]}, "re_name": "re0", "srx_cluster": false, "vc_mode": null, "vc_capable": false, "ifd_style":
"CLASSIC", "model_info": {"re0": "VSRX"}, "RE0": {"status": "OK", "last_reboot_reason": "0x4000:VJUNOS
reboot", "model": "VSRX-S", "up_time": "6 hours, 13 minutes, 3 seconds", "mastership_state":
"master"}, "RE1": null, "fqdn": "aragorn", "junos_info": {"re0": {"text": "15.1X49-D90.7", "object":
{"major": [15, 1], "type": "X", "build": 7, "minor": [49, "D", 90]}}}, "has_2RE": false, "switch_
style": "VLAN_L2NG", "model": "VSRX", "current_re": ["master", "node", "fwdd", "member", "pfem",
"fpc0", "re0", "fpc0.pic0"]}
```

The unformatted JSON data is very hard to read. We can use the following trick to "pretty-print" the JSON data:

```
mbp15:aja2 sean$ python -m json.tool < ~/ansible/facts/aragorn-facts.json
{
    "HOME": "/var/home/sean",
    "RE0": {
        "last_reboot_reason": "0x4000:VJUNOS reboot",
        "mastership_state": "master",
        "model": "VSRX-S",
        "status": "OK",
        "up_time": "6 hours, 13 minutes, 3 seconds"
    },
    "RE1": null,
...
    "has_2RE": false,
    "hostname": "aragorn",
    "hostname_info": {
        "re0": "aragorn"
    },
    "ifd_style": "CLASSIC",
...
    "master": "RE0",
    "master_state": true,
    "model": "VSRX",
...
    "virtual": null
}
```

Much better!

Saving Facts Using a Template – Get Device Facts Version 4

Recall that we wanted to create a CSV file with an inventory report. Saving the device facts as JSON data is nice but does not satisfy our objective.

Ansible does not have a "save CSV file" module, but we can use Ansible's core module `template` to help us 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 playbook we use `template` to take specific facts from the `junos` dictionary and write them to a file on disk.

Let's start with a very basic template. Create file `~/aja2/template/device-facts.j2` with the following content:

```
{{ junos }}
```

This template inserts the entire `junos` dictionary (`ansible_facts.junos`) into the result file. We will clean this up shortly.

Each device's data is saved in a separate file, so we need to assemble several devices' files together to create a single report. To help with this, save the individual devices' results into a *build* directory underneath our *facts* directory. We discuss the assembly process shortly.

Update the `get-device-facts.yaml` playbook as follows (remove the `savedir` argument from the "get device facts" task and add the boldfaced lines):

```
 1|---
 2|- name: Get facts from Junos device and save as CSV file
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    facts_dir: "{{ user_data_path }}/facts"
12|    facts_build_dir: "{{ user_data_path }}/facts/build"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|
16|  tasks:
17|    - name: confirm/create report directory
18|      file:
19|        path: "{{ facts_dir }}"
20|        state: directory
21|      run_once: yes
22|      delegate_to: localhost
23|
24|    - name: confirm/create build directory
25|      file:
26|        path: "{{ facts_build_dir }}"
27|        state: directory
```

```
28|      run_once: yes
29|      delegate_to: localhost
30|
31|    - name: get device facts
32|      juniper_junos_facts:
33|        provider: "{{ connection_settings }}"
34|
35|    - name: save device facts
36|      template:
37|        src: template/device-facts.j2
38|        dest: "{{ facts_build_dir }}/{{ inventory_hostname }}.txt"
```

Line 12 defines a variable with the path to our build directory.

Lines 24–29 ensure the build directory exists.

Lines 35–38 call the template module to save the device's facts to a file. The output file will be *inventory_hostname*.txt in the build directory.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-device-facts.yaml

PLAY [Get facts from Junos device and save as CSV file] ************************************************
***

TASK [confirm/create report directory] ****************************************************************
****
ok: [bilbo -> localhost]

TASK [confirm/create build directory] *****************************************************************
****
changed: [bilbo -> localhost]

TASK [get device facts] *******************************************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [save device facts] ******************************************************************************
****
changed: [aragorn]
changed: [bilbo]

PLAY RECAP ********************************************************************************************
*****
aragorn                    : ok=2    changed=1    unreachable=0    failed=0
bilbo                      : ok=4    changed=2    unreachable=0    failed=0
```

Check the results:

```
mbp15:aja2 sean$ ls -1 ~/ansible/facts/build/
aragorn.txt
bilbo.txt

mbp15:aja2 sean$ cat ~/ansible/facts/build/aragorn.txt
{u'domain': None, u'hostname_info': {u're0': u'aragorn'}, u'version_RE1': None, u'version_RE0':
```

u'15.1X49–D90.7', u're_master': {u'default': u'0'}, u'serialnumber': u'3EE63E490CDE', u'vc_master':
None, u'RE_hw_mi': False, u'HOME': u'/var/home/sean', u'master_state': True, u're_info': {u'default':
{u'default': {u'status': u'OK', u'last_reboot_reason': u'0x4000:VJUNOS reboot', u'model': u'VSRX–S',
u'mastership_state': u'master'}, u'0': {u'status': u'OK', u'last_reboot_reason': u'0x4000:VJUNOS
reboot', u'model': u'VSRX–S', u'mastership_state': u'master'}}}, u'srx_cluster_id': None,
u'hostname': u'aragorn', u'virtual': None, u'version': u'15.1X49–D90.7', u'master': u'RE0', u'vc_
fabric': None, u'personality': None, u'srx_cluster_redundancy_group': None, u'version_info':
{u'major': [15, 1], u'type': u'X', u'build': 7, u'minor': [49, u'D', 90]}, u're_name': u're0', u'srx_
cluster': False, u'vc_mode': None, u'vc_capable': False, u'ifd_style': u'CLASSIC', u'model_info':
{u're0': u'VSRX'}, u'RE0': {u'status': u'OK', u'last_reboot_reason': u'0x4000:VJUNOS reboot',
u'model': u'VSRX–S', u'up_time': u'1 day, 10 hours, 40 minutes, 50 seconds', u'mastership_state':
u'master'}, u'RE1': None, u'fqdn': u'aragorn', u'junos_info': {u're0': {u'text': u'15.1X49–D90.7',
u'object': {u'major': [15, 1], u'type': u'X', u'build': 7, u'minor': [49, u'D', 90]}}}, u'has_2RE':
False, u'switch_style': u'VLAN_L2NG', u'model': u'VSRX', u'current_re': [u'master', u'node', u'fwdd',
u'member', u'pfem', u'fpc0', u're0', u'fpc0.pic0']}

Yes, those are the device facts, albeit not formatted for human consumption. We can make this data more appealing by modifying the template to print each key:value pair on its own line.

Modify `template/device-facts.j2` as shown:

```
{% for fact_name, fact_data in junos.items() %}
    {{ fact_name }}: {{ fact_data }}
{% endfor %}
```

This template defines 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` *dictionary*, meaning each entry consists of both a *key* and a *value*.

Because we want to display both the key (name) and value (data) for each dictionary entry, the *for* loop declaration needs to follow this pattern:

```
{% for key, value in variable.items() %}
```

The new `items()` keyword (technically, a function or method of 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:aja2 sean$ cat ~/ansible/facts/build/aragorn.txt
    domain:
    hostname_info: {u're0': u'aragorn'}
    version_RE1:
    version_RE0: 15.1X49–D90.7
    re_master: {u'default': u'0'}
    serialnumber: 3EE63E490CDE
    vc_master:
    RE_hw_mi: False
```

```
    HOME: /var/home/sean
    master_state: True
    re_info: {u'default': {u'default': {u'status': u'OK', u'last_reboot_
reason': u'0x4000:VJUNOS reboot', u'model': u'VSRX-S', u'mastership_
state': u'master'}, u'0': {u'status': u'OK', u'last_reboot_
reason': u'0x4000:VJUNOS reboot', u'model': u'VSRX-S', u'mastership_state': u'master'}}}
    srx_cluster_id:
    hostname: aragorn
    virtual:
    version: 15.1X49-D90.7
    master: RE0
    vc_fabric:
    personality:
    srx_cluster_redundancy_group:
    version_info: {u'major': [15, 1], u'type': u'X', u'build': 7, u'minor': [49, u'D', 90]}
    re_name: re0
    srx_cluster: False
    vc_mode:
    vc_capable: False
    ifd_style: CLASSIC
    model_info: {u're0': u'VSRX'}
    RE0: {u'status': u'OK', u'last_reboot_
reason': u'0x4000:VJUNOS reboot', u'model': u'VSRX-S', u'up_
time': u'1 day, 11 hours, 12 minutes, 2 seconds', u'mastership_state': u'master'}
    RE1:
    fqdn: aragorn
    junos_info: {u're0': {u'text': u'15.1X49-D90.7', u'object': {u'major': [15, 1], u'type': u'X',
u'build': 7, u'minor': [49, u'D', 90]}}}
    has_2RE: False
    switch_style: VLAN_L2NG
    model: VSRX
    current_re: [u'master', u'node', u'fwdd', u'member', u'pfem', u'fpc0', u're0', u'fpc0.pic0']
```

Notice how each element of the `junos` 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 guarantee the sequence of elements in a dictionary. Still, this represents a significant improvement over the first template.

We could make this output more attractive by, for example, displaying the junos dictionary's keys in sorted order, and by further indenting and formatting the next level of dictionaries. The author will leave these tasks as excercises for the reader, as we need to get back to the task of creating a CSV report.

Should you iterate over a dictionary *without* using `items()` – in other words, iterate over the dictionary as if it were a list – you will get only the keys from the dictionary. If you wish to see this, modify the template as follows...

```
{% for fact in junos %}
    {{ fact }}
{% endfor %}
```

...and re-run the playbook. Go ahead, give it a try, we'll wait for you.

Creating a Single CSV Report – Get Device Facts Version 5

Right now, we have a separate result file for each device, not a single report, and the devices' results are not comma separated.

In order to generate a single CSV report, 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, we need to assemble the individual device files and column labels into a single report. Third, while not strictly necessary, it would be nice if each "column" in the CSV file had a label.

Adjusting the template to put all *desired* values (not necessarily all available values from the `junos` dictionary) on a single line, separated by commas, could be done as follows. This is a single line in the template, though it wraps to several lines here:

```
"{{ inventory_hostname }}","{{ junos.version }}","{{ junos.model }}","{{ junos.switch_style }}","{{
junos.serialnumber }}","{{ junos.has_2RE }}","{{ junos.master }}","{{ junos.vc_capable }}","{{ junos.
vc_fabric }}","{{ junos.vc_master }}","{{ junos.vc_mode }}","{{ junos.srx_cluster }}","{{ junos.srx_
cluster_id }}"
```

NOTE   This report uses a subset of the available device facts; feel free to add or remove facts to suit your reporting needs.

Run the playbook (not shown) and view the resulting files:

```
mbp15:aja2 sean$ cat ~/ansible/facts/build/aragorn.txt
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False","","","","Fal
se",""
```

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 these `null` values showed as a hyphen (-) or other placeholder instead of nothing; it helps assure anyone looking at the results that the value 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` (`none`) or return the value if the variable is not `null` (`none`). The basic format is:

```
1|  {% if junos.srx_cluster_id is none %}
2|      "-"
3|  {% else %}
4|      "{{ junos.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 `junos.srx_cluster_id is none` tests if the variable `junos.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 formatting. This following one line is logically equivalent to the five lines above, except for the leading spaces:

```
{% if junos.srx_cluster_id is none %}"-"{% else %}"{{ junos.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 different from the preferred code layout. This template is one of those times; we want the entire output to be on a single line. However, let's start by using the preferred code layout, as this makes it easier to debug any template problems other than line breaks.

Change the `device-facts.j2` template as follows (line numbers added for discussion, and line 1 may wrap in the book):

```
 1|"{{ inventory_hostname }}","{{ junos.version }}","{{ junos.model }}","{{ junos.switch_
style }}","{{ junos.serialnumber }}","{{ junos.has_2RE }}","{{ junos.master }}","{{ junos.vc_
capable }}",
 2|{% if junos.vc_fabric is none %}
 3|    "-",
 4|{% else %}
 5|    "{{ junos.vc_fabric }}",
 6|{% endif %}
 7|{% if junos.vc_master is none %}
 8|    "-",
 9|{% else %}
10|    "{{ junos.vc_master }}",
11|{% endif %}
12|{% if junos.vc_mode is none %}
13|    "-",
14|{% else %}
15|    "{{ junos.vc_mode }}",
16|{% endif %}
17|{% if junos.srx_cluster is none %}
18|    "-",
19|{% else %}
20|    "{{ junos.srx_cluster }}",
21|{% endif %}
22|{% if junos.srx_cluster_id is none %}
23|    "-"
```

```
24|{% else %}
25|    "{{ junos.srx_cluster_id }}"
26|{% endif %}
```

Line 1 includes the facts that should always have a value.

Lines 2-6 displays either a hyphen or the `junos.vc_fabric` fact, depending on whether or not `vc_fabric` is `null` (`none`).

Lines 7-11 do the same for the `junos.vc_master` fact.

And so on for the remaining three facts that may be `null` for some devices.

Run the playbook (not shown) and examine the results:

```
mbp15:aja2 sean$ cat ~/ansible/facts/build/aragorn.txt
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False",
    "-",
    "-",
    "-",
    "False",
    "-",
```

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 hyphens or data are not really desirable either. Jinja2 includes the leading spaces and the trailing newline characters from each line of the template which contains either plain text or 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? (Line numbers added for discussion.)

```
1|"{{ inventory_hostname }}","{{ junos.version }}","{{ junos.model }}","{{ junos.switch_
style }}","{{ junos.serialnumber }}","{{ junos.has_2RE }}","{{ junos.master }}","{{ junos.vc_
capable }}",
2|{% if junos.vc_fabric is none %}"-",{% else %}"{{ junos.vc_fabric }}",{% endif %}
3|{% if junos.vc_master is none %}"-",{% else %}"{{ junos.vc_master }}",{% endif %}
4|{% if junos.vc_mode is none %}"-",{% else %}"{{ junos.vc_mode }}",{% endif %}
5|{% if junos.srx_cluster is none %}"-",{% else %}"{{ junos.srx_cluster }}",{% endif %}
6|{% if junos.srx_cluster_id is none %}"-"{% else %}"{{ junos.srx_cluster_id }}"{% endif %}
```

Now the template's output looks something like this:

```
mbp15:aja2 sean$ cat ~/ansible/facts/build/aragorn.txt
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","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 2–6 are not included in the template's output. But the newline from line 1 is still present, as this line contains only variable references and text (the commas and quotation marks). This causes our complete results to be spread across two lines.

One approach to fixing this is to add a hyphen to the opening of the control structure on the lines following the newline we wish to suppress – in other words, control 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|"{{ inventory_hostname }}","{{ junos.version }}","{{ junos.model }}","{{ junos.switch_
style }}","{{ junos.serialnumber }}","{{ junos.has_2RE }}","{{ junos.master }}","{{ junos.vc_
capable }}",
2|{%- if junos.vc_fabric is none %}"-",{% else %}"{{ junos.vc_fabric }}",{% endif %}
3|{%- if junos.vc_master is none %}"-",{% else %}"{{ junos.vc_master }}",{% endif %}
4|{%- if junos.vc_mode is none %}"-",{% else %}"{{ junos.vc_mode }}",{% endif %}
5|{%- if junos.srx_cluster is none %}"-",{% else %}"{{ junos.srx_cluster }}",{% endif %}
6|{%- if junos.srx_cluster_id is none %}"-"{% else %}"{{ junos.srx_cluster_id }}"{% endif %}
```

While we needed to modify only line 2 to fix the current problem, the author chose to update lines 2–6. Adding the hyphen to lines 3–6 has no effect because Jinja2 is already suppressing the newlines from the preceding lines, but if we ever insert a new variable reference, say {{ junos.ifd_style }}, between the current lines 4 and 5, 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 for each file, though it may wrap as shown here):

```
mbp15:aja2 sean$ cat ~/ansible/facts/build/aragorn.txt
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False","-","-","-
","False","-"

mbp15:aja2 sean$ cat ~/ansible/facts/build/bilbo.txt
"bilbo","12.3R12.4","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl
ed","-","-"
```

We have CSV-formatted output for each device...but we are not quite done yet. Now we must assemble the data files for different devices into a single CSV file.

Ansible has a core module called assemble that concatenates a group of files into a new file. The files to be assembled must be within a single directory; assemble concatenates all the files in that directory into a new file. This is why we placed the template output files into a different directory than the one where we wanted to save the final inventory report. (We used a subdirectory of the report directory, but that is not required.)

Let's put a date stamp in the inventory report filename so the reports from differnt runs of the playbook will have unique names.

Modify the get-device-facts.yaml playbook as follows:

```
 1|---
 2|- name: Get facts from Junos device and save as CSV file
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    facts_dir: "{{ user_data_path }}/facts"
```

```
12|    facts_build_dir: "{{ user_data_path }}/facts/build"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|    systime: "{{ ansible_date_time.time | replace(':', '-') }}"
16|    timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
17|    report_file: "{{ facts_dir }}/device-facts_{{ timestamp }}.csv"
18|
19|  tasks:
20|    - name: get localhost environment info
21|      setup:
22|      run_once: yes
23|      delegate_to: localhost
24|
25|    - name: confirm/create report directory
26|      file:
27|        path: "{{ facts_dir }}"
28|        state: directory
29|      run_once: yes
30|      delegate_to: localhost
31|
32|    - name: confirm/create build directory
33|      file:
34|        path: "{{ facts_build_dir }}"
35|        state: directory
36|      run_once: yes
37|      delegate_to: localhost
38|
39|    - name: get device facts
40|      juniper_junos_facts:
41|        provider: "{{ connection_settings }}"
42|
43|    - name: save device facts
44|      template:
45|        src: template/device-facts.j2
46|        dest: "{{ facts_build_dir }}/{{ inventory_hostname }}.csv"
47|
48|    - name: assemble inventory report
49|      assemble:
50|        src: "{{ facts_build_dir }}"
51|        dest: "{{ report_file }}"
52|      run_once: yes
53|      delegate_to: localhost
```

Lines 15–17 add new variables that work together to define a report filename with a timestamp. We saw a similar pattern in Chapter 9 when creating backup files with a timestamp in the filename.

Lines 20–23 run the Ansible `setup` module to get local environment information; we are interested primarily in the local date and time to support the new `systime` and `timestamp` variables on lines 15 and 16. We need run this module only once on the localhost; lines 22 and 23 ensure this for us, without requiring that we specify `localhost` when running the playbook with `--limit`. We saw this in Chapter 9 when creating backup files with a timestamp in the filename.

On line 46 we change the extension of the `template` module's output file to *.csv*

from *.txt* to better reflect the updated nature of the template's results.

Lines 48–53 call the `assemble` module, concatenating the files in the build directory (`src` argument) and creating a single CSV file (`dest` argument). Because we need assemble the files only once, on the local host, the `run_once: yes` and `delegate_to: localhost` arguments are included.

Run the playbook and review the CSV file:

```
mbp15:aja2 sean$ ansible-playbook get-device-facts.yaml

PLAY [Get facts from Junos device and save as CSV file] *************************************************
***

TASK [get localhost environment info] ****************************************************************
****
ok: [bilbo -> localhost]

TASK [confirm/create report directory] ***************************************************************
****
ok: [bilbo -> localhost]

TASK [confirm/create build directory] ****************************************************************
****
ok: [bilbo -> localhost]

TASK [get device facts] ******************************************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [save device facts] *****************************************************************************
****
changed: [aragorn]
changed: [bilbo]

TASK [assemble inventory report] *********************************************************************
****
changed: [bilbo -> localhost]

PLAY RECAP *******************************************************************************************
*****
aragorn                    : ok=2    changed=1    unreachable=0    failed=0
bilbo                      : ok=6    changed=2    unreachable=0    failed=0

mbp15:aja2 sean$ cat ~/ansible/facts/device-facts_2018-04-22_20-32-05.csv
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False","-","-","-
","False","-"
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False","-","-","-
","False","-"
"bilbo","12.3R12.4","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl
ed","-","-"
"bilbo","12.3R12.4","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl
ed","-","-"
```

The CSV report looks good except that there are duplicate lines. Why? Take a look at the contents of the build directory:

```
mbp15:aja2 sean$ ls -l ~/ansible/facts/build/
aragorn.csv
aragorn.txt
bilbo.csv
bilbo.txt
```

We did not delete the old *.txt* output files when we renamed the template output files to *.csv*. Because the `assemble` module concatenates *all* files in the source directory, it happily included both the new *.csv* and old *.txt* files. A similar problem would occur if you ran the playbook using a `--limit`, then ran it again with a different `--limit` matching a different set of devices – the second run would include both sets of devices in the assembled report.

Let's fix this by adding a task to delete the build directory before the current task that creates it (or ensures that it exists). Update the playbook as follows:

```
...
19|  tasks:
20|    - name: get localhost environment info
21|      setup:
22|      run_once: yes
23|      delegate_to: localhost
24|
25|    - name: confirm/create report directory
26|      file:
27|        path: "{{ facts_dir }}"
28|        state: directory
29|      run_once: yes
30|      delegate_to: localhost
31|
32|    - name: delete old build directory if present
33|      file:
34|        path: "{{ facts_build_dir }}"
35|        state: absent
36|      run_once: yes
37|      delegate_to: localhost
38|
39|    - name: create build directory
40|      file:
41|        path: "{{ facts_build_dir }}"
42|        state: directory
43|      run_once: yes
44|      delegate_to: localhost
...
```

Lines 32–37 are the new task. We saw this pattern before when deleting temporary files; this time we are deleting a directory.

Run the playbook again and check the results:

```
mbp15:aja2 sean$ ansible-playbook get-device-facts.yaml

PLAY [Get facts from Junos device and save as CSV file] ************************************************
***

TASK [get localhost environment info] ****************************************************************
****
ok: [bilbo -> localhost]
```

```
TASK [confirm/create report directory] ****************************************
ok: [bilbo -> localhost]

TASK [delete old build directory if present] ********************************
changed: [bilbo -> localhost]

TASK [create build directory] *************************************************
changed: [bilbo -> localhost]

TASK [get device facts] *******************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [save device facts] ******************************************************
****
changed: [bilbo]
changed: [aragorn]

TASK [assemble inventory report] **********************************************
****
changed: [bilbo -> localhost]

