



Junos Platform Automation and DevOps

STUDENT GUIDE—Volume 1 of 2

Revision V-17.a

Education Services Courseware

Junos Platform Automation and DevOps

V-17.a

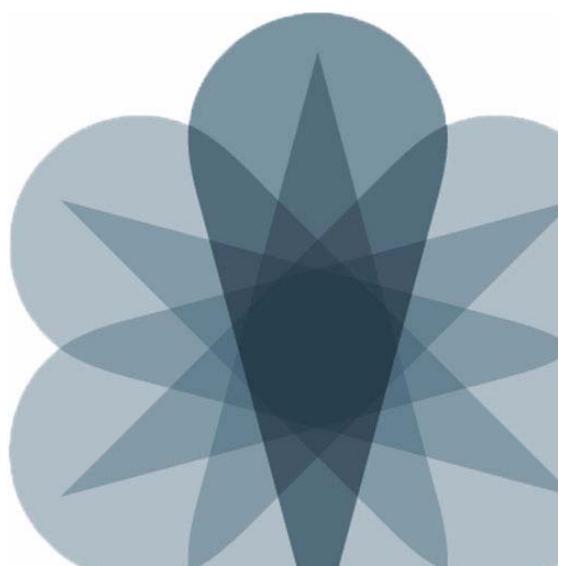
Student Guide
Volume 1



Worldwide Education Services

1133 Innovation Way
Sunnyvale, CA 94089
USA
408-745-2000
www.juniper.net

Course Number: EDU-JUN-JAUT



This document is produced by Juniper Networks, Inc.

This document or any part thereof may not be reproduced or transmitted in any form under penalty of law, without the prior written permission of Juniper Networks Education Services.

Juniper Networks, Junos, Steel-Belted Radius, NetScreen, and ScreenOS are registered trademarks of Juniper Networks, Inc. in the United States and other countries. The Juniper Networks Logo, the Junos logo, and JunosE are trademarks of Juniper Networks, Inc. All other trademarks, service marks, registered trademarks, or registered service marks are the property of their respective owners.

Junos Platform Automation and DevOps Student Guide, Revision V-17.a

Copyright © 2017 Juniper Networks, Inc. All rights reserved.

Printed in USA.

Revision History:

Revision 11.a—September 2011.

Revision 14.a—August 2015.

Revision V-17.a—June 2017.

The information in this document is current as of the date listed above.

The information in this document has been carefully verified and is believed to be accurate for software JJunos OS Release 17.1R1, PyEZ 2.0, Python 2.7, and Ansible 2.3. Juniper Networks assumes no responsibilities for any inaccuracies that may appear in this document. In no event will Juniper Networks be liable for direct, indirect, special, exemplary, incidental, or consequential damages resulting from any defect or omission in this document, even if advised of the possibility of such damages.

Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

YEAR 2000 NOTICE

Juniper Networks hardware and software products do not suffer from Year 2000 problems and hence are Year 2000 compliant. The Junos operating system has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

SOFTWARE LICENSE

The terms and conditions for using Juniper Networks software are described in the software license provided with the software, or to the extent applicable, in an agreement executed between you and Juniper Networks, or Juniper Networks agent. By using Juniper Networks software, you indicate that you understand and agree to be bound by its license terms and conditions. Generally speaking, the software license restricts the manner in which you are permitted to use the Juniper Networks software, may contain prohibitions against certain uses, and may state conditions under which the license is automatically terminated. You should consult the software license for further details.

Contents

Chapter 1:	Course Introduction	1-1
Chapter 2:	Junos Automation Architecture and Overview	2-1
	Why Automate?	2-3
	Junos mgd-Based Automation	2-6
	Junos jsd-Based Automation	2-17
	Automation Languages, Libraries, and Frameworks	2-20
	Automation Management Systems	2-30
	Junos Automation Tools	2-35
Chapter 3:	NETCONF and the XML API	3-1
	NETCONF	3-3
	XML API	3-12
	XML API Programming Languages	3-24
	XML API Tools	3-37
	Lab: NETCONF and the XML API	3-45
Chapter 4:	JSON and YAML	4-1
	JSON and YAML Overview	4-3
	JSON	4-7
	YAML	4-15
	Lab: JSON and YAML	4-24
Chapter 5:	Python and Junos PyEZ	5-1
	Introduction to Python and Junos PyEZ	5-3
	Python Development Environment	5-12
	Working with RPCs	5-37
	Working With and Unstructured Configuration	5-44
	Working With Tables and Views	5-51
	PyEZ Exception Handling	5-66
	Lab: Creating Python and Junos PyEZ Scripts	5-74
Chapter 6:	Jinja2 and Junos PyEZ	6-1
	Jinja2 Overview	6-3
	Jinja2 Syntax	6-6
	Creating a Junos PyEZ, YAML, and Jinja2 Solution	6-18
	Lab: Using Jinja2 Templates with PyEZ	6-33

Chapter 7: Ansible	7-1
Introduction to Ansible	7-3
Building a Basic Ansible Environment	7-10
Creating Ansible Playbooks	7-20
The junos_shutdown Module	7-33
The junos_get_facts Module	7-37
The junos_install_config Module	7-41
Advanced Configuration Deployment	7-47
Additional References	7-53
Lab: Working with Ansible in the Junos OS	7-57
Chapter 8: JSNAPy	8-1
JSNAPy Overview and Installation	8-3
Create JSNAPy Configuration Files	8-8
Create JSNAPy Test Files	8-15
Use the JSNAPy Command Line Tool	8-21
Lab: Using JSNAPy	8-32
Acronym List	ACR-1

Course Overview

This five-day course provides students with knowledge of how to automate Junos using DevOps automation tools, protocols, and technologies. Students receive hands-on development experience with tools and languages relevant to automating the Junos OS platform in a DevOps environment. The course includes an introduction to the Junos XML API, and NETCONF but focuses on using Python, PyEZ, and Ansible to automate Junos. The course introduces students to Junos commit, operation (op), event, and SNMP scripts. JSON, YAML, and Jinja2 are introduced as these languages facilitate Junos automation. The course also introduces the Junos Extension Toolkit and related APIs. Finally, the course discusses the use of JSANPy and Junos ZTP autoinstallation tools. Through demonstrations and hands-on labs, students will gain experience in automating the Junos operating system and device operations.

This course uses Junos OS Release 17.1R1, PyEZ 2.0, Python 2.7, and Ansible 2.3.

Course Level

Junos Platform Automation (JAUT) is an intermediate-level course.

Intended Audience

This course benefits individuals responsible for configuring and monitoring devices running the Junos OS.

Prerequisites

Students should have intermediate-level networking knowledge and an understanding of the Open Systems Interconnection (OSI) model and the TCP/IP protocol suite. Students should also have familiarity with XML basics and have introductory knowledge of the Python programming language. Students should also attend the *Introduction to the Junos Operating System* (IJOS) course prior to attending this class. Lastly, a high-level understanding of object-oriented programming is a plus, but not a requirement.

Objectives

After successfully completing this course, you should be able to:

- Describe the NETCONF protocol.
- Explain the capabilities of the Junos OS XML API.
- Describe the use of XSLT, SLAX, and XPath in the XML API.
- Describe the Junos Automation UI and explain
 - the role of gRPC, NETCONF, and REST in Junos Automation.
- Identify the languages, frameworks, management suites, and tools used in automating Junos.
- Describe the YANG Protocol and explain the capabilities of YANG.
- Use the YANG model to issue Junos commands and to configure Junos.
- Explain the benefits of using JSON and YAML.
- List where JSON and YAML are used in Junos Automation.
- Convert between JSON, YAML, and XML.
- Describe the features and benefits of using Python in Junos automation.
- Configure Junos devices to use Python and create simple Python scripts.
- Describe the function of Junos operation, commit, event, and SNMP scripts.
- Implement Junos operation, commit, event, and SNMP scripts using Python.
- Identify how Junos automation uses Jinja2 and create Python scripts that use Jinja2.
- Explain how PyEZ makes Junos automation easier
- Use PyEZ to gather facts from Junos, perform configuration tasks, and use PyEZ to manipulate the file system and perform system upgrades to Junos.
- Implement OpenConfig in the Junos OS.
- Describe the process

- Implement a translation script for a custom YANG module.
- Explain the use of the Junos REST API in automation.
- Use the Junos REST API to get information from Junos.
- Describe what JET is and what it includes.
- Create a project in the JET IDE.
- Execute scripts using on-box and off-box automation.
- Describe how Ansible is used in Junos automation and install Ansible.
- Create Ansible playbooks to automate Junos.
- Describe how JSNAPy can help automate Junos devices.
- Implement JSNAPy into a Junos environment.
- Describe how ZTP works.
- Configure in-band ZTP and out-of-band ZTP.

Course Agenda

Day 1

- Chapter 1: Course Introduction
- Chapter 2: Junos Automation Architecture and Overview
- Chapter 3: NETCONF and the XML API
 - Lab 1: Exploring the XML API

Day 2

- Chapter 4: JSON and YAML
 - Lab 2: Using JSON and YAML
- Chapter 5: Python and Junos PyEZ
 - Lab 3: Implementing Python and Junos PyEZ in Junos
- Chapter 6: Jinja2 and Junos PyEZ
 - Lab 4: Using Jinja2 Templates with PyEZ

Day 3

- Chapter 7: Using Ansible to Automate Junos
 - Lab 5: Automating Junos with Ansible
- Chapter 8: Junos Automation with JSNAPy
 - Lab 6: Using JSNAPy

Day 4

- Chapter 9: Junos OS Commit and Op Scripts
 - Lab 7: Junos Commit and Op Scripts
- Chapter 10: Junos OS Event and SNMP Scripts
 - Lab 8: Junos Event Policies and Scripts
- Chapter 11: YANG
- Chapter 12: OpenConfig
 - Lab 9: Implementing OpenConfig

Day 5

- Chapter 13: Junos Extension Toolkit (JET)
- Chapter 14: The Junos OS REST API
 - Lab 10: Implementing the Junos REST API
- Appendix A: Zero Touch Provisioning

Document Conventions

CLI and GUI Text

Frequently throughout this course, we refer to text that appears in a command-line interface (CLI) or a graphical user interface (GUI). To make the language of these documents easier to read, we distinguish GUI and CLI text from chapter text according to the following table.

Style	Description	Usage Example
Franklin Gothic	Normal text.	Most of what you read in the Lab Guide and Student Guide.
Courier New	<p>Console text:</p> <ul style="list-style-type: none">• Screen captures• Noncommand-related syntax <p>GUI text elements:</p> <ul style="list-style-type: none">• Menu names• Text field entry	<p>commit complete</p> <p>Exiting configuration mode</p> <p>Select File > Open, and then click Configuration.conf in the Filename text box.</p>

Input Text Versus Output Text

You will also frequently see cases where you must enter input text yourself. Often these instances will be shown in the context of where you must enter them. We use bold style to distinguish text that is input versus text that is simply displayed.

Style	Description	Usage Example
Normal CLI	No distinguishing variant.	Physical interface:fxp0, Enabled
Normal GUI		View configuration history by clicking Configuration > History.
CLI Input	Text that you must enter.	lab@San_Jose> show route
GUI Input		Select File > Save, and type config.ini in the Filename field.

Undefined Syntax Variables

Finally, this course distinguishes syntax variables, where you must assign the value (undefined variables). Note that these styles can be combined with the input style as well.

Style	Description	Usage Example
<u>CLI Undefined</u>	Text where the variable's value is the user's discretion or text where the variable's value as shown in the lab guide might differ from the value the user must input according to the lab topology.	Type set policy <u>policy-name</u>. ping 10.0.<u>x.y</u>
<u>GUI Undefined</u>		Select File > Save, and type <u>filename</u> in the Filename field.

Additional Information

Education Services Offerings

You can obtain information on the latest Education Services offerings, course dates, and class locations from the World Wide Web by pointing your Web browser to: <http://www.juniper.net/training/education/>.

About This Publication

This course was developed and tested using the software release listed on the copyright page. Previous and later versions of software might behave differently so you should always consult the documentation and release notes for the version of code you are running before reporting errors.

This document is written and maintained by the Juniper Networks Education Services development team. Please send questions and suggestions for improvement to training@juniper.net.

Technical Publications

You can print technical manuals and release notes directly from the Internet in a variety of formats:

- Go to <http://www.juniper.net/techpubs/>.
- Locate the specific software or hardware release and title you need, and choose the format in which you want to view or print the document.

Documentation sets and CDs are available through your local Juniper Networks sales office or account representative.

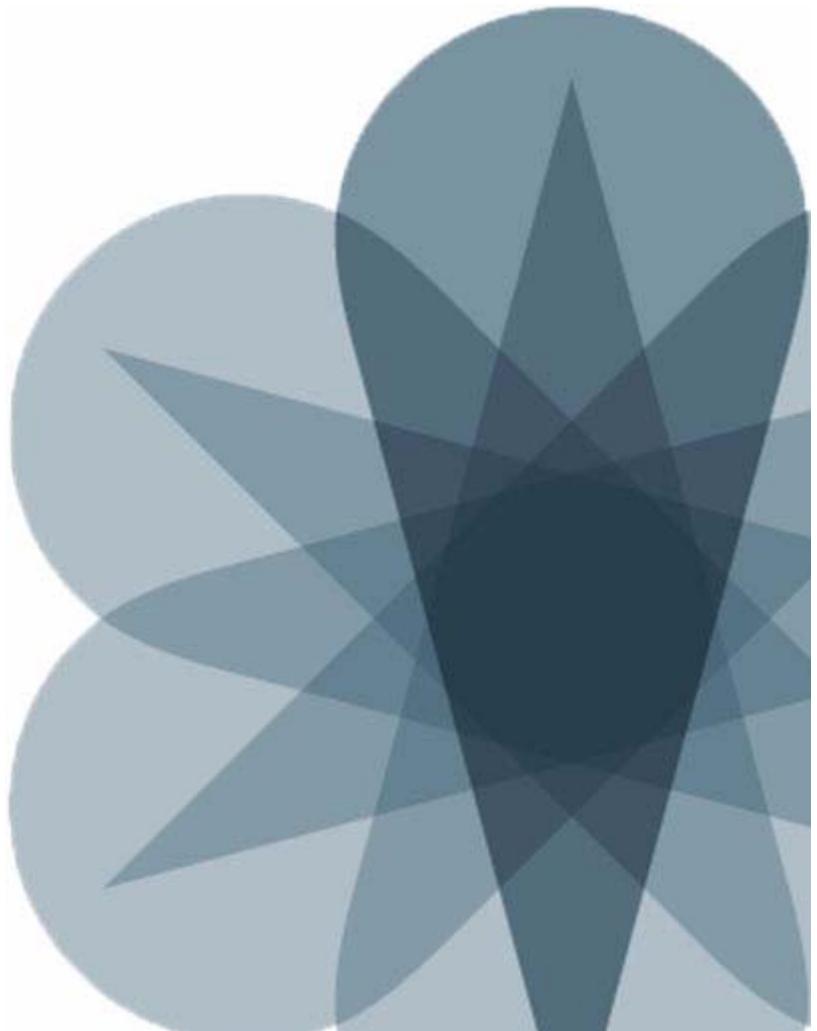
Juniper Networks Support

For technical support, contact Juniper Networks at <http://www.juniper.net/customers/support/>, or at 1-888-314-JTAC (within the United States) or 408-745-2121 (outside the United States).



Junos Platform Automation and DevOps

Chapter 1: Course Introduction



Objectives

- After successfully completing this content, you will be able to:
 - Get to know one another
 - Identify the objectives, prerequisites, facilities, and materials used during this course
 - Identify additional Education Services courses at Juniper Networks
 - Describe the Juniper Networks Certification Program

We Will Discuss:

- Objectives and course content information;
- Additional Juniper Networks, Inc. courses; and
- The Juniper Networks Certification Program.

Introductions

- Before we get started...
 - What is your name?
 - Where do you work?
 - What is your primary role in your organization?
 - What kind of network experience do you have?
 - Are you certified on Juniper Networks?
 - What is the most important thing for you to learn in this training session?



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER NETWORKS Worldwide Education Services

www.juniper.net | 3

Introductions

The slide asks several questions for you to answer during class introductions.

Course Contents (1 of 2)

- **Contents:**

- Chapter 1: Introduction
- Chapter 2: Junos Automation Architecture and Overview
- Chapter 3: NETCONF and the XML API
- Chapter 4: JSON and YAML
- Chapter 5: Python and Junos PyEZ
- Chapter 6: Jinja2 and Junos PyEZ
- Chapter 7: Using Ansible to Automate Junos
- Chapter 8: Junos Automation with JSNAPy

Course Contents: Part One

The slide lists the topics we discuss in this course.

Course Contents (2 of 2)

- **Contents:**

- Chapter 9: Junos OS Commit and Op Scripts
- Chapter 10: Junos OS Event and SNMP Scripts
- Chapter 11: YANG
- Chapter 12: OpenConfig
- Chapter 13: Junos Extension Toolkit (JET)
- Chapter 14: The Junos OS REST API
- Appendix A: Zero Touch Provisioning

Course Contents: Part Two

The slide lists the topics we discuss in this course.

Prerequisites

- The prerequisites for this course are the following:
 - Intermediate-level networking knowledge and an understanding of the Open Systems Interconnection (OSI) model and the TCP/IP protocol suite
 - Familiarity XML basics and have introductory knowledge of the Python programming language
 - Attend the *Introduction to the Junos Operating System* (IJOS) course prior to attending this class
 - High-level understanding of object-oriented programming is a plus but not a requirement

Prerequisites

The slide lists the prerequisites for this course.

Course Administration

- The basics:

- Sign-in sheet
- Schedule
 - Class times
 - Breaks
 - Lunch
- Break and restroom facilities
- Fire and safety procedures
- Communications
 - Telephones and wireless devices
 - Internet access



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 7

General Course Administration

The slide documents general aspects of classroom administration.

Education Materials

- Available materials for classroom-based and instructor-led online classes:
 - Lecture material
 - Lab guide
 - Lab equipment
- Self-paced online courses also available
 - <http://www.juniper.net/courses>



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER NETWORKS Worldwide Education Services

www.juniper.net | 8

Training and Study Materials

The slide describes Education Services materials that are available for reference both in the classroom and online.

Additional Resources

- For those who want more:

- Juniper Networks Technical Assistance Center (JTAC)
 - <http://www.juniper.net/support/requesting-support.html>
- Juniper Networks books
 - <http://www.juniper.net/books>
- Hardware and software technical documentation
 - Online: <http://www.juniper.net/techpubs>
 - Portable libraries: <http://www.juniper.net/techpubs/resources>
- Certification resources
 - <http://www.juniper.net/certification>

© 2017 Juniper Networks, Inc. All rights reserved.

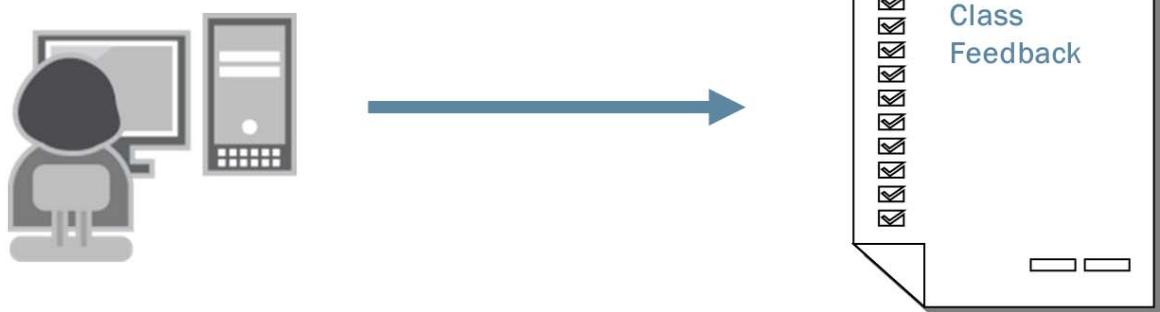
 Worldwide Education Services

www.juniper.net | 9

Additional Resources

The slide provides links to additional resources available to assist you in the installation, configuration, and operation of Juniper Networks products.

Satisfaction Feedback



- To receive your certificate, you must complete the survey
 - Either you will receive a survey to complete at the end of class, or we will e-mail it to you within two weeks
 - Completed surveys help us serve you better!

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 10

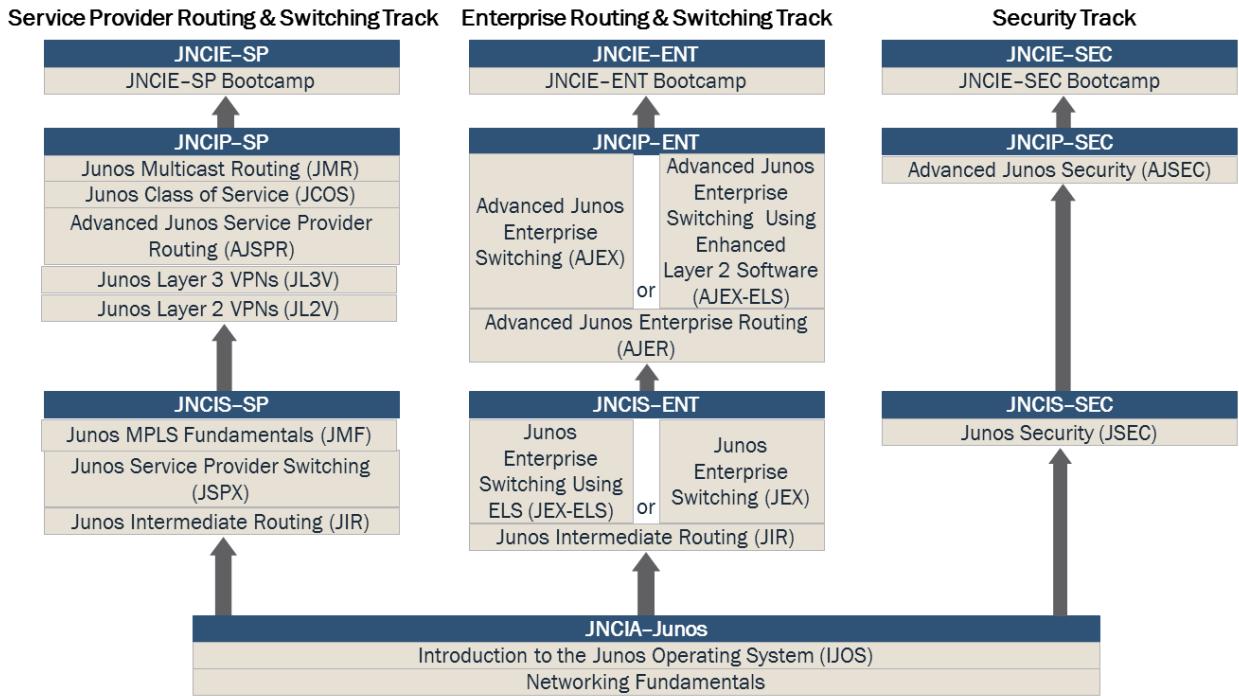
Satisfaction Feedback

Juniper Networks uses an electronic survey system to collect and analyze your comments and feedback. Depending on the class you are taking, please complete the survey at the end of the class, or be sure to look for an e-mail about two weeks from class completion that directs you to complete an online survey form. (Be sure to provide us with your current e-mail address.)

Submitting your feedback entitles you to a certificate of class completion. We thank you in advance for taking the time to help us improve our educational offerings.

Juniper Networks— Junos-Based Curriculum

Certification
 Course



Notes: Information current as of April 2017. Course and exam information (length, availability, content, etc.) is subject to change; refer to www.juniper.net/training for the most current information.

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 11

Juniper Networks Education Services Curriculum

Juniper Networks Education Services can help ensure that you have the knowledge and skills to deploy and maintain cost-effective, high-performance networks for both enterprise and service provider environments. We have expert training staff with deep technical and industry knowledge, providing you with instructor-led hands-on courses in the classroom and online, as well as convenient, self-paced eLearning courses. In addition to the courses shown on the slide, Education Services offers training in automation, E-Series, firewall/VPN, IDP, network design, QFabric, support, and wireless LAN.

Courses

Juniper Networks courses are available in the following formats:

- Classroom-based instructor-led technical courses
- Online instructor-led technical courses
- Hardware installation eLearning courses as well as technical eLearning courses
- Learning bytes: Short, topic-specific, video-based lessons covering Juniper products and technologies

Find the latest Education Services offerings covering a wide range of platforms at http://www.juniper.net/training/technical_education/.

Juniper Networks Certification Program

- Why earn a Juniper Networks certification?
 - Juniper Networks certification makes you stand out
 - Demonstrate your solid understanding of networking technologies
 - Distinguish yourself and grow your career
 - Capitalize on the promise of the New Network
 - Develop and deploy the services you need
 - Lead the way and increase your value
 - Unique benefits for certified individuals



CERTIFICATION
PROGRAM



© 2017 Juniper Networks, Inc. All rights reserved.

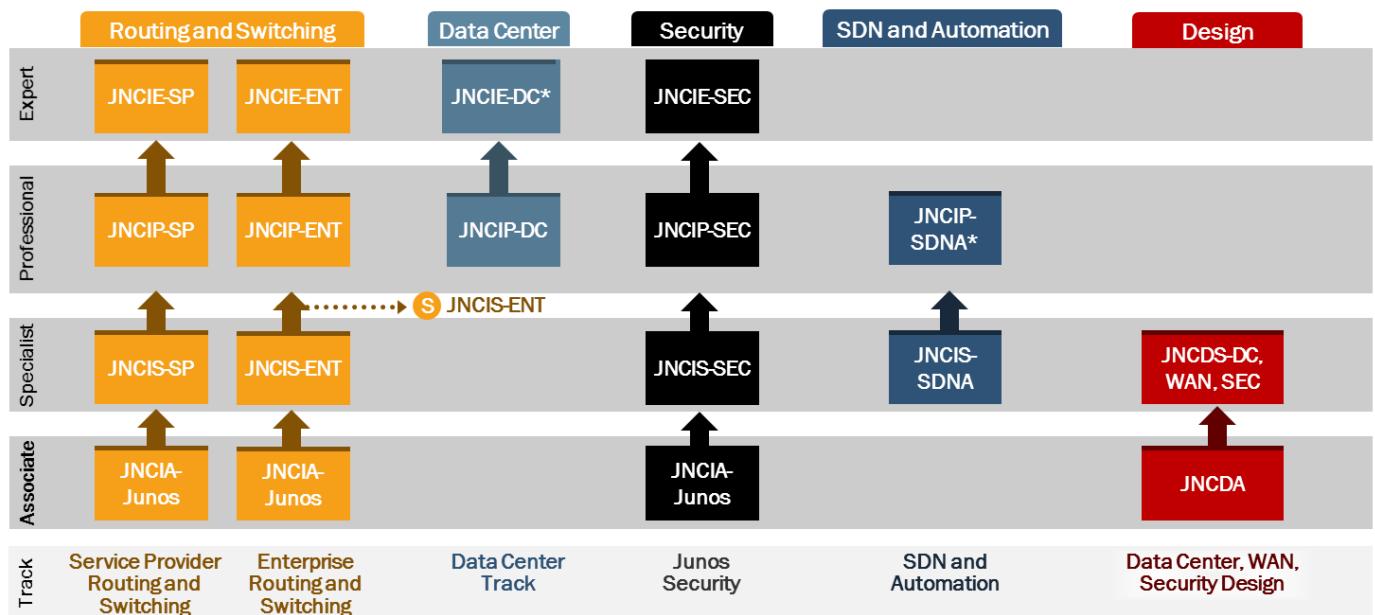
JUNIPER Worldwide Education Services

www.juniper.net | 12

Juniper Networks Certification Program

A Juniper Networks certification is the benchmark of skills and competence on Juniper Networks technologies.

Juniper Networks Certification Program Framework



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 13

Juniper Networks Certification Program Overview

The Juniper Networks Certification Program (JNCP) consists of platform-specific, mult-tiered tracks that enable participants to demonstrate competence with Juniper Networks technology through a combination of written proficiency exams and hands-on configuration and troubleshooting exams. Successful candidates demonstrate a thorough understanding of Internet and security technologies and Juniper Networks platform configuration and troubleshooting skills.

The JNCP offers the following features:

- Multiple tracks;
- Multiple certification levels;
- Written proficiency exams; and
- Hands-on configuration and troubleshooting exams.

Each JNCP track has one to four certification levels—Associate-level, Specialist-level, Professional-level, and Expert-level. The Associate-level, Specialist-level, and Professional-level exams are computer-based exams composed of multiple choice questions administered at Pearson VUE testing centers worldwide.

Expert-level exams are composed of hands-on lab exercises administered at select Juniper Networks testing centers. Please visit the JNCP website at <http://www.juniper.net/certification> for detailed exam information, exam pricing, and exam registration.

Certification Preparation

- Training and study resources:

- Juniper Networks Certification Program website:
www.juniper.net/certification
- Education Services training classes:
www.juniper.net/training
- Juniper Networks documentation and white papers:
www.juniper.net/techpubs

- Community:

- J-Net:
http://forums.juniper.net/t5/Training-Certification-and/bd-p/Training_and_Certification
- Twitter: @JuniperCertify

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 14

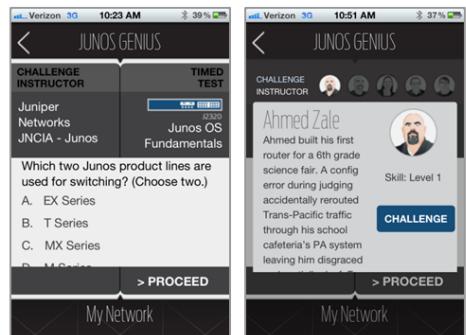
Preparing and Studying

The slide lists some options for those interested in preparing for Juniper Networks certification.

Junos Genius: Certification Preparation App

Unlock your Genius...

- Practice for multiple exams in *Study Mode*
 - Hundreds of multiple choice questions and answer explanations, many with CLI snapshots
- Simulate an exam in *Time Challenge Mode*
- Earn device achievements by winning in *Instructor Challenge Mode*
- Build a virtual network with device achievements
- Track your results in the app and Game Center; share your network through Facebook and Twitter



www.juniper.net/junosgenius

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER Worldwide Education Services

www.juniper.net | 15

Junos Genius

The Junos Genius application takes certification exam preparation to a new level. With Junos Genius you can practice for your exam with flashcards, simulate a live exam in a timed challenge, and even build a virtual network with device achievements earned by challenging Juniper instructors. Download the app now and *Unlock your Genius today!*

Find Us Online



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



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



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



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

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER Worldwide Education Services
NETWORKS

www.juniper.net | 16

Find Us Online

The slide lists some online resources to learn and share information about Juniper Networks.

Questions



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 17

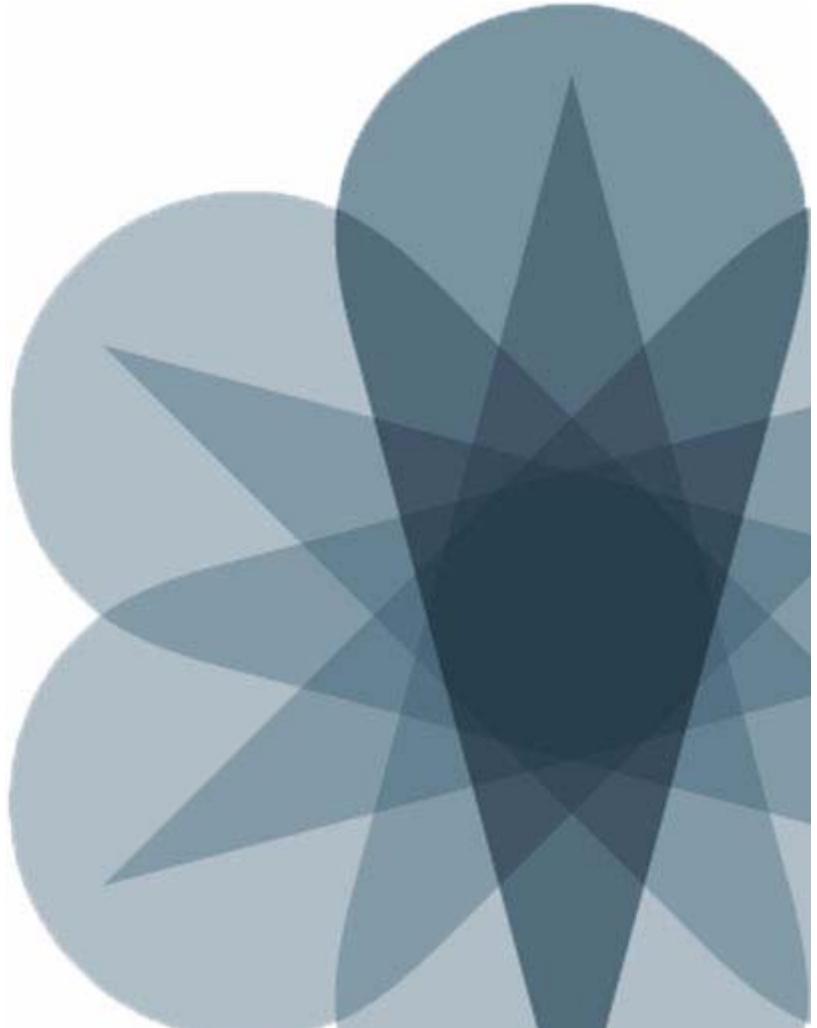
Any Questions?

If you have any questions or concerns about the class you are attending, we suggest that you voice them now so that your instructor can best address your needs during class.



Junos Platform Automation and DevOps

Chapter 2: Junos Automation Architecture and Overview



Objectives

- After successfully completing this content, you will be able to:
 - Describe the Junos Architecture and Automation UI
 - Explain the role of gRPC, NETCONF, and REST in Junos Automation
 - Identify the languages, frameworks, management suites, and tools commonly used in automating Junos

We Will Discuss:

- The Junos OS architecture and automation UI;
- The gRPC, NETCONF, and REST APIs; and
- The languages, frameworks, management suites and tools used to automate the Junos OS.

Agenda: Junos Automation Architecture and Overview

→Why Automate?

- Junos mgd-based Automation
- Junos jsd-based Automation
- Automation Languages, Libraries, and Frameworks
- Automation Management Systems
- Junos Automation Tools

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 3

Why Automate?

The slide lists the topics we will discuss. We discuss the highlighted topic first.

Why Automate?

- Configuration Management
- Simplify Complex Network
- Cross Platform Configuration
- Enforce Consistency
- Improve Security
- Improve Efficiency
- Increase Visibility
- Cut Costs



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 4

Why Automate?

Why are you implementing automation in your network? There are many reasons an organization would want to implement automation into their network, but it is important that you take the time to identify which reason or reasons you are implementing automation. As you go through this course, you will see that there are often multiple methods to achieve the same result. Each automation method has its own benefits and drawbacks. By knowing your primary reasons for network automation, you will be able to choose which automation method most closely matches your needs. A list of common reasons that organizations choose to add automation to their network environment is included here.

Implement a Configuration Management Solution – A configuration management solution—such as Ansible, SaltStack, Puppet and Chef—is a great tool that can be used to manage both compute and networking devices. They manage the whole life cycle of the device, including: install, operation, update, and retirement. Configuration management solutions address many of the individual goals listed here. Each configuration management solution has its own set of strengths and weaknesses. In this course, we focus on Ansible because of its wide networking industry acceptance, its broad support for the Junos OS, and that it does not need to have a special agent installed or running on devices running the Junos OS in order to manage them.

Simplify a Complex Network – The vast number and variety of network devices can often cause network administrators to look for better ways to simplify and automate their job tasks.

Cross Platform Configuration – Many organizations operate in a heterogeneous environment with equipment from multiple vendors, each with their own set of commands to complete the same task. Tools such as YAMG and OpenConfig can help organizations standardize and automate configurations across platforms.

Continued on the next page.

Why Automate? (contd.)

Enforce Consistency – With increasing government regulation and compliance issues facing organizations, network administrators are looking to automation for ways to implement consistent, error-free configurations that are simple to document.

Improve Security – Most security issues are caused by improperly configured devices; some turn to automation to not only automate device configuration, but also to employ automation as a means to monitor and more quickly react to security issues. Using automation to identify and react to security issues can include notifying administrators of the issues and automatically deploying counter measures to contain or mitigate issues.

Improve Efficiency – Configurations across devices are often redundant; administrators want to take advantage of automation to reduce configuration time from hours to minutes.

Increase Visibility – Monitoring is of no use if no one reviews the log files. Having automation in place to gather and interpret what is happening in the network can be a tremendous time-saver; the process of expanding visibility into the network is evolving from reading log files to correlating streams of telemetry data being pushed from individual devices in real-time.

Cut Costs – In the end, the motivation for automation boils down to cutting costs. Whether you are saving time, removing redundancy, increasing stability, or improving efficiency, the goal is doing more for less cost.

Agenda: Junos Automation Architecture and Overview

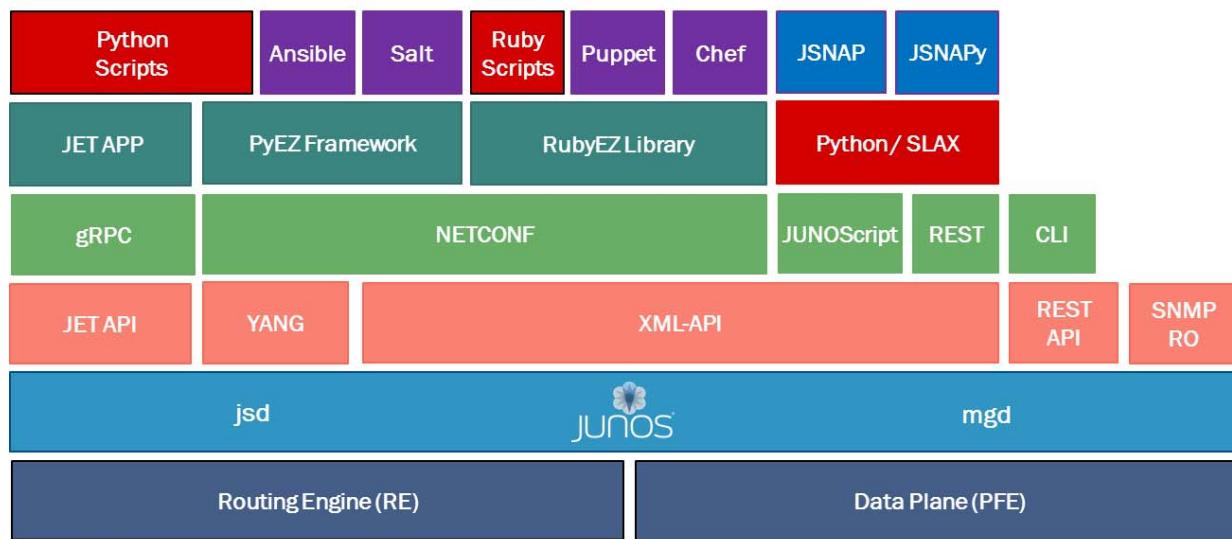
- Why Automate?
- Junos mgd-based Automation
- Junos jsd-based Automaton
- Automation Languages, Libraries, and Frameworks
- Automation Management Systems
- Junos Automation Tools

Junos mgd-Based Automation

The slide highlights the topic we discuss next.

Junos Processes

- Junos supports many forms of automation
- Junos automation ultimately goes through the mgd or jsd processes



© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 7

The Junos Processes

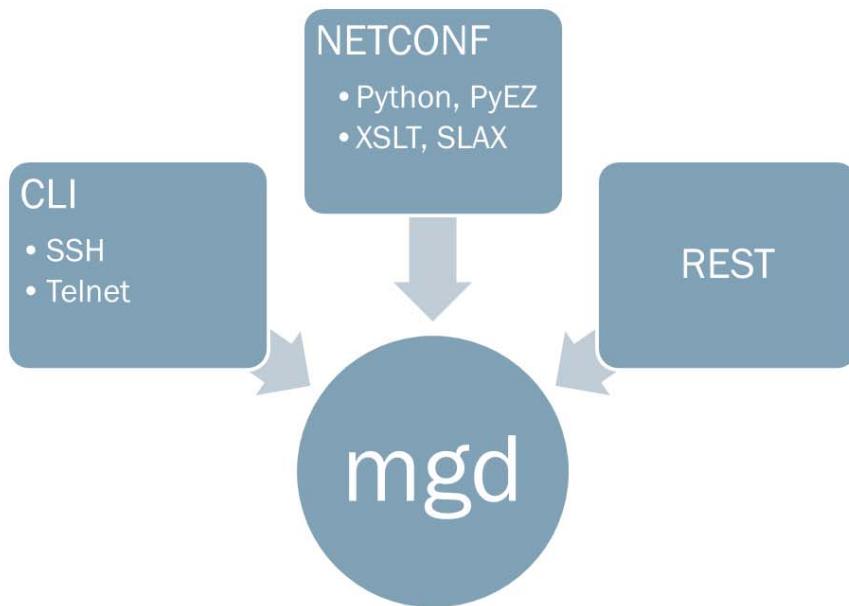
Automation has been part of the Junos OS since its inception. Over the past 15+ years, the Junos OS has developed into a rich automation environment that includes multiple application programming interfaces (APIs), libraries, tools, and supported languages. Before diving into automation it is important to understand the Junos automation architecture.

The slide shows a graphic representing the Junos Automation Stack and all of the various layers, protocols, and languages that the Junos OS supports for automation. We will refer to this graphic occasionally throughout the course. Note that just above the Routing Engine (RE) and the Packet Forwarding Engine (PFE) sits the Junos OS layer. The Junos OS core functions are carried out by processes within the OS. When you automate a device running the Junos OS, at some point—often after going through several other layers—it is these processes that eventually carry out the command or configuration change.

Junos has many different processes. The processes most important to Junos automation are the management process (mgd) and JET service processes (jsd). The Management process (mgd) handles the automation requests involving the Junos XML API, YANG, the REST API, and some SNMP functions. The JET service process (jsd) handles automation requests that use the recently released Juniper Extension Toolkit (JET) API. The next several pages discuss both the mgd and jsd-based forms of automation.

The mgd

- The mgd manages CLI, NETCONF, and REST connections



The mgd

The Management process (mgd) is central to any automation on Junos devices that connect to the Junos OS using NETCONF. As seen in the slide, the CLI and the Junos REST API are also handled through the mgd. We will discuss NETCONF and the REST API in more depth later in the course.

The mgd is just one part of the Junos OS User Infrastructure (UI). In order to understand how Junos automation works, you also need to understand the Junos User Infrastructure and the role that the mgd plays within it. We start with the Junos UI on the next page.

The Junos UI

Database

- Contains all configuration data

Schema

- Defines the layout of the database, as well as all JUNOS commands

mgd

- Manages user input, command validation, and configuration changes

CLI

- Forwards terminal input to mgd, handles command completion, and formats command output

© 2017 Juniper Networks, Inc. All rights reserved.



www.juniper.net | 9

The Junos User Infrastructure

The Junos OS User Infrastructure (UI) includes the code and processes created for users to interact with the Junos OS. Understanding how the UI works will help you understand which of several automation options would work best for you and also help you understand why Juniper touts that Junos is built for automation. Much of the information that follows comes from the writings and blog entries of Phil Shafer, one of the chief architects of the Junos UI.

The Junos UI has four main facets:

- The first facet is the Junos configuration database, which stores the active configuration. There is more than one copy of the configuration database and a text version of the configuration; we will cover those in more depth later on.
- The second facet is the UI schema, which defines the hierarchy of the Junos OS configuration. The schema also contains the details of all Junos OS operational mode commands and related arguments and options that can be executed along with them. In short, every detail of every possible configuration option and Junos command is contained in the schema.
- The management process (mgd) is the nexus of the Junos OS UI and the third facet. It handles input coming from the CLI as well as from NETCONF sessions. The mgd validates that the syntax of the commands and configuration data is correct. The mgd then calls appropriate internal processes as needed to execute operational mode commands or modify a configuration.
- The forth facet is the Junos Os Command Line Interface process (CLI) which hosts telnet, SSH, and console connections, and forwards user input to the mgd. The CLI is also responsible for formatting and displaying information returned back from the mgd.

The Junos DDL and ODL

```

■ attribute hdlc-scrambler {
    hidden support;                                Hidden command
                                                    only needed by support personnel
    help "Enable HDLC scrambling"
    interface-exclude ic-ih-sonet, INTF_CT3, INTF_XGE;
    product t320 m320 m120 MX_SERIES;              Interfaces not to
                                                    enable scrambling
    type enum int {
        choice "enable" {                          Devices this command
                                                    will work on
            define DDLAID_DCD SONET HDLC SCRAMBLER_ENABLE;
            help "Enable default scrambler";
        } choice "x43" {
            define DDLAID_DCD SONET HDLC SCRAMBLER_x43;
            help "Self-Synchronous x^43+1 scrambler";
        } choice "x29" {
            define DDLAID_DCD SONET HDLC SCRAMBLER_x29;
            help "Self-Synchronous x^29+1 scrambler";
        }
    }
}

```

Command Description

Choice of three possible scrambling algorythms

The Junos DDL and ODL Languages and their Historical Context

At the core of the Junos XML API and the UI are the Junos DDL and ODL languages. When Junos was in its infancy, Juniper took a lot of heat because the company didn't follow the pack and create a Cisco-like CLI. Instead, Juniper struck out in a different direction and created a configuration and command set based on a hierarchical data model that we now call the Junos schema. Data models are an organized set of hierarchical data that establishes the connections between related pieces of information. The Junos schema is implemented using the DDL and ODL languages. Despite the early criticism of this data model, it is the heart of Junos and has proven to be the 'secret sauce' that makes the Junos OS so popular today. This data model is the reason that Junos is inherently built for automation. Let's take a closer look at the DDL and ODL, which establish the Junos data model.;

This slide shows a sample of the DDL. Users don't interact with the DDL directly; it is used by Juniper Junos programmers. When new features and commands are added to Junos, those changes must be reflected in the CLI and in any APIs used for automation that are supported by Junos. The Juniper UI team implemented a modeling language, now called the Data Definition Language (DDL) that allows the Junos OS developers, but not the UI team, to implement new operational mode commands and configuration statements into the Junos OS.

The DDL allows programmers to include all of the information necessary to implement a feature or command. It includes where a command or configuration option fits within the command or configuration hierarchy, what Junos devices the command or feature runs on, the other Junos OS processes that are called to execute a command, any mandatory or optional parameters, acceptable input data, and the name by which the command is referred to in the XML API.

Continued on the next page.

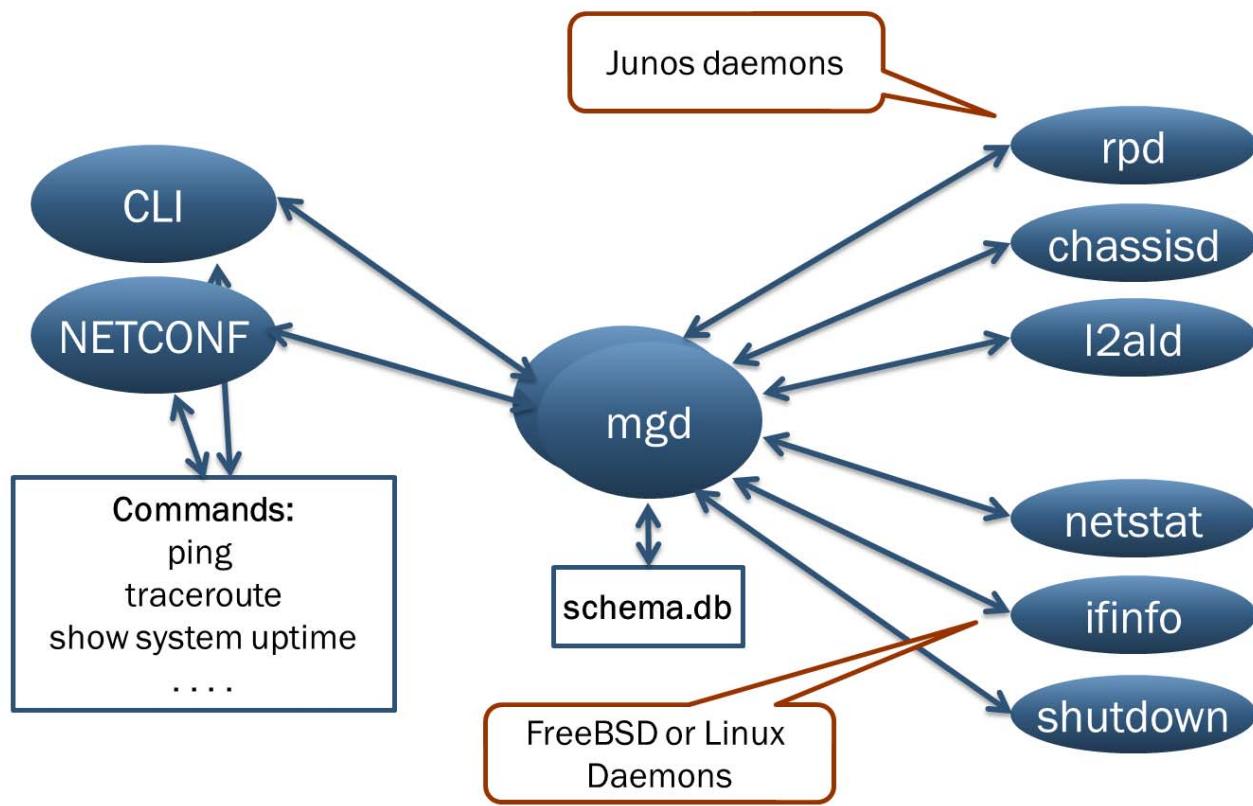
The Junos DDL and ODL Languages and their Historical Context (contd.)

Once the developer has added the requisite information into the DDL, the job of the UI team is 90% complete. After compilation, the DDL files are loaded at device boot and become the schema. There is no additional UI layer or process that needs to be added on top of the schema. There is also no additional preparation that needs to be done to make the command or configuration feature available over a remote NETCONF session. It was previously mentioned that the XML name is added to the DDL file when a feature is created; this is all that needs to be done to make a feature available for automation. In essence, the CLI and a remote automation session both use the same schema, mgd, and configuration database. So, at the same time a command or configuration feature is available to the CLI, it is also available for automation.

The output definition language ODL is similar in syntax to the DDL but different in purpose. Whenever the mgd responds to a request or whenever a snippet of configuration is returned to the CLI, the information is returned as an XML document. The ODL gives information to the CLI about how the information should be displayed. The Junos OS CLI process has a rendering engine that takes XML output and the instructions from the ODL for a command and generates the ASCII output for the CLI. Like the DDL, the ODL is not something with which users interact.

This explains in brief how Junos is built for automation. The next slides detail how Junos commits a configuration and processes commands.

Operational Mode Command Processing



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 12

The Junos OS Operational Command Processing

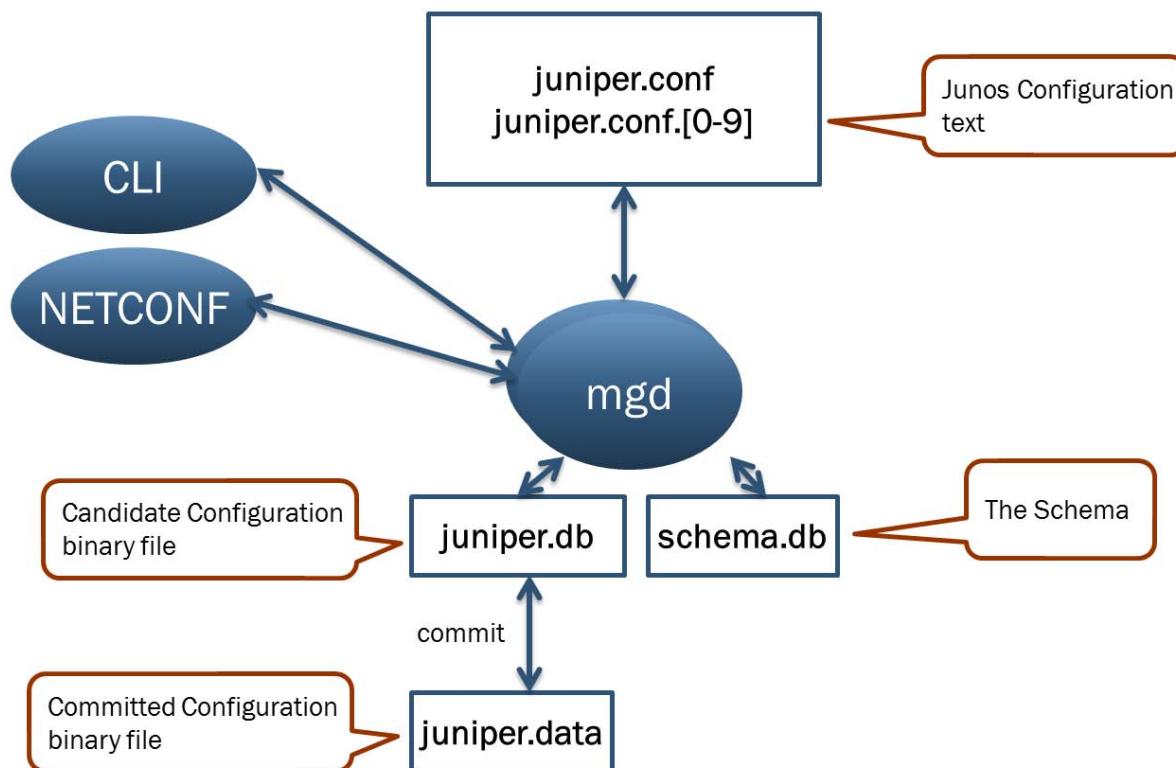
When a user issues a command—such as the ones listed in the slide—through a console session, telnet, or SSH, the command is passed on to the mgd. The mgd queries the schema that resides in the schema.db file and validates the command and any included parameters. Once the command is validated, it is executed by the mgd. Remote automation connections that use NETCONF sessions use an almost identical process; the difference is that a NETCONF session issues commands as specified by the Junos OS XML API.

For example, if you issue the show interfaces command in the CLI, the mgd validates the command syntax and then executes the command. This occurs by invoking the interface information process—known as the ifinfo process—in the underlying OS. To issue the same command through a NETCONF session, you would issue the get-interface-information XML RPC. When the mgd receives the get-interface-information XML API call, the mgd queries the schema for the get-interface-information API, maps it to the equivalent CLI command, and then executes it. Other than looking up the XML API, the process is the same.

Once a command is issued, the appropriate process returns the result in an XML document. This applies to commands issued in the CLI or in a NETCONF session. The NETCONF client application and server communicate using XML, however, the CLS renders the data in a more human-readable form.

The CLI uses a rendering engine along with the ODL schema to convert the XML document returned by the MGD into a readable format. The purpose of the ODL is to inform the rendering engine how to convert the XML-formatted command output into readable text. When the CLI receives an XML document from the mgd, it instructs the rendering engine to format the output, the rendering engine looks up the command in the ODL schema, and then renders the output to the screen according to the formatting instructions in the ODL.

Junos Configuration Commit Process



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 13

The Junos OS Configuration Commit Process

When the Junos OS boots, it loads the text-based configuration file `juniper.conf`. As part of the load process the Junos OS has to validate against the schema stored in the `schema.db` file. If validated, the configuration is compiled and stored in the `juniper.data` file for fast lookups. The `juniper.data` file is a read-only file; when a user enters configuration mode, a candidate copy of the configuration is placed in `juniper.db` and this file is what a user edits.

When a user or remote session commits a configuration, the `mgd` validates the contents of the `juniper.db` file against the schema stored in the `schema.db` file to ensure that any changes to the schema don't break the configuration. If the `mgd` finds an error, the error is reported and the commit fails. If the changes pass validation, the `mgd` notifies all affected processes and the candidate configuration in the `juniper.db` file becomes the active configuration, which is stored in the `juniper.data` file. Finally, the rollback files are created.

This explanation is a summary of the commit process, but illustrates in brief how the schema and the `mgd` are central to the UI and shows that the process is the same whether interacting with Junos from the CLI or from a remote automated session.

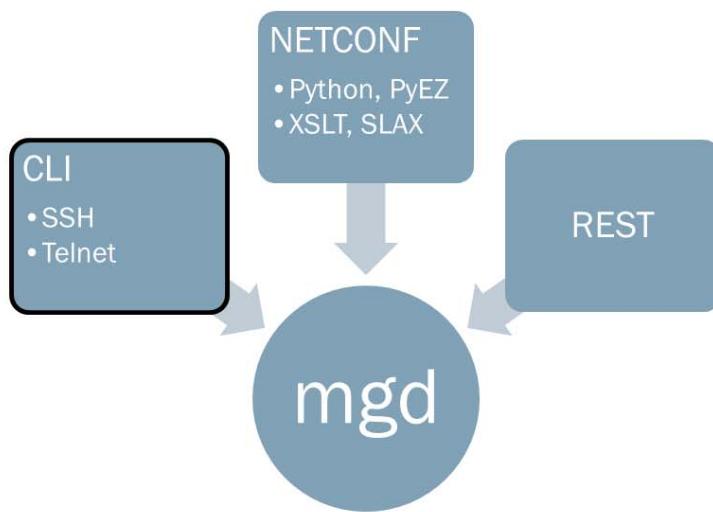
The `mgd` is central to much of what happens in the Junos OS. The `mgd` consults the schema located in the `Schema.db` file whenever:

- A configuration is parsed or exported
- A CLI user uses auto-completion or context-sensitive help
- A configuration database is read by a process
- Commands are executed in operational mode

Junos CLI

- The CLI handles:

- Console Connections
- Telnet
- SSH



The Junos CLI

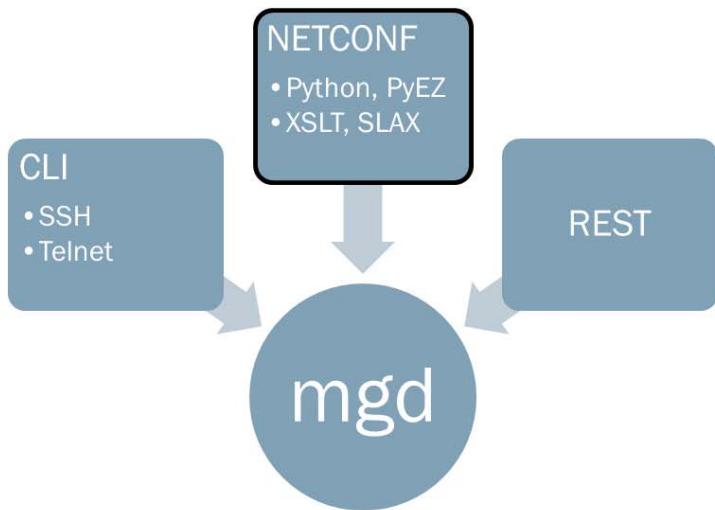
The next few pages will discuss the Junos CLI, NETCONF, and REST connections and how they relate to the mgd and Junos UI.

Several of the Junos CLI features stem from Junos being based on an object model and having a schema. While our message here is on methods of automation, many are unaware of the shortcuts and niceties within the Junos CLI that make configuration and maintenance simpler. If you are interested in learning more, read the Junos CLI Configuration Guide book found here: https://www.juniper.net/documentation/en_US/junos/information-products/pathway-pages/junos-cli/junos-cli.pdf

So even though the Junos CLI is robust, command line interfaces are a comparatively poor method of doing automation. Although CLI automation is possible, most programmers give it up in favor of better automation methods. As such, we will not be discussing using the CLI as a method of Junos automation.

Junos NETCONF Connections

- JUNOScript came first
- NETCONF is the IETF standard
- NETCONF sessions carry automation data for:
 - XSLT
 - SLAX
 - Python
 - PyEZ
 - Other Languages



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER NETWORKS Worldwide Education Services

www.juniper.net | 15

Junos NETCONF Connections

Just as there are many ways of carrying cargo from Los Angeles to Chicago, there are also many different types of protocols a client can use to transport commands and configuration data to a network device. The Network Configuration Protocol (NETCONF) is just one of many protocols that can be used to transport configuration data and operational commands. NETCONF has been Juniper's automation protocol of choice for many years.

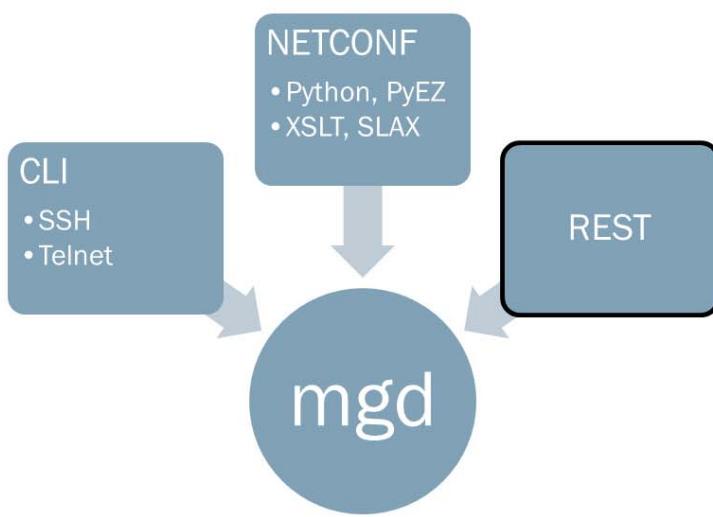
Before NETCONF there was JUNOScript, which has been available since Junos OS release 4.2 in 2001. JUNOScript has been re-branded as "Junos XML Management Protocol," or "Junos XML Protocol" for short. Like NETCONF, Junos XML Protocol is an XML-based network management API for managing devices running Junos.

With input from the Juniper UI team in 2006, Junos XML Protocol (JUNOScript) was modified, improved, then submitted to the IETF for ratification. Junos XML Protocol became NETCONF, a ratified IETF standard with RFC 6241. Since then, there have been modifications and additions to NETCONF by the IETF, all of which are supported by Juniper. NETCONF is the foundation upon which many Juniper automation technologies are built, including: XSLT, SLAX, Python, Junos PyEZ, YANG, and OpenConfig.

Throughout this chapter we will be looking at all of these automation technologies in more detail, but the main takeaway is that all automation using NETCONF is processed by the mgd and is treated the same.

Junos Rest API

- REST is a popular non-network specific Web protocol
- REST API calls initially received by micro web server
- Functionality limited due to limitation of REST and HTTP



Junos REST API

The Junos REST API is a Representational State Transfer (REST) interface that enables you to securely connect to Junos OS devices and execute remote procedure calls. The REST API uses HTTP as the protocol to make calls to the REST API. Because HTTP is a Web protocol and not specifically a network configuration protocol, there are limitations to the automation functionality of using the REST API. These limitations will be discussed more fully in the chapter dedicated to the REST API.

The advantage of the REST API is that the REST protocol is popular for Web applications and is supported by a wide variety of automation systems. The Junos REST API provides responses to REST API commands in a variety of formatting and display options, including JavaScript Object Notation (JSON), plain text, and XML.

Architecturally, the REST API is handled by a micro web server that can handle HTTP and HTTPS connections. The REST API calls are received by the micro web server, filtered through security and other processes, then arrive at the mgd, where they are handled similarly to XML RPC calls.

Agenda: Junos Automation Architecture and Overview

- Why Automate?
- Junos mgd-based Automation
- Junos jsd-based Automation
- Automation Languages, Libraries, and Frameworks
- Automation Management Systems
- Junos Automation Tools

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

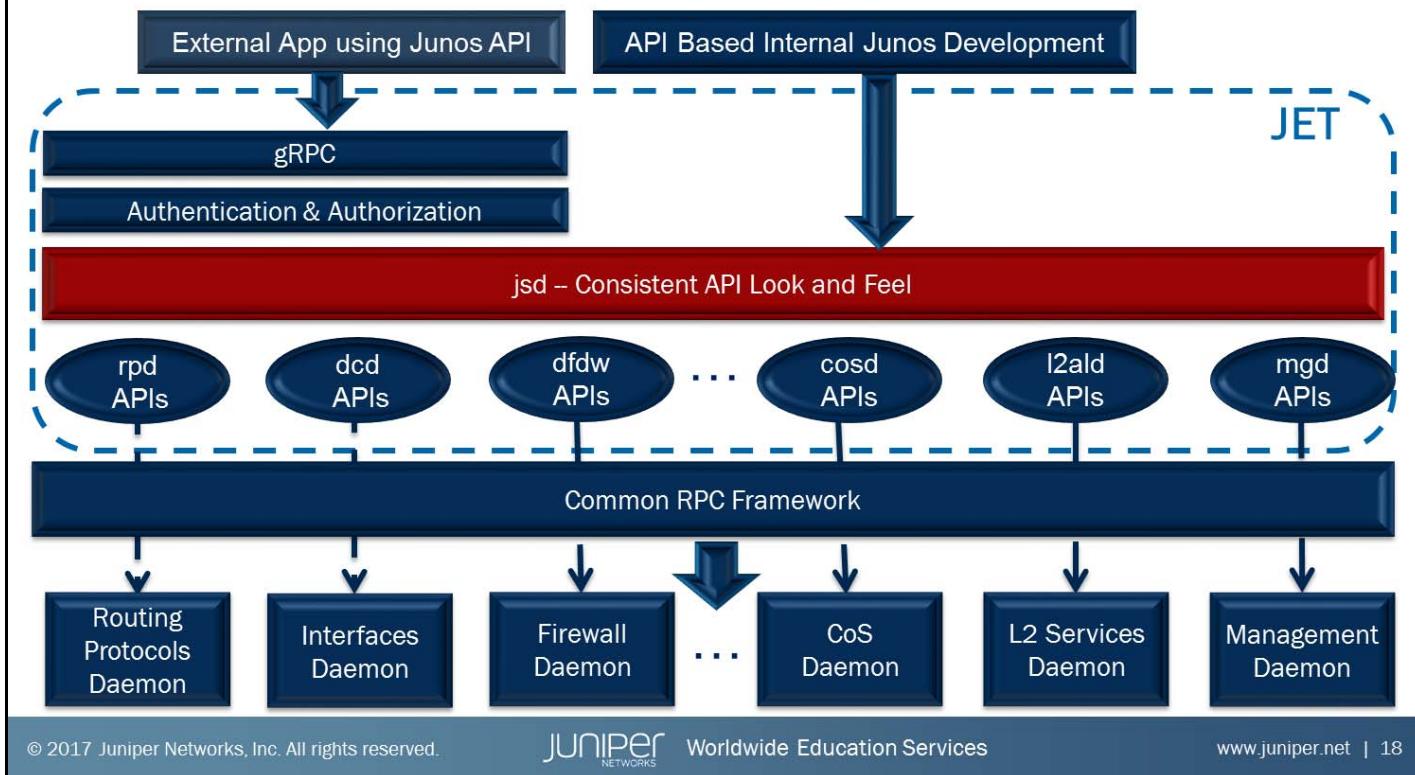
www.juniper.net | 17

Junos jsd-Based Automation

The slide highlights the topic we discuss next.

JET Service Process (jsd)

- The jsd exposes internal APIs for automation



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 18

The JET Service Process

As we have seen in the last several slides, all communication through the CLI, NETCONF, and REST connections goes through the mgd process (daemon). As Junos becomes more modularized, an additional automation process has been added to Junos: the JET Service process (jsd). The jsd supports an additional set of APIs; developers can now use the Juniper Extension Toolkit and use what are called the JET APIs to automate the Junos OS.

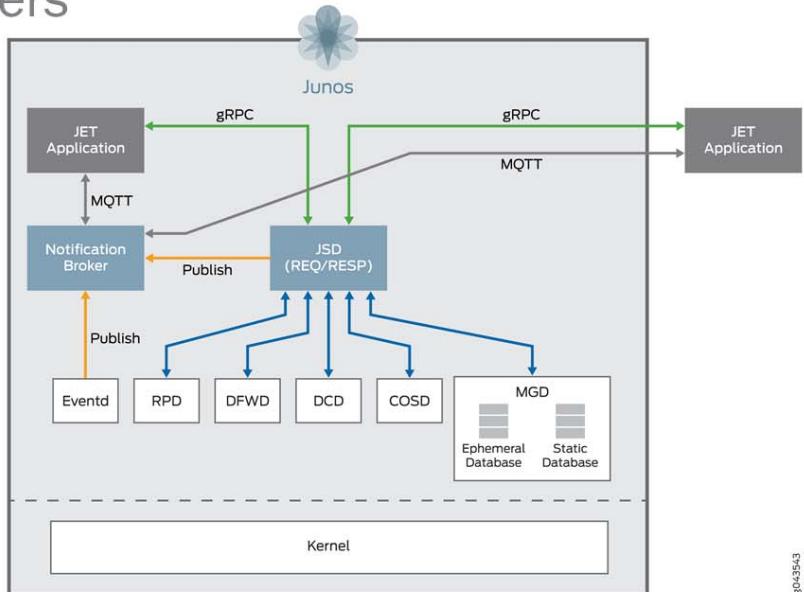
The jsd process aggregates the internal APIs belonging to Routing Protocol Process (rpp), the Firewall Filter Process (ffp), and other individual Junos processes, and then exposes their individual APIs with a consistent look and feel. These API remote procedure calls (RPCs) are transported using the gRPC protocol instead of NETCONF. The addition of the JET API now gives users a second set of APIs to use for automation. Some of the benefits of the JET APIs include faster commit times, improved device telemetry, and a wider range of possible languages available for automation. All of these will be covered in more depth when discussing JET.process

The JET APIs are a recent development and many of the RPCs for the JET API are still being added. We will cover the APIs available as of Junos release 16.2.

An additional point worth noting from the slide is that the jsd exposes users to the same APIs that Juniper developers use for internal development. This enables developers to create the Juniper Extension Toolkit name applications and automations as Juniper's internal developers do.

JET Architecture

- The jsd and the MQTT Notification Broker are separate services that run on different TCP port numbers



3/25/17

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 19

JET Architecture

The JET architecture is facilitated by two external connection points: the jsd, which takes automation requests and gives back related responses, and the Notification Broker, which handles data streaming or telemetry data. Each of these connections uses a separate protocol and a separate TCP port. As shown in the slide, connections to the jsd use the gRPC protocol and port number 32767. Connections to the Notification Broker use the MQTT protocol and standard MQTT TCP port number of 1883.

As a side note, notice that there are two JET Applications listed in the slide. One is running on the Junos device, also called on-box, and the other is outside of the Junos device, or off-box. This is one of the benefits of JET; the architecture contains all of the necessary compilers, interpreters, libraries, and processes to run JET applications, both on-box and off-box.

Note: Starting with Junos OS Release 16.2R1, JET no longer supports the Apache Thrift software framework for JET API remote procedure calls (RPCs). The gRPC framework replaces the functionality previously supplied by Thrift.

Agenda: Junos Automation Architecture and Overview

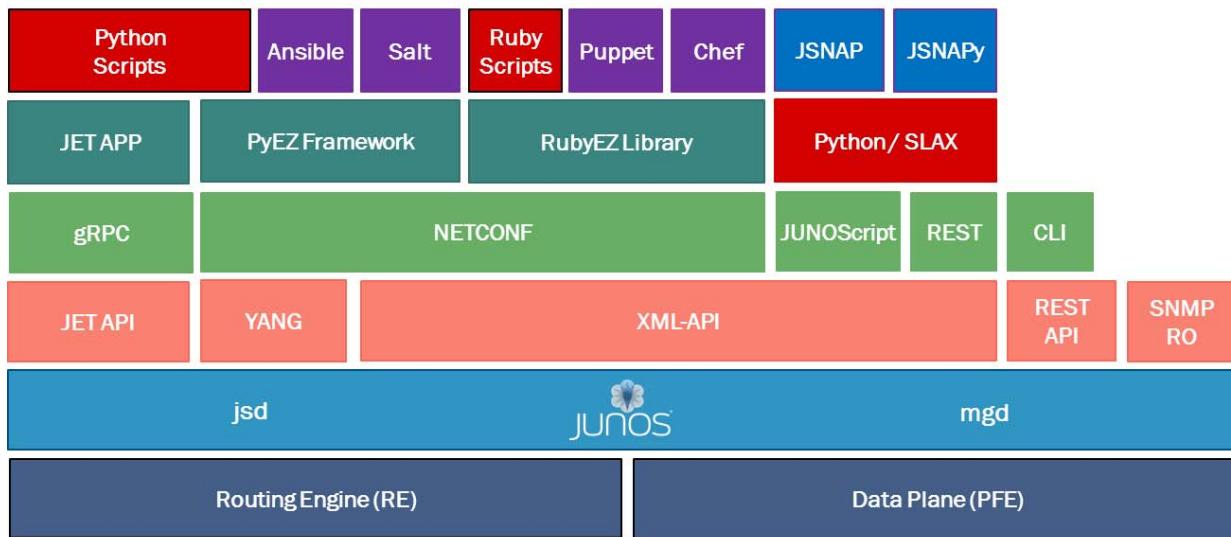
- Why Automate?
- Junos MGD-based Automation
- Junos jsd-based Automation
- Automation Languages, Libraries, and Frameworks
- Automation Management Systems
- Junos Automation Tools

Automation Languages, Libraries, and Frameworks

The slide highlights the topic we discuss next.

Automation Architecture Review

- Calls to JET API go through gRPC connection to jsd
- Calls to XML API and YANG go through NETCONF to mgd
- REST API calls go through REST indirectly to mgd



© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 21

Automation Architecture Review

At this point, you have a basic understanding of the underlying architecture of the Junos OS and its APIs. Let's review and then move to the upper layers of the Junos Automation Stack.

The Junos Management process (mgd) is the gateway process for the majority of Junos automation and the CLI. The mgd hosts NETCONF and Junos XML Protocol (JUNOScript) connections so that developers can use XML or YANG to make calls to the XML API and YANG APIs. Junos XML Protocol has been replaced by NETCONF, but is kept for backwards compatibility. The mgd also indirectly handles REST connections and calls made to the REST API.

The Jet Services process (jsd) handles calls made to the JET API. Connections to the jsd are made directly by using the gRPC protocol. Telemetry connections are also handled by the jsd indirectly. Telemetry connections are first made to an MQTT broker, that then connects to the jsd to pull the required information.

SNMP queries are handled by a separate SNMP agent residing on the Junos Device.

Now that we have all of these APIs, let's talk about what languages and frameworks we can use to take advantage of them.

Programming Languages

- Connection type dictates programming language

NETCONF	gRPC	REST
<ul style="list-style-type: none"> • Python / PyEZ • Perl • Ruby • Java • SLAX / XSLT • Others ... 	<ul style="list-style-type: none"> • Python • Others Possible – Need to Compile IDL 	<ul style="list-style-type: none"> • Languages that can do HTTP GET and POST

NETCONF

The programming languages available to automate the Junos OS depends upon the connection type. Other than Junos XML Protocol (JUNOScript), NETCONF has been in use the longest and has been the connection method of choice for over ten years. NETCONF also includes a wide array of languages with which to program. Python and the PyEZ is now the primary recommended language for automating Junos over a NETCONF connection. Juniper also has libraries for Perl, Ruby, and Java to aid programmers familiar with those languages. SLAX and XSLT are also still supported as Junos automation languages that use NETCONF. You can also use other languages to send data over NETCONF connections, however, Juniper doesn't have any libraries or support to other languages.

gRPC

The gRPC protocol is “the new kid on the block” and currently Juniper only provides the Python API out-of-the-box. However, you can compile additional Interface Definition Languages (IDLs) that support many different languages. Currently, gRPC supports: C++, Java, Python, Go, Ruby, C#, Node.js, Android Java, Objective C, and PHP.

REST

REST connections are the most widely used type of connection today. With the explosion of the Web, most programming languages have some type of support for the HTTP and HTTPS protocols. We will be using Python when discussing the REST API.

NETCONF Libraries

- NETCONF libraries make programming easier

API Language	Distribution Mode	Maturity	Support	Additional Notes
Ruby	Open Source	Most popular, 3200+ downloads	Open Source	Best ease of installation, packed with features, but has limited dependencies and active support
Java	Juniper website and Github	Already in use by enterprise customers	JTAC	Easy installation, simple start-up, single .jar file to use with zero dependencies
Python	Open Source	Based on a popular open source client	Open Source	Most popular scripting language
Perl	Juniper website	Oldest API; more difficult installation	JTAC	API installation needs simplification

NETCONF Libraries

Many of the tasks needed to work with NETCONF can be automated and simplified. To help users, the following libraries are available for working with NETCONF. We will introduce the Perl and Java libraries in the chapter on NETCONF.

Python On-Box

- Python on-box

- Python 2.7 on the box in Junos 16.1 and later
- Everything that is possible with SLAX can be done with Python
- Part of core Junos image



- Python off-box

- Also uses Python version 2.7 and PyEZ
- Python with PyEZ used with Junos 11.4 and later.

Python On-Box

Starting with Junos 16.1, the Python 2.7 interpreter is installed as part of the Junos OS software. This is useful for programmers who are more comfortable programming in Python than in SLAX. Python on-box programmers are able to take advantage of the Junos PyEZ framework also, which will be described on the next few pages.

The Python on-box capabilities are part of the overall Juniper Extension Toolkit (JET) and as such, can be used to automate using the XML API, the YANG API, the REST API, and the JET API.

Python Off-Box

Python off-box has the same capabilities as Python on-box. Python off-box has traditionally used NETCONF over SSH to connect to devices, but can now use the JET APIs also. Python off-box, using NETCONF over SSH and Junos PyEZ, can be used on versions of Junos as early as Junos 11.4.

PyEZ

- A powerful, easy-to-learn microframework
 - Python framework with easy learning curve
 - Works with any Junos device running 11.4 or later
- Run shell scripts
- Junos automation

Junos PyEZ Microframework

Using Python by itself as a method to create scripts can be a daunting task. Junos PyEZ, and all later instances, are a micro-framework that makes the task of programming and automating devices running Junos OS simpler. Junos PyEZ enables users to manage and automate Junos devices. The user does not need to be a programmer, have sophisticated knowledge of the Junos OS, or have a complex understanding of the Junos XML API.

Python as a Shell Scripting Language

Network engineers can still use the native Python shell on their management server, laptop, tablet, or phone as a point-of-control for remotely managing Junos devices. The Python shell is an interactive environment that provides the necessary means to perform common automation tasks, such as conditional testing, for-loops, macros, and templates. These building blocks are similar to other shell environments like Bash or PowerShell; they enable a non-programmer to use the Python shell as a power tool rather than as a programming language. From the Python shell, a user can manage Junos devices using native hash tables, arrays, and XML.

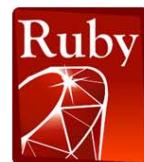
Python for Automaton

In order to automate the network infrastructure into larger IT systems, traditional software programmers and DevOps Engineers need an abstraction library of code to which they can connect their tools. The Junos PyEZ framework is designed for extensibility so that the programmer can quickly and easily add new widgets to the library in support of their specific project requirements. There is no need to wait on a vendor to provide new functionality because the programmer can add it without assistance. In this course, we will use Python and Ansible to show ways to automate larger IT systems.

Ruby and RubyEZ

▪ Ruby

- True object-oriented language
- Like Python, scripts are interpreted, not complied
- No on-box implementation



▪ RubyEZ

- RubyEZ provides classes and methods to allow Ruby programs to control Junos devices
- Data is formatted in XML, transported via NETCONF and sent securely over SSH
- Junos 11.4+ is recommended
- Ruby 1.9.3+ is required

Ruby

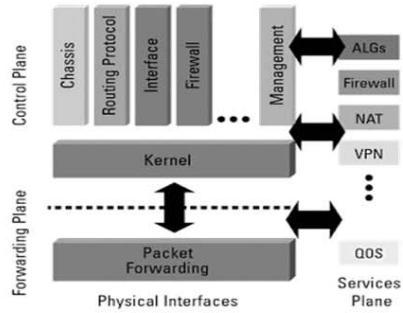
Ruby is a popular scripting language similar to Python. Some programmers prefer Ruby because it is a true object-oriented language. You can accomplish the same results in Ruby as Python when connected remotely through a NETCONF over SSH connection. However, there is not an on-box interpreter for Ruby, and so it cannot be used for on-box scripting.

RubyEZ

RubyEZ is similar in concept to Junos PyEZ and is meant as an automation option for those preferring to program in Ruby. Ruby offers similar libraries as those offered by Junos PyEZ, and using RubyEZ results in a similar code reduction as Junos PyEZ. Like Junos PyEZ, Junos 11.4 and later are recommended. Ruby 1.9.3 is the version of Ruby required for RubyEZ.

If you are new to automation, you might consider learning Junos PyEZ instead of RubyEZ because of its broader support and simpler syntax mastery.

Junos Extension Toolkit (JET)



Enable 3rd party apps to run on Junos

1

Program Junos control plane

2

Enable developers to create customized controls

3

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 27

Enable 3rd Party Apps on Junos

In the past, Junos automation was limited to apps that interact with the Junos OS. Now with JET, Junos can host a third party applications, such as a packet sniffing tool, an on-box Puppet and Chef agent, a Git client, or even a Twitter client that tweets when triggered by specific events.

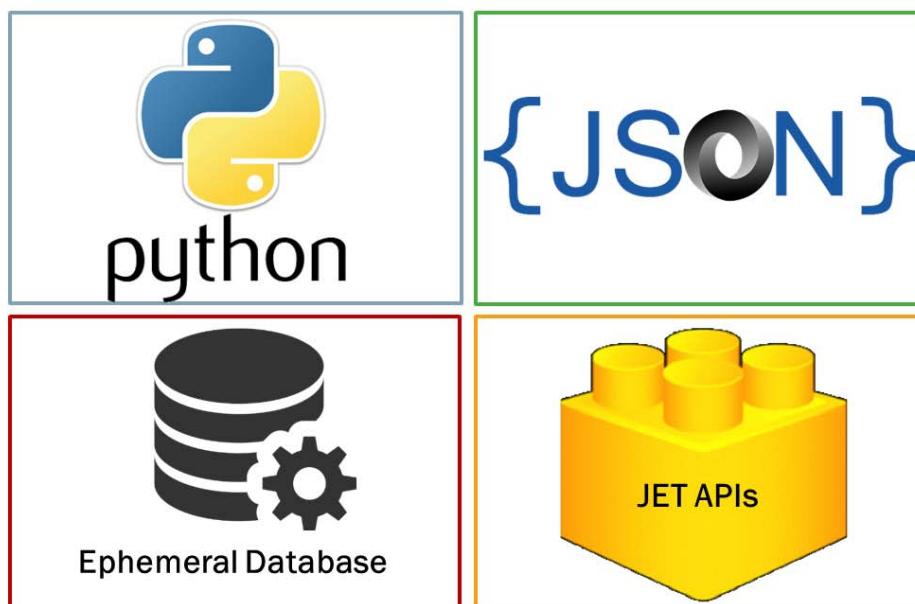
Program Junos Control Plane

JET also adds tools to aid in programming the Junos control plane. The Junos control plane has always been programmable, but now with JET it is easier and faster.

Enable Customer to Create Customized Controls

And finally, JET customers are now able to create customized CLI commands or SNMP MIBs which work together with an app without waiting for Juniper to modify the Junos OS at its next release. This flexibility broadens agility to meet market demands for both Juniper and customers.

JET Components



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 28

Python and Junos PyEZ on Junos

As already stated, Junos now has support for Python 2.7.8 on-box.

JSON on Junos

JavaScript Object Notation (JSON) is a popular data exchange format used by Python programmers. JSON is now an accepted input and output format for Junos. Junos configuration information can now be provided in JSON format. This allows programmers to push configuration information to the Junos device without using XML.

Fast Programmatic Configuration—Ephemeral DB

To keep up with the rapid configuration changes that automation systems make on Junos, the Juniper Extension Toolkit includes the Fast Programmatic Configuration feature, which is often referred to as the Ephemeral DB. The Ephemeral DB can commit over 1,000 changes per second. In order to reach this magnitude of commits per second, commits are not validated. This means that you need to insure that the app is pushing a valid configuration.

JET API

The JET API set consists of Route APIs, Interface APIs, Firewall APIs, and Management APIs. There are also Notifications included in the JET APIs. Notifications are used to send notifications of Junos events, such as additions and deletions, or interface changes, such as when an interface goes up or down.

JET Development Environment

- Installed as a Vagrant VM
- Hosted on Ubuntu
- Required for applications with C or C++ dependencies
- Needed to sign applications

JET Development Environment

If an application that you plan to develop has a dependency on C or C++ modules or the application needs to be signed, then you must use the Juniper Extension Toolkit (JET) virtual machine (VM) to develop the application.

The JET VM is a 64-bit Ubuntu 12.04 long-term support release. Application developers can use the JET IDE provided with the VM to develop applications. To set up the developer environment, download the JET bundle and client package from the Juniper Networks download site. Once the VM is up, install the JET tool chain, Eclipse integrated development environment (IDE), plug-ins, and other tools and libraries necessary for developing on-device or off-box applications. For more details on the installing the VM, see Setting Up the JET Virtual Machine later in this course.

Agenda: Junos Automation Architecture and Overview

- Why Automate?
- Junos mgd-based Automation
- Junos jsd-based Automation
- Automation Languages, Libraries, and Frameworks
- Automation Management Systems
- Junos Automation Tools

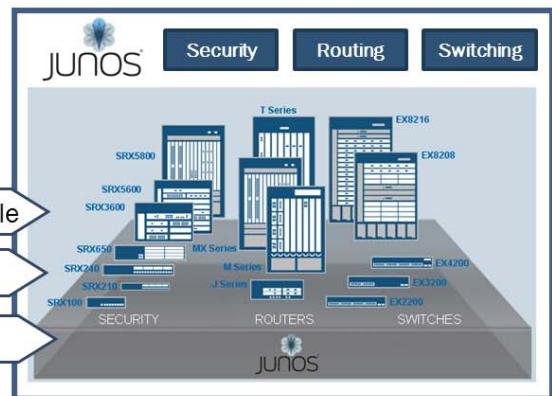
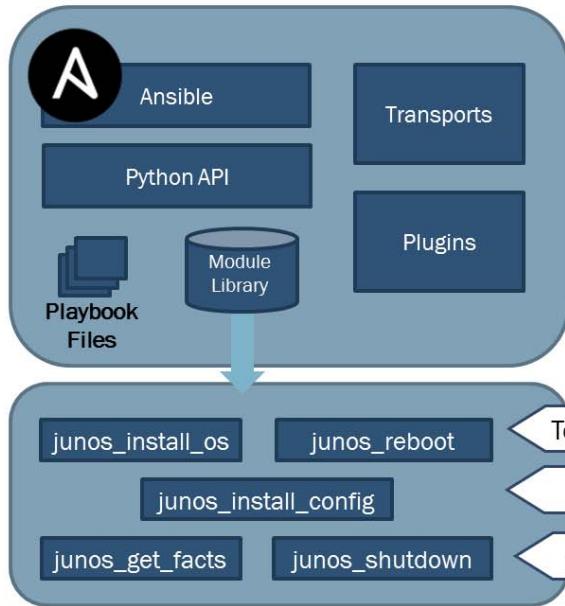
Automation Management Systems

The slide highlights the topic we discuss next.

Ansible

▪ Ansible

- Requires Python 2.7 and PyEZ
- Agentless and simple
- Minimal coding skills (YAML)
- Work flow engine



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 31

Ansible

Ansible is an uncomplicated IT automation platform that makes deploying Junos devices simple. You can avoid writing scripts or custom code to deploy and update devices, and you can automate in a language that approaches plain English (YAML). Another advantage is that it is not necessary to install agents on the devices.

Juniper has supported modules for push-scenarios only. Ansible is not designed for pulling data.

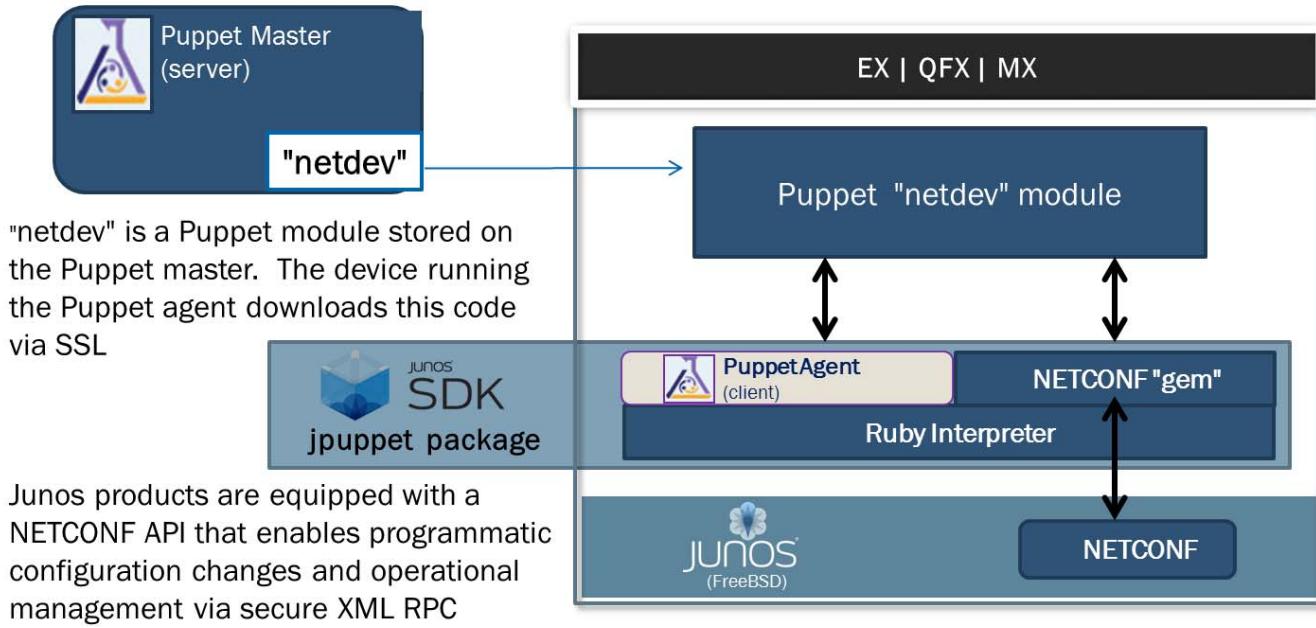
Juniper provides support for Ansible. Ansible performs specific operational and configuration tasks on devices running the Junos OS, including: installing and upgrading the Junos OS, deploying specific devices in the network, loading configuration changes, retrieving information, and resetting, rebooting, or shutting down managed devices. One of the advantages of using Ansible is that it does not need to have an Ansible-specific module loaded on the device.

Puppet



- Puppet

- Requires Ruby and Jpuppet module



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 32

Puppet

Puppet, software by Puppet Labs, is designed for configuration management. Puppet provides an efficient, scalable solution for managing the configurations on large numbers of devices. System administrators take advantage of Puppet to manage computer resources, such as physical and virtual servers. Juniper Networks provides support for Puppet to manage devices running the Junos OS.

Puppet software is deployed using a client-server arrangement where the server—called the Puppet master—manages one or more agent nodes. The client process—or Puppet agent—runs on each of the managed resources.

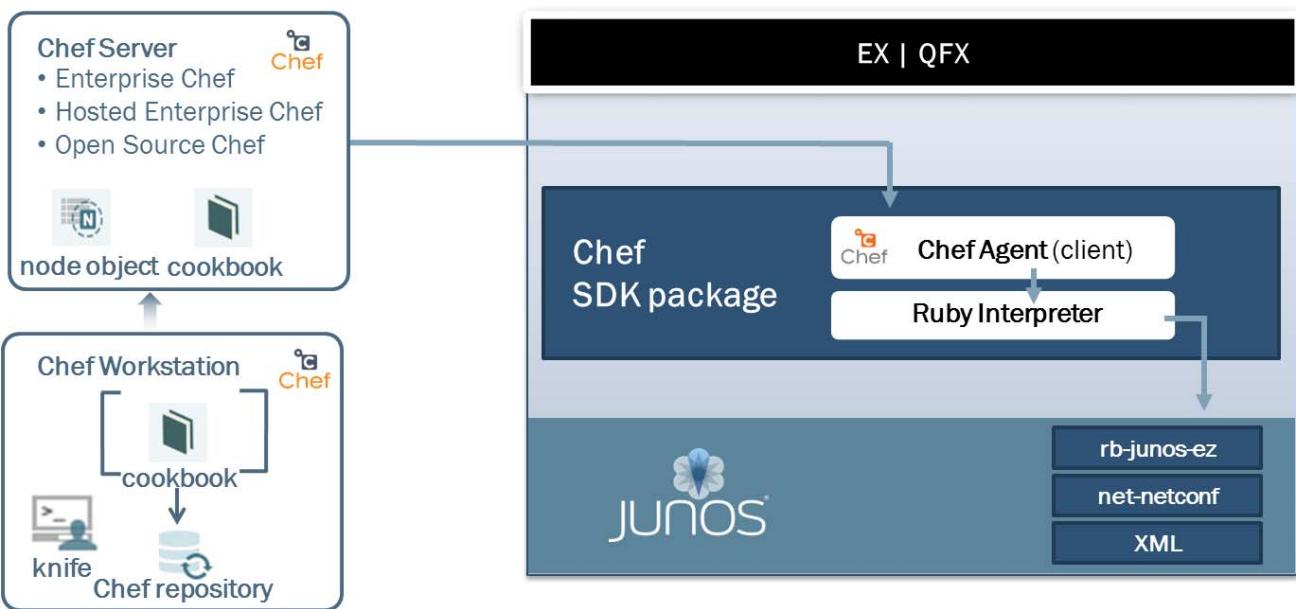
You create Puppet manifest files to describe your desired system configuration. The Puppet master compiles the manifests into catalogs, and the Puppet agent periodically retrieves the catalog and applies the necessary changes to the configuration.

Puppet is based on the Ruby Programming language. In order to use Puppet with Junos devices, you need to install the j puppet module on the Junos device. Documentation for Juniper support of Puppet can be found by searching the Web for Puppet for Junos OS.

Chef

■ Chef

- Requires Chef client and Ruby interpreter



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 33

Chef

Chef is software that automates the provisioning and management of computer networking, and storage resources. Resources can be on-site, in the cloud, or both. Chef transforms infrastructure into code, enabling you to configure, deploy, and scale in real time, while reducing the risk of human error.

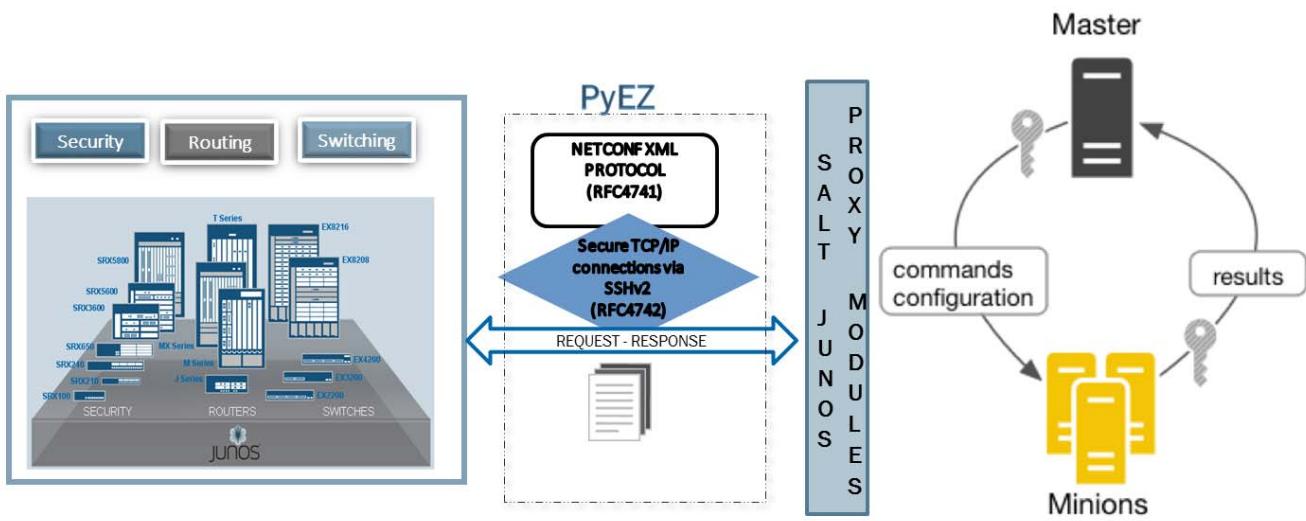
Using Chef, you write abstract definitions of your infrastructure in Ruby, and manage the definitions like you manage source code. These abstract definitions, which are called cookbooks, are applied to the nodes in your infrastructure by the Chef clients running on those nodes. When you bring a new node on-line, the only thing that the Chef client running on that node needs to know is which definitions to apply.

Chef for Junos OS enables Chef support on selected Juniper Networks devices. You can use Chef for Junos OS to automate common switching network configurations on these devices, such as physical and logical Ethernet link properties and VLANs.

To use Chef for Junos OS, the Chef client must first be installed on the Junos device. You can find installation and support information by searching the Web for Chef for Junos OS.

SaltStack

- SaltStack Features
 - Facts gathering, managing configuration, RPC execution, CLI support, install software, and file copy
- Closed loop automation



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 34

SaltStack

SaltStack is a growing configuration management platform. SaltStack is Python-based and utilizes a SaltProxy to connect to Junos devices. The SaltStack proxy supports the ability to gather facts from Junos devices, manage and commit configurations, execute RPCs and CLI commands, install software, and copy files.

One of the reasons SaltStack is popular to use with Junos is its ability to perform in a closed loop; automation is customized then triggered based on telemetry coming from the Junos device.

Agenda: Junos Automation Architecture and Overview

- Why Automate?
 - Junos mgd-based Automation
 - Junos jsd-based Automation
 - Automation Languages, Libraries, and Frameworks
 - Automation Management Systems
- Junos Automation Tools

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

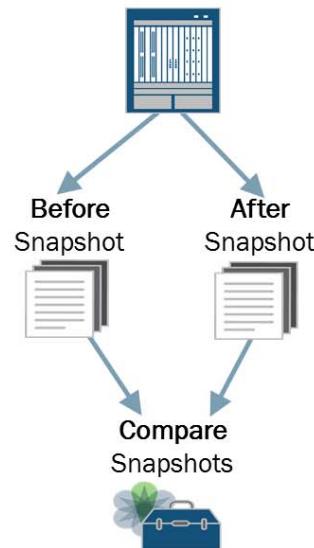
www.juniper.net | 35

Junos Automation Tools

The slide highlights the topic we discuss next.