PLAY RECAP ********************************************************************
*****
aragorn                 : ok=2    changed=1    unreachable=0    failed=0
bilbo                   : ok=7    changed=4    unreachable=0    failed=0

mbp15:aja2 sean$ ls -1 ~/ansible/facts/build/
aragorn.csv
bilbo.csv

mbp15:aja2 sean$ cat ~/ansible/facts/device-facts_2018-04-22_21-04-30.csv
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False","-","-","-
","False","-"
"bilbo","12.3R12.4","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl
ed","-","-"
```

Great! Now take a look at the CSV file using a spreadsheet or using the preview feature of MacOS:

| aragorn | 15.1X49-D90.7 | VSRX | VLAN_L2NG | 3EE63E490CDE | False | RE0 | False | - | | - | - | | False | - |
| bilbo | 12.3R12.4 | EX2200-C-12T-2G | VLAN | GP0211463844 | False | RE0 | True | False | 0 | Enabled | - | | - | |

The only thing missing is the column headers...let's add those next!

Column Headers – Get Device Facts Version 6

We can include column headers in our inventory report as part of the assemble process, provided we first place a file with the headers in our build directory. However, the order of assembly becomes a consideration; we want the column headers file to be the first file assembled, so the headers are the first row of the CSV report file.

According to Ansible's online documentation, the `assemble` module assembles files in "string sorting order." As long as the filename for the column headers will sort 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 filename starting with "AAA" should sort to the top of the list (remember to use capital "A" not lower-case "a" because sorts are case sensitive, and "A" precedes "a").

The author uses a slightly different approach, but his approach may not work for everyone. The author prefixes the filename for his column names with an underscore (_). The underscore is easily typed, commonly used in filenames, and has no other meaning to UNIX/Linux (many other symbols which can be used in filenames 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 ASCII or ANSI sort order.

Unfortunately, the underscore comes *after* a capital "A" when sorted, so this naming 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 build directory (keep in mind the build directory is a temporary 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 before the "assemble inventory report" task near the end of the playbook:

```
 1|---
 2|- name: Get facts from Junos device and save as CSV file
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    facts_dir: "{{ user_data_path }}/facts"
12|    facts_build_dir: "{{ user_data_path }}/facts/build"
13|    connection_settings:
14|      host: "{{ ansible_host }}"
15|    systime: "{{ ansible_date_time.time | replace(':', '-') }}"
16|    timestamp: "{{ ansible_date_time.date }}_{{ systime }}"
```

```
17|    report_file: "{{ facts_dir }}/device-facts_{{ timestamp }}.csv"
18|
19| tasks:
20|   - name: get localhost environment info
21|     setup:
22|     run_once: yes
23|     delegate_to: localhost
24|
25|   - name: confirm/create report directory
26|     file:
27|       path: "{{ facts_dir }}"
28|       state: directory
29|     run_once: yes
30|     delegate_to: localhost
31|
32|   - name: delete old build directory if present
33|     file:
34|       path: "{{ facts_build_dir }}"
35|       state: absent
36|     run_once: yes
37|     delegate_to: localhost
38|
39|   - name: create build directory
40|     file:
41|       path: "{{ facts_build_dir }}"
42|       state: directory
43|     run_once: yes
44|     delegate_to: localhost
45|
46|   - name: get device facts
47|     juniper_junos_facts:
48|       provider: "{{ connection_settings }}"
49|
50|   - name: save device facts
51|     template:
52|       src: template/device-facts-5.j2
53|       dest: "{{ facts_build_dir }}/{{ inventory_hostname }}.csv"
54|
55|   - name: copy column headers file
56|     copy:
57|       src: template/_device-facts-columns.txt
58|       dest: "{{ facts_build_dir}}/"
59|     run_once: yes
60|     delegate_to: localhost
61|
62|   - name: assemble inventory report
63|     assemble:
64|       src: "{{ facts_build_dir }}"
65|       dest: "{{ report_file }}"
66|     run_once: yes
67|     delegate_to: localhost
```

Lines 55–60 are the new task, which uses Ansible's copy method to copy the column headers file (src argument) into the build directory (dest argument).

Run the playbook and examine the results:

```
mbp15:aja2 sean$ ansible-playbook get-device-facts.yaml

PLAY [Get facts from Junos device and save as CSV file] *********************************************
***

TASK [get localhost environment info] **************************************************************
****
ok: [bilbo -> localhost]

TASK [confirm/create report directory] *****************************************
ok: [bilbo -> localhost]

TASK [delete old build directory if present] **********************************
changed: [bilbo -> localhost]

TASK [create build directory] **************************************************
changed: [bilbo -> localhost]

TASK [get device facts] ********************************************************
ok: [aragorn]
ok: [bilbo]

TASK [save device facts] *******************************************************
changed: [aragorn]
changed: [bilbo]

TASK [copy column headers file] ************************************************
changed: [bilbo -> localhost]

TASK [assemble inventory report] ***********************************************
changed: [bilbo -> localhost]

PLAY RECAP *****************************************************************************************
*****
aragorn                    : ok=2    changed=1    unreachable=0    failed=0
bilbo                      : ok=8    changed=5    unreachable=0    failed=0

mbp15:aja2 sean$ ls -1 ~/ansible/facts/build/
_device-facts-columns.txt
aragorn.csv
bilbo.csv

mbp15:aja2 sean$ cat ~/ansible/facts/device-facts_2018-04-22_22-17-12.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"
"aragorn","15.1X49-D90.7","VSRX","VLAN_L2NG","3EE63E490CDE","False","RE0","False","-","-","-
","False","-"
"bilbo","12.3R12.4","EX2200-C-12T-2G","VLAN","GP0211463844","False","RE0","True","False","0","Enabl
ed","-","-"
```

Hostname	Junos version	Model	Switch Style	Serial Number	Dual RE	Master	VC Capable	VC Fabric	VC Master	VC Mode	SRX Cluster	SRX Cluster ID
aragorn	15.1X49-D90.7	VSRX	VLAN_L2NG	3EE63E490CDE	False	RE0	False	-	-	-	False	-
bilbo	12.3R12.4	EX2200-C-12T-2G	VLAN	GP0211463844	False	RE0	True	False	0	Enabled	-	-

Perfect!

Device Configuration Based on Device Type – Base Settings 3

Junos configuration statements and options are remarkably consistent across different types of devices, but there are some places where configuration requirements 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 settings, the playbook will need to accommodate these differences.

For this chapter, we will use a simpler example: configuring the maximum number of concurrent SSH or NETCONF sessions allowed by a device. Most Junos devices allow large numbers of concurrent management connections, and you can set a limit of hundreds of concurrent 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@aragorn> configure
Entering configuration mode

[edit]
sean@aragorn# set system services ssh connection-limit ?
Possible completions:
  <connection-limit>   Maximum number of allowed connections (1..5)
[edit]
sean@aragorn# set system services ssh rate-limit ?
Possible completions:
  <rate-limit>         Maximum number of connections per minute (1..5)
[edit]
sean@aragorn#
```

Similarly, many branch SRX devices, like the SRX210 or SRX300, allow only 3 or 5 depending on model and Junos version.

That's a big difference. Limiting concurrent connections to a value far less than 250 can help mitigate the impact of some brute force or denial-of-service attacks, so setting a limit in our base settings playbook and template would be useful.

However, limiting all devices to 3 concurrent connections, the highest value allowed by the most restrictive device, is probably undesirable.

Let's update our `base-settings.yaml` playbook and `base-settings.j2` template to include connection limits for SSH and NETCONF, but set the limit based on the device type. The `basesettings.yaml` playbook will gather facts from the devices; 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:

```
 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|    connection_settings:
14|      host: "{{ ansible_host }}"
15|      timeout: 120
16|
17|  tasks:
18|    - name: confirm or create configs directory
19|      file:
20|        path: "{{ tmp_dir }}"
21|        state: directory
22|      run_once: yes
23|
24|    - name: get device facts
25|      juniper_junos_facts:
26|        provider: "{{ connection_settings }}"
27|
28|    - name: display device facts
29|      debug:
30|        var: junos
31|        verbosity: 1
32|
33|    - name: save device configuration using template
34|      template:
35|        src: template/base-settings.j2
36|        dest: "{{ conf_file }}"
37|
38|    - name: install generated configuration file onto device
39|      juniper_junos_config:
40|        provider: "{{ connection_settings }}"
41|        src: "{{ conf_file }}"
42|        load: replace
43|        comment: "playbook base-settings.yaml, commit confirmed"
44|        confirmed: 5
45|        diff: yes
46|        ignore_warning: yes
47|      register: config_results
```

```
48|      notify: confirm previous commit
49|
50|    - name: show configuration change
51|      debug:
52|        var: config_results.diff_lines
53|      when: config_results.diff_lines is defined
54|
55|    # - name: delete generated configuration file
56|    #   file:
57|    #     path: "{{ conf_file }}"
58|    #     state: absent
59|
60|  handlers:
61|    - name: confirm previous commit
62|      juniper_junos_config:
63|        provider: "{{ connection_settings }}"
64|        comment: "playbook base-settings.yaml, confirming previous commit"
65|        commit: yes
66|        diff: no
```

The new tasks are similar to tasks we used in the `get-device-facts.yaml` playbook. Lines 24–26 gather device facts (and save those facts in variable `junos`). Lines 28–31 display the device facts when the playbook is run in verbose mode.

Lines 55–58 are commented out to avoid deleting the generated configuration file. This is optional, but it helps when debugging the template.

To keep the logic for determining the connection limit fairly simple, we'll use a limit of 3 for all branch SRX devices (even though some will support 5 or more), 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|{% set model = junos.model.lower() %}
 3|{% set personality = junos.personality | lower %}
 4|
 5|{#- Determine SSH connection-limit and rate-limit based on device facts #}
 6|{% if model == 'vsrx' %}
 7|    {% set max_ssh = 5 %}
 8|{% elif personality == 'srx_branch' %}
 9|    {% set max_ssh = 3 %}
10|{% else %}
11|    {% set max_ssh = 10 %}
12|{% endif %}
13|
14|{#- Generate basic settings for the device #}
15|system {
16|    host-name {{ inventory_hostname }};
17|    login {
18|        user sean {
19|            uid 2000;
20|            class super-user;
21|            authentication {
22|                ssh-rsa "ssh-rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
23|            }
24|        }
```

```
25|    }
26|    replace:
27|    name-server {
28|      {% for server in aja2_host.dns_servers %}
29|        {{ server }};
30|      {% endfor %}
31|    }
32|    services {
33|        delete: ftp;
34|        netconf {
35|            ssh {
36|                connection-limit {{ max_ssh }};
37|                rate-limit {{ max_ssh }};
38|            }
39|        }
40|        ssh {
41|            connection-limit {{ max_ssh }};
42|            rate-limit {{ max_ssh }};
43|        }
44|        delete: telnet;
45|        delete: web-management;
46|    }
47|    replace:
48|    ntp {
49|      {% for ntp in aja2_site.ntp_servers %}
50|        server {{ ntp }};
51|      {% endfor %}
52|    }
53|}
54|snmp {
55|    description "{{ aja2_host.snmp.description}}"
56|    location "{{ aja2_host.snmp.location}}"
57|}
```

Lines 5 and 14 are comments – note the {#...#} delimiters. These lines are essentially ignored by Jinja2, but they help anyone reading the template understand what is happening.

Note the extra hyphen after the opening of the comments on lines 5 and 14 – 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 4 and 13 – 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 2 and 3 copy values from the device facts into template variables – the Jinja2 {% set var = value %} control structure assigns a value to a variable. At the same time, these lines convert to lower-case the text from the device facts before assigning the lower-case to the template variables – more on how this works in a moment.

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, for example, some Junos 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 `junos.personality` fact from "SRX_BRANCH" to "SRX_branch."

Two different approaches for making the text lower-case are shown: line 2 calls the `lower()` method of Python's string class, while line 3 uses the Jinja2 filter `|lower`. They both do the same job, taking text from the variable on which they are called and returning a lower-case version of that text – but there is an interesting difference between the two approaches. Should the variable have a `null` value, the `lower()` method throws an error ("AnsibleUndefinedVariable: 'None' has no attribute 'lower'") while the `|lower` filter does not. This is important on line 3 as *aragorn*, the author's vSRX, returns a `null` for its `personality` fact.

Lines 6–12 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 subsequent lines will be in the template output; if false, move on to the next *elif* statement (you can have more than one) or the *else* statement.

Note the double equal sign in the conditions on lines 6 and 8. 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 7, 9, and 11 use the Jinja2 `set` control structure to assign the desired maximum connections value to the `max_ssh` template variable.

Lines 35–43 add the additional `connection-limit` and `rate-limit` settings we want, using the `max_ssh` value determined earlier in the template.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml

PLAY [Generate and Install Configuration File] *******************************************************
****

TASK [confirm or create configs directory] *********************************************************
****
ok: [bilbo]

TASK [get device facts] ****************************************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [display device facts] *************************************************************************
****
```

```
skipping: [bilbo]
skipping: [aragorn]

TASK [save device configuration using template] *******************************************
****
changed: [aragorn]
changed: [bilbo]

TASK [install generated configuration file onto device] ************************************
****
changed: [aragorn]
changed: [bilbo]

TASK [show configuration change] **********************************************************
****
ok: [bilbo] => {
    "config_results.diff_lines": [
        "",
        "[edit system services ssh]",
        "+    connection-limit 10;",
        "+    rate-limit 10;",
        "[edit system services netconf ssh]",
        "+     connection-limit 10;",
        "+     rate-limit 10;",
        "[edit system ntp]",
        "-    server 172.29.135.60;"
    ]
}
ok: [aragorn] => {
    "config_results.diff_lines": [
        "",
        "[edit system services ssh]",
        "+    connection-limit 5;",
        "+    rate-limit 5;",
        "[edit system services netconf ssh]",
        "+     connection-limit 5;",
        "+     rate-limit 5;"
    ]
}

RUNNING HANDLER [confirm previous commit] *************************************************
****
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *******************************************************************************
*****
aragorn                    : ok=5    changed=2    unreachable=0    failed=0
bilbo                      : ok=6    changed=2    unreachable=0    failed=0
```

Observe that the configuration changes are exactly what we expected, with a limit of 10 for the EX *bilbo* and 5 for the vSRX *aragorn*. You can also inspect the generated configuration files:

```
mbp15:aja2 sean$ grep limit tmp/aragorn.conf
            connection-limit 5;
            rate-limit 5;
```

```
            connection-limit 5;
            rate-limit 5;

mbp15:aja2 sean$ grep limit tmp/bilbo.conf
                connection-limit 10;
                rate-limit 10;
            connection-limit 10;
            rate-limit 10;
```

Note that the configuration file for each device has the correct values.

References

Juniper_junos_facts module:
http://junos-ansible-modules.readthedocs.io/en/stable/juniper_junos_facts.html

PyEZ jnpr.junos.facts module:
http://junos-pyez.readthedocs.io/en/stable/jnpr.junos.facts.html

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 discuss the `ansible-vault` command for creating vault files and editing vault data, and we 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 (to create a new file encrypted with vault), use the command `ansible-vault create` followed by the name of the new file to create. This will prompt you for a password, then open your system 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:aja2 sean$ ansible-vault create vault1.yml
New Vault password: <enter password>
Confirm New Vault password: <re-enter password>
```

You should now be in your system's default 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:

```
mbp15:aja2 sean$ cat vault1.yml
$ANSIBLE_VAULT;1.1;AES256
37326463363539396634636430383737326465373635396665643661666539613934633237616265
30646366613531666430386437343564646531336639616620a37326265652353531616162626435
66376361373166393535656137326333363330336436336535343434643466363163330663537313037
64303363353306266610a37623937396435366561346339353531623132383931643464383435303
30373362336462653566638316637383331643732356237336232338535313333383930
```

If you have an existing file you wish to encrypt with vault, use `ansible-vault`'s en-crypt option instead of `create`. To try this, first create a text file called *vault2.yml* with the following data:

```
---
vault_test_2: goodbye cruel world
```

Now encrypt that file:

```
mbp15:aja2 sean$ ansible-vault encrypt vault2.yml
New Vault password:
Confirm New Vault password:
Encryption successful

mbp15:aja2 sean$ cat vault2.yml
$ANSIBLE_VAULT;1.1;AES256
36316661613636623063636263373039346630656236366238383035383338613834326431376439
62303866393839613663656537376533323303965313961620a646163363130356132303865353534
37346536633303131356261343466666231376661626466343938326537393266653562303432646
3337383364646534630a316663373965535323036363962353237643931306637323539666303735
653664623437393336653666383031333316664383962366437343833613836666165343531643
373331326366636343566303563383963333306361343131336466
```

Viewing or Editing the Contents of a Vault File

The `ansible-vault` command has a `view` option that displays the (decrypted) contents of a vault file:

```
mbp15:aja2 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 default text editor so you can modify the file:

```
mbp15:aja2 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`

Creating a Playbook That Reads a Vault File

Now let's create a simple playbook that displays a variable from a vault file. Create playbook `test-vault.yaml` with the following:

```
 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
```

Lines 9–10 use Ansible's `include_vars` module to import the data from the (decrypted) vault file. This is needed because our test vault files are not in a "standard" location for an Ansible data file, so Ansible will not automatically import their contents. By default, imported variables are made top-level variables.

Lines 12–14 display the contents of the imported `vault_test_1` variable.

TIP   The `include_vars` module can be used to import most YAML data files, whether vault-encrypted or not.

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:aja2 sean$ ansible-playbook test-vault.yaml --ask-vault-pass
Vault 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 plain-text contents of the variable in the encrypted 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 cannot decrypt the vault file:

```
mbp15:aja2 sean$ ansible-playbook test-vault.yaml

PLAY [Display variable from a vault] *******************************************

TASK [import vault data] *******************************************************
fatal: [localhost]: FAILED! => {"ansible_facts": {}, "ansible_included_var_
files": [], "changed": false, "message": "Attempting to decrypt but no vault secrets found"}
         to retry, use: --limit @/Users/sean/aja2/test-vault.retry

PLAY RECAP *********************************************************************
localhost                  : ok=0    changed=0    unreachable=0    failed=1
```

Note the error message: "Attempting to decrypt but no vault secrets found."

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 permits the user to specify multiple passwords, allowing a playbook to use several vault files with different passwords. This is accomplished with the new-in-2.4 option --vault-id.

NOTE   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 Ansible 2.4 or newer. 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 addition, 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 password. 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 authorized users, and consider making the password file hidden and putting it in a directory 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 `vault2.yml` file using the password file:

```
mbp15:aja2 sean$ ansible-vault view --vault-password-file=~/vault-pass.txt vault2.yml
---
vault_test_2: goodbye cruel world
```

Now run the playbook `test-vault.yaml` using the password file:

```
mbp15:aja2 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 monitoring 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. However, Junos prior to version 15.1 used MD5 for password hashing. MD5 is no longer 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_password` 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 difficult to see the connection.

For this second example, we use a different approach, one proposed by Ansible as a best practice (see the link in the References section at the end of the chapter). We modify one of our `group_vars` entries, creating a vault file that Ansible opens automatically, thus eliminating the need for an `include_vars` task. In addition, our Jinja2 template references a variable that is in a plain-text data file, but that variable in turn references a vault-encrypted variable in a file that is easy to locate, because it is in the same directory as the plain-text variables file. Over the next few pages we will update our data files to illustrate this approach.

There is a downside to this approach: we need to provide the vault password even for playbooks that do not use the vault data, because Ansible needs to read all `group_vars` data whether or not the playbook references any specific variable, including the 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 lets you create a subdirectory in `group_vars` for each group, and place multiple files containing group-specific variables in the group's directory. When running a playbook,

Ansible loads the variables from all the files in the group's subdirectory. This is the approach used for this example.

TIP   Ansible supports a similar option for individual device variables – you can create a subdirectory within host_vars for a device, and Ansible reads the data from all files in the device's subdirectory. We do not show an example with a host data directory, but you may find this feature useful if you need to encrypt host-specific data.

This example assumes all devices across the environment share the same root and management account passwords, and thus we 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 corresponds to the *all* group). Move the all.yaml variables file into the new directory, renaming it to vars.yaml:

```
mbp15:aja2 sean$ cd group_vars/

mbp15:group_vars sean$ ls -1
all.yaml
boston.yaml
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 following new (boldfaced) lines:

```
---
ansible_python_interpreter: /usr/local/bin/python
user_data_path: "{{ '~/ansible' | expanduser }}"
root_hash: "{{ vault_root_hash }}"
monitor_hash: "{{ vault_monitor_hash }}"
```

Observe that these variables, in a plain-text variables file, reference other variables. The two new *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. If you have a mix of Junos versions, particularly versions 12.3 or earlier, copy the root password hash from a device running the oldest Junos 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@bilbo> show configuration system root-authentication
encrypted-password "$1$7as5CZnA$Xc1QTe5dW2ph8Y59l8.0j1"; ## 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: "$1$7as5CZnA$Xc1QTe5dW2ph8Y59l8.0j1"
```

On one of your test devices (preferably one running the oldest version of Junos in your environment), 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@bilbo> configure
Entering configuration mode

{master:0}[edit]
sean@bilbo# set system login user monitor authentication plain-text-password
New password:
Retype new password:

{master:0}[edit]
sean@bilbo# show system login user monitor
authentication {
    encrypted-password "$1$ZAduRAJ9$42vsgW1i0kuZcPYww46Xq1"; ## SECRET-DATA
}
## Warning: missing mandatory statement(s): 'class'

{master:0}[edit]
sean@bilbo# rollback
load complete

{master:0}[edit]
sean@bilbo# exit
Exiting configuration mode
```

Copy that password hash and add it to `group_vars/all/vault.yaml`:

```
---
vault_root_hash: "$1$7as5CZnA$Xc1QTe5dW2ph8Y59l8.0j1"
vault_monitor_hash: "$1$ZAduRAJ9$42vsgW1i0kuZcPYww46Xq1"
```

Save the `vault.yaml` file, then use `ansible-vault encrypt` to encrypt the file:

```
mbp15:aja2 sean$ ansible-vault encrypt --vault-password-file=~/vault-pass.txt group_vars/all/vault.yaml
Encryption successful
```

If you view the `vault.yaml` file now it should be encrypted.

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 for discussion. Only the relevant portion of the file is shown; do *not* delete lines not shown below.)