JSNAPy

- Takes snapshots and compares Snapshots
- Used to audit environment against pre-defined criteria
- Python version of JSNAP



JSNAPy

JSNAPy, Junos Snapshot Administrator in Python captures and audits runtime environments of network devices running Junos OS. It automates network state verification by capturing and validating the status of a device. It is used to take pre-snapshots and then post-snapshots after modification, then compares them based on test cases provided. JSNAPy can also be used to audit the runtime environment of a device against pre-defined criteria.

JSNAP is similar to JSNAPy. JSNAP uses SLAX while JSNAPy uses Python. We will have an opportunity to use JSNAPy later in this course.

For more information about JSNAPy, consult <https://github.com/Juniper/jsnappy>

Junos ZTP

- ZTP is called Auto Installation on some devices
- Allow for zero-touch provisioning of Junos devices



Zero Touch Provisioning (ZTP)

Zero Touch Provisioning allows you to supply new devices in your network automatically, without manual intervention. When you connect a device to the network and boot it with a default configuration, it attempts to upgrade the Junos OS software automatically and auto-installs a configuration file from the network.

The device uses information that you configure on a Dynamic Host Control Protocol (DHCP) server to determine whether to perform these actions or not, and to locate the necessary software image and configuration files on the network. If you do not configure the DHCP server to provide this information, the switch boots with the pre-installed software and default configuration.

ZTP can be a time saver when installing multiple devices and is one of the automation tools we will examine in this course.

Summary

- In this content, we:

- Described the Junos Architecture and Automation UI
- Explained the role of gRPC, NETCONF, and REST in Junos Automation
- Identified the languages, frameworks, management suites, and tools commonly used in automating the Junos OS

We Discussed:

- The Junos architecture and automation UI;
- The role of gRPC, NETCONF, and REST in Junos Automation; and
- The languages, frameworks, management suites, and tools commonly used in automating the Junos OS.

Review Questions

1. What are the two main daemons in the Junos OS that handle automation?
2. Why was the JET API added to the Junos OS?
3. What does JSNAPy do?

Review Questions

- 1.
- 2.
- 3.

Answers to Review Questions

1.

The two main processes in the Junos OS used for automation are the mgd and the jsd.

2.

The JET API was added to the Junos OS to improve commit times and adapt to the growing modularity of the Junos OS.

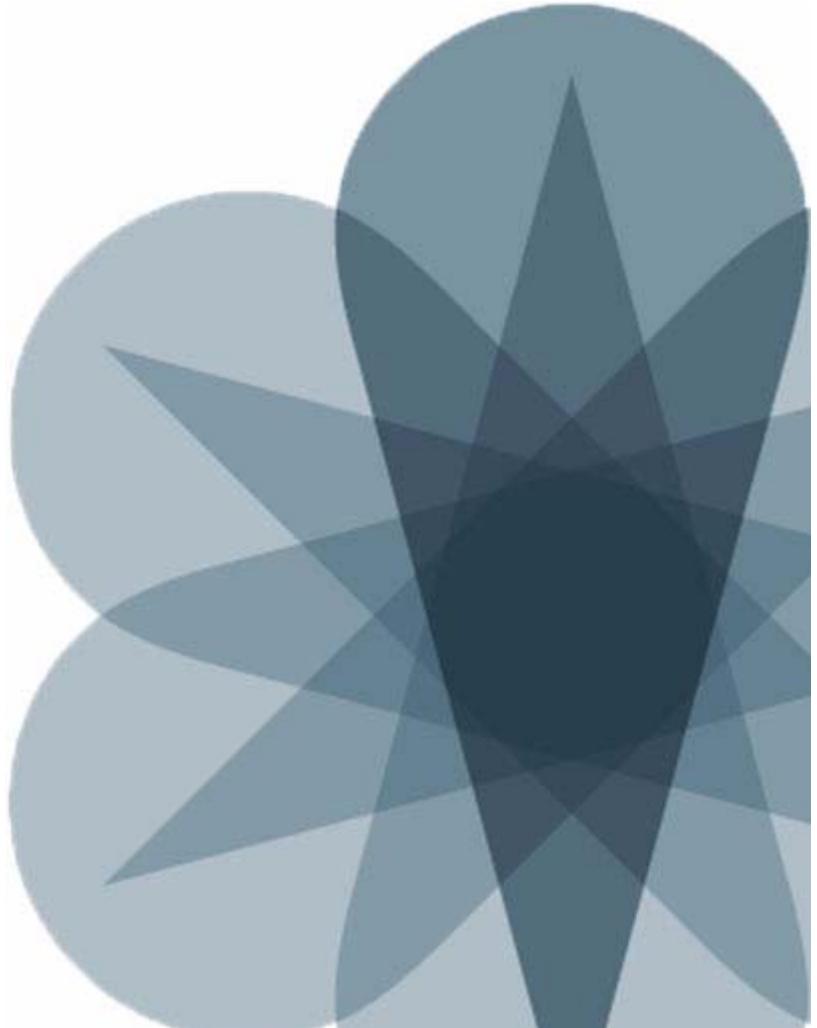
3.

JSNAPy is a Juniper tool written in Python that allows administrators to perform configuration management through the use of configuration snapshots.



Junos Platform Automation and DevOps

Chapter 3: NETCONF and the XML API



Objectives

- After successfully completing this content, you will be able to:
 - Describe the NETCONF Protocol
 - Explain the capabilities of the XML API
 - Describe the use of XSLT, SLAX and XPath in XML API development

We Will Discuss:

- The NETCONF Protocol;
- the capabilities of the XML API,
- and the use of XSLT, SLAX, and XPath in XML API.

Agenda: NETCONF and the XML API

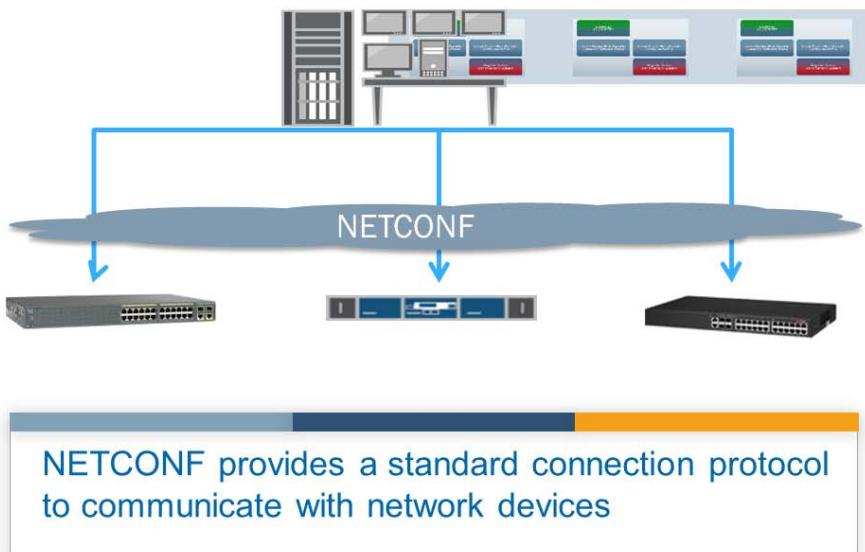
→NETCONF

- XML API
- XML API programming Languages
- XML API Tools

NETCONF

The slide lists the topics we will discuss. We discuss the highlighted topic first.

Why NETCONF?



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER NETWORKS Worldwide Education Services

www.juniper.net | 4

Why Does Juniper Use NETCONF?

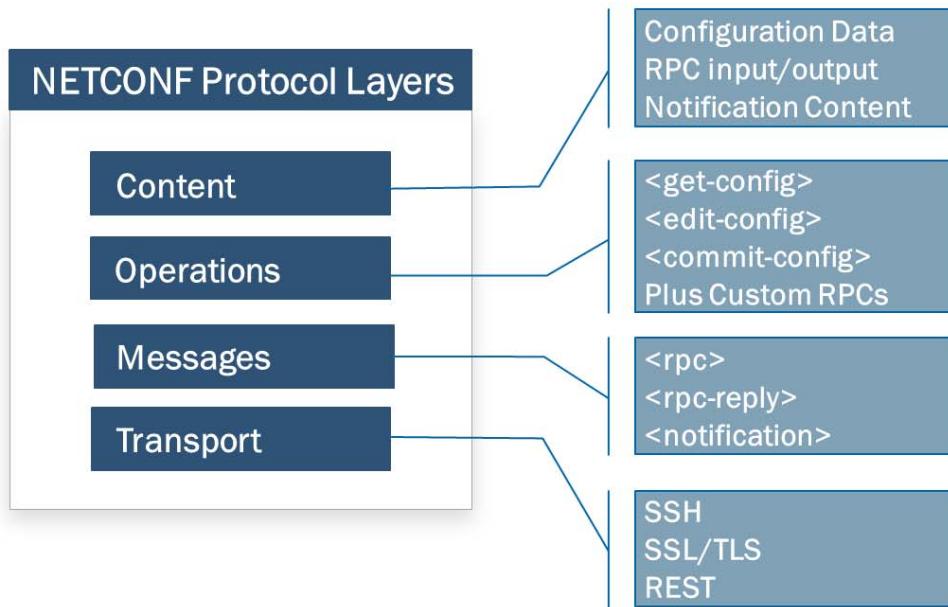
Before NETCONF, most organizations used SNMP or the CLI as a method of gathering data from devices. SNMP is widely used to gather information from devices, but as a tool to configure devices it was cumbersome so never caught on as a way to manage device configurations.

The CLI, while highly useful, is difficult to automate because it involves parsing screen output without specific data field delineators. How do you tell where one field ends and another begins? You can try to use white space, spaces and tabs, as a way to delineate fields but this can be difficult when managing devices with varying numbers of ports, routing engines, and feature sets.

Juniper developed the proprietary Junos XML Management Protocol (formally JUNOScript) as an easier, more reliable method of automating Junos OS. After using and improving JUNOScript for several years, Juniper then worked with the IETF and shared their creation with the broader networking community. NETCONF first appeared as RFC 4741, and support for SSH was added in RFC 4742. These were then updated in 2011 by RFCs 6241 and RFC 6242. The real power in NETCONF is that it is a management protocol developed specifically for networking devices with specific features. These features allow for locking a configuration, reading a configuration, modifying a configuration, deleting a configuration, and unlocking a configuration. NETCONF also has the ability to issue operational mode commands and the ability to distinguish between configuration commands, operational commands, and other messages.

Now that NETCONF is an IETF standard, it is used by vendors beyond Juniper, enabling multiple organizations to use the same management protocol to manage different vendor devices. NETCONF is a foundation for other technologies, such as YANG and OpenConfig. We will discuss these in upcoming chapters. Let's take a closer look at NETCONF.

The NETCONF Protocol Layers



The NETCONF Model

NETCONF is a protocol built specifically for networking and consists of four layers.

The Transport layer – NETCONF can create sessions using a number of different protocols as seen in the slide. Junos OS primarily uses SSH, and this is what we will focus on in this course.

In the next three pages we will discuss the Messages, Operations, and Content layers of the NETCONF protocol.

The NETCONF Model – Messages Layer

<rpc>

The <rpc> element encapsulates all remote procedure calls to the NETCONF server. This includes both operational mode and configuration RPCs.

<rpc-reply>

The <rpc-reply> element encapsulates all remote procedure call replies from the NETCONF server. This includes data returned from the NETCONF server and any OK, error, or warning messages.

<notification>

The <notification> element is a one-way message and is used to subscribe to a stream of data that the NETCONF server publishes.

NETCONF Protocol Layers

Content

Operations

Messages

Transport

NETCONF Messages

The Messages layer - The Messages layer provides a simple, transport-independent framing mechanism for encoding RPCs and notifications. The message layer consists of three XML elements: <rpc>, <rpc-reply>, and <notification>.

The <rpc> element encapsulates all remote procedure calls to the NETCONF server.

The <rpc-reply> element encapsulates all remote procedure call replies from the NETCONF server. This includes data returned from the NETCONF server and any OK, error, or warning messages.

The <notification> element is a one-way message and is used to subscribe to a stream of data that the NETCONF server publishes.

The NETCONF Messages are used in all NETCONF sessions and are used to encapsulate NETCONF operation commands and data elements.

The NETCONF Model – Operations Layer

- The NETCONF Operations layer includes the following commands:

```
<lock>
<unlock>
<get>
<get-config>
<edit-config>
<copy-config>
<commit>
<discard-changes>
<delete-config>
<validate>
<create subscription>
<close-session>
<kill-session>
```



© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 7

NETCONF Operations Layer

The Operations layer – As the name indicates, the Operations layer defines specific network operations to be performed on a device. Operation mode tags are placed within one of the message layer tags from the previous page. Following is the complete list of base protocol operations taken from the NETCONF RFC 6241.

- <lock> – The lock command is used to lock the configuration file before modifying it so that other users or RPCs cannot modify the config on a NETCONF device.
- <unlock> – The unlock command is used to unlock the configuration after modifying it so that other users may access it.
- <get> – The get command retrieves data from the running configuration database or device statistics. The get operation is used to retrieve the active configuration and device state information.
- <get-config> – The get-config command is used when retrieving the configuration off of a NETCONF device.
- <edit-config> – The edit-config command is used when changing the configuration on a NETCONF device.
- <copy-config> – The copy-config command is used to make a copy of the configuration on a NETCONF devices.
- <commit> – The commit command is used to commit the candidate configuration to the running configuration on a NETCONF device

Continued on the next page.

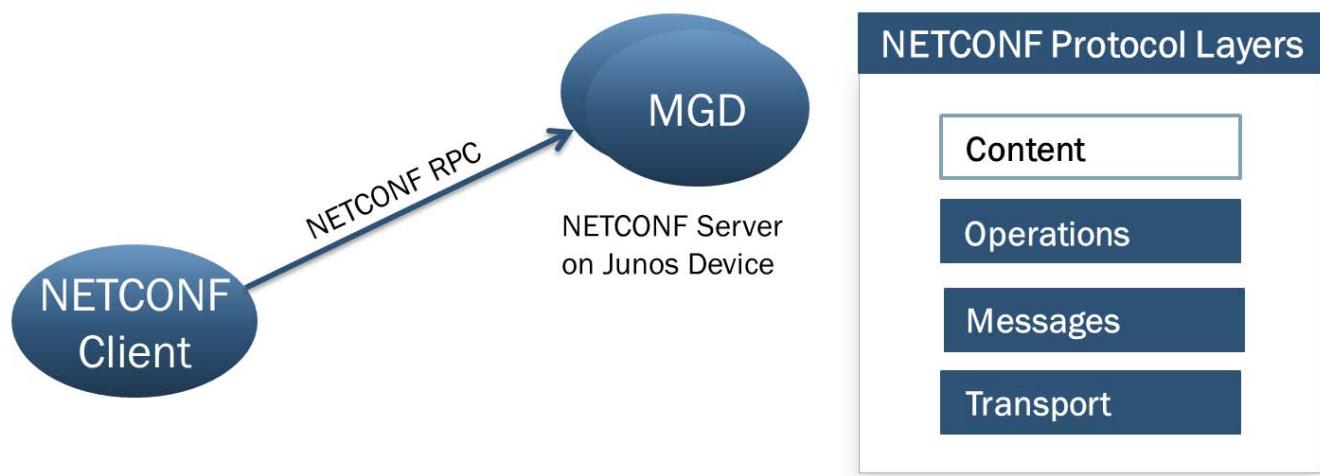
NETCONF Operations Layer (contd.)

- <delete-config> – This command is used to delete a section or all of a NETCONF device.
- <create-subscription> – The create subscription command is used to create a NETCONF subscription.
- <close-session> – This command is issued to close a NETCONF session.
- <kill-session> – This command is used to close a different NETCONF session than the one you are currently using.

Notice how the operations are specific to networking and network configurations. The get operation is used to execute operational mode commands. The get-config, edit-config, copy-config, delete-config, lock, and unlock operations are used to manage device configuration files. The <close-session> and <kill-session> operational commands are used to manage the NETCONF session. There are additional attributes within each of these operations that allow users to specify whether they are merging, replacing, or removing lines from a configuration. You can explore these further by consulting RFC 6241.

The NETCONF Model – Content Layer

- Contains RPCs
- Contains configuration data



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 9

The Content Layers

The Content layer – The Content layer is the layer where specific RPCs and configuration data are encapsulated and sent to a NETCONF server. The XML API on the NETCONF server accepts the incoming RPCs. Once the MGD receives an RPC, it forwards this to the appropriate Junos OS processes. In this chapter, we will be spending the majority of our focus on the XML API.

One of the nice features of NETCONF is that, other than the Transport layer that is encrypted using SSH, the other layers are all in clear text, making the protocol fairly easy to monitor.

Configure Junos OS for NETCONF

- Configure Junos to accept NETCONF SSH session over port #830

```
[edit system services]
lab@vMX-1# set netconf ssh
```

```
[edit system services]
lab@vMX-1# show
ssh;
netconf {
    ssh;
}
```

Configure the Junos OS for NETCONF

A good way to start understanding NETCONF is to open up a NETCONF session on your device running Junos OS and interact with it. All of the messages, operations, and content are text-based so that you can input them from the keyboard.

You can connect to NETCONF using SSH and over the standard SSH port of 22. To enable NETCONF over the standard SSH port, simply issue the **set system services ssh** command to enable SSH.

However, many NETCONF client applications use the designated NETCONF port 830, as specified in RFC 4742. To enable the Junos OS to accept NETCONF sessions on port 830, you need to issue the **set system services netconf ssh** command. You can also specify your own unique port by issuing the **# set system services netconf ssh port-number** command.

After starting a NETCONF session on the device (or remotely) the user cannot get back to the CLI prompt without terminating the session; the NETONF session can be closed by issuing the empty `<close-session/>` tag and closing brackets, as follows:

```
<rpc>
    <close-session/>
</rpc>
]]>]]>
```

Start a NETCONF Session

▪ Start a NETCONF Session

```
--- JUNOS 16.2R1.6 Kernel 64-bit JNPR-10.3-20161102.338446_build
lab@vMX-1> netconf
<!-- No zombies were killed during the creation of t Permissions used
interface -->
<!-- user lab, class j-super-user -->
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
    <capabilities>
        . . . Trimmed . . .
        <capability>urn:ietf:params:netconf:base:1.0</capability>
        <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
        <capability>http://xml.juniper.net/dmi/system/1.0</capability>
    </capabilities>
    <session-id>42708</session-id> Session ID
</hello>
]]>]]> NETCONF message termination signal
```

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 11

Starting a NETCONF Session

You can start up a NETCONF session in the Junos OS CLI by simply typing **netconf** while in operational mode. You can also connect remotely using SSH and then start a NETCONF session in the same manner. As seen in the slide, once you have connected over ssh you start the NETCONF session by typing **netconf**. You can combine the two steps by entering **ssh -s lab@vMX-1 netconf** on the remote device. The **-s** option tells the Junos OS to start up the NETCONF subsystem once logged in. You can also add the **-p830** option so that your NETCONF session uses the designated NETCONF port of 830.

The slide shows the response you receive once you enter into a NETCONF session. There are a few things to note when starting a NETCONF session. First, note that the user and corresponding permissions class are listed. Make sure the user account has sufficient permissions for performing any required configuration and operational tasks. In general, the same permissions are required when performing tasks through the CLI or using an RPC.

Note that the NETCONF server displays its capabilities. The Junos OS supports all NETCONF features. Also noted in the slide is the session ID; this can be useful if there are multiple, concurrent sessions.

Finally, note the **]]>]]>** string at the end. Every NETCONF message concludes with **]]>]]>**. For more specific information about the steps involved in setting up a NETCONF session, see RFC 6241 and the Juniper NETCONF XML Management Protocol Developer Guide.

Agenda: NETCONF and the XML API

- NETCONF
- XML API
- XML API Programming Languages
- XML API Tools

XML API

The slide highlights the topic we discuss next.

The Junos OS XML API

■ XML API Uses

- Issue operational mode commands
- Change the device configuration

■ XML based encoding

- All commands and configuration encapsulated in XML element tags

```
<rpc>
    <get-system-uptime-information>
    </get-system-uptime-information>
</rpc>
]]>]]>
```

The Junos OS XML API

Now that you have a NETCONF session open, what can you do with it? The same thing you can do with the CLI: you can issue operational mode commands and modify the device configuration. The Junos XML API Management Protocol was created by Juniper to do just that; it is an XML-based protocol that client applications use to request and change configuration information on devices running the Junos OS. The Junos XML Management Protocol uses XML-based data encoding for the configuration data and remote procedure calls.

XML-based encoding

NETCONF RPCs can be issued by a client application or by a user who has opened a NETCONF session with a device. As seen in the slide, all NETCONF RPC requests are enclosed in an `<rpc>` element tag then conclude with the `]]>]]>` signal. In XML all white space is ignored. The example in the slide is complete; you can type it in as shown.

Display XML RPC Pipe Command

- Example:

```
lab@vMX-1> show chassis alarms | display xml rpc
<rpc-reply>
<rpcxmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
  <rpc>
    <get-alarm-information>
    </get-alarm-information>
  </rpc>
  <cli>
    <banner></banner>
  </cli>
</rpc-reply>
```

The Display XML RPC Pipe Command

One method for mapping a given Junos OS operational mode command to the equivalent RPC is that the Junos OS uses to help us determine the appropriate XML API RPC to issue for a given command is the `display xml rpc pipe` command. Most Junos OS commands have an equivalent XML API command. Both the CLI command and XML RPC are linked together in the Junos schema that is rebuilt each time that a Junos device boots. The `display xml rpc pipe` command is, in essence, a reverse lookup that says, “give me the XML RPC equivalent of this Junos command.”

Note in the slide that the response is enclosed in `<rpc-reply>` tags. The NETCONF protocol stipulates that all replies are labeled this way. Also, note the XML namespace (`xmlns`) is `junos/16.2R1`. The information provided is specific to a version of Junos. As such, don’t expect all of the features to be supported in Junos 16.2 to also be supported in earlier versions of Junos.

The actual command that we are looking for is enclosed in the `<rpc>` tags. The information in the `<cli>` tags can be ignored. When issuing the `| display XML RPC` form within configuration mode the `<cli>` tags will show the current Junos hierarchy. To execute the RPC in a NETCONF session, include the `]]>]]>` termination signal after the RPC, making the complete XML RPC appear like this:

```
<rpc>
  <get-alarm-information>
  </get-alarm-information>
</rpc>
]]> ]]>
```

Display XML Pipe Command

- CLI:

```
lab@vMX-1> show chassis alarms
```

No alarms currently active

- | display xml:

```
lab@vMX-1> show chassis alarms | display xml
```

```
<rpc-reply
  xmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
  <alarm-information
    xmlns="http://xml.juniper.net/junos/17.1R1/junos-alarm">
    <alarm-summary>
      <no-active-alarms/>
    </alarm-summary>
  </alarm-information>
  . . . Trimmed . . .
```

The Display XML Pipe Command

The `display xml` pipe command is a useful tool that enables users to see what type of XML output to expect from the XML API. Note that CLI output is going to be different than XML output. The CLI has a rendering process that changes the XML output into a more readable format.

Display JSON Command

```
lab@vMX-1> show chassis alarms | display json
{
    "alarm-information" : [
        {
            "attributes" : {"xmlns" :
"http://xml.juniper.net/junos/17.1R1/junos-alarm"},
            "alarm-summary" : [
                {
                    "no-active-alarms" : [
                        {
                            "data" : [null]
                        }
                    ]
                }
            ] }] }] }
```

The Display JSON Pipe Command

The Junos operating system supports XML natively for the operation and configuration of devices running Junos OS, and the Junos OS command-line interface (CLI) and the Junos OS infrastructure communicate using XML. When you issue an operational command or display, the configuration in the CLI converts the output from XML into a readable text format for display.

Starting in the Junos OS Release 14.2, devices running the Junos OS also support a JavaScript Object Notation (JSON) representation of the operational command output and the Junos OS configuration hierarchy. To display the command output or configuration in JSON instead of in the default formatted ASCII text on the Junos OS CLI, append the `| display json` option to the command.

This is useful if you use or support a client application that handles JSON better than XML.

Find the Right RPC (1 of 4)

- Use the `| display xml rpc` command

```
lab@vMX-1> show interfaces ge-0/0/0 terse | display xml rpc
<rpc-reply
  xmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
    <rpc>
      <get-interface-information>
        <terse/>
        <interface-name>ge-0/0/0</interface-name>
      </get-interface-information>
    </rpc>
    <cli>
      <banner></banner>
    </cli>
  </rpc-reply>
```

Find the Correct RPC, Part 1

Let's walk through an example of how to find the right RPC for a project you may be working on. Suppose you want to figure out what the equivalent RPC is for the `show interfaces terse ge-0/0/0` command. One method is to issue the `show interfaces terse ge-0/0/0` command with the `| display xml rpc` command option at the end. We receive the `<rpc-reply>` shown in the slide.

Find the Right RPC (2 of 4)

- Use the JUNOS schema documents
- The schema (.XSD) document can be downloaded or generated from the Junos device

```

<xsd:element name="detail" minOccurs="0">
<xsd:element name="terse" minOccurs="0">
<xsd:element name="brief" minOccurs="0">
<xsd:element name="descriptions" minOccurs="0">
</xsd:choice>
<xsd:element name="snmp-index" minOccurs="0" type="xsd:string">
<!-- </snmp-index> -->
<xsd:element name="switch-port" minOccurs="0" type="xsd:string">
<!-- </switch-port> -->
<xsd:element name="interface-name" minOccurs="0" type="xsd:string">
<!-- </interface-name> -->

```

Annotations:

- A callout bubble points to the 'terse' element definition with the text "terse is a valid option".
- A callout bubble points to the 'interface-name' element definition with the text "Interface-name is also a valid option".

Find the Correct RPC, Part 2

The `| display xml rpc` command helps in figuring out the correct RPC, but it is not an authoritative source. The authoritative sources are the XML schemas themselves. The slide shows a screen capture from the operational data schema file. In it you can see that both the `terse` option and the `interface-name` option are valid. The schema also shows several other possible options. The schema can be useful if you have difficulties figuring out the correct RPC from the CLI or if you want to see what other options are available.

You will see how to access the Junos OS schemas shortly.

Find the Right RPC (3 of 4)

■ The XML API Explorer

The screenshot shows the Juniper XML API Explorer interface. On the left, there's a sidebar for selecting a software release (Junos 17.1 is selected) and operational tags for Junos 17.1. The main area displays the XML schema for the 'get-interface-information' RPC. It includes sections for Usage, Description, and Example. The Usage section shows the XML structure for the RPC, including its parameters and nested elements like routing-instance, satellite-device, aggregation-device, and zone. The Description section provides a brief explanation of what the RPC does, mentioning it shows interface information and routing status.

```

Usage
<usage>
<rpc>
<get-interface-information>
<routing-instance>routing-instance</routing-instance>
<satellite-device>satellite-device</satellite-device>
<aggregation-device></aggregation-device>
<zone></zone>
<extensive></extensive>
<statistics></statistics>
<media></media>
<detail></detail>
<terse></terse>
<brief></brief>
<descriptions></descriptions>
<snmp-index>snmp-index</snmp-index>
<switch-port>switch-port</switch-port>
<interface-name>interface-name</interface-name>
</get-interface-information>
</rpc>
</usage>

Description
Show interface information
<routing>—Show routing status
<get-mc-ae-interface-information>—Show MC-AE configured interface information

```

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 19

Find The Correct RPC, Part 3

The XML API Explorer is another resource you can consult to find the right RPC. The XML API explorer is easier to read than the Junos Schema files. It is also useful because it contains a description of each of the RPC elements.

You can find the XML API Explorer for operational tags at: <https://apps.juniper.net/xmlapi/operTags.jsp>

The XML API Explorer for configuration tags is located at: <https://apps.juniper.net/xmlapi/confTags.jsp>

Find the Right RPC (4 of 4)

- Junos XML API Operational Developer Reference



Junos OS

Junos XML API Ope

<get-interface-information>

Usage

```
<rpc>
<get-interface-information>
<routing-instance>routing-instance</routing-instance>
<extensive/>
<statistics/>
<media/>
<detail/>
<terse/>
<brief/>
<descriptions/>
<snmp-index>snmp-index</snmp-index>
<switch-port>switch-port</switch-port>
<interface-name>interface-name</interface-name>
</get-interface-information>
</rpc>
```

Description Show interface information

Contents <routing>—Show routing status

- all - All instances

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 20

Find the Correct RPC, Part 4

Finally if you prefer to work with print or PDF documents you can download the PDFs.

The Junos XML API Operational Developer Reference is located at: http://www.juniper.net/techpubs/en_US/junos/information-products/topic-collections/junos-xml-ref-oper/junos-xml-ref-oper.pdf and the Junos XML Management Protocol Developer Guide at: https://www.juniper.net/techpubs/en_US/junos/information-products/pathway-pages/junos-xml-management-protocol/junos-xml-management-protocol.pdf

The XML Schemas

- Consists of two files
 - XML Schema for Configurator Data
 - XML Schema for Operational Data
- Downloadable from Junos XML API download site
- Unpacked files become:
 - Config-16.2.xsd
 - operational-command-16.2.xsd
- Config schema retrievable from Junos device using NETCONF

The XML Schemas

The Junos schema documents, namely the XML Schema for configuration data and the XML Schema for Operational Data can be downloaded from the Junos XML API download site at <http://www.juniper.net/support/downloads/?p=junosxml#sw>

The first file contains the XML schema for the Junos configuration hierarchy, while the second contains the schema for the Junos operational mode commands. Once you unpack the files, they are called the config-16.2.xsd and operational-command-16.2.xsd files, respectively, for Junos OS Release 16.2. Ensure that you get the schema for the version of Junos OS that you are using.

You can also get the configuration schema from the device running Junos OS. In a NETCONF session with a device running Junos OS, to request an XML Schema-language representation of the entire configuration hierarchy, a client application emits the Junos XML <get-xnm-information> tag element and its <type> and <namespace> child tag elements with the indicated values in an <rpc> tag element:

```
<rpc>
  <get-xnm-information>
    <type>xml-schema</type>
    <namespace>junos-configuration</namespace>
  </get-xnm-information>
</rpc>
```

Note: this command may take several minutes to run.

Requesting Configuration Data using XML

```
<rpc>
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="subtree">
      <configuration>
        <system>
          <login/>
        </system>
      </configuration>
    </filter>
  </get-config>
</rpc>
]]> ]>
```

All configuration requests include a `<get-config>` element

The source is either running or candidate

This is the section of configuration you want to retrieve

Requesting Configuration Data Using the XML API

So far we have looked at issuing commands using the XML RPC; we can also retrieve and modify the configuration. The example in the slide shows an XML RPC that requests the configuration in the [edit system login] hierarchy of the configuration.

XML API Reply

```

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
  <data>
    <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm"
      junos:commit-seconds="1487612274" junos:commit-
      localtime="2017-02-20 17:37:54 UTC" junos:commit-user="lab">
      <system>
        <login>
          <user>
            <name>lab</name>
            <uid>2000</uid>
            <class>super-user</class>
            <authentication>
              <encrypted-
                password>$5$9qGmqqus$4ffC5hW6gmvg4VI6vFQwfff9fSJ3/g92AyS0b03w
                8t5</encrypted-password> ...
            </authentication>
          </user>
        </login>
      </system>
    </configuration>
  </data>
</rpc-reply>

```

Return data is enclosed in `<data>` tags

Junos version

Commit date and time

User who committed configuration

Requested data

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 23

XML API Reply

The slide shows the data returned from the configuration request from the previous page. Notice that the response places the configuration data within `<data>` and `<configuration>` elements. We will see more of this later on. The response also includes the Junos OS version, the date and time when the configuration was committed, or “of the commit,” and the user who committed the configuration.

We can use a similar process to save data and to commit data. We will go into more depth using NETCONF committing the configuration when discussing the YANG API. To get more information about modifying configurations using NETCONF, consult the Juniper NETCONF XML Management Protocol Developer Guide.

Agenda: NETCONF and the XML API

- NETCONF
- XML API
- XML API Programming Languages
- XML API Tools

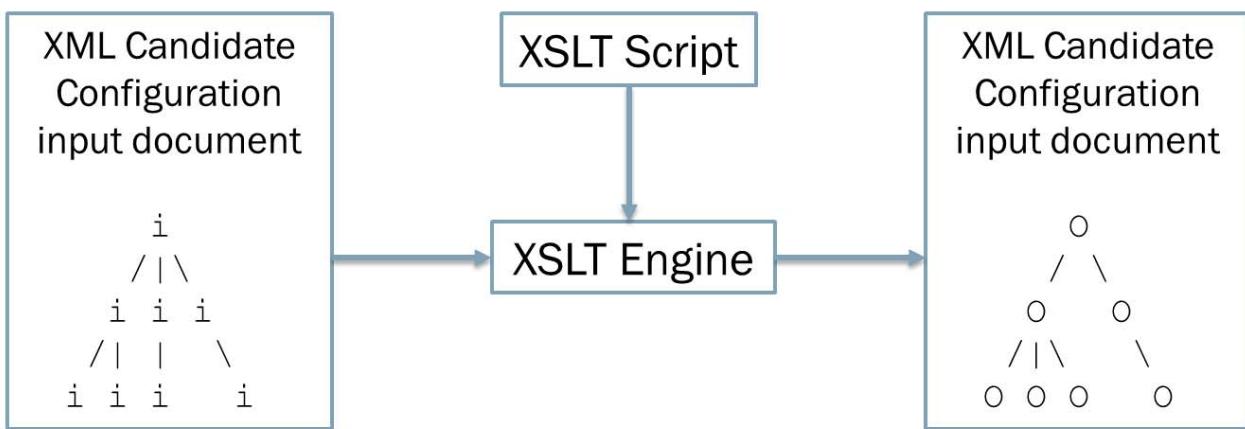
XML API Programming Languages

The slide highlights the topic we discuss next.

Using XSLT to Automate the XML API

▪ XSLT Abilities

- Designed for XML to XML transformations
- Find specific nodes in the configuration hierarch
- Perform if–then type logic
- Perform loops



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 25

Using XSLT To Automate the Junos XML API

So far, you have seen that the XML API can be used to execute RPCs and to retrieve and commit configurations. However, we have not seen how to test conditions, such as whether ‘the user Bob’ exists in the configuration. Also, we have not seen how to iterate through a list of interfaces and verify that they belong to the appropriate VLANs. To do so would require the ability to use `if` statements and perform loops. Neither NETCONF or the XML API have such capabilities.

Extensible Stylesheet Language Transformations (XSLT) to the rescue. XSLT is a natural match for the Junos OS, with its native XML capabilities. XSLT performs XML-to-XML transformations, turning one XML hierarchy into another. It offers a great degree of freedom and power in the way in which it transforms the input XML, allowing everything from making minor changes to the existing hierarchy to building a completely new document hierarchy.

XSLT is a language for transforming one XML document into another XML document. The basic model used for transformation is that an XSLT engine (or processor) reads a script (or style sheet) and an XML document. The XSLT engine uses the instructions in the script to process the XML document by traversing the document’s hierarchy. The script indicates what portion of the tree should be traversed, how it should be inspected, and what XML should be generated at each point.

Because XSLT was created to allow generic XML-to-XML transformations, it is a natural choice for both inspecting configuration syntax, which the Junos OS can easily express in XML, and for generating errors and warnings. XSLT includes powerful mechanisms for finding configuration statements that match specific criteria. XSLT can then generate the appropriate XML result tree from these configuration statements to instruct the Junos OS user-interface (UI) components to perform the desired behavior.

For more information about XSLT, consult the Juniper Automation Scripting Feature Guide.

XSLT Conditional Logic

```
<xsl:choose>
  <xsl:when test="system/host-name">
    <change>
      <system>
        <host-name>M320</host-name>
      </system>
    </change>
  </xsl:when>
  <xsl:otherwise>
    <xnm:error>
      <message>
        Missing [edit system host-name] M320.
      </message>
    </xnm:error>
  </xsl:otherwise>
</xsl:choose>
```

When the host-name statement is included at the [edit system] hierarchy level, change the hostname to M320. Otherwise, issue the warning message: Missing [edit system host-name] M320.

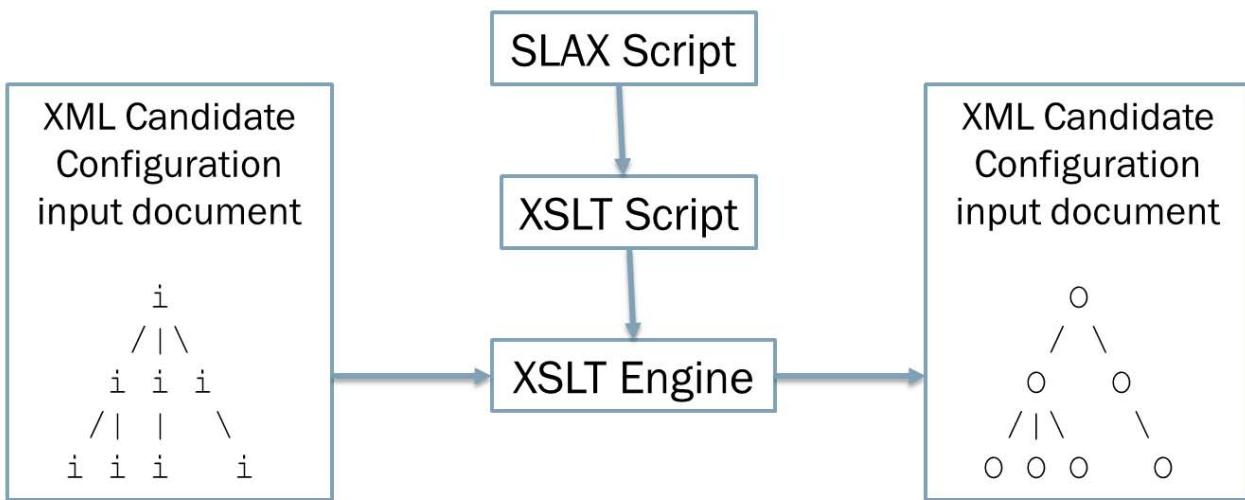
XSLT Conditional Logic

To give you a feel for XSLT, the slide above shows an example of an XSLT conditional logic statement. In the example, when the host-name statement is included at the [edit system] hierarchy level, change the host name to M320. Otherwise, issue the warning message: Missing [edit system host-name] M320.

XSLT has the advantage of being an XML programming language for manipulating XML. However, XSLT is not a preferred format for many programmers and as such will not be covered in any more depth in this course. For more information about XSLT syntax and capabilities, see the Juniper Automation Scripting Feature Guide.

The SLAX Programming Language

- Open source language
- Directly translates to XSLT
- Easier to learn and program than XSLT



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 27

The SLAX Programming Language

SLAX is a programming language developed by Juniper Engineers to make automating The Junos OS easier. SLAX has been open sourced and can be found at: <http://www.libslax.org/the-slax-language>.

The big draw for SLAX is that it is easier to program than XSLT. SLAX has the same capabilities as XSLT and you can translate back and forth between the two languages. Actually, as shown in the slide, SLAX code is rendered into XSLT before it is processed by the XSLT engine.

The benefits of SLAX are particularly strong for programmers who are not already accustomed to XSLT, because SLAX enables them to concentrate on the programming task rather than concentrating on learning a new syntax. For example, SLAX enables you to:

- Use if, else if, and else statements instead of <xsl:choose> and <xsl:if> elements
- Put test expressions in parentheses ()
- Use the double equal sign (==) to test equality instead of the single equal sign (=)
- Use curly braces to show containment instead of closing tags
- Perform concatenation using the underscore (_) operator, as in PERL, version 6
- Write text strings using simple quotation marks (" ") instead of the <xsl:text> element
- Simplify namespace declarations
- Reduce the clutter in your scripts
- Write more readable

Sample SLAX Script

```
if (../admin-status == "up" && ../oper-status == "up") {  
} else if (../admin-status == "down") {  
    expr "offline";  
} else if (../oper-status == "down" && ../admin-status == "down") {  
    expr "p-offline";  
} else if (../oper-status == "down" && ../oper-status == "down") {  
    expr "p-down";  
} else if (../oper-status == "down") {  
    expr "down";  
} else {  
    expr ../oper-status _ "/" _ ../admin-status;  
}
```

Sample SLAX Script

This slide is a portion of a SLAX script that checks the admin status and the operational status of an interface. In XSLT we used when statements, but with SLAX we can use the more familiar `if`, `else if`, and `else` statements. Also, notice that there are no XML tags, but instead the syntax is much closer to C or Java syntax. SLAX has been eagerly accepted and widely used by the Junos automation community.

The SLAX language, though similar to C and Java, still has its limitations because of how closely it is tied to XSLT. Junos OS Release 16.1 introduced a native Python compiler on devices running Junos OS. This course is going to focus primarily on using Python as a means of automating the Junos OS. For information about SLAX syntax and capabilities, consult the Junos Automation Scripting Feature Guide.

The XML Path Language (XPath)

■ XPath Nodes

- Element
- Text
- Attribute

```
<system>
    <host-name>my-router</host-name>
    <accounting inactive="inactive">
</system>
```

■ XPath Axes

- Ancestor or Parent
- Child
- Sibling

XPath Nodes

XSLT, SLAX, Python, and other languages that interact with the Junos OS use the XML Path Language (XPath). XPath is standard way to specify and locate XML elements in an XML document's hierarchy. XPath's powerful expression syntax enables you to define complex criteria for selecting portions of the XML input document.

XPath views every piece of the document hierarchy as a node. The important types of nodes are element nodes, text nodes, and attribute nodes. Consider the following XML tags:

```
<system>
    <host-name>my-router</host-name>
    <accounting inactive="inactive">
</system>
```

These XML tag elements show examples of the following types of XPath nodes:

Element node – <host-name>my-router</host-name>

Text node – my-router

Attribute node – inactive="inactive"

Continued on the next page.

XPath Axes

Nodes are viewed as being arranged in certain axes. The ancestor axis points from a node up through its series of parent nodes. The child axis points through the list of an element node's direct child nodes. The attribute axis points through the list of an element node's set of attributes. The following sibling axis points through the nodes that follow a node, but are under the same parent. The descendant axis contains all of the descendants of a node. There are numerous other axes that are not listed here.

Each XPath expression is evaluated from a particular node, which is referred to as the context node (or simply context). The context node is the node at which the XSLT processor is currently looking. XSLT changes the context as the document's hierarchy is traversed, and XPath expressions are evaluated from that particular context node.

An XPath expression contains two types of syntax, a path syntax and a predicate syntax. Path syntax specifies which nodes to inspect in terms of their path locations on one of the axes in the document's hierarchy, from the current context node. Several examples of path syntax follow:

`accounting-options` – Selects an element node named `accounting-options` that is a child of the current context.

`server/name` – Selects an element node named `name` that is a child of an element named `server` that is a child of the current context.

`/configuration/system/domain-name` – Selects an element node named `domain-name` that is the child of an element named `system` that is the child of the root element of the document (`configuration`).

`parent::system/host-name` – Selects an element node named `host-name` that is the child of an element named `system` that is the parent of the current context node. The `parent::` axis can be abbreviated as two periods (...).

XPath Predicates and Operators

▪ Predicates

```
server[name = '10.1.1.1']
*[@inactive]
route[starts-with(next-hop, '10.10.')] 
```

▪ Operators

Logical Operators

AND, OR

Comparison Operators

=, !=, <, and >

Numerical Operators

+, -, and *.

XPath Predicates

The XPath predicate syntax allows you to perform tests at each node selected by the path syntax. Only nodes that pass the test are included in the result set. A predicate appears inside square brackets ([]) after a path node. Following are several examples of predicate syntax.

`server[name = '10.1.1.1']` — Selects an element-named server that is a child of the current context and has a child element named name whose value is 10.1.1.1.

`*[@inactive]` — Selects any node (* matches any node) that is a child of the current context and that has an attribute (@ selects nodes from the attribute axis) named inactive.

`route[starts-with(next-hop, '10.10.')]` — Selects an element named route that is a child of the current context and that has a child element named next-hop whose value starts with the string 10.10.

The `starts-with` function is one of many functions that are built into XPath. XPath also supports relational tests, equality tests, and many more features not listed here.

Continued on the next page.

XPath Operators

XPath supports standard logical operators, such as AND and OR; comparison operators, such as =, !=, <, and >; and numerical operators, such as +, -, and *.

In XSLT, you must always represent the less-than (<) operator as <; and the less-than-or-equal-to (<=) operator as <=; because XSLT scripts are XML documents, and less-than signs are represented this way in XML.

For more information about XPath functions and operators, consult a comprehensive XPath reference guide. XPath is fully described in the W3C specification at <http://w3c.org/TR/xpath>.

We will encounter more XPath examples throughout the course.

Adding Selection Criteria (1 of 2)

- **Test your knowledge:** How do you get the user id for user ‘lab’?
 - Using system/login/user/uid is ambiguous

```
user@R1> show configuration system | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
  <configuration ... Trimmed ...>
    <version>17.1R1.8</version>
    <system>
      <login>
        <user>
          <name>lab</name>
          <uid>2000</uid>
          <class>super-user</class>
          <authentication>
            <encrypted-password>$1$8...Trimmed...</encrypted-password>
          </authentication>
        </user>
        <user>
          <name>user</name>
          <uid>2001</uid>[...]
```

Adding Selection Criteria, Part 1

Sometimes there is more than one element with the same name. For example, in the XML on the slide, there are two users. Let us imagine that you want to access the ‘lab’ user’s user id (or “UID”) value. If you simply specify “system slash login slash user slash UID”, you cannot be assured of getting the correct user’s information, because there is more than one user in the configuration.

Adding Selection Criteria (2 of 2)

- **Answer:** Use square brackets after the `user` element to specify that you want the user whose name is “lab”

`System/login/user [name=='lab']/uid`

```
user@R1> show configuration system | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
  <configuration ... Trimmed ...>
    <version>17.1R1.8</version>
    <system>
      <login>
        <user>
          <name>lab</name>
          <uid>2000</uid>
          <class>super-user</class>
          <authentication>
            <encrypted-password>$1$8...Trimmed...</encrypted-password>
          </authentication>
        </user>
        <user>
          <name>user</name>
          <uid>2001</uid>[...]
```

Adding Selection Criteria, Part 2

In these cases, you can use square brackets to add selection criteria. By specifying that you want the user whose name is lab in square brackets after the `<user>` element, you can select the correct user’s UID. This syntax tells Junos to look for the user element that contains an element called `<name>` with the value of lab. The software will then get that user record’s UID.

Elements With Attributes

- Elements

```
<wrapper>
    <element>
        <name>Bob</name>
    </element>
</wrapper>
```

Xpath: element/name

- Elements with and attribute

```
<wrapper>
    <element name="Bob" />
</wrapper>
```

Xpath: element/@name

Elements with Attributes

One final item to note is that you need to use an @ sign if you want to get the contents of an attribute. Attributes are the variables or name-value pairs that appear within an XML tag. The examples on the slide are rooted at the `<wrapper>` element. The first example shows a name as a hierarchical element. The second example shows a name as an attribute, and how you would use an XPath expression to access that attribute.

Common Attributes

- `junos:changed` is attached to nodes that have changed
- `junos:group` is attached to nodes that have been inherited from a configuration group

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.2R2/junos">
    <configuration junos:commit-seconds="1433870237" junos:commit-
localtime="2015-06-09 17:17:17 UTC" junos:commit-user="lab">
        <version>14.2R2.6</version>
        <system junos:changed="junos:changed">
            <host-name junos:group="re0">test</host-name>
        [...]
```

XPath to retrieve `junos:changed`: `system/@junos:changed`

XPath to retrieve `junos:group`: `system/host-name/@junos:group`

Common Attributes

For the purposes of this course, you are most likely to use attributes in commit scripts, where the `junos:changed` attribute is attached to nodes that have changed and the `junos:group` attribute is attached to nodes that have been inherited from a configuration group.

The bottom of the slide shows the XPath code needed to select the `junos:changed` and `junos:group` attributes shown in slide.

Agenda: NETCONF and the XML API

- NETCONF
 - XML API
 - XML API programming Languages
- XML API Tools

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 37

XML API Tools

The slide highlights the topic we discuss next.

Other Language Options

- Language Possibilities



The graphic displays ten programming languages arranged in two rows. The top row contains Python, C++, C, Ruby, C#, and Perl. The bottom row contains Java, Go, PHP, and Objective C.

- Language Libraries make it easier

Many Other Language Possibilities

There are many more languages than XSLT and SLAX, but were the first languages specifically designed to work with the Junos XML API. Any language that can be programmed to make a NETCONF connection, use the NETCONF protocol and send well-formed XML over the NETCONF connection can be used to automate devices running the Junos OS using the NETCONF API.

Language Libraries make it easier

Juniper has created libraries of code for Python, Perl, and Java. These libraries make your interaction with NETCONF and the Junos XML API easier. The next few pages discuss the Perl and Java libraries supported by Juniper. There are also NETCONF Libraries that are maintained by the open source community. There are Python, and Ruby libraries available for working with NETCONF and the XML API. We will be using the Python library later in this course.

Junos XML Protocol Perl Client

- Perl module -- JUNOS::Device
 - Execute XML API Operational Commands
 - Retrieve, modify, and commit configurations
- Includes sample scripts
- Download from:
 - <http://www.juniper.net/support/downloads/?p=junoscript>



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 39

The Junos XML Perl Module

To help with your XML API development projects, Juniper provides a Junos XML protocol Perl client and a set of Java classes. We will first discuss the Perl client.

Juniper Networks provides a Perl module JUNOS::Device to help you more quickly and easily develop custom Perl scripts for configuring and monitoring switches, routers, and security devices running the Junos OS. The module implements a JUNOS::Device object that client applications can use to communicate with the Junos XML protocol server on a device running the Junos OS. The Perl distribution includes several sample Perl scripts, which illustrate how to use the module in scripts that perform various functions. The Junos XML protocol Perl distribution uses the same directory structure for Perl modules as the Comprehensive Perl Archive Network (CPAN). This includes a lib directory for the JUNOS module and its supporting files, and an examples directory for the sample scripts. Client applications use the JUNOS::Device object to communicate with a Junos XML protocol server. The library contains several modules, but client applications directly invoke only the JUNOS::Device object. All of the sample scripts use this object.

Continued on the next page.

The Junos XML Perl Module (contd.)

The sample scripts illustrate how to perform the following functions:

- *diagnose_bgp.pl*—Illustrates how to write scripts to monitor device status and diagnose problems. The sample script extracts and displays information about a device's unestablished Border Gateway Protocol (BGP) peers from the full set of BGP configuration data. The script is in the examples/diagnose_bgp directory in the Junos XML protocol Perl distribution.
- *get_chassis_inventory.pl*—Illustrates how to use a predefined query to request information from a device. The sample script invokes the get_chassis_inventory method with the detail option to request the same information as the Junos XML get-chassis-inventorydetail/get-chassis-inventory tag sequence and the command-line interface (CLI) operational mode command show chassis hardware detail. The script is in the examples/get_chassis_inventory directory in the Junos XML protocol Perl distribution.
- *load_configuration.pl*—Illustrates how to change a device configuration by loading a file that contains configuration data formatted with the Junos XML tag elements. The distribution includes two sample configuration files, set_login_class_bar.xml and set_login_user_foo.xml; however, you can specify a different configuration file on the command line. The script is in the examples/load_configuration directory in the Junos XML protocol Perl distribution.

The following sample scripts are used together to illustrate how to store and retrieve data from the Junos XML API (or any XML-tagged data set) in a relational database. Although these scripts create and manipulate MySQL tables, the data manipulation techniques that they illustrate apply to any relational database. The scripts are provided in the examples/RDB directory in the Perl distribution:

- *get_config.pl*—Illustrates how to retrieve routing platform configuration information.
- *make_tables.pl*—Generates a set of Structured Query Language (SQL) statements for creating relational database tables.
- *pop_tables.pl*—Populates existing relational database tables with data extracted from a specified XML file.
- *unpop_tables.pl*—Transforms data stored in a relational database table into XML and writes it to a file.

For instructions on running the scripts, see the README or README.html file included in the Perl distribution.

The Junos XML protocol Perl client can be downloaded from <http://www.juniper.net/support/downloads/?p=junoscript>.

For detailed instructions on installing the Junos XML protocol Perl client and associated sample scripts, consult the Junos XML Management Protocol Developer Guide.

NOTE: The Junos XML protocol Perl client software should be installed and should run on a regular computer with a UNIX-like operating system; it is not meant to be installed on a Juniper Networks device.

NETCONF Java Toolkit

- Makes working with NETCONF in Java easier
- Includes four main classes

Class	Description
Device	Defines the device on which the NETCONF server runs, and represents the SSHv2 connection and default NETCONF session with that device
NetconfSession	Represents a NETCONF session established with the device on which the NETCONF server runs.
XMLBuilder	Creates XML-encoded data.
XML	XML-encoded data that represents an operational or configuration request or configuration data.

- Download from:
 - <https://github.com/Juniper/netconf-java/releases>

The NETCONF Java Toolkit Makes Working With NETCONF in Java Easier

The NETCONF Java toolkit provides an object-oriented interface for communicating with a NETCONF server. The toolkit enables programmers familiar with the Java programming language to create Java applications to connect to a device, open a NETCONF session, construct configuration hierarchies in XML, and create and execute operational and configuration requests. You can create your own Java applications to manage and configure routing, switching, and security devices. The NETCONFJava toolkit provides classes with methods that implement the functionality of the NETCONF protocol operations defined in RFC 4741. All basic protocol operations are supported. The NETCONF XML management protocol uses XML-based data encoding for configuration data and remote procedure calls. The toolkit provides classes and methods that aid in creating, modifying, and parsing XML. The NETCONF Java toolkit has four basic classes are:

Class Summary

There are four main NETCONF classes:

- *Device* – Defines the device on which the NETCONF server runs, and represents the SSHv2 connection and default NETCONF session with that device.
- *NetconfSession* - Represents a NETCONF session established with the device on which the NETCONF server runs.
- *XMLBuilder* -- Creates XML-encoded data.
- *XML* – XML-encoded data that represents an operational or configuration request or configuration data.

Continued on the next page.

Class Summary (contd.)

A configuration management server is generally a PC or workstation that is used to configure a router, switch, or security device remotely. The communication between the configuration management server and the NETCONF server through the NETCONF Java toolkit involves:

- Establishing a NETCONF session over SSHv2 between the configuration management server and the NETCONF server
- Creating RPCs corresponding to requests and sending these requests to the NETCONF server
- Receiving and processing the RPC replies from the NETCONF server

To use the NETCONF Java toolkit, you must install the toolkit and add the .jar path to your CLASSPATH. Once the toolkit is installed, you connect to a device, create a NETCONF session, and execute operations by adding the associated code to a Java program file, which is then compiled and executed. For more information about creating NETCONF Java toolkit programs, see “Creating and Executing a NETCONF Java Application.”

The NETCONF Java Toolkit can be downloaded from <https://github.com/Juniper/netconf-java/releases>.

For more information about how to use the NETCONF Java Toolkit, consult the NETCONF Java Toolkit Developer Guide.

Summary

- In this content, we:

- Described the NETCONF Protocol
- Explained the capabilities of the XML API
- Described the use of XSLT, SLAX, and XPath in XML API development

We Discussed:

- The NETCONF Protocol;
- The capabilities of the XML API; and
- The use of XSLT, SLAX, and XPath in XML API development.

Review Questions

1. List three NETCONF operation level commands.
2. What W3C language is designed to manipulate XML?
3. What is the purpose of XPath?
4. Name at least one language library that simplifies working with NETCONF and the XML API.

Review Questions

- 1.
- 2.
- 3.
- 4.

Lab: NETCONF and the XML API

- Open a NETCONF Session
- Explore and use the XML API

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 45

Lab: NETCONF and the XML API

The slide provides the objectives for this lab.

Answers to Review Questions

1.

There are 11 NETCONF Operational commands including :<lock>, <unlock>, <get>, <get-config>, <edit-config>, <copy-config>, <commit>, <discard-changes>, <delete-config>, <validate>, <create subscription>, <close-session>, and <kill-session>

2.

XSLT is the W3C language designed to manipulate XML.

3.

The purpose of XPath is to navigate an XML tree and identify elements for examination or manipulation.

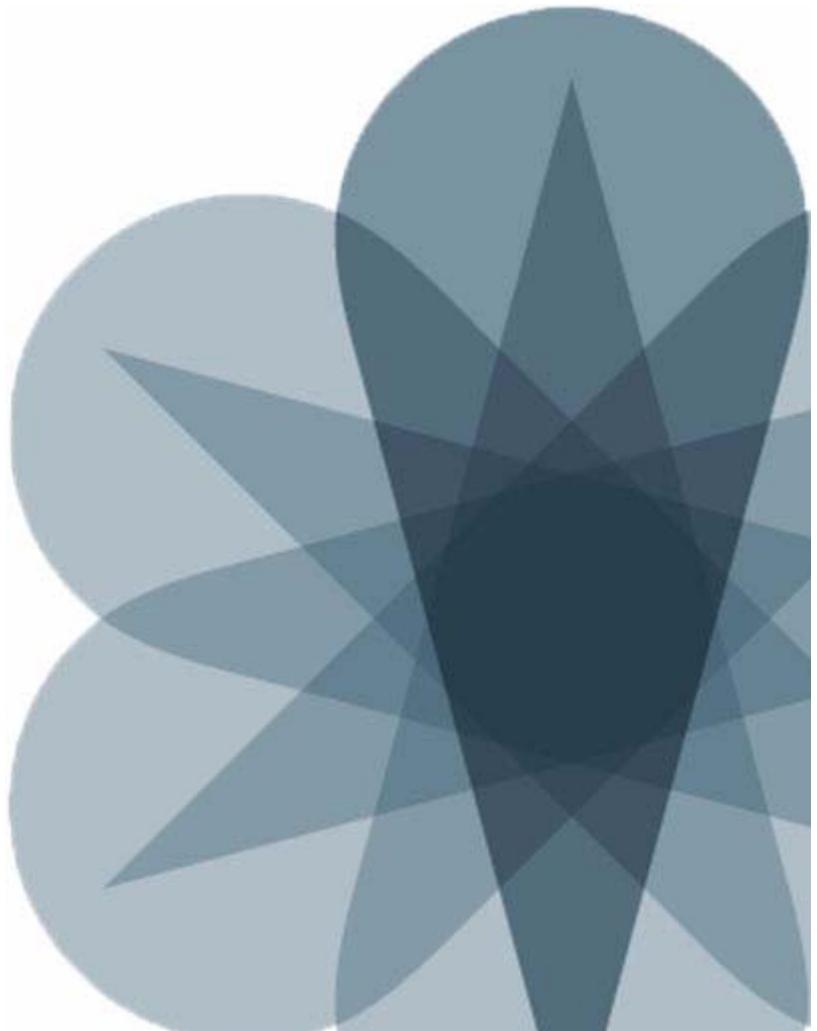
4.

The Junos Perl Module and NETCONF Java classes are libraries that make working with NETCONF and the XML API easier.



Junos Platform Automation and DevOps

Chapter 4: JSON and YAML



Objectives

- After successfully completing this content, you will be able to:
 - Identify where JSON and YAML are used in Junos OS Automation
 - Read JSON and YAML documents

We Will Discuss:

- Where JSON and YAML are used in Junos OS Automation; and
- The syntax and fundamental concepts of JSON and YAML.

Agenda: JSON and YAML

- JSON and YAML Overview
- JSON Basics
- YAML Basics

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 3

JSON and YAML Overview

The slide lists the topics we will discuss. We discuss the highlighted topic first.

What are JSON and YAML?

- **JavaScript Object Notation (JSON)**
 - Easy to read
 - Language independent
 - Familiar syntax for programmers
- **YAML Ain't Markup Language (YAML)**
 - Easier to read
 - Language independent

JavaScript Object Notation (JSON)

JSON is a lightweight data-interchange format. JSON is human-readable but even easier for machines to parse. JSON is language-independent, but uses conventions that are familiar to programmers of the C family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and others. These properties make JSON a popular data-interchange language.

YAML Ain't Markup Language (YAML)

YAML is a Unicode-based data serialization language designed around common native data types found in agile programming languages. The language is extremely human-friendly and is flexible for a variety of tasks, including: configuration files, Internet messaging, object persistence, and data auditing. For more on YAML, see: <http://yaml.org>

JSON and YAML Similarities

```
{
  "configuration" : {
    "@" : {
      "junos:changed-seconds" :
"1489765454",
      "junos:changed-localtime"
: "2017-03-17 15:44:14 UTC"
    },
    "system" : {
      "services" : {
        "ssh" : [null],
        "telnet" : [null],
        "netconf" : {
          "ssh" : [null]
        }
      }
    }
  }
}
```

JSON

```
configuration:
  "@":
    "junos:changed-seconds":
1489765454
    "junos:changed-localtime":
"2017-03-17 15:44:14 UTC"
  system:
    services:
      ssh:
        -
      null
      telnet:
        -
      null
    netconf:
      ssh:
        -
      null
```

YAML

JSON and YANG Similarities

JSON and YAML are similar enough that, in most cases, you can convert between the two languages. Not everything in YAML can convert to JSON, but everything in JSON can convert to YAML.

Both JSON and YAML, as data interchange formats, are human-readable. However, the foremost design goal of JSON is universality, so while the language is simple to generate and parse, there is a sacrifice of some human readability. JSON uses a lowest common denominator information model, ensuring any JSON data can be processed easily by every modern programming environment.

In contrast, the highest priority of YAML is human readability and support for serializing arbitrary native data structures. YAML has simple-to-read files, but the language is more complex to generate and parse. In addition, YAML does not follow the lowest common denominator data types, and so requires more complex processing when crossing between different programming environments.

YAML is a superset of JSON, with improved human readability and a more complete information model. Every JSON file is also a valid YAML file. This makes migration from JSON to YAML easy.

Where are JSON and YAML Used?

■ JSON

- Junos OS Configuration
- Junos REST API

■ YAML

- Junos PyEZ tables
- Ansible Playbooks
- JSNAPy

JSON

JSON is widely used on the Internet as a method to move data between Web servers and AJAX clients on web browsers. For example, small news feeds on web pages that update without refreshing the page often use JSON as the method to serialize the data.

JSON can be used to view an input configuration data through the CLI. JSON can also be used to view and input data to the configuration through NETCONF. JSON can be used to view operational mode commands in the CLI and through NETCONF.

JSON is also used by the Junos OS REST API to retrieve configuration data. You will see how that works in the REST API chapter.

YAML

YAML is used to define Junos PyEZ tables and views, which are used to map portions of an PRC response into a Python data structure. YAML is also used in Ansible to create Playbooks that automate Juniper devices. We will see examples of both later in this chapter.

Agenda: JSON and YAML

- JSON and YAML Overview
 - JSON
 - YAML

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 7

JSON

The slide highlights the topic we discuss next.

JSON Basics

- JSON is case sensitive
- JSON uses curly braces { } for structure
- Whitespace is ignored in JSON
- JSON does not offer a way to comment the code
- JSON Structures
 - Objects
 - Arrays

JSON Basics

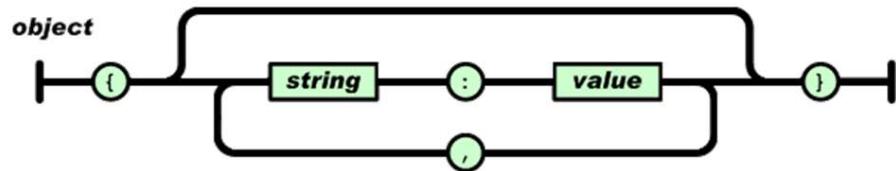
JSON is a data serialization language most often used to move data between an Internet client and server. JSON is not a programming language, so does not perform loops or have conditional statements. It has, in many instances, taken the place of XML as a method of structuring data.

Working with JSON is similar in many ways to C or Java-type languages. JSON uses curly brackets { } to structure blocks of code and whitespace is ignored. JSON is case-sensitive.

One thing worth noting is that JSON does not have a defined way to add comments in your code; all code is treated as data. JSON is a simple language; there are only two data structures: objects and arrays. We will discuss each of these next.

JSON Objects

- JSON objects contain key-value pairs



- Example:

```
{
  "interface" : "ge-0/0/0.0",
  "address"   : "172.17.1.1/24"
}
```

JSON Objects

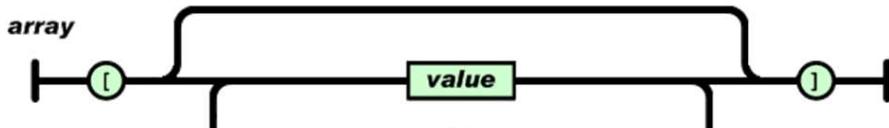
JSON objects begin with and end with curly brackets. As seen on the slide, objects are a collection of one or more key-value pairs. A key-value pair has an identifying key followed by a colon and then a value. Because of this format, objects are often referred to as key-value pairs. A single object can contain a series of key-value pairs, separated by commas.

The example in the slide shows an object with two key-value pairs.

The graphics from this and the next four slides come from the json.org website.

JSON Arrays

- Arrays encapsulate data in []



- Example

```
{
  "physical-interface": [
    {
      "name": [
        {
          "data": "ge-0/0/0"
        }
      ],
      "admin-status": [
        {
          "data": "up"
        }
      ]
    }
}
```

JSON Arrays

An array is an ordered collection of values. Array structures begin and end with square brackets, as seen in the slide. Arrays contain one or more values; array values are separated by commas.

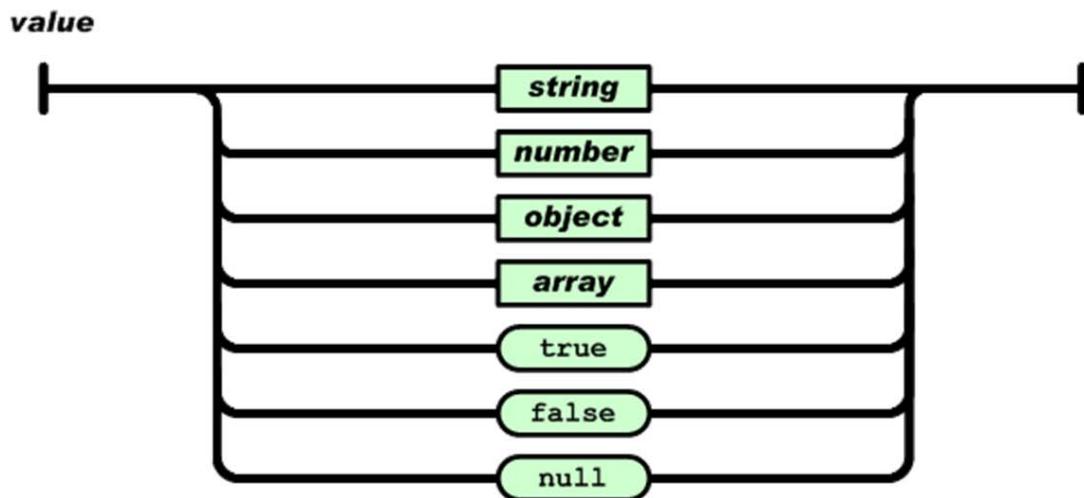
The example on the slide shows an object, and the object contains a key-value pair. The key is called “physical-interface”. The value is an array, as shown by the left square bracket. The array contains two objects: the first object contains a key called “name” and the second object contains a key called “admin-status”.

The value of the “name” key is another array with an object that has a key called “data”. The “data” key has a string value of “ge-0/0/0”. The “admin-status” key has an array as a value. The array contains an object with a key called “data”. The “data” key has a string value of “up”.

The example is an excerpt taken from a Junos OS CLI command. You will see how we generated it shortly.

JSON Values

- Object and array values enable nesting

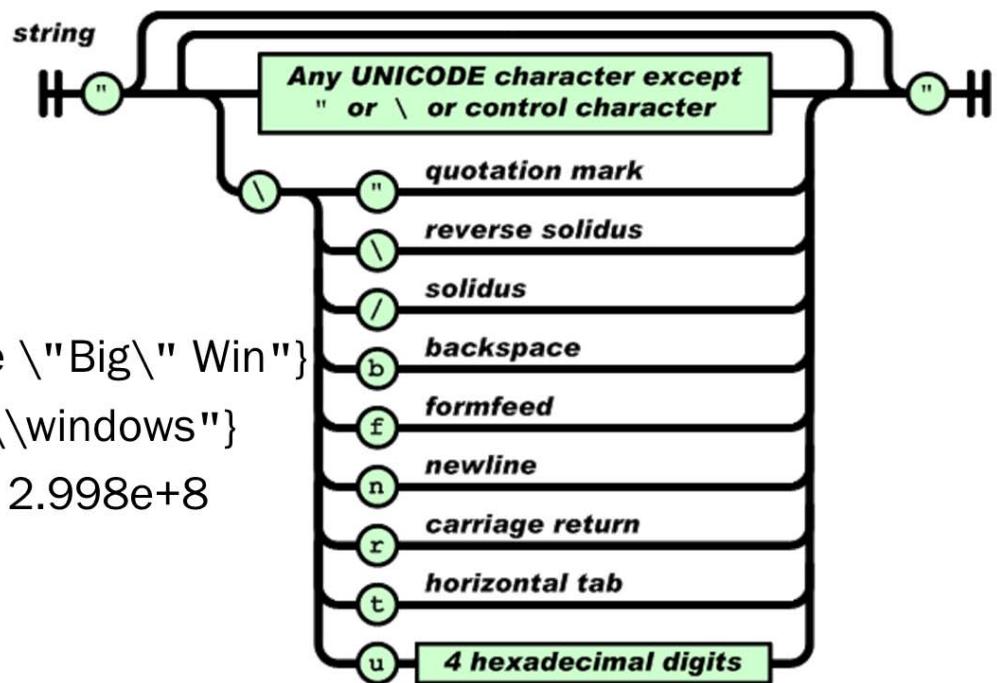


JSON Values

JSON has a limited number of values that can be used within objects and arrays. The slide illustrates possible JSON values: strings, numbers, true, false, null, objects, and arrays.

The nesting we saw previously is possible because JSON values can be objects and arrays.

JSON Strings and Numbers



■ Examples

```
{"title": "The \"Big\\\" Win"}
```

```
{"path": "c:\\windows"}
```

```
{"light m/s": 2.998e+8}
```

JSON Strings and Numbers

JSON strings can consist of any Unicode character except for double quotes and backslashes. However, you can use double quotes and backslashes if you use the escape sequence first, so precede the characters to be escaped with a backslash as you would in C or Java. The slide shows examples of escaping double quotes and backslashes.

JSON numbers can be either integers or floating point numbers with decimal points. You can express exponential numbers using either uppercase E or lowercase e. The slide shows an example of an exponent. Numbers enclosed in double quotes are treated as strings.

Configuration in JSON

```
[edit]
lab@vMX-1# show protocols bgp | display json
{
    "configuration" : {
        "@" : {
            "junos:changed-seconds" : "1489765454",
            "junos:changed-localtime" : "2017-03-17 15:44:14 UTC"
        },
        "protocols" : {
            "bgp" : {
                "group" : [
                    {
                        "name" : "ext-peers",
                        "type" : "external",
                        "peer-as" : "65513",
                        "neighbor" : [
                            {
                                "name" : "172.17.1.2"
                            }
                        ]
                    }
                ]
            }
        }
    }
}
```

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 13

Configuration in JSON

Displaying the Junos OS configuration using JSON

In previous chapters, we've seen how to use the format property to retrieve the configuration in text and XML formats. You can use the format property in NETCONF to retrieve JSON, similar to what is shown below.

```
<rpc>
  <get-configuration format="json">
    <!-- tag elements for the configuration elements to return -->
  </get-configuration>
</rpc>
]]>]]>
```

You can get the same result from the Junos OS CLI by issuing the `show configuration | display json` command while in operational mode.

The BGP information on the slide is generated by issuing the `show protocols bgp | display json` command from the [edit] hierarchy.

Operational Mode Command Output in JSON

```
lab@vMX-1> show chassis alarms | display json
{
    "alarm-information" : [
        {
            "attributes" : {"xmlns" :
"http://xml.juniper.net/junos/16.2R1/junos-alarm"},
            "alarm-summary" : [
                {
                    "no-active-alarms" : [
                        {
                            "data" : [null]
                        }
                    ]
                }
            ]
        }
    ]
}
```

Operational Mode Command Output in JSON

You can also use the `| display json` pipe command to show the output of operational commands in JSON. For example, the `show chassis alarms` command is shown on the slide.

To learn more about JSON, refer to <http://www.json.org>

Agenda: JSON and YAML

- JSON and YAML Overview
- JSON
- YAML

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 15

YAML

The slide highlights the topic we discuss next.

YAML Basics

- YAML is case-sensitive
- YAML uses indents for structure
- YAML uses spaces, not tabs
- YAML documents start with three dashes ---
- Comments begin with a #
- Strings do not need quotes unless they include special characters
- YAML Structures:
 - Mappings
 - Sequences

YAML Basics

YAML is widely used to store configuration data such as in Ansible Playbooks and OpenStack Heat templates. Like JSON, YAML is replacing many instances of XML, often because of its superior readability and ease of creation.

YAML is case-sensitive. YAML does not ignore whitespace; it uses whitespace for structure, similar to Python. YAML requires indents to be made with spaces, not tabs. Although not required, it is customary to start all YAML documents with three dashes ---. In YAML, strings are not required to have surrounding quotes, as JSON does. In YAML, comments start with the pound sign, or hash mark (#).

Like JSON, YAML has two basic structures: mappings and sequences. We will see how these structures work next.

YAML Mappings

- Mappings are key-value pairs
- Mappings are similar to JSON objects
- Example:

name : Bob

height : 6 foot

YAML Mappings

YAML mappings are equivalent to the key-value pairs found in JSON objects. Like the key-value pairs in JSON, the key (name) and value (Bob) on the first line are separated by a colon (:). Note that the strings do not require quotation marks.

YAML Sequences

- Sequences are lists or arrays of data
- Sequences are similar to JSON arrays
- Sequence items start with a dash -
- Example:

```
- apple
- orange
- banana
```

YAML Sequences

Sequences are similar to JSON arrays; they store lists, or sequences, of data. The slide shows a sequence. Sequences do not need to have a name associated with them.

Nested Mappings and Sequences

- Mappings and Sequences can be nested

Fruit:

- apple
- orange
- banana

Vegetable:

cruciferous:

- radish
- wasabi

gourd:

- cucumber
- pumpkin

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 19

Nested Mappings and Sequences

This slide illustrates nested mappings and sequences. The `Fruit` mapping or object has a nested sequence. The `Vegetable` mapping has nested mappings of `cruciferous` and `gourd`. The `cruciferous` and `gourd` mappings each have their own nested sequences. Note how whitespace is used to indicate hierarchy, or nesting.

Next, we look at how these are used in Junos OS automation.

Junos PyEZ Example

```
---
```

```
bgpTable:  
    rpc: get-bgp-neighbor-information  
    item: bgp-peer  
    view: bgpView  
    key: peer-id  
bgpView:  
    fields:  
        local_as: local-as  
        peer_as: peer-as  
        local_address: local-address  
        peer_id: peer-id  
        local_id: local-id  
        route_received: bgp-rib/received-prefix-count
```

Junos PyEZ Example

The slide shows the YAML *bgp.yml* file. YAML files use the *.yml* extension. The *bgp.yml* file is used by Junos PyEZ to format data retrieved by the *get-bgp-neighbor-information* RPC. This particular YAML file uses nested mappings but no sequences.

The slide shows the YAML *bgp.yml* file. YAML files use the *.yml* extension. The *bgp.yml* file is used by Junos PyEZ to filter and map the data to a Python data structure, data retrieved by the *get-bgp-neighbor-information* RPC.

This particular YAML file uses nested mappings but no sequences.

Ansible Example

```
---
- name: Install Junos OS
  hosts: dc1
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no
  vars:
    wait_time: 3600
    pkg_dir: /var/tmp/junos-install
    OS_version: 14.1R1.10
    OS_package: jinstall-14.1R1.10-domestic-signed.tgz
    log_dir: /var/log/ansible
  . . . Trimmed. . .
```

Ansible Example

This slide shows a sample YAML file used to install the Junos OS on a host device called dc1. This example begins with a sequence and has both nested mappings and sequences. Note how much easier it is to read this document compared to an equivalent XML document.

To learn more about YAML, see <http://www.yaml.org>

Summary

- In this content, we:

- Identified where JSON and YAML are used in Junos OS Automation
- Learned to read JSON and YAML documents

We Discussed:

- Where JSON and YAML are used in Junos OS automation; and
- The syntax and fundamental concepts of JSON and YAML

Review Questions

1. What language are JSON and YAML replacing?
2. Is whitespace important in JSON?
3. How do you create a comment in YAML?

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 23

Review Questions:

- 1.
- 2.
- 3.

Lab: JSON and YAML

- View Junos OS configuration in JSON
- View Junos OS operational mode command responses in JSON
- Create and validate a JSON and YAML file

Lab: JSON and YAML

The slide provides the objectives for this lab.

Answers to Review Questions

1.

JSON and YAML are primarily replacing XML.

2.

Whitespace is ignored in JSON.

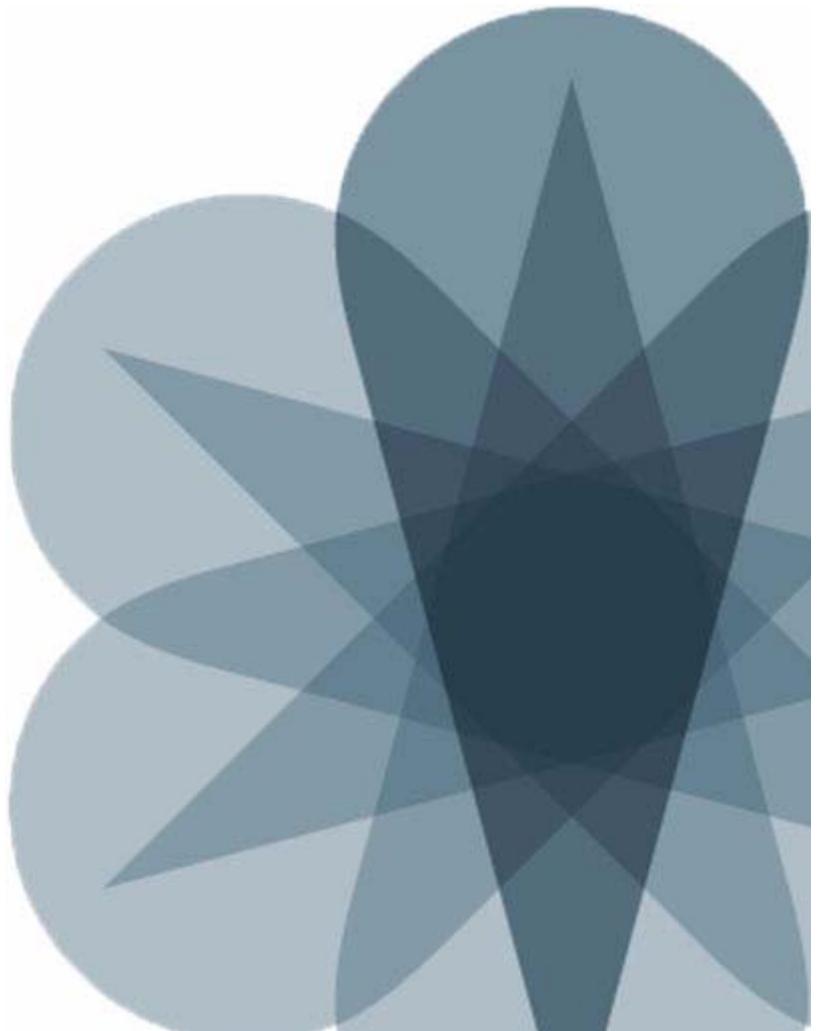
3.

You use a # sign to create a comment in YAML.



Junos Platform Automation and DevOps

Chapter 5: Python and Junos PyEZ



Objectives

- After successfully completing this content, you will be able to:
 - Build a Python environment where you can manage devices running the Junos OS
 - Get information from Junos OS RPCs using PyEZ scripts
 - Modify the Junos OS configuration using PyEZ
 - Use PyEZ tables and views to view RPCs and modify the Junos OS configuration
 - Use the PyEZ exception handling modules

We Will Discuss:

- How to build a Python development environment and how to connect to devices running the Junos OS;
- How to use PyEZ scripts to retrieve information from Junos OS RPCs;
- How to modify the configuration on devices running the Junos OS;
- How to use PyEZ tables and views to extract operational and configuration data and modify the Junos OS configuration RPCs. and modify the Junos OS configuration; and
- How to use PyEZ exception handling modules.

Agenda: Python and PyEZ

- Introduction to Python and PyEZ
- Python Development Environment
- Working with RPCs
- Working with an Unstructured Configuration
- Working with Tables and Views
- PyEZ Exception Handling

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 3

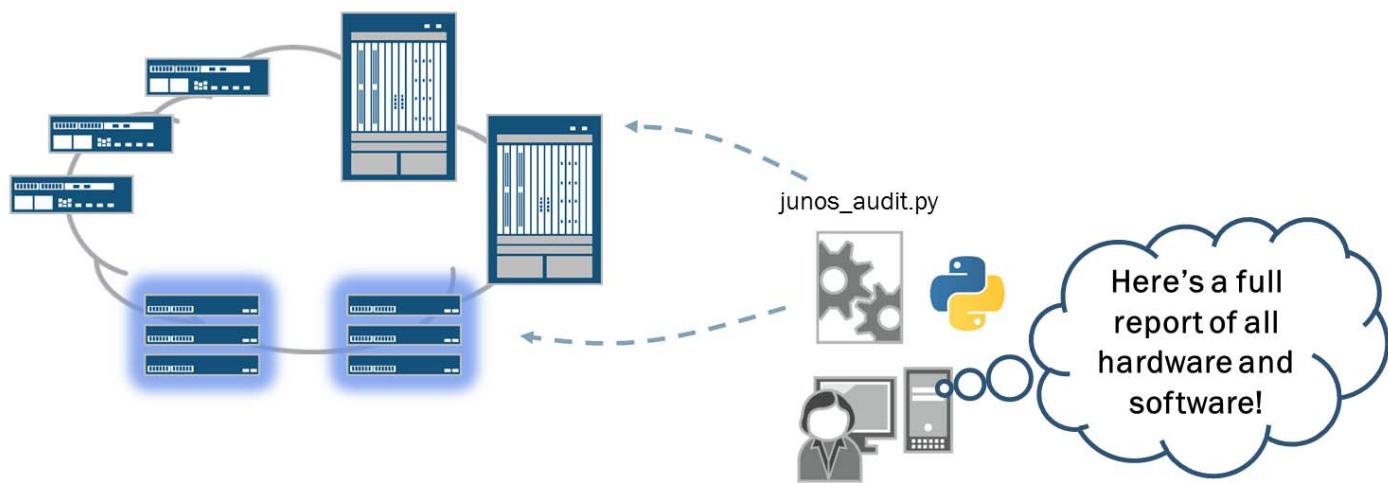
Introduction to Python and Junos PyEZ

The slide lists the topics we will discuss. We discuss the highlighted topic first.

PyEZ Use Cases (1 of 3)

- Software and hardware audits

- On-demand access to all hardware and software running on Junos platforms on devices running Junos OS

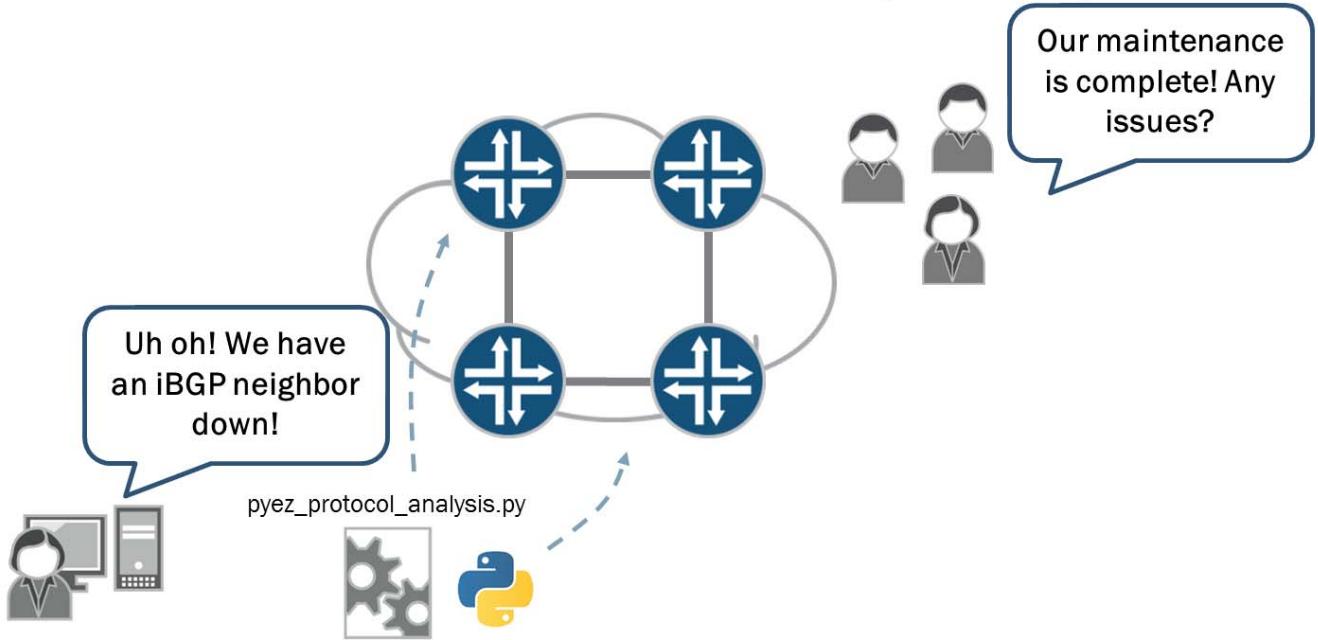


PyEZ Use Cases, Part 1

As network engineers, we have all had to drudge through some sort of network audit. One way to simplify this task is to open an SSH terminal to every device and manually populate a spreadsheet with the relevant audit information. PyEZ enables you to have the necessary code to automate this repetitive task easily.

PyEZ Use Cases (2 of 3)

- Automated data collection
 - Analyze device and protocol state before and after maintenance windows to catch issues proactively



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

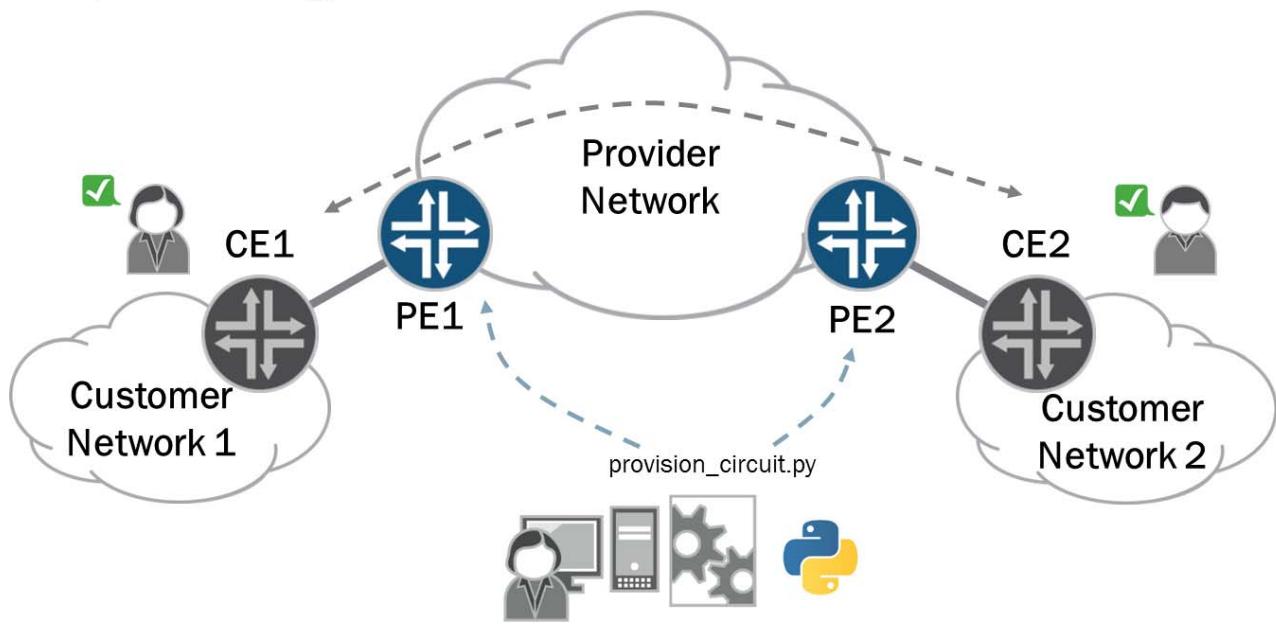
www.juniper.net | 5

PyEZ Use Cases, Part 2

Network operations can be a risky business. Sometimes you have to analyze situations and make your best guess on the impacts of changes to protocol configurations or services. This hinders progress and makes the network a bottleneck for needed change. PyEZ provides the necessary code to reduce the risk associated with network operations. You can gather protocol state before and after maintenance windows in order to analyze for potential impacts.

PyEZ Use Cases (3 of 3)

- Basic automated configuration deployment
 - Deploy basic Junos OS configuration templates for service provisioning to eliminate errors



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 6

PyEZ Use Cases, Part 3

Network configurations are almost always deployed manually. PyEZ provides the code to automate configuration deployment. This frees up engineers to perform strategic tasks instead of spending valuable time on manual CLI changes.

Junos Automation Now Includes Python On-Box

- Python extensions package embedded in Junos OS
 - On-box as of Junos 16.1R3
 - Supports Python 2.7
- Uses PyEZ APIs
 - Easy to execute RPCs
 - Easy to perform operational and configuration tasks
- Able to perform tasks not possible in XSLT and SLAX

Junos Automation Now Includes Python On-Box

Junos automation consists of a suite of tools used to automate operational and configuration tasks on network devices running the Junos operating system. The Junos automation toolkit is part of the standard Junos OS available on all switches, routers, and security devices running Junos OS. Junos automation tools, which leverage the native XML capabilities of Junos OS, include commit scripts, operation (op) scripts, event policies and event scripts, SNMP scripts, and macros.

The Junos Automation toolkit now includes Python 2.7 on-box. Python is currently on MX, PTX, EX, and QFX platforms.

Junos automation simplifies complex configurations and reduces potential configuration errors. It saves time by automating operational and configuration tasks. It also speeds troubleshooting and maximizes network uptime by warning of potential problems and automatically responding to system events.

Because of this, Junos automation can utilize the knowledge and expertise of experienced network operators and administrators to allow a business to use this combined expertise across the organization.

Junos automation scripts can be written in several scripting languages: Extensible Stylesheet Language Transformations (XSLT), Stylesheet Language Alternative Syntax (SLAX), or Python. XSLT is a standard for processing Extensible Markup Language (XML) data and is designed to convert one XML document into another. SLAX is an alternative to XSLT. It has a simple syntax that follows the style of C and PERL, but retains the same semantics as XSLT. Programmers who are familiar with C often find it easier to learn and use SLAX. You can easily convert SLAX scripts into XSLT and convert XSLT scripts into SLAX. Python is a commonly used, open-source programming language with extensive standard and community libraries. On devices running Junos OS, Python automation scripts can leverage Junos PyEZ APIs to simplify many operational and configuration tasks.

Junos OS Python Modules

- Junos OS includes many different open-source Python modules to facilitate automation



© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 8

Junos OS Python Modules

Devices running Junos OS include several open-source Python modules that can be used in Python applications. For example:

- **Crypto** - Collection of Python modules implementing cryptographic algorithms and protocols. Crypto modules provide various secure hash and encryption functions.
- **ECDSA** - Provides an implementation of ECDSA cryptography, which can be used to create key pairs, sign messages, and verify signatures.
- **Jinja2** - Fast, secure, designer-friendly templating language for Python. For more information about Jinja2, see <http://jinja.pocoo.org/docs/dev/>.
- **jnpr.junos (Junos PyEZ)** - Microframework for Python that enables users to automate devices running Junos OS. Junos PyEZ is designed to provide the capabilities that a user would have on the Junos OS command-line interface (CLI) in an environment built for automation tasks.
- **lxml** - XML processing library that combines the speed and XML feature completeness of the C libraries libxml2 and libxslt with the simplicity of a native Python API. For more information about the lxml Python module, see <http://lxml.de/tutorial.html>.
- **MarkupSafe** - Provides the ability to escape and format an XML, HTML, or XHTML markup safe string.
- **ncclient** - Facilitates client scripting and application development through the NETCONF protocol. For more information about ncclient, including documentation for some external APIs, see <http://ncclient.grnet.gr/>.

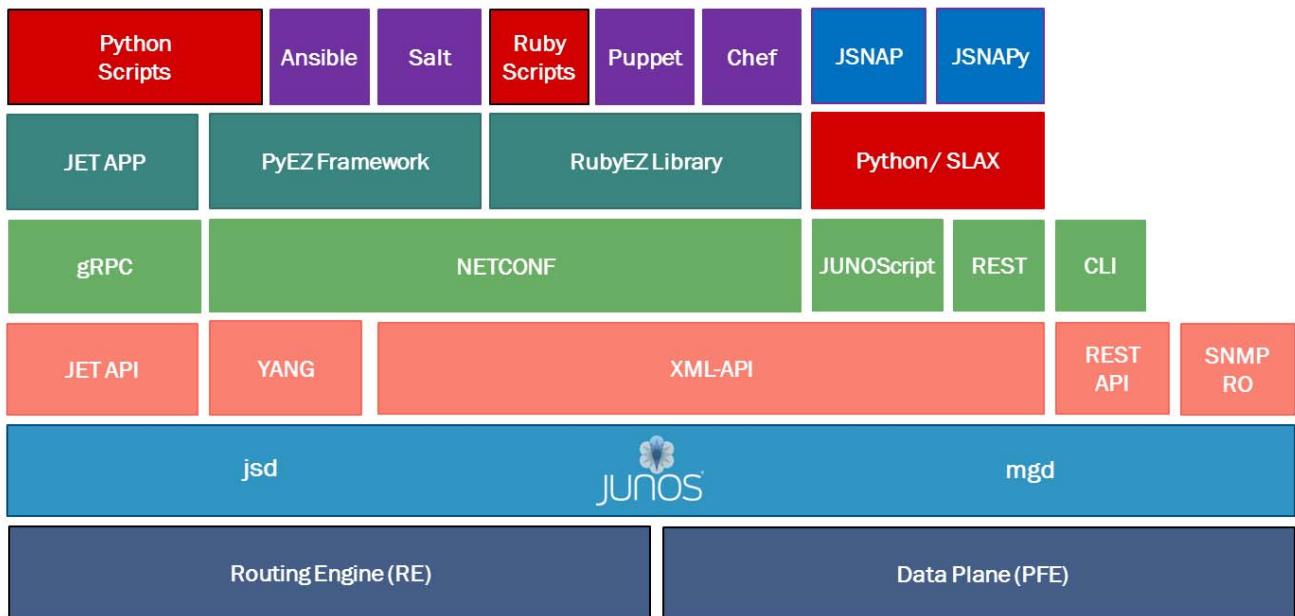
Continued on the next page.

Junos OS Python Modules (contd.)

- **netaddr** - Network address manipulation library that enables processing of Layer 2 and Layer 3 network addresses.
- **Paho** - Serves as a client class that enables applications to connect to an MQTT broker to publish messages, to subscribe to topics, and receive published messages.
- **Paramiko** - SSH2 protocol library that provides the ability to make SSH2 protocol-based connections. This module supports all major ciphers and hash methods for both client and server modes.
- **pyang** - Extensible YANG validator and converter that enables the processing, validation, and conversion of YANG modules.
- **PyYAML** - Python module used to serialize and deserialize data in YAML format.
- **Thrift** - Provides Python bindings for the Apache Thrift framework.
- **scp** - Implementation of the SCP protocol for Paramiko that uses Paramiko transport to send and receive files via the SCP protocol.

The PyEZ Package

- PyEZ is a “microframework” allowing for “Pythonic” access to Junos



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER NETWORKS Worldwide Education Services

www.juniper.net | 10

The PyEZ Package

PyEZ is a microframework to operate Junos OS devices using Python. A microframework is a package or library that provides code that is useful for a larger application. This slide illustrates that we may use PyEZ in conjunction with other packages/modules, such as ncclient/paramiko, as well as customize Python to operate networks leveraging Junos OS. As we will see shortly, the PyEZ methods hide some of complexity from the individual developing the Python/PyEZ scripts.

PyEZ Modules

Modules	Description
device	Defines the Device class, which represents the device running Junos OS and enables you to connect to and retrieve facts from the device.
exception	Defines exceptions encountered when accessing, configuring, and managing devices running Junos OS.
factory	Contains code pertaining to Tables and Views, including the <code>loadyaml()</code> method, which is used to load custom Tables and Views.
op	Includes predefined operational Tables and Views that can be used to filter output for common operational commands.
resources	Includes predefined configuration Tables and Views representing specific configuration resources, which can be used to programmatically configure devices running Junos OS.
transport	Contains code used by the Device class to support different connection types such as telnet and serial console connections.
utils	Includes configuration utilities, file system utilities, shell utilities, software installation utilities, and secure copy utilities.

PyEZ Modules

To better understand how PyEZ aids in Junos OS automation, examine the various PyEZ modules listed on the slide and their purposes. In Junos PyEZ, each device is modeled as an instance of the `jnpr.junos.device.Device` class. The `device` module provides access to devices running Junos OS through a serial console connection, telnet, or a NETCONF session over SSH. All connection methods support retrieving device facts, performing operations, and executing RPCs on demand. Support for serial console connections and for telnet connections through a console server enables you to connect to and initially configure new or zeroized devices that are not yet configured for remote access.

The `utils` module defines submodules and classes that handle software installation, file system and copy operations, and configuration management. The `exception` module defines exceptions encountered when managing devices are running Junos OS.

The `op`, `resources`, and `factory` modules pertain to Tables and Views. The `op` module contains predefined operational Tables and Views that can be used to extract specific information from the output of common operational commands on devices running Junos OS. The `resources` module contains predefined configuration Tables and Views that can be used to configure specific resources on devices running Junos OS. The `factory` module contains methods that enable you to load your own custom Tables and Views in Junos PyEZ applications.

Agenda: Python and PyEZ

- Introduction to Python and PyEZ
→ Python Development Environment
- Working with RPCs
- Working with an Unstructured Configuration
- Working with Tables and Views
- PyEZ Exception Handling

Python Development Environment

The slide highlights the topic we discuss next.

Python Development Environments

- Python and Python modules are included on devices running Junos OS
 - Junos OS Release 16.1R3 for MX Series and PTX Series devices
- Included in the JET virtual machine
- Install on personal workstation
 - Install Python
 - Install PyEZ Dependencies
 - Install PyEZ Package

Python Development Environment

To work with Python and PyEZ, you will need a development environment. Python is now included on several Juniper devices. You may wonder if you can develop your scripts on box; the libraries needed to run Python and PyEZ are included, however, there is no Python editor and you cannot run the Python interpreter on box.

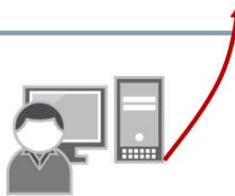
You may be curious to know if you can use the JET VM as a Python and PyEZ development environment. The JET VM is optimized to work specifically with the JET APIs and is not configured to work with the XML Management Protocol APIs, and so does not have PyEZ installed. Potentially, it could be configured with the additional libraries needed, but it does not come set up to do PyEZ development.

Your best bet, and what we have done for the labs at the end of the chapter, is to configure a virtual machine with Python 2.7 or 3.4, the PyEZ dependencies, and also PyEZ, which we will discuss next.

Building a Basic PyEZ Environment

▪ Installation and Setup of PyEZ