```
...
15|system {
16|    host-name {{ inventory_hostname }};
17|    root-authentication {
18|        encrypted-password "{{ root_hash }}";
19|    }
20|    login {
21|        user monitor {
22|            uid 2005;
23|            class read-only;
24|            authentication {
25|                encrypted-password "{{ monitor_hash }}";
26|            }
27|        }
28|        user sean {
29|            uid 2000;
30|            class super-user;
31|            authentication {
32|                ssh-rsa "ssh-rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
33|            }
34|        }
35|    }
...
```

Run the `base-settings.yaml` playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml --ask-vault-pass --limit=aragorn
Vault password:

PLAY [Generate and Install Configuration File] ********************************

TASK [confirm or create configs directory] ***********************************
changed: [aragorn]

TASK [get device facts] ******************************************************
ok: [aragorn]

TASK [display device facts] **************************************************
skipping: [aragorn]

TASK [save device configuration using template] ******************************
changed: [aragorn]

TASK [install generated configuration file onto device] **********************
changed: [aragorn]

TASK [show configuration change] *********************************************
ok: [aragorn] => {
    "config_results.diff_lines": [
        "",
```

```
        "[edit system root-authentication]",
        "-    encrypted-password \"$5$2VhUeUC5$ba4WLLZc8SoifKhetMBN5M16BnIs2KCKLZ90MV5L
6i.\"; ## SECRET-DATA",
        "+    encrypted-password \"$1$7as5CZnA$Xc1QTe5dW2ph8Y59l8.0j1\"; ## SECRET-DATA",
        "[edit system login]",
        "+    user monitor {",
        "+        uid 2005;",
        "+        class read-only;",
        "+        authentication {",
        "+            encrypted-password \"$1$ZAduRAJ9$42vsgW1i0kuZcPYww46Xq1\"; ## SECRET-DATA",
        "+        }",
        "+    }",
        "[edit snmp]",
        "-  description \"vSRX for writing AJA2\";",
        "+  description \"virtual SRX for testing\";"
    ]
}

RUNNING HANDLER [confirm previous commit] ***************************************
ok: [aragorn]

PLAY RECAP *********************************************************************
aragorn                    : ok=6    changed=3    unreachable=0    failed=0
```

Observe that no change to the playbook itself was needed. The additional variables, 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:aja2 sean$ ansible-vault decrypt group_vars/all/vault.yaml
Vault password:
Decryption successful

mbp15:aja2 sean$ cat group_vars/all/vault.yaml
---
vault_root_hash: "$1$7as5CZnA$Xc1QTe5dW2ph8Y59l8.0j1"
vault_monitor_hash: "$1$ZAduRAJ9$42vsgW1i0kuZcPYww46Xq1"
```

If you do not wish to decrypt the vault, remember to use either the --ask-vault-pass or the --vault-password-file options 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/playbooks_best_practices.
html#variables-and-vaults

Junos hashing algorithms by version:
https://kb.juniper.net/InfoCenter/index?page=content&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; they 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 automatically looks for roles in a *roles* subdirectory within the playbook'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 discuss the following subdirectories, but Ansible supports a few others:

- *tasks*: Tasks that 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 the examples).

The *templates* subdirectory should contain one or more Jinja2 templates. The templates' filenames are not dictated by Ansible; the author suggests using descriptive names for the template files.

Create directory `~/aja2/roles` to contain the roles we create in this chapter:

```
mbp15:aja2 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 create the playbook that uses the role.

It is the author's experience that, at least initially, a role and playbook are developed in parallel, with a lot of switching back-and-forth between role and playbook 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 information 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 settings across all our devices, but that we have different SNMP community names for managing firewalls versus switches. The role uses different Jinja2 templates for the common settings and the different communities.

To start a role for SNMP settings, create directory `~/aja2/roles/snmp`:

```
mbp15:aja2 sean$ mkdir roles/snmp
```

Our SNMP role needs tasks and templates; create the respective subdirectories:

```
mbp15:aja2 sean$ mkdir roles/snmp/tasks
mbp15:aja2 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 "{{ aja2_host.snmp.description }}";
    location "{{ aja2_host.snmp.location }}";
    contact "netadmin@aja.com";
```

```
    {# ensure there will be no community public on a device #}
    delete: community public;
}
```

File `community_fw.j2` *for the firewall management community*:

```
#jinja2: lstrip_blocks: True
snmp {
    replace:
    community aja2_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 aja2_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 configuration 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 several template files, the fragments need to be assembled into a single configuration file before installing the assembled file on the device. We put the assembly step in playbook that uses the role. For this to work, we need a "build" subdirectory for each device that contains the configuration fragments to be assembled for that device.

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 reference 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 needed only for switches.

The role should be complete. Do a quick check that you have the following files and directories for the role:

```
mbp15:aja2 sean$ tree roles/snmp/
roles/snmp/
├── tasks
│   ├── main.yml
└── templates
    ├── community_fw.j2
    ├── community_sw.j2
    └── snmp.j2

2 directories, 4 files
```

A Playbook for the SNMP Role

Now create playbook `snmp-settings.yaml` in your `~/aja2` directory (line numbers added 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|    connection_settings:
17|      host: "{{ ansible_host }}"
18|      timeout: 120
19|
20|  pre_tasks:
21|    - name: confirm or create device config directory
22|      file:
23|        path: "{{ config_assemble }}"
24|        state: directory
25|
26|    - name: delete previous build directory
27|      file:
28|        path: "{{ config_assemble_build }}"
29|        state: absent
30|
31|    - name: create build directory
32|      file:
33|        path: "{{ config_assemble_build }}"
34|        state: directory
35|
36|  tasks:
37|    - name: assemble config fragments
38|      assemble:
39|        src: "{{ config_assemble_build }}"
40|        dest: "{{ config_file }}"
41|      notify: install config onto device
42|
43|  handlers:
44|    - name: install config onto device
45|      juniper_junos_config:
46|        provider: "{{ connection_settings }}"
47|        src: "{{ config_file }}"
48|        load: replace
49|        confirmed: 5
50|        diff: yes
51|        ignore_warning: yes
52|        comment: "playbook {{ playbook_name }}, commit confirmed"
53|      notify: confirm commit
54|
55|    - name: confirm commit
56|      juniper_junos_config:
57|        provider: "{{ connection_settings }}"
58|        commit: yes
59|        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 our new snmp role (line 7). This addition causes the play to, in essence, import the variables and execute the tasks defined in the role.

NOTE   Yes, the Galaxy modules we have been using (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, use the following command to locate the directory where `ansible-galaxy` installed the role, and look in the role's `Juniper.junos/library` subdirectory.

```
mbp15:aja2 sean$ sudo ansible-galaxy info Juniper.junos | grep 'path:'
        path: [u'/etc/ansible/roles']
```

Lines 11–18 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 20 introduces something new, a `pre_tasks` section of the play. Ansible imports a role and executes the role's tasks *before* it executes the tasks defined in the `tasks` section of the play that imports the role. However, we need to create the directories 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 21–34 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 37–41 define the task that assembles the configuration fragments into a single configuration file, then notifies the handler that will install that configuration file onto a Junos device. Note the references to variables defined on lines 13 and 14 of the playbook.

Finally, lines 43–59 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 one handler can notify another handler to run (line 53).

Now let's run the playbook. As the playbook runs, observe the order of the different tasks; you can see how the play's `pre_tasks` run first, followed by the `snmp` role's tasks, and finally the play's `tasks` and `handlers`. Also notice that the role's tasks are prefixed with the role's name, for example, `snmp : common snmp settings`.

```
mbp15:aja2 sean$ ansible-playbook snmp-settings.yaml

PLAY [Generate and Install Configuration File] *********************************

TASK [confirm or create device config directory] ******************************
changed: [bilbo]
changed: [aragorn]

TASK [delete previous build directory] ****************************************
ok: [aragorn]
ok: [bilbo]
```

```
TASK [create build directory] ***************************************************
changed: [bilbo]
changed: [aragorn]

TASK [snmp : common snmp settings] **********************************************
changed: [bilbo]
changed: [aragorn]

TASK [snmp : firewall community] ************************************************
skipping: [bilbo]
changed: [aragorn]

TASK [snmp : switch community] **************************************************
skipping: [aragorn]
changed: [bilbo]

TASK [assemble config fragments] ************************************************
changed: [bilbo]
changed: [aragorn]

RUNNING HANDLER [install config onto device] ************************************
changed: [aragorn]
changed: [bilbo]

RUNNING HANDLER [confirm commit] ************************************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *********************************************************************
aragorn                    : ok=8    changed=6    unreachable=0    failed=0
bilbo                      : ok=8    changed=6    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:aja2 sean$ tree ~/ansible/config/
/Users/sean/ansible/config/
├── aragorn
│   ├── build
│   │   ├── snmp.conf
│   │   └── snmp_community_fw.conf
│   └── snmp.conf
└── bilbo
    ├── build
    │   ├── snmp.conf
    │   └── snmp_community_sw.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`).

NOTE   If you are currently thinking "we could have written this playbook without creating 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 little in this playbook that is specific to SNMP information. The 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 assigned 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` value):

```
---
- name: install config onto device
  juniper_junos_config:
    provider: "{{ connection_settings }}"
    src: "{{ config_file }}"
    load: replace
    confirmed: "{{ confirm_time }}"
    diff: yes
    ignore_warning: yes
    comment: "playbook {{ playbook_name }}, commit confirmed {{ confirm_time }}"
  notify: confirm commit
```

```
- name: confirm commit
  juniper_junos_config:
    provider: "{{ connection_settings }}"
    commit: yes
    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 are new, used to replace static values later in the variables file or in the handlers file. The remaining variables were copied from the `vars` section of the playbook.

```
---
commit_timeout: 120
confirm_time: 10
config_assemble: "{{ user_data_path }}/config/{{ inventory_hostname }}"
config_assemble_build: "{{ config_assemble }}/build"
connection_settings:
  host: "{{ ansible_host }}"
  timeout: "{{ commit_timeout }}"
```

Now delete the `pre_tasks` and `handlers` sections of the `snmp-settings.yaml` playbook, and delete the three 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 discuss momentarily). The playbook now looks like this:

```
 1|---
 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 assembled snmp.conf files from the last playbook run.

Run the playbook again:

```
mbp15:aja2 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: [aragorn]

TASK [config_setup_commit : delete previous build directory] *******************
changed: [bilbo]
changed: [aragorn]

TASK [config_setup_commit : create build directory] ****************************
changed: [bilbo]
changed: [aragorn]

TASK [snmp : common snmp settings] *********************************************
changed: [bilbo]
changed: [aragorn]

TASK [snmp : firewall community] **********************************************
skipping: [bilbo]
changed: [aragorn]

TASK [snmp : switch community] ************************************************
skipping: [aragorn]
changed: [bilbo]

TASK [assemble config fragments] **********************************************
changed: [bilbo]
changed: [aragorn]

RUNNING HANDLER [config_setup_commit : install config onto device] *************
changed: [aragorn]
changed: [bilbo]

RUNNING HANDLER [config_setup_commit : confirm commit] ************************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *******************************************************************
aragorn                    : ok=8    changed=6    unreachable=0    failed=0
bilbo                      : ok=8    changed=6    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 before 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` (specifically, the handlers) 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/aja2/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 another 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
{% set model = junos.model.lower() %}
{% set personality = junos.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 aja2.com;
    domain-search [ aja2.com aja2.net ];
    replace:
    name-server {
      {% for server in aja2_host.dns_servers %}
        {{ server }};
      {% endfor %}
    }
    services {
        delete: ftp;
        netconf {
            ssh {
                connection-limit {{ max_ssh }};
                rate-limit {{ max_ssh }};
            }
        }
        ssh {
            connection-limit {{ max_ssh }};
            rate-limit {{ max_ssh }};
        }
        delete: telnet;
        delete: web-management;
    }
    replace:
    ntp {
      {% for ntp in aja2_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|       juniper_junos_facts:
19|         provider: "{{ connection_settings }}"
20|
21|   tasks:
22|     - name: assemble config fragments
23|       assemble:
24|         src: "{{ config_assemble_build }}"
25|         dest: "{{ config_file }}"
26|       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 bassettings.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.

NOTE   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 device facts? Do we run the fact gathering in both roles, duplicating code and 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:aja2 sean$ ansible-playbook system-settings.yaml

PLAY [Generate and Install Configuration File] *********************************

TASK [get device facts] *******************************************************
ok: [aragorn]
ok: [bilbo]

TASK [config_setup_commit : confirm or create device config directory] *********
ok: [bilbo]
ok: [aragorn]

TASK [config_setup_commit : delete previous build directory] *******************
changed: [bilbo]
changed: [aragorn]
```

```
TASK [config_setup_commit : create build directory] ***************************
changed: [aragorn]
changed: [bilbo]

TASK [system : system settings] **********************************************
changed: [aragorn]
changed: [bilbo]

TASK [assemble config fragments] *********************************************
changed: [bilbo]
changed: [aragorn]

RUNNING HANDLER [config_setup_commit : install config onto device] ************
changed: [aragorn]
changed: [bilbo]

RUNNING HANDLER [config_setup_commit : confirm commit] ***********************
ok: [aragorn]
ok: [bilbo]

PLAY RECAP *******************************************************************
aragorn                    : ok=8    changed=5    unreachable=0    failed=0
bilbo                      : ok=8    changed=5    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 important, as they do not depend on each other.) Also update the two playbook variables 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|      juniper_junos_facts:
20|        provider: "{{ connection_settings }}"
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
```

In order for the playbook to confirm the commit you need to roll back the last two changes on some of 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 *aragorn*; observe the differences between the devices in the output for the two handlers. Run the playbook:

```
mbp15:aja2 sean$ ansible–playbook all–settings.yaml

PLAY [Generate and Install Configuration File] **********************************

TASK [get device facts] ********************************************************
ok: [aragorn]
ok: [bilbo]

TASK [config_setup_commit : confirm or create device config directory] *********
ok: [bilbo]
ok: [aragorn]

TASK [config_setup_commit : delete previous build directory] *******************
changed: [bilbo]
changed: [aragorn]

TASK [config_setup_commit : create build directory] ****************************
changed: [bilbo]
changed: [aragorn]

TASK [snmp : common snmp settings] *********************************************
changed: [bilbo]
changed: [aragorn]

TASK [snmp : firewall community] **********************************************
skipping: [bilbo]
changed: [aragorn]

TASK [snmp : switch community] ************************************************
skipping: [aragorn]
changed: [bilbo]

TASK [system : system settings] ***********************************************
changed: [bilbo]
changed: [aragorn]

TASK [assemble config fragments] **********************************************
changed: [aragorn]
changed: [bilbo]
```

```
RUNNING HANDLER [config_setup_commit : install config onto device] *************
ok: [aragorn]
changed: [bilbo]

RUNNING HANDLER [config_setup_commit : confirm commit] *************************
ok: [bilbo]

PLAY RECAP *********************************************************************
aragorn                    : ok=9    changed=6    unreachable=0    failed=0
bilbo                      : ok=10   changed=7    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 configuration 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 general 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.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|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|
14|  tasks:
15|    – name: get Junos version
16|      juniper_junos_rpc:
17|        rpcs:
18|          – get-software-information
19|        provider: "{{ connection_settings }}"
20|        formats: text
21|      register: jversion
22|      retries: 2
23|      delay: 15
24|      until: jversion is success
25|
26|    – name: display Junos version
27|      debug:
28|        var: jversion
```

Most of the playbook's contents are already familiar. We used Juniper's `juniper_ju-nos_rpc` module in Chapter 5. The RPC `get-software-information` is the equivalent of the Junos CLI command "show version."

Lines 22–24 are new. These lines together tell Ansible how it should retry the task should it fail. Observe that these lines are indented to the level of the task; these are Ansible arguments to the task, not arguments to the `juniper_junos_rpc` module.

Line 22, 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 23, the `delay` argument, specifies how many seconds to wait between a failure and the (next) retry.

Line 24, 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 (in other words, the loop repeats while the condition is false). The condition shown here, `jversion is success`, uses the `success` test on the `jversion` registered variable to return *true* if the task succeeded, or *false* if the task failed. So, line 24 tells Ansible to repeat the task, the call to the `juniper_junos_rpc` module, until either it succeeds (the `jversion is success` condition returns *true*) or the maximum number of `retries` have been attempted.

NOTE   In version 2.5, Ansible changed the syntax for using the `success` test. If you are using Ansible 2.4 or earlier, line 24 should be written like a filter, using the pipe character (`|`) rather than using the keyword `is`:

```
until: jversion | success
```

TIP   Ansible also has a `failure` test that returns *true* if the task failed or *false* if the task succeeded (the opposite of the `success` test used above).

Disconnect one of your test devices – the author disconnected his switch *bilbo* – and run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-version.yaml

PLAY [Get Junos version] *********************************************************

TASK [get Junos version] *********************************************************
FAILED - RETRYING: get Junos version (2 retries left).
ok: [aragorn]
FAILED - RETRYING: get Junos version (1 retries left).
fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "Unable to make a PyEZ connection:
ConnectTimeoutError(198.51.100.5)"}

TASK [display Junos version] *****************************************************
ok: [aragorn] => {
    "jversion": {
        "attempts": 1,
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "text",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "rpc": "get-software-information",
        "stdout": "\nHostname: aragorn\nModel: vsrx\nJunos: 15.1X49-D90.7\
nJUNOS Software Release [15.1X49-D90.7]\n",
        "stdout_lines": [
            "",
            "Hostname: aragorn",
            "Model: vsrx",
            "Junos: 15.1X49-D90.7",
            "JUNOS Software Release [15.1X49-D90.7]"
        ]
    }
}
        to retry, use: --limit @/Users/sean/aja2/get-version.retry

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

Observe that *aragorn* succeeded, but that *bilbo* failed. During TASK [get Junos version] 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 device). Only after the retries do we see the failure message for *bilbo*:

```
fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "Unable to make a PyEZ connection:
ConnectTimeoutError(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:aja2 sean$ ansible-playbook get-version.yaml

PLAY [Get Junos version] *********************************************************

TASK [get Junos version] *********************************************************
ok: [aragorn]
FAILED - RETRYING: get Junos version (2 retries left).
ok: [bilbo]

TASK [display Junos version] *****************************************************
ok: [bilbo] => {
    "jversion": {
        "attempts": 2,
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "text",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "rpc": "get-software-information",
        "stdout": "\
nfpc0:\n--------------------------------------------------------------------------\nHostname: bilbo\
nModel: ex2200-c-12t-2g\nJUNOS Base OS boot [12.3R12.4]\nJUNOS Base OS Software Suite [12.3R12.4]\
nJUNOS Kernel Software Suite [12.3R12.4]\nJUNOS Crypto Software Suite [12.3R12.4]\nJUNOS Online
Documentation [12.3R12.4]\nJUNOS Enterprise Software Suite [12.3R12.4]\nJUNOS Packet Forwarding
Engine Enterprise Software Suite [12.3R12.4]\nJUNOS Routing Software Suite [12.3R12.4]\nJUNOS Web
Management [12.3R12.4]\nJUNOS FIPS mode utilities [12.3R12.4]\n",
        "stdout_lines": [
            "",
            "fpc0:",
            "--------------------------------------------------------------------------",
            "Hostname: bilbo",
            "Model: ex2200-c-12t-2g",
            "JUNOS Base OS boot [12.3R12.4]",
            "JUNOS Base OS Software Suite [12.3R12.4]",
            "JUNOS Kernel Software Suite [12.3R12.4]",
            "JUNOS Crypto Software Suite [12.3R12.4]",
            "JUNOS Online Documentation [12.3R12.4]",
            "JUNOS Enterprise Software Suite [12.3R12.4]",
            "JUNOS Packet Forwarding Engine Enterprise Software Suite [12.3R12.4]",
            "JUNOS Routing Software Suite [12.3R12.4]",
            "JUNOS Web Management [12.3R12.4]",
            "JUNOS FIPS mode utilities [12.3R12.4]"
        ]
    }
}
ok: [aragorn] => {
    "jversion": {
        "attempts": 1,
        "attrs": null,
        "changed": false,
        "failed": false,
        "format": "text",
        "kwargs": null,
        "msg": "The RPC executed successfully.",
        "rpc": "get-software-information",
        "stdout": "\nHostname: aragorn\nModel: vsrx\nJunos: 15.1X49-D90.7\
nJUNOS Software Release [15.1X49-D90.7]\n",
```

```
        "stdout_lines": [
            "",
            "Hostname: aragorn",
            "Model: vsrx",
            "Junos: 15.1X49-D90.7",
            "JUNOS Software Release [15.1X49-D90.7]"
        ]
    }
}
```

```
PLAY RECAP *******************************************************************
aragorn                        : ok=2    changed=0    unreachable=0    failed=0
bilbo                          : ok=2    changed=0    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*.

Refining the Until Condition

Our get-version.yaml playbook will retry the juniper_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 juniper_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 disconnected for the entire playbook run:

```
fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "Unable to make a PyEZ connection:
ConnectTimeoutError(198.51.100.5)"}
```

The phrase "ConnectTimeoutError" suggests the nature of the problem, which in our test was a disconnected network cable. (If your computer connects directly to the test device, so that disconnecting the test device causes your computer's interface to be down, you might get a "ConnectRefusedError" error instead.) 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, which may return "ConnectTimeoutError" or "ConnectRefusedError" messages.

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.yaml playbook to avoid retrying the task if we get an authentication failure, while still retrying the task for any other failure:

```
...
24|       until: (jversion is 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 is success)` is the same condition we had before, applying the `success` test to the registered variable `jversion` and returning a Boolean value indicating whether or not the task succeeded. The parentheses ensure this expression 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 the value -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 containing "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 retries left.

TIP   To help understand the results of `find()`, run the following simple playbook 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.

Let's run our updated get-version.yaml playbook with one test device disconnected; the author unplugged the network cable for *bilbo*. This run should look similar to what we saw previously (results here edited for length):

```
mbp15:aja2 sean$ ansible-playbook get-version.yaml

PLAY [Get Junos version] ********************************************************

TASK [get Junos version] ********************************************************
ok: [aragorn]
FAILED - RETRYING: get Junos version (2 retries left).
FAILED - RETRYING: get Junos version (1 retries left).
fatal: [bilbo]: FAILED! => {"attempts": 2, "changed": false, "msg": "Unable to make a PyEZ connection:
ConnectTimeoutError(198.51.100.5)"}

TASK [display Junos version] ****************************************************
ok: [aragorn] => {
...
}
        to retry, use: --limit @/Users/sean/aja2/get-version.retry

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

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 behavior changes. The author disabled his account on his *aragorn* firewall, as follows:

```
root@aragorn> configure
Entering configuration mode

[edit]
root@aragorn# deactivate system login user sean

[edit]
root@aragorn# show | compare
[edit system login]
!    inactive: user sean { ... }
```

```
[edit]
root@aragorn# 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@aragorn>
```

Run the playbook again (results edited for length):

```
mbp15:aja2 sean$ ansible-playbook get-version.yaml

PLAY [Get Junos version] ********************************************************

TASK [get Junos version] ********************************************************
fatal: [aragorn]: FAILED! => {"attempts": 1, "changed": false, "msg": "Unable to make a PyEZ
connection: ConnectAuthError(192.0.2.10)"}
ok: [bilbo]

TASK [display Junos version] ****************************************************
ok: [bilbo] => {
...
}
          to retry, use: --limit @/Users/sean/aja2/get-version.retry

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

The attempt to communicate with *aragorn* failed, as expected, but notice there were no retries – the test for "ConnectAuthError" worked!