```
bash$ sudo apt-get install python-pip
.
bash$ sudo apt-get install python-dev
.
bash$ sudo apt-get install libxml2-dev
.
bash$ sudo apt-get install libxslt-dev
.
bash$ sudo pip install junos-eznc
```



https://www.juniper.net/techpubs/en_US/junos-pyez2.0/topics/task/installation/junos-pyez-server-installing.html

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 14

Building a Basic PyEZ Environment

This slide illustrates the basics of getting PyEZ up and running on Cent OS. In the environment running the PyEZ and Python code, we must install a series of packages and libraries. The steps are similar for other distributions of Linux. For information about installing Junos PyEZ and any prerequisites on other operating systems, see the INSTALL file for your specific operating system in the Junos PyEZ GitHub repository at <https://github.com/Juniper/py-junos-eznc>.

Here are the most common PyEZ prerequisite packages:

- **pip** - Recall from the Python chapter that this is a Linux utility to install packages and modules from the Python Package Index (PyPI).
- **python-devel** - This provides additional libraries required for Python (Note that PyEZ was developed using Python 2.7. Earlier versions of Python do not work with PyEZ, and newer versions have not yet been tested, except for Python 3.4.)
- **libxml2-devel** - This provides additional XML libraries.
- **libxslt-devel** - This provides additional XSLT libraries.
- **gcc**
- **openssl**
- **libffi-devel**

See this link for all dependences for different operating systems: https://www.juniper.net/documentation/en_US/junos-pyez2.0/topics/task/installation/junos-pyez-server-installing.html.

Python virtualenv Package

- Used to create Python development environments
- Activate a virtualenv

```
[lab@jaut-desktop ~]$ source virtualenvs/jaut_env/bin/activate
(jaut_env) [lab@jaut-desktop ~]$
```

- Install Software in a virtualenv

pip install junos-eznc

- Deactivate a virtualenv

Deactivate

- Do not store project files in virtualenv

Python virtualenv Package

The virtualenv package enables developers to have custom Python environments for individual Python projects. A developer may need a specific version of Python for one project and have different requirements for a separate project. Virtual environments are perfectly suited for such situations. The lab accompanying this chapter uses a virtual environment.

To activate a virtual environment, use the source command and the path to the activate file, as shown on the slide. Once a virtualenv has been activated, the prompt will be prefaced with the name of the virtual env. On the slide, the virtual environment is jaut_env.

Once in the virtual environment, use pip to install any packages you need, including PyEZ. Once you are through with the virtual environment, deactivate it using the deactivate command.

Your virtual environment is designed to contain a particular version of Python and any packages necessary to the environment.

The virtual environment folder is not the right place to store your project files. The virtual environment is meant to store environment files that will be used for many projects. Use a separate directory to avoid confusion.

To learn more about virtualenv, visit the virtualenv website at <https://virtualenv.pypa.io/en/stable/#>.

Prepare Devices to be Managed

- Configure Management Network Connections

- Console, telnet, SSH

```
[edit system services]
```

```
lab@vMX-1# set netconf ssh
```

- Create a user account with sufficient permissions

- Configure public/private keys for authentication
(optional)

- Configure the language python statement

```
[edit system scripts]
```

```
lab@vMX-1# set language python
```

Prepare Devices to be Managed

In order for Python and PyEZ to connect to a device running the Junos OS, the device must be configured to accept connections. Typically developers use netconf over ssh to connect, but as of PyEZ 2.0, telnet and serial console connections can also be used. You must also have a user account with sufficient privileges. The account privileges necessary are the same as those needed to perform functions through the Junos OS CLI.

For added security, you can configure public/private keys for authentication. The process for creating and using public/private keys will be described shortly.

To execute Python scripts on a device running the Junos OS, you must configure the language Python statement at the [edit system scripts] hierarchy level, as shown on the slide.

Executing Unsigned Python Scripts (1 of 4)

- Python does not execute unsigned scripts by default
- Additional steps to executing unsigned scripts
 - Set language statement
 - Enable individual scripts
 - Store script in appropriate directory
 - Script must meet permission requirements

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 17

Executing Unsigned Python Scripts

Your scripts will most often be running in the development environment while in development, but once your scripts are completed, they can be moved to the device running the Junos OS. There, they can be stored and executed locally.

By default, you cannot execute unsigned Python scripts on devices running Junos OS. To enable the execution of unsigned Python automation scripts, you must configure the `language python` statement.

You must also enable each individual script by configuring the script filename under the hierarchy level appropriate to the script. You must store each file in specific directories, depending the use of the script. Finally, the scripts need to meet permission requirements.

Each of these steps are discussed in the next three slides.

Executing Unsigned Python Scripts (2 of 4)

- Enable individual scripts

```
[edit system scripts commit]  
lab@vMX-1#set file filename
```

```
[edit system scripts op]  
lab@vMX-1#set file filename
```

```
[edit event-options event-script]  
lab@vMX-1#set file filename
```

```
[edit system scripts snmp]  
lab@vMX-1#set file filename
```

Enabling Individual Scripts

In order for scripts to be executed on a device running the Junos OS, each script needs to be individually enabled in the Junos OS configuration hierarchy.

Commit scripts need to be specified in the [edit system scripts commit] hierarchy. Junos OS op, event, and SNMP scripts also need to be specified under the appropriate hierarchy.

We will go into depth on what constitutes commit, op, event, and SNMP scripts in the chapter on Junos automation scripting.

Executing Unsigned Python Scripts (3 of 4)

- Save scripts to the appropriate location

Script Type	Hard Drive	Flash Storage
Commit	/var/db/scripts/commit	/config/scripts/commit
OP	/var/db/scripts/op	/config/scripts/op
Event	/var/db/scripts/event	/config/scripts/event
SNMP	/var/db/scripts/snmp	/config/scripts/snmp

- Add the following if storing scripts on flash storage

[edit]

```
lab@vMX-1# set system scripts load-scripts-from-
flash
```

Save Scripts To the Appropriate Location

Python automation scripts must be stored in the appropriate directory on the device. The slide shows the appropriate location based on script type.

Note that there is a separate location if you are using flash storage.

Note: The Junos OS supports using symbolic links for files in the script directories, but the device will only execute the script at the target location if it is signed.

Executing Unsigned Python Scripts (4 of 4)

▪ Unsigned script permission requirements

- Script owner is either root or user in Junos OS super-user login class
- Only the script owner has write permission for the file
- Script executer needs read permissions
- Event and SNMP scripts need to configure the *python-script-user username* statement

Unsigned Script Permission Requirements

In order for scripts to run on a device running the Junos OS, there are three permissions required:

- The script owner either needs to be the root user or a user in the Junos OS super-user login class.
- The only user account that can have write permission for the file is the owner of the file.
- The user account used to execute the script needs to have execute permissions.
- Event and SNMP scripts, by default, still execute under the privileges of the user and group `nobody`. To execute the scripts using the access privileges of a specific user, you must configure the `python-script-user username` statement at the `[edit event-options event-script file filename]` hierarchy level for event scripts, or the `[edit system scripts snmp file filename]` hierarchy level for SNMP scripts, and specify a user configured at the `[edit system login]` hierarchy level.

These permissions are strict and help to keep the scripts and the device running the Junos OS safe from malicious intent.

The Python Interpreter

- Reads and executes commands interactively
- To start the Python interpreter

`python or python2.7`

- To exit the Python interpreter

`Ctrl+d or exit()`

Python Interpreter Output

```
(jaut_env) [lab@jaut-desktop ~]$ python
Python 2.7.12 (default, Apr 22 2017, 07:04:30)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-18)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> exit()
```

The Python Interpreter

Once Python is installed, you can start the Python interpreter. The Python interpreter allows you to have an interactive session with Python in much the same way that the Junos OS CLI allows you to have an interactive session with the Junos OS.

You can start the Python interpreter, sometimes called the Python shell, by typing `python` or `python2.7` at the command prompt. Once you are finished with the Python Interpreter, exit by typing `exit()` or pressing `Ctrl+d`.

You can use the Python Interpreter to do the same tasks you do in PyEZ scripts, especially short, familiar tasks. As tasks become more complex, many developers prefer to write scripts, which enable them to debug easily and save for future use.

Using the Python Interpreter to Get Help with PyEZ

```
(jaut_env) [lab@jaut-desktop ~]$ python
Python 2.7.12 (default, Apr 22 2017, 07:04:30)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-18)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.

>>> from jnpr.junos import Device
>>> help(Device)
>>>
```

Python Interpreter Output

```
Help on class Device in module jnpr.junos.device:
```

```
class Device(_Connection)
|   Junos Device class.
|
|   :attr:`ON_JUNOS`:
|       **READ-ONLY** -
|       Auto-set to ``True`` when this code is running on a Junos device,
|       vs. running on a local-server remotely connecting to a device.
```

Using the Python Interpreter to Get Help with PyEZ

PyEZ comes with a built-in help system. While working in the Python Interpreter, you can import a PyEZ library and then receive help using the syntax shown in the slide. You may find it useful to keep a separate instance of the Python interpreter open as you work so that you can reference the PyEZ help system as needed.

Once in the PyEZ help system you can navigate using the arrow keys on your keyboard and pressing **q** to quit the help system.

You can also find the Online PyEZ documentation at <http://junos-pyez.readthedocs.io/en/latest/>.

Executing Scripts

- Executing scripts by calling the Python interpreter followed by the name of the script

```
python myScript.py
```

- You can also configure direct execution

- Add `#!/usr/bin/env python` to the top of a script and the script will call the Python interpreter

myScript.py

- `#!` Is called the shebang

Don't need to
Explicitly call the
Python interpreter

Executing Scripts

When executing scripts from the command line, you call the Python program and follow this with the name of the script you want to run, which is `myScript.py` in the first example on the slide.

If you add the shebang (`#!`) followed by the path to the Python program to the top of a Python script, you can execute the script with just the script name, as seen in the second example in the slide.

Python Script Structure

1. Import libraries

```
from jnpr.junos import Device  
from pprint import pprint
```

2. Create device connection string

```
dev = Device(host='vMX-1', user='lab',  
password='lab123')
```

3. Open connection to a device

```
dev.open()
```

4. Perform task

```
pprint(dev.facts)
```

5. Close connection

```
dev.close()
```

Python Script Structure

When creating Python and PyEZ scripts, follow this process:

1. Import the needed libraries. As you go through the course, pay attention to which libraries are needed. You won't want to add libraries that you don't need as this can slow the loading process and use up more memory than required.
2. Create a device connection string. The connection string contains all the information needed to connect to a device running the Junos OS.
3. Once you have given the Device object the needed information, you can open the connection using the `open()` method.
4. Write the script. This is where most of the coding happens.
5. Once your script is finished, close the connection using the `close()` method.

Device Connection Strings

- NETCONF session over SSH

```
dev = Device(host=hostname, user=username, passwd=password)
```

- Telnet connection

```
dev = Device(host=hostname, user=username, passwd=password,
mode='telnet', port='23', gather_facts=True)
```

- Serial console connection

```
dev = Device(host=hostname, user=username, passwd=password,
mode='serial', port='/dev/ttyUSB0', gather_facts=True)
```

Device Connection Strings

PyEZ was originally designed to run over a NETCONF connection. With the release of PyEZ 2.0, PyEZ can now also connect over Telnet and serial console connections. The slide shows the connection strings required for each connection type. The `gather_facts` argument is optional. When connecting to a device, a NETCONF connection by default queries the device for device facts, Telnet and serial connections have to explicitly be told to gather device facts.

An upcoming slide depicts how to display device facts.

Prompting the User For Authentication

- Use `getpass()` to conceal user input

```
from jnpr.junos import Device
from getpass import getpass

hostname = raw_input("Device hostname: ")
username = raw_input("Device username: ")
password = getpass("Device password: ")

# NETCONF session over SSH
dev = Device(host=hostname, user=username,
passwd=password)
```

Prompting the User For Authentication

It is the user's responsibility to obtain the username and password authentication credentials in a secure manner appropriate for their environment. It is best practice to prompt for these authentication credentials during each invocation of the script, as shown in the following example, rather than storing the credentials in an unencrypted format.

The `getpass()` function allows users to input a password without having it display on the screen.

Securing Automated Access to Junos (1 of 4)

- Generate the SSH key on the Linux host

```
python-guru@ubuntu:~$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/python-guru/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again: ← Enter a passphrase if desired.
Your identification has been saved in /home/python-guru/.ssh/id_rsa.
Your public key has been saved in /home/python-guru/.ssh/id_rsa.pub.
The key fingerprint is:
d8:94:74:1e:1c:04:ec:8d:d5:84:4f:81:9c:ae:0f:37 python-guru@ubuntu
The key's randomart image is:
+--[ RSA 2048]----+
|       .o==Bo.   |
|       ..+B.o    |
|       .o=.o     |
. . . Trimmed . . .
python-guru@ubuntu:~$
```



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 27

Generate SSH Key on Linux Host

Recall from our previous example that we needed to put credentials into the Python script to establish the SSH session. This may present a security risk since the credentials could be viewed by any user with access to the file. A better approach would be to use SSH keys to authenticate access between the host running Python and the device running Junos OS.

This is a common approach used in applications such as GitHub for authentication. The first step is to create a new user on the Linux host that will log in to the device running Junos OS. Then, generate a new public/private SSH key pair for that user. Note that there is an option to add a passphrase to access the SSH key.

Securing Automated Access to Junos (2 of 4)

- Prepare the Linux and Junos OS environments
 - Copy the public key to required Junos hosts
 - Validate the default USER environment variable

```
python-guru@ubuntu:~$ scp .ssh/id_rsa.pub  
root@192.168.64.51:/tmp  
.  
.python-guru@ubuntu:~$ env | grep USER  
USER=python-guru  
.
```



Prepare the Linux and Junos OS Environments

After generating the SSH private/public key, copy the public key to every Junos device with which the host will interface. Also, note the default USER environment variable for the shell. If you do not specify a username in the Python script, the username defaults to this variable.

Securing Automated Access to Junos (3 of 4)

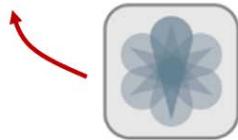
- Configure username on required Junos hosts

```
[edit]
root@R1# edit system login user python-guru

[edit system login user python-guru]
root@R1# set class super-user

[edit system login user python-guru]
root@R1# set authentication load-key-file /tmp/id_rsa.pub

[edit system login user python-guru]
root@R1# commit
```



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 29

Configure Username on Junos Hosts

Next, set the authentication for the user on all relevant devices running Junos OS. Instead of plain-text authentication, specify the location of the public key based on the directory where the key was copied in the previous step.

Securing Automated Access to Junos (4 of 4)

- Verify automated access to Junos
 - If needed, allow the Junos host to read the private SSH key



```
python-guru@ubuntu:~/ $ ssh python-guru@192.168.64.51 -I id_rsa.pub
--- JUNOS 12.1X46-D20.5 built 2014-05-14 20:38:10 UTC
python-guru@R1>
```



Verify Access to the Device Running Junos OS

Last, verify that the Linux host is able to access the Junos device without being prompted for a password. When first logging in, you may need to allow remote hosts to access the private key in the users SSH directory if you created a passphrase for the SSH key.

Connect Using an SSH Key Agent

- Using an SSH key agent with actively loaded keys

```
dev = Device(host=hostname, user=username)
```

- Note: The SSH key agent supplies the password protection passphrase

- You can omit the `user` if the user account is the same on both client and target

```
dev = Device(host=hostname)
```

Connect Using an SSH Key Agent

You can use an SSH key agent to securely store private keys and avoid repeatedly retying the passphrase for password-protected keys. If you do not provide a password or SSH key file in the arguments of the `Device` constructor, Junos PyEZ first checks the SSH keys that are actively loaded in the SSH key agent and then checks for SSH keys in the default location.

If you do not specify a `user` in the parameter list, the user defaults to the `$USER` variable, which holds the name of the current user. If you have a user account is the same on both the client and target, you can safely omit the `user` argument.

Connect Without Using an SSH Key Agent

- Omit the `passwd` parameter if your SSH key does not have a password protection passphrase

```
dev = Device(host=hostname, user=username, )
```

- The `passwd` parameter holds the passphrase when using SSH keys

```
dev = Device(host=hostname, user=username,  
passwd=passphrase)
```

- SSH keys that are not stored in the default location need the `ssh_private_key_file` parameter

```
dev = Device(host=hostname, user=username,  
passwd=passphrase,  
ssh_private_key_file='/home/user/.ssh/id_rsa_dc')
```

Connect Without Using an SSH Key Agent

As mentioned previously, when using SSH keys, the `passwd` argument holds the SSH key password protection passphrase rather than a login password. If you did not configure an SSH key passphrase when you generated the SSH keys, then you can safely omit the `passwd` argument.

Example Using SSH Keys

- Example using SSH keys without a key manager

```
from jnpr.junos import Device
from getpass import getpass

hostname = raw_input("Device hostname: ")
username = raw_input("Device username: ")
passphrase = getpass('Enter password for SSH private
key file: ')

# NETCONF session over SSH
dev = Device(host=hostname, user=username,
passwd=passphrase)

dev.open()
```

Example Using SSH Keys without a Key Manager

The example in the slide shows the code you could use to authenticate a user who has set up SSH keys, and a passphrase to protect the keys, but is not using a SSH key manager.

Catching dev.open() Errors

```

import sys           Used by sys.exit() below
from jnpr.junos import Device
from jnpr.junos.exception import ConnectError           Used to determine cause of PyEZ Connection Errors

dev = Device(host='vMX-1', user='lab', passwd='lab123')
try:                                         Exception Block
    dev.open()
except ConnectError as err:
    print ("Cannot connect to device: {}".format(err))
    sys.exit(1)

print(dev.facts)

dev.close()

```

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER Worldwide Education Services

www.juniper.net | 34

Catching dev.open() Errors

To review how Python catches errors: The `try` statement works as follows:

First, the `try` clause, which are the statements between the `try` and `except` keywords, is executed. If no exception occurs, the `except` clause is skipped and execution of the `try` statement is finished.

If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then, if its exception type matches the exception named after the `except` keyword, the `except` clause is executed. Then execution continues after the `try` statement.

If an exception occurs that does not match the exception named in the `except` clause, it is passed on to an outer `try` statement; if no handler is found, it is an unhandled exception and the execution stops with the message shown in the slide.

When the `dev.open()` statement is executed error-free, the `except` block is skipped. If there is a `ConnectError`, the `except` block will execute. In this case, the `except` block prints “Cannot connect to device:” along with the detected error. Following the `print` statement, the `sys.exit(1)` call exits the program with an error code of 1.

Adding error checking to your code is recommended. Much of the code you see in this text has had the error code omitted to allow the code fit onto the slides.

The `print(dev.facts)` statement prints the device statistics for the connected device. We will discuss more about `dev.facts` next.

Displaying Device Facts

```
from jnpr.junos import Device

dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()
print (dev.facts)
dev.close()
```

Script Output

```
{'2RE': False, 'HOME': '/var/home/lab', 'RE0': {'status': 'OK', 'last_reboot_reason': 'Router rebooted after a normal shutdown.', 'model': 'RE-VMX', 'up_time': '3 hours, 47 minutes, 39 seconds', 'mastership_state': 'master'}, 'RE1': None, 'RE_hw_mi': False, 'current_re': ['re0', 'master', 'node', 'fwdd', 'member', 'pfem'], 'domain': None, 'fqdn': 'vMX-1', 'hostname': 'vMX-1', 'hostname_info': {'re0': 'vMX-1'}, 'ifd_style': 'CLASSIC', 'junos_info': {'re0': {'text': '17.1R1.8', 'object': junos.version_info(major=(17, 1), type=R, minor=1, build=8)}}, 'master': 'RE0', 'model': 'VMX', 'model_info': {'re0': 'VMX'}, 'personality': 'MX', 're_info': {'default': {'default': {'status': 'OK', 'last_reboot_reason': 'Router rebooted after a normal shutdown.', 'model': 'RE-VMX', 'mastership_state': 'master'}, '0': {'status': 'OK', 'last_reboot_reason': 'Router rebooted after a normal shutdown.', 'model': 'RE-VMX', 'mastership_state': 'master'}}, 're_master': {'default': '0'}, 'serialnumber': 'VM58D97E4E26', 'srx_cluster': None, 'switch_style': 'BRIDGE_DOMAIN', 'vc_capable': False, 'vc_fabric': None, 'vc_master': None, 'vc_mode': None, 'version': '17.1R1.8', 'version_RE0': '17.1R1.8', 'version_RE1': None, 'version_info': junos.version_info(major=(17, 1), type=R, minor=1, build=8), 'virtual': True}

-----
(program exited with code: 0)
Press return to continue
```

Displaying Device Facts

When establishing a NETCONF session over SSH, Junos PyEZ automatically gathers facts from the device and returns them as a Python dictionary unless the `gather_facts` parameter is set to `False`. For Telnet and serial console connections, you must explicitly include the `gather_facts=True` argument in the `Device` argument list to gather device facts.

A description of all the device facts gathered is explained at: <https://junos-pyez.readthedocs.io/en/latest/jnpr.junos.facts.html>.

Displaying Device Facts - pprint

```
from jnpr.junos import Device
from pprint import pprint
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()
pprint(dev.facts)
dev.close()
```

Script Output

```
{'2RE': False,
'HOME': '/var/home/lab',
'RE0': {'last_reboot_reason': 'Router rebooted after a normal shutdown.',
'mastership_state': 'master',
'model': 'RE-VMX',
'status': 'OK',
'up_time': '3 hours, 50 minutes, 49 seconds'},
'RE1': None,
'RE_hw_mi': False,
'current_re': ['re0', 'master', 'node', 'fwdd', 'member', 'pfem'],
'domain': None,
'fqdn': 'vMX-1',
'hostname': 'vMX-1',
'hostname_info': {'re0': 'vMX-1'},
'ifd_style': 'CLASSIC',
... Trimmed ...}
```

Displaying Device Fact with PrettyPrint

You can make the display more readable by importing and using the PrettyPrint library. The slide shows the amended code as well as the output.

By default, Junos PyEZ returns the facts as a Python dictionary. To view a YAML or JSON representation of the facts, import the `yaml` or `json` module respectively, and call either the `yaml.dump()` or `json.dumps()` function, as appropriate. Note that the `yaml.dump()` or `json.dumps()` feature currently does not work on virtual devices. For more information, see the Junos PyEZ Developer Guide.

Agenda: Python and PyEZ

- Introduction to Python and PyEZ
- Python Development Environment
- Working with RPCs
- Working with an Unstructured Configuration
- Working with Tables and Views
- PyEZ Exception Handling

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 37

Working with RPCs

The slide highlights the topic we discuss next.

Using PyEZ To Call RPCs (1 of 4)

- Find the RPC using the CLI

```
lab@vMX-1> show interfaces terse | display xml rpc
<rpc-reply
  xmlns:junos="http://xml.juniper.net/junos/17.1R1/junos">
    <rpc>
      <get-interface-information>
        <terse/>
      </get-interface-information>
    </rpc>
    <cli>
      <bANNER></bANNER>
    </cli>
  </rpc-reply>
```

Find the RPC Using the CLI, Part 1

To execute an RPC using PyEZ, you first need to know the name of the RPC. You can find the name of the RPC quickly by using the `| display xml rpc` command at the end of a Junos OS command, as shown on the slide.

Using PyEZ To Call RPCs (2 of 4)

- Show the XML RPC using PyEZ

```
from jnpr.junos import Device
from lxml import etree
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()
print (dev.display_xml_rpc('show interfaces terse',
format='text'))
```

dev.close()

```
<get-interface-information>
  <terse/>
</get-interface-information>

-----
(program exited with code: 0)
Press return to continue
```

Show the XML RPC Using PyEZ, Part 2

If you are not near a CLI session, you can also use PyEZ to show the XML RPC, as in the slide above.

Using PyEZ To Call RPCs (3 of 4)

- The RPC call can be translated by changing hyphens to underscores and tags to arguments

These XML Tags:

```
<get-interface-information>
  <terse/>
  <interface-name>lo0</interface-name>
</get-interface-information>
```

Translate into:

```
dev.rpc.get_interface_information (interface_name='lo0',
terse=True)
```

Translate XML RPC to PyEZ, Part 3

Each instance of Device has an `rpc` property that enables you to execute any RPC available through the Junos XML API. In Junos PyEZ code, after establishing a connection with the device, you can execute the RPC by appending the `rpc` property and RPC method name to the Device instance.

You can map the request tags for an operational command to a Junos PyEZ RPC method name. To derive the RPC method name, replace any hyphens in the request tag with underscores (`_`) and remove the enclosing angle brackets. For example, the `<get-route-information>` request tag maps to the `get_route_information()` method name.

Using PyEZ To Call RPCs (4 of 4)

■ Example RPC Calls

```
from jnpr.junos import Device
from lxml import etree
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()
#1. Basic RPC call - Returns XML
op = dev.rpc.get_interface_information(dev_timeout = 60)
#2. Basic RPC call - Returns Text
op = dev.rpc.get_interface_information({'format':'text'})
#3. RPC Call with additional attributes
op = dev.rpc.get_interface_information (interface_name='lo0',
extensive=True)
print (etree.tostring(op))
#4. Display a specific piece of information using xpath
for i in op.xpath('.//link-level-type'):
    print i.text
dev.close()
```

Example RPC Calls, Part 4

The slide shows four example RPC calls. The first example illustrates a basic RPC call that will return information in XML. The second RPC call does the same RPC, but returns the information in text format. The third RPC specifies the `lo0` interface; note that the `extensive` option is specified. The fourth item in the slide uses XPath to specify a piece of information, which is the `link-level-type` to view for each interface.

This is a sample of the power of using PyEZ to retrieve Junos OS XML RPC data. To see all the possible information that may be retrieved through an RPC, consult the Junos OS XML API Guide.

Using an RPC to Retrieve the Configuration

- You can use an RPC to view the active configuration

```
from jnpr.junos import Device
from lxml import etree
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()

#1. View the whole configuration
cnf = dev.rpc.get_config(dev_timeout = 60)

#2. View partial configuration using XPath
cnf = dev.rpc.get_config(filter_xml=etree.XML
('<configuration><interfaces/></configuration>')) 

print etree.tostring(cnf)
dev.close()
```

Using an RPC to Retrieve the Junos OS Configuration

PyEZ RPCs are also useful to retrieve a copy of the active configuration. The slide shows the code for retrieving the whole configuration, and it also shows how you can use XPath and the lxml module to filter the results. The second example retrieves only the [edit interfaces] hierarchy of the configuration.

Notice in the first example that there is a `dev_timeout = 60` argument. By default, PyEZ gives 30 seconds for an RPC to execute before it times out. If you need more time, you can increase the time on individual RPCs by using the `dev_timeout` argument, as seen in the slide. You can also use `dev.timeout = 60` after the `dev.open()` statement to increase the timeout for the whole NETCONF session.

Normalizing the XML RPC Reply

- Normalizing RPC output removes excess whitespace
- Normalizing RPC output makes it easier to parse
- Three places to normalize

```
dev = Device(host='vMX-1', user='lab', passwd='lab123',
normalize=True)

dev.open(normalize=True)

op = dev.rpc.get_alarm_information(normalize=True)
```

Default RPC Reply

```
<alarm-information>
<alarm-summary>
<no-active-alarms/>
</alarm-summary>
</alarm-information>

-----
(program exited with code: 0)
Press return to continue
```

Normalized RPC Reply

```
<alarm-information><alarm-
summary><no-active-alarms/></alarm-
summary></alarm-information>

-----
(program exited with code: 0)
Press return to continue
```

Normalizing the XML RPC Reply

When you execute an RPC, the RPC reply can include data that is wrapped in new lines, or contains other superfluous whitespace. Unnecessary whitespace can make it difficult to parse the XML and find information using text-based searches. You can normalize an RPC reply, which strips out all leading and trailing whitespace and replaces sequences of internal whitespace characters with a single space. The slide compares a default RPC reply to the normalized version. The line-breaks between parent and child elements doesn't often create issues. Problems occur when you have unanticipated line-breaks or whitespace within the opening and closing tags of a single element. For example `<name>\nfxp0\n</name>` becomes `<name>fxp0</name>` once it is normalized.

You can enable normalization for the duration of a session with a device, or you can normalize an individual RPC reply when you execute the RPC. To enable normalization for a single RPC call, include `normalize=True` in the parameter list of the RPC. To enable normalization for the entire session, include `normalize=True` in the parameter list when you connect to the device using the `open()` method. Both methods are shown in the slide.

Agenda: Python and PyEZ

- Introduction to Python and PyEZ
- Python Development Environment
- Working with RPCs
- Working with an Unstructured Configuration
- Working with Tables and Views
- PyEZ Exception Handling

Working With and Unstructured Configuration

The slide highlights the topic we discuss next.

Unstructured Configuration Changes (1 of 3)

- You can use XML, JSON, Junos OS set commands, or ASCII text to make configuration changes
- Begin by creating a config object and associating it with the Device object

```
dev = Device(host='vMX-1', user='lab', passwd='lab123')
cu = Config(dev)
```

- Follow the process

1. Lock the configuration using `lock()`
2. Load the new configuration or configuration changes using `load()`, roll back the configuration using `rollback()`, or revert to a rescue configuration using `rescue()`
3. Commit the configuration using `commit()`
4. Unlock the configuration using `unlock()`

Unstructured Configuration Changes, Part 1

Unstructured configuration changes, which consist of static or templated configuration data that is formatted as ASCII text, Junos XML elements, Junos OS set commands, or JSON, are performed using the `jnpr.junos.utils.config.Config` utility.

When you use unstructured changes to modify the configuration of devices running Junos OS, you can change any portion of the configuration but you must use one of the accepted formats for the configuration data and the correct syntax for that format. Users who are familiar with the supported configuration formats and want the option to modify any portion of the configuration may favor this method for configuration changes.

Junos PyEZ enables you to make configuration changes on devices running Junos OS. After successfully connecting to the device, you create a Config object and associate it with the Device object, as shown on the slide.

The process for making configuration changes is to lock the configuration, load the configuration changes, commit the configuration to make it active, and then unlock the configuration database.

Unstructured Configuration Changes (2 of 3)

- Example using text data

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()
cu = Config(dev)
data = """interfaces {
    ge-0/0/1 {
        description "Test interface";
        unit 0 {
            family inet {
                address 172.17.1.150/24;
            }
        }
    }
}
"""
#Continued on the next slide
```

Unstructured Configuration Changes Using Text Data, Part 2

The slide shows an example of replacing an already configured ge-0/0/1 interface with an updated configuration and adding a description. Notice that the `Config` object has been imported from `jnpr.junos.utils.config`. After `dev.open()` is called, the `Config` object is associated with the `Device` instance `dev` and called `cu`. Next, the text data is stored in a `data` variable as a string.

Unstructured Configuration Changes (3 of 3)

```
#Continued from previous slide
cu.lock()
cu.load(data, format='text')
cu.pdiff()
if cu.commit_check():
    cu.commit()
else:
    cu.rollback()
cu.unlock()
dev.close()
```

Script Output

```
[edit interfaces ge-0/0/1]
+    description "Test interface";
[edit interfaces ge-0/0/1 unit 0 family inet]
        address 172.17.1.100/24 { ... }
+
        address 172.17.1.150/24;

-----
(program exited with code: 0)
Press return to continue
```

Unstructured Configuration Changes Using Text Data, Part 3

The `load()` method uses the configuration stored in the `data` variable and the `format='text'` argument to load the data.

The `format` argument could be `text`, `set`, `json`, or `xml`, depending upon the format of the `data` variable. If you do not specify a `format`, the default is `xml`.

The `cu.pdiff()` method prints the difference between the current configuration and the updated configuration to the screen. The `if` statement performs a commit check, and if the commit check returns true, then the configuration is committed. Otherwise, the configuration is rolled back. After the commit operation, the database is automatically unlocked.

NOTE: You do not need to enclose configuration data that is formatted as ASCII text, Junos OS set commands, or JSON in `<configuration-text>`, `<configuration-set>`, or `<configuration-json>` tags, as is required when configuring the device directly within a NETCONF session.

Using set Commands to Change the Configuration

```

from jnpr.junos import Device
from jnpr.junos.utils.config import Config
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()

with Config(dev, mode='private') as cu:
    data='set system services netconf traceoptions file
test.log'
    cu.load(data, format='set', merge=False, overwrite=False)
    cu.pdiff()
    cu.commit()

dev.close()

```

Script Output

```

[edit system services netconf]
+     traceoptions {
+         file test.log;
+     }

-----
(program exited with code: 0)
Press return to continue

```

Using set Commands to Change the Configuration

The script in the slide illustrates how to use set commands to modify the configuration. Notice that this time, the `with` statement is used to associate the `Config` and `Device` objects, and that the `cu.load()`, `cu.pdiff()`, and `cu.commit()` methods are all indented and lie within the block. This has the same effect as the previous example with the added benefit of additional safety measures to insure that an error will not leave your device with a locked database. For more information on the `with` statement, consult the Python documentation. The use of `with` is required whenever you specify a mode.

Line 5 includes the `mode='private'` argument. This instructs PyEZ to open the configuration database in private mode. You can specify `exclusive`, `dynamic`, or `batch` mode. The default mode is `none`.

The example turns on NETCONF trace options and stores output in the file `test.log`. The data can contain many `set` statements. For brevity, the example shows just one.

The `cu.load()` method also contains the optional `merge` and `overwrite` arguments. These arguments are used together to specify the load type. If you want to have the effect of a load merge operation, use `merge=True` and `overwrite=False`. If you want the effect of a load overwrite operation, use `merge=False` and `overwrite=True`. If you want the effect of a load replace command, set `merge=False` and `overwrite=False`. If you omit them both, the load method defaults to load replace.

NOTE: When the `overwrite` parameter is set to `True`, you cannot use the Junos OS `set` command format.

The output of the script is shown on the slide.

Load Configuration From a File

```
from jnpr.junos import Device
from jnpr.junos.utils.config import Config
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()

with Config(dev, mode='private') as cu:
    conf_file = "test.conf"
    cu.load(conf_file, merge=True)
    cu.pdiff()
    cu.commit()

dev.close()
```

ScriptOutput

```
[edit routing-options static]
    route 0.0.0.0/0 { ... }
+    route 172.31.2.0/24 next-hop 172.25.11.254;

-----
(program exited with code: 0)
Press return to continue
```

Load Configuration From a File

In this example, the configuration is loaded from a file. The test.conf file is located in the same directory as the script. The file format argument is not used in this example on the slide. If you use a PyEZ-recognizable file format extention, you don't need to specify a format argument. Text files can have a .conf, .text, or .txt extension. JSON data needs to have a .json extension. Junos OS set command files need to have a .set extension. Finally, XML files need to have a .xml file extension.

The slide shows the output of the script.

Installing Software

```

from jnpr.junos import Device
from jnpr.junos.utils.sw import SW

dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()

def update_progress(dev, report):
    print dev.hostname, '>', report

sw = SW(dev)
ok = sw.install(package=r'/home/lab/Downloads/jinstall-
1x.1xxxx.tgz', progress=update_progress)
if ok:
    print 'rebooting'
    sw.reboot()
dev.close()

```

Installing Software

PyEZ can be used to upgrade your Junos OS device. To upgrade software, import the `SW` class from the `jnpr.junos.utils.sw` library.

The `def update_progress(dev, report)` is a function prototype method that allows you to customize progress reports supplied by the Juniper device. In this case, the progress message is customized so that each message is prefaced with the device hostname and a “>” sign.

After the `def` statement, the `SW` object is associated with the `dev` object and assigned the name “`sw`”. Next, the `install()` method is called. The `install()` method performs the complete installation of the package, which includes the following steps:

1. Computes the local MD5 checksum if not provided in `:checksum`:
2. Performs a storage cleanup if `:cleansfs` is True
3. SCP copies the package to the `:remote_path` directory
4. Computes remote MD5 checksum and matches it to the local value
5. Validates the package if `:validate` is True
6. Installs the package

The example specifies where the package tar file is located on the file system and that the device should give progress updates. There are many other optional arguments that can be found at <http://junos-pyez.readthedocs.io/en/latest/>.

Agenda: Python and PyEZ

- Introduction to Python and PyEZ
- Python Development Environment
- Working with RPCs
- Working with an Unstructured Configuration
- Working with Tables and Views
- PyEZ Exception Handling

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 51

Working With Tables and Views

The slide highlights the topic we discuss next.

PyEZ Tables and Views

- PyEZ tables and views provide a structured way to view RPC and configuration data
 - Tables hold data for an RPC or configuration
 - Views or a subset or specific fields of the table
- Tables and views are defined using YAML files
- PyEZ has many predefined tables and views included with the PyEZ installation
- You can define your own tables and views

PyEZ Tables and Views

Tables and views make working with RPC and configuration data more approachable for non-networkers. It enables the Junos OS RPC data and configuration data to be put into a structured table-like format with which programmers are more familiar. A table holds all of the retrieved data from an RPC or configuration. A view holds a limited view or a subset of the table data.

Tables and views are defined using YAML files. You'll see a sample on the next page. Each table comes with at least one view and sometimes more than one. As PyEZ tables are defined using YAML, you can define your own table and views if you don't find what you are looking for in the existing ones.

PyEZ Table and View Files

ArpTable:

```

rpc: get-arp-table-information
item: arp-table-entry
key: mac-address
view: ArpView

```

ArpView:

```

fields:
  mac_address: mac-address
  ip_address: ip-address
  interface_name: interface-name

```

PyEZ Table and View Files

The slide shows the py-junos-eznc-master\lib\jnpr\junos\op\arp.yml file found in the PyEZ installation directory. The name of the table is ArpTable. We discussed YAML syntax in a previous chapter, so this topic won't be covered here.

The ArpTable has a YAML `rpc` mapping that holds the Junos OS RPC name. The `item` mapping specifies that the table holds the `arp-table-entry` values found in the `get-arp-table-information` RPC. The `key` is a primary key in a database and the unique key that serves to differentiate each entry. The last mapping is `view` and specifies the name of the related view. The ArpView mapping is listed beneath the ArpTable mapping and contains three fields. If we look at the YANG model for the `get-arp-table-information` RPC that is in the `show-arp.yang` file, shown on the next page, we can see that the `arp-table-entry` container has only the same three leaf nodes defined. At the bottom, there is also an `arp-table-entry-flags` container. These leaf nodes are not part of the defined view. So although the `arp-table-entry` container has many leaf nodes, the ArpView only included three. If you want to include some of the additional values, you could modify the existing ArpView or build your own.

Continued on the next page.

PyEZ Table and View Files (contd.)

```
container arp-table-entry {
    description "Information about an entry in the ARP table";
    leaf mac-address {
        type mac-addr;
        description "Hardware MAC address associated with the ARP entry";
    }
    leaf ip-address {
        type ipaddr;
        description "IP address of the ARP entry";
    }
    leaf hostname {
        type string;
        description "Resolved hostname of the ARP entry";
    }
    leaf interface-name {
        type string;
        description "Name of the interface associated with the ARP entry";
    }
    container arp-table-entry-flags {
        description "One or more status flags for the ARP entry";
        leaf none {
            type empty;
            description "ARP entry has no flags";
        }
        leaf permanent {
            type empty;
            description "ARP entry is permanent";
        }
        leaf published {
            type empty;
            description "ARP entry is published";
        }
        leaf dead {
            type empty;
            description "ARP entry is dead";
        }
    }
}
```

Loading a Table and View (1 of 2)

1. Import the table

```
from jnpr.junos.op.arp import ArpTable
```

2. Associate ArpTable with the Device

```
arp = ArpTable(dev)
```

3. Retrieve items

```
arp.get()
```

4. Output results

```
for mac in arp:
    print ("{}: {}  {}".format(mac.mac_address,
mac.ip_address, mac.interface_name))
```

Loading a Table and View, Part 1

The first step in loading a PyEZ table and view is to import the table from the `junpr.junos.op.(module)` library. This table name and module may seem confusing at first. Each of the op modules and their corresponding table names, along with the associated JUNOS RPC and CLI command, are listed here: https://www.juniper.net/techpubs/en_US/junos-pyez2.0/topics/reference/general/junos-pyez-tables-op-predefined.html.

After importing the table, it is associated with the Device `dev` and given the name `arp`. Next, the `get()` method retrieves the information. Steps 2 and 3 can be combined into a single statement: `arp = ArpTable(dev).get()`

Last, the results can be outputted, as shown in the slide. You can choose which fields you want to output. On the previous page, notice that the field names have had the hyphens converted to underscores. Field names must conform to Python naming conventions for variables, so cannot contain hyphens.

Loading a Table and View (2 of 2)

```
from jnpr.junos import Device
from jnpr.junos.op.arp import ArpTable
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()
arp = ArpTable(dev).get()

for mac in arp:
    print ("{}: {}  {}".format(mac.mac_address, mac
.ip_address, mac.interface_name))
```

```
dev.close()
```

Script Output

```
00:50:56:a9:61:dd: 128.0.0.16  em1.0
00:50:56:a9:5f:58: 172.25.11.254  fxp0.0
```

```
-----
```

```
(program exited with code: 0)
Press return to continue
```

Loading a Table and View, Part 2

The slide shows the complete script code and output. Notice that there is a colon between the MAC address and the IP address, as specified in the print statement.

Tables and Views to Access Junos Configuration

- PyEZ tables and views provide structured access to the Junos OS configuration data
- Access can be read-only using get property
- Access can be read-write using the set property

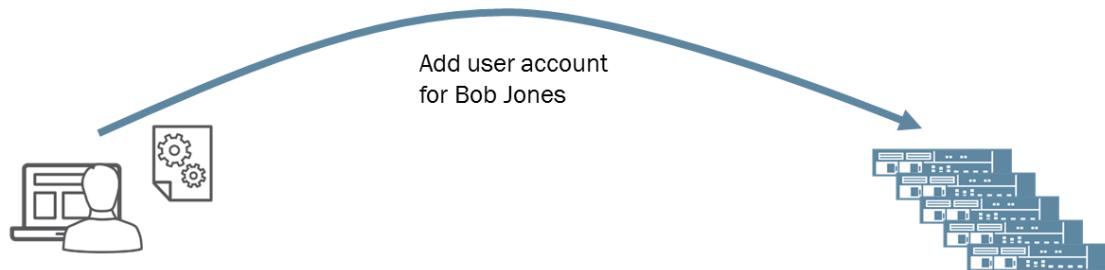
Using Tables and Views to Access the Junos OS Configuration

In addition to using PyEZ tables and views to access the Junos XML RPCs, you can also use PyEZ tables and views to provide structured access to the Junos OS configuration data. When accessing the configuration through tables and views, you can access it in a read-only manner using the YAML get mapping. You can also access the configuration in a read-write manner using the set mapping. You'll see how that is done next.

Modifying the Configuration Using Tables (1 of 7)

▪ Case Study Scenario

- Create a Python script using PyEZ that allows a user who doesn't know the Junos OS to add a user account to a device running the Junos OS



Modifying the Configuration Using Tables, Part 1

In this scenario, we will create a PyEZ script that will be used by a non-Junos OS user who needs to add user accounts to a device running the Junos OS.

During this process you will see how to use structured configurations and how to create your own tables using YAML.

Modifying the Configuration Using Tables (2 of 7)

- Identify the Junos OS configuration hierarchy level

```
lab@vMX-1> show configuration system login | display xml
<rpc-reply>
  <configuration>
    <system>
      <login>
        <user>
          <name>lab</name>
          <uid>2000</uid>
          <class>super-user</class>
          <authentication>
            <encrypted-password>$1kj3.</encrypted-password>
          </authentication>
        </user>
      </login>
    </system>
  </configuration>
  . . . Trimmed . . .
</rpc-reply>
```

This is the Hierarchy we need to add a new user

Modifying the Configuration Using Tables, Part 2

In this case study, you won't need the entire configuration. For the next step, you will need an XPath expression that identifies which part of the configuration will be your focus. The easiest way to build the XPath expression is to issue the command shown on the slide. The XPath expression you will need is: `system/login/user`. The configuration element is not necessary.

Modifying the Configuration Using Tables (3 of 7)

- Create table definitions for the structured resource

UserAccountTable:

```
set: system/login/user
key-field: username
view: UserAccountView
```

Modifying the Configuration Using Tables, Part 3

The table definitions are created using YAML. You define the table name. The first YAML mapping under the table name needs to be either a `set:` mapping or a `get:` mapping. If the table is going to enable users to change the configuration, it needs to be a `set:` mapping. The value for the `set:` mapping is the an XPath path to the relevant configuration hierarchy level. Be sure not to include a final forward slash at the end of the path, or it will generate an error.

Each set table also needs a `key-field` mapping that lists the field, at that hierarchy level, that uniquely identifies the resources. Sometimes the key field may contain more than one mapping. For example, a logical interfaces requires both the `interface-name` and the `unit-number` to uniquely identify the logical interface. In the example above, the `key-field` is `username`.

The `view` mapping identifies the `view` linked to the table. We will create the `UserAccountView` next.

Modifying the Configuration Using Tables (4 of 7)

- Create the view definitions.

```
UserAccountView:
  groups:
    auth: authentication
  fields:
    username: name
    userclass:
      class:
        default: unauthorized
    uid:
      uid:
        type: int
        default: 1001
        minValue: 100
        maxValue: 64000
    fullname: full-name
  fields_auth:
    password: encrypted-password
```

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 61

Create the View Definitions, Part 4

You define the view name, but it must match the one you defined for the view in the table. Before explaining the UserAccountView, recall that you are going to be creating a configuration that stores data in the system/login/user hierarchy. As a reference, here is the XML configuration snippet:

```
<user>
  <name>bob</name>
  <uid>1001</uid>
  <class>super-user</class>
  <authentication>
    <encrypted-password>lab123</encrypted-password>
  </authentication>
</user>
```

The UserAccountView enables you to associate variable names that you use in your Python script with XML tags, within which, data will be stored. The view also lets you specify data types, default values, and constraints.

Let's start by looking at the fields mapping. The `username` maps to the XML tag `<name>`.

The `userclass` is stored in `<class>` tag. If there is no value specified, it defaults to `unauthorized`.

The `uid` is stored in the `uid` element. The `uid` also specifies a data type of `int` and a default value of `1001`. There are also constraints specified by the `minValue` and `maxValue` mappings.

Continued on the next page.

Create the View Definitions, Part 4 (contd.)

The `fullname` is stored in the `<full-name>` element.

The `password` mapping is a bit more complicated. Notice that the `<encrypted-password>` tag is a child tag of the `<authentication>` tag. To ensure that the password is put in the right place, notice the `groups:` mapping at the top of the `UserAccessView`. The `groups` mapping contains the `auth:` mapping with a value of `authentication`. Any fields listed in `Fields_auth` at the bottom of the `UserAccountView` are placed within the `authentication` hierarchy as specified in the `groups:` mapping. Simply said, values assigned to the `password` variable are put into the `<encrypted-password>` element, which is nested within the `<authentication>` tag.

As an alternative to using the method above for mapping the `<encrypted-password>` you could instead have an mapping of `password: authentication/encrypted-password`.