Repeating a Task Based on a List

In Chapter 8 we talked about a *for* loop in a Jinja2 template, and showed an example using a list of DNS server IP addresses. This section shows how to do something similar in a playbook using Ansible's `loop` option.

NOTE   The `loop` option was introduced in Ansible 2.5. Earlier versions of Ansible used a set of *with_X* looping options, such as `with_items`. If you are using Ansible 2.4 or earlier, you can replace `loop` with `with_items` in the examples in this chapter. See the References section for a link to Ansible's loop documentation.

The examples in this section use XML data from Junos devices. (We discussed XML and XPath in Chapter 5.) In many of our playbooks, we request data in text format because it is easier for humans to read. The playbook in this section will process the data, and XML is easier to work with for automated processing.

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 intention of using the name of the neighbor to create a description of the local interface 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 descriptions 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 configuration for the device's "uplink" interfaces similar to this:

```
interfaces {
    ge-0/1/0 {
        description "to device aragorn port ge-0/0/0";
    }
    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-0/1/0.0         -                   4c:96:14:0c:de:40 ge-0/0/0       aragorn
ge-0/1/1.0         -                   78:fe:3d:3d:f6:40 ge-0/1/1.0     frodo
ge-0/0/8.0         -                   7c:25:86:c1:af:07 ge-0/0/11      elrond
```

The uplink interfaces are ge-0/1/0 and ge-0/1/1, so the systems *aragorn* and *frodo* are of interest. The device *elrond* connected to ge-0/0/8 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 `aja2_host` dictionary already defined in that file. The author is updating his `host_vars/bilbo.yaml` file as follows (boldfaced lines):

```
---
ansible_host: 198.51.100.5
aja2_host:
  dns_servers:
    - 8.8.4.4
    - 8.8.8.8
    - 198.51.100.101
  snmp:
    description: EX2200-C for testing
    location: Sean's home office
  uplinks:
    - ge-0/1/0
    - ge-0/1/1
```

The two LLDP playbooks we will create in the following pages repeat certain tasks for each element (interface) in the `aja2_host.uplinks` list.

LLDP as XML

Let's start by getting a feel for what the LLDP data looks like in XML format. On the author's switch *bilbo*, the LLDP data is:

```
sean@bilbo> show lldp neighbors | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.3R12/junos">
    <lldp-neighbors-information junos:style="brief">
        <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>4c:96:14:0c:de:40</lldp-remote-chassis-id>
            <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>
            <lldp-remote-system-name>aragorn</lldp-remote-system-name>
        </lldp-neighbor-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-0/0/8.0</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>7c:25:86:c1:af:07</lldp-remote-chassis-id>
            <lldp-remote-port-description>ge-0/0/11</lldp-remote-port-description>
            <lldp-remote-system-name>elrond</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 there is an element `lldp-neighbor-information` for each neighbor. Within each `lldp-neighbor-information` element is a series of elements describing the neighbor or the local interface to which the neighbor connects.

We want the `lldp-remote-system-name` elements, but only for the desired local interfaces. 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">
    <rpc>
```

```
    <get-lldp-neighbors-information>
    </get-lldp-neighbors-information>
</rpc>
<cli>
    <banner>{master:0}</banner>
</cli>
</rpc-reply>
```

We can specify an interface with the command, either on the CLI (shown here) or as part of an RPC request (used later in this chapter):

```
sean@bilbo> show lldp neighbors interface ge-0/1/0 | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/12.3R12/junos">
    <lldp-neighbors-information junos:style="detail">
        <lldp-neighbor-information>
            <lldp-index>1</lldp-index>
            <lldp-ttl>120</lldp-ttl>
            <lldp-timemark>Fri Feb 19 05:42:07 2016</lldp-timemark>
            <lldp-age>23</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>527</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>4c:96:14:0c:de:40</lldp-remote-chassis-id>
            <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>
            <lldp-remote-port-id>510</lldp-remote-port-id>
            <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>
            <lldp-remote-system-name>aragorn</lldp-remote-system-name>
...
        </lldp-neighbor-information>
    </lldp-neighbors-information>
    <cli>
        <banner>{master:0}</banner>
    </cli>
</rpc-reply>
```

We get a lot more detail about the neighbor attached to the specified interface. However, we get neighbor information only for the specified local interface (there is only one `lldp-neighbor-information` element).

The RPC changes name when called for a single interface:

```
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">
    <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>
```

Observe the RPC is `get-lldp-interface-neighbors-information`, with argument `interface-name` to specify the interface.

LLDP by Interface – Get LLDP Interfaces Version 1

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 information
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|
14|  tasks:
15|    - name: get lldp neighbor table
16|      juniper_junos_rpc:
17|        provider: "{{ connection_settings }}"
18|        rpcs: get-lldp-neighbors-information
19|        format: xml
20|      register: lldp
21|
22|    - name: display lldp neighbor data
23|      debug:
24|        var: lldp.stdout_lines
```

Lines 15–20 ask the `juniper_junos_rpc` module to run the `get-lldp-neighbors-infor-mation` RPC and register the results in variable `lldp`.

Lines 22–24 display the LLDP results from the registered variable.

Run the playbook on your test device(s) which have LLDP data:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-interface.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] ********************************************

TASK [get lldp neighbor table] ********************************************
ok: [bilbo]

TASK [display lldp neighbor data] ********************************************
ok: [bilbo] => {
    "lldp.stdout_lines": [
        "<lldp-neighbors-information style=\"brief\">",
        "  <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>4c:96:14:0c:de:40</lldp-remote-chassis-id>",
        "    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>",
        "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
        "  </lldp-neighbor-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-0/0/8.0</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>7c:25:86:c1:af:07</lldp-remote-chassis-id>",
"      <lldp-remote-port-description>ge-0/0/11</lldp-remote-port-description>",
"      <lldp-remote-system-name>elrond</lldp-remote-system-name>",
"    </lldp-neighbor-information>",
"</lldp-neighbors-information>"
    ]
}

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

Now let's update the playbook to gather LLDP data for a single interface. Initially we specify a single interface as part of the playbook; in the next section of this chapter, we add the loop to iterate over our list of uplink interfaces.

Modify the `get-lldp-interface.yaml` playbook by changing or adding the boldfaced lines:

```
...
14|  tasks:
15|    - name: get lldp neighbor table
16|      juniper_junos_rpc:
17|        provider: "{{ connection_settings }}"
18|        rpcs: get-lldp-interface-neighbors-information
19|        kwargs:
20|          interface_name: ge-0/1/0
21|        format: xml
22|      register: lldp
...
```

Line 18 changes the name of the RPC call.

Lines 19–20 provide the argument to the RPC that specifies the interface name. Notice that the hyphen in the RPC argument `interface-name` shown above is replaced with an underscore in the Ansible playbook. This replacement does not seem to be required in newer versions of the `juniper_junos_rpc` module but was necessary in earlier versions.

Run the playbook again. There should be data for a single interface, ge-0/1/0.

```
mbp15:aja2 sean$ ansible-playbook get-lldp-interface.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] *******************************************

TASK [get lldp neighbor table] ************************************************
ok: [bilbo]
```

```
TASK [display lldp neighbor data] **********************************************
ok: [bilbo] => {
    "lldp.stdout_lines": [
        "<lldp-neighbors-information style=\"detail\">",
        "  <lldp-neighbor-information>",
        "    <lldp-index>1</lldp-index>",
        "    <lldp-ttl>120</lldp-ttl>",
        "    <lldp-timemark>Fri Feb 19 06:11:05 2016</lldp-timemark>",
        "    <lldp-age>3</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>527</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>4c:96:14:0c:de:40</lldp-remote-chassis-id>",
        "    <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>",
        "    <lldp-remote-port-id>510</lldp-remote-port-id>",
        "    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>",
        "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
...
        "  </lldp-neighbor-information>",
        "</lldp-neighbors-information>"
    ]
}

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

Looping Through Interfaces – Get LLDP Interfaces Version 2

Now let's have the playbook loop through the aja2_host.uplinks list we created in our host_vars file(s), executing the juniper_junos_rpc task for each interface on the list. Modify the get-lldp-interface.yaml playbook as shown (boldfaced lines):

```
...
14|  tasks:
15|    - name: get lldp neighbor table
16|      juniper_junos_rpc:
17|        provider: "{{ connection_settings }}"
18|        rpcs: get-lldp-interface-neighbors-information
19|        kwargs:
20|          interface_name: "{{ item }}"
21|        format: xml
22|      loop: "{{ aja2_host.uplinks }}"
23|      register: lldp
24|
25|    - name: display lldp neighbor data
26|      debug:
27|        var: lldp
```

Line 22, the loop option, tells Ansible that the task is a loop and should be repeated once for each element of the list in the aja2_host.uplinks variable. Ansible takes the first element from aja2_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. In other words, Ansible iterates over the `aja2_host.uplinks` list and puts each element in the `item` variable.

Line 20 provides the value of variable `item`, defined by the `loop` construct, to the keyword argument `interface_name`. This causes the task to get LLDP data for a different interface during each iteration of the loop.

Line 27 is changed to display the entire registered output from the "get lldp neighbor table" task as the data's structure changes due to the loop option. This will display a lot of data; we will revise this momentarily to limit the output.

> 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.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-interface.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] *******************************************

TASK [get lldp neighbor table] ************************************************
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)

TASK [display lldp neighbor data] *********************************************
ok: [bilbo] => {
    "lldp": {
        "changed": false,
        "msg": "All items completed",
        "results": [
            {
                "_ansible_ignore_errors": null,
                "_ansible_item_label": "ge-0/1/0",
                "_ansible_item_result": true,
                "_ansible_no_log": false,
                "_ansible_parsed": true,
                "attrs": null,
                "changed": false,
                "failed": false,
                "format": "xml",
...
                "item": "ge-0/1/0",
                "kwargs": {
                    "interface_name": "ge-0/1/0"
                },
...
                "stdout_lines": [
                    "<lldp-neighbors-information style=\"detail\">",
                    "  <lldp-neighbor-information>",
                    "    <lldp-index>1</lldp-index>",
                    "    <lldp-ttl>120</lldp-ttl>",
```

```
                  "      <lldp-timemark>Fri Feb 19 07:21:02 2016</lldp-timemark>",
                  "      <lldp-age>24</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>527</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>4c:96:14:0c:de:40</lldp-remote-chassis-id>",
                  "      <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-
subtype>",
                  "      <lldp-remote-port-id>510</lldp-remote-port-id>",
                  "      <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>",
                  "      <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
...
                  "   </lldp-neighbor-information>",
                  "</lldp-neighbors-information>"
              ]
          },
          {
              "_ansible_ignore_errors": null,
              "_ansible_item_label": "ge-0/1/1",
              "_ansible_item_result": true,
              "_ansible_no_log": false,
              "_ansible_parsed": true,
              "attrs": null,
              "changed": false,
              "failed": false,
              "format": "xml",
...
              "item": "ge-0/1/1",
              "kwargs": {
                  "interface_name": "ge-0/1/1"
              },
...
              "stdout_lines": [
                  "<lldp-neighbors-information style=\"detail\">",
                  "   <lldp-neighbor-information>",
                  "      <lldp-index>4</lldp-index>",
                  "      <lldp-ttl>120</lldp-ttl>",
                  "      <lldp-timemark>Fri Feb 19 07:21:24 2016</lldp-timemark>",
                  "      <lldp-age>8</lldp-age>",
                  "      <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>529</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>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>",
                  "      <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-
subtype>",
                  "      <lldp-remote-port-id>531</lldp-remote-port-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-neighbors-information>"
              ]
          }
```

```
        ]
    }
}
```

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

Notice that the task "get lldp neighbor table" shows it ran twice for *bilbo*, using two different values for `item` (two interface names):

```
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)
```

Perfect!

Now look at the output for the "display lldp neighbor data" task. Because the registered variable `lldp`, created by the "get lldp neighbor table" task, contains the results of multiple iterations of the task, Ansible stores the results for each iteration of the task in a `results` list. Each element in `results` is a dictionary that contains roughly the same fields that we would expect in the registered variable for a similar task without the `loop`, plus a few new fields. Among the new fields in each `result` dictionary is `item`, which contains the contents of the `item` variable during the loop iteration that created that result (we use `item` later in this chapter). Review the output so you understand the structure.

We want to display only the `stdout_lines` field for each element in `lldp.results`. Said differently, we want to iterate over the `lldp.results` list and display the stdout_lines field. Iteration...that sounds like a loop!

Modify the last task of the playbook as follows:

```
...
25|   – name: display lldp neighbor data
26|     debug:
27|       var: item.stdout_lines
28|     loop: "{{ lldp.results }}"
```

Line 28 creates a loop to iterate over the different results.

Line 27 now displays the `stdout_lines` field from the current `item` from the loop.

Because of how Ansible displays loop status and how the `debug` module works in a loop, we will get a lot of extra output when we run this playbook (edited for length in the output below). We'll discuss the reasons and minimize this extra output momentarily.

Run the playbook:

```
mbp15:aja2 sean$ ansible–playbook get–lldp–interface.yaml ––limit=bilbo
```

```
PLAY [Get LLDP neighbor information] ******************************************
```

```
TASK [get lldp neighbor table] **************************************************
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)

TASK [display lldp neighbor data] ***********************************************
ok: [bilbo] => (item={'_ansible_parsed': True, u'changed': False, ..., u'stdout_lines': [u'<lldp-
neighbors-information style="detail">', ..., u'    <lldp-remote-system-name>aragorn</lldp-remote-
system-name>', ..., u'</lldp-neighbors-information>'], '_ansible_ignore_errors': None, '_ansible_no_
log': False}) => {
    "item": {
        "attrs": null,
        "changed": false,
...
        "stdout_lines": [
            "<lldp-neighbors-information style=\"detail\">",
...
            "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
...
            "</lldp-neighbors-information>"
        ]
    },
    "item.stdout_lines": [
        "<lldp-neighbors-information style=\"detail\">",
...
        "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
...
        "</lldp-neighbors-information>"
    ]
}
ok: [bilbo] => (item={'_ansible_parsed': True, u'changed': False, ..., u'stdout_lines': [u'<lldp-
neighbors-information style="detail">', ..., u'    <lldp-remote-system-name>frodo</lldp-remote-
system-name>', ..., u'</lldp-neighbors-information>'], '_ansible_ignore_errors': None, '_ansible_no_
log': False}) => {
    "item": {
        "attrs": null,
        "changed": false,
...
        "stdout_lines": [
            "<lldp-neighbors-information style=\"detail\">",
...
            "    <lldp-remote-system-name>frodo</lldp-remote-system-name>",
...
            "</lldp-neighbors-information>"
        ]
    },
    "item.stdout_lines": [
        "<lldp-neighbors-information style=\"detail\">",
...
        "    <lldp-remote-system-name>frodo</lldp-remote-system-name>",
...
        "</lldp-neighbors-information>"
    ]
}

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

This output of the "display lldp neighbor data" task has been significantly short-

ened in the book. There are two reasons for the extra output.

First, Ansible displays each `item`'s contents as it reports success (or failure) for each iteration through the loop. Labelling each loop iteration lets us see what the loop is working on and monitor the status of playbook operation. This is usually a benefit, as we can see with the "get lldp neighbor table" task where `item` contains just an interface name:

```
TASK [get lldp neighbor table] ****************************************************
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)
```

However, in the "display lldp neighbor data" task, `item` contains the entire results from one iteration of the previous task's loop. That's a lot of data! In fact, it is so much data that it makes it difficult for us to realize the intended benefit of labelling each loop iteration.

```
TASK [display lldp neighbor data] ************************************************
ok: [bilbo] => (item={'_ansible_parsed': True, u'changed': False, ..., u'stdout_lines': [u'<lldp-
neighbors-information style="detail">', ..., u'    <lldp-remote-system-name>aragorn</lldp-remote-
system-name>', ..., u'</lldp-neighbors-information>'], '_ansible_ignore_errors': None, '_ansible_no_
log': False}) => {
...
}
ok: [bilbo] => (item={'_ansible_parsed': True, u'changed': False, ..., u'stdout_lines': [u'<lldp-
neighbors-information style="detail">', ..., u'    <lldp-remote-system-name>frodo</lldp-remote-
system-name>', ..., u'</lldp-neighbors-information>'], '_ansible_ignore_errors': None, '_ansible_no_
log': False}) => {
...
}
```

Fortunately, Ansible offers some loop control options, including a `label` option, that we can use to specify a different (smaller!) identifier for each iteration of the loop.

The second source of extra output is the `debug` module. When called in a loop using its `var` argument and told to display a field from a dictionary, `debug` displays the full dictionary followed by the requested field. Note, for example, how the output shows both the full `item` dictionary and the `item.stdout_lines` field, though our playbook requested only the `item.stdout_lines` field:

```
"item": {
    "attrs": null,
    "changed": false,
...
},
"item.stdout_lines": [
    "<lldp-neighbors-information style=\"detail\">",
...
    "    <lldp-remote-system-name>frodo</lldp-remote-system-name>",
...
    "</lldp-neighbors-information>"
]
```

If we call `debug` with the `msg` argument instead of the `var` argument, `debug` does not

display the extra output.

Modify the last task of the playbook as follows:

```
25|   - name: display lldp neighbor data
26|     debug:
27|       msg: "{{ item.stdout_lines }}"
28|     loop: "{{ lldp.results }}"
29|     loop_control:
30|       label: "{{ item.item }}"
```

Line 27 uses the `msg` argument to the `debug` module instead of the `var` argument.

Lines 29 and 30 add the `loop_control` option `label`, which provides an alternative identifier for each loop iteration. Here we use `item.item`, which is interface name recorded during the "get lldp neighbor table" loop and stored in its `results` list.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-interface.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] *********************************************

TASK [get lldp neighbor table] **************************************************
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)

TASK [display lldp neighbor data] ***********************************************
ok: [bilbo] => (item=ge-0/1/0) => {
    "msg": [
        "<lldp-neighbors-information style=\"detail\">",
        "  <lldp-neighbor-information>",
        "    <lldp-index>4</lldp-index>",
        "    <lldp-ttl>120</lldp-ttl>",
        "    <lldp-timemark>Thu Jan 21 05:47:47 2016</lldp-timemark>",
        "    <lldp-age>14</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>527</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>4c:96:14:0c:de:40</lldp-remote-chassis-id>",
        "    <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>",
        "    <lldp-remote-port-id>510</lldp-remote-port-id>",
        "    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>",
        "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
...
        "  </lldp-neighbor-information>",
        "</lldp-neighbors-information>"
    ]
}
ok: [bilbo] => (item=ge-0/1/1) => {
    "msg": [
        "<lldp-neighbors-information style=\"detail\">",
        "  <lldp-neighbor-information>",
        "    <lldp-index>3</lldp-index>",
        "    <lldp-ttl>120</lldp-ttl>",
```

```
   "    <lldp-timemark>Thu Jan 21 05:48:00 2016</lldp-timemark>",
   "    <lldp-age>7</lldp-age>",
   "    <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>529</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>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>",
   "    <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>",
   "    <lldp-remote-port-id>531</lldp-remote-port-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-neighbors-information>"
   ]
}

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

Much better! Now our loop iterations are identified with only the relevant interface name from item.item, and the debug output includes only the stdout_lines field.

Querying our LLDP Data – Get LLDP Interfaces Version 3

Now let's add an XML XPath query to our get-lldp-interface.yaml playbook to extract only the specific elements we want from our XML LLDP data.

Modify the playbook by removing the "display lldp neighbor data" task and adding the boldfaced lines as follows:

```
...
14| tasks:
15|   - name: get lldp neighbor table
16|     juniper_junos_rpc:
17|       provider: "{{ connection_settings }}"
18|       rpcs: get-lldp-interface-neighbors-information
19|       kwargs:
20|         interface_name: "{{ item }}"
21|       format: xml
22|     loop: "{{ aja2_host.uplinks }}"
23|     register: lldp
24|
25|   - name: get neighbor details
26|     xml:
27|       xmlstring: "{{ item }}"
28|       xpath: //lldp-remote-system-name | //lldp-remote-port-description
29|       content: text
30|     loop: "{{ lldp.results | map(attribute='stdout') | list }}"
31|     register: neighbors
32|
33|   - name: show neighbor details
34|     debug:
35|       var: neighbors
```

Lines 25–31 call the xml module and register the results in variable neighbors.

Line 30 uses the `loop` option to iterate over a list of XML results extracted from the `lldp` variable registered by the "get lldp neighbor table" task. We use the `map()` and `list` filters we saw in Chapter 9 to extract only the desired field, in this case the `stdout` field, from the original `lldp.results` data and create a new list. This new list is presented to `loop`, which means for each iteration through the loop the `item` variable will contain a string with XML data for one interface.

For the reader's consideration: why does line 30 extract the `stdout` field, while the previous version of this playbook displayed the `stdout_lines` field?

Line 27, the `xmlstring` argument, provides a text string representing the XML data to query. We provide the `item` variable from the loop, letting us query the XML data for a single interface with each iteration of the loop.

Line 28 is the XPath path expression. We do not need to worry about multiple matches here, as the XML data is for a single LLDP neighbor, so we do not need a predicate. However, we use the pipe ("`|`") to return two different elements from our data set.