Modifying the Configuration Using Tables (5 of 7)

- Save your files

1. Create a folder called “myTables” in your project folder to store your table definitions
2. Save the file with your UserAccountTable and UserAccountView as configTables.yml in myTables folder
3. Create a file called configTables.py in the myTables folder with the following four lines of code

```
from jnpr.junos.factory import loadyaml
from os.path import splitext
__YAML__ = splitext(__file__)[0] + '.yml'
globals().update(loadyaml(__YAML__))
```

4. Create a blank file called __init__.py and save it to the myTables folder

Save the Files, Part 5

Inside your project folder you will need a separate folder to store the module you just created. You can give the folder whatever valid folder name you like. The example uses the folder name myTables.

Next, save the YAML file that contains the UserAccountTable and UserAccountView. To avoid confusion, you will want to give it a name that is different than the table name. The example uses configTable.yml for the file name.

In order for the configTable.yml file to be loaded in as a module, you need a python file by the same name as the YAML file with the following four lines of code:

```
from jnpr.junos.factory import loadyaml
from os.path import splitext
__YAML__ = splitext(__file__)[0] + '.yml'
globals().update(loadyaml(__YAML__))
```

Finally, because the myTables folder is a sub folder to your main Python project folder, you will need to create a blank file with the name __init__.py. This tells the Python interpreter to look into this folder for modules.

Modifying the Configuration Using Tables (6 of 7)

- Write the script

```
from jnpr.junos import Device
from myTables.configTables import UserAccountTable
dev = Device(host='vMX-1', user='lab', passwd='lab123')
dev.open()

ua = UserAccountTable(dev)
ua.username = 'bob'
ua.userclass = 'super-user'
ua.password = 'lab123'
ua.append()
ua.set(merge=True, comment="Junos PyEZ commit")

#ua.lock()
#ua.load(merge=True)
#ua.commit(comment="Junos PyEZ commit")
#ua.unlock()
dev.close()
```

Write the Script, Part 6

The first thing the script does is to import the needed modules. You have seen the `Device` module before. The `UserAccountTable` is the table you saved on the previous page. Notice that it is from `myTables.configTables`. This is the path to the `configTables` module which holds the `UserAccountTable`.

After the `dev.open()` statement, the `UserAccountTable` is associated with the `Device` object `dev`. Next the `username`, `userclass`, and `password` information is collected. Note that the password is not encrypted. There are Python password libraries that can help you to encrypt. Ideally, you would ask the user to input this information when the script is run.

Each call to the `append()` method generates the Junos XML configuration data for the current set of changes and adds it to the `Ixml` object, which stores the master set of configuration changes. The `ua.set()` method unlocks the database, loads the amended configuration into the database, commits the change, and then unlocks the database.

The bottom of the script shows an alternative to using the `set()` method and enables you to have more control by performing the lock, load, commit, and unlock methods separately. The advantage to performing the methods separately is that it enables you to add error checking at each step.

Modifying the Configuration Using Tables (7 of 7)

- Test the result

```
from jnpr.junos import Device
from myTables.configTables import UserAccountTable
dev = Device(host='vMX-1', user='lab', passwd='lab123')
ua = UserAccountTable(dev)
dev.open()

ua.get()
for account in ua:
    print("Username is {}\\nUser class is
{}".format(account.username, account.userclass))

dev.close()
```

Script Output

```
Username is bob
Userclass is super-user
Username is lab
Userclass is super-user
-----
(program exited with code: 0)
Press return to continue
```

Test the Result, Part 7

To see if your script worked correctly, you can modify the previous script as seen in the slide. The `ua.get()` pulls a fresh copy of the configuration data. The `for` loop then prints out information about all the user accounts in the system.

The bottom of the slide shows the script output.

Agenda: Python and PyEZ

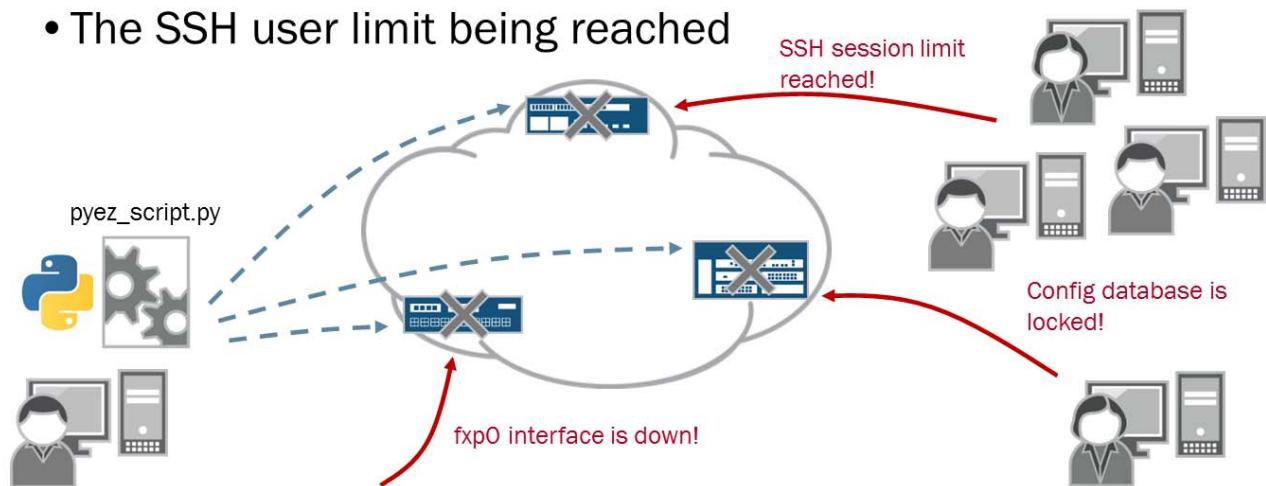
- Introduction to Python and PyEZ
 - Python Development Environment
 - Working with RPCs
 - Working with an Unstructured Configuration
 - Working with Tables and Views
- PyEZ Exception Handling

PyEZ Exception Handling

The slide highlights the topic we discuss next.

Introduction to Exception Handling

- Exception handling – important to protect against these scenarios:
 - Hosts becoming unreachable periodically
 - Users locking the configuration database
 - The SSH user limit being reached



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 67

Introduction to Exception Handling

Exceptions are errors that occur during run-time. Issues outside of the program may occur, such as a device's SSH user limit being reached. Robust code requires accurate exception handling. Note that exceptions do not necessarily mean that programs must terminate. For example, if the configuration database is locked, you may want the program to retry an operation after waiting for a period of time.

PyEZ try/except Statement

- The Python try/except statement allows the application/script to deal with exceptions

```

from jnpr.junos import Device
from jnpr.junos.exception import *
from jnpr.junos.utils.config import Config
import sys

try:
    dev = Device(host='192.168.64.51', user='bunk',
    password='bunk')
    dev.open()
    dev.close()
    print "Logged in!"
except Exception as error:
    if type(error) is ConnectAuthError:
        print "Authentication error!"
        sys.exit()
    else:
        print "Encountered an exception!"
        print error
        sys.exit()

```

try statement

except statement

Bad username/password

PyEZ try/except Statement

Python leverages the try/except statement for exception handling. In this scenario, Python will first attempt to execute the operations within the try statement. If no exceptions are caught, the operations complete and the program continues. However, if the code does not complete and exceptions are thrown, the code within the except statement is executed.

PyEZ try/except Example

- The Python try/except statement allows the application/script to deal with exceptions

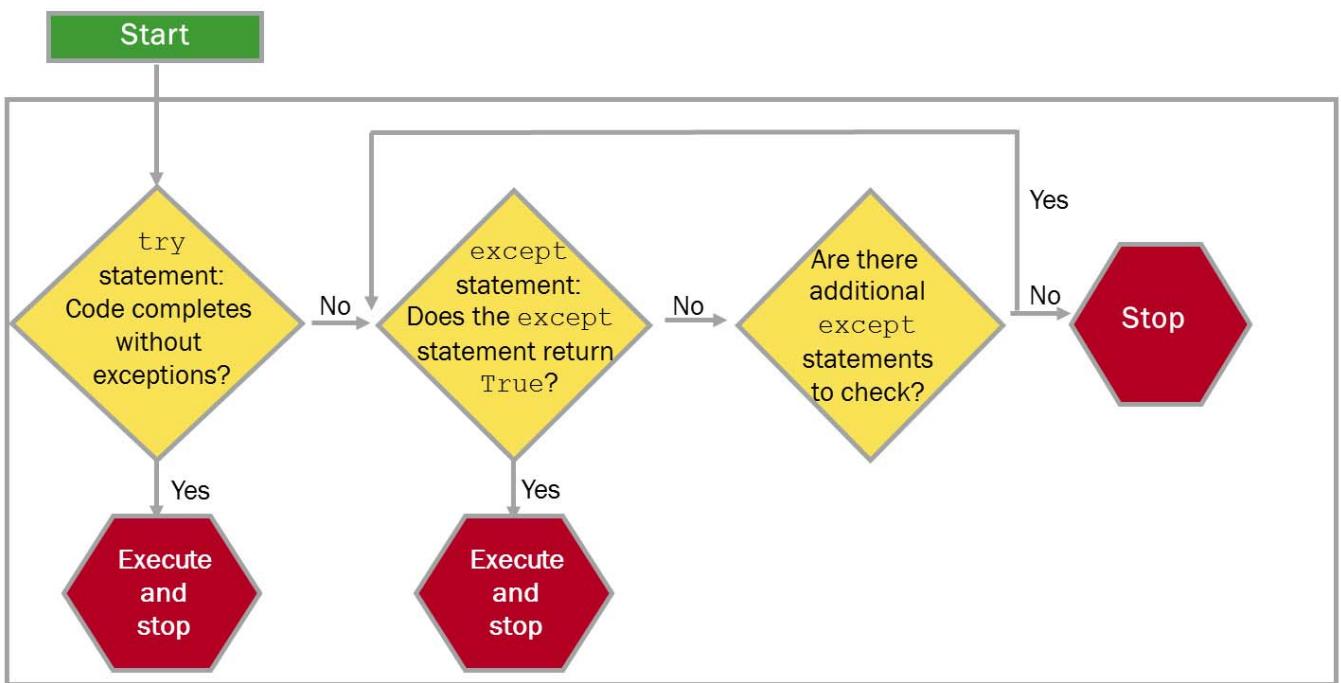
```
bash $ python pyez_exception_handling_1.py
Authentication error!
bash $
```

PyEZ try/except Example

In the example on the previous page, we attempted to authenticate using unknown credentials. As a result, the PyEZ module generated an exception. Our except statement uses an if/else to check the object type of the exception in question. Since the exception is an authentication error, the code prints the appropriate output.

PyEZ try/except Statement Flowchart

- The try/except statement processing:



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 70

PyEZ try/exception Statement Flowchart

When you have a series of try/except statements, the interpreter runs the code within the try first. If the code executes without issues, the program continues past the try/except. If exceptions are caught, then Python executes the code within the relevant except statement.

Note for Python experts: the try statement also allows for an else statement (not shown) which allows code to execute should the try statement complete without errors. Also, after all except statements, a finally may also be leveraged. All code within a finally statement is executed, whether an exception has occurred or not. Using a finally requires deep knowledge of each exception since the code executes for all exceptions.

Troubleshooting PyEZ

- There are a wide array of exceptions that PyEZ catches. Consult the PyEZ API Docs for details

```

try:
    dev = Device(host='192.168.64.51', user='root', passwd='root123')
    dev.open()
    config_change = Config(dev)
    config_change.load( path="bgp-config.conf", merge=True )
    config_change.commit()
    dev.close()
    print "Configuration applied!"
except ConnectAuthError:
    print "Authentication error!"
except ConnectTimeoutError:
    print "Timeout error!"
except ConfigLoadError:
    print "Couldn't unlock the config database!"
    dev.close()
    sys.exit()
except Exception as error:
    print "There are a lot more exceptions to cover!"
    print error
    sys.exit()

```

There are number of different Exception objects that are returned depending on the type of error.

When operating at scale, use exceptions to clean up broken network connections, open files, etc.

Troubleshooting PyEZ

Exception handling is critical for robust applications. Depending on the nature of the exception, we may want to perform different actions. The PyEZ package has the ability to throw a number of different exceptions depending on the result of a particular operation. Note that we are taking a more elegant approach to identifying different types of exceptions. We can reference the object type directly within our except statements. Also, use except statements to clean up broken network sessions or open files as needed, especially when operating at a large scale, to avoid consuming unnecessary resources.

Summary

- In this content, we:

- Built a Python environment to manage devices running the Junos OS
- Retrieved information from Junos OS RPCs using PyEZ scripts
- Modified the Junos OS configuration using PyEZ
- Used PyEZ tables and views to execute RPCs, view RPC data, and modify the Junos OS Configuration
- Used the PyEZ exception handling modules

We Discussed:

- Building a Python development environment and connecting to devices running the Junos OS;
- Utilizing PyEZ scripts to retrieve operational information from devices running Junos OS;
- Modifying the configuration on devices running the Junos OS;
- Utilizing PyEZ tables and views to extract operational information from RPCs and modify the Junos OS configuration; and
- Utilizing PyEZ exception handling modules.

Review Questions

1. What is the purpose of a Python virtualenv?
2. What three ways can PyEZ connect to devices running the Junos OS?
3. What is the shebang !# used for?
4. What does it mean to normalize the output?
5. What do you have to do to the RPC get-interface-information so that it can be called by PyE?

Review Questions

- 1.
- 2.
- 3.
- 4.
- 5.

Lab: Creating Python and PyEZ Scripts

- Use PyEZ to retrieve RPC information.
- Use PyEZ to modify the configuration.
- Use PyEZ tables and views.

Lab: Creating Python and Junos PyEZ Scripts

This slide provides the objective for this lab.

Review Questions

1.

The Python Virtualenv is used to create an isolated development environment.

2.

Junos PyEZ can connect to devices using a console connection, a telnet connection, and a NETCONF SSH connection.

3.

The !# is put at the beginning of a python file and enables a python script to be executed by just calling the name of the file.

4.

Normalizing the output removes leading and trailing whitespace.

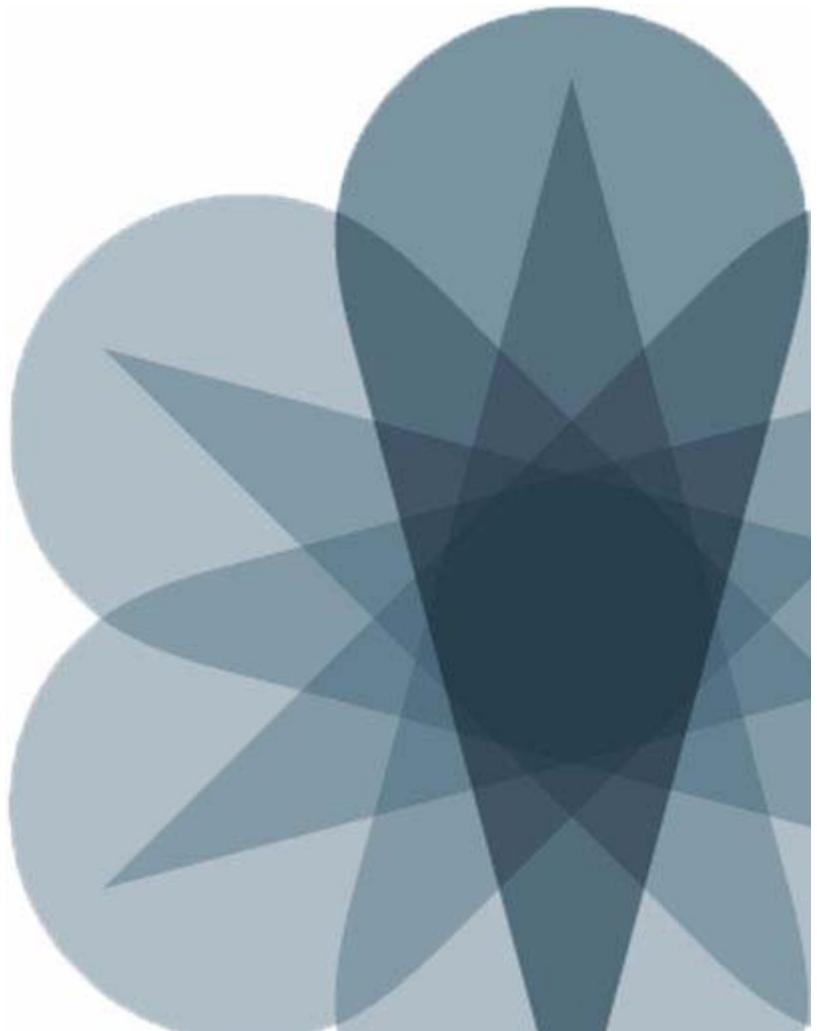
5.

In order for a get-interface-information RPC to be called by PyEZ the RPC has to be converted by changing the hyphens to underscores (i.e., get_interface_information).



Junos Platform Automation and DevOps

Chapter 6: Jinja2 and Junos PyEZ



Objectives

- After successfully completing this content, you will be able to:
 - Understand how Jinja2 can help you automate the Junos OS
 - Understand the syntax of Jinja2
 - Create Jinja2 templates to automate a Junos OS configuration

We Will Discuss:

- How Jinja2 can help you automate the Junos OS;
- The Jinja2 Syntax; and
- How Jinja2 is used to help create a Junos OS automation solution.

Agenda: Jinja2 and Junos PyEZ

→Jinja2 Overview

- Jinja2 Syntax
- Creating a Junos PyEZ, YAML, and Jinja2 Solution

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

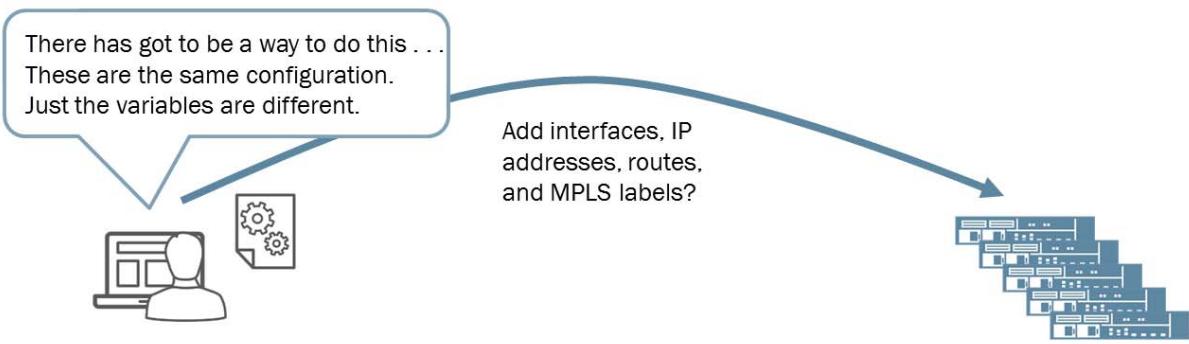
www.juniper.net | 3

Jinja2 Overview

The slide lists the topics we will discuss. We discuss the highlighted topic first.

The Challenge

- How do I automate multiple devices?
 - What if the IP addresses, OSPF areas, or MPLS labels are different on each device?



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 4

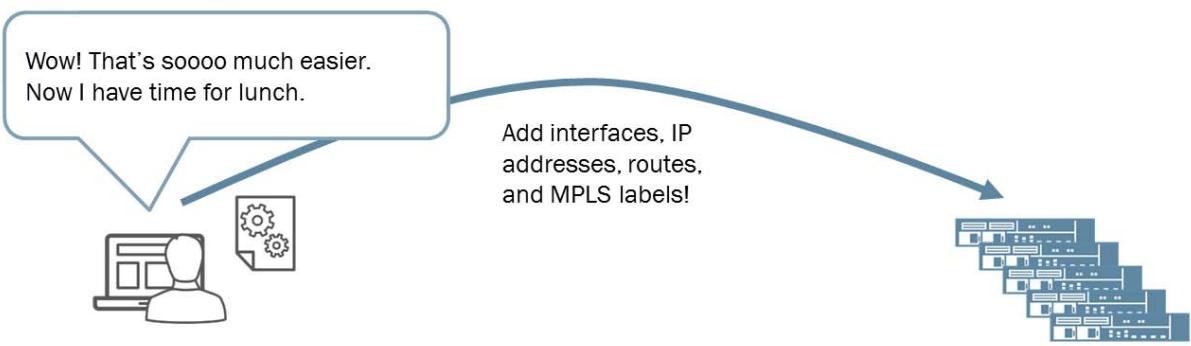
The Challenge

In the chapter on Junos PyEZ, we learned how to use automation to deploy a Junos OS configuration. If you have a variety of similar devices that must be deployed, you can modify the script to customize to each individual device. The challenge is to find a time-saving solution that enables you to run the script once to configure all of your devices simultaneously.

A similar challenge involves switches; if you have a single switch with many ports and you need to configure each port, how can you accomplish the same basic configuration to each port succinctly, without creating a lengthy script?

The Solution: Jinja2, YAML, and Junos PyEZ

1. Turn your Junos OS configuration into a Jinja2 template
2. Put interface IDs, IP addresses, and route information into a YAML data file
3. Use Junos PyEZ to merge the two files and load the configurations



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 5

The Solution - Jinja2, YAML, and Junos PyEZ

Turn your Junos OS configuration into a template using Jinja2. Put the unique parameter data for each device into a YAML file. Finally, use Junos PyEZ to load the Jinja2 template, merge it with the data stored in the YAML file, then deploy the configurations out to each device. You will walk through the process of creating a Junos OS template with Jinja2 after reviewing syntax.

Agenda: Jinja2 and Junos PyEZ

- Jinja2 Overview
- Jinja2 Syntax
- Creating a Junos PyEZ, YAML, and Jinja2 Solution

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 6

Jinja2 Syntax

The slide highlights the topic we discuss next.

Jinja2 Overview

- Jinja2 can turn any text file into a template
- Jinja2 is the standard for templating in Python
- Jinja2 documentation:
<http://jinja.pocoo.org/docs/2.9/>
- To install Jinja2:
`pip install jinja2`

Jinja2 Overview

To review so far, Junos PyEZ supports using Jinja2 templates to render Junos OS configuration data. Jinja2 is a template engine for Python that enables you to generate documents from predefined templates. The templates, which are text files in the desired language, provide flexibility through the use of expressions and variables.

You can create Junos OS configuration data using Jinja2 templates in one of the supported configuration formats, including JSON, ASCII text, Junos XML elements, and Junos OS `set` commands. Junos PyEZ uses the Jinja2 template and a supplied dictionary of variables to render the configuration data.

Jinja2 templates provide a powerful method to generate configuration data, particularly for similar configuration stanzas. For example, rather than manually adding the same configuration statements for each interface on a device, you can create a template that iterates over a list of interfaces and creates the required configuration statements for each one.

Jinja2 is the templating language standard for Python, and Python is designed to work with Jinja2. The Jinja2 documentation and URL are listed in the slide. Once your Python environment is functioning, you can install Jinja2 from the Python interpreter using the `pip install jinja2` command.

Variable Expansion

- To expand or replace a section of code, use double curly brackets {{ }}

```
interfaces {
    {{interface_ID}} {
        unit {{unit_num}} {
            family inet {
                address {{inf_address}};
            }
        }
    }
}
```

Example.yml

```
---
interface_ID: ge-0/0/0
unit_num: 0
inf_address: 172.25.11.1
```

Variable Expansion

The most fundamental task of the Jinja2 templating language is to pull variables from a data file, then insert these into the Jinja2 template. This task is called variable replacement, or variable expansion. Jinja2 identifies the variables by using the double curly brackets. The brackets indicate that corresponding data to fill the field must be pulled from a data file with a corresponding label.

Jinja2 if Statements

- Conditionally importing templates
- Determining if variables are defined

```
{% if install == 'new' %}
{% include 'new_install.conf' %}
{% else %}
{% include 'merge_install.conf' %}
{% endif %}
```

Jinja2 if Statements

Jinja2 supports conditional statements, such as `if`, `else`, `elif` statements. Conditional statements begin with a curly opening bracket then a percent sign, and close with the percent sign followed by a closing curly bracket, as shown in the slide.

Two common places where `if` statements are used with Junos automation are in conditionally importing a template. Conditionally importing templates allows you to introduce sub-templates, within given parameters, to meet your requirements. A second use is to check if a variable has a value. If there is no value defined, you can opt to supply a default value.

IF structures contain one `if` statement and may contain multiple `else` statements. The structure must close with the `ENDIF` statement, as seen in the slide.

Jinja2 for Loops

▪ Repeat tasks

```
ethernet-switching-options {
    analyzer multi-session {
        input {
            ingress {
                {%- for iface in host_ports %}
                    interface {{ iface }}.0;
                {%- endfor %}
            }
            egress {
                {%- for iface in host_ports %}
                    interface {{ iface }}.0;
                {%- endfor %}
            }
        }
    ... Trimmed ...
}
```

Script Output

```
ethernet-switching-options {
    analyzer multi-session {
        input {
            ingress {
                interface ge-0/0/1.0;
                interface ge-0/0/2.0;
                interface ge-0/0/3.0;
                interface ge-0/0/4.0;
                interface ge-0/0/5.0;
            }
        egress {
            interface ge-0/0/1.0;
            interface ge-0/0/2.0;
            interface ge-0/0/3.0;
            interface ge-0/0/4.0;
            interface ge-0/0/5.0;
        }
    }
},
```

Example.yml

```
---
host_ports:
  - ge-0/0/1
  - ge-0/0/2
  - ge-0/0/3
  - ge-0/0/4
  - ge-0/0/5
```

Jinja2 for Loops

A `for` loop opens and closes using the same curly bracket and percent sign pattern as the `IF` statement. In the example, you configure the ingress and egress interfaces for a switch. Each use of the `for` loop in this example configures five variables, as seen in the script output on the right.

Jinja2 Comments

- Use {# . . . #} to enclose comments

```
{# This code not needed, keep for reference  
  % for user in users %}  
  ...  
  % endfor %}  
#}
```

Jinja2 Comments

In Jinja2, the comment must be fully enclosed in curly brackets and pound signs, as illustrated in the slide. There isn't a single-line-comment feature in Jinja2 as is common in C, C++, Java, and Java-related languages.

A popular use for comments is to enclose a piece of code that is temporarily not being used in the program. The {# and #} symbols render the code on the slide inactive.

Jinja2 include Directive

- Use include statements to pull in other Jinja2 templates

- Example 1:

```
{% if install == 'new' %}  
  {% include 'new_install.conf' %}  
  {% else %}  
    {% include 'merge_install.conf' %}  
  {% endif %}
```

- Example 2:

```
{% include 'merge_install.conf' ignore missing %}
```

Jinja2 include Directive

The first example on the slide is the same code you viewed when discussing `if` statements. This time, you are using the code to illustrate the `include` directive; this tool can be used to bring in other Jinja2 templates. Included Jinja2 templates have access to the variables of the parent template.

The second example shown on the slide is also an `include` statement but contains the `ignore missing` clause at the end. This particular feature prevents Jinja2 from throwing an error if there is a missing file.

Jinja2 set Directive

- Use the `set` directive to assign values to variables

```
{% set install = 'new' %}  
{% if install == 'new' %}  
  {% include 'new_install.conf' %}  
  {% else %}  
    {% include 'merge_install.conf' %}  
  {% endif %}
```

Jinja2 Set Directive

There are three ways to insert values into a script: the first is to pass the values as an argument when the script is executed, a second method is to read in variables from a data file as we discussed previously, and the third method, which is illustrated on this slide, is to use the `set` directive statement to define the variable within the code itself.

Jinja2 Custom Filters

- Jinja2 filters modify the content of variables.
- Filters are separated from the content by a pipe (|) symbol

```
interfaces {
    {{interface_ID | lower}} {
        unit {{unit_num}} {
            family inet {
                address {{inf_address}};
            }
        }
    }
}
```

Convert interface_ID
to a lower-case value

Over 30 filters. See
<http://jinja.pocoo.org/docs/2.9/templates/#builtin-filters>

Jinja2 Custom Filters

Variables can be modified by filters. Filters are separated from the variable by a pipe symbol (|) and may have optional arguments in parentheses. Multiple filters can be chained. If you put the filters in succession using the pipe symbol, the output of one filter is applied to the next. This example uses a filter called *lower* to take the `interface_ID` and convert it to lowercase.

For a complete list of custom filters, see <http://jinja.pocoo.org/docs/2.9/templates/#builtin-filters>.

Jinja2 Math Operators

Operator	Description
+	Adds two objects together. Usually the objects are numbers, but if both are strings or lists, you can concatenate them this way. This, however, is not the preferred way to concatenate strings! For string concatenation, see at the (~) operator. {{ 1 + 1 }} is 2.
-	Subtract the second number from the first one. {{ 3 - 2 }} is 1.
/	Divide two numbers. The return value will be a floating point number. {{ 1 / 2 }} is {{ 0.5 }}. (Just like from __future__ import division.)
//	Divide two numbers and return the truncated integer result. {{ 20 // 7 }} is 2.
%	Calculate the remainder of an integer division. {{ 11 % 7 }} is 4.
*	Multiply the left operand with the right one. {{ 2 * 2 }} would return 4. This can also be used to repeat a string multiple times. {{ '=' * 80 }} would print a bar of 80 equal signs.
**	Raise the left operand to the power of the right operand. {{ 2**3 }} would return 8.

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 15

Jinja2 Math Operators

The slide lists the Jinja2 math functions. Of note, two operators that may not be as intuitive are the (//) operator and the (** operator. The (//) performs division but drops the remainder, simply showing a truncated integer result. The (**) symbol is for specifying exponents.

Jinja2 Comparison Operators

Operator	Description
<code>==</code>	Compares two objects for equality.
<code>!=</code>	Compares two objects for inequality.
<code>></code>	true if the left hand side is greater than the right hand side.
<code>>=</code>	true if the left hand side is greater or equal to the right hand side.
<code><</code>	true if the left hand side is less than the right hand side.
<code><=</code>	true if the left hand side is less than or equal to the right hand side.

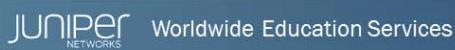
Jinja2 Comparison Operators

Most of these are standard comparison operators. The exclamation point in the `!=` operator is referred to as a “bang” and means *not*, therefore, `!=` means “not equal.”

Jinja2 Logic and Other Operators

Logic	Description
and	Return true if the left and the right operand are true
or	Return true if the left or the right operand are true
not	negate a statement
(expr)	group an expression
Other	Description
in	Perform a sequence / mapping containment test. Returns true if the left operand is contained in the right. {{ 1 in [1, 2, 3] }} would, for example, return true
is	Performs a test
	Applies a filter
~	Converts all operands into strings and concatenates them

© 2017 Juniper Networks, Inc. All rights reserved.



www.juniper.net | 17

Jinja2 Logic and Other Operators

These are intuitive logic statements. One requiring additional explanation is the logic statement *is*, which can be used to test a variable against a common expression. To test a variable or expression, you add *is*, then the name of the test after the variable. For example, to find out if a variable is defined, you can code: `name is defined` and this will return true or false, depending on whether the name is defined in the current template context.

Jinja2 has many built-in tests. For a complete listing, see <http://jinja.pocoo.org/docs/2.9/templates/#builtin-tests>.

Agenda: Jinja2 and Junos PyEZ

- Jinja2 Overview
 - Jinja2 Syntax
- Creating a Junos PyEZ, YAML, and Jinja2 Solution

Creating a Junos PyEZ, YAML, and Jinja2 Solution

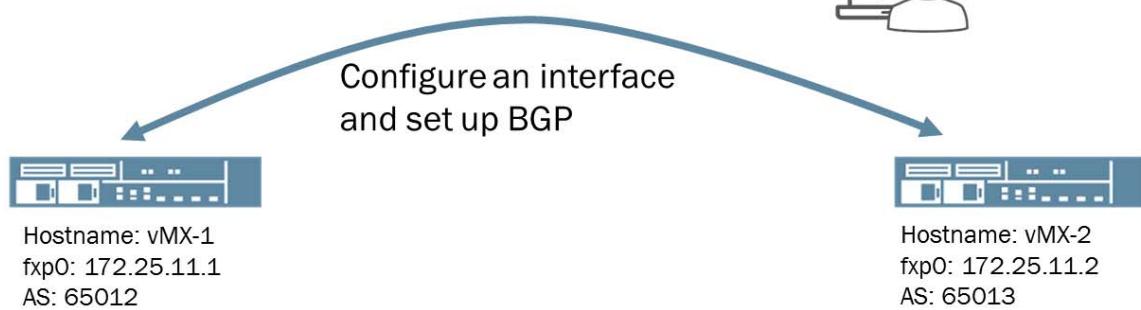
The slide highlights the topic we discuss next.

Jinja2 Case Study (1 of 10)

- The case study scenario:

- Routers have management access to fxp0
- Need to configure an interface
- Need to configure BGP

There has got to be a way to do this . . .
 This is the same configuration, just
 The variables are different.



Jinja2 Case Study, Part 1

To illustrate how Junos PyEZ, Jinja2 and YAML can be used to automate devices running the Junos OS, we are going to walk you through a case study step-by-step. In this scenario, you have two vMX devices: vMX-1 and vMX-2. Both devices have only an out-of-band management interface fxp0 configured. We want to apply the same configuration to both devices.

As part of the case study, you will view how to build the Jinja2 templates, put relevant data into YAML files, use a Junos PyEZ script to merge that data with the template, then deploy the new configurations to the devices.

Let's get started.

Jinja2 Case Study (2 of 10)

- Identify relevant parts of the configuration – part 1

```
interfaces {
    ge-0/0/0 {
        unit 0 {
            family inet {
                address 172.17.1.2/24;
            }
        }
    }
    lo0 {
        unit 0 {
            family inet {
                address 192.168.2.1/32;
            }
        }
    }
}
```

Jinja2 Case Study, Part 2

Identifying relevant parts of the configuration; the first step is to get or build a sample configuration, then identify the parts of it that you want to turn into template values. The values that you will use for this case study are outlined on the slide. You will choose which pieces you want to build into your unique template.

Also note that we are using a text configuration for this particular case study, but you could use XML, JSON, or Junos OS set commands just as simply.

Jinja2 Case Study (3 of 10)

- Identify relevant parts of the configuration – part 2

```
routing-options {
    static {
        route 0.0.0.0/0 next-hop 172.25.11.254;
    }
    autonomous-system 65513;
}
protocols {
    bgp {
        group ext-peers {
            type external;
            peer-as 65512;
            neighbor 172.17.1.1;
        }
    }
}
```

Jinja2 Case Study, Part 3

To identify relevant parts of the configuration; note that this slide identifies values in the [edit routing-options] and [edit protocols] hierarchy that you may want to include in our template.

Jinja2 Case Study (4 of 10)

▪ Create the template – part 1

```
interfaces {
    {{bgp.interface}} {
        unit {{bgp.unit_num}} {
            family inet {
                address {{bgp.address}}/{{bgp.address_cidr}};
            }
        }
    }
    lo0 {
        unit {{loopback.unit_num}} {
            family inet {
                address {{loopback.address}}/32;
            }
        }
    }
}
```

Jinja2 Case Study, Part 4

The slide shows the same basic configuration as before. This time however, you have replaced the key values of interest with Jinja2 variables. The names of the variables are not set, so they can be anything that you select. You will see how these variable names are used in the YAML file in a few moments. Notice how the variable names are prefaced by either `bgp` or `loopback`. You'll see that the YAML file is divided into two sections: `bgp` and `loopback`. The `bgp` and `loopback` prefixes specify in which section the variable is located.

Jinja2 Case Study (5 of 10)

- Create the template – part 2

```
routing-options {  
    static {  
        route 0.0.0.0/0 next-hop 172.25.11.254;  
    }  
    autonomous-system {{bgp.as_num}};  
}  
protocols {  
    bgp {  
        group ext-peers {  
            type external;  
            peer-as {{bgp.peer_as}};  
            neighbor {{bgp.neighbor_ip}};  
        }  
    }  
}
```

- Save the completed file as case1.j2

Jinja2 Case Study, Part 5

This slide shows the rest of the Jinja2 template. Once the template is created, save it with a `.j2` extension.

Jinja2 Case Study (6 of 10)

- Build the YAML file for vMX-1

```
---
```

```
bgp:
    interface: ge-0/0/0
    unit_num: 0
    address: 172.17.1.1
    address_cidr: 24
    as_num: 65012
    peer_as: 65013
    neighbor_ip: 172.17.1.2

loopback:
    unit_num: 0
    address: 192.168.2.1
```

- Save file as vMX-1.yml

Jinja2 Case Study, Part 6

Notice the `bgp` and `loopback` YAML mappings. Each of these mappings contain sub-mappings. To access any of the sub-mappings, the Jinja2 template has to use the dot (.) notation: `bgp.interface`. The YAML file on the slide contains all of the values needed to populate the Jinja2 variables in the Jinja2 template.

Once the YAML file is created, it is saved as a `.yml` file. The slide shows the YAML file for the vMX-1 device.

Jinja2 Case Study (7 of 10)

- Build the YAML file for vMX-2

```
---
```

```
bgp:
    interface: ge-0/0/0
    unit_num: 0
    address: 172.17.1.2
    address_cidr: 24
    as_num: 65013
    peer_as: 65012
    neighbor_ip: 172.17.1.1

loopback:
    unit_num: 0
    address: 192.168.2.2
```

- Save file as vMX-2.yml

Jinja2 Case Study, Part 7

This slide looks the same as the YAML file for vMX-1, as it should, because it is going to be used by the same Jinja2 template. If you look closely at the values for each mapping, they are the unique values needed to configure the vMX-2 device.

Jinja2 Case Study (8 of 10)

■ Create the Junos PyEZ Script – part 1

```

1 from jnpr.junos import Device
2 from jnpr.junos.utils.config import Config
3 from jnpr.junos.exception import *
4 from jinja2 import Template
5 import yaml
6 import sys
7
8 junos_hosts = [ 'vMX-1', 'vMX-2' ]
9 for host in junos_hosts:
10     try:
11         # Open and read the YAML file.
12         myFile = host + '.yml'
13         with open(myFile, 'r') as fh:
14             data = yaml.load(fh.read())
15         # Open and read the Jinja2 template file.
16         with open('case1.j2', 'r') as t_fh:
17             t_format = t_fh.read()

```

Jinja2 Case Study, Part 8

The Junos PyEZ script is what ties everything together and pushes the result to the routers. The first six lines of the script import the necessary modules and classes. In the previous chapter, you saw all of these libraries except for the YAML library. The YAML library is used to read and import the YAML file correctly.

On line 8, the vMX-1 and vMX2 devices are assigned to a Python list. In this fashion you could create a list of many devices and have configurations assigned to them as well. Line 9 starts a `for` loop that will apply all of the steps in the loop to each device in turn.

Line 10 begins a `try` block; if an error occurs between line 10 and line 30, execution of the script will immediately pass to line 31, listed on the next page, where it will test for various errors.

Line 12 assigns the value of `vMX-1.yml` to the `myFile` variable the first time through the loop, and `vMX-1.yml` to the `myFile` variable the second time through the loop.

Line 13 opens the file as read-only and assigns the name `fh` to the open file. Line 13 then reads the file and parses it with the `yaml.load()` method.

Line 16 opens the `case1.j2` file as read-only. Line 17 then reads the file and stores the Jinja2 template in the variable `t_format`.

Continued on the next page.

Jinja2 Case Study (9 of 10)

▪ Create the Junos PyEZ Script – part 2

```

18     # Associate the t_format template with the Jinja2 module
19     template = Template(t_format)
20     # Merge the data with the template
21     myConfig = template.render(data)
22
23     dev = Device(host=host, user='lab', password='lab123')
24     dev.open()
25     config = Config(dev)
26     config.lock()
27     config.load(myConfig, merge=True, format="text")
28     config.pdiff()
29     config.commit()
30     dev.close()
31 except LockError as e:
32     print "The config database was locked!"
33 except ConnectTimeoutError as e:
34     print "Connection timed out!"
```

Jinja2 Case Study, Part 9

Line 19 associates the Python Jinja2 library template the t_format.jinja2 file. Line 19 the merges the Jinja2 template with the YAML file.

Line 23 creates the NETCONF connection string for the device, and line 24 opens the connection. Line 25 associates the Config module with the dev device instance.

Line 26 locks the Junos OS configuration database. Line 27 loads the new configuration. On line 27 we also specify format="text" to indicate the format of the configuration data. This is necessary when the format of the text is not known. If we didn't specify anything, the configuration would default to XML..

Line 28 shows the difference between the active configuration and the newly-loaded candidate configuration. Line 29 commits the configuration.

Line 30 unlocks the database and closes the connection with the device. (Note, typically a config.unlock() line is used, however the dev.close() also unlocks.)

Lines 31 through 34 handle any errors that may have occurred.

This script does not include the necessary error checking and secure authentication practices; these are omitted due to space limitations.

Jinja2 Case Study (10 of 10)

Script Output vMX-1

```
[edit interfaces]
+  ge-0/0/0 {
+    unit 0 {
+      family inet {
+        address 172.17.1.1/24;
+      }
+    }
+  lo0 {
+    unit 0 {
+      family inet {
+        address 192.168.2.1/32;
+      }
+    }
+  }
[edit]
+  routing-options {
+    static {
+      route 0.0.0.0/0 next-hop 172.25.11.254;
+    }
+    autonomous-system 65012;
+  }
  protocols {
+    bgp {
+      group ext-peers {
+        type external;
+        peer-as 65013;
+        neighbor 172.17.1.2;
+      }
+    }
+  }
+ }
```

Script Output vMX-2

```
edit interfaces]
+  ge-0/0/0 {
+    unit 0 {
+      family inet {
+        address 172.17.1.2/24;
+      }
+    }
+  lo0 {
+    unit 0 {
+      family inet {
+        address 192.168.2.2/32;
+      }
+    }
+  }
[edit]
+  routing-options {
+    static {
+      route 0.0.0.0/0 next-hop 172.25.11.254;
+    }
+    autonomous-system 65013;
+  }
  protocols {
+    bgp {
+      group ext-peers {
+        type external;
+        peer-as 65012;
+        neighbor 172.17.1.1;
+      }
+    }
+  }
+ }
```

Jinja2 Case Study, Part 10

As you can see from the generated script output of the `config.pdiff()` method, both configurations were applied correctly. The output below is for easier reading.

Script Output for vMX-1:

```
[edit interfaces]
+  ge-0/0/0 {
+    unit 0 {
+      family inet {
+        address 172.17.1.1/24;
+      }
+    }
+  }
+  lo0 {
+    unit 0 {
+      family inet {
+        address 192.168.2.1/32;
+      }
+    }
+  }
+ }
```

Continued on the next page.

Script Output for vMX-1 (contd.)

```
[edit]
+ routing-options {
+     static {
+         route 0.0.0.0/0 next-hop 172.25.11.254;
+     }
+     autonomous-system 65012;
+ }
+ protocols {
+     bgp {
+         group ext-peers {
+             type external;
+             peer-as 65013;
+             neighbor 172.17.1.2;
+         }
+     }
+ }
```

Script Output for vMX-2

```
edit interfaces]
+ ge-0/0/0 {
+     unit 0 {
+         family inet {
+             address 172.17.1.2/24;
+         }
+     }
+ lo0 {
+     unit 0 {
+         family inet {
+             address 192.168.2.2/32;
+         }
+     }
+ }
[edit]
+ routing-options {
+     static {
+         route 0.0.0.0/0 next-hop 172.25.11.254;
+     }
+     autonomous-system 65013;
+ }
+ protocols {
+     bgp {
+         group ext-peers {
+             type external;
+             peer-as 65012;
+             neighbor 172.17.1.1;
+         }
+     }
+ }
```

More Information on Jinja2 and Junos PyEZ

- Information on Jinja2
 - <http://jinja.pocoo.org/docs/2.9/intro/>
- Jinja2/YAML/Python examples
 - <https://github.com/Juniper/community-NCE>
- Junos PyEZ Developer Guide
 - https://www.juniper.net/techpubs/en_US/junos-pyez2.0/information-products/pathway-pages/junos-pyez-developer-guide.html

More Information on Jinja2 and Junos PyEZ

This chapter gave a brief overview of Jinja2 and showed how Jinja2, along with Junos PyEZ and YAML, can be used to automate multiple devices running Junos OS. The slide show suggested additional resources where you can learn more about Jinja2 and Junos PyEZ.

Summary

- In this content, we:

- Learned how Jinja2 can help you automate the Junos OS
- Gained understanding of the syntax of Jinja2
- Created Jinja2 templates and automated a Junos OS configuration

We Discussed

- How Jinja2 can help you automate the Junos OS,
- The Jinja2 syntax, and
- How Jinja2 is used to help create a Junos OS automation solution.

Review Questions

1. Which command is needed to install Jinja2?
2. What is the syntax to create a comment in Jinja2?
3. What does the set command do?
4. What does this statement do? config.pdiff()

Review Questions

- 1.
- 2.
- 3.
- 4.

Lab: **Jinja2 and Junos PyEZ**

- Learn to automate a Junos OS device using a Jinja2 template with PyEZ and YAML.

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 33

Lab: Using Jinja2 Templates with PyEZ

The slide lists the objective for this lab.

Answers to Review Questions

1.

Use the `pip install jinja2` command to install Jinja2.

2.

Create a comment in Jinja2 by using the following format `{# . . . #}`.

3.

The `set` command assigns a value to variable.

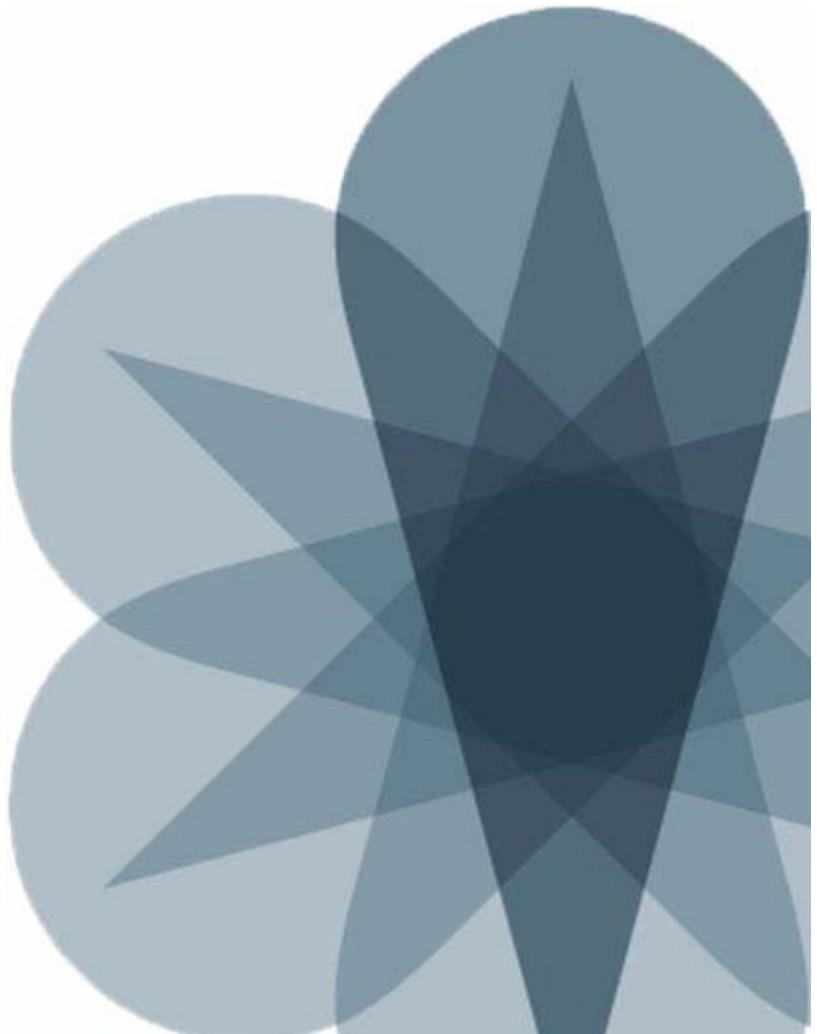
4.

The `config.pdiff()` statement shows the difference between the candidate configuration and the active configuration.



Junos Platform Automation and DevOps

Chapter 7: Ansible



Objectives

- After successfully completing this content, you will be able to:
 - Describe the format of an Ansible YAML playbook
 - Describe how Ansible interfaces with Junos OS devices
 - Explain how to operate Junos OS devices using Ansible

We Will Discuss:

- The format of an Ansible playbook;
- How Ansible interfaces with the Junos OS; and
- How to operate Junos devices using Ansible.

Agenda: Ansible

- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The **junos_shutdown** Module
- The **junos_get_facts** Module
- The **junos_install_config** Module
- Advanced Configuration Deployment
- Additional References

Introduction to Ansible

The slide lists the topics we will discuss. We discuss the highlighted topic first.

Introduction to Ansible

▪ Ansible overview

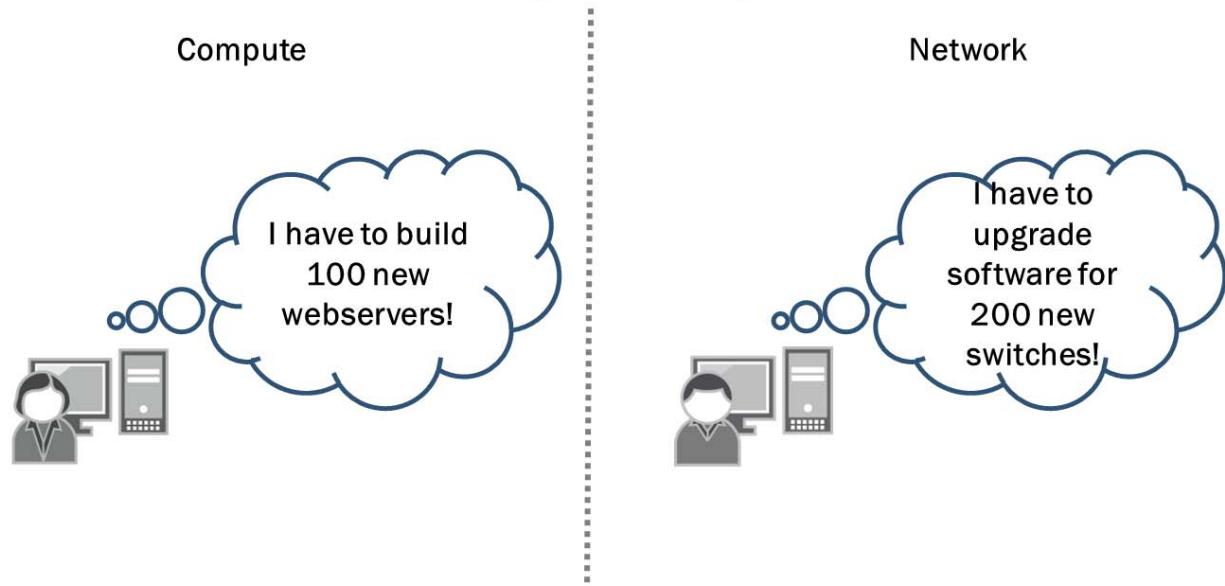
- Ansible provides tools to accelerate the deployment of infrastructure
- It also provides the ability to manage configuration and assist with operations
- It was initially created for compute and cloud but now has been ported to support network infrastructure
- Ansible provides idempotent operations
 - Ansible playbooks may be run multiple times to yield the same result
 - Ansible modules only execute a change if required

Introducing the Ansible Tool

Ansible's goals are similar to other tools in the configuration management space such as Puppet, Salt, and Chef. One of the differences as we shall see is Ansible is noticeably simpler. Also, an important concept in configuration management is idempotence. Idempotence means we can run the same operation multiple times to yield the same result. In other words, we should be able to execute an Ansible playbook twice without changing anything after the first execution (assuming we haven't changed any files etc that Ansible is deploying).

Why Use Ansible?

- The challenge of scaling operations
 - Ansible uses simple tools and syntax to automate repetitive tasks for both networking and compute



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

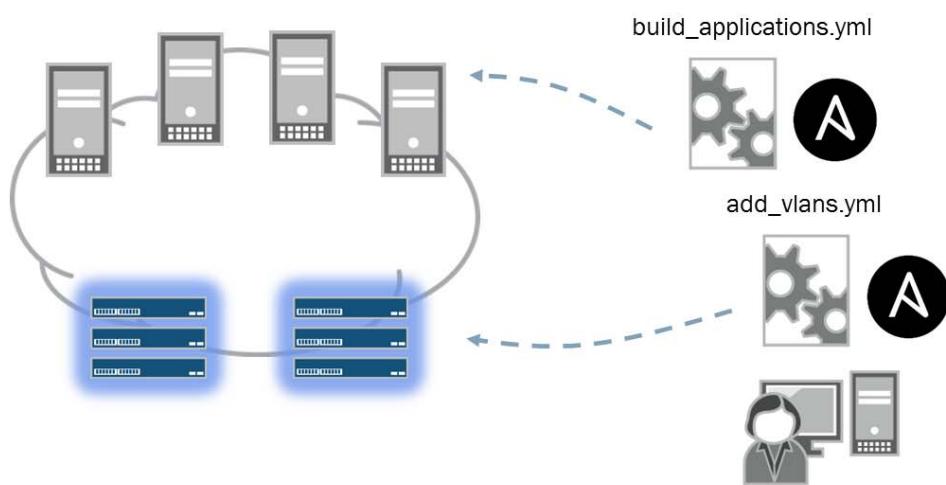
www.juniper.net | 5

Why Use Ansible?

The purpose of this illustration is to show the usefulness of Ansible. Although not necessarily comprehensive, Ansible offers tools to automate repetitive tasks using simple components such as SSH for connectivity and YAML to express tasks which are units of work (e.g., apply configuration, get device information, etc.).

Use Cases (1 of 2)

- Reducing time to deployment
 - Using Ansible tools, network changes can be automated with application rollouts and server infrastructure updates

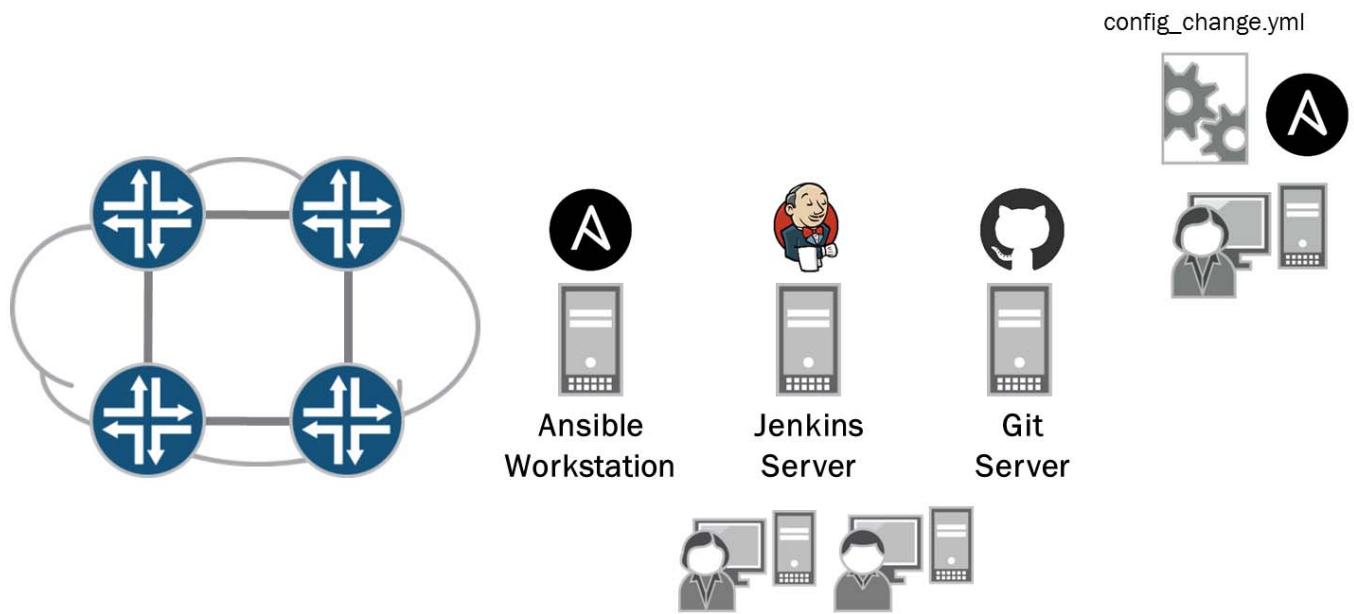


Use Cases: Part 1

Ansible is helpful for reducing the time to deploy new applications. Engineers may be using Ansible to spin up and configure server infrastructure to prepare for applications. Similarly, Ansible may be used to deploy network configurations. Using the Ansible tool sets, we can couple the deployment of network and server infrastructure to reduce the time to deploy new services.

Use Cases (2 of 2)

- Improving change management
 - Reduce the risk associated with infrastructure changes by integrating Ansible into change management workflows



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

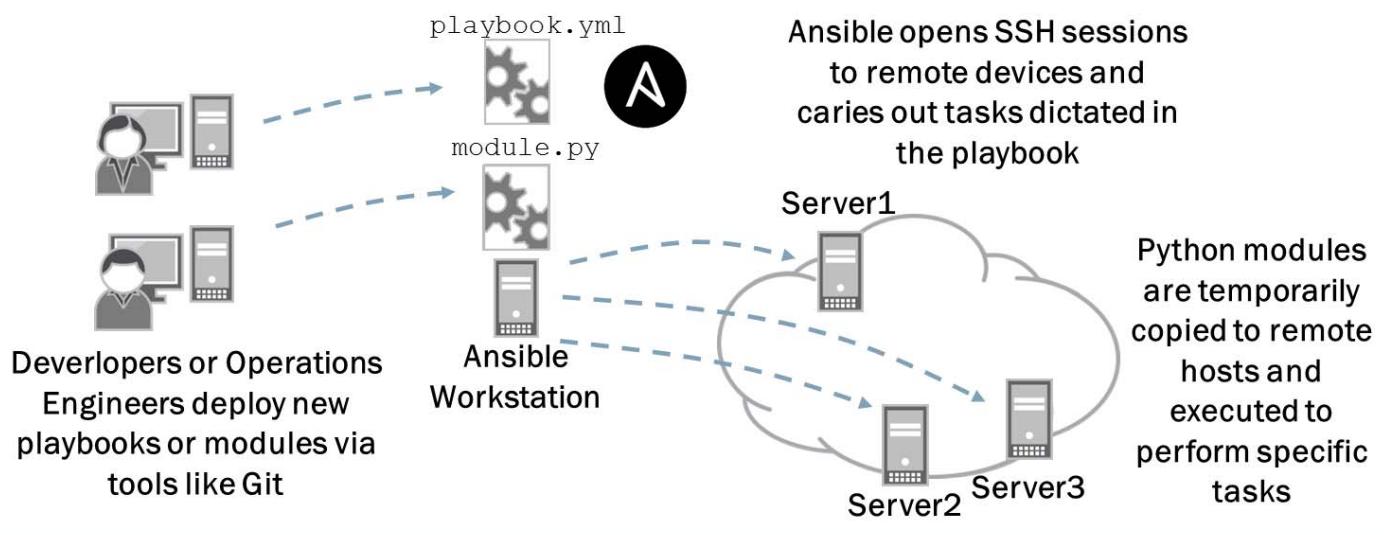
www.juniper.net | 7

Use Cases: Part 2

You can also use Ansible to improve change management. Software engineers perform rigorous checks on changes at a rapid pace. Using Ansible, we can vet infrastructure changes by running Ansible playbooks through the same tool chains as developers to ensure success of new deployments without slowing down the deployment process. As an example, playbooks may be checked into a version control system such as Git to ensure any playbooks may be rolled back to previous versions if required. Also, engineers may use existing continuous delivery tools such as Jenkins to help validate the functionality of all playbooks.

Normal Ansible Operation

- How does Ansible typically work?
 - Python modules are copied and executed on each relevant remote device over SSH
 - Playbooks written in YAML carry out the tasks defined in Python modules



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

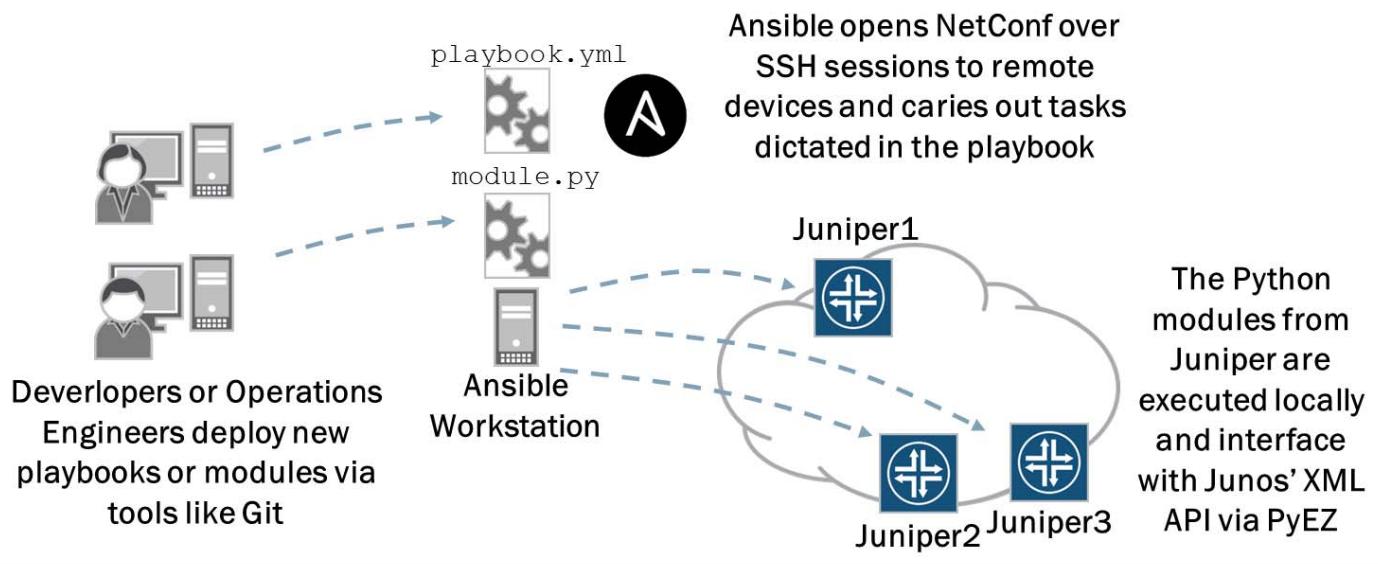
www.juniper.net | 8

How Does Ansible Typically Work?

Those familiar with Ansible will notice that the Juniper implementation differs slightly from a typical implementation. Here we provide some background on a typical high level Ansible deployment. Normally, the Ansible workstation uses SSH to connect to remote hosts and copies (temporarily) the relevant Ansible modules (typically Python code) to carry out an operation (or task in Ansible terms) on the infrastructure.

Ansible Operation with Junos

- How does Ansible work with Junos?
 - Junos platforms modules are executed locally and leverage PyEZ



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 9

How Does Ansible Work with Junos?

Notice from the previous slide that there is no agent on the remote device to run Ansible. As such, rather than require installing a package to every Junos device to get up and running with the tool, the Juniper implementation of Ansible requires all tasks to run locally on the Ansible Workstation. Using NetConf over SSH, Ansible uses the Juniper modules to interface with the XML API of the Junos device using PyEZ on the local Ansible Workstation.

Agenda: Ansible

- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The `junos_shutdown` Module
- The `junos_get_facts` Module
- The `junos_install_config` Module
- Advanced Configuration Deployment
- Additional References

Building a Basic Ansible Environment

The slide highlights the topic we discuss next.

Building a Basic Ansible Environment (1 of 2)

- Python 2.7
- Install PyEZ latest version
- Install Ansible latest version
 - Using pip – pip install ansible
 - http://docs.ansible.com/ansible/intro_installation.html
 - Install Junos Ansible Galaxy role

```
(jaut_env) [lab@jaut-desktop ~]$ ansible-galaxy install
Juniper.junos -p ~/virtualenv/jaut_env/
- downloading role 'junos', owned by Juniper
- downloading role from https://github.com/Juniper/ansible-
junos-stdlib/archive/1.4.2.tar.gz
- extracting Juniper.junos to
/home/lab/virtualenv/jaut_env/Juniper.junos
- Juniper.junos (1.4.2) was installed successfully
```

Building a Basic Ansible Environment, Part 1

This slide illustrates the basics of getting Ansible up and running with a Junos device. First note that Ansible uses PyEZ to interface with the Junos OS. As such, you must first perform a PyEZ installation. To take advantage of all the options you will want the latest version of PyEZ. Juniper's Ansible modules are actively being updated so you will want to install the latest version of Ansible as well.

Installing the Juniper.Junos Ansible-Galaxy role as shown above will allow you to use Ansible modules not available as part of the Ansible core modules

Building a Basic Ansible Environment (2 of 2)

- Configure NETCONF on the Junos OS device

```
root@R1> show configuration system services
ssh;
netconf {
    ssh;
}
web-management {
    http {
        interface ge-0/0/0.0;
    }
}
```

Building a Basic Ansible Environment, Part 2

Lastly within Junos, we need to make sure NETCONF over SSH is enabled under the `system services` hierarchy.

Ansible “Hello World!” (1 of 2)

■ “Hello World!” for Ansible

hello_world.yml

```
---
- name: Hello World!
  hosts: vsrx
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  vars_prompt:
    - name: USERNAME
      prompt: User name
      private: no
    - name: DEVICE_PASSWORD
      prompt: Device password
      private: yes

  tasks:
    - name: Get Junos device information
      junos_get_facts:
        host={{ inventory_hostname }}
        user={{ USERNAME }}
        passwd={{ DEVICE_PASSWORD }}
      register: junos

    - name: Print Junos facts
      debug: msg="{{ junos.facts }}"
```

Play name.

Group of hosts.

A play is the grouping of hosts to tasks.
This playbook contains a single play.
Multiple plays are allowed per playbook.

Use the prompt to
gather username and
password for the Junos
device.

This play executes two tasks.
The first retrieves device
facts. The second prints
them to the terminal.

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 13

Ansible “Hello World!”, Part 1

Here we illustrate our first Ansible playbook. We are performing a “Hello World!” by printing basic device facts. Note that Ansible playbooks are expressed in YAML for simplicity and readability. The below describes the high level playbook operation.

1. Name the playbook “Hello World!”
2. Operate on the group of hosts from the inventory named `vsrx`.
3. Import the role named `Juniper.junos`, which contains Python modules to operate Junos devices.
4. Specify `connection: local` to prevent Ansible from trying to SSH to the remote device.
5. Specify `gather_facts: no` to prevent Ansible from gathering facts on the local Ansible Workstation.
6. Use the `vars_prompt:` keyword to have Ansible gather username/password from a prompt.
7. Execute a task called `Get Junos device information` to have Ansible retrieve facts from the Junos device which shall be saved to a variable called `junos`.
8. Execute a task called `Print Junos facts` to have Ansible print the facts from the Junos device to the terminal window.

Ansible “Hello World!” (2 of 2)

▪ “Hello World!” for Ansible

Terminal Output

```

bash $ ansible-playbook hello_world.yml
User name: root
Device password: Type out root user password for the Junos device. Use the ansible-playbook utility and pass the playbook name as an argument to execute.

PLAY [Hello World!] ****
TASK: [Get Junos device information] ****
ok: [192.168.64.51]

TASK: [Print Junos facts] ****
ok: [192.168.64.51] => {
    "msg": {"u'domain': None, u'serialnumber': u'b34235e6bda7', u'ifd_style': u'CLASSIC', u'version_info': {u'major': [12, 1], u'type': u'X', u'build': 5, u'minor': [46, u'D', 20]}, u'REO': {u'status': u'Testing', u'last_reboot_reason': u'Router rebooted after a normal shutdown.', u'model': u'FIREFLY-PERIMETER RE', u'up_time': u'4 days, 6 hours, 7 minutes, 17 seconds'}, u'hostname': u'R1', u'fqdn': u'R1', u'vertual': True, u'has_2RE': False, u'switch_style': u'NONE', u'version': u'12.1X46-D20.5', u'srx_cluster': False, u'HOME': u'/cf/root', u'model': u'FIREFLY-PERIMETER', u'personality': u'SRX_BRANCH'}}

Two tasks completed. No changes to any hosts.

PLAY RECAP ****
192.168.64.51 : ok=2    changed=0    unreachable=0    failed=0

```

The `u` simply indicates Unicode encoding. It may be safely ignored.

Ansible “Hello World!”, Part 2

Here are the results of our first “Hello World!” playbook. We execute the playbook using the `ansible-playbook` CLI utility and passing the playbook name as an argument. Notice that running a playbook allows us to see some the results of each task as well as the changes providing a log for change management purposes. In this playbook, Ansible is making an API call to PyEZ to retrieve facts from our Junos device. The facts are being printed to the terminal during the playbook execution.

The Ansible Command Line

- Ansible has a number of CLI utilities to operate infrastructure

- ansible
 - Allows you to run a module or basic commands on another host without playbooks
- ansible-doc
 - Shows relevant documentation for modules
- ansible-galaxy
 - Creates role skeletons or downloads roles from the Galaxy repository
- ansible-playbook
 - Executes a playbook passed as an argument on remote hosts via push operations
- ansible-pull
 - Executes plays on a local host via pull operations
- ansible-vault
 - Creates and manages an encrypted password “vault” that saves SSH passwords

The Ansible Command Line

The open source Ansible tool provides a number of helpful utilities to operate infrastructure.

- `ansible` – This CLI utility allows Ansible modules to be executed without creating playbooks. A module name and various parameters for modules may be passed as arguments to execute plays from the CLI of the Ansible workstation.
- `ansible-doc` – This utility accepts a module name as an argument and prints relevant documentation such as module arguments and example usage.
- `ansible-galaxy` – This utility may be used to create new roles or download existing shared roles from Ansible’s public repository called the Ansible Galaxy.
- `ansible-playbook` – This utility accepts a playbook name as an argument to execute plays against groups of hosts.
- `ansible-pull` – A typical Ansible operation happens via a push model. The Ansible workstation initiates configuration changes to devices. This utility inverts the operational model allowing hosts to pull configuration changes.
- `ansible-vault` – This utility manages a password encrypted YAML file that may be used to store device passwords. Passwords may be retrieved from the encrypted YAML file instead of via user prompt.

Securing Automated Access to Junos (1 of 2)

- With Ansible, as seen on the previous slide, we may prompt users for username and password to get up and running quickly
 - We may also exchange public keys then follow the steps below to setup an SSH agent
 - Alternatively, you can also add the SSH key to the inventory file or a playbook as an Ansible variable or use an Ansible vault

```
bash$ ssh-agent bash
.
.
.
Enter password if needed.

bash$ ssh-add ~/.ssh/id_rsa
Enter passphrase for /home/ansible-guru/.ssh/id_rsa:
```



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 16

Securing Automated Access to Junos, Part 1

Generating and copying SSH keys is helpful for automating secure access to Junos. In addition, to ensure the remote host may access the private key via this shell, we can perform the steps illustrated.

Securing Automated Access to Junos (2 of 2)

- The environment impacts which arguments are required when connecting to Junos devices

hello_world.yml

```
- name: Retrieve information from devices running Junos OS
  junos_get_facts:
    host={{ inventory_hostname }}
    user={{ USERNAME }}
    passwd={{ DEVICE_PASSWORD }}
  register: junos
```

Without transferring SSH certificates,
user and passwd must be
arguments for all tasks/

hello_world_certs.yml

```
- name: Retrieve information from devices running Junos OS
  junos_get_facts:
    host={{ inventory_hostname }}
  register: junos
```

After transferring certificates, if the USER
environment variable is adequate, both user and
passwd arguments may be left off.

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 17

Securing Automated Access to Junos, Part 2

Note that the environment determines what login variables are required as arguments for tasks. Once the public SSH key is generated and shared with the Junos device, users no longer need to use the user and passwd variables for tasks in their Ansible playbooks.

Ansible Command Line

- You can use Ansible to quickly run plays from the CLI without playbooks

Terminal Output

```
bash$ ansible -c local 192.168.64.51 -m junos_get_facts -M
/etc/ansible/roles/Juniper.junos/library/ -a "host={{inventory_hostname}}"
192.168.64.51 | success >> {
    "changed": false,
    "facts": {
        "HOME": "/cf/var/home/lab",
        "RE0": {
            "last_reboot_reason": "Router rebooted after a normal shutdown.",
            "model": "FIREFLY-PERIMETER RE",
            "status": "Testing",
            "up_time": "4 hours, 49 minutes, 1 second"
        },
        "domain": null,
        "fqdn": "R1",
        "has_2RE": false,
        "hostname": "R1",
        "ifd_style": "CLASSIC",
        "model": "FIREFLY-PERIMETER",
        .
        .
        .
        .
    }
}
```

The host must reside in the inventory file and the proper certificates must be copied to the Junos host in this scenario.

Data is stored and passed via JSON formatting.

Ansible Command Line

In the example above, the SSH key has been copied to the Junos device. Now using the `ansible` CLI tool, we may execute Ansible tasks from the CLI without creating playbooks.

Ansible Command Line Arguments

- Ansible also uses arguments to control operations

Terminal Output

```
bash $ ansible-playbook hello_world.yml --check --diff
User name: root
Device password:
PLAY [Hello World!] ****
TASK: [Retrieve information from devices running Junos OS] ****
ok: [192.168.64.51]
TASK: [Print Junos facts] ****
ok: [192.168.64.51] => {
    "msg": {"u'domain': None, u'serialnumber': u'b34235e6bda7', u'ifd_style': u'CLASSIC', u'version_info': {u'major': [12, 1], u'type': u'X', u'build': 5, u'minor': [46, u'D', 20]}, u'REO': {u'status': u'Testing', u'last_reboot_reason': u'Router rebooted after a normal shutdown.', u'model': u'FIREFLY-PERIMETER RE', u'up_time': u'2 days, 12 hours, 56 minutes, 52 seconds'}, u'hostname': u'R1', u'fqdn': u'R1', u'vertual': True, u'has_2RE': False, u'switch_style': u'NONE', u'version': u'12.1X46-D20.5', u'srx_cluster': False, u'HOME': u'/cf/root', u'model': u'FIREFLY-PERIMETER', u'personality': u'SRX_BRANCH"}}

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



Worldwide Education Services

www.juniper.net | 19

Ansible Command Line Arguments

Before running through the details of Ansible, note that we can execute playbooks with command line arguments. In this example, we show the usage of `--check` and `--diff`. Note that not all Ansible modules support all arguments, so it is best to test any new arguments beforehand in a staging or test environment.

- `--check` – This argument allows Ansible to perform a dry-run. Ansible shall not make any changes when this argument is used which is helpful for staging/testing.
- `--diff` – This argument allows Ansible to report any changes that would be made for staging/testing.

Note that in the scenario above no changes are being made via the playbook, so there is nothing reported from these arguments.

Agenda: Ansible

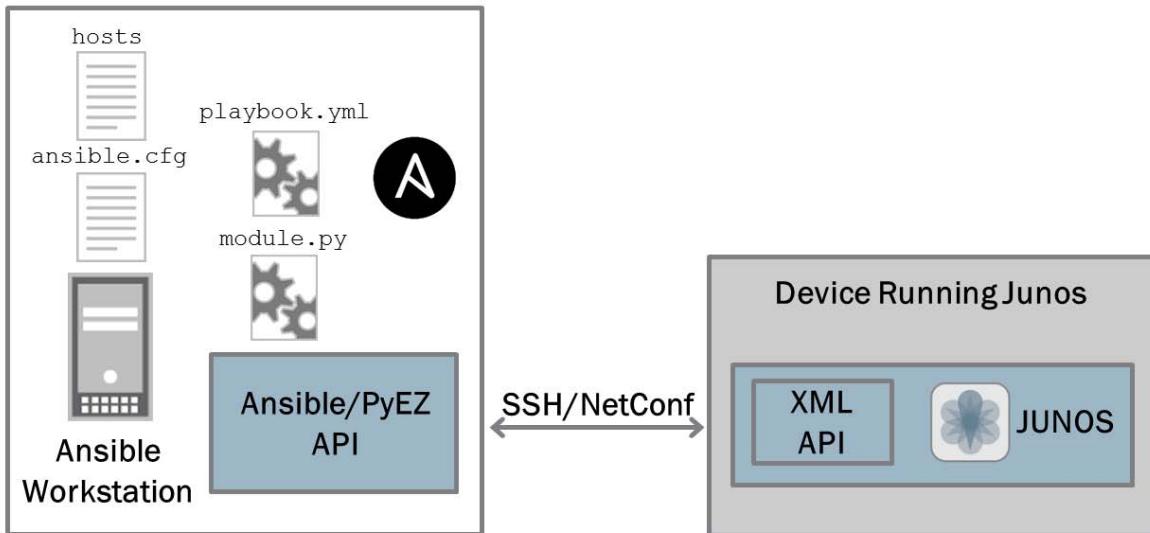
- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The `junos_shutdown` Module
- The `junos_get_facts` Module
- The `junos_install_config` Module
- Advanced Configuration Deployment
- Additional References

Creating Ansible Playbooks

The slide highlights the topic we discuss next.

Ansible and Junos Integration

▪ Ansible integration



Ansible and Junos Integration

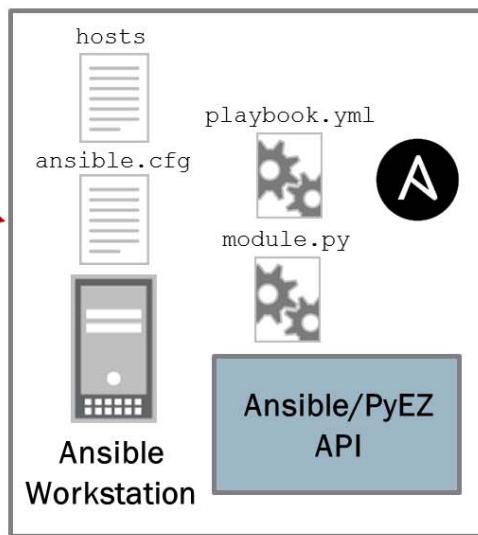
This illustrates the integration between Junos and the Ansible Workstation. Note that there are two primary components to implement Ansible and integrate it with a network leveraging the Junos OS. First is the Ansible Workstation. This can be any device including a physical server, virtual-machine, or even a laptop. Anything that can run the `ansible` CLI utility and Python can be an Ansible Workstation. Second, is a device (or devices) that run Junos with NetConf over SSH enabled/accessible as well as user that can operate the Junos device. The Ansible Workstation shall interface with the XML API of Junos via PyEZ. The Ansible Workstation has a few components for configuration and operation that will be discussed in depth in the next slides. In the meantime, here is a brief overview.

- config – controls the operation of Ansible
- hosts – contains the list of inventory that Ansible shall operate
- module – a Python file (same as normal Python modules) that performs a task on a host
- playbook – a YAML file that collects the inventory and relevant modules to perform a series of tasks
- PyEZ API – behind the scenes, Ansible is using the PyEZ API to interface with the XML API of Junos devices

Ansible Configuration File

- Ansible has a configuration file to specify certain parameters for operation
 - Examples: SSH username, SSH port, directory for modules, etc.

Config files are simple text files to declare specifics for the deployment. The location is configurable, but by default, it resides in /etc/ansible.

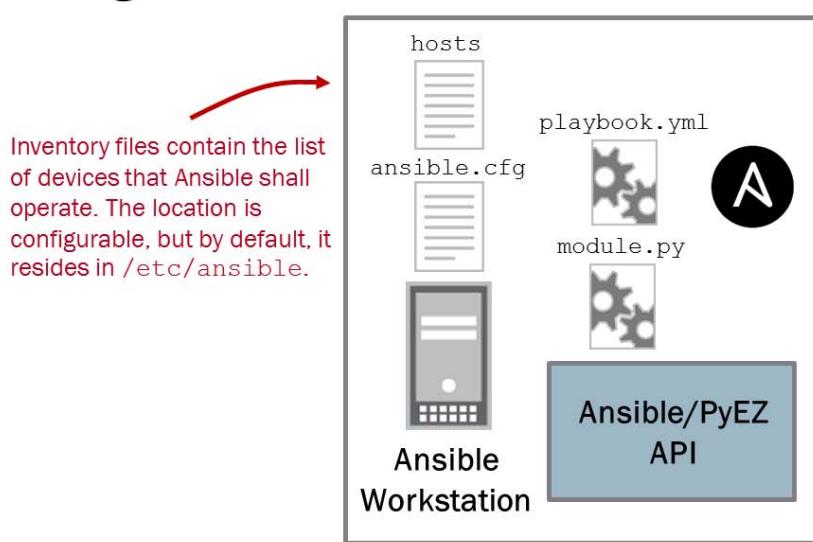


The Ansible Configuration File

Ansible requires a configuration file to control its operation. In the configuration file, we can control the SSH port, usernames, directory to libraries, etc.

Ansible Inventory File

- Ansible has an inventory file called `hosts` which Ansible checks by default
 - You can also create custom inventory files and pass them as arguments to Ansible

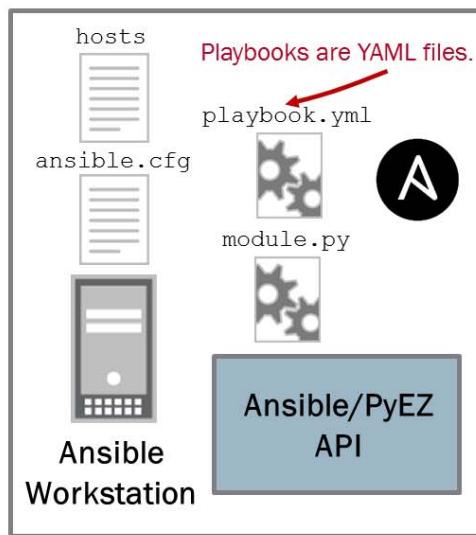


The Ansible Inventory File

Inventory files contain the list of devices that Ansible shall operate. We can identify different hosts using groups. For example, we might want to differentiate by device (such as `srx`, `mx`, `ex`). We can also get granular in case there are tasks common to specific device groups (such as `ex_leaf`, `ex_spine`, `mx_internet_edge`, `mx_core`). Also, if desired, we can pass the inventory file as an argument when running Ansible. This allows us to sort different devices into different inventory files if desired. In our previous example we created a group named `vsrx` and added the IP address `192.168.64.51`.

Overview of Playbooks

- A playbook is a unit of work represented in YAML
 - We can create playbooks to carry out plays or series of tasks on multiple hosts



Ansible Playbooks

Playbooks are created by engineers in YAML. Playbooks allow engineers to automate workflows on multiple devices. Individual tasks (written in Python modules as noted previously) are organized into playbooks to complete a larger group of work. Ansible comes with a number of modules already created by the Ansible community. We can combine these with the Juniper ones to generate our own custom playbooks.

Different type of Ansible Modules

▪ Ansible Module Library

- Officially supported by Ansible and the community, follow Ansible best practices and guidelines
- Supported in Ansible Tower
- Ships with Ansible

▪ Ansible Galaxy

- App Store for Ansible, anyone is free to write and publish a module
- Separate installation is required

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 25

Ansible Modules

Modules are chunks of code typically written in the Python language. Ansible comes with some core modules that are part of the Ansible Module Library to perform common tasks (gathering host facts, installing packages, deploying configurations, and many others). Also, as shown in earlier slides, we can extend Ansible's functionality by downloading modules from a repository called the Ansible Galaxy. Using the repository of modules or by creating custom ones, we can call modules in playbooks to complete larger workflows such as preparing a virtual-machine or network device for production.

In earlier versions of Ansible Juniper only had modules in the community supported Galaxy modules. Now, there are Juniper modules that are part of the Ansible Module Library. The functionality of some of the modules at the in the Ansible Module Library and Galaxy level overlap. You can mix and match Core and Galaxy modules. Choose the one that best meets your needs. Also, if you have existing playbooks that use the Galaxy modules, they do not have to be moved over to use the Core modules.

Modules in the Galaxy Library are supported by Juniper. Modules in the Ansible Module Library are developed and maintained by Ansible and support for them is through Ansible.

Ansible Library and Ansible Galaxy Modules

Ansible Library	
junos_command	N
junos_config	N
junos_netconf	N
junos_package	N
junos_facts	N
Junos_rpc	N
Junos_template	N
Junos_user	N

Ansible Galaxy / Juniper.junos	
junos_get_facts	N, C
junos_install_config	N, C
Junos_ping	N
Junos_get_tables	N
junos_zeroize	N, C
junos_srx_cluster	N, C
junos_install_os	N
junos_shutdown	N
junos_get_config	N
junos_rollback	N
junos_commit	N
junos_rpc	N
junos_cli	N

N: Netconf
N, C: Netconf & Console

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 26

Junos Ansible Modules

The slide shows a listing of Ansible Library modules and Ansible Galaxy modules.

To learn more about the Juniper modules in the Ansible Library see: http://docs.ansible.com/ansible/list_of_network_modules.html#junos.

To explore how to use the Galaxy modules visit the Galaxy module website at <http://junos-ansible-modules.readthedocs.io/>.

Ansible Playbook Template

- Ansible playbook template for Juniper devices
 - Now that we have covered Ansible, we shall begin detailing how to produce playbooks to automate tasks
 - Ansible has a number of built-in features
 - The below illustrates a template that may be used to start creating Ansible playbooks for Juniper tasks

juniper_template.yml

```
---
- name: Play Name
  hosts: juniper_hosts
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no
```

name: Play Name → Create a name for the play.

hosts: juniper_hosts → Identify the host group from the inventory file.

roles:
- Juniper.junos → Import the Juniper.junos role to access relevant modules.

connection: local → Specify to execute locally. Recall we must use this option since we are not executing modules on the remote devices. They are executed locally using the PyEZ API which interfaces with the XML API of Junos. Since we are running locally, there is no reason to gather facts.

gather_facts: no →

Ansible Playbook Template

This slide provides a basic template for an Ansible playbook operating on Junos. We can copy and paste this template to start any new Ansible playbook. Note though that the `hosts:` referenced must be updated for the environment.

Creating Ansible Playbooks (1 of 5)

▪ Tasks

- Tasks execute an operation for a playbook
- Playbooks may contain multiple tasks, but a task may only perform a single operation

playbook_example_1.yml

```
---
- name: Play Name
  hosts: juniper_hosts
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no
  tasks: ← The tasks: section contains a list of operations to perform
    - name: Checking NETCONF connectivity ← Naming each task simplifies debug
      wait_for: host={{ inventory_hostname }} port=830 timeout=5
```

In this example, the task simply checks to make sure port 830 (NetConf) is reachable.

Creating Ansible Playbooks, Part 1

Over the next few slides, we shall introduce some built-in features and modules that are useful for Ansible playbooks. The first is the `tasks:` built in keyword which allows us to organize units of work presented by modules. In this example, we have a single task running a `wait_for:` to verify connectivity to hosts defined by `junos_hosts` via NetConf (destination-port 830). Note that 830 is the default port that Junos uses to listen for NetConf over SSH sessions when the service is enabled on Junos devices. Also, regarding Ansible syntax, we can execute multiple tasks in the same workbook, but we can only execute a single unit of work per task.

Creating Ansible Playbooks (2 of 5)

▪ Variable substitution

- To reference variables, we may use basic Jinja2 syntax
- In the example below, `inventory_hostname` is a reserved variable containing the hostname or IP address from the inventory

`playbook_example_1.yml`

```
---
- name: Play Name
  hosts: juniper_hosts
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:
    - name: Checking NETCONF connectivity
      wait_for: host={{ inventory_hostname }} port=830 timeout=5
```

We can substitute variables using Jinja2 syntax where double brackets surround the variable name.



© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 29

Creating Ansible Playbooks, Part 2

This slide illustrates basic variable substitution. Ansible uses Jinja2 syntax which is a templating language used in Python. We shall cover the details of Jinja2 shortly. For now, we just need to get accustomed to seeing the `{ { }}` syntax for substituting variables. Ansible uses the `inventory_hostname` as a reserved variable to iterate over each device in the `hosts:` device group.

Creating Ansible Playbooks (3 of 5)

- Saving results to variables

- The `register:` keyword allows us to save results from tasks to a variable

`playbook_example_2.yml`

```
---
- name: Play Name
  hosts: juniper_hosts
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:
    - name: Checking NETCONF connectivity
      wait_for: host={{ inventory_hostname }} port=830 timeout=5

    - name: Retrieve information from devices running Junos OS
      junos_get_facts:
        host={{ inventory_hostname }}
      register: junos
```

← Using `register:` we can save the results of commands to variables.

Creating Ansible Playbooks, Part 3

The `register:` keyword allows us to save the results of tasks to variables. In this example, we are calling the `junos_get_facts:` method in a task to gather facts on the remote device. We can later operate on those facts by referencing the variable `junos`.

Creating Ansible Playbooks (4 of 5)

▪ Conditionals

- The `when:` keyword operates like an `if` statement
- A task is only executed if the `when:` returns true

`playbook_example_3.yml`

```
---
- name: Play Name
  hosts: juniper_hosts
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve information from devices running Junos OS
      junos_get_facts:
        host={{ inventory_hostname }}
      register: junos

    - name: Reboot select devices
      junos_shutdown:
        host={{ inventory_hostname }}
        shutdown="shutdown"
        reboot="yes"
      when: junos.facts.hostname == "R1" We can use the when: conditional like an if
```

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 31

Creating Ansible Playbooks, Part 4

Ansible provides a `when:` statement which acts like an `if` from other programming languages. Using facts gathered from other tasks we can operate on specific devices using conditional matching. In the above, the first task saves device facts to a variable named `junos`. Then, we can perform a conditional match should we only want to operate on a host with a hostname of `R1`. In this example playbook, our second task shall only reboot the Junos device if the hostname of the device is `R1`. Also, notice the position of the `when:` statement relative to `junos_shutdown:`. Programmers may expect the `when:` statement to appear earlier in the task to read like the following: “when hostname equals R1, then reboot.” Note though that the ordering of the statements in a task is arbitrary. The core of Ansible is written in Python and each statement of a task becomes an element of a Python dictionary. Members of Python dictionaries are unordered. As such, the task could be written as follows and the behavior would be the same.

```
- name: Reboot select devices
  when: junos.facts.hostname == "R1"
  junos_shutdown:
    host={{ inventory_hostname }}
    shutdown="shutdown"
    reboot="yes"
```

Creating Ansible Playbooks (5 of 5)

▪ Loops

- The `with_items:` keyword operates like a `for-each` loop

`playbook_example_4.yml`

```
---
- name: Play Name
  hosts: juniper_hosts
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:

  - name: Retrieve information from devices running Junos OS
    junos_get_facts:
      host={{ inventory_hostname }}
    register: junos

  - name: Loop over a dictionary
    debug: msg={{ 'host-' + junos.facts.hostname }}
    when: junos.facts.hostname == item.host
    with_items:
      - { host: 'R1' }
      - { host: 'R2' }
```

We can use the `with_items:` keyword like a `for-each` loop from other languages. Rather than executing for all hosts, we can iterate over a specific host using `with_items:`.

Creating Ansible Playbooks, Part 5

We can also create loops using the `with_items:` keyword. We can pass lists or dictionaries to `with_items:` to execute a single task multiple times.

Agenda: Ansible

- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The **junos_shutdown** Module
- The **junos_get_facts** Module
- The **junos_install_config** Module
- Advanced Configuration Deployment
- Additional References

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 33

The **junos_shutdown** Module

The slide highlights the topic we discuss next.

The `junos_shutdown` Module

- The `junos_shutdown` module allows Ansible to shutdown or reboot remote devices

`reboot_hosts.yml`

```
---
- name: Reboot Hosts
  hosts: vsrx
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:
    - name: Reboot select devices
      junos_shutdown:
        host={{ inventory_hostname }}
        shutdown="shutdown"
        reboot="yes"

      # Annotations
      # The junos_shutdown module allows us to reboot or shutdown junos devices.
      # Required to shutdown/reboot devices.
      # By default, without the reboot argument, the Junos device shall be powered off.
```

Example of the `junos_shutdown` Module

A previous slide illustrated different modules provided by Juniper to operate Juno via Ansible. This slide illustrates the basic usage of the `junos_shutdown` module. We can use this module to automate tasks around shutting down or rebooting Junos devices. This is useful when used with the `junos_install_os` module to update several devices.

The `junos_shutdown` Example

- The CLI output below shows the execution of the `junos_shutdown` module

Terminal Output

```
bash $ more /etc/ansible/hosts
# Ansible Operating vSRX

[vsrx]
192.168.64.51
192.168.64.52 ← The inventory file has two hosts in the vsrx group.

bash $ ansible-playbook reboot_hosts.yml

PLAY [Reboot Hosts] ****
TASK: [Reboot select devices] ****
changed: [192.168.64.51]
changed: [192.168.64.52] On both devices, both tasks completed and caused a single change as a result of the reboot.
PLAY RECAP ****
192.168.64.51 : ok=1    changed=1    unreachable=0    failed=0
192.168.64.52 : ok=1    changed=1    unreachable=0    failed=0
bash $
```

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

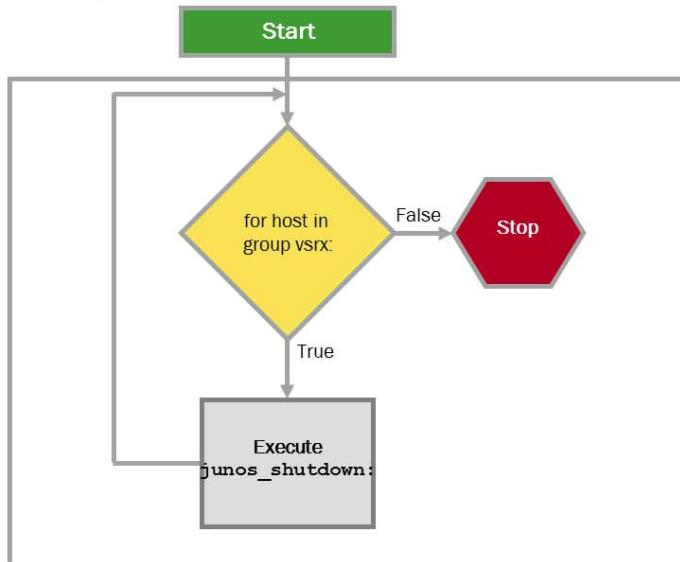
www.juniper.net | 35

The `junos_shutdown` Example

For the next set of examples, we shall use an inventory file that has two virtual SRX hosts in a group called `vsrx`. Note that when we run the playbook illustrated in the previous slide, Ansible details the results. Notice that we attempted a single task per device which completed successfully as noted by the `ok` column. Also, since we are rebooting a device, Ansible is reporting the change in the `changed` column.

The `junos_shutdown` Flow Chart

- The below illustrates the flow chart for the playbook running `junos_shutdown`



The `junos_shutdown` Flow Chart

Here we dive into the details of the flow of Ansible using pseudo-code and flow charts. Ansible iterates over each host identified by the `hosts:` argument using a for loop. A task executes `junos_shutdown:` for every host.

Agenda: Ansible

- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The `junos_shutdown` Module
- The `junos_get_facts` Module
- The `junos_install_config` Module
- Advanced Configuration Deployment
- Additional References

The `junos_get_facts` Module

The slide highlights the topic we discuss next.

The junos_get_facts Module

- The `junos_get_facts` module allows Ansible to retrieve information from Junos devices

`get_facts.yml`

```
---
- name: Get Facts
  hosts: vsrx
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:

    - name: Retrieve information from devices running Junos OS
      junos_get_facts:
        host={{ inventory_hostname }}
      register: junos
      # The junos_get_facts module allows us to retrieve facts from Junos devices.

    - name: Print Junos software version
      debug:
        msg="{{ junos.facts.version }}"
      # The register key word is used to save the result of a task to a variable.
      # The debug: module is meant for printing output during playbook runs.
```

The `junos_get_facts` Module

This slide illustrates the basic usage of the `junos_get_facts` module. We can use this module to gather data on remote Junos devices which is helpful for operating on specific hosts. Note, typically Ansible gathers these facts by default and stores them in a reserved variable. Since we are using the local execution feature, we must leverage a task to gather remote facts about Junos devices.

The junos_get_facts Example

- The CLI output below shows the execution of the `junos_get_facts` module

Terminal Output

```
bash $ ansible-playbook get_facts.yml

PLAY [Get Facts] ****

TASK: [Retrieve information from devices running Junos OS] ****
ok: [192.168.64.51]
ok: [192.168.64.52]

TASK: [Print Junos software version] ****
ok: [192.168.64.52] => {
    "msg": "12.1X46-D20.5" ← Illustrates the output from the debug: task in the playbook.
}
ok: [192.168.64.51] => {
    "msg": "12.1X46-D20.5"
}

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

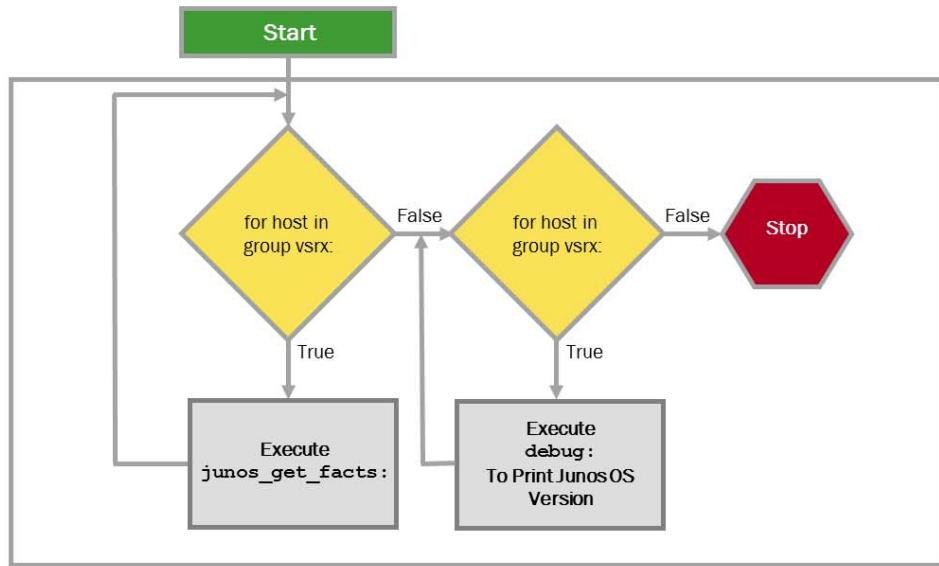
On both devices, both tasks completed, but no changes were required since we only gathered tasks.