Lines 33–35 display the results from the XPath operations, contained in the `neighbors` variable registered by task "get neighbor details." From Chapter 5 we know that we probably only care about the `matches` field, which contains the XPath query results, but let's start by viewing the complete `neighbors` data. This will give us a lot of output (edited for length below) but we will reduce it momentarily.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-interface.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] ***************************************************
****

TASK [get lldp neighbor table] *********************************************************
****
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)

TASK [get neighbor details] ****************************************************
ok: [bilbo] => (item=<lldp-neighbors-information style="detail">
  <lldp-neighbor-information>
    <lldp-index>4</lldp-index>
    <lldp-ttl>120</lldp-ttl>
    <lldp-timemark>Thu Jan 21 07:17:01 2016</lldp-timemark>
    <lldp-age>5</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>527</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>4c:96:14:0c:de:40</lldp-remote-chassis-id>
    <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>
    <lldp-remote-port-id>510</lldp-remote-port-id>
    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>
    <lldp-remote-system-name>aragorn</lldp-remote-system-name>
```

```
...
  </lldp-neighbor-information>
</lldp-neighbors-information>
)
ok: [bilbo] => (item=<lldp-neighbors-information style="detail">
  <lldp-neighbor-information>
    <lldp-index>3</lldp-index>
    <lldp-ttl>120</lldp-ttl>
    <lldp-timemark>Thu Jan 21 07:17:05 2016</lldp-timemark>
    <lldp-age>7</lldp-age>
    <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>529</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>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>
    <lldp-remote-port-id-subtype>Locally assigned</lldp-remote-port-id-subtype>
    <lldp-remote-port-id>531</lldp-remote-port-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-neighbors-information>
)

TASK [show neighbor details] ****************************************************
ok: [bilbo] => {
    "neighbors": {
        "changed": false,
        "msg": "All items completed",
        "results": [
            {
...
            "changed": false,
            "count": 2,
            "failed": false,
...
            "item": "<lldp-neighbors-information style=\"detail\">\n  <lldp-neighbor-information>\n
<lldp-index>4</lldp-index>\n    <lldp-ttl>120</lldp-ttl>\n    <lldp-timemark>Thu Jan 21 07:17:01
2016</lldp-timemark>\n    <lldp-age>5</lldp-age>\n    <lldp-local-interface>ge-0/1/0.0</lldp-local-
interface>\n    <lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>\n    <lldp-
local-port-id>527</lldp-local-port-id>\n    <lldp-local-port-ageout-count>0</lldp-local-port-ageout-
count>\n    <lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>\n
<lldp-remote-chassis-id>4c:96:14:0c:de:40</lldp-remote-chassis-id>\n    <lldp-remote-port-id-
subtype>Locally assigned</lldp-remote-port-id-subtype>\n    <lldp-remote-port-id>510</lldp-remote-
port-id>\n    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>\n    <lldp-remote-
system-name>aragorn</lldp-remote-system-name>\n    <lldp-system-description>\n
<lldp-remote-system-description>Juniper Networks, Inc. vsrx internet router, kernel JUNOS
15.1X49-D90.7, Build date: 2017-04-29 06:17:35 UTC Copyright (c) 1996-2017 Juniper Networks, Inc.</
lldp-remote-system-description>\n    </lldp-system-description>\n    <lldp-remote-system-capabilities-
supported>Bridge Router</lldp-remote-system-capabilities-supported>\n    <lldp-remote-system-
capabilities-enabled>Bridge Router</lldp-remote-system-capabilities-enabled>\n    <lldp-remote-
management-address-type>IPv4</lldp-remote-management-address-type>\n
<lldp-remote-management-address>198.51.100.1</lldp-remote-management-address>\n    <lldp-remote-
management-address-port-id>510</lldp-remote-management-address-port-id>\n    <lldp-remote-management-
address-sub-type>1</lldp-remote-management-address-sub-type>\n    <lldp-remote-management-address-
interface-subtype>ifIndex(2)</lldp-remote-management-address-interface-subtype>\n
<lldp-remote-management-addr-oid>1.3.6.1.2.1.31.1.1.1.1.510</lldp-remote-management-addr-oid>\n
```

```
<lldp-remote-org-def-info-oui>0.12.f</lldp-remote-org-def-info-oui>\n    <lldp-remote-org-def-info-
subtype>1</lldp-remote-org-def-info-subtype>\n    <lldp-remote-org-def-info-index>1</lldp-remote-org-
def-info-index>\n    <lldp-remote-org-def-info>036C1D0000</lldp-remote-org-def-info>\n    <lldp-
remote-org-def-info-oui>0.12.f</lldp-remote-org-def-info-oui>\n
<lldp-remote-org-def-info-subtype>3</lldp-remote-org-def-info-subtype>\n    <lldp-remote-org-def-
info-index>2</lldp-remote-org-def-info-index>\n    <lldp-remote-org-def-info>0100000000</lldp-remote-
org-def-info>\n    <lldp-remote-org-def-info-oui>0.12.f</lldp-remote-org-def-info-oui>\n    <lldp-
remote-org-def-info-subtype>4</lldp-remote-org-def-info-subtype>\n
<lldp-remote-org-def-info-index>3</lldp-remote-org-def-info-index>\n    <lldp-remote-org-def-
info>05EA</lldp-remote-org-def-info>\n </lldp-neighbor-information>\n</
lldp-neighbors-information>\n",
            "matches": [
                {
                    "lldp-remote-port-description": "ge-0/0/0"
                },
                {
                    "lldp-remote-system-name": "aragorn"
                }
            ],
...
        },
        {
...
            "changed": false,
            "count": 2,
            "failed": false,
...
            "item": "<lldp-neighbors-information style=\"detail\">\n <lldp-neighbor-information>\n
<lldp-index>3</lldp-index>\n    <lldp-ttl>120</lldp-ttl>\n    <lldp-timemark>Thu Jan 21 07:17:05
2016</lldp-timemark>\n    <lldp-age>7</lldp-age>\n    <lldp-local-interface>ge-0/1/1.0</lldp-local-
interface>\n    <lldp-local-parent-interface-name>-</lldp-local-parent-interface-name>\n    <lldp-
local-port-id>529</lldp-local-port-id>\n    <lldp-local-port-ageout-count>0</lldp-local-port-ageout-
count>\n    <lldp-remote-chassis-id-subtype>Mac address</lldp-remote-chassis-id-subtype>\n
<lldp-remote-chassis-id>78:fe:3d:3d:f6:40</lldp-remote-chassis-id>\n    <lldp-remote-port-id-
subtype>Locally assigned</lldp-remote-port-id-subtype>\n    <lldp-remote-port-id>531</lldp-remote-
port-id>\n    <lldp-remote-port-description>ge-0/1/1.0</lldp-remote-port-description>\n    <lldp-
remote-system-name>frodo</lldp-remote-system-name>\n    <lldp-system-description>\n
<lldp-remote-system-description>Juniper Networks, Inc. ex2200-c-12t-2g Ethernet Switch, kernel JUNOS
15.1R6.7, Build date: 2017-04-23 00:39:39 UTC Copyright (c) 1996-2017 Juniper Networks, Inc.</
lldp-remote-system-description>\n    </lldp-system-description>\n    <lldp-remote-system-capabilities-
supported>Bridge Router</lldp-remote-system-capabilities-supported>\n    <lldp-remote-system-
capabilities-enabled>Bridge Router</lldp-remote-system-capabilities-enabled>\n    <lldp-med-remote-
system-class>Network Connectivity</lldp-med-remote-system-class>\n
<lldp-remote-org-def-info-oui>0.12.f</lldp-remote-org-def-info-oui>\n    <lldp-remote-org-def-info-
subtype>1</lldp-remote-org-def-info-subtype>\n    <lldp-remote-org-def-info-index>1</lldp-remote-org-
def-info-index>\n    <lldp-remote-org-def-info>036C110000</lldp-remote-org-def-info>\n    <lldp-
remote-org-def-info-oui>0.12.f</lldp-remote-org-def-info-oui>\n
<lldp-remote-org-def-info-subtype>3</lldp-remote-org-def-info-subtype>\n    <lldp-remote-org-def-
info-index>2</lldp-remote-org-def-info-index>\n    <lldp-remote-org-def-info>0100000000</lldp-remote-
org-def-info>\n    <lldp-remote-org-def-info-oui>0.12.f</lldp-remote-org-def-info-oui>\n    <lldp-
remote-org-def-info-subtype>4</lldp-remote-org-def-info-subtype>\n
<lldp-remote-org-def-info-index>3</lldp-remote-org-def-info-index>\n    <lldp-remote-org-def-
info>05EA</lldp-remote-org-def-info>\n    <lldp-remote-org-def-info-oui>0.80.c2</lldp-remote-org-def-
info-oui>\n    <lldp-remote-org-def-info-subtype>1</lldp-remote-org-def-info-subtype>\n    <lldp-
remote-org-def-info-index>4</lldp-remote-org-def-info-index>\n    <lldp-remote-org-def-info>0000</
lldp-remote-org-def-info>\n    <lldp-remote-org-def-info-oui>0.90.69</lldp-remote-org-def-info-oui>\n
<lldp-remote-org-def-info-subtype>1</lldp-remote-org-def-info-subtype>\n    <lldp-remote-org-def-
info-index>5</lldp-remote-org-def-info-index>\n    <lldp-remote-org-def-
```

```
info>475030323131343633353138</lldp-remote-org-def-info>\n    <lldp-remote-org-def-info-oui>0.80.c2</
lldp-remote-org-def-info-oui>\n    <lldp-remote-org-def-info-subtype>3</lldp-remote-org-def-info-
subtype>\n    <lldp-remote-org-def-info-index>6</lldp-remote-org-def-info-index>\n    <lldp-remote-
org-def-info>000004616A6132</lldp-remote-org-def-info>\n    <lldp-remote-org-def-info-oui>0.12.bb</
lldp-remote-org-def-info-oui>\n    <lldp-remote-org-def-info-subtype>1</lldp-remote-org-def-info-
subtype>\n    <lldp-remote-org-def-info-index>7</lldp-remote-org-def-info-index>\n    <lldp-remote-
org-def-info>000F04</lldp-remote-org-def-info>\n </lldp-neighbor-information>\n</lldp-neighbors-
information>\n",
            "matches": [
                {
                    "lldp-remote-port-description": "ge-0/1/1.0"
                },
                {
                    "lldp-remote-system-name": "frodo"
                }
            ],
...
        }
    ]
  }
}


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

Because the first two tasks are both loops, we get the contents of the `item` variable for each iteration of the loop in each task. As we saw previously, this adds some additional output to our results – just the local interface name on the "get lldp neighbor table" task, but the full XML string being queried on the "get neighbor details" task.

Can we use a loop control label to display something shorter for the "get neighbor details" task? Ideally, we should display just the local interface name, so the loop label would match the "get lldp neighbor table" task. However, despite its length, the `item` variable for the "get neighbor details" task is a single string of XML data, not a dictionary in which we can easily reference a field. Yet within that XML is the `lldp-local-interface` tag, we just need to figure out a way to isolate its contents.

Regular expressions, or *regex*, are a powerful tool for searching for patterns of text in strings. We can use Ansible's `regex_search()` filter to look for `'<lldp-local-interface>[^<]+'`. That will search for the opening tag `<lldp-local-interface>` followed by any text that matches the regex pattern `[^<]+`, which matches any text that does not contain a '<' character. We want the regex stop matching when it finds the next '<' because that marks the start of the closing tag, meaning the regex has matched the complete interface name.

NOTE   Regular expressions are a complex topic. A detailed explanation of how regex work or the full meaning of the pattern `[^<]+` is outside the scope of this book. Please see the References section at the end of this chapter for additional resources.

Now turn your attention to the output of the "show neighbor details" task. Observe the now-familiar `results` list, and that each dictionary within the `results` list contains a `matches` list with the XML tags found by our XPath query. Also observe that each dictionary in the `results` list also contains an `item` field with the XML data that was queried. We can use the `matches` and `item` fields to clean up the output of this task.

Modify the last two tasks as follows:

```
25|    - name: get neighbor details
26|      xml:
27|        xmlstring: "{{ item }}"
28|        xpath: //lldp-remote-system-name | //lldp-remote-port-description
29|        content: text
30|      loop: "{{ lldp.results | map(attribute='stdout') | list }}"
31|      register: neighbors
32|      loop_control:
33|        label: "{{ item | regex_search('<lldp-local-interface>[^<]+') }}"
34|
35|    - name: show neighbor details
36|      debug:
37|        msg: "{{ item.matches }}"
38|      loop: "{{ neighbors.results }}"
39|      loop_control:
40|        label: "{{ item.item | regex_search('<lldp-local-interface>[^<]+') }}"
```

Lines 32 and 33 add the loop control label discussed above, using the `regex_search()` filter to extract only the opening tag and interface name from the XML data.

TIP   It is possible to get just the interface name without the tag by adding a second filter step using `regex_replace()`. This will be left as an exercise for the reader.

Line 38 adds a loop to the "show neighbor details" task to iterate over the `neighbors.results` list.

Line 37 displays only the `matches` list from each iteration of the loop.

Lines 39 and 40 use basically the same loop control label as lines 32 and 33 to display the local interface name, except it references the "inner" `item` variable that was part of the `neighbors.results` list.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-interface.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] *******************************************

TASK [get lldp neighbor table] ************************************************
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)
```

```
TASK [get neighbor details] ****************************************************
ok: [bilbo] => (item=<lldp-local-interface>ge-0/1/0.0)
ok: [bilbo] => (item=<lldp-local-interface>ge-0/1/1.0)

TASK [show neighbor details] ***************************************************
ok: [bilbo] => (item=<lldp-local-interface>ge-0/1/0.0) => {
    "msg": [
        {
            "lldp-remote-port-description": "ge-0/0/0"
        },
        {
            "lldp-remote-system-name": "aragorn"
        }
    ]
}
ok: [bilbo] => (item=<lldp-local-interface>ge-0/1/1.0) => {
    "msg": [
        {
            "lldp-remote-port-description": "ge-0/1/1.0"
        },
        {
            "lldp-remote-system-name": "frodo"
        }
    ]
}

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

Nice!

Using a Single RPC Call – Get LLDP List Version 1

One concern with the get-lldp-interface.yaml playbook is that it calls the juniper_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 juniper_junos_rpc to retrieve the entire LLDP neighbor table. We can then use XPath expressions with predicates to extract only the interfaces of interest.

Enter the following playbook, get-lldp-list.yaml (lines 30 and 31 are long and may wrap in the book; each should be a single line in your playbook):

```
1|---
2|- name: Get LLDP neighbor information
3|  hosts:
4|    - all
5|  roles:
6|    - Juniper.junos
7|  connection: local
8|  gather_facts: no
9|
10|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
```

```
13|
14|  tasks:
15|    – name: get lldp neighbor table
16|      juniper_junos_rpc:
17|        provider: "{{ connection_settings }}"
18|        rpcs: get-lldp-neighbors-information
19|        format: xml
20|      register: lldp
21|
22|    – name: display lldp neighbor results
23|      debug:
24|        var: lldp.stdout_lines
25|
26|    – name: get neighbor details
27|      xml:
28|        xmlstring: "{{ lldp.stdout }}"
29|        xpath: >
30|          //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp-remote-
system-name |
31|          //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp-remote-
port-description
32|        content: text
33|      loop: "{{ aja2_host.uplinks }}"
34|      register: neighbors
35|
36|    – name: show neighbor results
37|      debug:
38|        var: neighbors
```

Lines 15–20 get the device's entire LLDP neighbor table as XML data. Notice we no longer need a loop on this task.

Lines 22–24 display the XML LLDP data, just so we can confirm that we get the entire table (including interfaces in which we are not interested).

Lines 26–34 query the saved XML data using XPath path expressions with predicates to filter for a specific interface. Note the loop (line 34) which iterates over our list of uplink interfaces, `aja2_host.uplinks`. Note also the reference to the `item` variable in the XPath predicates, `[starts-with(lldp-local-interface, '{{ item }}')]`, on lines 30 and 31.

Because the `xpath` argument (lines 29–31) is rather long, 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 30.

Lines 36–38 display the complete neighbor results.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-list.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] ************************************************************
****

TASK [get lldp neighbor table] ************************************************************
```

```
****
ok: [bilbo]

TASK [display lldp neighbor results] ********************************************
****
ok: [bilbo] => {
    "lldp.stdout_lines": [
        "<lldp-neighbors-information style=\"brief\">",
        "  <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>4c:96:14:0c:de:40</lldp-remote-chassis-id>",
        "    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>",
        "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
        "  </lldp-neighbor-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-0/0/8.0</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>7c:25:86:c1:af:07</lldp-remote-chassis-id>",
        "    <lldp-remote-port-description>ge-0/0/11</lldp-remote-port-description>",
        "    <lldp-remote-system-name>elrond</lldp-remote-system-name>",
        "  </lldp-neighbor-information>",
        "</lldp-neighbors-information>"
    ]
}

TASK [get neighbor details] ****************************************************
****
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)

TASK [show full neighbor results] *********************************************
****
ok: [bilbo] => {
    "neighbors": {
        "changed": false,
        "msg": "All items completed",
        "results": [
            {
...
                "changed": false,
                "count": 2,
                "failed": false,
...
                    "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"
                }
```

```
            },
            "item": "ge-0/1/0",
            "matches": [
                {
                    "lldp-remote-port-description": "ge-0/0/0"
                },
                {
                    "lldp-remote-system-name": "aragorn"
                }
            ],
            "msg": 2,
...
        },
        {
...
            "changed": false,
            "count": 2,
            "failed": false,
...
                "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"
            }
        },
        "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=0    unreachable=0    failed=0
```

Observe in the "display lldp neighbor results" task's output that our XML LLDP data contains the local interface ge-0/0/8.0, whose neighbor we do not care about.

However, observe in the "show neighbor results" task's output that we recieved results for only the two uplink interfaces. The `loop` and XPath expressions in the "get neighbor details" task effectively extracted only the interfaces of interest.

Let's clean up the output of the "show neighbor results" task. Modify the playbook as follows:

```
36|   - name: show neighbor results
37|     debug:
38|       msg: "{{ item.matches }}"
```

```
39|     loop: "{{ neighbors.results }}"
40|     loop_control:
41|       label: "{{ item.item }}"
```

This uses the same technique we discussed for the `get-lldp-interface.yaml` playbook to loop through the results list and display only the desired data.

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook get-lldp-list.yaml --limit=bilbo

PLAY [Get LLDP neighbor information] *******************************************

TASK [get lldp neighbor table] ************************************************
ok: [bilbo]

TASK [display lldp neighbor results] ******************************************
ok: [bilbo] => {
    "lldp.stdout_lines": [
        "<lldp-neighbors-information style=\"brief\">",
        "  <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>4c:96:14:0c:de:40</lldp-remote-chassis-id>",
        "    <lldp-remote-port-description>ge-0/0/0</lldp-remote-port-description>",
        "    <lldp-remote-system-name>aragorn</lldp-remote-system-name>",
        "  </lldp-neighbor-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-0/0/8.0</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>7c:25:86:c1:af:07</lldp-remote-chassis-id>",
        "    <lldp-remote-port-description>ge-0/0/11</lldp-remote-port-description>",
        "    <lldp-remote-system-name>elrond</lldp-remote-system-name>",
        "  </lldp-neighbor-information>",
        "</lldp-neighbors-information>"
    ]
}

TASK [get neighbor details] ***************************************************
ok: [bilbo] => (item=ge-0/1/0)
ok: [bilbo] => (item=ge-0/1/1)

TASK [show neighbor results] **************************************************
ok: [bilbo] => (item=ge-0/1/0) => {
    "msg": [
        {
            "lldp-remote-port-description": "ge-0/0/0"
        },
```

```
        {
            "lldp-remote-system-name": "aragorn"
        }
    ]
}
ok: [bilbo] => (item=ge-0/1/1) => {
    "msg": [
        {
            "lldp-remote-port-description": "ge-0/1/1.0"
        },
        {
            "lldp-remote-system-name": "frodo"
        }
    ]
}

PLAY RECAP *********************************************************************
bilbo                      : ok=4   changed=0   unreachable=0   failed=0
```

Now the output of the last task is much shorter and contains only data of interest.

Two Templates for Interface Descriptions – Get LLDP List Version 2

Let's create two templates that take the results of our XML query from the registered variable `neighbors` and create Junos configuration snippets that assign descriptions 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 (remove the two `debug` tasks and add or update the boldfaced lines):

```
1|---
2|- name: Get LLDP neighbor information and save interface descriptions using templates
3|  hosts:
4|    - all
5|  roles:
6|    - Juniper.junos
7|  connection: local
8|  gather_facts: no
9|
10|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|    tmp_dir: "{{ user_data_path }}/tmp"
14|    template_dir: "template"
15|
16|  tasks:
17|    - name: get lldp neighbor table
18|      juniper_junos_rpc:
19|        provider: "{{ connection_settings }}"
20|        rpcs: get-lldp-neighbors-information
21|        format: xml
22|      register: lldp
23|
24|    - name: get neighbor details
25|      xml:
```

```
26|      xmlstring: "{{ lldp.stdout }}"
27|      xpath: >
28|        //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp-remote-
system-name |
29|        //lldp-neighbor-information[starts-with(lldp-local-interface, '{{ item }}')]/lldp-remote-
port-description
30|      content: text
31|    loop: "{{ aja2_host.uplinks }}"
32|    register: neighbors
33|
34|  - name: save interface descriptions, template 1
35|    template:
36|      src: "{{ template_dir }}/int-desc-1.j2"
37|      dest: "{{ tmp_dir }}/{{ inventory_hostname }}-{{ item.item | replace('/', '-') }}.conf"
38|    loop: "{{ neighbors.results }}"
```

Lines 34–38 use the template module to generate a configuration file based on the template file int-desc-1.j2 for each interface in our results. Because we get a configuration file for each interface, the configurations' filenames should be distinct, which we accomplish by including the interface name in the filename.

This task uses a loop over the neighbors.results list to populate the item variable with data about each matching interface in turn, mostly for use in the template which we create momentarily. However, line 37 uses the item element of the item variable (item.item), which contains the interface name, to help generate the filename in which we store the template's results. The replace() filter replaces the slashes in Junos' interface names with hyphens (for example, ge-0/0/0 becomes ge-0-0-0) because putting slashes in Unix filenames is poor practice.

Now, let's create our first template. Create, if needed, a template directory within your playbook directory:

mbp15:aja2 sean$ **mkdir template**

Create file int-desc-1.j2 in the template directory (line numbers added for discussion):

```
 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 dictionary (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, assigns the value to the appropriate variable.

Lines 12–16 are the Junos configuration 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:aja2 sean$ ansible-playbook get-lldp-list.yaml --limit=bilbo

PLAY [Get LLDP neighbor information and save interface descriptions using templates]
********************

TASK [get lldp neighbor table] *******************************************************
****
ok: [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] ***************************************
****
changed: [bilbo] => (item={...})
changed: [bilbo] => (item={...})

PLAY RECAP **************************************************************************
*****
bilbo                      : ok=3    changed=1    unreachable=0    failed=0

mbp15:aja2 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 hyphens from lines 2 and 3 of the template?

With Jinja2 templates, when you update a simple variable within a loop, the update 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 interesting 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 Jinja2 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 argument. 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 (not shown) and confirm we get the desired results in the two *.conf files:

```
mbp15:aja2 sean$ cat ~/ansible/tmp/bilbo-ge-0-1-0.conf
interfaces {
    ge-0/1/0 {
        description "to device aragorn port ge-0/0/0";
    }
}

mbp15:aja2 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* statements) 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; 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:

```
...
34|  - name: save interface descriptions, template 1
35|    template:
36|      src: "{{ template_dir }}/int-desc-1.j2"
37|      dest: "{{ tmp_dir }}/{{ inventory_hostname }}-{{ item.item | replace('/', '-') }}.conf"
38|    loop: "{{ neighbors.results }}"
39|
40|  - name: save interface descriptions, template 2
41|    template:
42|      src: "{{ template_dir }}/int-desc-2.j2"
43|      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`, replacing the `loop: "{{ neighbors.results }}"` loop used with the previous 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 `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:aja2 sean$ ansible-playbook get-lldp-list.yaml --limit=bilbo

PLAY [Get LLDP neighbor information and save interface descriptions using templates]
********************

TASK [get lldp neighbor table] *******************************************************************
****
ok: [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] *********************************************
****
ok: [bilbo] => (item={...})
ok: [bilbo] => (item={...})

TASK [save interface descriptions, template 2] *********************************************
****
changed: [bilbo]

PLAY RECAP *****************************************************************************************
*****
bilbo                    : ok=4    changed=1    unreachable=0    failed=0

mbp15:aja2 sean$ cat ~/ansible/tmp/bilbo.conf
interfaces {
    ge-0/1/0 {
        description "to device aragorn port ge-0/0/0";
    }
    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 aragorn port ge-0/0/0";
    }
}
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:
https://docs.ansible.com/ansible/latest/user_guide/playbooks_loops.html

Ansible tests:
https://docs.ansible.com/ansible/latest/user_guide/playbooks_tests.html

Ansible error handling:
https://docs.ansible.com/ansible/latest/user_guide/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/

A Regular Expressions book the author likes:
http://www.regular-expressions-cookbook.com/

Short-circuit evaluation:
https://en.wikipedia.org/wiki/Short-circuit_evaluation

Chapter 14

Custom Ansible Modules

There will come a time when you want a playbook to do something for which an Ansible module does not yet exist, or which cannot be done (the way you want it done) by combining existing modules. 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 the Python language. Ansible does not really care what programming language you use to write your module, provided your module complies with Ansible's module interface standard. However, 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 only at a fairly high level.

NOTE   The author uses a somewhat different development process than what Ansible proposes in their developer documentation, using their "hacking/test-module script." The reader is encouraged to explore Ansible's process as well and adopt whichever approach you find works best.

The Problem

Recall in Chapter 10 we updated our base-settings playbook and template to set connection-limit and rate-limit values for SSH and NETCONF connections. Because some devices only support 3 or 5 connections, while others support hundreds, we used a test based on device model and personality to determine if we should set the various *-limit values to 3, 5, or 10. That test worked well as an illustration of using *if-elif-else* to make decisions in a template, but it is simplified relative to the real-world variation in maximum values across various Junos devices and versions.

It would be nice if we could just query a device to find out the maximum value it supports for its SSH and NETCONF *-limit settings. Unfortunately, if there is a command or RPC that returns that value, the author has not been able to find it.

However, there are two ways we can find the value for a given device at the Junos command line in configuration mode. We saw the first in Chapter 10, leveraging the CLI's interactive help feature:

```
{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# set system services ssh rate-limit
```

The second is to try setting one of the *-limit settings to an unsupported value; the error message tells us the supported range for the device:

```
{master:0}[edit]
sean@bilbo# set system services ssh rate-limit 1000
                                               ^
Value 1000 is not within range (1..250) at '1000'

{master:0}[edit]
sean@bilbo#
```

Can we use either of these approaches with the juniper_junos_config module? You can use the following playbook to test each approach (line numbers added), just uncomment either line 21 or 22 to test either approach:

```
 1|---
 2|- name: Find max SSH -- testing Junos CLI approaches
 3|  hosts:
 4|    - all
 5|  roles:
 6|    - Juniper.junos
 7|  connection: local
 8|  gather_facts: no
 9|
10|  vars:
11|    connection_settings:
12|      host: "{{ ansible_host }}"
13|      timeout: 120
14|
```

```
15|  tasks:
16|    - name: install set commands onto device
17|      juniper_junos_config:
18|        provider: "{{ connection_settings }}"
19|        load: set
20|        lines:
21|          - set system services ssh rate-limit ?
22|         # - set system services ssh rate-limit 1000
```

The first approach fails because '?' is not a numeric value and is not recognized during RPC calls as a request for help. Even though we provide a CLI-style "set" command as an argument, the `juniper_junos_config` module does not access the device's command line. The actual error message received by the author:

```
...
TASK [install set commands onto device] ****************************************
fatal: [bilbo]: FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: error, bad_element: ?, message: error: Invalid numeric value: '?')"}
...
```

This error message does not help us; it does not reveal the range of valid values.

The second approach fails, as expected, because the value is out of range:

```
...
TASK [install set commands onto device] ****************************************
fatal: [bilbo]: FAILED! => {"changed": false, "msg": "Failure loading the configuraton:
ConfigLoadError(severity: error, bad_element: 1000, message: error: Value 1000 is not within range
(1..250))"}
...
```

This error message contains the range of valid values, and we could process that in a playbook to extract the maximum allowed value. However, doing this in a playbook will require several tasks, and if the playbook's focus is installing a configuration template (similar to our `base-settings` playbook) then the series of tasks devoted to interpreting the error might be confusing. A single task – a custom module – that returns the desired value for a device would make such a playbook much cleaner.

NOTE   See `test-max-ssh-ansible-only.yaml` in the GitHub repository for this book for an example of a Ansible playbook that implements the second test, forcing a failure and interpreting the message, without a custom module.

Functional Description of Our Custom Module

Our custom module will use PyEZ to implement both approaches discussed above for determining the SSH limit allowed by a given device. Why both approaches? Because we can! The first approach, not possible with the `juniper_junos_config` module, can be done with some trickery in PyEZ. Our module will use one approach to find the maximum allowed value for `rate-limit`, the other to find `connection-limit`. In every device the author has checked the same maximum applies to both settings but testing them separately lets us illustrate both approaches.

The module will return both the maximum values allowed by the device for its `connection-limit` and `rate-limit` settings, and suggested values for the device's `connection-limit` and `rate-limit` settings that, in most cases, will be much lower than the maximum allowed values. The suggested values will let us remove the `max_ssh` decision logic from our `base-settings.j2` template.

Our module should accept the following arguments from the playbook:

- `host`: The hostname or IP address of the device being tested. (Required; normally set to `ansible_host` by the playbook.)

- `test_value`: The out-of-range value used for the second approach (catching the exception). Overriding the default test value is useful for testing some aspects of the module. (Optional; default to 0.)

- `connection_limit`: The value to which we want to set our device's `connection-limit` settings, provided the device supports at least the indicated number of connections. (Optional; default to 15.)

- `rate_limit`: The value to which we want to set our device's `rate-limit` settings, provided the device supports at least this rate. (Optional; default to 10.)

For simplicity we assume SSH key-based authentication using the logged-in user's username, so we do not need to pass username and password to the module. (PyEZ automatically reads the current user's name from the operating system when one is not provided).

The functionality of the module will divide into two major areas:

- A new class that handles the communication with the Junos device and subsequent 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 handles most of the interface work. This means the interface code looks similar across different custom modules. Part of the functionality provided by the *AnsibleModule* class includes assigning default values to arguments that may not have been passed by the playbook. We further discuss *AnsibleModule* in the "Creating the Ansible Module" section of this chapter.

Developing the Class

Let's start by creating the *MaxSSHConnections* class that communicates with the Junos device and processes the results.

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 playbook.

- 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.

Create file `test_max_ssh_connections.py` with the following Python code (line numbers added for discussion):

```
 1|#!/usr/bin/env python
 2|"""Query devices for maximum allowed SSH connection-limit and rate-limit."""
 3|
 4|import re
 5|import sys
 6|from jnpr.junos import Device
 7|from jnpr.junos.exception import ConfigLoadError
 8|from jnpr.junos.utils.config import Config
 9|from jnpr.junos.utils.start_shell import StartShell
10|from pprint import pprint
11|
12|
13|####################################################################
14|
15|class MaxSSHConnections(object):
16|    """Class to query devices or maximum connection-limit setting."""
17|
18|    def __init__(self, device, **kwargs):
19|        """Initialize instance variables."""
20|        self.dev = Device(host=device, normalize=True)
21|
22|        self.desired_connection_limit = kwargs.get('connection_limit', 15)
23|        self.desired_rate_limit = kwargs.get('rate_limit', 10)
24|        self.test_value = kwargs.get('test_value', 0)
25|
26|        self.results = {'host': device,
27|                        'connection_max': 0,
28|                        'rate_max': 0,
29|                        'connection_limit': 0,
30|                        'rate_limit': 0,
31|                        'exception_message': '',
32|                        'shell_results': [],
33|                        'warnings': []
34|                        }
35|
```

```
36|    # ------------------------ #
37|
38|    def get_max_connections(self):
39|        """Use shell commands to find maximum allowed connection-limit."""
40|        # the list of commands that will:
41|        # - exit from the command shell to the Junos CLI
42|        # - enter configuration mode
43|        # - issue the command "set system services ssh connection-limit ?",
44|        #     which will return help information we want to process
45|        # - exit configuration mode
46|        shell_commands = [
47|            {'command': 'exit', 'prompt': '> ', 'max': False},
48|            {'command': 'configure', 'prompt': '# ', 'max': False},
49|            {'command': 'set system services ssh connection-limit ?',
50|             'prompt': '# ', 'max': True},
51|            {'command': 'exit', 'prompt': '> ', 'max': False}
52|        ]
53|
54|        # open a command shell on the device
55|        shell = StartShell(self.dev)
56|        shell.open()
57|
58|        # iterate over the list of commands, capturing the output from
59|        #   the command in whose results we are interested ('max' = True)
60|        max_msg = None
61|        for shellcmd in shell_commands:
62|            shellout = shell.run(shellcmd['command'], shellcmd['prompt'])
63|            self.results['shell_results'].append(shellout)
64|
65|            if shellout[0] is False:
66|                msg = 'Shell command "%s" did not complete as expected: %s' \
67|                    % (shellcmd['command'], shellout[1])
68|                raise RuntimeError(msg)
69|
70|            if shellcmd['max']:
71|                max_msg = shellout[1]
72|
73|        shell.close()
74|
75|        # process the command output to find the max allowed value
76|        if max_msg is not None:
77|            max_arr = max_msg.splitlines()
78|            regex = r'connection-limit[^\(\[]*[\(\[]\d+\.\.(\d+)'
79|            max_str = None
80|            for line in max_arr:
81|                m = re.search(regex, line, flags=re.IGNORECASE)
82|                if m is not None:
83|                    max_str = m.group(1)
84|                    break
85|
86|            if max_str is not None:
87|                reported_max = int(max_str)
88|                self.results['connection_max'] = reported_max
89|                if reported_max < self.desired_connection_limit:
90|                    self.results['connection_limit'] = reported_max
91|                else:
92|                    self.results['connection_limit'] = \
93|                        self.desired_connection_limit
94|            else:
```

```
 95|                 msg = 'Regex match expected but not found in command results'
 96|                 raise ValueError(msg)
 97|         else:
 98|             msg = 'Missing expected results from shell commands.'
 99|             raise ValueError(msg)
100|
101|     # ------------------------- #
102|
103|     def get_max_rate(self):
104|         """Set an invalid value for rate-limit and process the exception."""
105|         # configuration object for Junos device
106|         cfg = Config(self.dev)
107|
108|         # make sure no config change is pending before our set command
109|         diff = cfg.diff()
110|         if diff is not None:
111|             msg = 'Uncommitted change found: %s' % str(diff)
112|             raise RuntimeError(msg)
113|
114|         # try to set a invalid (too large) value for rate-limit
115|         set_cmd = 'set system services ssh rate-limit ' + str(self.test_value)
116|         try:
117|             cfg.load(set_cmd, format='set')
118|             # Config load should raise exception if the test value is invalid.
119|             # If we got here, it means the device accepted the (apparently
120|             #  valid) rate-limit, so roll back the change and assume the
121|             #  test value is the maximum allowed rate limit
122|             cfg.rollback()
123|             msg = 'Test configuration loaded without error, actual max '
124|             msg += 'rate limit may be higher than the test value '
125|             msg += '%s.' % str(self.test_value)
126|             self.results['warnings'].append(msg)
127|             self.results['rate_max'] = self.test_value
128|             if self.test_value < self.desired_rate_limit:
129|                 self.results['rate_limit'] = self.test_value
130|             else:
131|                 self.results['rate_limit'] = self.desired_rate_limit
132|         except ConfigLoadError as err:
133|             self.results['exception_message'] = err.message
134|             # catch the expected ConfigLoadError from the invalid rate-limit
135|             match = re.search(r'\(\d+\.\.(\d+)\)', err.message)
136|             if match is not None:
137|                 max_str = int(match.group(1))
138|                 reported_max = int(max_str)
139|                 self.results['rate_max'] = reported_max
140|                 if reported_max < self.desired_rate_limit:
141|                     self.results['rate_limit'] = reported_max
142|                 else:
143|                     self.results['rate_limit'] = self.desired_rate_limit
144|             else:
145|                 msg = 'Regex match expected but not found in caught '
146|                 msg += 'exception: %s' % str(err)
147|                 raise ValueError(msg)
148|
149|     # ------------------------- #
150|
151|     def run(self):
152|         """Run the device test and return result."""
```

```
153|        # open a PyEZ connection to the device
154|        self.dev.open()
155|
156|        # get max connection limit (first approach)
157|        self.get_max_connections()
158|
159|        # get max rate limit (second approach)
160|        self.get_max_rate()
161|
162|        # close device connection
163|        self.dev.close()
164|
165|
166|#####################################################################
167|
168|def main():
169|    """Test the MaxSSHConnections class."""
170|    desired_connections = 15
171|    desired_rate = 10
172|    test_value = 0
173|    device = 'bilbo'
174|
175|    find_max = MaxSSHConnections(device, test_value=test_value,
176|                                 rate_limit=desired_rate,
177|                                 connection_limit=desired_connections)
178|    try:
179|        find_max.run()
180|    except Exception as err:
181|        print(str(err))
182|        sys.exit(1)
183|
184|    pprint(find_max.results)
185|
186|
187|#####################################################################
188|
189|if __name__ == '__main__':
190|    main()
```

Lines 15 – 163 define the class *MaxSSHConnections*, which uses the two approaches discussed earlier to get the device's maximum connection-limit and rate-limit values. The class has four methods.

Lines 18 – 34 comprise the class' __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.results to contain the results that the class returns to the calling process.

Lines 38 – 99 comprise the class's get_max_connections() method. This method connects to the device and determines the maximum value for the SSH connection-limit setting. This method uses the first approach discussed earlier in the chapter, leveraging the CLI help system.

NOTE   The get_max_connections() method uses a technique sometimes called *screen scraping*, running commands at the device's user-interactive command line and gathering text results that need to be processed. Screen scraping is *not* the

preferred approach to automation with Junos devices; your automation should use the RPC API whenever possible, whether directly or via frameworks like PyEZ or Ansible. Reserve screen scraping for the rare situations that are not addressed by a device's API, or for automation with devices that do not have an API.

Lines 103 – 147 comprise the class's `get_max_rate()` method. This method connects to the device and determines the maximum value for the SSH `rate-limit` setting. This method uses the second approach discussed earlier in the chapter, setting `rate-limit` to an invalid value and catching the resulting exception.

Lines 151 – 163 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 opens the initial PyEZ device connection, then calls the `get_max_connections()` and `get_max_rate()` methods to perform most of the work of the class.

Lines 168 – 184 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 `device` variable to one of your test devices), instantiates an object of the `MaxSSHConnections` class, and then calls the `run()` method on the object. This method also prints the results of the class' work, contained in the `find_max.results` variable.

There are a couple things to observe in this program:

- The *MaxSSHConnections* 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; however, you must remove all print statements before creating the Ansible module. The Ansible module returns JSON data to the playbook on STDOUT. Because print statements also send their results to STDOUT, the printed output is likely to interfere with the playbook's interpretation of the module's results; the combination of the module's results and print output is unlikely to be valid JSON. (See also "Python logging module" in the References at the end of the chapter.)

- Exceptions, with meaningful messages, are used to indicate problems during class execution. It is up to the `main()` method to catch the exceptions. To some extent this is a deliberate simplification for the book, but this approach legitimately helps with the design Ansible module as well. Raising exceptions for all errors means the Ansible module will need one simple technique for handling module failures, which we will see later in this chapter.

- Module results that should be passed back to the Ansible playbook are collected in Python data structure(s), such as this example's `results` dictionary. Because playbook results are converted to JSON format, try to use simple variables, lists and dictionaries.

- The `results` dictionary includes eight fields, two of which are helpful but not strictly necessary, and two of which help with understanding the module (or

with debugging) but which should probably be removed from a production version of this module. More on the `results` fields momentarily.

Now run the test program:

```
mbp15:aja2 sean$ python test_max_ssh_connections.py
{'connection_limit': 15,
 'connection_max': 250,
 'exception_message': 'error: Value 0 is not within range (1..250)',
 'host': 'bilbo',
 'rate_limit': 10,
 'rate_max': 250,
 'shell_results': [(True,
                u'exit\r\r\nexit\r\n\r\n{master:0}\r\nsean@bilbo> '),
               (True,
                u'configure \r\nEntering configuration mode\r\n\r\n{master:0}[edit]\r\nsean@bilbo#
'),
               (True,
                u'set system services ssh connection-limit ?\r\nPossible completions:\r\n
<connection-limit>   Maximum number of allowed connections (1..250)\r\n{master:0}[edit]\r\n\rsean@
bilbo# set system services ssh connection-limit    \x08\x08\x08\r\n
^\r\nsyntax error, expecting <data>.\r\n\r\n{master:0}[edit]\r\nsean@bilbo# '),
               (True,
                u'exit \r\nExiting configuration mode\r\n\r\n{master:0}\r\nsean@bilbo> ')],
 'warnings': []}
```

Let's briefly discuss the output, which is the contents of the *MaxSSHConnections* class' `results` dictionary. The `connection_max` and `rate_max` fields contain the values detected by the class as the maximum values allowed by the device for the respective settings. The `connection_limit` and `rate_limit` fields contain suggested values for configuring the respective settings on the device, using the lesser of the detected maximums or the arguments of the same names passed to the class when it was instantiated. These four values are the required output of this class as discussed earlier in this chapter.

The `host` field simply confirms the device that was queried. The `warnings` field contains messages related to the class' results, things that the user might want to be aware of but that were not a fatal error.

Finally, the `shell_results` and `exception_message` fields are included to help during development and debugging of the module but should be removed from a production version. The `exception_message` field contains the error message from the exception raised by trying to set an invalid `rate-limit` value on the device. The `shell_results` field contains the results of executing the series of shell and CLI command to get the allowed range of values for the device's `connection-limit` setting.

Change the variables defined in the `main()` method and run the program again to see how the results change for different arguments. For example, set `test_value = 12` (line 172) to see how the results change when the test value is valid rather than invalid:

```
mbp15:aja2 sean$ python test_max_ssh_connections.py
{'connection_limit': 15,
 'connection_max': 250,
 'exception_message': '',
 'host': 'bilbo',
 'rate_limit': 10,
 'rate_max': 12,
 'shell_results': [(True,
                   u'exit\r\r\nexit\r\n\r\n{master:0}\r\nsean@bilbo> '),
                  (True,
                   u'configure \r\nEntering configuration mode\r\n\r\n{master:0}[edit]\r\nsean@bilbo#
'),
                  (True,
                   u'set system services ssh connection-limit ?\r\nPossible completions:\r\n
<connection-limit>  Maximum number of allowed connections (1..250)\r\n{master:0}[edit]\r\n\rsean@
bilbo# set system services ssh connection-limit    \x08\x08\x08\r\n
^\r\nsyntax error, expecting <data>.\r\n\r\n{master:0}[edit]\r\nsean@bilbo# '),
                  (True,
                   u'exit \r\nExiting configuration mode\r\n\r\n{master:0}\r\nsean@bilbo> ')],
 'warnings': ['Test configuration loaded without error, actual max rate limit may be higher than the
test value 12.']}
```

Notice that the key `'exception_message'` has an empty string for a value (because there was no exception), and the `'warnings'` key contains a message that informs the user that the test value might not have provided the best results.

Creating the Ansible Module

Now let's convert our test program into a custom Ansible module. This should require no changes to the *MaxSSHConnections* 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 for custom modules is a *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 easy inclusion in the code repository for your playbooks.

Within your ~/aja2 directory, create a new `library` subdirectory, then copy the `test_max_ssh_connections.py` program into the `library` subdirectory as `max_ssh_connections.py`.

```
mbp15:aja2 sean$ mkdir library
```

```
mbp15:aja2 sean$ cp test_max_ssh_connections.py library/max_ssh_connections.py
```

Open the `library/max_ssh_connections.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|"""Query devices for maximum allowed SSH connection-limit and rate-limit."""
```

```
 3|
 4|import re
 5|from ansible.module_utils.basic import AnsibleModule
 6|from jnpr.junos import Device
 7|from jnpr.junos.exception import ConfigLoadError
 8|from jnpr.junos.utils.config import Config
 9|from jnpr.junos.utils.start_shell import StartShell
10|
11|
12|####################################################################
...
```

Line 5 imports the *AnsibleModule* class, which provides a lot of the interface between the playbook and our new module. The import lines for sys and pprint have been removed as we no longer need them.

When a playbook calls a module, Ansible assembles the arguments from the playbook 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 when a custom module is called:

- 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 playbook.

- Confirm the data type of arguments (e.g. integer vs. string). If no type is specified, arguments are assumed to be strings.

AnsibleModule also helps return results from the module back to the calling playbook. It provides methods for indicating whether the module had a failure or changed the target device, and returning any additional data the module needs to return to the calling playbook.

Let's put *AnsibleModule* to work. Delete the current main() method and replace it with the following (line numbers added for discussion):

```
...
165|####################################################################
166|
167|def main():
168|    """Test the MaxSSHConnections class."""
169|    # define arguments from Ansible
170|    module = AnsibleModule(
171|        argument_spec=dict(
172|            host=dict(required=True),
173|            test_value=dict(required=False, type='int', default=0),
174|            rate_limit=dict(required=False, type='int', default=10),
175|            connection_limit=dict(required=False, type='int', default=15)
176|            )
177|        )
178|
179|    # copy playbook arguments into local variables
```

```
180|    host = module.params['host']
181|    test_value = module.params['test_value']
182|    rate_limit = module.params['rate_limit']
183|    connection_limit = module.params['connection_limit']
184|
185|    # instantiate MaxSSHConnections and run
186|    find_max = MaxSSHConnections(host, test_value=test_value,
187|                          rate_limit=rate_limit,
188|                          connection_limit=connection_limit)
189|    try:
190|        find_max.run()
191|    except Exception as err:
192|        module.fail_json(msg=str(err), results=find_max.results)
193|
194|    module.exit_json(changed=False, results=find_max.results,
195|                 rate_limit=find_max.results['rate_limit'],
196|                 connection_limit=find_max.results['connection_limit'])
197|
198|
199|#####################################################################
200|
201|if __name__ == '__main__':
202|    main()
```

Lines 170 – 177 instantiate an object `module` from the *AnsibleModule* class. The `argument_spec` dictionary tells *AnsibleModule* about the arguments it should expect from the playbook.