The `junos_get_facts` Example

In our playbook, we are retrieving facts from our two Junos devices and using the `register:` keyword to store the data in a variable called `junos`. In a second task, we are printing the Junos software version of each device. Note that we do not have to worry about the full logic of iterating over hosts. As an example, we aren't concerned about variables in their relation to the iteration over the group of hosts. Ansible allows us to declare the variable once, and it handles the iteration over multiple hosts for us.

The `junos_get_facts` Flow Chart

- The below illustrates the flow chart for the playbook running `junos_get_facts`



The `junos_get_facts` Flow Chart

Here we again dive into the details of the flow of Ansible using pseudo-code and flow charts. Ansible iterates over each host identified by the `hosts:` argument using a for loop. The first task is to execute `junos_get_facts:`, so this task is executed for every host. The next task is to print the Junos software version.

Agenda: Ansible

- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The `junos_shutdown` Module
- The `junos_get_facts` Module
- The `junos_install_config` Module
- Advanced Configuration Deployment
- Additional References

The `junos_install_config` Module

The slide highlights the topic we discuss next.

The junos_install_config Module (1 of 2)

- The junos_install_config module allows Ansible to install Junos configuration templates

load_config.yml

```
---
- name: Install Junos Config
  hosts: vsrx
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no

  tasks:
    - name: Retrieve information from devices running Junos OS
      junos_get_facts:
        host={{ inventory_hostname }}
      register: junos

    - name: Install Junos configuration template
      junos_install_config:
        host={{ inventory_hostname }}
        file={{ item.config }}
        diffs_file={{ item.host }}
      when: junos.facts.hostname == item.host
      with_items:
        - { host: 'R1', config: 'bgp-config-1.conf' }
        - { host: 'R2', config: 'bgp-config-2.conf' }
```

The annotations explain the parameters of the junos_install_config module:

- host={{ inventory_hostname }}**: The junos_install_config module allows for installation of configuration templates.
- file={{ item.config }}**: Install the config file template for the proper host.
- diffs_file={{ item.host }}**: The diffs_file argument allows us to save the output of a "show | compare" operation to a file.

The junos_install_config Module, Part 1

This slide illustrates the basic usage of the `junos_install_config` module. We can use this module to install configuration templates onto Junos devices. Note that similar to PyEZ, the extension of the configuration file determines how the configuration is applied. Recall that a `.conf` file should contain Junos configuration syntax to merge with the existing configuration. A `.set` file should contain a series of set commands. Lastly, a `.xml` file should contain the structured XML configuration change. Also, the `junos_install_config` module leverages some default variables as follows.

- host** – This variable requires the IP address or hostname of the device that Ansible is trying to reach. As shown in previous examples, we are using the default `{{ inventory_hostname }}` variable to iterate over each host for the proper group in the inventory file.
- file** – This variable requires the filename of the configuration to deploy to hosts. In this scenario, we must control which configuration file to deploy to a specific host (e.g., we only want `bgp-config-1.conf` to be deployed to R1 and `bgp-config-2.conf` to be deployed to R2). We have created a mapping in YAML which uses `host` as a key to retrieve a hostname and `config` as a key to retrieve a config file (as seen under `with_items`). Using a `when:` statement we can control when a specific config file is deployed to a specific device as Ansible is iterating over each host. When the hostname of the current host (noted by `junos.facts.hostname`), matches our mapping created under `with_items`, we then deploy the relevant configuration file.
- diffs_file** – This variable is optional. Using this variable causes Ansible to save the configuration change to a flat file with a name of our choosing. In this scenario for simplicity, the filename is the same as the device hostname, which we retrieve using `{{ item.hostname }}`.

The `junos_install_config` Module (2 of 2)

- The `junos_install_config` module shall install the Junos configuration templates illustrated

Terminal Output

```
bash $ more bgp-config-1.conf
protocols {
    bgp {
        group ANSIBLE-DEPLOY {
            type internal;
            local-address 10.0.0.1;
            log-updown;
            family inet {
                unicast;
            }
            neighbor 10.0.0.2;
            neighbor 10.0.0.3;
            neighbor 10.0.0.4;
            neighbor 10.0.0.5;
        }
    }
}
```

Config template for R1.

Terminal Output

```
bash $ more bgp-config-2.conf
protocols {
    bgp {
        group ANSIBLE-DEPLOY {
            type internal;
            local-address 10.0.0.2;
            log-updown;
            family inet {
                unicast;
            }
            neighbor 10.0.0.1;
            neighbor 10.0.0.3;
            neighbor 10.0.0.4;
            neighbor 10.0.0.5;
        }
    }
}
```

Config template for R2.

The `junos_install_config` Module, Part 2

This slide illustrates the configurations applied to the Junos devices for reference. We are deploying an internal BGP (iBGP) configuration as an example.

The junos_install_config Example (1 of 2)

- The CLI output below shows the execution of the `junos_install_config` module

Terminal Output

```
bash $ ansible-playbook load_config.yml

PLAY [Install Junos Config] *****
TASK: [Retrieve information from devices running Junos OS] *****
ok: [192.168.64.51]
ok: [192.168.64.52]

TASK: [Install Junos configuration template] *****
skipping: [192.168.64.52] => (item={'host': 'R1', 'config': 'bgp-config-1.conf'})
changed: [192.168.64.51] => (item={'host': 'R1', 'config': 'bgp-config-1.conf'})
skipping: [192.168.64.51] => (item={'host': 'R2', 'config': 'bgp-config-2.conf'})
changed: [192.168.64.52] => (item={'host': 'R2', 'config': 'bgp-config-2.conf'})

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

The configuration loops over the two hashes identified by the `with_items:` keyword. The `when:` statement allows us to apply configurations to specific hosts.

On both devices, both tasks completed, but one change is completed per router.

The `junos_install_config` Example, Part 1

In our playbook, after retrieving facts from remote devices we are applying the configuration. We are iterating over each device using `with_items:` to which contains key,value pairs to map the host to the proper configuration file. Using the `when:` statement, we are performing a comparison operation to ensure R1 and R2 only get the proper configuration as noted by the `skipping:` message from the Ansible run.

The junos_install_config Example (2 of 2)

- The junos_install_config module can produce a log file illustrating the configuration change applied

Terminal Output

```
bash $ ls | grep R
R1
R2
bash $ more R1

[edit protocols]
+  bgp {
+    group ANSIBLE-DEPLOY ←
+      type internal;
+      local-address 10.0.0.2;
+      log-updown;
+      family inet {
+        unicast;
+      }
+      neighbor 10.0.0.1;
+      neighbor 10.0.0.3;
+      neighbor 10.0.0.4;
+      neighbor 10.0.0.5;
+    }
+ }
```

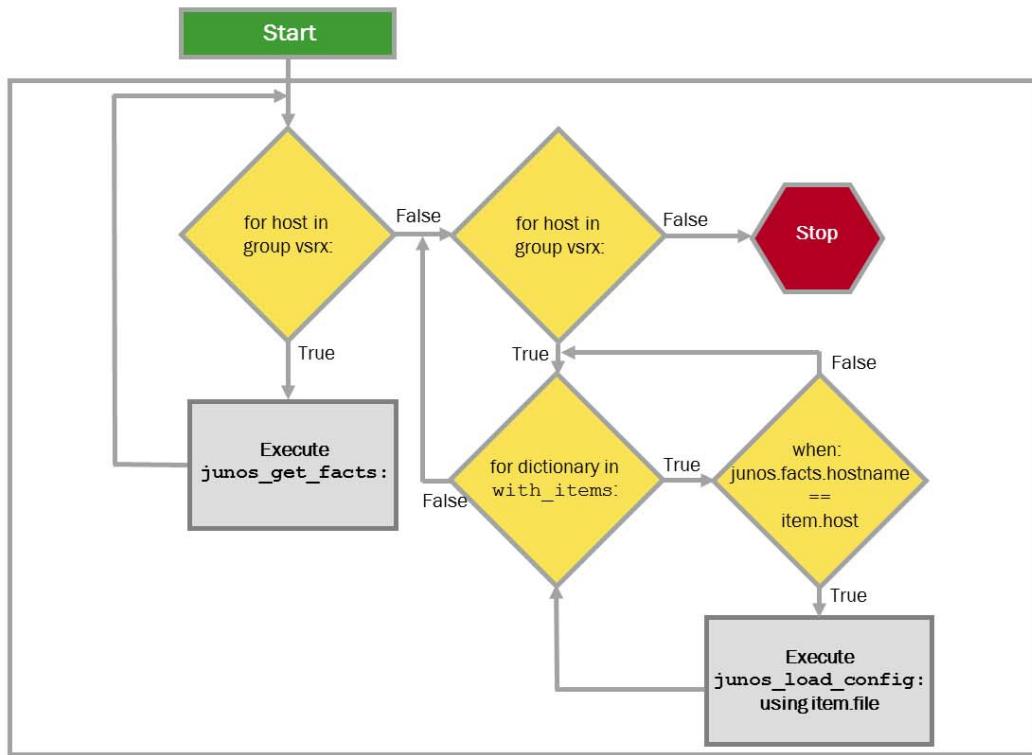
The `diffs_file:` argument allows for review of changes after running the playbook.

The junos_install_config Example, Part 2

Recall we ran our playbook with the `diffs_file:` argument for the task running `junos_install_config`. We can view the resulting change after the playbook completes for auditing/error checking.

The `junos_install_config` Flow Chart

- The below illustrates the flow chart for the playbook



The `junos_install_config` Flow Chart

Here we again dive into the details of the flow of Ansible using pseudo-code and flow charts. Ansible iterates over each host identified by the `hosts:` argument using a for loop. The first task is to execute `junos_get_facts:`, so this task is executed for every host. The next task is to iterate over two dictionaries containing key,value pairs. The first contains the hostname and config filename for R1 while the second has the same information for R2. Using the loops and conditionals in the Ansible playbook, we can apply the configuration to the proper remote host as Ansible iterates over the `vsrx` group.

Agenda: Ansible

- Introduction to Ansible
- Building a Basic Ansible Environment
- Creating Ansible Playbooks
- The `junos_shutdown` Module
- The `junos_get_facts` Module
- The `junos_install_config` Module
- Advanced Configuration Deployment
- Additional References

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 47

Advanced Configuration Deployment

The slide highlights the topic we discuss next.

Introducing Jinja2

▪ Overview of jinja2

- Jinja2 is a templating language commonly used with Python
- Using Jinja2 templates and YAML files containing variable definitions, we can automate creating common configurations
- Also, Ansible uses some Jinja2 syntax in playbooks for conditionals and variable replacement



Introducing Jinja2

Jinja2 is a templating language for Python. In our previous example, we pre-generated configuration files to deploy to each Junos device. Using Jinja2, we can automatically generate device specific configurations by calling Jinja2 templates within Ansible and iterating over variables to perform variable substitution.

Deploying Jinja2 Templates (1 of 4)

- The Jinja2 template below shall be used to generate a single Junos configuration to install via Ansible

`bgp_jinja_template.j2`

```
{% for bgp_vars in item.itervalues() %} ←
protocols {
    bgp {
        group {{ bgp_vars.bgp_group_name }} {
            type internal;
            local-address {{ bgp_vars.local_address }};
            log-updown; Using the {{}} notation, we can perform
            family inet { variable substitution. bgp_vars is
                unicast; the dictionary of key value pairs.
            }
        }
    }
}
{%
for host, ip in bgp_vars.neighbors.iteritems() %} ←
    neighbor {{ ip }};
{%
endfor %} ← We can also embed for loops. The
{%
endfor %} only syntax difference from Python, is
{%
endfor %} the endfor statement.
```

`deploy_bgp.yml`

```
--- → When the YAML file is passed to the
bgp_data: Jinja2 template, the first for loop allows
- R1: us to iterate over the list of routers.
    bgp_group_name: ANSIBLE-DEPLOY
    local_address: 10.0.0.1
    neighbors:
        r2: 10.0.0.2
        r3: 10.0.0.3
        r4: 10.0.0.4
        r5: 10.0.0.5
```

In the Jinja2 template,
bgp_vars is a
dictionary holding
the (key, value) pairs.

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS Worldwide Education Services

www.juniper.net | 49

Deploying Jinja2 Templates, Part 1

Jinja2 refers to flow control components as statements. Here we illustrate a for loop in a Jinja2 template. This template is meant to re-create the iBGP configuration shown in the previous example. Instead of statically creating the configuration file, we can create a single Jinja2 template. Then to generate the configuration dynamically, we have separated the configuration syntax from the variable data. Using a YAML file, we can define the variables that we wish to substitute in the Jinja2 template.

Engineers looking to perform this type of automation should start getting accustomed to separating data from configurations. In the Jinja2 template, we have the bare configuration syntax. In the YAML file, we have the data used to create the device specific configuration. As engineers get more comfortable with separating data from configuration, this type of exercise gets easier.

Deploying Jinja2 Templates (2 of 4)

load_jinja_configs.yml

```
---
- name: Load Configs From Jinja2 Templates
  hosts: vsrx
  roles:
    - Juniper.junos
  connection: local
  gather_facts: no
  vars:
    - junos_jinja_template: "bgp_jinja_template.j2"
    - path: "junos_configs"
  vars_files:
    - "junos_yaml_templates/deploy_bgp.yml"
  tasks:
    - name: Retrieve information from devices running Junos OS
      junos_get_facts:
        host={{ inventory_hostname }}
      register: junos

    - name: Create Junos configurations from Jinja2
      template: src={{ junos_jinja_template }} dest="{{ path }}/{{ junos.facts.hostname }}.conf"
      with_items: bgp_data
      After importing deploy_bgp.yml, we can iterate over the variables contents using the with_items: keyword to produce the target configuration from the Jinja2 template.

    - name: Install Junos configuration template
      junos_install_config:
        host={{ inventory_hostname }}
        file="{{ path }}/{{ junos.facts.hostname }}.conf"
        diff_file="{{ junos.facts.hostname }}.log"
      Juniper Ansible modules are executed locally. As such, we must explicitly run the junos_install_config: to install the generated template.
```

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 50

Deploying Jinja2 Templates, Part 2

This playbook combines the Jinja2 template with the variable declaration file called `deploy_bgp.yml` to produce a Junos configuration which is then installed by Ansible. We can use the keyword `vars_files:` to import our `deploy_bgp.yml` file. After doing so, we can reference variables contained within the YAML definition.

The playbooks first retrieves Junos device facts much like we have already seen in previous examples. The next task uses the `template:` module to generate the configuration files specific to each device. As noted earlier, we can iterate over the dictionary called `bgp_data` which we imported via the YAML variable file.

As illustrated in the previous slide, the `deploy_bgp.yml` defines a dictionary with one key: `R1`. The key refers to a hash of variables for the relevant BGP configuration. The `template:` module takes the Jinja2 template called `bgp_jinja_template.j2` as a `src` argument and iterates over the `bgp_data` hash to perform the necessary variable substitution. The `dst` argument is the target directory and file name for the resulting Junos configuration. Once this task completes, the final task is to deploy the configuration to the relevant remote Junos device.

Deploying Jinja2 Templates (3 of 4)

- The template: module in Ansible playbook from the previous slide produces the Junos configuration below

Terminal

```
bash$ more junos_configs/R1.conf
protocols {
    bgp {
        group ANSIBLE-DEPLOY {
            type internal;
            local-address 10.0.0.1;
            log-updown;
            family inet {
                unicast;
            }
            neighbor 10.0.0.4;
            neighbor 10.0.0.5;
            neighbor 10.0.0.2;
            neighbor 10.0.0.3;
        }
    }
}
```

deploy_bgp.yml

```
---
bgp_data:
- R1:
  bgp_group_name: ANSIBLE-DEPLOY
  local_address: 10.0.0.1
  neighbors:
    r2: 10.0.0.2
    r3: 10.0.0.3
    r4: 10.0.0.4
    r5: 10.0.0.5
```

© 2017 Juniper Networks, Inc. All rights reserved.

JUNIPER
NETWORKS

Worldwide Education Services

www.juniper.net | 51

Deploying Jinja2 Templates, Part 3

This slide illustrates the result of the template: module from the previous playbook. Once that task completes, the Jinja2 template and deploy_bgp.yml are combined to produce the Junos configuration above. Ansible then installs the Junos configuration via the junos_install_config: module.

Deploying Jinja2 Templates (4 of 4)

- After producing the Junos configuration, the `junos_install_config:` module installs the configuration file

Terminal Output

```
bash $ ansible-playbook load_jinja_configs.yml

PLAY [Load Configs From Jinja2 Templates] ****
TASK: [Retrieve information from devices running Junos OS] ****
ok: [192.168.64.51]

TASK: [Create Junos configurations from Jinja2] ****
changed: [192.168.64.51] => (item={'R1': {'neighbors': {'r4': '10.0.0.4', 'r5': '10.0.0.5', 'r2': '10.0.0.2', 'r3': '10.0.0.3'}, 'bgp_group_name': 'ANSIBLE-DEPLOY', 'local_address': '10.0.0.1'}})

TASK: [Install Junos configuration template] ****
changed: [192.168.64.51]

PLAY RECAP ****
192.168.64.51 : ok=3    changed=2    unreachable=0    failed=0
```

Three tasks completed and two changes executed (template generation and config change).

Deploying Jinja2 Templates, Part 4

Here we illustrate the successful operation of the configuration deployment. Note we have 2 changes for each host: 1 for the template creation and 1 for the configuration deployment.

This example was created in Ansible 1.2 and may need slight syntax modifications to work with the most recent versions of Ansible. Consult the <https://github.com/Juniper/ansible-junos-stdlib> website for the latest information on the `Junos_install_config` module and other Juniper modules.

Agenda: Ansible

- Introduction to Ansible
 - Building a Basic Ansible Environment
 - Creating Ansible Playbooks
 - The `junos_shutdown` Module
 - The `junos_get_facts` Module
 - The `junos_install_config` Module
 - Advanced Configuration Deployment
- Additional References

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 53

Additional References

The slide highlights the topic we discuss next.

Additional References

- You are encouraged to review other resources to become proficient at Ansible, YAML, or Jinja2
 - We have not covered a number of useful topics
 - Additional useful Ansible modules and built in features: <http://docs.ansible.com>
 - Detailed error handling
 - Producing Jinja2 templates: <http://jinja.pocoo.org>

The screenshot shows two side-by-side browser windows. The left window displays the Jinja2 documentation, featuring a header 'Working with Jinja2' and a code snippet example. The right window displays the Ansible documentation, featuring a header 'ANSIBLE' and a sidebar with navigation links like 'Introduction', 'Quickstart Video', 'Playbooks', 'About Modules', etc.

© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 54

Additional References

There is plenty more to learn with Ansible! Other topics not covered include additional useful Ansible modules, error handling when things go wrong, and macros (or functions) in Jinja2. The great thing about Ansible is that there is a large developer community from which to learn from. Remember though that Ansible is yet another tool. Although the goal may be to become a programmer, Ansible be yet another tool to help us as network engineers to simplify or optimize tasks.

Summary

- In this content, we:
 - Described the format of an Ansible YAML playbook
 - Described how Ansible interfaces with Junos OS devices
 - Explained how to operate Junos OS devices using Ansible

We Discussed:

- The Ansible playbook format;
- How Ansible interfaces with Junos; and
- How to operation Junos devices using Ansible.

Review Questions

1. What is a playbook?
2. What is the required playbooks format?
3. How can you control flow in a playbook?
4. What is an example scenario where Jinja2 templates may be used with Ansible?

Review Questions

- 1.
- 2.
- 3.
- 4.

Lab: Working with Ansible in the Junos OS

- Build and debug Ansible playbooks.
- Work with Jinja2 templates to deploy Junos configurations.

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 57

Lab: Working with Ansible in the Junos OS

This slide provides the objective for this lab.

Answers to Review Questions

1.

A playbook contains a list of tasks for Ansible to perform. When we want to automate repetitive tasks using Ansible, we create playbooks to iterate over hosts to complete common tasks.

2.

Ansible playbooks are written in YAML. However, tasks are executed using Python modules, so some syntax looks similar to Python and Python's templating language: Jinja2.

3.

There are number of ways to perform flow control in Ansible. We discussed the usage of the conditional `when:` which operates like an if statement and the loop `with_items:` which operates like a for-each loop.

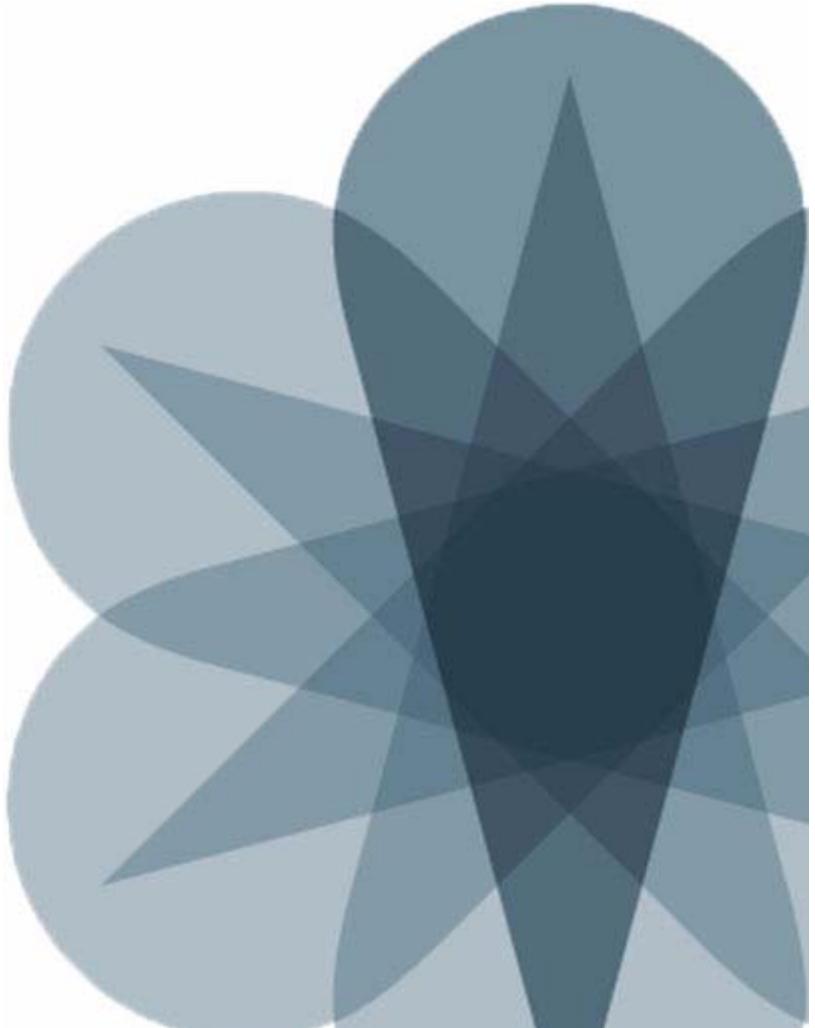
4.

Jinja2 templates can be used to automatically generate Junos configurations. We can perform variable substitution to generate device specific configuration files.



Junos Platform Automation and DevOps

Chapter 8: JSNAPy



Objectives

- After successfully completing this content, you will be able to:
 - Describe how JSNAPy can help automate Junos
 - Install JSNAPy
 - Use JSNAPy to create snapshots

We Will Discuss:

- How JSNAPy can help automate devices running the Junos OS;
- Installing JSNAPy; and
- Creating snapshots using JSNAPy.

Agenda: JSNAPy

- JSNAPy Overview and Installation
- Create JSNAPy Configuration Files
- Create JSNAPy Test Files
- Use the JSNAPy Command Line Tool

JSNAPy Overview and Installation

The slide lists the topics we will discuss. We discuss the highlighted topic first.

What Does JSNAPy Do?

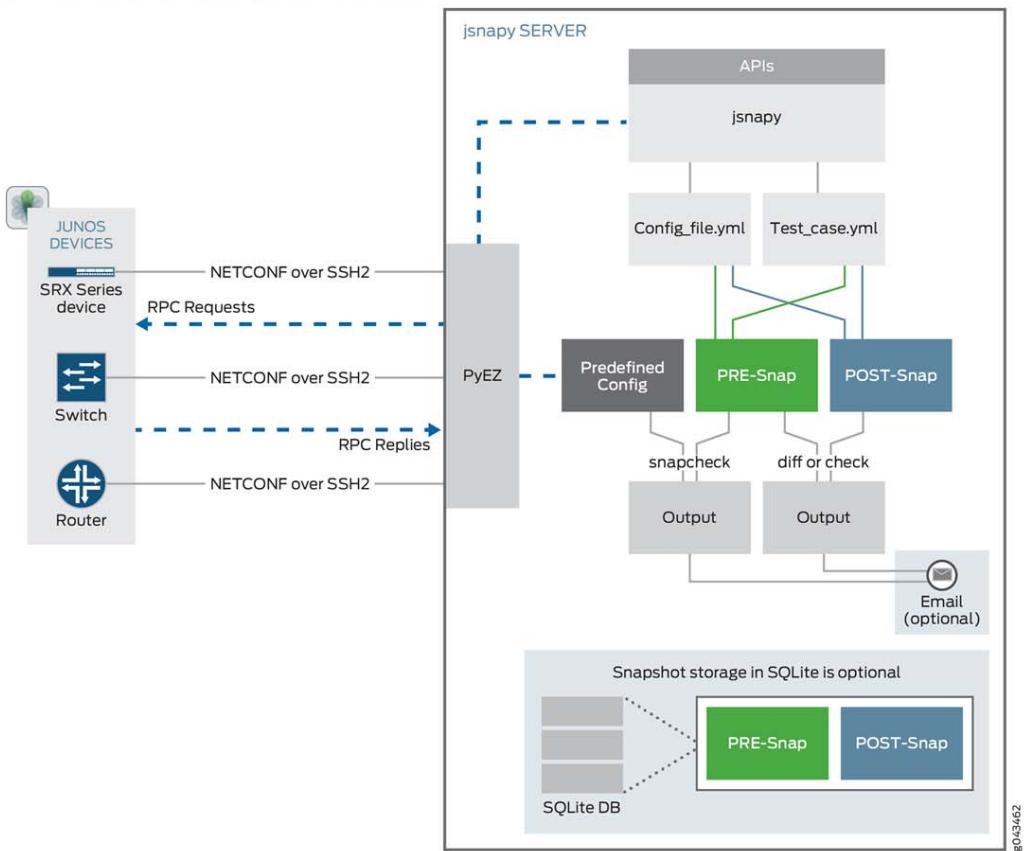
- Automates Network Verifications

- Create Snapshots
 - Take snapshots of Junos OS configuration files
 - Take snapshots of Junos RPC call results
- Analyze Snapshots
 - Compare a snapshot to a standard
 - Compare a pre change snapshot with a post change snapshot
 - Get the difference between two snapshots
- Store snapshots in text files or a SQLite database
- Email snapshot results to administrators
- Embed JSNAPy into Python scripts

What Does JSNAPy Do?

JSNAPy is the Junos Snapshot Administrator in Python. JSNAPy is a Python version of the Junos Snapshot Administrator (SNAP) application which was written in SLAX. The slide lists the important characteristics of JSNAPy that you will learn about in the chapter.

JSNAPy Architecture



© 2017 Juniper Networks, Inc. All rights reserved.



Worldwide Education Services

www.juniper.net | 5

JSNAPy Architecture

Junos Snapshot Administrator in Python is installed on a remote server running an OS that can support Python, including Mac OS X, and many Linux distributions. The jsnapy server uses Juniper PyEZ to make NETCONF connections over SSHv2 to your networked Junos OS devices. Using YAML-formatted configuration and test files for connection and test criteria, jsnapy sends RPC requests to the devices over the NETCONF connections. RPC replies are received back at the server in the form of snapshots. You can run jsnapy from the server command line or can be included as a module in other Python applications.

The snapshots are formatted as text or XML files and are stored on the server in a location designated by the `jsnapy.cfg` file located in the directory `/etc/jsnapy/`.

Optionally, the snapshots can be stored in an SQLite database on the server. Jsnapy can then compare the snapshots either to other snapshots or to pre-defined criteria in order to audit the effects of configuration changes or to confirm proper device configuration.

JSNAPy Prerequisites

■ JSNAPy Server

- Linux or Mac OS (not Windows)
 - OS Dependencies seen below
 - Same dependencies as PyEZ
- Python 2.6 or later
- Optional SQLite

■ Device Running Junos OS

- Management Port Reachability
- NETCONF Enabled

```
[edit system services]
lab@vMX-1# set netconf ssh
```

JSNAPy Prerequisites

You install Junos Snapshot Administrator in Python (jsnapy) on a remote server in the network. Prior to installing jsnapy, ensure that the remote server is running an OS that is capable of running Python 2.6 or later. This includes, but is not limited to:

- Linux (Debian, Ubuntu, Fedora, CentOS, and FreeBSD)
- Mac OS X

Note: Although Microsoft Windows can run Python 2.6, jsnapy is not supported on Windows.

Due to the wide range of possible supported OSs, there are dependencies within each OS that must also be fulfilled. These are the same dependencies as PyEZ

CentOS - pip, python-devel, libxml2-devel, libxslt-devel, gcc, openssl, and libffi-devel

Debian - python-pip, python-dev, libxml2-dev, libxslt-dev, libssl-dev, and libffi-dev

Fedora - python-pip, python-devel, libxml2-devel, libxslt-devel, gcc ,openssl, and libffi-devel

FreeBSD - py27-pip, libxml2, libxslt, OSX, xcode, xquartz, and pip

Ubuntu - python-pip, python-dev, libxml2-dev, libxslt-dev, libssl-dev, and libffi-dev

Installing JSNAPy

- Install using pip

```
user@jsnappy-server:~> sudo pip install jsnappy  
user@jsnappy-server:~> sudo pip install  
git+https://github.com/Juniper/jsnappy.git
```

- Update to the latest version with the -U option

```
user@jsnappy-server:~> sudo pip install jsnappy -U  
user@jsnappy-server:~> sudo pip install  
git+https://github.com/Juniper/jsnappy.git -U
```

- Download or clone the source code from this git repository

```
user@jsnappy-server:~> git clone  
https://github.com/Juniper/jsnappy
```

Installing JSNAPy

You can use the pip application do download from the Python Package Index or from Juniper Network's JSNAPy git repository using the instructions above. The Git repository will have the most recent updates. The slide shows how you can use the -U option to update JSNAPy.

You can also download the JSNAPy source code and compile it yourself. You can find instruction on how to compile from source code at <https://github.com/Juniper/jsnappy>.

Agenda: JSNAPy

- JSNAPy Overview and Installation
- Create JSNAPy Configuration Files
- Create JSNAPy Test Files
- Use the JSNAPy Command Line Tool

Create JSNAPy Configuration Files

The slide highlights the topic we discuss next.

JSNAPy Configuration Files

- JSNAPy configuration files are written in YAML and include the follow four parts

```
---
hosts:
  - include: devices.yml
    group: EX
tests:
  - test_is_equal.yml
  - test_is_in.yml
sqlite:
  - store_in_sqlite: yes
    check_from_sqlite: yes
    database_name: jbb.db
mail: send_mail.yml
```

The diagram illustrates the structure of a JSNAPy configuration file. It highlights four main sections: **hosts**, **tests**, **sqlite**, and **mail**. Each section is accompanied by a callout box containing its description and whether it is mandatory or optional.

- hosts:** The **hosts:** section lists hosts and may also include an external file. It is marked as **(Mandatory)**.
- tests:** The **tests:** section contains a list of Test files that contain tests. It is marked as **(Mandatory)**.
- sqlite:** The **sqlite:** section contains Information needed to connect to sqlite Database. It is marked as **(Optional)**.
- mail:** The **mail:** section lists the file with Information needed to send results via Email. It is marked as **(Optional)**.

JSNAPy Configuration Files

JSNAPy configuration files are written in YAML and contain four sections. The **hosts:** and **tests:** sections are mandatory while the **mail:** and **sqlite:** sections are optional. We will look at each of these sections in more detail on the following slides.

Because these are YAML files, the spacing and indent is important. For more information on YAML formatting see the chapter on JSON and YAML earlier in the course.

Configuration File hosts: section

- The hosts: section can specify one or more hosts

Hosts:

- **device**: 172.25.11.1
 - username**: lab
 - passwd**: lab123
- **device**: 172.25.11.2
 - username**: lab
 - passwd**: lab123
- **include**: devices.yml
 - group**: MX

You can include
one or more devices

You can also import a YAML
file with a list of devices
Organized in groups

Configuration File hosts Section

The hosts: section of the configuration file can specify an IP address and credentials for a single host or multiple hosts as seen in the slide.

You can also arrange your hosts into groups and store them in a separate file that is included into the configuration file. The slide shows the devices.yml file. The configuration file will import the devices.yml file and apply tests to those hosts in the MX group. You can have multiple groups within a single file and a single host can exist within more than one group.

Sample devices.yml File

- The hosts : section can specify one or more hosts

MX:

```
- 10.20.1.20:  
  username: root  
  passwd: root123  
- 10.21.13.14:  
  username: root  
  passwd: root123
```



You can include
one or more devices
per group

EX:

```
- 10.2.15.210:  
  username: root  
  passwd: root123  
- 10.9.16.22:  
  username: abc  
  passwd: pqr
```

Sample devices.yml File

The slide shows a sample YAML file containing devices arranged in groups.

Configuration File tests : Section

- The tests : section gives a list of one or more YAML files that contain tests

tests :

- test_not_less.yml
- test_not_more.yml
- one_more_test.yml

You can have one or many tests

Configuration File tests Section

The tests section contains a list of YAML files that contain test parameters. The tests: section can contain one or multiple tests as shown in the slide. We will build test files shortly.

Configuration File `mail:` and `sqlite:` Sections

- The `mail:` section lists the YAML file that contains the email settings

```
mail: send_mail.yml
```

- The `sqlite:` section lists the YAML file that contains the email settings

```
sqlite:  
  - store_in_sqlite: yes  
    check_from_sqlite: yes  
    database_name: jsnappy.db
```

Configuration File `mail` and `sqlite` Sections

The `mail:` section specifies the name of the YAML files that contains the SNMP and mail settings.

The `sqlite:` section contains three key value pairs with information necessary to connect to the SQLite database

Sample send_mail.yml File

- Sample send_mail.yml file

```
to: foo@gmail.com
from: bar@gmail.com
sub: "Sample Jsnappy Results, please verify"
recipient_name: Foo
passwd: 123
server: smtp.gmail.com
port: 587
sender_name: "Juniper Networks"
```

- The mail server and port sections are used only if you want to override the defaults

Sample send_mail.yml File

The slide shows a sample mail configuration file. By default the gail server is configured in jsnappy. If u want to send from some other email account apart from gmail, then please specify that server.

The port is also optional. by default it is SMTP port 587. You can override this if necessary.

Agenda: JSNAPy

- JSNAPy Overview and Installation
- Create JSNAPy Configuration Files
- Create JSNAPy Test Files
- Use the JSNAPy Command Line Tool

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 15

Create JSNAPy Test Files

The slide highlights the topic we discuss next.

The test_sw_version Test

```
tests_include:
  - test_sw_version

test_sw_version:
  - command: show version
  - item:
    xpath: '//software-information'
    tests:
      - all-same: junos-version
        err: "Test Failed!!! The versions are not the same.
From the PRE snapshot, the version is: <{{pre['junos-
version']}}>. From the POST snapshot, the version is
<{{post['junos-version']}}>!! "
        info: "Test Succeeded!! The Junos OS version is:
<{{post['junos-version']}}>!!!!"
```

The test_sw_version Test

A good way to get a feel for how JSNAPy tests work is to take a look at a few of them. The slide shows a simple test that checks to see if the junos-version has remained the same.

At the top of the test is a tests_include section. You can run multiple tests within a single test file. If you decide not to run a particular test you can remove it from the tests_include: section. If you omit the tests_include: section, all of the tests within the file will be ran.

The – command: show version key value pair specifies that JSNAPy will execute the show version command and examine its results.

The – item tells JSNAPy to look for the first instance of the item in the xpath: '//software-information' key value pair. The response to the show version command is in XML by default and the top element in the reply is the <software-information> element.

The tests: section lists the tests that will be performed on the results of the show version command. The first and only test in this case is to test that the data contained within the <junos-version> XML element is the same. If it changes you will see the message listed in err:. If the test succeeds you will see the message listed in info:. Notice how messages use Jinja2 templates to display data from the pre and post snapshots.

The test_rpc_filtering Test

```
tests_include:
- test_rpc_filtering

test_rpc_filtering:
- rpc: get-config
- kwargs:
  filter_xml: configuration/system/host-name
```

The test_rpc_filtering Test

Here is another sample test file that doesn't perform any tests. The test_rpc_filtering test issues the get-config RPC. Notice that it is an RPC and not a command as we saw previously. The keyword arguments or kwargs: section tell JSNAPy that there are arguments which follow. In this case the argument is an XML filter. The filter_xml: configuration/system/host-name key value pair tells JSNAPy to retrieve only the section beneath the configuration/system/host-name hierarchy.

This type of test could be used by a JSNAPy diff comparison where the results of one snapshot is compared against the results of a second snapshot.

The `test_interfaces_terse` **test**

```
test_interfaces_terse:
    - rpc: get-interface-information
      kwargs:
        - terse: True
    - iterate:
      xpath: //physical-interface
      tests:
        - is-equal: admin-status, up
          info: "Test Succeeded !! admin-status is equal to
{{pre['admin-status']}} with oper-status {{pre['oper-
status']}}"
          err: "Test Failed !! admin-status is not equal to
up, it is {{pre['admin-status']}} with oper-status
{{pre['oper-status']}}"
```

- There are many more test operators!

The `test_interfaces_terse` Test

The `test_interfaces_terse` test is similar to our last test. It issues the RPC `get-interface-information` with an argument of `terse`. This time rather than using an `-item:` key, it uses `- iterate:` which causes JSNAPy to not just find the first instance of `/physical-interface`, but all instances.

The test looks to see if the result has an `admin-status` that is-equal to `up`. If it is `up` you will get the `info` message. If it is not `up` you will see the `err` message. This message will show once for each interface found.

There are many more operators that are not covered in this course. You can see a list of operators below. This list comes from: https://www.juniper.net/documentation/en_US/junos-snapshot1.0/topics/reference/general/automation-junos-snapshot-python-operators-summary.html

JSNAPy Test Operators (1 of 2)

Compare Elements or Element Values in Two Snapshots	
delta	Compare the change in value of a specified data element, which must be present in both snapshots, to a specified delta. You can specify the delta as an absolute, positive, or negative percentage, or an absolute, positive, or negative fixed value.
list-not-less	Determine if the specified items are present in the first snapshot but are not present in the second snapshot.
list-not-more	Determine if the specified items are present in the second snapshot but are not present in the first snapshot.
no-diff	Compare specified data elements that are present in both snapshots, and verify that the value is the same.
Operate on Elements with Numeric or String Values	
all-same	Check if all content values for the specified elements are the same. Optionally, you can check if all content values for the specified elements are the same as the content value of a reference item.
is-equal	- Test if an XML element string or integer value matches a given value.
not-equal	- Test if an XML element string or integer value does not match a given value.

Compare Elements or Element Values in Two Snapshots

delta - Compare the change in value of a specified data element, which must be present in both snapshots, to a specified delta. You can specify the delta as an absolute, positive, or negative percentage, or an absolute, positive, or negative fixed value.

list-not-less - Determine if the specified items are present in the first snapshot but are not present in the second snapshot.

list-not-more - Determine if the specified items are present in the second snapshot but are not present in the first snapshot.

no-diff - Compare specified data elements that are present in both snapshots, and verify that the value is the same.

Operate on Elements with Numeric or String Values

all-same - Check if all content values for the specified elements are the same. Optionally, you can check if all content values for the specified elements are the same as the content value of a reference item.

is-equal - Test if an XML element string or integer value matches a given value.

not-equal - Test if an XML element string or integer value does not match a given value.

JSNAPy Test Operators (2 of 2)

Operate on Elements with Numeric Values	
in-range	Test if the XML element value is within a given numeric range.
not-range	Test if the XML element value is outside of a given numeric range
is-gt	Test if the XML element value is greater than a given numeric value.
is-lt	Test if the XML element value is less than a given numeric value

Operate on Elements with String Values	
contains	Determine if an XML element string-value contains the provided string value.
is-in	Determine if an XML element string-value is included in from a specified list of string values.
not-in	Determine if an XML element string-value is excluded from a specified list of string values.

Operate on XML Elements	
exists	Verify the existence of an XML element in the snapshot.
not-exists	Verify the lack of existence of an XML element in the snapshot.

Operate on Elements with Numeric Values

in-range - Test if the XML element value is within a given numeric range.

is-gt - Test if the XML element value is greater than a given numeric value.

is-lt - Test if the XML element value is less than a given numeric value.

not-range - Test if the XML element value is outside a given numeric range.

Operate on Elements with String Values

contains - Determine if an XML element string-value contains the provided string value.

is-in - Determine if an XML element string-value is included in a specified list of string values.

not-in - Determine if an XML element string value is excluded from a specified list of string values.

Operate on XML Elements

exists - Verify the existence of an XML element in the snapshot.

not-exists - Verify the lack of existence of an XML element in the snapshot.

Agenda: JSNAPy

- JSNAPy Overview and Installation
 - Create JSNAPy Configuration Files
 - Create JSNAPy Test Files
- Use the JSNAPy Command Line Tool

Use the JSNAPy Command Line Tool

The slide highlights the topic we discuss next.

Snapshot File Names

Device Name	Snapshot Name	Test Name
vMX-1	Week1-Snap	OSPF-test BGP-test

- The snapshot file name is a combination of the device name, the snapshot name and the test name
- Each test will be in its own file even multiple tests were performed within the same test file
 - vmX-1_Week1-Snap_OSPF-test
 - vmX-1_Week1-Snap_BGP-test
- By default snapshots are stored in /etc/jsnappy/snapshots.

Snapshot File Names

When snapshots are created the name is created from a combination of the device name, the snapshot name, and the test name. Each test will be put into its own file even if multiple tests were performed within the same test file. The slide show an example of a sample snapshot name and how it is generated.

Taking a Snapshot

- Creating a Snapshot syntax

```
user@jsnappy-server$ jsnappy --snap snapshot-name -f configuration-filename
```

- Creating a pre and post snapshot example

```
lab@jaut-desktop$ jsnappy --snap snap1 -f mySnapshotConfig.yml
```

```
lab@jaut-desktop$ jsnappy --snap snap2 -f mySnapshotConfig.yml
```

- Use the same configuration file for both snapshots

Taking a Snapshot

The slide shows the syntax for creating a snapshot. Typically you will want to create two snapshots; one you take before an event and the other after the event.

Be sure to use the same configuration name for both snapshots.

Comparing Two Snapshots

- Compare two snapshots using check

```
user@jsnappy-server$ jsnappy --check snapshot1  
snapshot2 -f configuration-filename
```

- Example from previous slide

```
lab@jaut-desktop$ jsnappy --check snap1 snap2  
-f mySnapshotConfig.yml
```

```
[lab@jaut-desktop jsnappy]$ jsnappy --check snap1 snap2 -f mySnapshotConfig.yml  
***** Device: 172.25.11.1 *****  
Tests Included: test_sw_version  
***** Command: show version *****  
PASS | Value of all "junos-version" at xpath "//software-information" is same  
[ 1 matched ]  
----- Final Result!! -----  
test_sw_version : Passed  
Total No of tests passed: 1  
Total No of tests failed: 0  
Overall Tests passed!!!  
[lab@jaut-desktop jsnappy]$
```

Comparing Two Snapshots

Once you have created the two snapshots you can then check to see if they match by using the -check argument. The slide shows the syntax and an example taken from the two snaps we took on the previous page.

Performing a Snapcheck

- Perform a snapcheck

```
user@jsnappy-server$ jsnappy --snapcheck snapshot2 -f configuration-filename
```

- Example from previous slide

```
lab@jaut-desktop$ jsnappy --snapcheck snap2 -f mySnapshotConfig.yml
```

Performing a Snapcheck

If you have a standard configuration you can check your snapshot against that configuration. You build your checks into the YAML configuration file.

At anytime you can check to see how your configuration meets the standard. You can include a count of interfaces in an admin up state or the number of OSPF neighbor relationships etc.

Compare Two Snapshots Using --diff

- Perform a snapcheck

```
user@jsnappy-server$ jsnappy --diff snapshot1 snapshot2  
-f configuration-filename
```

- Example from previous slide

```
lab@jaut-desktop$ jsnappy -diff snap1 snap2  
-f mySnapshotConfig.yml
```

Comparing Two Snapshots Using --dif

The –diff option compares two snapshots in XML or text word by word. This option is only supported from the command line.

Use JSNAPy as a Python Module (1 of 2)

```
### Example showing how to pass yaml data in same file ####
from jnpr.jsnapy import SnapAdmin
from pprint import pprint
from jnpr.junos import Device

js = SnapAdmin()

config_data = """
hosts:
    - device: 198.51.100.10
        username : <username>
        passwd: <password>
tests:
    - test_exists.yml
    - test_contains.yml
    - test_is_equal.yml
"""
"""


```

© 2017 Juniper Networks, Inc. All rights reserved.

 Worldwide Education Services

www.juniper.net | 27

Use JSNAPy as a Python Module: Part 1

In addition to using JSNAPy as a stand alone command line application it can also be used as a module within Python. This and the next slide show an example of how you can implement JSNAPy into your python application.

The only additional new module in the script is the SnapAdmin function from the jnp.jsnapy module.

`js = SnapAdmin()` assigns the `SnapAdmin()` function to `js` for use throughout the script.

Rather than read in a JSNAPy configuration file the configuration data is placed within the script and assigned to the `config_data` variable.

Continued on the next page.

Use JSNAPy as a Python module (2 of 2)

```
snapchk = js.snapcheck(config_data, "pre")
for val in snapchk:
    print "Tested on", val.device
    print "Final result: ", val.result
    print "Total passed: ", val.no_passed
    print "Total failed:", val.no_failed
    #pprint(dict(val.test_details))
```

Use JSNAPy as a Python Module: Part 2

Next `js.snapcheck(config_data, "pre")` performs a snapcheck using the `config_data` and naming the snapcheck “pre”. The result from the sanpcheck is stored in the `snapchk` variable.

The for loop then prints out the results. The last line has been commented out so that all the details of the snapcheck are not printed out.

This is a simple example. There are more examples in the `/etc/jsnapy/samples/` directory on the server where JSNAPy is installed.

Additional Places For Information

- Junos Snapshot Administrator in Python Guide

https://www.juniper.net/documentation/en_US/junos-snapshot1.0/information-products/pathway-pages/junos-snapshot-python.pdf

- The JSNAPy GitHub site

<https://github.com/Juniper/jsnapy>

- Samples and Examples that install with JSNAPy

/etc/jsnapy/samples/

Additional Places For Information

The websites shown in the slide above provide additional information about JSNAPy.

Summary

- In this content, we:

- Described how JSNAPy can help automate Junos
- Installed JSNAPy
- Used JSNAPy to create snapshots

We Discussed:

- How JSNAPy can help automate devices running the Junos OS;
- Installing JSNAPy; and
- Creating snapshots using JSNAPy.

Review Questions

1. What application is the predecessor of JSNAPy?
2. The result of snapshots are returned in what format?
3. What Python module is needed to run JSNAPy within a Python script

Review Questions

- 1.
- 2.
- 3.

Lab: Using JSNAPy

- Create a JSNAPy configuration file.
- Create a JSNAPy test file.
- Use JSNAPy to create and compare snapshots.

Lab: Using JSNAPy

The slide lists the objectives for the lab.

Answers to Review Questions

1.

JSNAP is the predecessor to JSNAPPY and was written in SLAX.

2.

Snapshot results are returned in XML

3.

You need the SnapAdmin module to run JSNAPy within a Python script.

Acronym List

API	application programming interface
AS	autonomous system
BGP	Border Gateway Protocol
CLI	command-line interface
CNAME	canonical name
DNS	Domain Name System
DSL	Domain Specific Language
FQDN	fully qualified domain name
GUI	graphical user interface
iBGP	internal BGP
IGP	interior gateway protocol
IPv6	IP version 6
ISO	International Organization for Standardization
ISP	Internet service provider
lo0	loopback interface
NOC	network operations center
op	operation script
OSI	Open Systems Interconnection
pvm	Python virtual machine
pypi	Python Package Index
RPC	remote procedure call
scp	secure copy
SLAX	Stylesheet Language Alternative Syntax
UTC	Coordinated Universal Time
VRRP	Virtual Router Redundancy Protocol
XML	Extensible Markup Language
XPath	Extensible Markup Path Language
XSD	XML Schema Definition
XSLT	Extensible Stylesheet Language Transformations

Corporate and Sales Headquarters

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, CA 94089 USA
Phone: 888.JUNIPER (888.586.4737)
or 408.745.2000
Fax: 408.745.2100
www.juniper.net

APAC and EMEA Headquarters

Juniper Networks International B.V.
Boeing Avenue 240
1110 PZ SCHIPHOL-RIJK
Amsterdam, The Netherlands
Phone: 31.0.207.125.700
Fax: 31.0.207.125.701

Copyright 2017

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

❖ Printed on recycled paper