Line 172 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 173 says we might get an argument called `test_value`. Because `required=False`, this argument is optional. Should `test_value` not be provided, `default=0` automatically sets the value of `test_value` to `0`. Because *AnsibleModule* assumes arguments are strings unless we tell it otherwise, the `type='int'` setting tells *AnsibleModule* that this particular argument must be an integer. Other types include 'bool' for Boolean values, 'dict' for dictionary (key:value) data, and 'list' for list/array data.

Lines 174 and 175 do the same thing as line 173 but for the arguments `rate_limit` and `connection_limit`, with appropriate default values.

Lines 180 – 183 copy each of the arguments from the `module.params` dictionary into local variables. The *AnsibleModule* object `module` keeps all the arguments in a dictionary called `params`. While not strictly necessary, it is often easier to work with the argument's values if they are copied into local variables.

Lines 186 – 188 instantiates object `find_max` from our class *MaxSSHConnections*, providing all the arguments needed for the class to do its work.

Lines 189 – 192 calls `find_max.run()`, the `run()` method of our *MaxSSHConnections* class, to process the device and generate the required output. The call to `run()` is within a `try–except` block to catch any exceptions raised by the module. In this module we do not need to do anything with exceptions other than report the

failure back to the Ansible playbook, which is handled by the `module.fail_json()` method call on line 192. The structure of our example module allows all errors to be handled in a single `try-except` block, 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.

Lines 194 – 196 call the `module.exit_json()` method to exit the module without an error.

Note we have two different exits from our playbook, the `module.fail_json()` call on line 192 and the `module.exit_json()` call on lines 194–196. Both `exit_json()` and `fail_json()` are methods defined in the *AnsibleModule* class. The major difference between them is that the `fail_json()` method indicates the module encountered an error – the calling playbook will record a *fatal* result for the task – while `exit_json()` indicates a successful completion of the module – the calling playbook will record an *ok* or *changed* result for the task.

When an Ansible module exits, it returns results to the calling playbook as a JSON dictionary (*key:value* data). The return dictionary should include at least the key `"failed"` with a Boolean value indicating whether or not the module encountered a fatal error (this value, when True, tells the playbook to display the red "fatal" status for the task), and the key `"changed"` with a Boolean value indicating whether the module changed the target host in some way (this value tells the calling playbook whether to display the green "ok" or yellow "changed" status for the task).

The return data can also include a `"msg"` key with a status or error message. While a `"msg"` key is optional, a module should include an error message when it encounters a failure so users of the playbook understand why the module failed.

The module can also include any other *key:value* data it needs to return to the calling playbook, where the *key* should be a meaningful identifier and the *value* can be anything (simple value, list, dictionary) in JSON format.

Take another look at lines 194–196 showing a normal exit (no error), and line 192 showing an exit with error. Notice that in both cases we return the entire `results` dictionary from the `find_max` object. Observe that the names of the keyword arguments to the `exit_json()` and `fail_json()` methods are the *unquoted* keys desired in the JSON results.

Lines 194–196 also copy the most important two `results` values into their own keys `connection_limit` and `rate_limit`. Duplicating selected results is not required, but in some cases, it makes it a little easier for the playbook to reference the most important return data from the module.

Note also that we return `changed=False` because this module does change the target device. A module that can change the target device may have two `exit_json()` calls, one that sets `changed=False` and another that sets `changed=True`. Alternately, the module can set a Boolean variable, named something like `host_changed`, to True or False and return that variable's value in the `exit_json()` call (`changed=host_changed`).

On line 192, notice the `msg` key is set to the exception's error message. We do not need to explicitly set `failed=True` because `fail_json()` does this for us.

TIP   In order to keep this example fairly straightforward, the author suggested copying the `test_max_ssh_connections.py` test program to create the `library/max_ssh_connections.py` custom module. This creates a problem should the *MaxSSHConnections* class need to be modified in the future: you have two copies of the class definition, one in the test program and one in the module. If you update the class in `test_max_ssh_connections.py`, you need to manually replicate those changes to `library/max_ssh_connections.py`, an error-prone process. It might 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 GitLab repository for this book contains a second example of selected files from this chapter showing one approach to putting the class in a separate, importable file.

Testing the Custom Module

Let's create a simple playbook to test the completed module. In your `~/aja2` directory, create playbook `get-max-ssh.yaml`:

```
 1|---
 2|- name: Get maximum ssh rate-limit and connection-limit
 3|  hosts:
 4|    - all
 5|  connection: local
 6|  gather_facts: no
 7|
 8|  tasks:
 9|    - name: get max ssh
10|      max_ssh_connections:
11|        host: "{{ ansible_host }}"
12|        test_value: 50
13|        # rate_limit: 15
14|        connection_limit: 25
15|      register: max_ssh
16|
17|    - debug:
18|        var: max_ssh
```

Run the playbook and observe the results:

```
mbp15:aja2 sean$ ansible-playbook get-max-ssh.yaml

PLAY [Get maximum ssh rate-limit and connection-limit] **********************************************
****

TASK [get max ssh] *********************************************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [debug] ***************************************************************************************
*****
```

```
ok: [bilbo] => {
    "max_ssh": {
        "changed": false,
        "connection_limit": 25,
        "failed": false,
        "rate_limit": 10,
        "results": {
            "connection_limit": 25,
            "connection_max": 250,
            "exception_message": "",
            "host": "198.51.100.5",
            "rate_limit": 10,
            "rate_max": 50,
            "shell_results": [
                [
                    true,
                    "exit\r\r\nexit\r\n\r\n{master:0}\r\nsean@bilbo> "
                ],
                [
                    true,
                    "configure \r\nEntering configuration mode\r\n\r\n{master:0}[edit]\r\nsean@bilbo# "
                ],
                [
                    true,
                    "set system services ssh connection-limit ?\r\nPossible completions:\r\n  <connection-
limit>   Maximum number of allowed connections (1..250)\r\n{master:0}[edit]\r\n\rsean@bilbo# set
system services ssh connection-limit    \b\b\b\r\n                                      ^\r\
nsyntax error, expecting <data>.\r\n\r\n{master:0}[edit]\r\nsean@bilbo# "
                ],
                [
                    true,
                    "exit \r\nExiting configuration mode\r\n\r\n{master:0}\r\nsean@bilbo> "
                ]
            ],
            "warnings": [
                "Test configuration loaded without error, actual max rate limit may be higher than the
test value 50."
            ]
        }
    }
}
ok: [aragorn] => {
    "max_ssh": {
        "changed": false,
        "connection_limit": 5,
        "failed": false,
        "rate_limit": 5,
        "results": {
            "connection_limit": 5,
            "connection_max": 5,
            "exception_message": "error: Value 50 is not within range (1..5)",
            "host": "192.0.2.10",
            "rate_limit": 5,
            "rate_max": 5,
            "shell_results": [
                [
                    true,
                    "exit\r\r\nexit\r\n\r\nsean@aragorn> "
```

```
            ],
            [
              true,
              "configure \r\nEntering configuration mode\r\n\r\n[edit]\r\nsean@aragorn# "
            ],
            [
              true,
              "set system services ssh connection-limit ?\r\nPossible completions:\r\n  <connection-
limit>   Maximum number of allowed connections (1..5)\r\n[edit]\r\n\rsean@aragorn# set system services
ssh connection-limit    \b\b\b\r\n                                         ^\r\nsyntax error,
expecting <data>.\r\n\r\n[edit]\r\nsean@aragorn# "
            ],
            [
              true,
              "exit \r\nExiting configuration mode\r\n\r\nsean@aragorn> "
            ]
          ],
          "warnings": []
      }
    }
}

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

Review the output and note how the arguments in the playbook affected the re-sults. For example, note that the `test_value` of 50 provides a good test for *aragorn*, but *bilbo* warns that this value was valid. Change some of the arguments in the playbook – for example, uncomment the `rate_limit` argument or comment out the `test_value` argument – and run the playbook again, observing how the results change.

Now simulate a failure – the author disconnected his *bilbo* switch – and run the playbook again:

```
mbp15:aja2 sean$ ansible-playbook get-max-ssh.yaml

PLAY [Get maximum ssh rate-limit and connection-limit] *********************************************
****

TASK [get max ssh] ********************************************************************************
****
ok: [aragorn]
fatal: [bilbo]: FAILED! => {"changed": false, "msg": "ConnectTimeoutError(198.51.100.5)", "results":
{"connection_limit": 0, "connection_max": 0, "exception_message": "", "host": "198.51.100.5", "rate_
limit": 0, "rate_max": 0, "shell_results": [], "warnings": []}}

TASK [debug] **************************************************************************************
*****
ok: [aragorn] => {
...
}
        to retry, use: --limit @/Users/sean/aja2/get-max-ssh.retry
```

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

Observe that *bilbo* records a failure with an error message.

Adding max_ssh_connections to the Base Settings Playbook

Let's update our `base-settings.yaml` playbook and `base-settings.j2` template to use our new `max_ssh_connections` module and its results. This same change can be made to the `system` role we created in Chapter 12, but doing so will be left as an exercise for the reader.

Modify the `base-settings.yaml` playbook as shown, replacing the tasks that called the `juniper_junos_facts` module and displayed its results with the tasks that call our `max_ssh_connections` module and display its results:

```
 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|    connection_settings:
14|      host: "{{ ansible_host }}"
15|      timeout: 120
16|
17|  tasks:
18|    - name: confirm or create configs directory
19|      file:
20|        path: "{{ tmp_dir }}"
21|        state: directory
22|      run_once: yes
23|
24|    - name: get max ssh values
25|      max_ssh_connections:
26|        host: "{{ ansible_host }}"
27|        rate_limit: 8
28|      register: jmax
29|
30|    - name: display max ssh values
31|      debug:
32|        var: jmax
33|
34|    - name: save device configuration using template
35|      template:
36|        src: template/base-settings.j2
37|        dest: "{{ conf_file }}"
38|
```

```
39|    – name: install generated configuration file onto device
40|      juniper_junos_config:
41|        provider: "{{ connection_settings }}"
42|        src: "{{ conf_file }}"
43|        load: replace
44|        comment: "playbook base–settings.yaml, commit confirmed"
45|        confirmed: 5
46|        diff: yes
47|        ignore_warning: yes
48|      register: config_results
49|      notify: confirm previous commit
50|
51|    – name: show configuration change
52|      debug:
53|        var: config_results.diff_lines
54|      when: config_results.diff_lines is defined
55|
56|    # – name: delete generated configuration file
57|    #   file:
58|    #     path: "{{ conf_file }}"
59|    #     state: absent
60|
61|  handlers:
62|    – name: confirm previous commit
63|      juniper_junos_config:
64|        provider: "{{ connection_settings }}"
65|        comment: "playbook base–settings.yaml, confirming previous commit"
66|        commit: yes
67|        diff: no
```

Lines 24—28 call our new module and register the results in variable `jmax`.

Modify the `template/base-settings.j2` template to remove the logic that calculated the `max_ssh` variable (was lines 2—12) and instead use the results from our new registered variable `jmax` to set our `rate-limit` and `connection-limit` values:

```
1|#jinja2: lstrip_blocks: True
2|
3|{#– Generate basic settings for the device #}
4|system {
5|    host–name {{ inventory_hostname }};
6|    root–authentication {
7|        encrypted–password "{{ root_hash }}";
8|    }
9|    login {
10|        user monitor {
11|            uid 2005;
12|            class read–only;
13|            authentication {
14|                encrypted–password "{{ monitor_hash }}";
15|            }
16|        }
17|        user sean {
18|            uid 2000;
19|            class super–user;
20|            authentication {
21|                ssh–rsa "ssh–rsa AAAAB3NzaC1y...vPzOaX3gt8Uv sean@mbp15.local";
```

```
22|            }
23|         }
24|      }
25|      replace:
26|      name-server {
27|        {% for server in aja2_host.dns_servers %}
28|           {{ server }};
29|        {% endfor %}
30|      }
31|      services {
32|         delete: ftp;
33|         netconf {
34|            ssh {
35|               connection-limit {{ jmax.connection_limit }};
36|               rate-limit {{ jmax.rate_limit }};
37|            }
38|         }
39|         ssh {
40|            connection-limit {{ jmax.connection_limit }};
41|            rate-limit {{ jmax.rate_limit }};
42|         }
43|         delete: telnet;
44|         delete: web-management;
45|      }
46|      replace:
47|      ntp {
48|        {% for ntp in aja2_site.ntp_servers %}
49|         server {{ ntp }};
50|        {% endfor %}
51|      }
52|}
53|snmp {
54|   description "{{ aja2_host.snmp.description}}"
55|   location "{{ aja2_host.snmp.location}}"
56|}
```

Run the playbook:

```
mbp15:aja2 sean$ ansible-playbook base-settings.yaml

PLAY [Generate and Install Configuration File] *********************************************************
****

TASK [confirm or create configs directory] ************************************************************
****
ok: [bilbo]

TASK [get max ssh values] *****************************************************************************
****
ok: [aragorn]
ok: [bilbo]

TASK [display max ssh values] *************************************************************************
****
ok: [bilbo] => {
    "jmax": {
        "changed": false,
        "connection_limit": 15,
```

```
        "failed": false,
        "rate_limit": 8,
        "results": {
            "connection_limit": 15,
            "connection_max": 250,
            "exception_message": "error: Value 0 is not within range (1..250)",
            "host": "198.51.100.5",
            "rate_limit": 8,
            "rate_max": 250,
            "shell_results": [
                [
                    true,
                    "exit\r\r\nexit\r\n\r\n{master:0}\r\nsean@bilbo> "
                ],
                [
                    true,
                    "configure \r\nEntering configuration mode\r\n\r\n{master:0}[edit]\r\nsean@bilbo# "
                ],
                [
                    true,
                    "set system services ssh connection-limit ?\r\nPossible completions:\r\n  <connection-
limit>   Maximum number of allowed connections (1..250)\r\n{master:0}[edit]\r\n\rsean@bilbo# set
system services ssh connection-limit    \b\b\b\r\n                                        ^\r\
nsyntax error, expecting <data>.\r\n\r\n{master:0}[edit]\r\nsean@bilbo# "
                ],
                [
                    true,
                    "exit \r\nExiting configuration mode\r\n\r\n{master:0}\r\nsean@bilbo> "
                ]
            ],
            "warnings": []
        }
    }
}
ok: [aragorn] => {
    "jmax": {
        "changed": false,
        "connection_limit": 5,
        "failed": false,
        "rate_limit": 5,
        "results": {
            "connection_limit": 5,
            "connection_max": 5,
            "exception_message": "error: Value 0 is not within range (1..5)",
            "host": "192.0.2.10",
            "rate_limit": 5,
            "rate_max": 5,
            "shell_results": [
                [
                    true,
                    "exit\r\r\nexit\r\n\r\nsean@aragorn> "
                ],
                [
                    true,
                    "configure \r\nEntering configuration mode\r\n\r\n[edit]\r\nsean@aragorn# "
                ],
                [
                    true,
```

```
            "set system services ssh connection-limit ?\r\nPossible completions:\r\n  <connection-
limit>   Maximum number of allowed connections (1..5)\r\n[edit]\r\n\r\nsean@aragorn# set system services
ssh connection-limit    \b\b\b\r\n                                         ^\r\nsyntax error,
expecting <data>.\r\n\r\n[edit]\r\nsean@aragorn# "
            ],
            [
                true,
                "exit \r\nExiting configuration mode\r\n\r\nsean@aragorn> "
            ]
        ],
        "warnings": []
    }
  }
}

TASK [save device configuration using template] ********************************************************
****
ok: [aragorn]
changed: [bilbo]

TASK [install generated configuration file onto device] ***********************************************
****
ok: [aragorn]
changed: [bilbo]

TASK [show configuration change] **********************************************************************
****
skipping: [aragorn]
ok: [bilbo] => {
    "config_results.diff_lines": [
        "[edit system services ssh]",
        "-    connection-limit 10;",
        "+    connection-limit 15;",
        "-    rate-limit 10;",
        "+    rate-limit 8;",
        "[edit system services netconf ssh]",
        "-     connection-limit 10;",
        "+     connection-limit 15;",
        "-     rate-limit 10;",
        "+     rate-limit 8;"
    ]
}

RUNNING HANDLER [confirm previous commit] *************************************************************
****
ok: [bilbo]

PLAY RECAP *******************************************************************************************
*****
aragorn                    : ok=4    changed=0    unreachable=0    failed=0
bilbo                      : ok=7    changed=2    unreachable=0    failed=0
```

In this example run, there is no change to *aragorn's* configuration but *bilbo* was updated with new `connection-limit` and `rate-limit` settings.

Review the generated configuration files. Note that *aragorn's* settings are all 5

because that is the maximum allowed by the device, but *bilbo*, which allows a maximum setting of 250, instead gets the higher `rate-limit` setting specified in the playbook and the `connection-limit` that is the default in the module.

```
mbp15:aja2 sean$ grep "limit" tmp/*.conf
tmp/aragorn.conf:              connection-limit 5;
tmp/aragorn.conf:                rate-limit 5;
tmp/aragorn.conf:            connection-limit 5;
tmp/aragorn.conf:            rate-limit 5;
tmp/bilbo.conf:                connection-limit 15;
tmp/bilbo.conf:                 rate-limit 8;
tmp/bilbo.conf:            connection-limit 15;
tmp/bilbo.conf:            rate-limit 8;
```

References

Ansible's documentation about developing modules:
http://docs.ansible.com/ansible/latest/dev_guide/developing_modules.html

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 traditional programmer treats their source code.

This chapter 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 include 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 refers 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 automation engineers, programmers, web site developers, and similar information workers.

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 participated 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 "master" 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 employs 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 information 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 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 ~/aja2 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 developers 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 system 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 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 settings 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:aja2 sean$ git config --global user.name "Sean Sawtell"

mbp15:aja2 sean$ git config --global user.email "my_email@juniper.net"

mbp15:aja2 sean$ git config --global --list
user.name=Sean Sawtell
user.email=my_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:aja2 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:widget 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

No commits yet

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.

"No commits yet" 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

No commits yet

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 repository – 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

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   ansible.cfg
        new file:   play-widget.yaml
```

TIP   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 provide 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 `inventory` 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 or --all flag to the `git commit` command, which tells `git` to include changes to all tracked files that have changed (but not new, untracked files):

```
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
```

TIP   You can combine the –a and –m options if you wish: `git commit -am "mes-sage"`.

Now that we have two commits, let's use the `git log` command to review the commit log (history):

```
mbp15:widget sean$ git log
commit 057299ab9b60810ca3b06ec4da0458462b649a84 (HEAD -> master)
Author: Sean Sawtell <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>
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 that 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 wish, 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$
```

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 modified 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, if `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'
```

Change the 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.

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:

```
tasks:
  - 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, followed 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>
Date:   Tue Oct 3 11:17:36 2017 –0400

    fix merge conflict on playbook

commit 5317c89976db45009327fe234ae61604a20ddc2e (play1)
Author: Sean Sawtell <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>
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 delete 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 commit, 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 commit ID of the commit, either the long commit ID from `git log` or the short commit ID from `git log --oneline`:

```
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 convenient 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 inventory 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, June 29, 2018]. Developers can create 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 chapter were taken on June 29, 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:
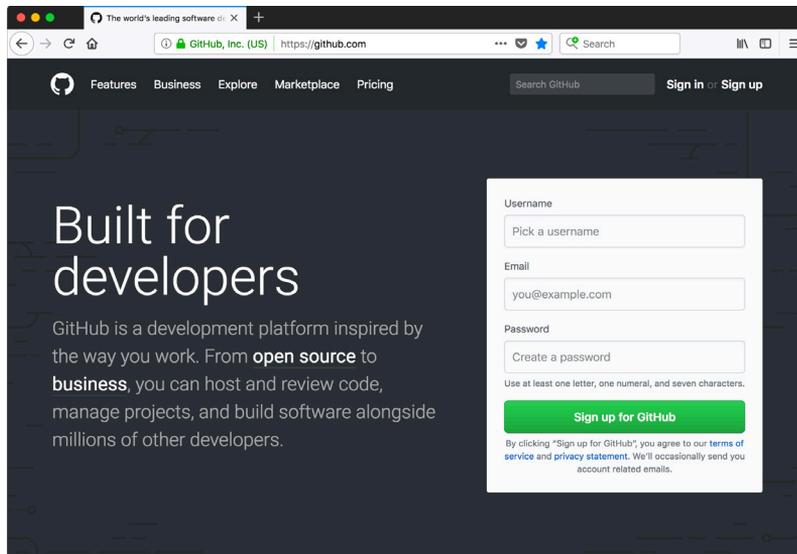
If this is a new account, or if your existing account has no associated repositories, the screen after logging in will offer only a couple of choices.
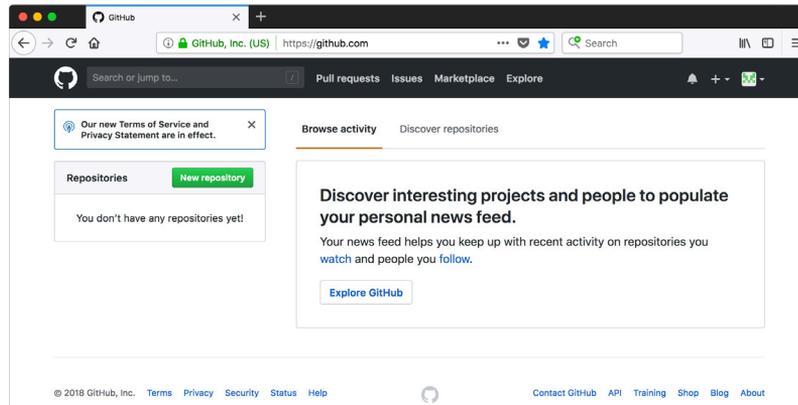
If your account already has one or more associated repositories, the screen after logging in will display the list of your repositories.
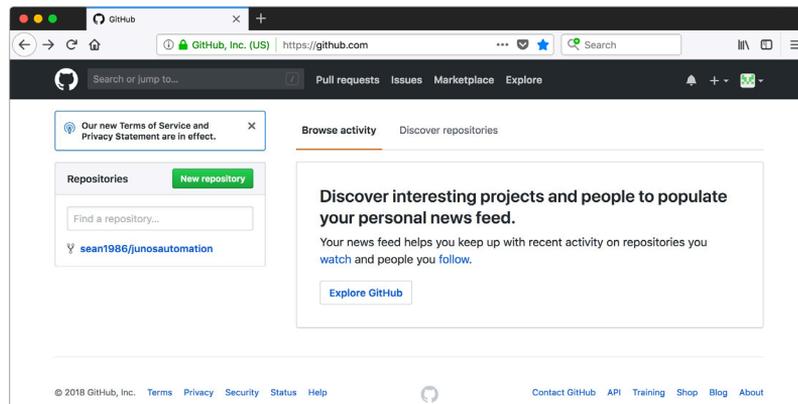
Click the *New repository* button to begin a new project in a new repository. Fill in the fields for creating a new repository as shown in the following screen capture, except that the *Owner* should be your account:
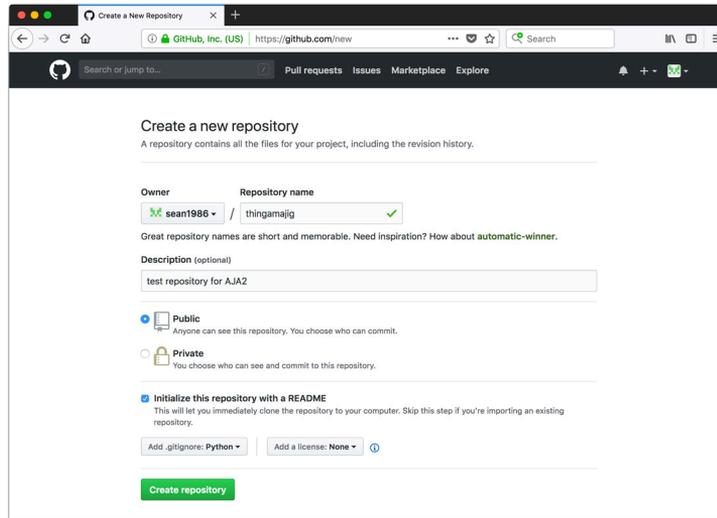
The fields on this page are:

Owner: The GitHub account that has administrative authority for the repository. 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.

Description: An optional description for the repository.

Public or *Private*: GitHub allows public repositories that can be seen by anyone, and private repositories that are visible only to accounts selected by the repository owner. GitHub charges for private repositories, so we will use public 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:
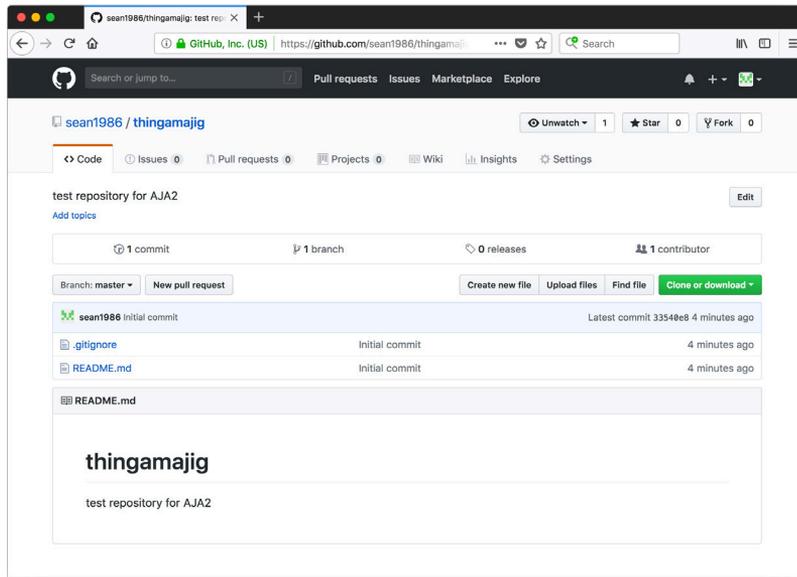
You can see the new repository, including the file list showing the `.gitignore` and `README.md` files. Let's edit one of the files. Click the blue `README.md` in the file list. The next screen will show the file's contents:
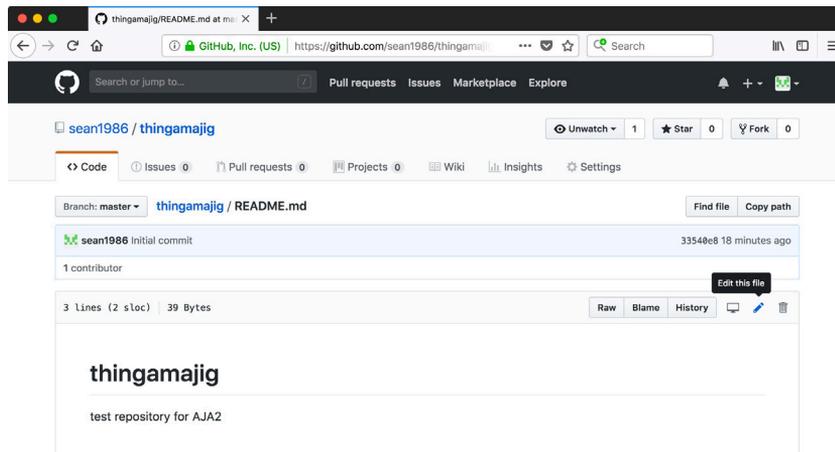
Click the pencil icon to the upper right of the file's contents to edit the file. Modify the file as shown here:
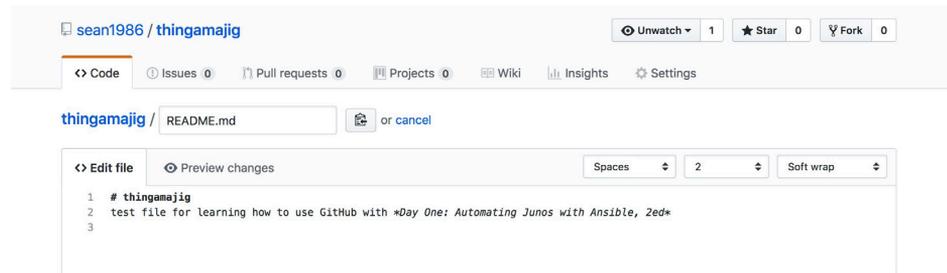
To commit (save) the change, scroll to the bottom of the page, add a commit message in the small text box (where the picture shows "add description to READ-ME") and click the *Commit changes* button:
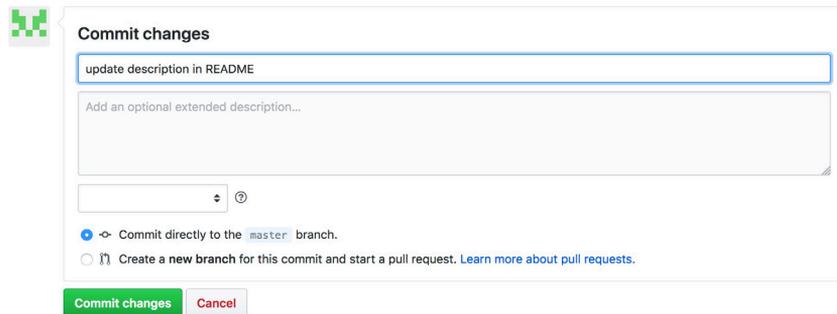
The screen should now look similar to this:
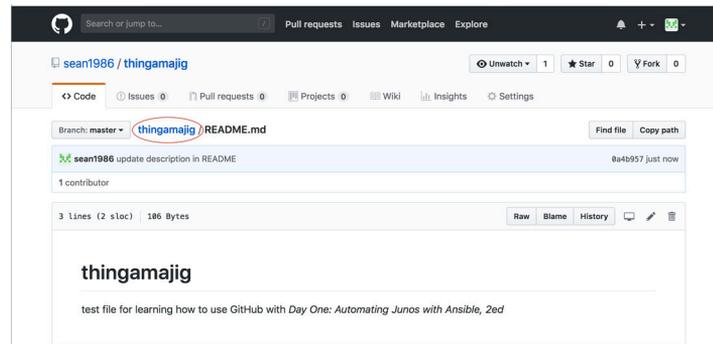
Click the *thingamajig* link (circled above) to return to the project's file list:
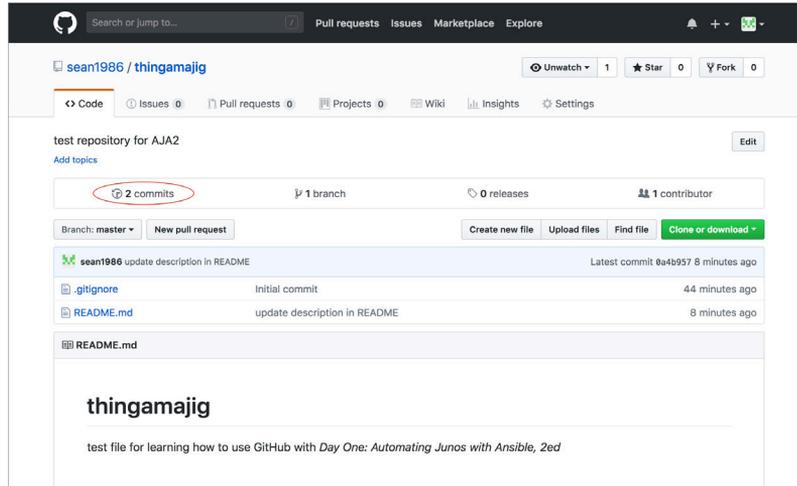
You can see the commit history by clicking the *2 commits* link (circled above; the number will change as more changes are committed):
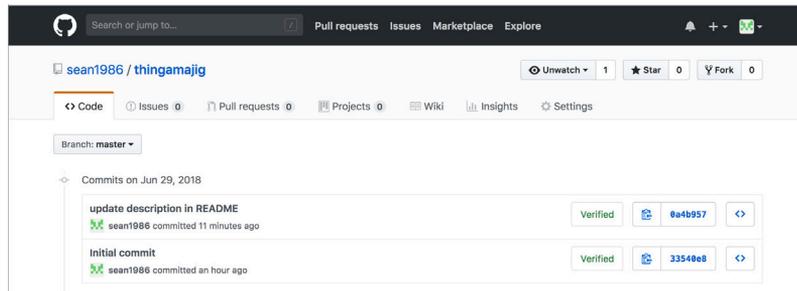
Using Git with GitHub

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, then synchronize changes between their local repository and GitHub.

GitHub allows developers to use either HTTPS or SSH when synchronizing repositories. 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 chapter 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 situation: there is an existing project on GitHub that you wish to copy to a local repository 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:
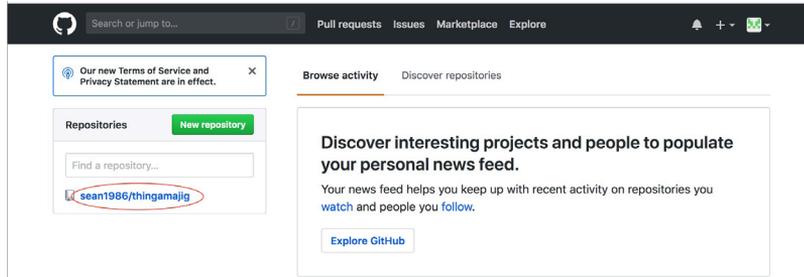
Click the *Clone or download* button to expose the options for cloning the repository. Click the "copy to clipboard" button to the right of the URL to copy the URL to your system's clipboard:
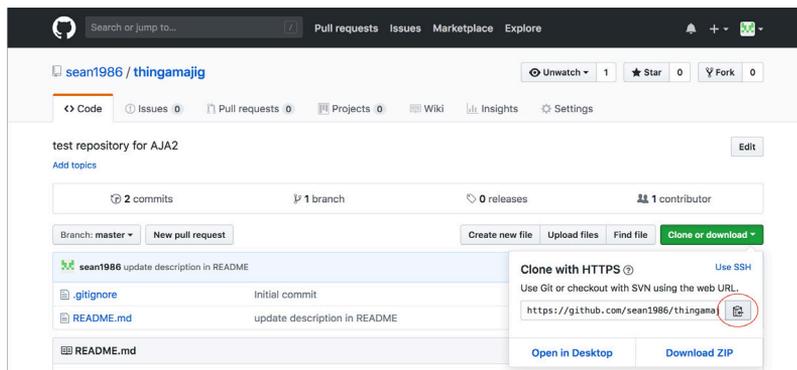
In your command shell, change to your home directory (or whatever directory you want to be the parent of the repository directory). Then run the command `git clone` with the copied URL to create a new repository in a new directory containing a copy of the *thingamajig* project. Your URL will differ from 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 directory 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 file for learning how to use GitHub with *Day One: Automating Junos with Ansible, 2ed*
```

In order to simulate a second team member who is also working with the thingamajig repository, create a directory ~/user2 and clone *thingamajig* into that directory 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:thingamajig sean$ cd ~

mbp15:~ sean$ mkdir user2

mbp15:~ sean$ cd user2/

mbp15:user2 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:user2 sean$ cd thingamajig/

mbp15:thingamajig sean$ pwd
/Users/sean/user2/thingamajig
```

Let's make a change and push that change back to GitHub. In your first command shell window, or in the ~/thingamajig directory, open README.md in your text editor and add a copyright notice as follows:

```
# thingamajig
test file for learning how to use GitHub with *Day One: Automating Junos with Ansible, 2ed*

(c) 2018 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 commit 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 ~/user2/thingamajig and look at the README file:

```
mbp15:thingamajig sean$ pwd
/Users/sean/user2/thingamajig

mbp15:thingamajig sean$ cat README.md
# thingamajig
test file for learning how to use GitHub with *Day One: Automating Junos with Ansible, 2ed*
```

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 member," 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 file for learning how to use GitHub with *Day One: Automating Junos with Ansible, 2ed*

(c) 2018 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 locally 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:
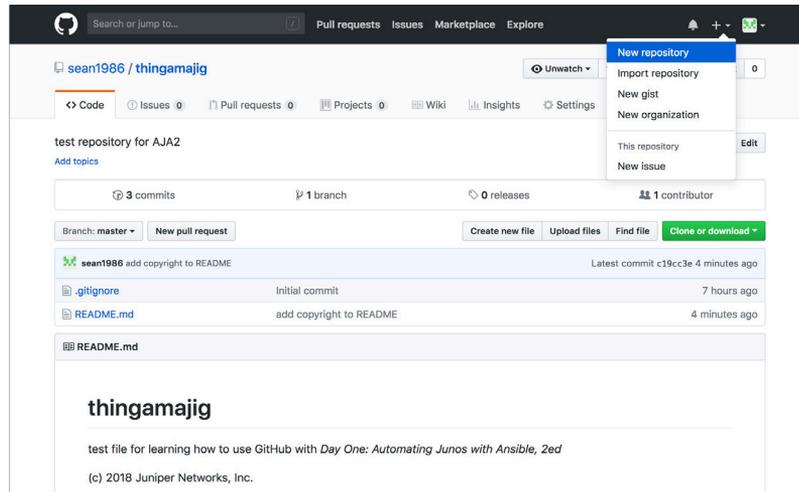
We want a completely empty GitHub repository, because we will upload the files from our local Git repository. Enter the repository name *widget* and, optionally, a description, as shown, then click the *Create repository* button. (Do *not* include a README.md, .gitignore, or license file):
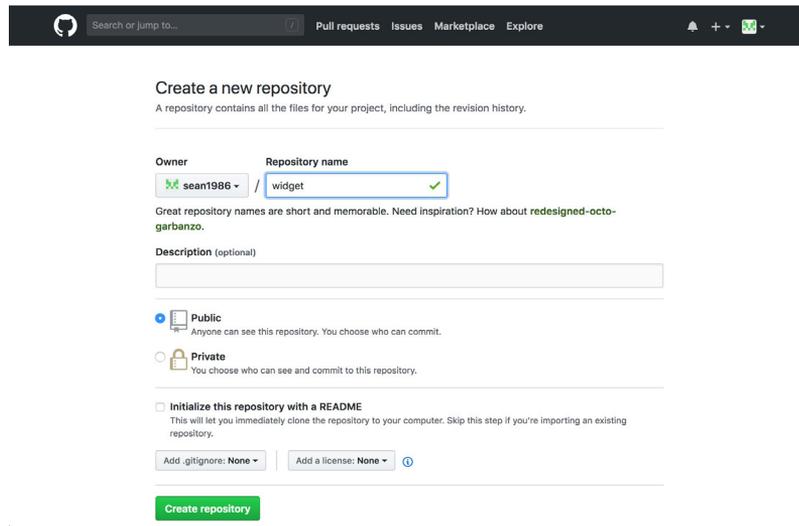
This time GitHub, seeing the repository is empty, should display some instructions for uploading repository files from other locations:
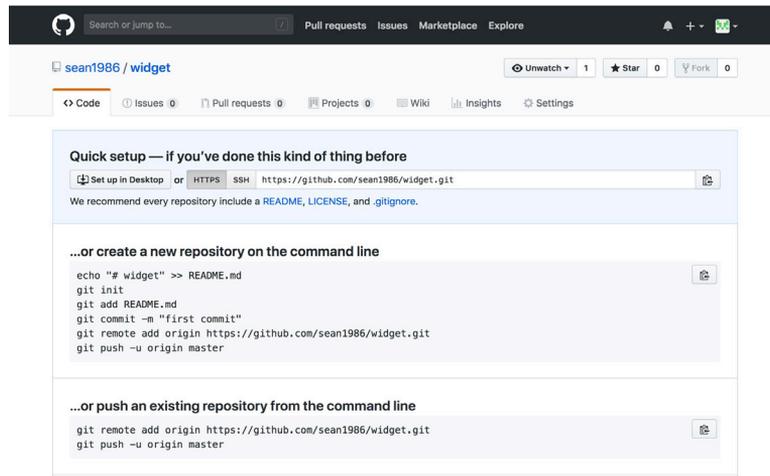
Note the instructions to "push an existing repository from the command line." This is what we want to do! Keep in mind that your URL will be different from what is shown, as your URL will include your GitHub username, not that author's username.

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 repository. 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 include 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 `origin` server, you should repeat this command for any other branches you wish to upload 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 additional command line options. To demonstrate this, add a new play to `play-widget`.

yaml:

```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 ~/user2 directory or second terminal tab, then clone the *widget* repository using its URL (obtained from GitHub):

```
mbp15:thingamajig sean$ cd ..

mbp15:user2 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:user2 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 are likely to be some files that need to exist within the repository directory, but that you do 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, if our ~/aja2/group_vars/all.yaml file contained the following:

```
---
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 username. Should you choose to place the *aja2* 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 ~/aja2/tmp/ directory 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 are ignored by Git. Create ~/widget/.gitignore 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
```

Lines starting with hash marks ('#') are comments. The line `group_vars/all.yaml` instructs Git to ignore that file.

The following lines instruct Git to ignore the contents of our `tmp` directory and other common temporary files and directories:

```
*.tmp
tmp/
temp/
```

The line `*.retry` tells Git to ignore Ansible's "retry" files, created when a playbook encounters an error for one or more hosts.

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 so it gets shared with other team members. (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 -m "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:
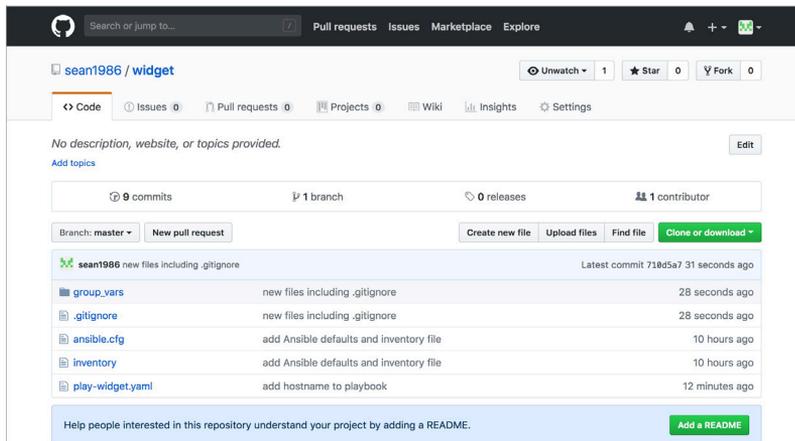
Deleting the Example Repositories

Feel free to delete the local repositories created during this chapter by simply deleting the directories:

```
mbp15:repos sean$ cd ~
mbp15:~ sean$ rm -rf user2/
mbp15:~ sean$ rm -rf widget/
mbp15:~ sean$ rm -rf thingamajig/
```

To delete a repository from GitHub, view the repository and click the *Settings* link:
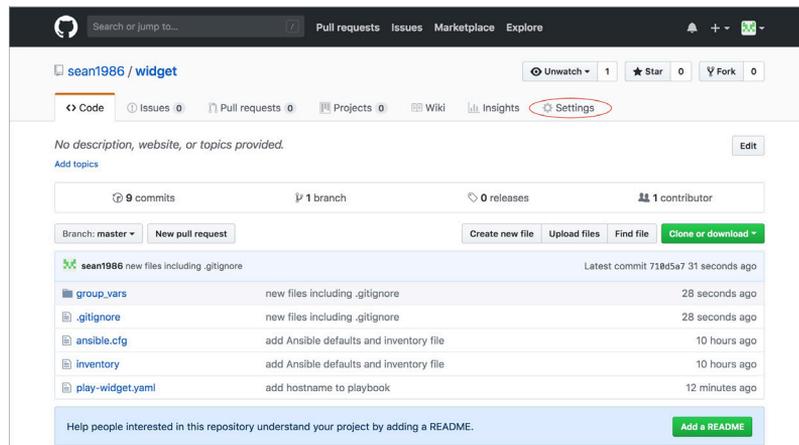
Scroll to the bottom of the Settings page and click the *Delete this repository* button inside the *Danger Zone* box:

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:
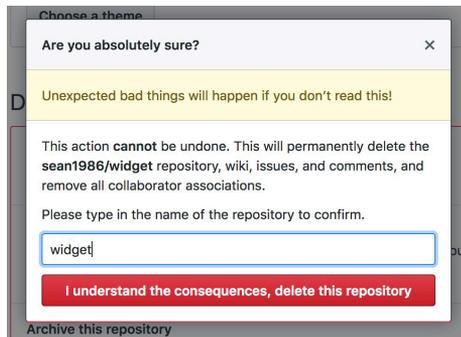
You may be prompted to enter your GitHub password as well; do so if prompted.

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/